Марк Дж. Прайс

C# 7 и .NET Core

КРОСС-ПЛАТФОРМЕННАЯ РАЗРАБОТКА ДЛЯ ПРОФЕССИОНАЛОВ

3-е издание

Packt

ПИТЕР

C# 7.1 and .NET Core 2.0 – Modern Cross-Platform Development

Third Edition

Create powerful applications with .NET Standard 2.0, ASP.NET Core 2.0, and Entity Framework Core 2.0, using Visual Studio 2017 or Visual Studio Code

Mark J. Price

BIRMINGHAM - MUMBAI

Марк Дж. Прайс

C# 7 и .NET Core

КРОСС-ПЛАТФОРМЕННАЯ РАЗРАБОТКА ДЛЯ ПРОФЕССИОНАЛОВ

3-е издание

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2018

ББК 32.973.2-018.1
УДК 004.43
П68

Прайс Марк Дж.

П68 C# 7 и .NET Core. Кросс-платформенная разработка для профессионалов. 3-е изд. — СПб.: Питер, 2018. — 640 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0516-8

C# 7 — новая мощная и многофункциональная версия популярнейшего языка программирования от Майкрософт. Вы встретите немало интересных книг по языку C# и платформе .NET, однако в большинстве из них лишь мельком рассматривается столь важный аспект, как кросс-платформенная разработка. Научитесь писать приложения, которые работают всегда и везде, на ПК и мобильных устройствах. Познакомьтесь с инструментом Xamarin.Forms, освойте тонкости работы с Visual Studio 2017, добейтесь многогранности и универсальности ваших программ на C#.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1788398077 англ.

ISBN 978-5-4461-0516-8

Краткое содержание

Часть III. Модели приложений

Оглавление

Часть I. C# 7.1

Часть II. .NET Core 2.0 и .NET Standard 2.0

Часть III. Модели приложений

Об авторе

Марк Дж. Прайс (Mark J. Price) — обладатель сертификатов Microsoft Certified Solutions Developer (MCSD), Microsoft Specialist: Programming in C# и Episerver Certified Developer с более чем 20-летним опытом в области обучения и программирования.

С 1993 года Марк сдал свыше 80 экзаменов корпорации Microsoft по программированию и специализируется на подготовке других людей к успешному прохождению тестирования. Его студенты — как 16-летние новички, так и профессионалы с многолетним опытом. Марк ведет эффективные тренинги, сочетая образовательную деятельность с реальной практикой в консалтинге и проектировании систем для корпораций по всему миру.

В период с 2001 по 2003 год Марк посвящал все свое время разработке официального обучающего программного обеспечения в штаб-квартире Microsoft в американском городе Редмонд. В составе команды он написал первый обучающий курс по C#, когда была выпущена еще только альфа-версия языка. Во время сотрудничества с Microsoft он работал инструктором по повышению квалификации сертифицированных корпорацией специалистов на специальных тренингах, посвященных C# и .NET.

Microsoft CERTIFIED Solutions Developer App Builder

Microsoft Specialist Programming in C#

Episerver CMS Certified Developer

В настоящее время Марк разрабатывает и поддерживает обучающие курсы для системы Digital Experience Cloud компании Episerver, лучшей .NET CMS в сфере цифрового маркетинга и электронной коммерции.

В 2010 году Марк получил свидетельство об окончании последипломной программы обучения, дающее право на преподавание. Он преподает в Лондоне в двух средних школах математику старшеклассникам, готовящимся к получению сертификатов GCSE и A-Level. Кроме того, Марк получил сертификат Computer Science BSc. Hons. Degree в Бристольском университете (University of Bristol), Англия.

Спасибо моим родителям, Памеле и Яну, за мое воспитание, прививание трудолюбия и любопытства к миру. Спасибо моим сестрам, Эмили и Джульетте, за то, что полюбили меня, неуклюжего старшего братца. Спасибо моим друзьям и коллегам, вдохновляющим меня технически и творчески. Наконец, благодарю всех, кого я учил на протяжении многих лет, за то, что побудили меня быть лучшим преподавателем, чем я есть.

О рецензентах

Дастин Хеффрон (Dustin Heffron) — разработчик программного обеспечения и игр. Имеет более чем десятилетний опыт программирования на разных языках, восемь из которых связаны с C# и .NET.

В настоящее время разрабатывает программы для автоматизации и тестирования медицинских инструментов в компании Becton Dickinson. Кроме того, он соучредитель и генеральный директор компании SunFlake Studios.

Дастин уже долгое время сотрудничает с издательством Packt и принимал участие в работе над такими книгами, как *XNA 4.0 Game Development by Example: Beginner's Guide*, *C# 6 and .NET Core 1.0: Modern Cross-Platform Development*, а также над серией видеоуроков *XNA 3D Programming by Example*. Кроме того, Дастин наряду с Ларри Луизианой (Larry Louisiana) выступает соавтором серии видеоуроков XNA 3D Toolkit.

Я хотел бы поблагодарить свою жену за то, что она ежедневно помогает мне превзойти самого себя.

Эфраим Кириакидис (Efraim Kyriakidis) — инженер-программист с более чем десятилетним опытом разработки и реализации программных решений для различных клиентов и проектов. Он хорошо разбирается во всех этапах цикла создания ПО. Его первое знакомство с компьютерами и программированием произошло в детстве, во времена популярности компьютера Commodore 64, в 80-х годах XX века. С тех пор он вырос и получил диплом в университете Аристотеля в Салониках (Aristotle University Thessaloniki), Греция. На протяжении своей карьеры работал в основном с технологиями Microsoft, используя C# и .NET, начиная с версии 1.0. В настоящее время трудится в корпорации Siemens AG в Германии разработчиком ПО.

Предисловие

В книжном магазине вы увидите книги по языку C# объемом в тысячи страниц с исчерпывающим материалом по платформе .NET и программированию на C#.

Эта книга другая. Она наполнена практическими пошаговыми инструкциями. Я стремился написать эту книгу как лучшее пошаговое руководство по современным практическим приемам кросс-платформенного программирования на языке C# с использованием платформы .NET Core.

Я расскажу о крутых фишках и секретах языка C#, чтобы вы могли впечатлить коллег и потенциальных работодателей и быстро начать зарабатывать деньги. Вместо того чтобы тоскливо обсуждать каждую деталь, я буду придерживаться принципа «не знаете термин — Google в помощь».

В конце каждой главы вы найдете раздел «Практические задания». В нем приводятся тематические вопросы, на которые вам нужно будет ответить, а также конкретные упражнения, которые желательно выполнить. В подразделе «Дополнительные ресурсы» перечислены различные ресурсы, которые позволят вам подробнее рассмотреть затронутые в главе темы.

Файлы примеров для выполнения упражнений из книги можно бесплатно загрузить со страницы репозитория GitHub по адресу github.com/markjprice/cs7dotnetcore2. Как это сделать в Visual Studio 2017 и Visual Studio Code, я расписал в конце главы 1.

Структура издания

Глава 1 «Привет, C#! Здравствуй, .NET Core!» посвящена настройке среды разработки и использованию различных инструментов для создания простейшего приложения на языке C#. Вы узнаете, как писать и компилировать код с помощью среды разработки Visual Studio 2017 в операционной системе Windows, Visual Studio для Mac в системе macOS и Visual Studio Code в macOS, Linux и Windows. Помимо этого, вы найдете сведения о различных технологиях .NET: .NET Framework, .NET Core, .NET Standard и .NET Native.

Часть I. C# 7.1

Глава 2 «Говорим на языке C#» рассказывает о языке C#, его грамматике и лексике, которые вы будете применять каждый день, разрабатывая исходный код своих приложений. В частности, вы узнаете, как объявлять и использовать переменные разных типов.

Глава 3 «Управление потоком выполнения и преобразование типов» касается написания кода, который принимает решения, повторяет блок инструкций, преобразует типы и обрабатывает неизбежные ошибки. В ней также перечислены отличные источники справочной информации.

Глава 4 «Создание, отладка и тестирование функций» написана с учетом принципа DRY (Do not Repeat Yourself — не повторяйся). Она освещает приемы работы с инструментами отладки, мониторинга, диагностики проблем и тестирования кода для устранения ошибок и обеспечения высокой производительности, стабильности и надежности до ввода его в эксплуатацию.

Глава 5 «Создание пользовательских типов с помощью объектно-ориентированного программирования» рассказывает обо всех возможных видах членов типа, включая поля для хранения данных и методы для выполнения действий. Вы изучите концепции объектно-ориентированного программирования, такие как агрегирование и инкапсуляция, и возможности языка C# 7.1, например литералы `default` и предположительные имена кортежей.

Глава 6 «Реализация интерфейсов и наследование классов» посвящена производству новых типов на основе существующих с использованием объектно-ориентированного программирования. Вы узнаете, как определять операторы и локальные функции C# 7, делегаты и события, реализовывать интерфейсы, переопределять член типа. Кроме того, вы изучите базовые и производные классы, концепции полиморфизма, способы создания методов расширения и усвоите, как выполнять приведение классов в иерархии наследования.

Часть II. .NET Core 2.0 и .NET Standard 2.0

Глава 7 «Обзор и упаковка типов .NET Standard» описывает типы .NET Core 2.0, входящие в состав .NET Standard 2.0, и то, как они связаны с C#. Вдобавок вы узнаете о приемах развертывания и упаковки приложений и библиотек.

Глава 8 «Использование распространенных типов .NET Standard» описывает широко применяемые типы .NET Standard, которые позволяют коду выполнять рутинные задачи, такие как операции с числами и текстом, хранение элементов в коллекциях и глобализация приложений.

Глава 9 «Работа с файлами, потоками и сериализация» касается управления файловой системой, чтения и записи файлов и потоков, кодирования текста и сериализации.

Глава 10 «Защита данных и приложений» научит с помощью шифрования защищать данные от просмотра злоумышленниками, а также применять хеширование

и цифровые подписи для защиты от операций с данными или их искажения. Кроме того, вы узнаете об аутентификации и авторизации для защиты приложений от несанкционированного использования.

Глава 11 «Работа с базами данных с помощью Entity Framework Core» объясняет принципы чтения и записи в базы данных, такие как Microsoft SQL Server и SQLite, с использованием технологии объектно-реляционного отображения данных Entity Framework Core.

Глава 12 «Создание запросов и управление данными с помощью LINQ» рассказывает о технологии «Запрос, интегрированный в язык» (Language INtegrated Query) — наборе расширений языка, позволяющем работать с последовательностями элементов, выполняя их фильтрацию, сортировку и динамическое преобразование (проецирование) в иные структуры.

Глава 13 «Улучшение производительности и масштабируемости с помощью многозадачности» рассказывает, как организовать одновременное выполнение нескольких действий, чтобы повысить производительность, масштабируемость и продуктивность. Вы узнаете о добавленной в версии C# 7.1 возможности `async Main` и о том, как использовать типы в пространстве имен `System.Diagnostics` для контроля за производительностью и эффективностью кода.

Часть III. Модели приложений

Глава 14 «Создание сайтов с помощью ASP.NET Core Razor Pages» обучает основам построения сайтов с современной архитектурой HTTP на стороне сервера с применением ASP.NET Core. Вы познакомитесь с новой технологией Razor Pages в составе ASP.NET Core — она упрощает создание веб-страниц для небольших сайтов.

Глава 15 «Разработка сайтов с помощью ASP.NET Core MVC» посвящена разработке с использованием ASP.NET Core крупных, сложных сайтов с тем учетом, чтобы группа программистов могла легко их тестировать и поддерживать. Вы узнаете об основах конфигурации, аутентификации, авторизации, маршрутах, моделях, представлениях и контроллерах, которые составляют ASP.NET Core MVC.

Глава 16 «Создание веб-сервисов и приложений с помощью ASP.NET Core» описывает, как ASP.NET Core Web API помогают проектировать веб-приложения с применением современных технологий фронтенд-разработки, таких как Angular, React и фоновая архитектура REST.

Глава 17 «Разработка Windows-приложений с помощью языка XAML и системы проектирования Fluent» посвящена изучению основ языка XAML и системы проектирования Fluent, которые позволяют определять пользовательский интерфейс графических приложений для универсальной платформы Windows (UWP). Такие приложения можно запускать на Windows 10, Windows 10 Mobile, Xbox One и даже HoloLens.

Глава 18 «Разработка мобильных приложений с помощью XAML и Xamarin.Forms» рассказывает о создании кросс-платформенных веб-приложений для платформ iOS и Android. Клиентские мобильные приложения могут быть написаны

с помощью программы Visual Studio для Mac с применением технологий XAML и Xamarin.Forms.

Приложение содержит ответы на проверочные вопросы, приведенные в конце каждой главы.

Необходимое программное обеспечение

Разрабатывать и разворачивать приложения C# можно на многих операционных системах, включая Windows, macOS и большинство модификаций Linux. Чтобы сделать процесс программирования наиболее удобным и охватить большинство конфигураций, я советую познакомиться со всеми представителями семейства Visual Studio: Visual Studio 2017, Visual Studio Code и Visual Studio для Mac.

Я рекомендую воспользоваться следующими комбинациями операционной системы и среды разработки:

- Visual Studio 2017 в Windows 10;
- Visual Studio для Mac в macOS;
- Visual Studio Code в Windows 10 и macOS.

Лучше всего подойдет версия Microsoft Windows 10, поскольку она понадобится во время разработки приложений для универсальной платформы Windows в главе 17. Более ранние версии Windows, такие как 7 или 8.1, подойдут для выполнения упражнений во всех остальных главах.

Применительно к macOS, наиболее удобна версия Sierra или High Sierra. Операционная система macOS понадобится при создании мобильных приложений для устройств под управлением iOS в главе 18. Хотя вы и можете использовать Visual Studio 2017 в системе Windows, чтобы писать код мобильных приложений для устройств под управлением iOS и Android, система macOS и инструментарий Xcode необходимы для их компиляции.

Для кого эта книга

Эта книга для вас, если вы слышали, что:

- C# — популярный универсальный кросс-платформенный язык программирования, используемый для разработки чего угодно, начиная с бизнес-приложений, сайтов и сервисов и заканчивая играми;
- этот язык позволяет создавать программное обеспечение, которое работает на широком диапазоне устройств, от настольных компьютеров до серверов, от мобильных устройств до игровых систем, таких как Xbox One;
- с помощью .NET Core корпорация Microsoft сделала ставку на кросс-платформенное будущее .NET с оптимизацией разработки для серверной части, работающей в облаке, а также для устройств дополненной (AR) и виртуальной реальности (VR), таких как HoloLens;

- Microsoft выпустила популярный кросс-платформенный инструмент разработки под названием Visual Studio Code, с помощью которого и создаются эти кросс-платформенные приложения, а вам интересно его попробовать.

Условные обозначения

В книге вы увидите текст, оформленный различными стилями. Ниже я привел примеры стилей и объяснил, что означает все это форматирование.

Имена таблиц баз данных, имена компонентов проектов, URL-адреса выделены особым шрифтом, а ввод пользователя, имена каталогов и файлов (а также пути к ним и расширения файлов) — моноширинным шрифтом. Например, так: «Перейдите по ссылке github.com/markjprice/cs7dotnetcore2 и загрузите файл `Northwind.sql` для работы с Microsoft SQL Server в Windows».

Ключевые слова языка C# и листинги отформатированы моноширинным шрифтом, так: «Обратите внимание, что ключевое слово `public` указывается перед словом `class`» и так:

```
// хранение элементов с индексами позиций
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";
```

Если я хочу обратить ваше внимание на определенную часть приведенного кода, соответствующие строки или элементы выделены **полужирным моноширинным** шрифтом:

```
// хранение элементов с индексами позиций
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";
```

Таким же образом выделен весь ввод/вывод в командной строке:

```
dotnet new console
```

Новые термины и важные слова выделены *курсивным* шрифтом. Слова, которые вы видите на экране, к примеру, в меню или на кнопке в диалоговом окне, отображаются в тексте так: «В окне Install (Установка) нажмите кнопку Next (Далее) для перехода к следующему экрану».

Предупреждения или важные примечания даны с таким значком.

А советы экспертов по разработке — с таким.

Загрузка примеров кода

Примеры кода для выполнения упражнений из этой книги доступны для скачивания по адресу goo.gl/1wFh7Y.

Скачать файлы можно, выполнив следующие шаги.

1. Перейдите по адресу github.com/markjprice/cs7dotnetcore2.
2. Нажмите кнопку Clone or Download (Клонировать или скачать).
3. На открывшейся панели щелкните на ссылке Download Zip (Скачать Zip).

После того как архив будет загружен, можете распаковать его в нужную папку, используя последнюю версию одной из нижеперечисленных программ:

- WinRAR или 7-Zip для Windows;
- Zipeg или iZip или UnRarX для macOS;
- 7-Zip или PeaZip для Linux.

Файлы с примерами для книги также размещены на сайте GitHub по адресу github.com/PacktPublishing/CSharp-7.1-and-.NET-Core-2.0-Modern-Cross-Platform-Development-Third-Edition. Дерзайте!

Цветные изображения из книги

Вы можете просмотреть PDF-файл с цветными версиями рисунков из книги. Полноцветные изображения помогут быстрее разобраться в примерах. Файл доступен по адресу https://www.packtpub.com/sites/default/files/downloads/CSharp71andNETCore20ModernCrossPlatformDevelopmentThirdEdition_ColorImages.pdf.

1 Привет, C#! Здравствуй, .NET Core!

Эта глава посвящена настройке вашей среды разработки, сходствам и различиям между .NET Core, .NET Framework, .NET Standard и .NET Native, а также использованию различных инструментов для создания простейших приложений с помощью C# 7 и .NET Core.

Многие люди читают сложную документацию, предпочитая имитировать и повторять код за автором вместо глубокого изучения теории. Поэтому я тоже не стану объяснять каждое ключевое слово или шаг. Идея в следующем: дать вам задание написать некий код, собрать приложение и посмотреть, что происходит при запуске. Вам не нужно будет разбираться, как все работает.

Выражаясь словами Сэмюэля Джонсона (Samuel Johnson), составившего в 1755 году толковый словарь английского языка, я, вероятно, допустил «несколько диких промахов и забавных несуразиц, без которых не обходится ни одна из работ подобной сложности». Я принимаю на себя полную ответственность за них и надеюсь, что вы оцените мою попытку *«попасть в струю»* и написать книгу о .NET Core и инструментах командной строки этой среды в процессе ее непростого рождения в 2016–2017 годах.

В этой главе:

- настройка среды разработки;
- знакомство с .NET;
- написание и компиляция кода с помощью интерфейса командной строки .NET Core;
- написание и компиляция кода с применением Microsoft Visual Studio 2017;
- написание и компиляция кода с использованием Microsoft Visual Studio Code;
- написание и компиляция кода с помощью Microsoft Visual Studio для Mac;
- управление исходным кодом с применением GitHub.

Настройка среды разработки

Прежде чем приступать непосредственно к программированию, вам нужно настроить *интегрированную среду разработки* (Integrated Development Environment, IDE), содержащую редактор кода C#. Корпорация Microsoft выпустила целое семейство IDE:

- Visual Studio 2017;
- Visual Studio для Mac;
- Visual Studio Code.

Самая зрелая и полнофункциональная среда — *Microsoft Visual Studio 2017*, но, к сожалению, ее можно запустить на компьютерах только под управлением операционной системы Windows.

Самой современной и упрощенной кросс-платформенной средой разработки, также созданной корпорацией Microsoft, является *Microsoft Visual Studio Code*. Ее можно запустить во всех распространенных операционных системах, включая Windows, macOS и множество разновидностей Linux, таких как *Red Hat Enterprise Linux (RHEL)* и Ubuntu.

Чтобы решить, подходит ли вам среда Visual Studio Code, я рекомендую посмотреть видеоролик, доступный по ссылке https://channel9.msdn.com/Blogs/raw-tech/Beginners-Guide-to-VS-Code.

Для создания мобильных приложений наиболее подходит *Visual Studio для Mac*. Разрабатывать приложения для устройств под управлением операционной системы iOS (iPhone и iPad), tvOS, macOS и watchOS поможет компьютер с установленной системой macOS и среда Xcode. Хотя вы и можете использовать среду Visual Studio 2017 с расширениями Xamarin для *написания* кода кросс-платформенного мобильного приложения, вам все равно понадобится система macOS и среда Xcode для *компиляции* этого приложения.

В табл. 1.1 я перечислил среды разработки и операционные системы, которые могут или должны использоваться для выполнения примеров из каждой главы этой книги.

Таблица 1.1

Глава	Среда разработки	Операционная система
1–16	Visual Studio 2017	Windows 7 SP1 или выше
1–16	Visual Studio Code	Windows, macOS или Linux
1–16	Visual Studio для Mac	macOS
17	Visual Studio 2017	Windows 10
18	Visual Studio для Mac	macOS

Если есть такая возможность, я рекомендую попытаться выполнить все упражнения в обеих средах: как в Visual Studio 2017 в операционной системе Windows, так и в Visual Studio Code в macOS, Linux или Windows. Так вы получите прекрасный опыт программирования на C# и .NET Core с помощью различных инструментов разработки в разных операционных системах.

При работе над этой книгой я использовал программное обеспечение следующих версий (рис. 1.1):

- Visual Studio 2017 в операционной системе Windows 10 (в виртуальной машине);
- Visual Studio для Mac в операционной системе macOS;
- Visual Studio Code в операционной системе macOS;
- Visual Studio Code в операционной системе Red Hat Enterprise Linux (не показано на рисунке).

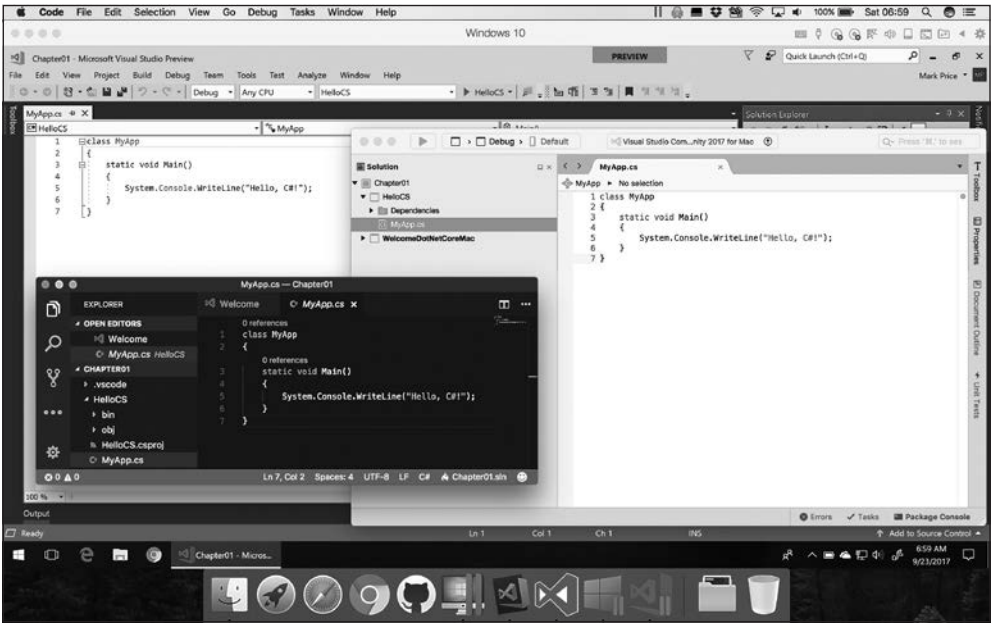


Рис. 1.1

Альтернативные среды разработки C#

Существуют и альтернативные среды разработки для программирования на языке C#, например *MonoDevelop* и *JetBrains Rider*. Вы можете установить их, воспользовавшись следующими URL.

- Для установки среды разработки MonoDevelop посетите сайт http://www.monodevelop.com/.

- Для установки среды разработки JetBrains Rider посетите сайт https://www.jetbrains.com/rider/.

Кроме того, существует среда разработки *Cloud9*, запускаемая прямо в браузере, так что она еще более кросс-платформенная, чем остальные. Благодаря этой особенности среда набирает популярность. См. дополнительную информацию по адресу https://c9.io/web/sign-up/free.

Кросс-платформенное развертывание

Выбор среды разработки и используемой операционной системы не влияет на место развертывания программы. .NET Core 2.0 поддерживает для развертывания следующие платформы:

- Windows 7 SP1 или версии выше;
- Windows Server 2008 R2 SP1 или версии выше;
- Windows IoT 10 или версии выше;
- macOS Sierra (версия 10.12) или версии выше;
- Red Hat Enterprise Linux 7.3 или версии выше;
- Ubuntu 14.04 LTS или версии выше;
- Fedora 25 или версии выше;
- Debian 8.7 или версии выше;
- OpenSUSE 42.2 или версии выше;
- Tizen 4 или версии выше.

Операционные системы Linux как платформы для хостинга серверов популярны, поскольку относительно легковесны и более рентабельны в плане масштабируемости, в отличие от операционных систем наподобие Windows и macOS, скорее направленных на конечного пользователя.

В следующем подразделе вы установите среду Microsoft Visual Studio 2017 в операционной системе Windows. Если предпочитаете использовать программу Microsoft Visual Studio Code, то пропустите этот текст и переходите к подразделу «Установка Microsoft Visual Studio Code для Windows, macOS или Linux». Если же предпочитаете Microsoft Visual Studio для Mac, то переходите к подразделу «Установка Microsoft Visual Studio для macOS».

Установка Microsoft Visual Studio 2017

Для выполнения примеров из большинства глав этой книги можно использовать операционную систему Windows версии 7 SP1 или более поздней, но работать будет удобнее, если применять Windows 10 с установленным обновлением Fall Creators Update.

Начиная с октября 2014 года корпорация Microsoft стала выпускать новую полнофункциональную редакцию среды Visual Studio, доступную всем желающим бесплатно. Эта редакция называется *Community Edition*.

Скачайте пакет Microsoft Visual Studio 2017 версии не ниже 15.4 с сайта https://www.visualstudio.com/downloads/.

Вы должны установить пакет Visual Studio 2017 версии не ниже 15.4, чтобы иметь возможность взаимодействовать с .NET Core для универсальной платформы Windows. Версия Visual Studio 2017 не ниже 15.3 требуется для работы с .NET Core 2.0. Более старые версии Visual Studio 2017 поддерживают только .NET Core 1.0 и 1.1.

Выбор рабочих нагрузок

На вкладке Workloads (Рабочие нагрузки) установите флажки напротив следующих компонентов (рис. 1.2):

- Universal Windows Platform development (Разработка приложений для универсальной платформы Windows);
- .NET desktop development (Разработка классических приложений .NET);
- ASP.NET and web development (ASP.NET и разработка веб-приложений);
- Azure development (Разработка для Azure);
- Node.js development (Разработка Node.js);
- .NET Core cross-platform development (Кросс-платформенная разработка .NET Core).

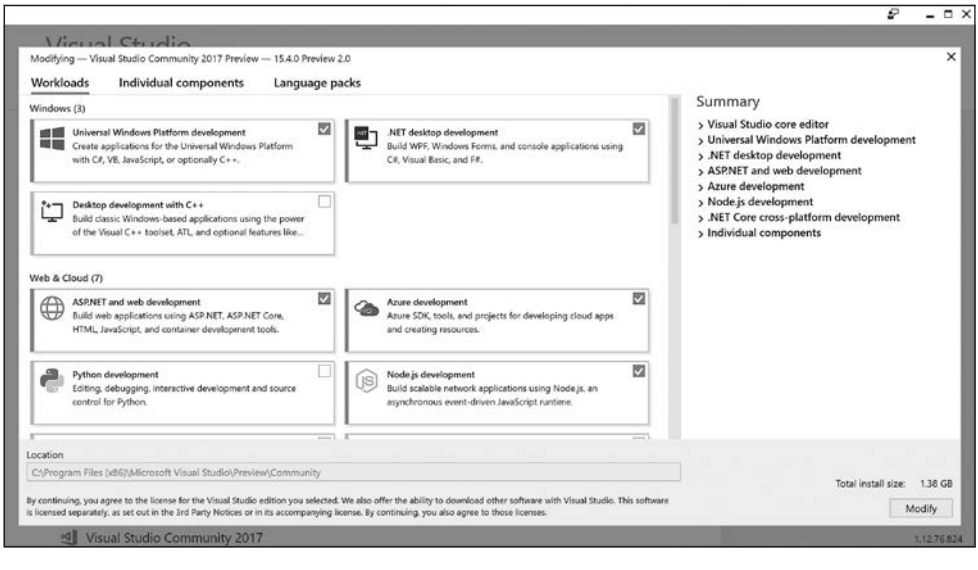


Рис. 1.2

Выбор дополнительных компонентов

На вкладке Individual components (Отдельные компоненты) установите флажки у следующих компонентов:

- Class Designer (Конструктор классов);
- GitHub extension for Visual Studio (Расширение GitHub для Visual Studio);
- PowerShell tools (Инструменты PowerShell).

Нажмите кнопку Install (Установить) и подождите, пока выбранные компоненты не будут установлены. После завершения процесса установки нажмите кнопку Launch (Запустить).

Пока вы дожидаетесь установки среды разработки Visual Studio 2017, можно перейти к разделу «Знакомство с .NET» далее в этой главе.

Когда вы запустите Visual Studio 2017 в первый раз, увидите окно с запросом авторизации. Если вы ранее зарегистрировали учетную запись Microsoft, то можете воспользоваться регистрационными данными такого аккаунта. В случае отсутствия учетной записи зарегистрируйте новую, перейдя по ссылке https://signup.live.com/.

При первом запуске Visual Studio 2017 вы увидите окно с предложением настроить среду под свои предпочтения. В раскрывающемся списке Development Settings (Настройки разработки) выберите пункт Visual C#. В качестве цветовой схемы я выбрал вариант Blue (Голубая), но ваш выбор может быть другим.

После запуска Microsoft Visual Studio вы увидите пользовательский интерфейс программы с открытой начальной страницей в центральной области. Как и большинство других классических Windows-приложений, интерфейс Visual Studio содержит строку меню, панель инструментов с часто вызываемыми командами и строку состояния внизу окна. В правой части окна программы расположена панель Solution Explorer (Обозреватель решений) со списком открытых проектов (рис. 1.3).

Для последующего быстрого запуска среды Visual Studio щелкните правой кнопкой мыши на значке этой программы на панели задач Windows и в контекстном меню выберите пункт Pin this program to taskbar (Закрепить на панели задач).

Для работы с главами 14, 15 и 16 необходимо установить платформу Node.js и менеджер пакетов NPM.

Скачайте дистрибутив Node.js для операционной системы Windows с сайта https://nodejs.org/en/download/.

Запустите мастер установки Node.js (рис. 1.4).

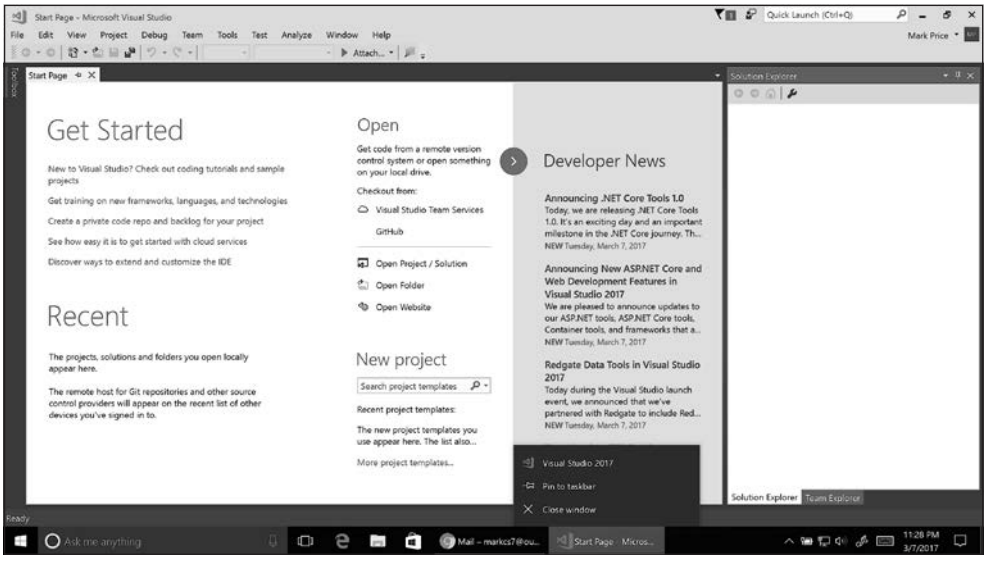

Start Page - Microsoft Visual Studio
File Edit View Project Debug Team Tools Test Analyze Window Help
Start Page
Get Started
New to Visual Studio? Check out coding tutorials and sample projects
Get training on new frameworks, languages, and technologies
Create a private code repo and backlog for your project
See how easy it is to get started with cloud services
Discover ways to extend and customize the IDE
Recent
The projects, solutions and folders you open locally appear here.
The remote host for Git repositories and other source control providers will appear on the recent list of other devices you've signed in to.
Open
Get code from a remote version control system or open something on your local drive.
Checkout from:
Visual Studio Team Services
GitHub
Open Project / Solution
Open Folder
Open Website
New project
Search project templates
Recent project templates:
The new project templates you use appear here. The list also...
More project templates...
Developer News
Announcing .NET Core Tools 1.0
Today, we are releasing .NET Core Tools 1.0. It's an exciting day and an important milestone in the .NET Core journey. Th...
NEW Tuesday, March 7, 2017
Announcing New ASP.NET Core and Web Development Features in Visual Studio 2017
We are pleased to announce updates to our ASP.NET tools, ASP.NET Core tools, Container tools, and frameworks that a...
NEW Tuesday, March 7, 2017
Redgate Data Tools in Visual Studio 2017
Today during the Visual Studio launch event, we announced that we've partnered with Redgate to include Red...
NEW Tuesday, March 7, 2017
Visual Studio 2017
Pin to taskbar
Close window
Solution Explorer
Team Explorer
Ready
Ask me anything
11:26 PM
3/7/2017

Рис. 1.3

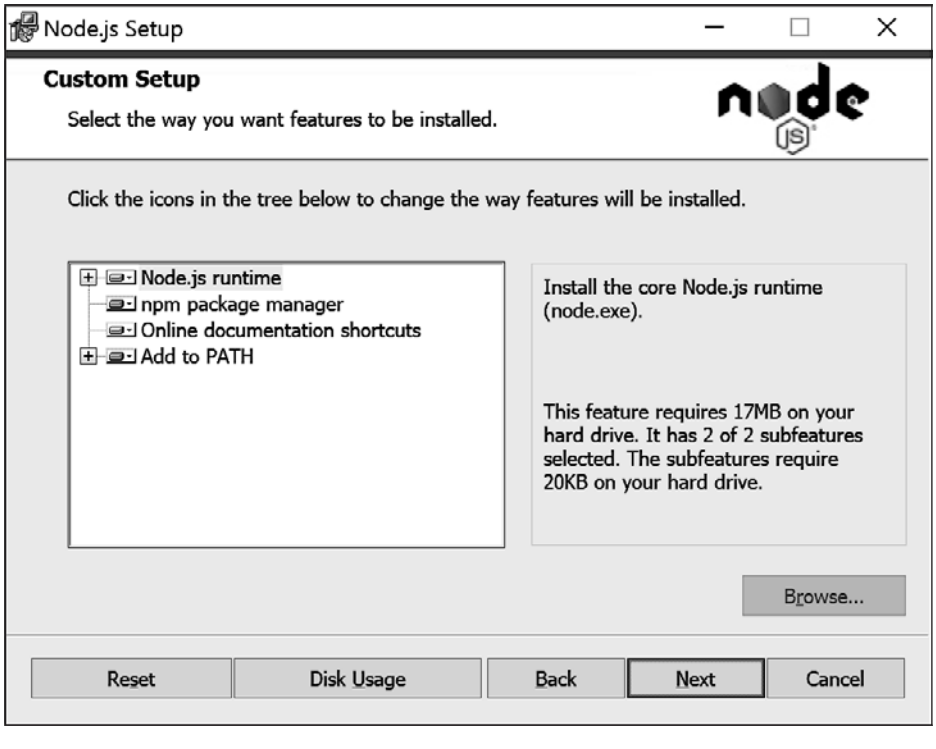

Node.js Setup
Custom Setup
Select the way you want features to be installed.
Click the icons in the tree below to change the way features will be installed.
Node.js runtime
npm package manager
Online documentation shortcuts
Add to PATH
Install the core Node.js runtime (node.exe).
This feature requires 17MB on your hard drive. It has 2 of 2 subfeatures selected. The subfeatures require 20KB on your hard drive.
Browse...
Reset
Disk Usage
Back
Next
Cancel

Рис. 1.4

Установка Microsoft Visual Studio Code

В период с июня 2015-го по сентябрь 2017 года корпорация Microsoft выпускала новую версию среды Visual Studio Code ежемесячно. Эта среда стала быстро развиваться и поразила сотрудников корпорации Microsoft растущей популярностью. Даже если вы планируете пользоваться Visual Studio 2017 или Visual Studio для Mac как основным инструментом разработки, я рекомендую научиться применять среду Visual Studio Code и средства командной строки .NET Core.

Скачать дистрибутив Visual Studio Code можно на сайте https://code.visualstudio.com/.

Ознакомиться с планами корпорации Microsoft для Visual Studio Code на 2018 год можно, перейдя по ссылке https://github.com/Microsoft/vscode/wiki/Roadmap.

Установка Microsoft Visual Studio Code для Windows, macOS или Linux

Выполняя примеры и создавая снимки рабочего процесса, я использовал версию Visual Studio Code для операционной системы macOS. Шаги для версий Visual Studio Code для операционной системы Windows и различных модификаций Linux практически неотличимы, поэтому я не буду приводить инструкции для каждой из платформ.

Инструкции по выбору языка интерфейса среды разработки Visual Studio Code приведены на странице code.visualstudio.com/docs/getstarted/locales.

После загрузки дистрибутива Visual Studio Code для операционной системы macOS перетащите его значок на папку Applications (Приложения) (рис. 1.5).

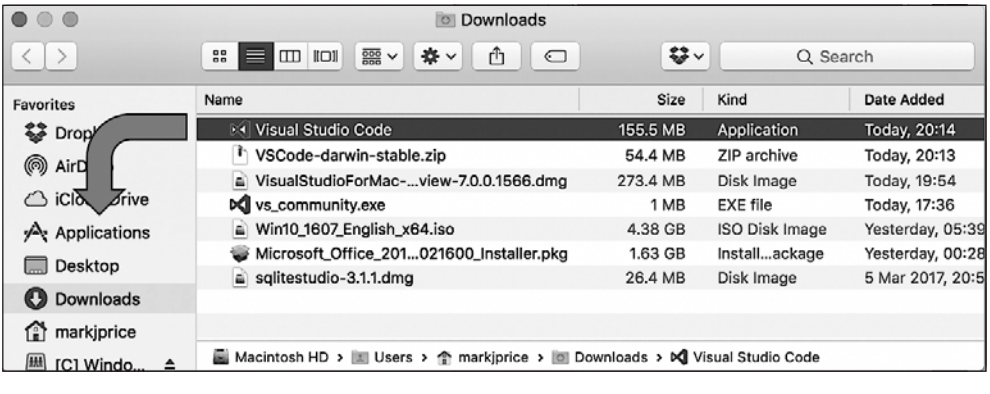


Рис. 1.5

Теперь нужно установить пакет инструментов .NET Core SDK для операционной системы macOS. Полная инструкция вместе с обучающим видеороликом доступна

по ссылке https://www.microsoft.com/net/core#macos, а в книге я приведу лишь базовые инструкции, как мы и договаривались.

Сначала требуется установить менеджер пакетов Homebrew (если, конечно, он еще не установлен).

Запустите встроенное в операционную систему macOS приложение Terminal (Терминал) и введите следующую команду в окне консоли:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Приложение Terminal (Терминал) выведет запрос о необходимости нажатия клавиши Enter для продолжения, а затем предложит ввести пароль администратора.

Если вы пользуетесь версией .NET Core 1.0 или 1.1, то на данном этапе Homebrew понадобится для установки криптографического пакета OpenSSL, необходимого для работы старых версий .NET Core в операционной системе macOS.

Установка .NET Core SDK в macOS

Следующий шаг — загрузка и установка дистрибутива .NET Core SDK для операционной системы macOS (x64). Скачать этот пакет можно по ссылке https://www.microsoft.com/net/download/core.

Запустите установочный пакет `dotnet-sdk-2.0.0-sdk-osx-x64.pkg`, чтобы перейти к его установке (рис. 1.6).

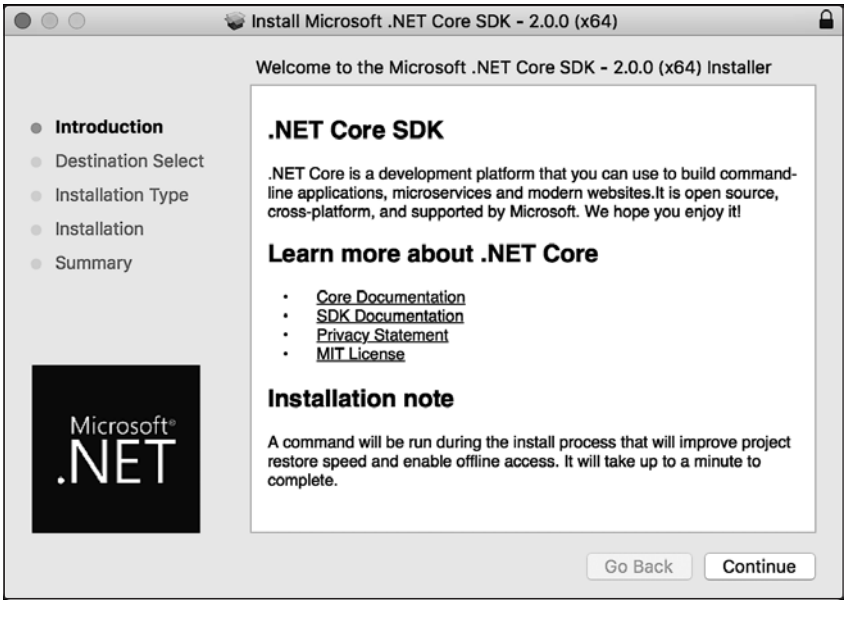


Рис. 1.6

Нажмите кнопку Continue (Продолжить), примите лицензионное соглашение, нажмите кнопку Install (Установить), а по завершении процесса установки — кнопку Close (Закрыть).

Установка менеджера пакетов NPM в macOS

Для работы с главами 14, 15 и 16 необходимо установить менеджер пакетов NPM (Node Package Manager).

Запустите встроенное в операционную систему macOS приложение Terminal (Терминал) и введите в окне консоли команды для установки платформы Node.js и менеджера NPM. Затем проверьте их версии (на момент написания этой книги — Node.js 8.4 и NPM 5.3) (рис. 1.7).

```
brew install node
node -v
npm -v
```

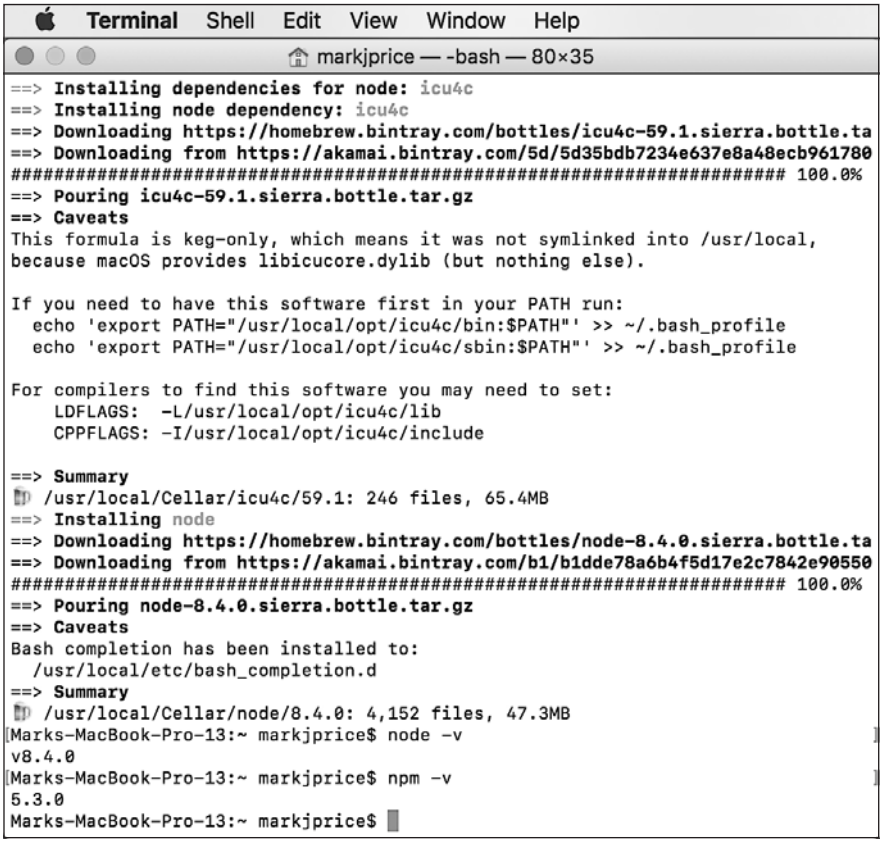


Рис. 1.7

Установка расширения C# в среде Visual Studio Code

Данное расширение не обязательно, но оно встраивает технологию автодополнения IntelliSense по мере ввода кода, поэтому рекомендуется к установке.

Запустите Visual Studio Code и перейдите на вкладку Extensions (Расширения) либо выполните команду View ▸ Extensions (Вид ▸ Расширения) или нажмите сочетание клавиш Cmd+Shift+X.

C# — очень популярное расширение, поэтому, скорее всего, вы увидите его в верхней части списка (рис. 1.8).

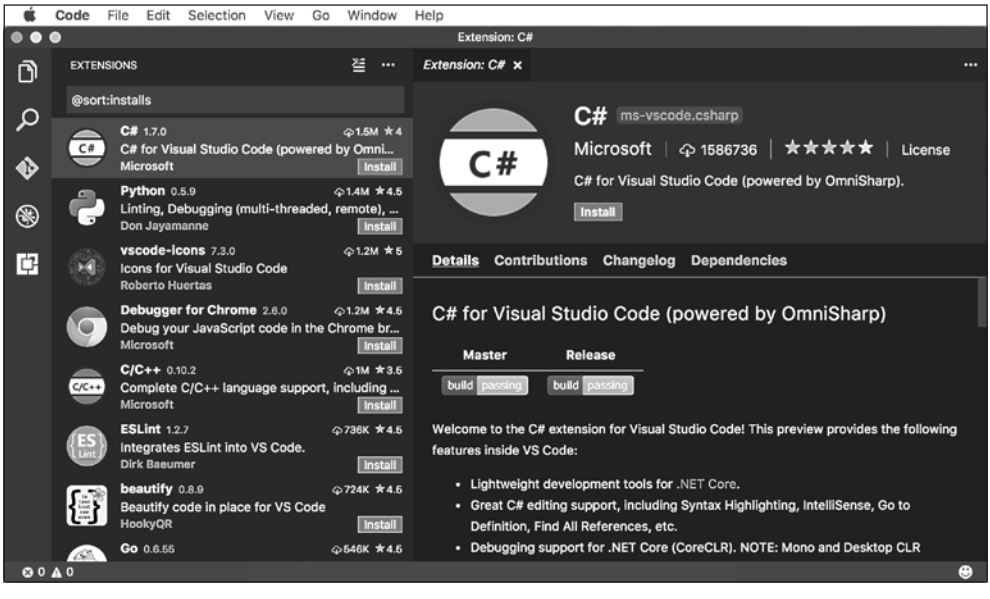


Рис. 1.8

Нажмите кнопку Install (Установить), а затем кнопку Reload (Перезагрузка) для перезагрузки окна программы и активизации установленного расширения.

Установка Microsoft Visual Studio для Mac

В ноябре 2016 года корпорация Microsoft выпустила предварительную версию Visual Studio для Mac. Первоначально она годилась только для разработки мобильных приложений Xamarin, поскольку версия являлась ответвлением продукта Xamarin Studio. Финальный релиз, ставший доступным в мае 2017 года, поддерживает создание библиотек классов .NET Standard 2.0, веб-приложений и сервисов ASP.NET Core, а также консольных приложений, поэтому Visual Studio для Mac можно применять для выполнения (практически) всех упражнений данной книги.

Хотя Visual Studio 2017 для Windows и подходит для написания кода мобильных приложений для iOS и Android, только среда Xcode, запущенная под управлением macOS или OS X, позволяет скомпилировать приложения под операционную систему iOS. Поэтому я считаю, что для разработки мобильных приложений нужно использовать программу Visual Studio для Mac.

Установка среды разработки Xcode

Если вы еще не установили эту среду на свой компьютер Mac, то сделайте это прямо сейчас из магазина App Store.

В строке меню щелкните на значке в виде яблока и выберите пункт App Store.

В окне приложения App Store введите `xcode` в поле поиска, и одним из первых результатов будет среда разработки Xcode (рис. 1.9).

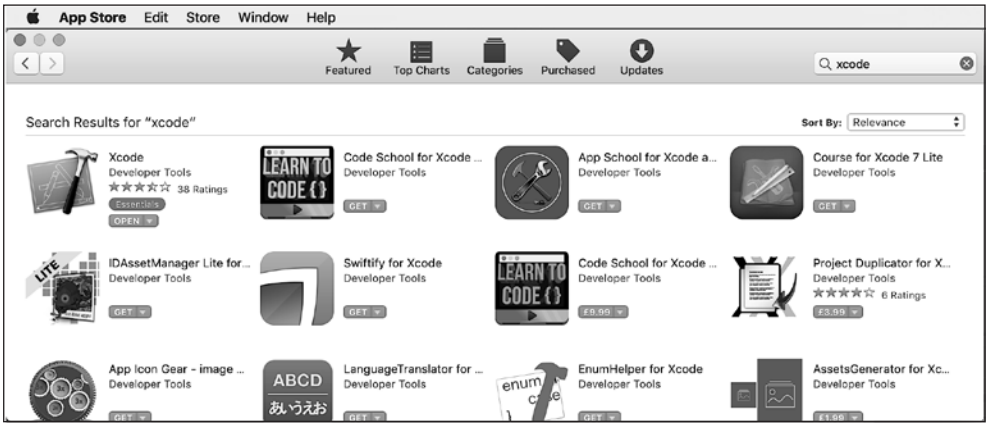


Рис. 1.9

Нажмите кнопку Get (Получить) и дождитесь установки Xcode.

Загрузка и установка Visual Studio для Mac

Перейдите по ссылке https://www.visualstudio.com/vs/visual-studio-mac/, чтобы скачать дистрибутив среды разработки Visual Studio для Mac и запустить его установку.

В окне Visual Studio for Mac Installer (Установщик Visual Studio для Mac) примите условия лицензионного соглашения, выберите все компоненты для установки, а затем нажмите кнопку Continue (Продолжить) (рис. 1.10).

Нажмите кнопку Continue (Продолжить), а затем Install (Установить).

Примите условия лицензионного соглашения для компонентов, таких как Android SDK, нажмите кнопку Continue (Продолжить) и дождитесь полной загрузки и установки среды разработки Visual Studio для Mac.

Запустите Visual Studio для Mac. Вы увидите начальную страницу среды разработки (рис. 1.11).

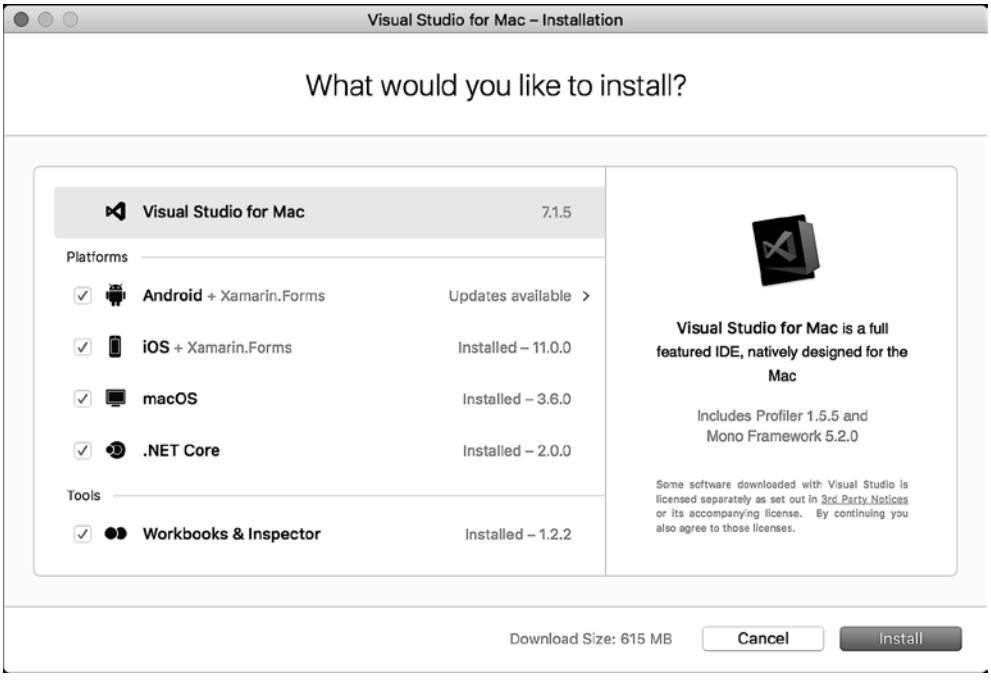


Рис. 1.10

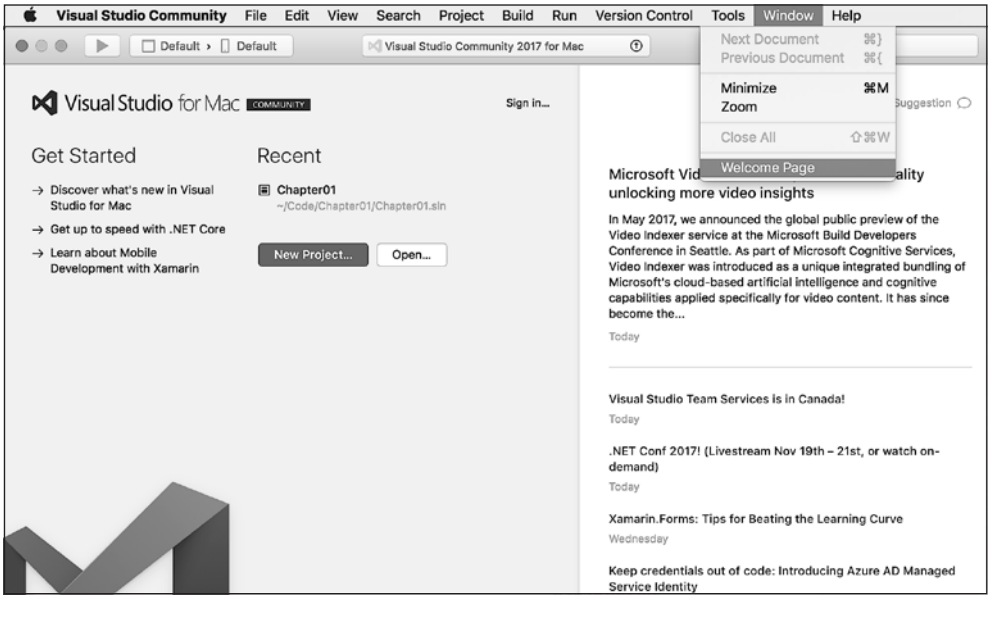


Рис. 1.11

Если появится уведомление о необходимости обновления компонентов, то нажмите кнопку Restart and Install Updates (Перезагрузить и установить обновления).

Теперь, после установки и настройки среды разработки, вы познакомитесь с основами .NET, прежде чем погрузиться в написание кода.

Знакомство с .NET

.NET Framework, .NET Core, .NET Standard и .NET Native — связанные и иногда пересекающиеся платформы для разработки приложений и сервисов.

Обзор .NET Framework

Microsoft .NET Framework — платформа для разработки, включающая *общеязыковую исполняющую среду* (Common Language Runtime, CLR), которая управляет выполнением кода и обеспечивает создателей программ богатыми библиотеками классов для написания приложений.

.NET Framework изначально проектировалась как кросс-платформенная, но затем Microsoft сконцентрировалась на внедрении платформы и обеспечении максимально эффективной работы в операционной системе Windows.

В сущности, данная платформа работает только в среде Windows и считается устаревшей.

Проекты Mono и Xamarin

Сторонние разработчики создали реализацию .NET под названием Mono, о которой можно почитать на сайте http://www.mono-project.com/.

Хоть Mono и был кросс-платформенным проектом, он значительно отставал от официальной реализации .NET Framework. Позднее Mono занял нишу в качестве основы мобильной платформы Xamarin одноименной компании.

В 2016 году корпорация Microsoft приобрела компанию Xamarin и теперь бесплатно предоставляет пользователям когда-то дорогостоящее решение Xamarin в качестве расширения для среды разработки Visual Studio 2017. Кроме того, Microsoft переименовала инструменты для разработчика Xamarin Studio в Visual Studio для Mac и внедрила возможность создавать веб-API с помощью ASP.NET Core. Проект Xamarin нацелен на создание мобильных приложений и облачных сервисов для их поддержки.

Обзор .NET Core

Сегодня мы живем в истинно кросс-платформенном мире. Современные методы мобильной и облачной разработки уменьшили прежнюю значимость операционной

системы Windows. Поэтому Microsoft активно работает над отделением платформы .NET от Windows, прерыванием их тесных связей.

При переписывании кода .NET для обеспечения истинной кросс-платформенности сотрудники корпорации Microsoft воспользовались моментом, чтобы реорганизовать компоненты .NET и удалить те из них, которые не считаются ядром.

В итоге свет увидел новый продукт — *.NET Core*, включающий кросс-платформенную реализацию рабочей общеязыковой исполняющей среды под названием *CoreCLR* и набор библиотек классов *CoreFX*.

«Сорок процентов пользователей .NET Core — новые разработчики, — утверждает директор .NET-подразделения Microsoft Скотт Хантер (Scott Hunter). — Мы хотим привлекать новых людей».

В табл. 1.2 отражены основные выпуски .NET Core.

Таблица 1.2

Версия	Дата выпуска
.NET Core RC1	Ноябрь 2015 года
.NET Core 1.0	Июнь 2016 года
.NET Core 1.1	Ноябрь 2016 года
.NET Core 1.0.4 (LTS) и .NET Core (текущая версия) 1.1.1	Март 2017 года
.NET Core 2.0	Август 2017 года
.NET Core 2.0 для UWP в версии Windows 10 Fall Creators Update	Октябрь 2017 года
.NET Core 2.1	Февраль 2018 года

Если вы намерены взаимодействовать с версиями .NET Core 1.0 и 1.1, то я рекомендую прочитать статью с замечаниями к выпуску .NET Core 1.1, которая будет полезна для всех разработчиков на платформе .NET Core. Статья доступна по адресу https://blogs.msdn.microsoft.com/dotnet/2016/11/16/announcing-net-core-1-1/.

Платформа .NET Core стала существенно меньше, чем текущая версия .NET Framework, так как многие компоненты были удалены.

К примеру, компоненты Windows Forms и Windows Presentation Foundation (WPF) могут служить для создания приложений с *графическим пользовательским интерфейсом* (graphical user interface, GUI), но тесно связаны с Windows, вследствие чего были удалены из .NET Core. В настоящее время для проектирования приложений Windows применяется технология под названием «Универсальная платформа Windows» (Universal Windows Platform, UWP), внедренная в .NET Core. Об этом вы узнаете в главе 17.

Компоненты ASP.NET Web Forms и Windows Communication Foundation (WCF) — устаревшие технологии для создания веб-приложений и сервисов,

используемые сегодня лишь некоторыми разработчиками, так что эти компоненты тоже были удалены из .NET Core. Вместо них разработчики предпочитают компоненты ASP.NET MVC и ASP.NET Web API. Две последние технологии были реорганизованы и объединены в новый продукт, ASP.NET Core, работающий на платформе .NET Core. Вы узнаете о технологии ASP.NET Core MVC в главе 15, о технологии ASP.NET Core Razor Pages — в главе 14, а о технологии ASP.NET Web API и одностраничных приложениях (SPA) типа Angular и React — в главе 16.

Entity Framework (EF) 6 — технология объектно-реляционного отображения для работы с информацией, хранящейся в реляционных базах данных, таких как Oracle и Microsoft SQL Server. За годы развития данная технология накопила приличный багаж дополнений, вследствие чего кросс-платформенная версия была оптимизирована и приобрела название Entity Framework Core. Об этом вы узнаете в главе 11.

Помимо удаления крупных компонентов из .NET Framework при подготовке платформы .NET Core, Microsoft распределила компоненты .NET Core в виде небольших NuGet-пакетов, которые могут быть развернуты независимо друг от друга.

Основная цель корпорации Microsoft — не сделать .NET Core меньше, чем .NET Framework. Задача состоит в следующем: разделить .NET Core на компоненты для поддержки современных технологий и уменьшения числа зависимостей, чтобы для развертывания требовались только те пакеты, которые необходимы для работы приложения.

Обзор .NET Standard

На сегодня с платформой .NET сложилась такая ситуация: существует три ее ветви, и все разрабатываются корпорацией Microsoft:

- .NET Framework;
- .NET Core;
- Xamarin.

Все они имеют различные сильные и слабые стороны, поскольку предназначены для разных ситуаций. Это привело к тому, что разработчик должен изучить три платформы, каждая из которых раздражает своими странностями и ограничениями.

Вот как корпорация Microsoft описывает .NET Standard 2.0: это спецификация для набора API, который будет реализован всеми платформами .NET. Вы не можете установить .NET Standard 2.0 так же, как не можете установить HTML5. Чтобы использовать HTML5, вы должны установить браузер, который реализует спецификацию HTML5. А для применения .NET Standard 2.0 следует установить платформу .NET, реализующую данную спецификацию.

.NET Standard 2.0 реализована в последних версиях .NET Framework, .NET Core и Xamarin. Она значительно упрощает процесс совместной работы над кодом на разных платформах .NET.

.NET Standard 2.0 добавляет многие из недостающих API, которые требуются разработчикам для портирования старого кода, написанного под .NET Framework, на платформу .NET Core. Тем не менее отдельные API *реализованы* так, чтобы при запуске вызывать исключение, указывающее на то, что на самом деле данные методы не должны использоваться! Чаще это связано с различиями в операционных системах, в которых ведется работа с .NET Core. Вы узнаете, как справиться с описанной проблемой, в главе 2.

Ниже представлен графический обзор упомянутых трех вариаций .NET (иногда называемых моделями приложений) и их места в общей библиотеке .NET Standard 2.0 и инфраструктуре (рис. 1.12).

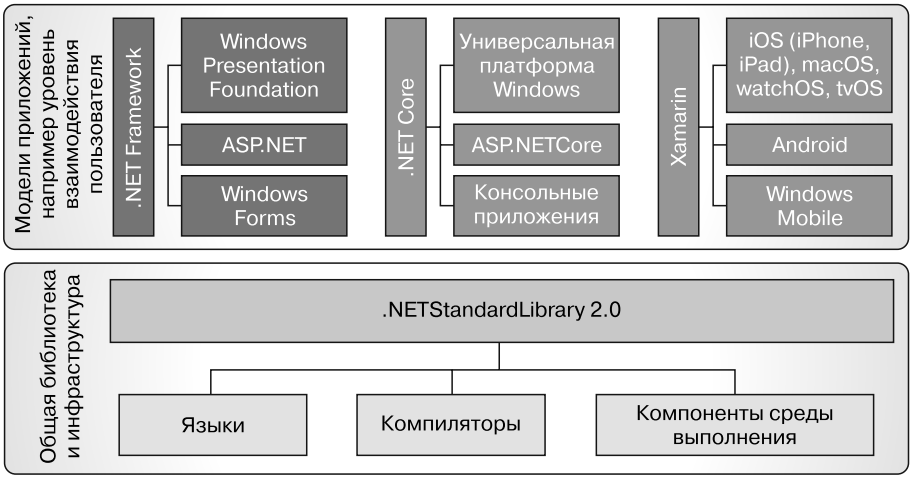


Рис. 1.12

В первом издании этой книги я сфокусировался на .NET Core, но использовал .NET Framework, если важные или полезные функции не были реализованы в .NET Core. Для большинства примеров я задействовал среду разработки Visual Studio 2015 и лишь кратко упомянул Visual Studio Code.

Из второго издания я удалил все примеры кода, относящиеся к .NET Framework.

В третьем издании работа была завершена. Все примеры были переписаны с учетом .NET Core и могут быть выполнены в Visual Studio 2017, Visual Studio для Mac или Visual Studio Code в любой поддерживаемой операционной системе. Исключения касаются только двух последних глав. В главе 17 .NET Core используется для разработки UWP-приложений и требует запуска среды Visual Studio 2017 в операционной системе Windows 10, а в главе 18 будет применяться Xamarin вместо .NET Core.

Обзор .NET Native

Другой инициативой в области .NET является технология .NET Native. С ее помощью код на языке C# компилируется в машинный код методом *компиляции перед исполнением* (ahead-of-time, AoT) вместо общеязыковой исполняющей среды, применяемой для *динамической компиляции* (just-in-time, JIT) в *промежуточный язык* (intermediate language, IL) и только затем в машинный язык.

Технология .NET Native увеличивает скорость выполнения приложений и уменьшает выделяемые объемы памяти. Поддерживаются следующие типы приложений:

- UWP-приложения для систем Windows 10, Windows 10 Mobile, Xbox One, HoloLens и устройств *«Интернет вещей»* (Internet of Things, IoT) типа микрокомпьютера Raspberry Pi;
- серверная веб-разработка с помощью ASP.NET Core;
- консольные приложения для использования в командной строке.

Сравнение технологий .NET

В табл. 1.3 перечисляются и сравниваются технологии .NET.

Таблица 1.3

Технология	Возможности	Компилирует в	Хостовая ОС
.NET Framework	Как отработанные, так и современные	Промежуточный язык	Только Windows
Xamarin	Только мобильные	Промежуточный язык	iOS, Android, Windows Mobile
.NET Core	Только современные	Промежуточный язык	Windows, Linux, macOS
.NET Native	Только современные	Машинный код	Windows, Linux, macOS

Написание и компиляция кода с помощью интерфейса командной строки .NET Core

Во время установки среды Visual Studio 2017, Visual Studio для Mac или .NET Core SDK универсальный драйвер для цепочки инструментов *интерфейса командной строки* (Command Line Interface, CLI) под названием `dotnet` устанавливается вместе с исполняющей средой .NET Core.

Прежде чем использовать инструменты интерфейса командной строки, такие как `dotnet`, попрактикуемся в написании кода!

Кодирование в простом текстовом редакторе

Если вашей операционной системой является Windows, то запустите приложение Notepad (Блокнот).

Если вы пользуетесь операционной системой macOS, то запустите приложение TextEdit. В меню TextEdit перейдите в Preferences (Настройки), снимите флажок Smart quotes (Смарт-кавычки), а затем закройте диалоговое окно. Выполните команду Format ▸ Make Plain Text (Формат ▸ Простой текст).

Или же запустите простой текстовый редактор, с которым работаете.

Введите следующий код:

```
class MyApp { static void Main() {
  System.Console.WriteLine("Hello, C#!"); } }
```

Язык C# чувствителен к регистру символов, поэтому нужно вводить строчные и прописные буквы точно так же, как показано в примере. Однако данный язык не учитывает пробельные символы; следовательно, с точки зрения языка совершенно неважно, как вы используете отступы, пробелы и символы возврата каретки для организации кода.

Можно ввести весь код в виде одной строки или же распределить на несколько строк, украсив отступами. Так, код, показанный ниже, после компиляции приведет к тому же результату, который выдает и первый пример:

```
class
          MyApp     {
  Static                        void
Main            (){System.        Console.
      WriteLine(       "Hello, C#!");        }        }
```

Разумеется, код стоит писать так, чтобы другие программисты и вы сами смогли без проблем прочитать его через месяцы или даже годы!

Если вы используете Notepad (Блокнот) Windows

В приложении Notepad (Блокнот) в меню File (Файл) выполните команду Save As (Сохранить как).

В открывшемся диалоговом окне выберите диск C: (или другой диск, на котором вы будете хранить свои проекты), нажмите кнопку New folder (Новая папка) и присвойте папке имя `Code`. Откройте ее, вновь нажмите кнопку New folder (Новая папка) и присвойте папке имя `Chapter01`. Откройте ее, вновь нажмите кнопку New folder (Новая папка) и присвойте папке имя `HelloCS`. Откройте ее.

В раскрывающемся списке Save as type (Тип файла) выберите пункт All Files (Все файлы), чтобы избавиться от файлового расширения `.txt`, а затем введите имя файла `MyApp.cs` (рис. 1.13).

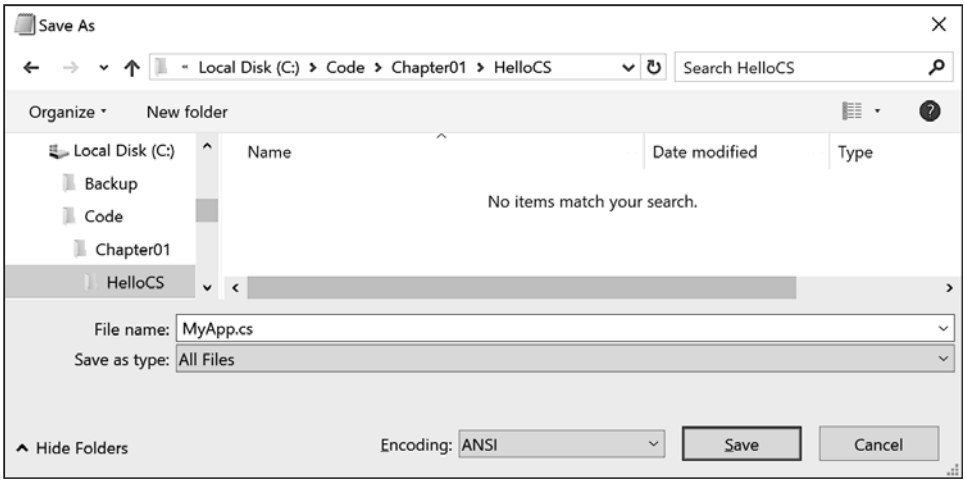


Рис. 1.13

Набранный код в приложении Notepad (Блокнот) должен выглядеть примерно так (рис. 1.14).

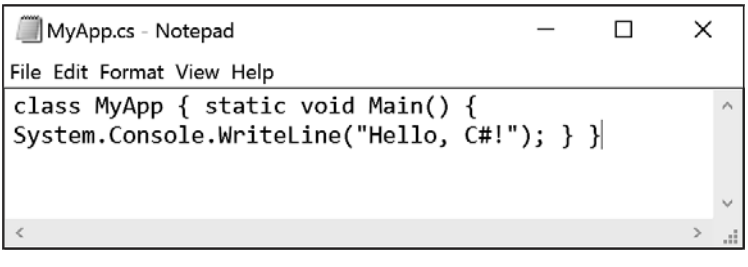


Рис. 1.14

Если вы используете TextEdit (macOS)

В приложении TextEdit в меню File (Файл) выполните команду Save (Сохранить) или нажмите сочетание клавиш Cmd+S.

В открывшемся окне смените свою пользовательскую папку (моя называется markjprice) на каталог, в котором будете хранить свои проекты, нажмите кнопку New folder (Новая папка) и присвойте папке имя Code. Откройте ее, вновь нажмите кнопку New folder (Новая папка) и присвойте папке имя `Chapter01`. Откройте ее, вновь нажмите кнопку New folder (Новая папка) и присвойте папке имя `HelloCS`. Откройте ее.

В раскрывающемся списке Plain Text Encoding (Кодировка простого текста) выберите пункт Unicode (UTF-8) (Юникод (UTF-8)), снимите флажок If no extension is provided, use ".txt" (Если расширение не указано, использовать .txt), чтобы избавиться от файлового расширения `.txt`, а затем введите имя файла `MyApp.cs` и нажмите кнопку Save (Сохранить).

Создание и компиляция приложений с помощью интерфейса командной строки .NET Core

Если вашей операционной системой является Windows, то запустите приложение Command Prompt (Командная строка).

Если вы пользуетесь операционной системой macOS, то запустите приложение Terminal (Терминал).

В окне консоли введите команду `dotnet`. Вы должны увидеть следующий результат выполнения команды, содержащий информацию о введенной команде (рис. 1.15).

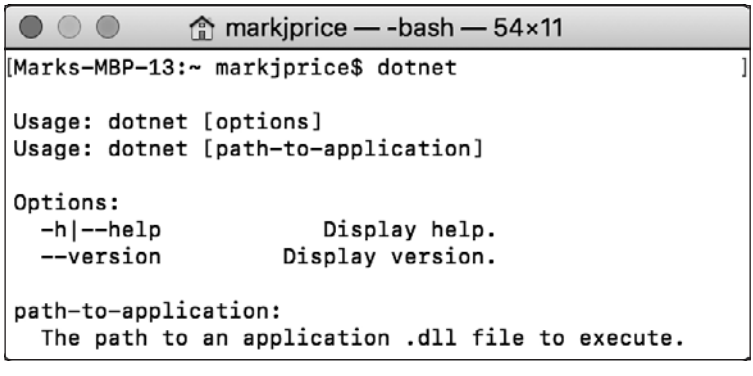


Рис. 1.15

Вывод команды dotnet одинаков во всех операционных системах — Windows, macOS и Linux.

Создание консольного приложения с помощью интерфейса командной строки

Введите указанные далее команды в окне консоли. Они позволят выполнить следующее:

- сменить каталог проекта;
- создать новое консольное приложение в выбранном каталоге;
- вывести перечень всех файлов, созданных инструментом командной строки `dotnet`.

При использовании операционной системы Windows в окне программы Command Prompt (Командная строка) введите такие команды:

```
cd C:\Code\Chapter01\HelloCS
dotnet new console
dir
```

Если вы работаете с операционной системой macOS, то в приложении Terminal (Терминал) введите следующие команды:

```
cd Code/Chapter01/HelloCS
dotnet new console
ls
```

Вы увидите, что инструмент `dotnet` создал два новых файла (рис. 1.16, снимок экрана из операционной системы Windows):

- `Program.cs` — исходный код простого консольного приложения;
- `HelloCS.csproj` — файл проекта, содержащий зависимости и настройки проекта.

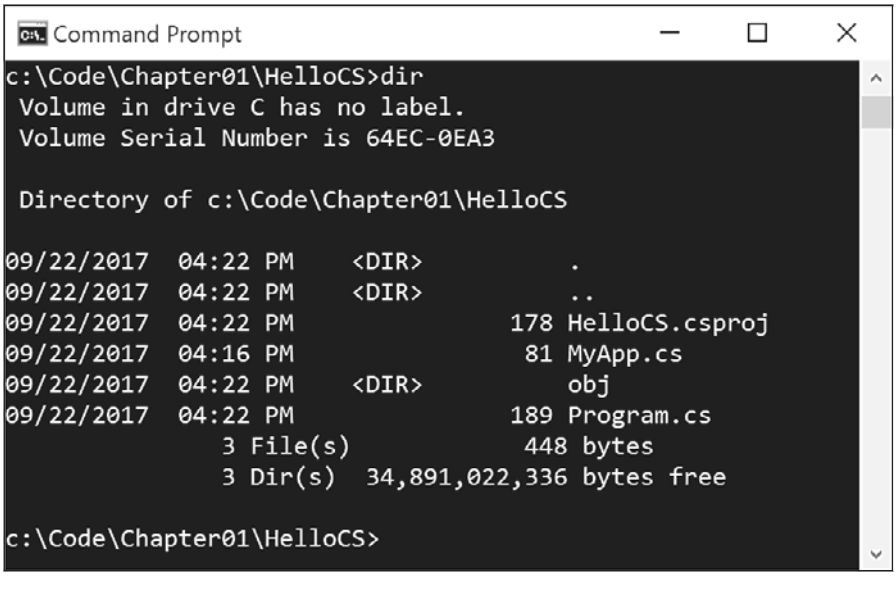


Рис. 1.16

В этом примере нужно удалить файл `Program.cs`, так как мы уже создали собственный класс в файле `MyApp.cs`.

При использовании операционной системы Windows в окне программы Command Prompt (Командная строка) введите такую команду:

```
del Program.cs
```

Если вы работаете с операционной системой macOS, то в приложении Terminal (Терминал) введите следующую команду:

```
rm Program.cs
```

Во всех будущих примерах мы будем пользоваться файлом Program.cs, генерируемым командой, а не создавать каждый раз собственный.

Восстановление пакетов, компиляция кода и запуск приложения

В окне консоли введите команду `dotnet run`. Через несколько секунд все пакеты, необходимые нашей программе, будут загружены, исходный код скомпилирован, а приложение запущено, о чем сообщается в выводе в операционной системе macOS (рис. 1.17):

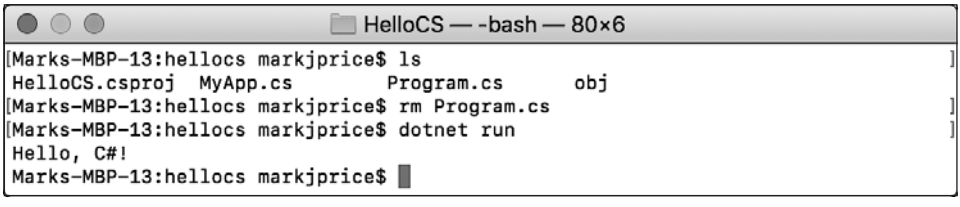


Рис. 1.17

Ваш исходный код, сохраненный в файле `MyApp.cs`, был скомпилирован в сборку `HelloCS.dll` и записан в подкаталог `bin/Debug/netcoreapp2.0`. (Найдите его в файловой системе, если очень хочется, а затем возвращайтесь к чтению.)

На данный момент эта сборка может быть запущена только с помощью команды `dotnet run`. Из главы 7 вы узнаете, как упаковывать и публиковать сборки для использования в любых операционных системах, поддерживающих .NET Core.

Решение проблем компиляции

Если компилятор выдаст сообщения об ошибках, внимательно прочитайте их и внесите исправления в код в текстовом редакторе. Сохраните документ и попробуйте снова скомпилировать приложение.

В окне консоли можно использовать клавиши ↑ и ↓ для цикличного переключения между введенными командами.

Обычно ошибки заключаются в использовании неправильного регистра букв, пропуска точки с запятой в конце строки или пропуска парной фигурной скобки. К примеру, если по ошибке вы напишете название метода `Main` со строчной буквы `m`, то увидите следующее сообщение об ошибке:

```
error CS5001: Program does not contain a static 'Main' method suitable for an entry
point
```

Знакомство с промежуточным языком

Компилятор C# (под названием Roslyn), используемый командой `dotnet`, конвертирует ваш исходный код на языке C# в код на промежуточном языке (IL) и сохраняет его в *сборке* (DLL- или EXE-файле).

Инструкции на IL похожи на код ассемблера, только выполняются с помощью виртуальной машины CoreCLR в .NET Core.

Во время работы CoreCLR загружает IL-код из сборки, динамически (JIT) компилирует в машинные инструкции для процессора, после чего процессор вашего компьютера выполняет код.

Преимущество такого двухэтапного процесса компиляции в том, что Microsoft может создавать общеязыковые исполняющие среды (CLR) и для Linux с macOS, а не только для Windows. Один и тот же IL-код можно выполнить на любой платформе благодаря второму этапу процесса компиляции, во время которого генерируется код для соответствующей операционной системы и процессора.

Вне зависимости от того, на каком языке написан исходный код, C# или, к примеру, F#, во всех приложениях .NET используется IL-код для хранения инструкций в сборках. Существуют дизассемблеры корпорации Microsoft и других компаний, позволяющие открывать сборки и извлекать из них IL-код.

На самом деле не во всех приложениях .NET используется IL-код! Для некоторых применяется компилятор .NET Native, генерирующий машинный код вместо IL-кода, что положительно сказывается на производительности и занимает меньший объем памяти, но за счет переносимости.

Написание и компиляция кода с применением Microsoft Visual Studio 2017

Теперь мы попробуем создать аналогичное приложение с помощью среды разработки Visual Studio 2017. Если ваш выбор пал на программное обеспечение Visual Studio Code, то я рекомендую все равно просмотреть изложенные далее инструкции и приведенные снимки экрана, поскольку процесс работы в Visual Studio Code очень схож, несмотря на отсутствие такого широкого функционала.

Я обучаю студентов работе в Visual Studio более десятилетия и всегда удивляюсь, как много программистов не способны полноценно использовать этот инструмент.

Несколько следующих страниц мы вместе станем писать одну строку кода. С первого взгляда данное действие может показаться излишним, но вам будет полезно увидеть, какую помощь и дополнительную информацию предоставляет Visual Studio по мере того, как вы вводите свой код. Если вы хотите писать код быстро и без ошибок, то позвольте Visual Studio набирать бóльшую часть кода за вас — это огромное преимущество!

Кодирование в Microsoft Visual Studio 2017

Запустите Visual Studio 2017.

Выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект) или нажмите сочетание клавиш Ctrl+Shift+N.

В списке Installed (Установленные) в левой части диалогового окна раскройте раздел Visual C# и выберите пункт .NET Core. В списке, расположенном в центре

этого окна, выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)). Присвойте ему имя `WelcomeDotNetCore`, укажите расположение проекта по адресу `C:\Code`, используйте имя решения `Chapter01`, а затем нажмите кнопку OK или клавишу Enter (рис. 1.18).

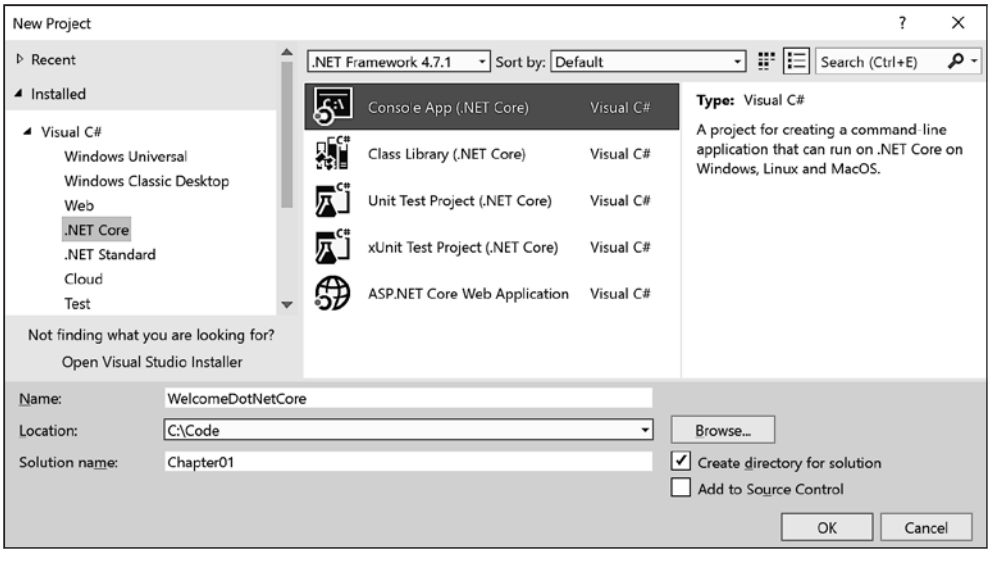


Рис. 1.18

Не обращайте внимания на пункт .NET Framework 4.7.1 в раскрывающемся списке в верхней части диалогового окна. Этот параметр не влияет на проекты .NET Core!

В области редактора кода удалите инструкцию, расположенную в строке 9:

```
Console.WriteLine("Hello World!");
```

В методе `Main` начните набирать буквы `sy` (рис. 1.19), и вы увидите появившееся меню IntelliSense.

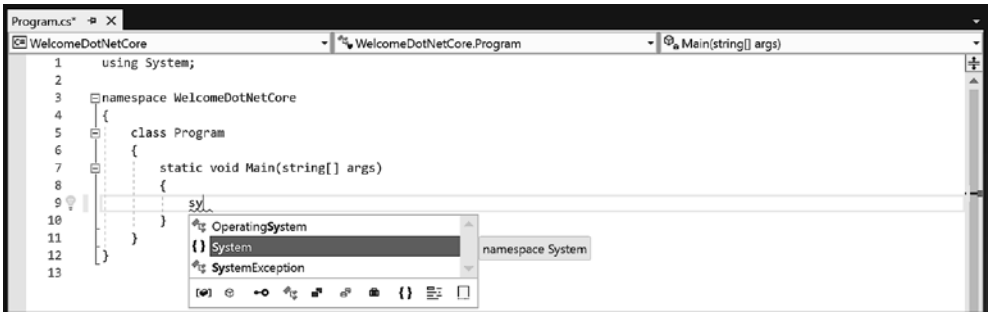


Рис. 1.19

Данное меню отображает отфильтрованный список *ключевых слов*, *пространств имен* и *типов*, имена которых начинаются с букв `sy`, и выделяет полужирным вхождения этих букв. В нашем примере с них начинается одно имя — пространство имен `System`.

Напечатайте точку (которую обычно ставите в конце предложения).

IntelliSense автоматически завершит ввод слова `System` вместо вас, добавит точку, а затем отобразит список типов, таких как `AggregateException` и `Action`, доступных в пространстве имен `System` (рис. 1.20).

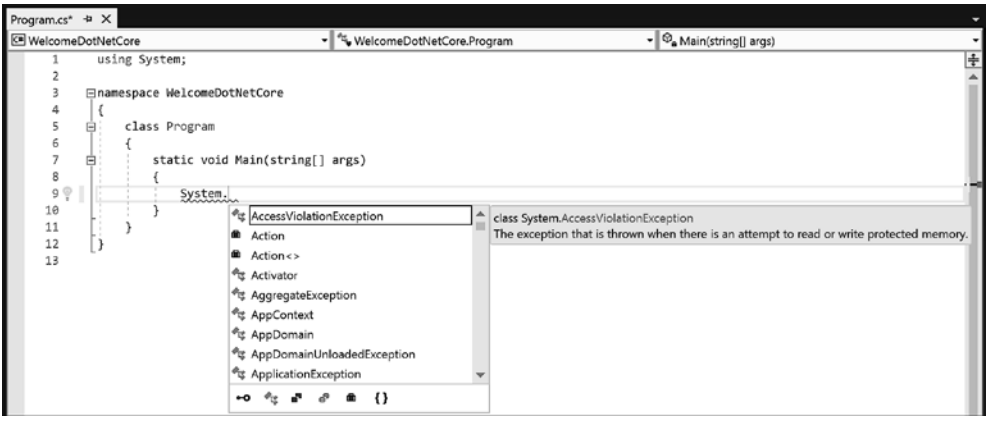


Рис. 1.20

Начните ввод букв `con`, чтобы IntelliSense отобразило список соответствующих типов и пространств имен (рис. 1.21).

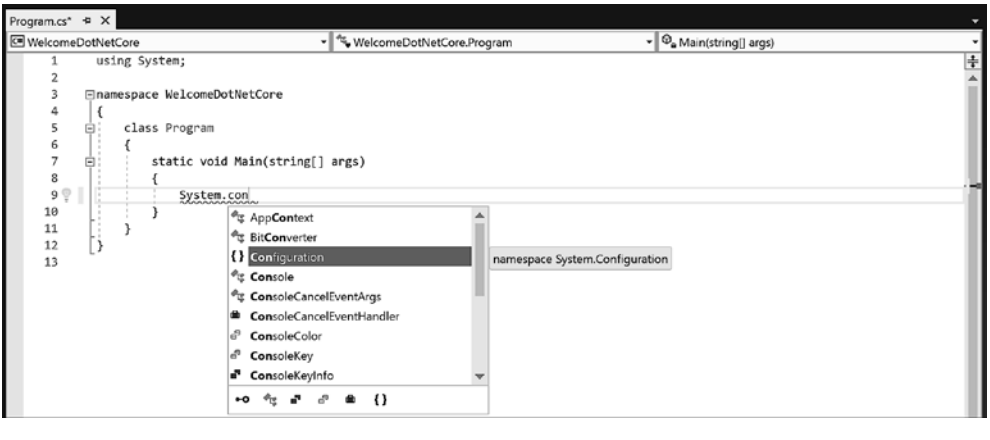


Рис. 1.21

Нам нужен пункт `Console`. Нажмите клавишу ↓, чтобы выбрать его. После того как данный пункт выделен, нажмите клавишу с точкой.

IntelliSense выведет список *членов* класса `Console` (рис. 1.22).

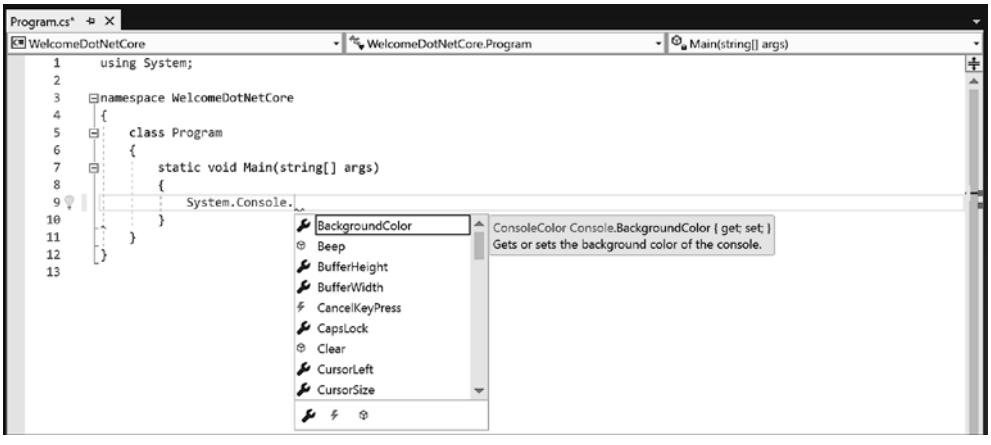


Рис. 1.22

К членам класса относятся свойства (атрибуты объекта, такие как BackgroundColor), методы (действия, которые объект может выполнять; например, Beep), события и другие связанные элементы.

Начните вводить буквы `wl`. IntelliSense отобразит два члена класса, `WindowLeft` и `WriteLine`, содержащих эти буквы (рис. 1.23).

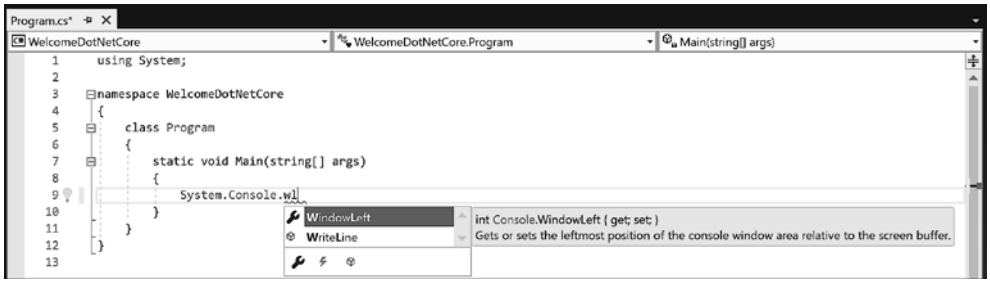


Рис. 1.23

Нажмите клавишу ↓, чтобы выделить пункт `WriteLine`, а затем напечатайте открывающую круглую скобку, `(`.

IntelliSense автоматически завершит ввод имени метода `WriteLine` и добавит закрывающую круглую скобку.

Кроме того, вы увидите сообщение о том, что существует 18 вариантов метода `WriteLine` (рис. 1.24).

Введите символ двойной кавычки, `"`. IntelliSense добавит второй символ двойной кавычки и установит между ними текстовый курсор.

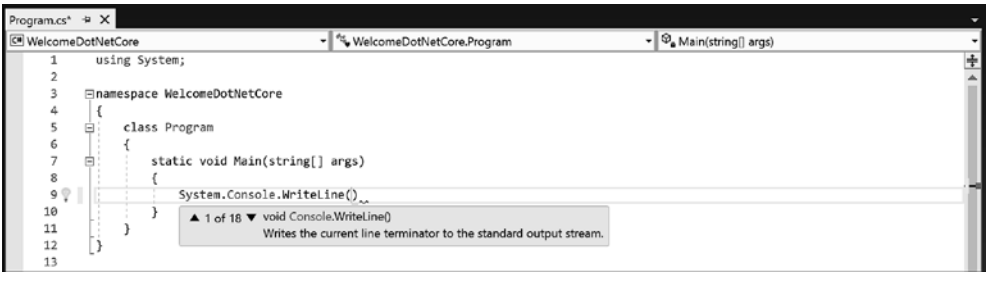


Рис. 1.24

Введите текст `Welcome, .NET Core!` (рис. 1.25).

Рис. 1.25

Красная волнистая черта в конце строки уведомляет об ошибке, поскольку каждая инструкция в языке C# должна завершаться точкой с запятой. Установите курсор в конец строки и введите точку с запятой для исправления ошибки.

Компиляция кода в Visual Studio 2017

В меню Debug (Отладка) выберите пункт Start Without Debugging (Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5.

В строке состояния Visual Studio сначала появится сообщение, что сборка начата (Build started), а затем — что она успешно завершена (Build succeeded). Ваше консольное приложение будет запущено в окне командной строки (рис. 1.26).

Рис. 1.26

Чтобы не увеличивать книгу до огромных размеров и сделать результат выполнения более наглядным, в дальнейшем я не стану приводить снимки с изображением

вывода консольного приложения, как показано на предыдущем рисунке. Вместо этого я буду приводить результат вывода, в данном примере так:

```
Welcome, .NET Core!
```

Работа над ошибками

Нарочно добавьте пару ошибок:

- измените прописную букву `M` в названии метода `Main` на строчную `m`;
- удалите последнюю букву `e` из имени метода `WriteLine`.

В меню Debug (Отладка) выберите пункт Start Without Debugging (Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5.

Через несколько секунд в строке состояния появится сообщение об ошибке сборки (Build failed) и диалоговое окно с ошибкой. Нажмите кнопку No (Нет).

Откроется панель Error List (Список ошибок) (рис. 1.27). Вы можете вызвать ее самостоятельно, нажав сочетание клавиш Ctrl+W или Ctrl+E.

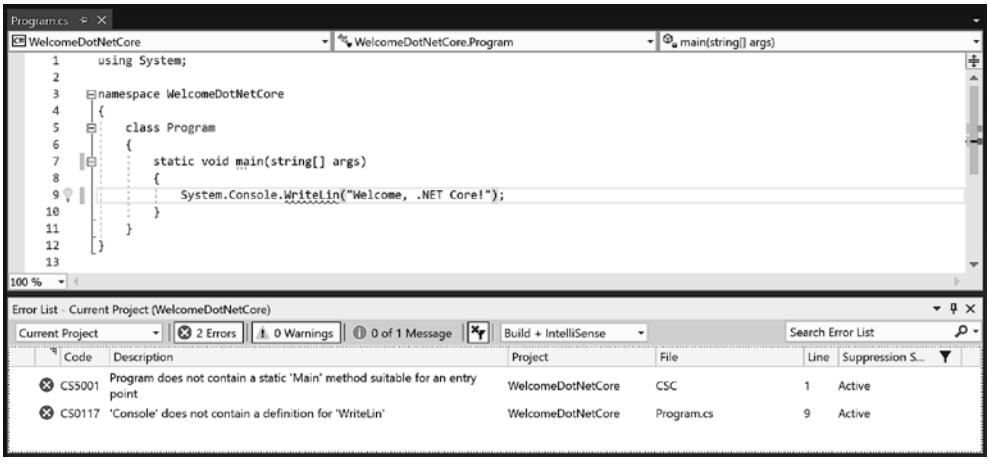


Рис. 1.27

Содержимое списка можно фильтровать по типу ошибки: Errors (Ошибки), Warnings (Предупреждения) и Messages (Сообщения), для чего следует нажать соответствующую кнопку-переключатель в верхней части панели Error List (Список ошибок).

Если в строке с ошибкой указаны имя файла и номер строки, к примеру File (Файл): Program.cs и Line: 9, то можно дважды щелкнуть на ошибке для перехода к строке в коде, которая содержит ошибку.

При более универсальном характере ошибки (например, при отсутствии метода `Main`) компилятор не отобразит номер строки. Вероятно, вы действительно хотите, чтобы у вас в коде был метод `main`, а не только `Main` (помните, язык C# чувствителен к регистру, поэтому можно использовать оба метода в коде одной программы).

Однако Visual Studio может проанализировать код и вывести предложения по его улучшению. Например, что имена методов должны начинаться с прописной буквы (рис. 1.28).

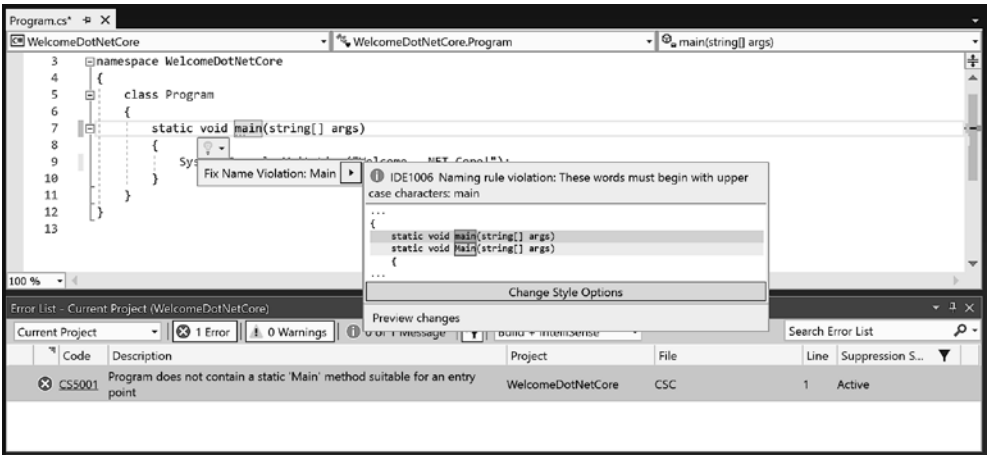


Рис. 1.28

Исправьте две ошибки, показанные на предыдущем снимке экрана, и перезапустите приложение, чтобы проверить, успешно ли оно работает, прежде чем переходить к следующему подразделу. Обратите внимание: содержимое панели Error List (Список ошибок) обновится и ошибки исчезнут.

Добавление ранее созданных проектов в Visual Studio 2017

Ранее вы разработали проект с помощью команды `dotnet`. Теперь, когда у вас есть решение, созданное в Visual Studio 2017, может понадобиться добавить в него этот проект.

Выполните команду File ▸ Add ▸ Existing Project (Файл ▸ Добавить ▸ Существующий проект), перейдите в каталог `C:\Code\Chapter01\HelloCS`, выберите файл `HelloCS.csproj` и нажмите кнопку Open (Открыть).

Чтобы получить возможность запустить этот проект, на панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на пункте Solution 'Chapter01' (2 projects) (Решение Chapter01 (проектов: 2)) и в контекстном меню выберите пункт Properties (Свойства) или нажмите сочетание клавиш Alt+Enter.

В категории Startup Project (Запускаемый проект) установите переключатель в положение Current selection (Текущий проект) и нажмите кнопку OK.

На панели Solution Explorer (Обозреватель решений) щелкните на любом файле в проекте HelloCS, а затем в меню Debug (Отладка) выберите пункт Start Without Debugging (Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5.

Автоматическое форматирование кода

Код проще читать и понимать, если при его оформлении применяются общие правила использования отступов и пробелов.

Если ваш код может быть скомпилирован, то Visual Studio 2017 позволяет автоматически отформатировать его с помощью отступов и пробелов.

На панели Solution Explorer (Обозреватель решений) дважды щелкните на файле `MyApp.cs` (рис. 1.29).

Рис. 1.29

Выполните команду Build ▸ Build HelloCS (Сборка ▸ Построить HelloCS) или нажмите сочетание клавиш Shift+F6 и дождитесь завершения сборки, а затем выполните команду Edit ▸ Advanced ▸ Format Document (Редактировать ▸ Дополнительно ▸ Форматировать документ) или нажмите сочетание клавиш Ctrl+E, D. Ваш код будет автоматически отформатирован (рис. 1.30).

Рис. 1.30

Интерактивный C#

Хотя среда разработки Visual Studio всегда содержала панель Immediate (Интерпретация) с ограниченной поддержкой функции REPL для реализации интерактивной среды языка программирования, версия Visual Studio 2017 включает улучшенный REPL-инструмент — панель с полноценным меню IntelliSense и цветовой дифференциацией кода под названием C# Interactive (Интерактивный C#).

Для доступа к этой панели в Visual Studio 2017 откройте меню View (Вид), раскройте в нем пункт Other Windows (Другие окна), а затем выберите C# Interactive (Интерактивный C#).

В интерактивном режиме мы напишем некий код для загрузки страницы About с публичного сайта корпорации Microsoft.

Код приведен исключительно в качестве примера. На данный момент вам не нужно полностью понимать его!

На панели C# Interactive (Интерактивный C#) с помощью инструкций необходимо выполнить следующее:

- сослаться на сборку `System.Net.Http`;
- импортировать пространство имен `System.Net.Http`;
- объявить и присвоить значение переменной HTTP-клиента;
- установить значение адреса клиента со ссылкой на сайт Microsoft;
- ожидать ответ после отправки асинхронного GET-запроса страницы About;
- прочитать код состояния, возвращенный веб-сервером;
- прочитать заголовок типа содержимого;
- прочитать содержимое HTML-страницы в виде строки.

Введите каждую из приведенных ниже команд после приглашения >, а затем нажмите клавишу Enter:

```
> #r "System.Net.Http"
> using System.Net.Http;
> var client = new HttpClient();
> client.BaseAddress = new Uri("http://www.microsoft.com/");
> var response = await client.GetAsync("about");
> response.StatusCode
OK
> response.Content.Headers.GetValues("Content-Type")
string[1] { "text/html" }
> await response.Content.ReadAsStringAsync()
"<!DOCTYPE html ><html
xmlns:mscom="http://schemas.microsoft.com/CMSvNext"
xmlns:md="http://schemas.microsoft.com/mscom-data" lang="en"
xmlns="http://www.w3.org/1999/xhtml"><head><meta http-equiv="X-UACompatible"
content="IE=edge" /><meta charset="utf-8" /><meta
name="viewport" content="width=device-width, initial-scale=1.0"
/><link rel="shortcut icon"
href="//www.microsoft.com/favicon.ico?v2" /><script
type="text/javascript"
src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-
1.7.2.min.js">rn // Third party scripts and code linked to
or referenced from this website are licensed to you by the parties
that own such code, not by Microsoft. See ASP.NET Ajax CDN Terms of
Use - http://www.asp.net/ajaxlibrary/CDN.ashx.rn
```

```
</script><script type="text/javascript"
language="javascript">/*<![CDATA[*/if($(document).bind("mobileinit
",function(){$.mobile.autoInitializePage=!1}),navigator.userAgent.ma
tch(/IEMobile\/10\.0/)){var
msViewportStyle=document.createElement("style ...
```

На рис. 1.31 показано, как Visual Studio 2017 отреагировала на ввод описанных команд на панели C# Interactive (Интерактивный C#).

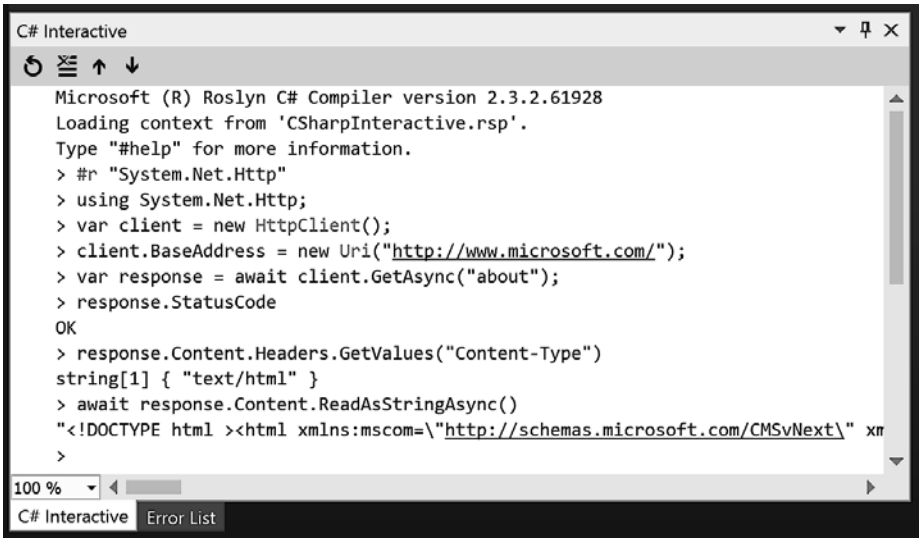


Рис. 1.31

Компилятор C# носит название Roslyn. Версия 1.0 использовалась в C# 6, 2.0 — в языке C# 7. Версия Roslyn 2.3 применяется в языке C# 7.1.

Другие полезные окна

Среда разработки Visual Studio 2017 содержит и другие полезные панели, включая следующие:

- на панели Solution Explorer (Обозреватель решений) вы можете управлять своими проектами и файлами;
- на панели Team Explorer найдете инструменты для управления исходным кодом;
- панель Server Explorer (Обозреватель серверов) используется для управления подключениями к базам данных и ресурсами на платформе Microsoft Azure.

Если в окне среды разработки не отображаются нужные вам панели, то перейдите в меню View (Вид), чтобы открыть их или подсмотреть соответствующее сочетание клавиш (рис. 1.32).

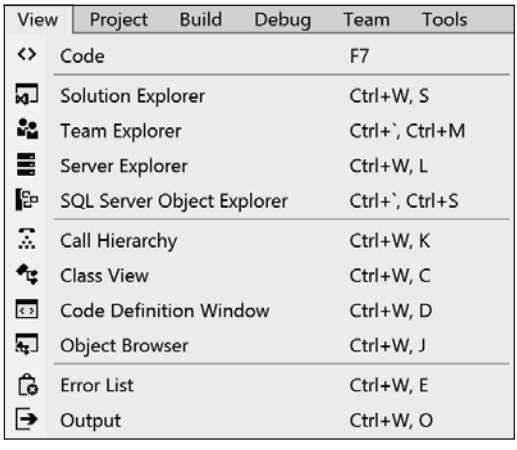


Рис. 1.32

Если сочетания клавиш в вашей программе отличаются от перечисленных на предыдущем снимке экрана, значит, вы выбрали другой набор параметров клавиатуры при установке среды разработки Visual Studio. Чтобы перенастроить сочетания клавиш на те, которые используются в этой книге, откройте меню Tools (Сервис), выберите пункт Import and Export Settings (Импорт и экспорт параметров). В появившемся диалоговом окне установите переключатель в положение Reset all settings (Сбросить все параметры), а затем выберите набор параметров Visual C#.

Написание и компиляция кода с использованием Microsoft Visual Studio Code

Инструкции и снимки экрана в этом разделе приведены для версии Visual Studio Code для операционной системы macOS. В операционных системах Windows и Linux выполняются аналогичные действия. Основные различия заключаются в собственных консольных командах, таких как удаление файла: и синтаксис команды, и путь могут различаться. Интерфейс `dotnet` идентичен для всех платформ.

Кодирование в Visual Studio Code

Запустите Visual Studio Code.

Выполните команду File ▸ Open (Файл ▸ Открыть) или нажмите сочетание клавиш Cmd+O.

В открывшемся диалоговом окне откройте каталог `Code`, а затем — папку `Chapter01`. Нажмите кнопку New Folder (Новая папка), присвойте ей имя `WelcomeDotNetCore` и нажмите кнопку Create (Создать). Выберите эту папку и нажмите кнопку Open (Открыть) или клавишу Enter.

В Visual Studio Code выполните команду View ▸ Integrated Terminal (Вид ▸ Интегрированный терминал) или нажмите сочетание клавиш Ctrl+`.

В области TERMINAL (Терминал) введите следующую команду:

```
dotnet new console
```

Вы увидите, что инструмент `dotnet` создаст проект нового консольного приложения в текущем каталоге, а на панели EXPLORER (Проводник) появятся два новых файла (рис. 1.33).

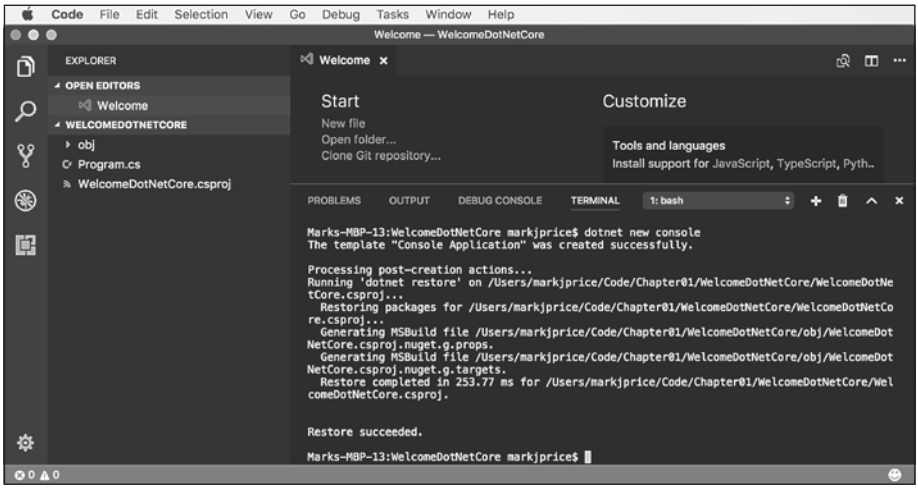


Рис. 1.33

На панели EXPLORER (Проводник) щелкните на файле `Program.cs`, чтобы открыть его в окне редактора.

Если появится сообщение о требуемых ресурсах и нарушенных зависимостях, то нажмите кнопки Yes (Да) и Restore (Восстановить) (рис. 1.34).

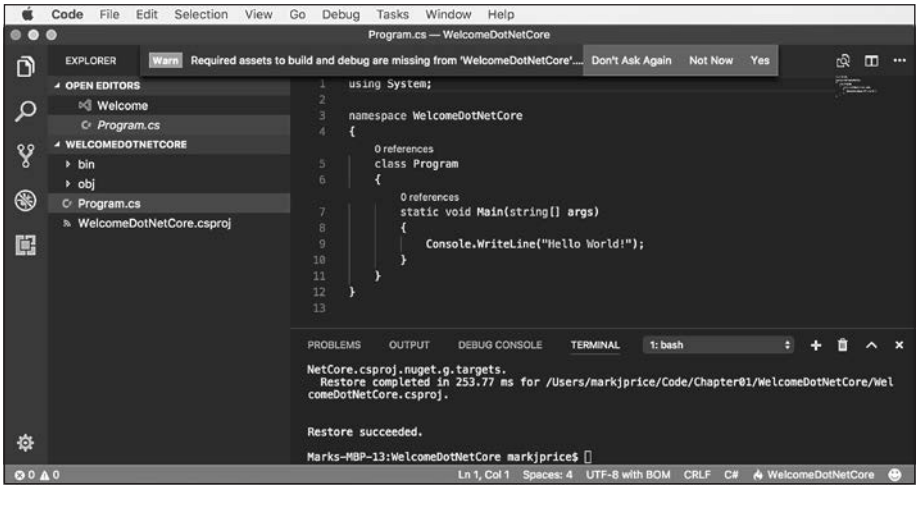


Рис. 1.34

Измените текст, который должен выводиться в окне консоли, на значение `Welcome, .NET Core!`.

В меню File (Файл) выберите пункт Auto Save (Автосохранение). Так вы избавите себя от сообщений, досаждающих напоминанием о необходимости сохранять проект перед каждой сборкой!

Компиляция кода в Visual Studio Code

Выполните команду View ▸ Integrated Terminal (Вид ▸ Интегрированный терминал) или нажмите сочетание клавиш Ctrl+`, а затем введите следующую команду в области TERMINAL (Терминал):

```
dotnet run
```

В области TERMINAL (Терминал) вы увидите вывод с результатом запуска вашего приложения.

Автоматическое форматирование кода

В Visual Studio Code выполните команду File ▸ Open (Файл ▸ Открыть) или нажмите сочетание клавиш Cmd+O, а затем откройте каталог `Chapter01`.

На панели EXPLORER (Проводник) разверните пункт HelloCS и щелкните на файле `Program.cs`.

Если появится сообщение о требуемых ресурсах, то щелкните на ссылке Yes (Да).

В Visual Studio Code в области кода щелкните правой кнопкой мыши и в контекстном меню выберите пункт Format Document (Форматировать документ) или нажмите сочетание клавиш Alt+Shift+F (рис. 1.35).

Visual Studio Code по количеству функций быстро нагоняет среду Visual Studio 2017 для Windows.

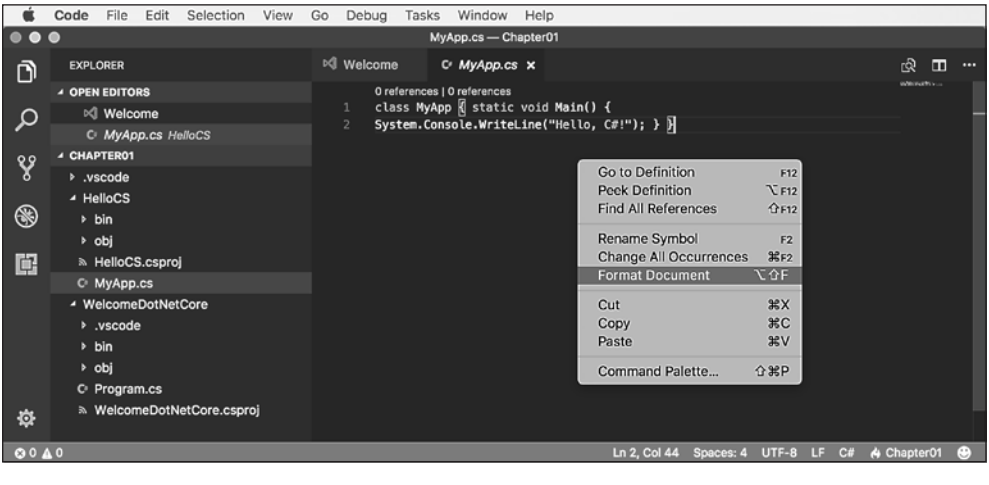


Рис. 1.35

Написание и компиляция кода с помощью Microsoft Visual Studio для Mac

Запустите Visual Studio для Mac и выполните команду File ▸ New Solution (Файл ▸ Новое решение).

В списке слева в разделе .NET Core выберите пункт App (Приложение).

В списке шаблонов проекта в центре выберите пункт Console Application (Консольное приложение) и нажмите кнопку Next (Далее) (рис. 1.36).

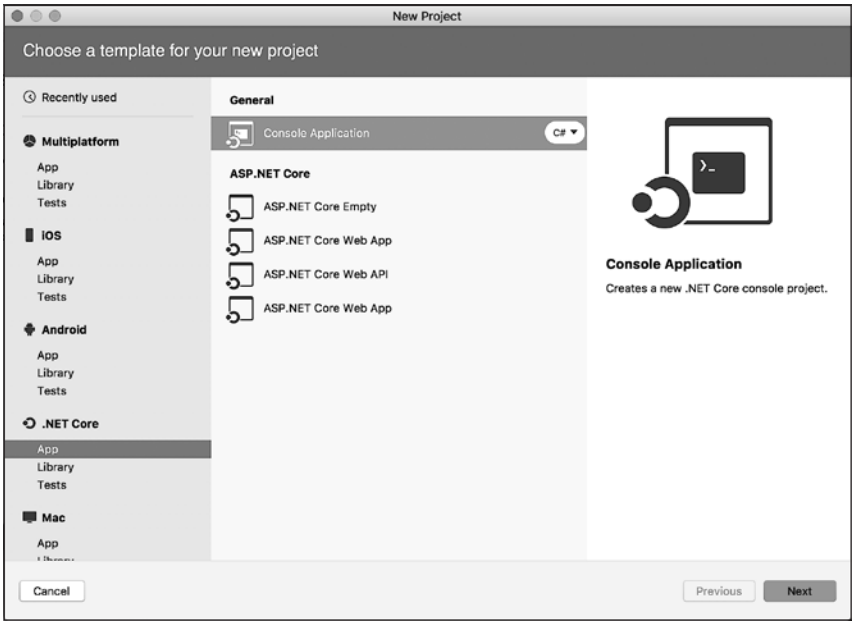


Рис. 1.36

На шаге Configure your new Console Application (Настройка нового консольного приложения) выберите целевую платформу .NET Core 2.0 и нажмите кнопку Next (Далее).

На следующем шаге Configure your new Console Application (Настройка нового консольного приложения) введите в поле Project Name (Имя проекта) значение `WelcomeDotNetCoreMac`, в поле Solution Name (Имя решения) — значение `Chapter01`, в поле Location (Расположение) укажите путь к папке `Code` и нажмите кнопку Create (Создать) (рис. 1.37).

Измените текст, который выводится в консоли, — `Welcome, .NET Core on the Mac!`.

В окне Visual Studio для Mac выполните команду Run ▸ Start Without Debugging (Запуск ▸ Запуск без отладки) или нажмите сочетание клавиш Cmd+Option+Enter.

В окне консоли вы увидите результат запуска своего приложения.

На панели Solution (Решение) щелкните правой кнопкой мыши на пункте Chapter01 и в контекстном меню выполните команду Add ▸ Add Existing Project (Добавить ▸ Добавить существующий проект) (рис. 1.38).

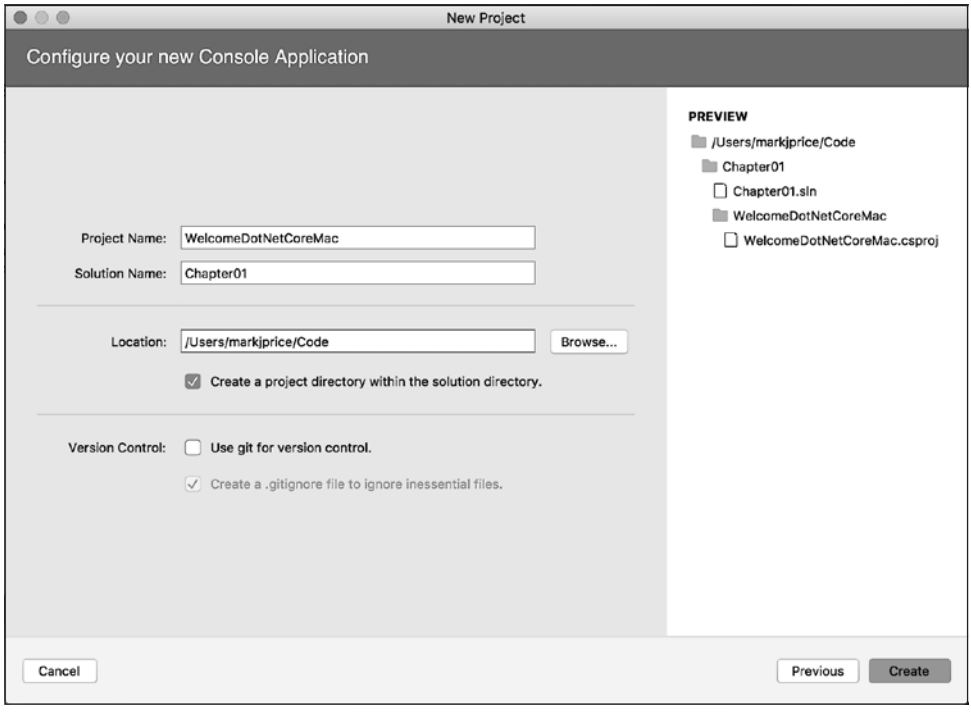


Рис. 1.37

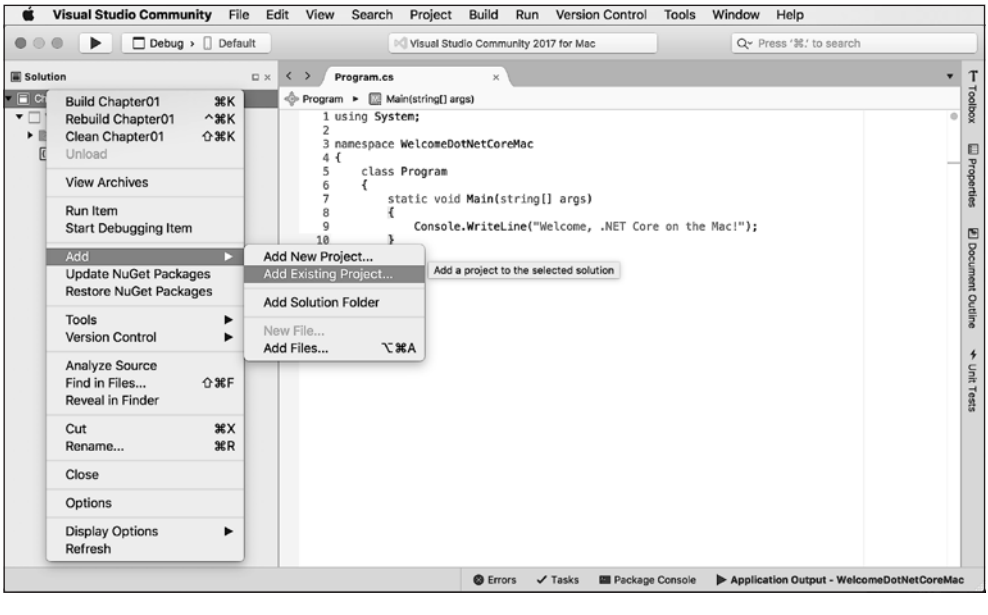


Рис. 1.38

В папке `HelloCS` выберите файл `HelloCS.cs.proj`.

На панели Solution (Решение) щелкните правой кнопкой мыши на пункте HelloCS и выберите пункт Run Item (Запустить элемент) (рис. 1.39).

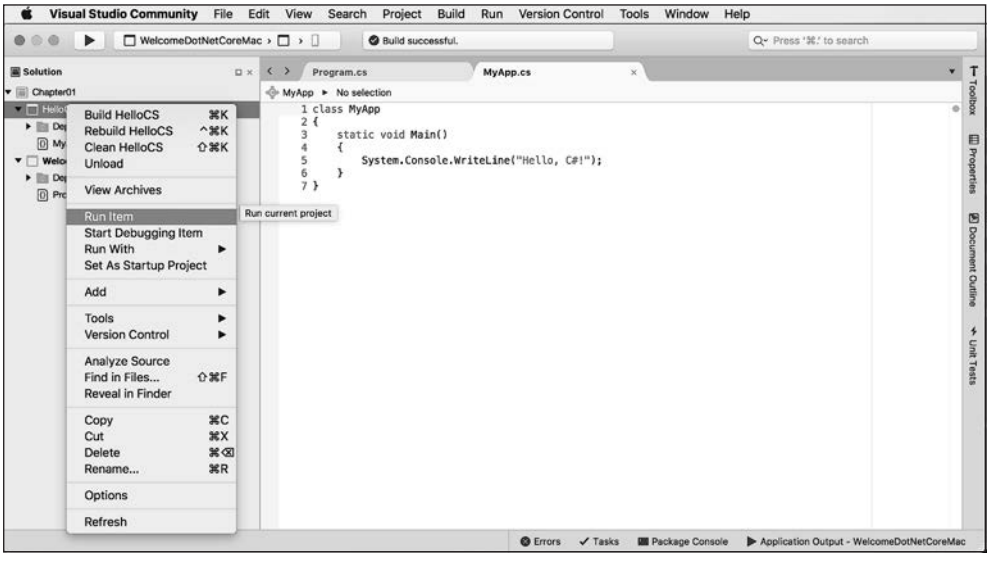


Рис. 1.39

Дальнейшие шаги. Теперь вы знаете, как создавать простые приложения .NET Core в операционных системах Windows и macOS (а в Linux этот процесс еще проще!).

Вы уже способны выполнить примеры из практически всех глав данной книги, используя Visual Studio 2017 в операционной системе Windows, Visual Studio для Mac или Visual Studio Code в среде macOS, Windows или Linux.

Управление исходным кодом с применением GitHub

Git — это широко используемая, распределенная система управления версиями исходного кода. *GitHub* — компания, сайт и настольное приложение, упрощающая взаимодействие с системой Git.

Я использовал GitHub для хранения ответов ко всем упражнениям, приведенным в конце каждой главы этой книги. Получить к ним доступ можно по адресу https://github.com/markjprice/cs7dotnetcore2.

Использование системы Git в Visual Studio 2017

Среда разработки Visual Studio 2017 имеет встроенную поддержку системы Git через сервис GitHub наравне с собственной системой управления версиями исходного кода корпорации Microsoft, называющейся *Visual Studio Team Services (VSTS)*.

Панель Team Explorer

В Visual Studio 2017 выполните команду View ▸ Team Explorer (Вид ▸ Team Explorer), чтобы отобразить эту панель (рис. 1.40).

Рис. 1.40

Хотя я рекомендую создать учетную запись на сайте провайдера системы управления версиями исходного кода, вы можете клонировать репозиторий GitHub и без регистрации аккаунта.

Клонирование репозитория GitHub

На панели Team Explorer раскройте категорию Local Git Repositories (Локальные репозитории Git), нажмите кнопку Clone (Клонировать), а затем введите следующий URL в появившемся поле для клонирования репозитория Git:

```
https://github.com/markjprice/cs7dotnetcore2.git
```

Введите путь к каталогу клонированного репозитория Git:

```
C:\Code\Repos\cs7dotnetcore2
```

Нажмите кнопку Clone (Клонировать) (рис. 1.41).

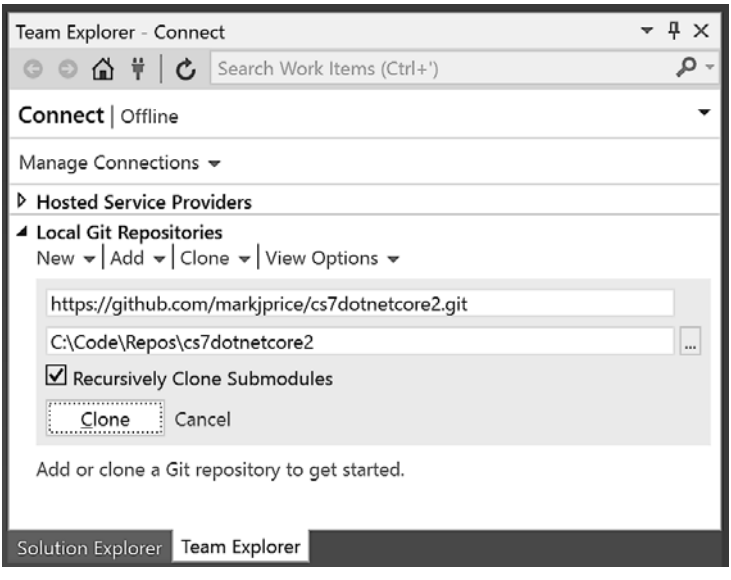


Рис. 1.41

Дождитесь, когда будет создана локальная копия репозитория Git.

Теперь у вас есть локальная копия полных решений всех упражнений из этой книги.

Управление репозиторием GitHub

Дважды щелкните на репозитории `cs7dotnetcore` для отображения содержимого в детализированном режиме.

Вы можете выбирать соответствующие параметры в разделе Project (Проект) для просмотра запросов на включение, выдач и других аспектов репозитория.

Двойной щелчок на записи в категории Solutions (Решения) откроет ее на панели Solution Explorer (Обозреватель решений).

Использование системы Git в Visual Studio Code

Среда разработки Visual Studio Code также поддерживает систему Git, но сначала в операционной системе нужно установить приложение Git версии 2.0 или более позднее, иначе взаимодействие с системой не получится. Скачать дистрибутив приложения Git можно на странице https://git-scm.com/ ownload.

Если вы предпочитаете применять графический пользовательский интерфейс, то можете скачать настольное приложение GitHub по адресу https://desktop.github.com.

Настройка системы Git через командную строку

Запустите приложение Terminal (Терминал) и введите следующую команду для проверки конфигурации:

```
git config --list
```

В выводе должны содержаться ваши имя и адрес электронной почты, поскольку эти данные используются с каждым вашим действием в системе:

```
...other configuration...
user.name=Mark J. Price
user.email=markjprice@gmail.com
```

Если имя и адрес электронной почты не указаны, то введите такие команды, подставив собственные данные:

```
git config --global user.name "Mark J. Price"
git config --global user.email markjprice@gmail.com
```

Кроме того, вы можете проверить конфигурацию на своем компьютере с помощью следующей команды:

```
git config user.name
```

Управление системой Git в Visual Studio Code

Запустите Visual Studio Code.

Выполните команду View ▸ Integrated Terminal (Вид ▸ Интегрированный терминал) или нажмите сочетание клавиш Ctrl+` и введите следующие команды:

```
mkdir Repos
cd Repos
git clone https://github.com/markjprice/cs7dotnetcore2.git
```

Через пару минут на жестком диске будет создана локальная копия полных решений всех упражнений из этой книги (рис. 1.42).

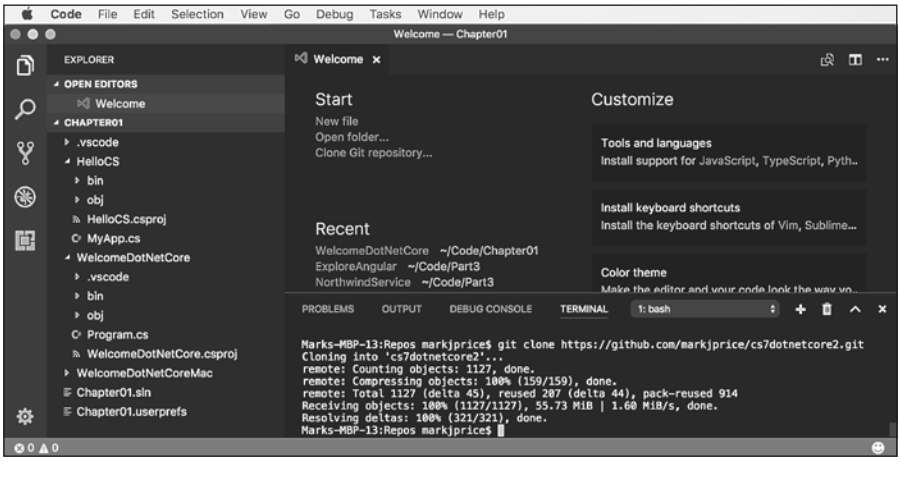


Рис. 1.42

Дополнительную информацию об управлении версиями исходного кода в Visual Studio Code вы найдете на сайте https://code.visualstudio.com/Docs/editor/versioncontrol.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенное упражнение и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

1. Почему на платформе .NET Core для разработки приложений программисты могут использовать разные языки, например C# и F#?
2. Какие команды нужно ввести в окне консоли для сборки и запуска исходного кода на языке C#?
3. Какое сочетание клавиш разработчики применяют в режиме Visual C# для сохранения, компиляции и запуска приложения без отладки?
4. С помощью какого сочетания клавиш в Visual Studio Code открывается панель Integrated Terminal (Интегрированный терминал)?
5. Среда разработки Visual Studio 2017 лучше, чем Visual Studio Code?
6. Платформа .NET Core эффективнее .NET Framework?
7. Чем .NET Native отличается от .NET Core?
8. Что такое .NET Standard и почему эта технология так важна?
9. В чем заключается разница между Git и GitHub?
10. Как называется метод точки входа консольного приложения .NET и как его объявить?

Упражнение 1.1

Вам не нужно устанавливать среду разработки Visual Studio 2017, Visual Studio для Mac или Visual Studio Code, чтобы практиковаться в кодировании на языке C#. Посетите один из этих сайтов и начните кодировать:

- .NET Fiddle: https://dotnetfiddle.net/;
- Cloud9: https://c9.io/web/sign-up/free.

Дополнительные ресурсы

- Знакомство с .NET Core: http://dotnet.github.io.
- Инструменты интерфейса командной строки .NET Core: https://github.com/dotnet/cli.
- CoreCLR, общеязыковая исполняющая среда .NET Core: https://github.com/dotnet/coreclr/.

- Путеводитель по .NET Core: https://github.com/dotnet/core/blob/master/roadmap.md.
- Часто задаваемые вопросы о .NET Standard: https://github.com/dotnet/standard/blob/master/docs/faq.md.
- Документация по Visual Studio: https://docs.microsoft.com/en-us/visualstudio/.
- Блог, посвященный Visual Studio: https://blogs.msdn.microsoft.com/visualstudio/.
- Git и сервисы Team: https://www.visualstudio.com/en-us/docs/git/overview.
- Расширение GitHub для Visual Studio: https://visualstudio.github.com/.

Резюме

В этой главе вы настроили среду разработки, а затем в операционной системе Windows воспользовались программой Command Prompt (Командная строка) и в macOS приложением Terminal (Терминал) для компиляции и запуска консольного приложения. Научились пользоваться средами разработки Visual Studio 2017, Visual Studio для Mac и Visual Studio Code для создания похожего приложения и обсудили различия между платформами .NET Framework, .NET Core, .NET Standard и .NET Native.

В следующей главе вы научитесь говорить на языке C#.

Часть I
C# 7.1

Эта часть книги посвящена языку C# — его грамматике и лексике, которые вы будете ежедневно использовать при написании исходного кода своих приложений.

Языки программирования очень похожи на человеческий язык, за исключением того, что в языках программирования мы можем придумывать собственные слова, как доктор Сьюз!

В книге «Если я стану зоопарком», написанной доктором Сьюзом в 1950 году, есть такие стихи:

После охоты в джунглях Хиппо-но-Хангуса,

Я привезу стаю диких Биппо-но-Бангусов!

Биппо-но-Бангусы, что из Хиппо-но-Хангуса, —

Лучше чем те, что живут в Диппо-но-Дангусе,

И умнее чем те, что вне Ниппо-но-Нангуса.

В данной книге описана версия C# 7.1. В корпорации Microsoft есть следующие планы на будущие версии C# (табл. I.1).

Таблица I.1

Версия	Новые возможности
C# 7.2	Модификатор ref readonly, непреобразуемые типы, строгие имена, внутренние указатели, неконечные именованные аргументы, модификатор private protected, нижнее подчеркивание в числовых литералах
C# 7.3	Диапазоны с указанием двух точек, например, 1..10
C# 8.0	Методы интерфейсов по умолчанию, ссылочные типы, принимающие нулевое значение

Подробную информацию можно получить, перейдя по ссылке https://github.com/dotnet/roslyn/blob/master/docs/Language%20Feature%20Status.md.

Изучить C# лучше на примере создания нескольких простых приложений. Во избежание перегрузки слишком большим количеством информации в этой части книги вы будете работать с приложениями простейшего типа — консольными.

В главах части I раскрываются следующие темы.

1. Грамматика и терминология языка C#.
2. Инструкции ветвления и циклов языка C# и способы преобразования типов в нем.
3. Многократное применение кода с помощью функций C#, а также отладка и тестирование кода на данном языке.
4. Использование объектно-ориентированных свойств языка C#.
5. Продвинутые объектно-ориентированные свойства языка C#.

2 Говорим на языке C#

В этой главе рассказывается об основах языка программирования C#. Вы узнаете, как составлять инструкции, следуя правилам грамматики и применяя ряд ключевых слов C#, которыми будете пользоваться ежедневно. Вы также научитесь временно сохранять данные в памяти вашего компьютера и выполнять с ними простые операции.

В этой главе:

- основы языка C#;
- объявление переменных;
- конструирование консольных приложений;
- работа с переменными.

Основы языка C#

Начнем с изучения основ грамматики и терминологии языка C#. В этой главе вы создадите несколько консольных приложений, каждое из которых будет демонстрировать возможности данного языка.

Для каждого консольного приложения требуется создать проект. Часто разработчики открывают несколько проектов одновременно. Вы тоже можете так поступить, добавив проекты в решение.

Управлять своими проектами в Visual Studio 2017 можно, поместив их все в одно решение. Эта среда позволяет открывать только одно решение в один момент времени, но каждое из них может содержать несколько проектов. С помощью проекта можно построить консольное приложение, настольное приложение Windows, веб-приложение и др.

Для управления проектами в Visual Studio Code, в которой не поддерживаются решения, понадобится вручную создать каталог-контейнер `Chapter02`. Если вы работаете с Visual Studio Code, то пропустите следующий подраздел и переходите к подразделу «Visual Studio Code в среде macOS, Linux или Windows».

Visual Studio 2017

Запустите Microsoft Visual Studio 2017. Нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте категорию Other Project Types (Другие типы проектов) и выберите пункт Visual Studio Solutions (Решения Visual Studio). В центре диалогового окна выберите пункт Blank Solution (Пустое решение), присвойте ему имя `Chapter02`, укажите расположение `C:\Code`, а затем нажмите кнопку OK (рис. 2.1).

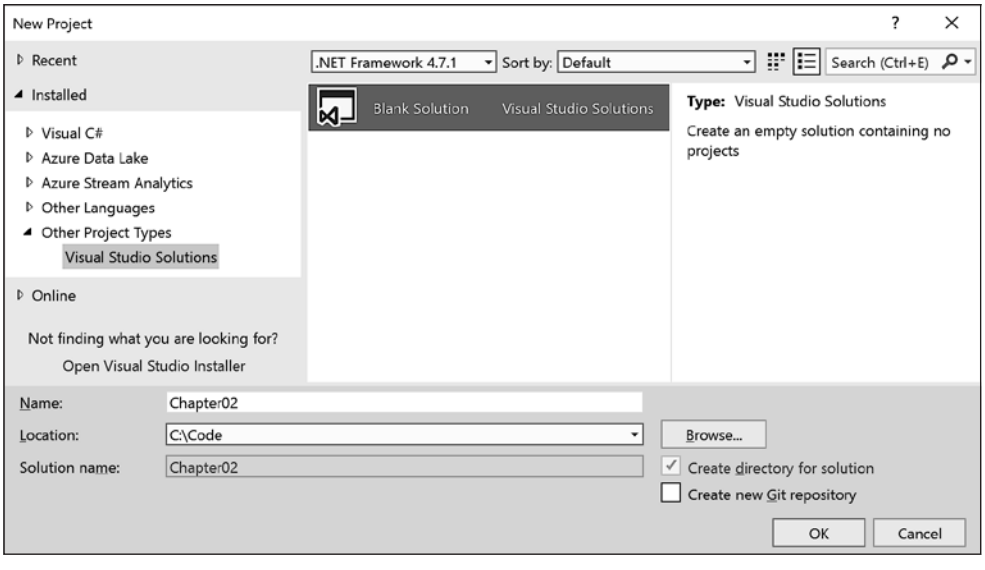


Рис. 2.1

Если вы запустите File Explorer (Обозреватель файлов), то увидите, что в Visual Studio был создан каталог `Chapter02` с решением `Chapter02.sln` внутри него (рис. 2.2).

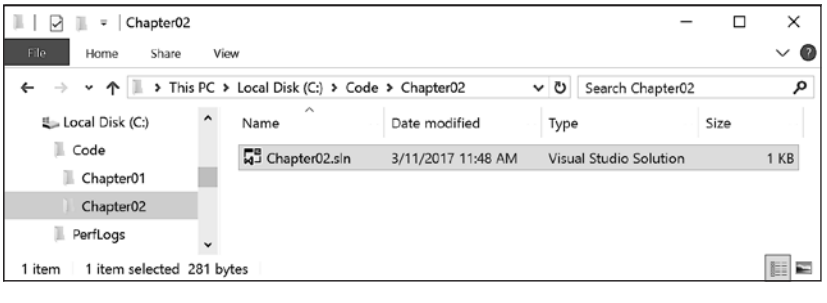


Рис. 2.2

В Visual Studio выполните команду File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект) (рис. 2.3). Так вы добавите новый проект в созданное решение.

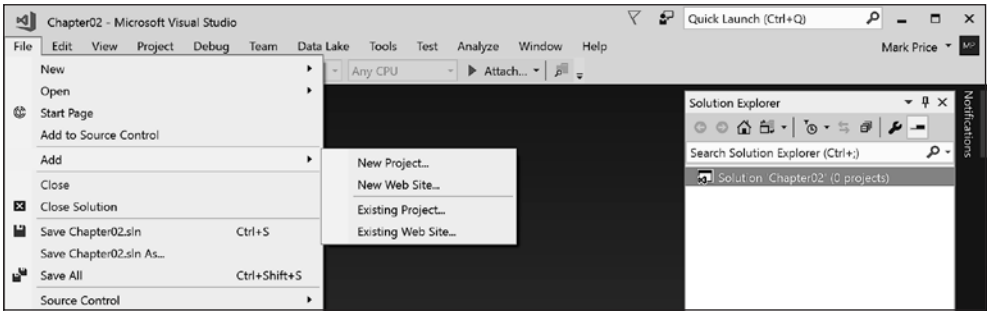


Рис. 2.3

В диалоговом окне Add New Project (Добавить новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Core. В центре диалогового окна выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)), присвойте ему имя `Basics`, а затем нажмите кнопку OK (рис. 2.4).

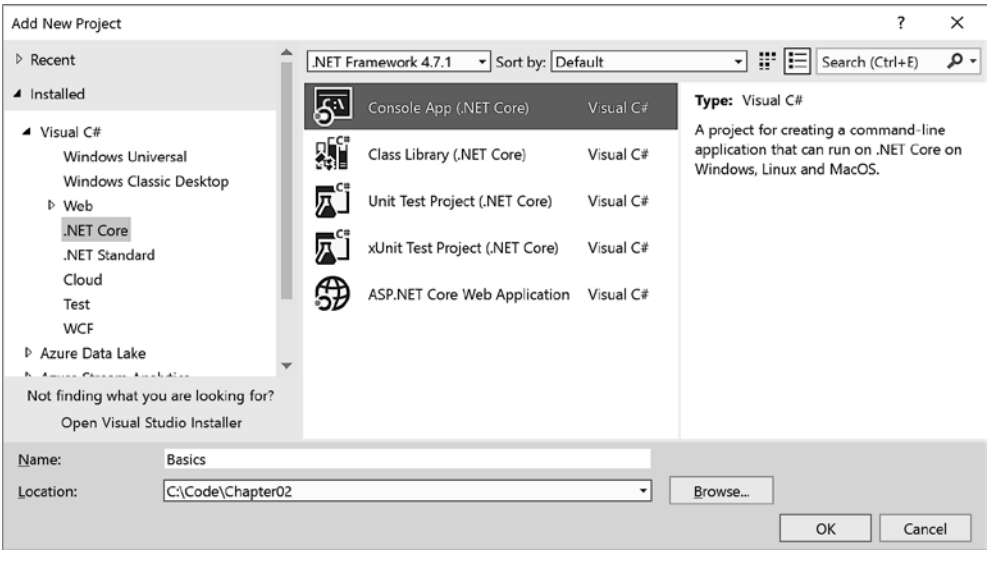


Рис. 2.4

Если вы запустите File Explorer (Обозреватель файлов), то увидите: Visual Studio создала каталог с несколькими подкаталогами и файлами. Сейчас вам не нужно знать, что это за файлы. Код, который вы напишете, будет сохраняться в файл `Program.cs` (рис. 2.5).

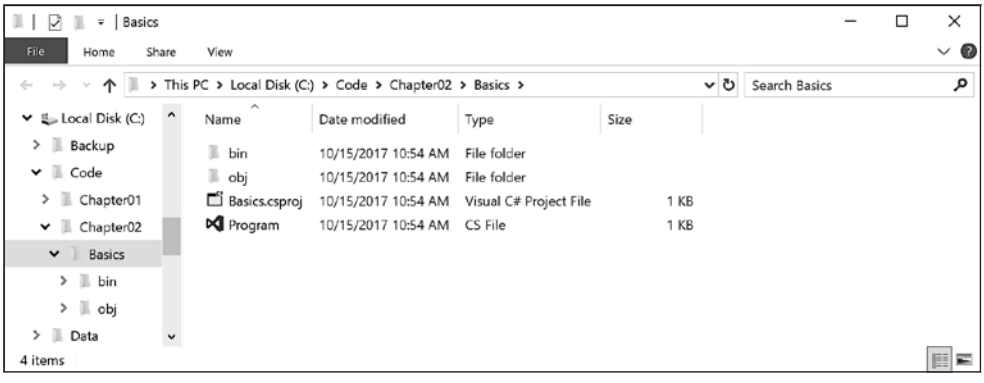


Рис. 2.5

В Visual Studio на панели Solution Explorer (Обозреватель решений) отображаются те же файлы, что и на предыдущем снимке экрана файловой системы.

Некоторые каталоги и файлы, к примеру каталог `bin`, на панели Solution Explorer (Обозреватель решений) по умолчанию скрыты. Отобразить их поможет кнопка Show All Files (Показать все файлы) в верхней части панели. Нажимайте ее для отображения/скрытия каталогов и файлов (рис. 2.6).

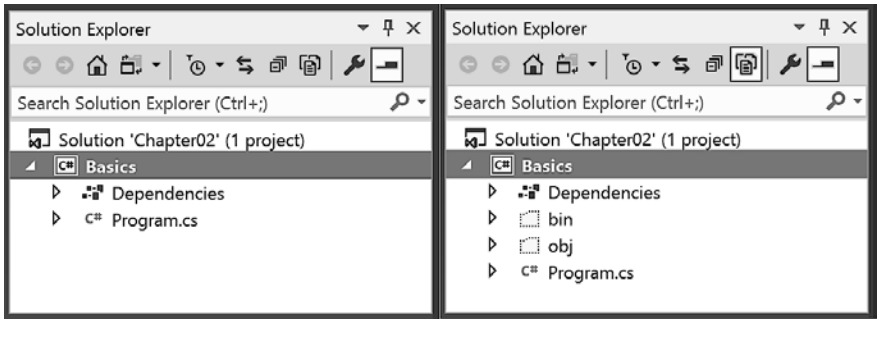


Рис. 2.6

Visual Studio Code в среде macOS, Linux или Windows

Если вы выполнили упражнения из главы 1, то должны были создать папку `Code` в своем пользовательском каталоге. В противном случае создайте ее, а в ней — подкаталог `Chapter02`, содержащий еще один подкаталог — `Basics` (рис. 2.7).

Запустите Visual Studio Code и откройте каталог `/Chapter02/Basics/`.

Выполните команду View ▸ Integrated Terminal (Вид ▸ Интегрированный терминал) и в области TERMINAL (Терминал) введите следующую команду:

```
dotnet new console
```

На панели EXPLORER (Проводник) щелкните на файле `Program.cs`, а затем на ссылке Yes (Да) для восстановления зависимостей (рис. 2.8).

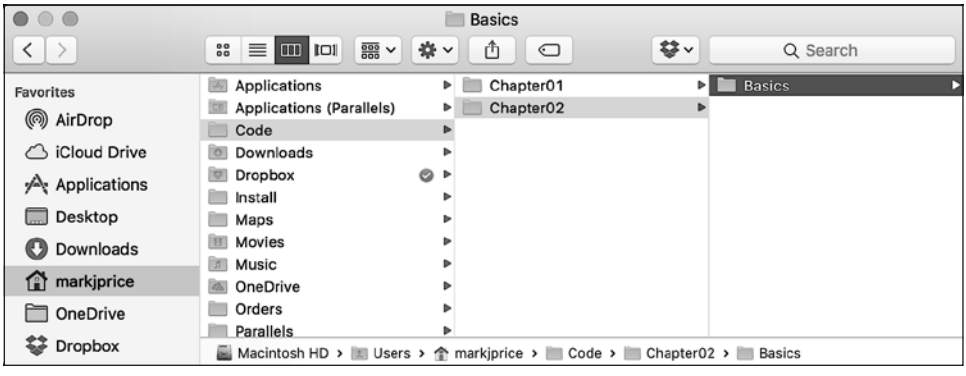


Рис. 2.7

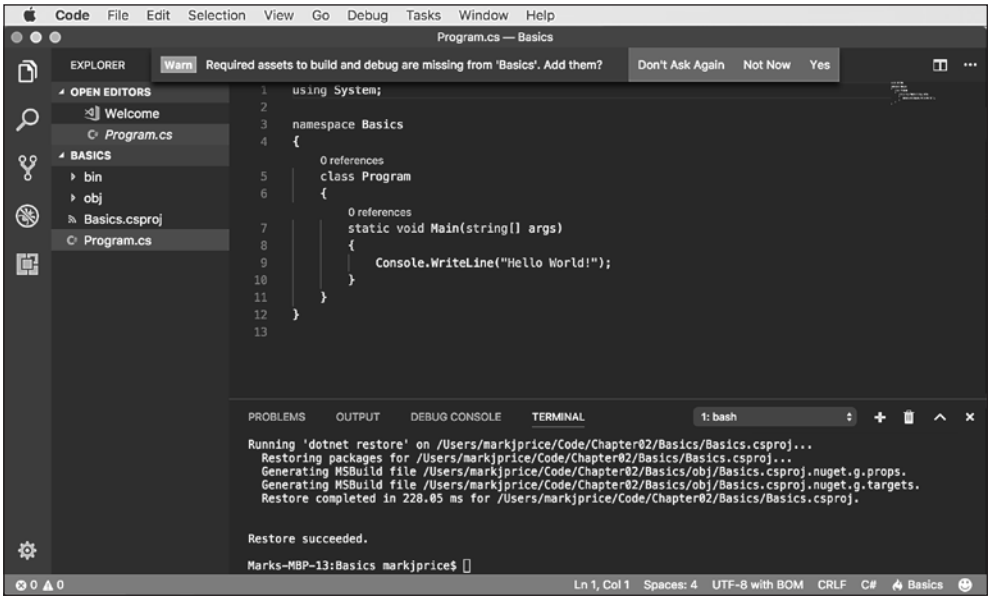


Рис. 2.8

Грамматика языка C#

К грамматике данного языка относятся *инструкции* и *блоки*. Для документирования кода используются *комментарии*.

Инструкции

В русском языке мы обозначаем конец повествовательного предложения точкой. Предложение может состоять из разного количества слов. Порядок слов в нем тоже относится к правилам грамматики. К примеру, по-русски мы говорим: черный кот.

Прилагательное «черный» предшествует существительному «кот». Во французском языке порядок иной; прилагательное указывается после существительного: le chat noir. Порядок имеет значение.

Язык C# определяет конец *инструкции* по символу точки с запятой. При этом инструкция может состоять из нескольких *переменных* и *выражений*.

В инструкции, показанной ниже, `FullName` — *переменная*, а `FirstName + LastName` — *выражение*:

```
var FullName = FirstName + LastName;
```

Выражение состоит из *операнда* (`FirstName`), *оператора* (`+`) и второго операнда (`LastName`). Порядок имеет значение.

Комментарии

Чтобы описать предназначение кода, можно добавлять однострочные примечания, предваряя их двумя символами косой черты (слеша): `//`.

Компилятор игнорирует любой текст после символов `//` и до конца строки, например:

```
var TotalPrice = Cost + Tax; // Налог составляет 20 % от цены
```

Visual Studio 2017 и Visual Studio Code позволяют добавлять и удалять комментарии (два символа слеша) в начале выбранной строки путем нажатия сочетания клавиш Ctrl+K+C и Ctrl+K+U. В операционной системе macOS вместо клавиши Ctrl используйте Cmd.

Для оформления многострочного комментария используйте символы `/*` в начале комментария и `*/` в его конце, как показано в коде ниже:

```
/*
Это многострочный
комментарий.
*/
```

Блоки

В русском языке мы обозначаем абзацы, начиная последующий текст с новой строки. В языке C# *блоки* кода заключаются в фигурные скобки: `{ }`. Каждый блок начинается с объявления, описывающего то, что мы определяем. Например, блок может определять *пространство имен*, *класс*, *метод* или *инструкцию*. Мы разберемся со всем этим позднее.

В текущем проекте обратите внимание на грамматику языка C#, написанную с помощью шаблона Visual Studio или инструмента `dotnet`.

В следующем примере я добавлю несколько комментариев для описания кода:

```
using System; // точка с запятой указывает на конец инструкции

class Program
{
```

```
  static void Main(string[] args)
  { // начало блока
    Console.WriteLine("Hello World!"); // инструкция
  } // конец блока
}
```

Словарь языка C#

Словарь данного языка состоит из ключевых слов, символов и типов.

В этой главе вам встретятся некоторые из 79 предопределенных, зарезервированных ключевых слов, в том числе `using`, `namespace`, `class`, `static`, `int`, `string`, `double`, `bool`, `var`, `if`, `switch`, `break`, `while`, `do`, `for` и `foreach`.

Среда разработки Visual Studio 2017 окрашивает ключевые слова C# в синий цвет, чтобы их можно было легко заметить. На рис. 2.9 слова `using`, `namespace`, `class`, `static`, `void` и `string` относятся к терминологии C#:

Рис. 2.9

Тот же код в Visual Studio Code показан на рис. 2.10.

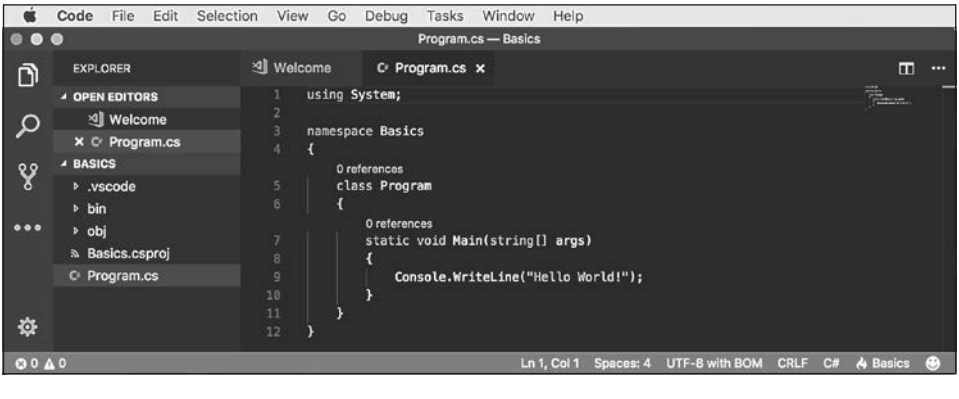


Рис. 2.10

И Visual Studio 2017, и Visual Studio Code позволяют изменить цветовую схему. В Visual Studio 2017 выполните команду Tools ▸ Options (Сервис ▸ Параметры) и выберите категорию Environment ▸ Fonts and Colors (Окружение ▸ Шрифты и цвета). В Visual Studio Code выполните команду File ▸ Preferences ▸ Color Theme (Файл ▸ Настройки ▸ Цветовая схема).

Существует еще 25 контекстных ключевых слов, которые имеют специальное значение только в определенных условиях. Итого получается, что в языке C# используются 104 ключевых слова.

Английский язык насчитывает свыше 250 000 словарных слов. Как же язык C# может обходиться только 104 ключевыми словами? Почему C# очень трудно изучать, хотя он содержит так мало слов?

Одно из ключевых отличий человеческого языка и языка программирования состоит в том, что разработчикам нужно определять новые «слова» с новыми значениями.

Помимо 104 ключевых слов, использующихся в языке C#, эта книга научит вас применять некоторые из сотен тысяч «слов», определяемых программистами. Вы также научитесь определять собственные «слова».

Программисты со всего мира должны изучать английский язык, поскольку большинство языков программирования используют английские слова (например, namespace и class). Существуют языки программирования, основанные на других человеческих языках, к примеру на арабском, но это скорее исключение. Посмотреть видеоролик, демонстрирующий арабский язык программирования, можно на сайте YouTube, перейдя по ссылке https://www.youtube.com/watch?v=77KAHPZUR8g.

Помощь в написании корректного кода

Простые текстовые редакторы, такие как приложение Notepad (Блокнот), не помогут исправить опечатки (рис. 2.11).

Не исправят они ошибки и в коде C# (рис. 2.12).

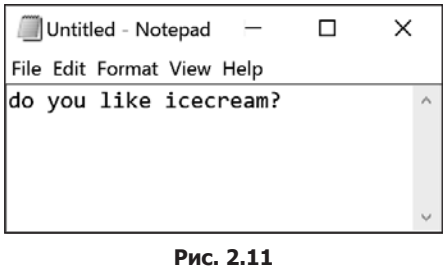


Рис. 2.11

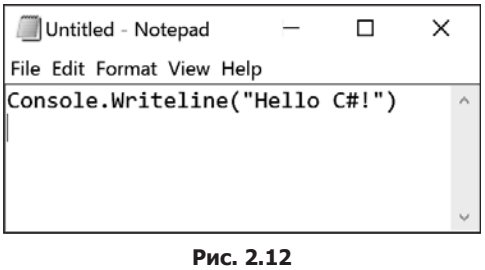


Рис. 2.12

Microsoft Word помогает писать без ошибок, подчеркивая орфографические ошибки красной волнистой чертой (правильно: ice cream), а грамматические —

зеленой (предложения должны начинаться с прописной буквы в первом слове) (рис. 2.13).

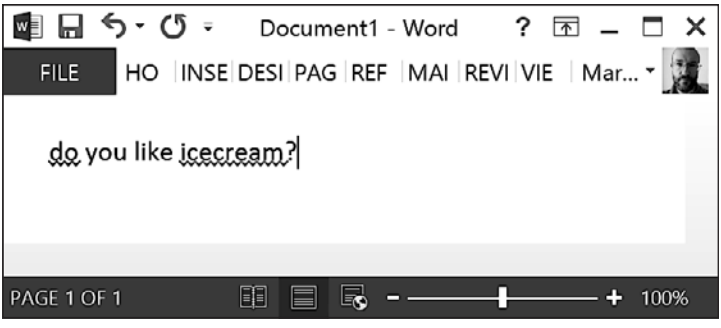


Рис. 2.13

Аналогичным образом Visual Studio 2017 и Visual Studio Code помогают писать код на языке C#, подчеркивая орфографические ошибки (буква `L` в имени метода `WriteLine` должна быть прописной) и грамматические (инструкции нужно завершать символом точки с запятой).

Среда разработки Visual Studio 2017 постоянно отслеживает код, который вы набираете, и предоставляет обратную связь, выделяя проблемные участки кода цветными волнистыми линиями и отображая ошибки на панели Error List (Список ошибок) (рис. 2.14). В Visual Studio Code эта панель известна под названием PROBLEMS (Проблемы).

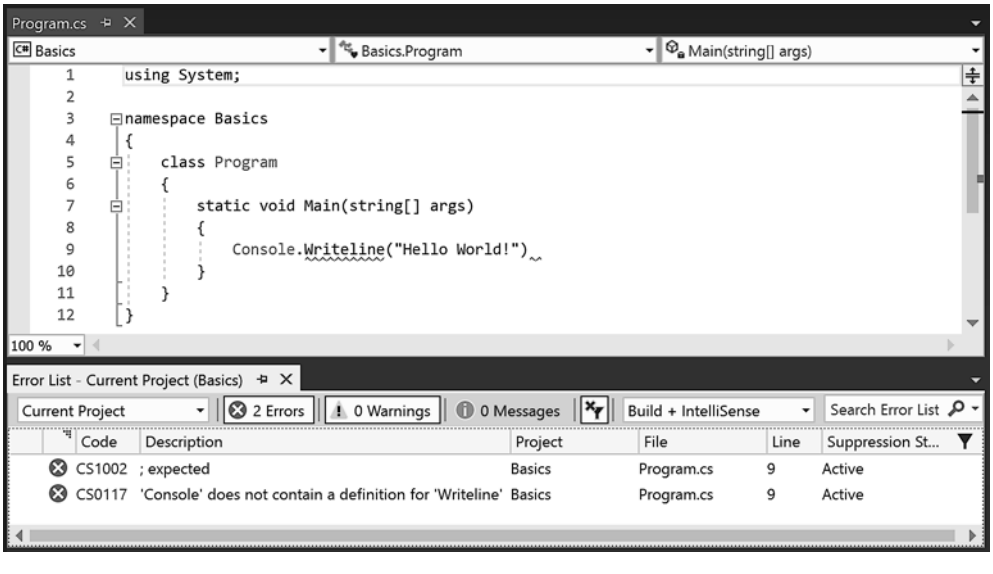


Рис. 2.14

В Visual Studio 2017 можно выполнить комплексную проверку набранного кода, выбрав команду Build ▸ Build Solution (Сборка ▸ Собрать решение) или нажав клавишу F6. В Visual Studio Code введите следующую команду в Integrated Terminal (Интегрированный терминал): dotnet build.

В Visual Studio Code для отслеживания ошибок используется аналогичная панель под названием PROBLEMS (Проблемы) (рис. 2.15).

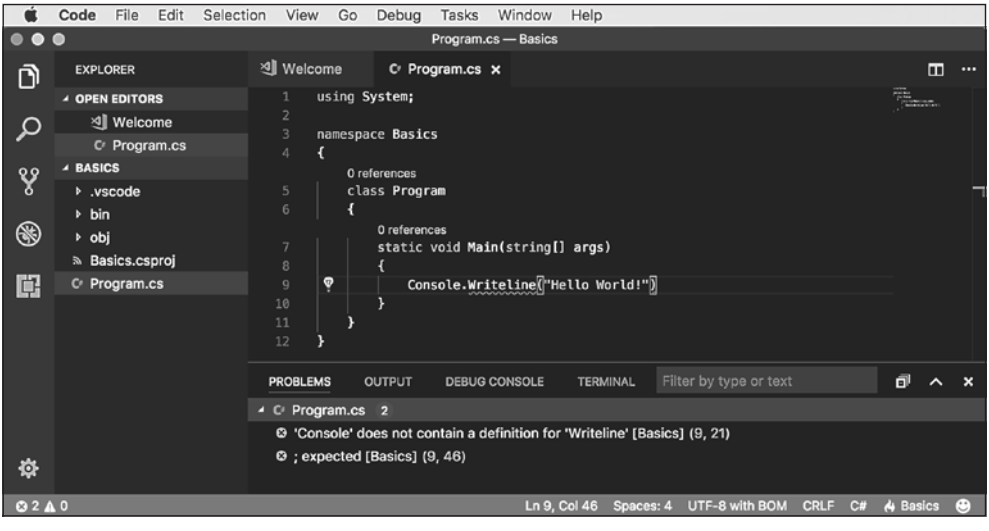


Рис. 2.15

Глаголы = методы

В русском языке глаголы используются для описания действия. В C# действия называются *методами*. Существуют буквально сотни тысяч методов, доступных в языке C#.

В русском языке глаголы меняют свою форму в зависимости от времени: происходило, происходит или будет происходить действие. К примеру, Амир *попрыгал* в прошлом, Берт *прыгает* в настоящем, они *прыгали* в прошлом и Чарли *будет прыгать* в будущем.

В языке C# вызов или выполнение методов, подобных `WriteLine`, различны в зависимости от специфики действия. Это так называемая перегрузка, которую мы подробнее изучим в главе 5. Рассмотрим следующий пример:

```
// выводит возврат каретки
Console.WriteLine();
// выводит приветствие и возврат каретки
Console.WriteLine("Hello Ahmed");
// выводит отформатированное число и дату
Console.WriteLine("Temperature on {0:D} is {1}°C.", DateTime.Today, 23.4);
```

Другая аналогия заключается в том, что некоторые слова пишутся одинаково, но имеют разные значения в зависимости от контекста.

Существительные = типы данных, поля и переменные

В русском языке существительные — названия предметов или живых существ. К примеру, Бобик — это название (имя) собаки. Словом «собака» называется тип живого существа с именем Бобик. Скомандовав Бобику принести мяч, мы используем его имя и название предмета, который нужно принести.

В языке C# используются эквиваленты существительных — *типы данных* (чаще называемые просто типами), *поля* и *переменные*. *Для* языка C# доступны десятки тысяч типов. Обратите внимание: я сказал не «*В* языке C# доступны десятки тысяч типов».

Разница едва уловимая, но весьма важная. C# (язык) содержит лишь несколько ключевых слов для типов данных, таких как `string` и `int`. Строго говоря, C# не определяет какие-либо типы. Ключевые слова типа `string` определяют типы как *псевдонимы*. Последние представляют типы на платформе, на которой запускается C#.

C# не способен работать независимо. Это язык приложений, запускаемых на разных платформах .NET. Теоретически можно написать компилятор C# под другую платформу, с другими базовыми типами. На практике платформа для C# — одна из платформ .NET. Это платформа .NET, предоставляющая десятки тысяч типов для C#. Последние содержат `System.Int32`, к которым относится ключевое слово-псевдоним `int` языка C#, и более сложные типы, такие как `System.Xml.Linq.XDocument`.

Обратите внимание: термин «тип» часто путают с «классом». Существует игра «20 вопросов», в которой первым делом спрашивается категория загаданного предмета: «животное», «растение» или «минерал»? В языке C# каждый тип может быть отнесен к одной из категорий: `class` (класс), `struct` (структура), `enum` (перечисление), `interface` (интерфейс) или `delegate` (делегат). В C# ключевое слово `string` представляет собой `class` (класс), но `int` — это `struct` (структура). Так что лучше использовать термин «тип» для обозначения их обоих.

Подсчет количества типов и методов

В нашем простом консольном приложении напишем код, позволяющий подсчитать количество типов и методов, доступных в языке C#.

Не волнуйтесь, если не понимаете, как работает данный код. В нем используется техника под названием *«отражение»*, описание которой выходит за рамки этой книги.

Начнем с добавления следующих инструкций в начало файла `Program.cs`:

```
using System.Linq;
using System.Reflection;
```

В методе Main удалите инструкцию, выводящую текст Hello World!, и замените ее кодом, показанным ниже:

```
// перебор сборок, на которые ссылается приложение
foreach (var r in Assembly.GetEntryAssembly()
  .GetReferencedAssemblies())
{
  // загрузка сборки для чтения данных
  var a = Assembly.Load(new AssemblyName(r.FullName));
  // объявление переменной для подсчета методов
  int methodCount = 0;
  // перебор всех типов в сборке
  foreach (var t in a.DefinedTypes)
  {
    // добавление количества методов
    methodCount += t.GetMethods().Count();
  }
  // вывод количества типов и их методов
  Console.WriteLine($"{a.DefinedTypes.Count():N0} types " +
   $"with {methodCount:N0} methods in {r.Name} assembly.");
}
```

Сборка и запуск в Visual Studio 2017

В меню Debug (Отладка) выберите пункт Start Without Debugging (Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5 для сохранения, компиляции и запуска вашего приложения без отладчика.

Вы увидите результат вывода, показанный ниже. В нем отобразится актуальное количество типов и методов, доступных вашему простейшему приложению, запущенному в операционной системе Windows:

```
25 types with 290 methods in System.Runtime assembly.
95 types with 1,029 methods in System.Linq assembly.
43 types with 652 methods in System.Console assembly.
```

Сборка и запуск в Visual Studio Code

На панели Integrated Terminal (Интегрированный терминал) введите следующую команду:

```
dotnet run
```

Вы увидите результат вывода, показанный ниже, в котором отобразится актуальное количество типов и методов, доступных вашему простейшему приложению, запущенному в операционной системе macOS:

```
25 types with 290 methods in System.Runtime assembly.
95 types with 1,029 methods in System.Linq assembly.
53 types with 691 methods in System.Console assembly.
```

Отображенное количество типов и методов может отличаться в зависимости от используемой операционной системы. Например, хотя сборки System.Runtime и System.Linq включают одинаковое количество типов и методов в Windows и macOS, System.Console в macOS содержит дополнительные десять типов и 39 методов.

Добавление типов в Visual Studio 2017 и Visual Studio Code

Добавьте следующие инструкции в начало метода `Main`. Если объявляются переменные, использующие типы в других сборках, то эти сборки загружаются вместе с приложением. Данное обстоятельство позволяет коду видеть все типы и методы в них.

```
static void Main(string[] args)
{
  System.Data.DataSet ds;
  System.Net.Http.HttpClient client;
```

Панель Error List (Список ошибок) в среде разработки Visual Studio 2017 и панель PROBLEMS (Проблемы) в Visual Studio Code Problems отобразят три предупреждения об использовании в коде переменных, которые были объявлены, но не применяются. Вы можете спокойно игнорировать эти предупреждения.

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+F5.

В Visual Studio Code введите команду `dotnet run` на панели Integrated Terminal (Интегрированный терминал). Просмотрите результат вывода, который должен выглядеть примерно так, как показано ниже:

```
25 types with 290 methods in System.Runtime assembly.
349 types with 6,327 methods in System.Data.Common assembly.
169 types with 1,882 methods in System.Net.Http assembly.
95 types with 1,029 methods in System.Linq assembly.
43 types with 652 methods in System.Console assembly.
```

Надеюсь, теперь вы понимаете, почему изучение языка C# — та еще задача. Так много типов, с множеством методов, причем методы — это только одна категория членов, которую может иметь тип, а программисты постоянно определяют новые члены!

Объявление переменных

Любое приложение занимается обработкой данных. Они поступают, обрабатываются и выводятся.

Обычно данные поступают в программы из файлов, баз данных или пользовательского ввода. Данные могут быть на время помещены в переменные, которые хранятся в памяти работающей программы. Когда последняя завершает работу, данные стираются из памяти. Данные обычно выводятся в файлы и базы или на экран или принтер.

При использовании переменных вы должны учитывать два фактора. Во-первых, как много объема памяти им требуется и, во-вторых, насколько быстро их можно обработать.

Управлять этими характеристиками позволяет выбор определенного типа. Простые распространенные типы, такие как `int` и `double`, можно представить как коробки разного размера. Маленькая занимает меньше памяти, но способна замедляться во время обработки. Некоторые коробки можно сложить в аккуратную стопочку прямо под рукой, а другие — убрать подальше и свалить в большую кучу.

Присвоение имен переменным

Теперь обсудим правила именования переменных, которым рекомендуется следовать (табл. 2.1).

Таблица 2.1

Правило	Примеры	Использование
Верблюжий регистр	cost, orderDetail, dateOfBirth	Локальные переменные и закрытые члены
Стиль Pascal/прописной стиль	Cost, OrderDetail, DateOfBirth	Имена типов и открытые члены

Следуя правилам именования, вы сделаете свой код понятным для других разработчиков (и для себя в будущем!). Изучите соглашения по именованию на сайте https://msdn.microsoft.com/en-us/library/ms229002(v=vs.110).aspx.

Блок кода, показанный ниже, отражает пример объявления и инициализации локальной переменной путем присвоения ей значения. Обратите внимание: можно вывести переменную, используя ключевое слово `nameof`, появившееся в версии C# 6:

```
// присвоение переменной значения, равного 1,88
double heightInMetres = 1.88;
Console.WriteLine($"The variable {nameof(heightInMetres)} has the value
{heightInMetres}.");
```

Сообщение в двойных кавычках в предыдущем фрагменте кода было перенесено на вторую строку, потому что страница книги слишком узкая. При вводе инструкций наподобие этой в окне редактора кода печатайте их в одну строку.

Литеральные значения

Часто используются при присвоении значения переменной. *Литерал* — это нотация, которая представляет собой фиксированное значение. Существуют различные нотации для определения литеральных значений разных типов. В следующих нескольких подразделах вы увидите примеры использования букв для присвоения значений переменным.

Хранение текста

При работе с текстом отдельная буква, например, `A`, сохраняется как тип `char` и присваивается с использованием *одинарных* кавычек, оборачивающих литеральное значение или возвращаемое значение вызова функции:

```
char letter = 'A';
char digit = '1';
char symbol = '$';
char userChoice = GetCharacterFromUser();
```

Если же применяется последовательность символов, скажем, слово `Bob`, то такое значение сохраняется как тип `string` и присваивается с помощью *двойных* кавычек, оборачивающих литеральное значение:

```
string firstName = "Bob";
string lastName = "Smith";
string phoneNumber = "(215) 555-4256";
string address = GetAddressFromDatabase(id: 563);
```

Хранение чисел

Числа — это данные, которые можно использовать для выполнения арифметических операций, например, умножения.

Телефонный номер — не число. Чтобы определиться, какое значение следует присваивать переменной — числовое или нет, спросите себя, нужно ли вам умножать два телефонных номера или задействовать в значении специальные символы, например, так: (414) 555-1234. В этих случаях число представляет собой последовательность символов, поэтому должно храниться как строка.

Числа бывают *натуральными* (скажем, 42) и используются для подсчета, могут быть отрицательными (например, –42). Это *целые числа*.

Еще числа бывают *вещественными* (или *действительными*), например 3,9 (с дробной частью). В информатике они называются числами *одинарной* и *двойной точности с плавающей запятой*.

```
uint naturalNumber = 23;
// unsigned integer означает положительное целое число
int integerNumber = -23;
// integer означает отрицательное или положительное целое число
double realNumber = 2.3;
// double означает число двойной точности с плавающей запятой
```

Наверняка вы знаете, что компьютеры хранят всю информацию в битах. Каждый *бит* может быть равен или 0, или 1. Это так называемая *двоичная* (бинарная) система счисления. Люди же пользуются *десятичной*.

В основе десятичной системы счисления лежит число 10. Ею чаще всего пользуются люди в повседневной жизни, а другие системы распространены в научных, инженерных и компьютерных дисциплинах.

Хранение целых чисел

В табл. 2.2 показано, как компьютеры хранят число 10 в двоичной системе счисления. Обратите внимание на 1 бит в столбцах 8 и 2; *8 + 2 = 10*.

Таблица 2.2

128	64	32	16	8	4	2	1
0	0	0	0	1	0	1	0

Таким образом, десятичное число 10 в двоичной системе счисления можно изобразить как 00001010.

Новые возможности в версии C# 7

Одна из новых возможностей версии C# 7 касается использования символа подчеркивания (_) в качестве *разделителя групп разрядов чисел*, а вторая внедряет поддержку *двоичных литералов*.

Символ подчеркивания можно использовать в любых числовых литералах, включая десятичную, двоичную и шестнадцатеричную систему счисления для удобства восприятия. К примеру, значение в один миллион можно записать в десятичной системе счисления (основание 10) следующим образом: `1_000_000`.

В двоичной системе счисления (основание 2) используются только единицы и нули, и двоичные литералы задаются с помощью префикса `0b`. В шестнадцатеричной (основание 16) применяются цифры от 0 до 9 и буквы A — F, а шестнадцатеричные целочисленные литералы начинаются с префикса `0x`, как показано в коде, приведенном ниже:

```
int decimalNotation = 2_000_000; // два миллиона
int binaryNotation = 0b0001_1110_1000_0100_1000_0000; // два миллиона
int hexadecimalNotation = 0x001E_8480; // два миллиона
Console.WriteLine($"{decimalNotation == binaryNotation}"); // => true
Console.WriteLine($"{decimalNotation == hexadecimalNotation}"); // => true
```

Компьютеры всегда могут точно представлять целые числа (как положительные, так и отрицательные), используя тип `int` или один из его родственных типов, например, `short`.

Хранение вещественных чисел

Компьютеры не всегда могут точно представлять числа с плавающей запятой. С помощью типов `float` и `double` можно задавать вещественные числа одинарной и двойной точности с плавающей запятой.

В табл. 2.3 показано, как компьютер хранит число 12,75. Обратите внимание на 1 бит в столбцах 8, 4, ½ и ¼.

8 + 4 + ½ + ¼ = 12¾ = 12,75.

Таблица 2.3

128	64	32	16	8	4	2	1	.	½	¼	1/8	1/16
0	0	0	0	1	1	0	0	.	1	1	0	0

Таким образом, десятичное число 12,75 в двоичной системе счисления можно изобразить как 00001100.1100.

Как вы поняли, число 12,75 может быть точно представлено в двоичной системе. Однако такая форма представления применима не ко всем числам. И скоро вы в этом убедитесь.

Visual Studio 2017. Выполните команду File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект). В диалоговом окне Add New Project (Добавить новый проект) в списке Installed (Установленные) выберите пункт Visual C#. В центре диалогового окна выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)), присвойте ему имя `Numbers`, а затем нажмите кнопку OK.

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства) или нажмите сочетание клавиш Alt+Enter. В категории Startup Project (Запускаемый проект) установите переключатель в положение Current selection (Текущий проект). С этого момента вы можете просто щелкнуть на названии проекта на панели Solution Explorer (Обозреватель решений), а затем нажать сочетание клавиш Ctrl+F5 для сохранения, компиляции и запуска проекта (рис. 2.16).

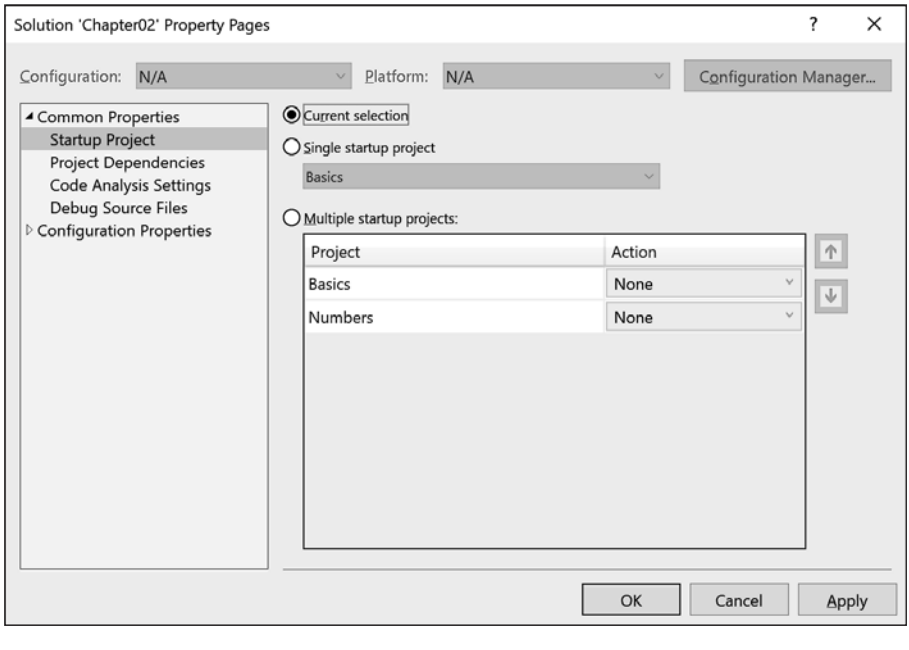


Рис. 2.16

Visual Studio Code. Создайте новую папку в каталоге `Chapter02` и присвойте ей имя `Numbers`.

Откройте папку `Numbers` и воспользуйтесь областью TERMINAL (Терминал) для создания консольного приложения с помощью команды `dotnet new console`. Когда вы откроете файл `Program.cs`, появится уведомление об обновлении пакетов.

Пример работы с числами

Наберите в методе `Main` следующий код:

```
Console.WriteLine($"int uses {sizeof(int)} bytes and can store numbers in the range
{int.MinValue:N0} to {int.MaxValue:N0}.");
Console.WriteLine($"double uses {sizeof(double)} bytes and can store numbers in the
range {double.MinValue:N0} to {double.MaxValue:N0}.");
Console.WriteLine($"decimal uses {sizeof(decimal)} bytes and can store numbers in
the range {decimal.MinValue:N0} to {decimal.MaxValue:N0}.");
```

Помните о том, что инструкции в двойных кавычках нужно вводить в одну строку.

Запустите консольное приложение, нажав сочетание клавиш Ctrl+F5 или введя команду `dotnet run`, и проанализируйте результат вывода:

```
int uses 4 bytes and can store numbers in the range -2,147,483,648 to
2,147,483,647.
double uses 8 bytes and can store numbers in the range -
179,769,313,486,232,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000 to 179,769,313,486,232,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000.
decimal uses 16 bytes and can store numbers in the range –
79,228,162,514,264,337,593,543,950,335 to
79,228,162,514,264,337,593,543,950,335.
```

Переменная int занимает четыре байта памяти и может хранить положительные или отрицательные числа в пределах до двух миллиардов. Переменная double использует восемь байт и может хранить еще бóльшие значения! Переменная decimal занимает 16 байт памяти и может хранить большие числа, но не настолько большие, как double.

Почему переменная `double` способна хранить бóльшие значения, чем переменная `decimal`, хотя задействует вполовину меньше памяти? Разберемся!

Сравнение типов double и decimal. Ниже инструкций из предыдущего упражнения введите приведенный далее код. Не беспокойтесь, если пока не понимаете синтаксис, хотя он и не настолько сложен:

```
double a = 0.1;
double b = 0.2;
if (a + b == 0.3)
{
  Console.WriteLine($"{a} + {b} equals 0.3");
}
else
{
  Console.WriteLine($"{a} + {b} does NOT equal 0.3");
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
0.1 + 0.2 does NOT equal 0.3
```

Тип `double` не обеспечивает точность. Используйте данный тип, только если точность не важна, особенно при сравнении двух чисел, к примеру, при измерении роста человека.

Проблема в предыдущем коде заключается в том, как компьютер хранит число 0,1 или кратное 0,1. Чтобы представить 0,1 в двоичном формате, компьютер хранит 1 в столбце 1/16, 1 в столбце 1/128, 1 в столбце 1/1024 и т. д. Десятичное число 0,1 равно бесконечно повторяющемуся двоичному 0.0001001001001 (табл. 2.4).

Таблица 2.4

4	2	1	.	½	¼	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048
0	0	0	.	0	0	0	1	0	0	1	0	0	1	0

Никогда не сравнивайте числа двойной точности с плавающей запятой с помощью оператора ==. Во время войны в Персидском заливе американский противоракетный комплекс Patriot был запрограммирован с использованием чисел двойной точности с плавающей запятой в вычислениях. Неточность в расчетах привела к тому, что комплекс не смог перехватить иракскую ракету P-17 и она попала в американские казармы в городе Дхарам, в результате чего погибли 28 американских солдат. Подробнее см. на сайте https://www.ima.umn.edu/~arnold/disasters/patriot.html.

Скопируйте и вставьте код, который вы написали до использования переменных `double`, а затем измените его так, как показано в коде, приведенном ниже:

```
decimal c = 0.1M; // M обозначает десятичный литерал
decimal d = 0.2M;
if (c + d == 0.3M)
{
  Console.WriteLine($"{c} + {d} equals 0.3");
}
else
{
  Console.WriteLine($"{c} + {d} does NOT equal 0.3");
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
0.1 + 0.2 equals 0.3
```

Тип `decimal` точен, поскольку хранит значение как большое целое число и смещает десятичную запятую. Например, 0,1 хранится как 1 с записью, что десятичная запятая смещается на один разряд влево. Число 12,75 хранится как 1275 с записью, что десятичная запятая смещается на два разряда влево.

Используйте тип int для натуральных, a double для вещественных чисел. Тип decimal служит для денежных расчетов, измерений в чертежах и машиностроительных схемах и повсюду, где важна точность вещественных чисел.

Тип double поддерживает несколько полезных специальных значений. Так, double.NaN представляет значение, не являющееся числом, double. Epsilon — наименьшее положительное число, которое может быть сохранено как значение double, и double.Infinity — бесконечно большое значение. Эти специальные значения можно использовать при сравнении значений double.

Хранение логических значений

Логическое значение может быть только одним из двух: или `true`, или `false`, как показано в коде, приведенном ниже. Логические значения чаще всего используются при ветвлении и зацикливании (см. главу 3).

```
bool happy = true;
bool sad = false;
```

Тип object

Специальный тип `object` позволяет хранить данные любого типа, но ради такой гибкости приходится жертвовать: код получается более сложным и менее производительным из-за процессов упаковки/распаковки типа значения. По возможности лучше избегать использования типа `object`.

С этого момента договоримся: вы уже знаете, как создавать консольные приложения в Visual Studio 2017 и Visual Studio Code, поэтому я буду приводить краткие инструкции.

Создайте в консольном приложении новый проект с именем `Variables` и добавьте код, показанный ниже, в метод `Main`:

```
object height = 1.88; // хранение double в виде объекта
object name = "Amir"; // хранение string в виде объекта
int length1 = name.Length; // выдает ошибку компиляции!
int length2 = ((string)name).Length; // приведение для доступа к членам
```

Тип `object` доступен с самой первой версии языка C#, но в версии C# 2 и выше в качестве альтернативы используются более эффективные *универсальные шаблоны* (мы рассмотрим их позже), которые обеспечивают желаемую гибкость без снижения производительности.

Тип dynamic

Существует еще один специальный тип `dynamic`, который тоже позволяет хранить данные любого типа и, подобно типу `object`, делает это в ущерб производительности. В отличие от последнего, значение, хранящееся в переменной, может содержать члены типа без явного приведения, как показано в коде, представленном ниже:

```
// хранение строки как dynamic
dynamic anotherName = "Ahmed";
// компилируется, но может вызвать исключение!
int length = anotherName.Length;
```

Ограничения типа `dynamic` заключаются в том, что Visual Studio не отображает меню IntelliSense, призванное облегчить набор кода, поскольку компилятор не выполняет проверку во время сборки. Вместо этого общеязыковая исполняющая среда проверяет член во время выполнения. Ключевое слово `dynamic` было добавлено в версии C# 4.

Локальные переменные

Локальные переменные объявляются внутри методов и существуют только во время вызова этих методов. После возвращения метода память, выделенная для хранения любых локальных переменных, освобождается.

Собственно говоря, типы значений освобождаются, а ссылочные типы ожидают сборки мусора. О различиях между типами значений и ссылочными типами вы узнаете позже.

Указание типа локальной переменной

Наберите следующий код для объявления и присвоения значений нескольким локальным переменным в методе `Main`. Обратите внимание: тип указывается перед именем каждой переменной:

```
int population = 66_000_000; // 66 миллионов человек в Англии
double weight = 1.88; // в килограммах
decimal price = 4.99M; // в фунтах стерлингов
string fruit = "Apples"; // строки в двойных кавычках
char letter = 'Z'; // символы в одиночных кавычках
bool happy = true; // логическое значение — true или false
```

Visual Studio 2017 и Visual Studio Code подчеркивают зеленой волнистой чертой имена переменных, значения которым присвоены, но нигде не используются.

Объявление типа локальной переменной

Ключевое слово `var` можно применять для объявления локальных переменных. Компилятор определит тип данных по литеральному значению, введенному после оператора присваивания `=`.

Числовой литерал без десятичной запятой определяется как переменная `int`, если не добавлен суффикс `L`. В последнем случае определяется переменная `long`. Числовой литерал с десятичной запятой определяется как `double`; если добавить суффикс `M`, то как переменная `decimal`; в случае добавления суффикса `F` — как переменная `float`. Двойные кавычки обозначают переменную `string`, а одинарные — переменную `char`. Значения `true` и `false` определяют переменную `bool`.

Измените свой код так, чтобы использовать ключевое слово `var`:

```
var population = 66_000_000; // 66 миллионов человек в Англии
var weight = 1.88; // в килограммах
var price = 4.99M; // в фунтах стерлингов
var fruit = "Apples"; // строки в двойных кавычках
var letter = 'Z'; // символы в одиночных кавычках
var happy = true; // логическое значение — true или false
```

Несмотря на несомненное удобство ключевого слова var, умные программисты стараются избегать его, чтобы обеспечить удобочитаемость своего кода и определения типов на глаз. Если говорить обо мне, то я использую это ключевое слово, только если тип и без того ясен. Например, в коде, показанном ниже, первая инструкция так же понятна и ясна, как и вторая, в которой указывается тип переменной xml, но первая короче. С другой стороны, третья инструкция не ясна на первый взгляд, поэтому лучше применить четвертый вариант. Короче говоря, сомневаешься — пиши полностью!

```
// удачное применение var
var xml1 = new XmlDocument();
// излишний повтор XmlDocument
XmlDocument xml2 = new XmlDocument();

// неудачное применение var; каков тип данных переменной file1?
var file1 =
  File.CreateText(@"C:\something.txt");
// прекрасный пример объявления определенного типа
StreamWriter file2 =
File.CreateText(@"C:\something.txt");
```

Создаем тип, допускающий значение null

Большинство примитивных типов, кроме `string`, представляют собой *типы значений*. Это значит, что им должны присваиваться значения. Можно определить зна-

чение по умолчанию типа, используя оператор `default()`. Значение по умолчанию для переменной `int` равно `0` (нулю).

```
Console.WriteLine($"{default(int)}"); // 0
Console.WriteLine($"{default(bool)}"); // False
Console.WriteLine($"{default(DateTime)}"); // 1/01/0001 00:00:00
```

Строки — *ссылочные типы*. То есть они содержат адрес переменной в памяти, а не значение самой переменной. Переменная ссылочного типа может иметь значение `null`. Это специальное литеральное значение, указывающее: переменная ни на что не ссылается (пока).

Вы узнаете больше о типах значений и ссылочных типах из главы 6.

Иногда требуется, чтобы типу можно было присвоить значение `null`. Это можно сделать, добавив символ вопроса в качестве суффикса к типу при объявлении переменной, как показано в коде, приведенном ниже:

```
int ICannotBeNull = 4;
int? ICouldBeNull = null;
Console.WriteLine(ICouldBeNull.GetValueOrDefault()); // 0
ICouldBeNull = 4;
Console.WriteLine(ICouldBeNull.GetValueOrDefault()); // 4
```

Обзор ссылочных типов, допускающих значение null

Пожалуй, самое значительное нововведение, которое корпорация Microsoft планирует для языка C# 8.0, — это появление ссылочных типов, допускающих значение `null`. «Постойте!» — возможно, подумаете вы. — «Ссылочные типы и так допускают значение `null`!» И будете правы, однако в C# 8.0 ссылочные типы больше не станут допускать пустое значение (`null`) по умолчанию. В C# 8.0 при необходимости присвоить ссылочному значению `null` придется воспользоваться точно таким же синтаксисом, как и для значимых типов, допускающих данное значение, иными словами, добавить символ `?` после объявления типа.

Это серьезное изменение, и оно затронет весь код, скомпонованный с помощью компилятора языка C# 8.0, так что компания Microsoft должна провести расширенное тестирование перед выпуском C# 8.0, причем сделать это нужно на наиболее ранних этапах. Версия для предварительного ознакомления с данной функцией была выпущена за год до финального релиза языка C# 8.0. Более подробную информацию о данной версии можно узнать, перейдя по следующей ссылке (по ней же можно загрузить эту версию и протестировать ее): https://github.com/dotnet/csharplang/wiki/Nullable-Reference-Types-Preview.

Ошибка ценой миллиард долларов

Значение `null` используется так часто и в стольких языках, что многие опытные программисты никогда не подвергают сомнению необходимость его существования. Однако есть множество ситуаций, когда можно было бы писать более качественный и простой код, если бы переменным нельзя было присваивать это значение. Получить более подробную информацию можно, перейдя на сайт www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare, где изобретатель значения `null` сэр Чарльз Энтони Ричард Хоар (Sir Charles Antony Richard Hoare) признает свою ошибку в записанном часовом разговоре.

Изменение установок по умолчанию для типов, допускающих значение null, в C# 8.0

Корпорация Microsoft могла бы принять решение оставить установки по умолчанию для ссылочных типов: разрешить задание значения `null` для обеспечения совместимости с сегодняшними программами и добавить новый синтаксис, позволяющий указать, что тот или иной ссылочный тип не допускает значение `null`. Однако Microsoft планирует нечто более радикальное: нововведение, которое изначально будет очень болезненным, но в долгосрочной перспективе переориентирует язык C# в лучшую сторону. Это нововведение также уравняет установки и поведение по умолчанию для ссылочных и значимых типов, что упростит синтаксис, а следовательно, изучение и использование языка.

Каким же образом будут работать ссылочные типы, допускающие значение `null`? Рассмотрим пример. При сохранения информации об адресе можно задать в качестве обязательных название улицы, города и региона, но при этом разрешить не указывать номер дома (то есть `null`), как показано в следующем листинге:

```
class Address
{
  string? Building;       // допускается null
  string Street;          // необходимо значение
  string City;            // необходимо значение
  string Region;          // необходимо значение
}
```

Вот почему эта новая особенность языка называется ссылочными типами, допускающими значение `null`. Начиная с C# 8.0, обычные ссылочные типы не допускают значение `null` по умолчанию. Чтобы тот или иной ссылочный тип допускал данное значение, используется такой же синтаксис, как и в случае со значимыми типами.

Проверка на null

Важно проверять, содержит ли значение `null` переменная ссылочного типа или типа, допускающего такое значение. В противном случае может возникнуть ис-

ключение `NullReferenceException`, которое приведет к ошибке при выполнении кода.

```
// проверить ICouldBeNull на значение null перед использованием
if (ICouldBeNull != null)
{
  // сделать что-нибудь с ICouldBeNull
}
```

Если вы пытаетесь получить поле или свойство из переменной, которая может быть `null`, то используйте оператор доступа к членам с проверкой `null` (`?.`), как показано в коде, приведенном ниже:

```
string authorName = null;
// если authorName равно null, то вместо вызова исключения вернуть null
int? howManyLetters = authorName?.Length;
```

Иногда требуется либо назначить переменную в качестве результата, либо использовать альтернативное значение, например, нуль, если переменная равна `null`. Это действие выполняется с помощью оператора объединения с `null` (`??`), как показано в коде, показанном ниже:

```
// результат равен трем, если howManyLetters равна null
var result = howManyLetters ?? 3;
Console.WriteLine(result);
```

Хранение группы значений в массиве

При необходимости сохранить несколько значений одного и того же типа можно объявить массив. В качестве примера сохраним четыре имени в строковом массиве.

Код, показанный ниже, объявляет массив для хранения четырех строковых значений. Затем в нем сохраняются строковые значения с индексами позиций от 0 до 3 (индексация массивов ведется с нуля, поэтому последний элемент всегда на единицу меньше, чем длина массива). И наконец, каждый элемент в массиве перебирается с помощью инструкции `for`, о чем будет подробнее сказано в главе 3.

Добавьте следующие строки кода в конец метода `Main`:

```
// объявление размера массива
string[] names = new string[4];
// хранение элементов с индексами позиций
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";
for (int i = 0; i < names.Length; i++)
{
  Console.WriteLine(names[i]); // прочитать элемент с данным индексом позиции
}
```

Массивы всегда имеют фиксированный размер, поэтому предварительно нужно решить, сколько элементов в массиве вы хотите сохранить, прежде чем создавать его. Массивы удобны для временного хранения нескольких элементов, а коллекции — при динамическом добавлении и удалении элементов. О последних мы поговорим в главе 8.

Анализ консольных приложений

Вы уже научились создавать и использовать простые консольные приложения, и теперь время изучить их более глубоко.

Консольные приложения основаны на тексте и запускаются в командной строке. Обычно они выполняют простые задачи, которые требуется автоматизировать, такие как компиляция файла или шифрование раздела файла конфигурации. Им можно передавать аргументы для управления поведением. Так, для шифрования раздела со строками подключения к базе данных в файле `Web.config` используйте следующую строку кода:

```
aspnet_regiis -pdf "connectionStrings" "c:\mywebsite"
```

Отображение вывода пользователю

Две основные задачи любого консольного приложения: запись и чтение данных. Мы уже использовали метод `WriteLine` для вывода. Если бы не требовался возврат каретки в конце каждой строки, то можно было бы применить метод `Write`.

Язык C# 6 и более поздние версии содержит удобную функцию *интерполяции строк*. Она позволяет легко выводить одну или несколько переменных в удобном отформатированном виде. Строка с префиксом `$` должна содержать фигурные скобки вокруг имени переменной для вывода текущего значения этой переменной в данной позиции строки.

В проекте `Variables` введите следующие инструкции в конец метода `Main`:

```
Console.WriteLine($"The UK population is {population}.");
Console.Write($"The UK population is {population:N0}. ");
Console.WriteLine($"{weight}kg of {fruit} costs {price:C}.");
```

Запустите консольное приложение и просмотрите результат вывода:

```
The UK population is 66000000.
The UK population is 66,000,000. 1.88kg of Apples costs J4.99.
```

Значение переменной может быть отформатировано с помощью специального кода. `N0` форматирует число с запятыми[1] в качестве разделителей тысяч и без

[1] В зависимости от настроек операционной системы будут использованы либо запятые, либо точки, либо пробелы (см., например: https://docs.oracle.com/cd/E19455-01/806-0169/overview-9/index.html). — *Примеч. науч. ред.*

дробной части. Код C форматирует число как значение валюты. Формат валюты определяется текущим потоком. Если вы запустите этот код на компьютере в Великобритании, то значение будет выведено в фунтах стерлингов. А если в Германии — то в евро.

Получение пользовательского ввода

Мы можем получать ввод от пользователя с помощью метода `ReadLine`. Этот метод ожидает, пока пользователь не начнет набирать некий текст. После того как пользователь нажал клавишу Enter, весь его ввод возвращается как строка.

Спросим, как зовут пользователя и сколько ему лет. Позднее мы преобразуем значение возраста в число, но на данный момент оставим его строковым:

```
Console.Write("Type your first name and press ENTER: ");
string firstName = Console.ReadLine();
Console.Write("Type your age and press ENTER: ");
string age = Console.ReadLine();
Console.WriteLine($"Hello {firstName}, you look good for {age}.");
```

Запустите консольное приложение и проанализируйте результат вывода.

Введите `name` и `age`, как показано ниже:

```
Type your name and press ENTER: Gary
Type your age and press ENTER: 34
Hello Gary, you look good for 34.
```

Импорт пространства имен

Возможно, вы заметили, что в отличие от нашего самого первого приложения мы не набирали слово `System` перед `Console`.

`System` является пространством имен. Такое пространство — это нечто вроде адреса типа. Например, вы могли бы точно дать понять, кого имеете в виду, с помощью кода типа `Oxford.HighStreet.BobSmith`; он сообщает, что надо искать человека по имени Боб Смит на улице Хай-стрит в городе Оксфорд.

Строка `System.Console.WriteLine` сообщает компилятору: следует искать метод `WriteLine` в типе `Console` в пространстве имен `System`.

Чтобы упростить код, среда Visual Studio 2017 или команда `dotnet new console` в Visual Studio Code добавляют в начало файла с кодом инструкцию, указывающую компилятору всегда искать в пространстве имен `System` типы, при использовании которых не были явно указаны их пространства имен. Вот как это выглядит в коде:

```
using System;
```

Это так называемый *импорт пространства имен*. Его результат заключается в том, что все доступные типы из данного пространства будут доступны вашей программе и отобразятся в IntelliSense во время написания кода.

Упрощение работы с командной строкой

В языке C# 6 и более поздних версий инструкцию `using` можно использовать для дальнейшего упрощения кода.

Добавьте следующую строку в начало файла:

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

Теперь в коде не нужно указывать тип `Console`. Можно использовать функцию Find and Replace (Найти и заменить), чтобы его убрать.

Первым делом выделите текст `Console.` в вашем коде (убедитесь, что выделили и точку после слова `Console`).

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+H для выполнения быстрой замены (рис. 2.17) (убедитесь, что в поле Replace (Заменить) не указано какое-либо значение).

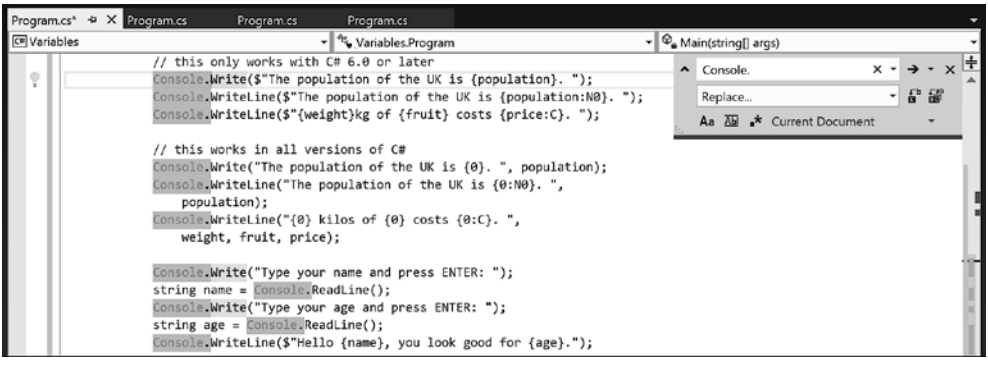


Рис. 2.17

В Visual Studio Code выполните команду Edit ▸ Replace (Редактировать ▸ Заменить) (рис. 2.18).

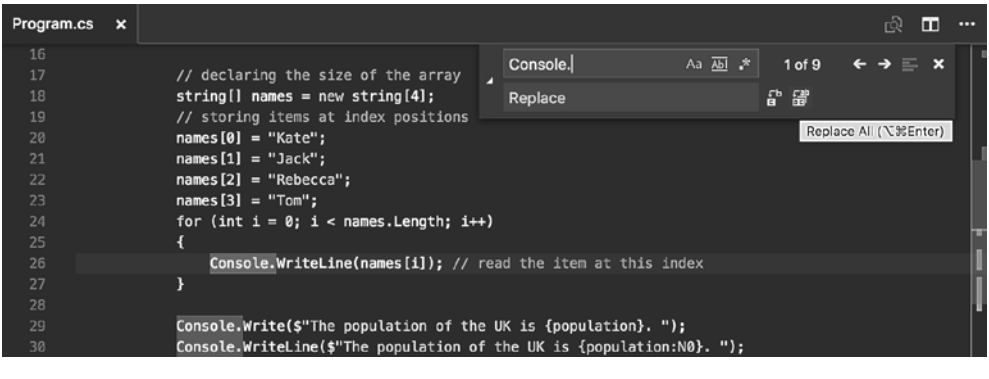


Рис. 2.18

И в Visual Studio 2017, и в Visual Studio Code нажмите кнопку Replace All (Заменить все) либо сочетание клавиш Alt+A или Alt+Cmd+Enter для замены всех вхождений, нажмите кнопку OK, а затем закройте панель замены, щелкнув на значке × в правом верхнем углу.

Чтение аргументов и работа с массивами

Разберемся, что за аргумент `string[] args` в методе `Main`. Это массив, используемый для передачи аргументов в консольное приложение.

Создайте новый проект консольного приложения с именем `Arguments`.

Допустим, у нас есть возможность ввести следующую команду в командной строке:

```
Arguments apples bananas cherries
```

Мы могли бы читать имена фруктов и ягод, считывая их из массива `args`, который всегда передается в метод `Main` консольного приложения.

Как было сказано выше, квадратные скобки используются для того, чтобы обозначить массив — объект, способный содержать несколько значений. У массивов есть свойство `Length`, которое сообщает, сколько элементов в данный момент находится в массиве. Если есть хотя бы один элемент, то можно получить к нему доступ, зная его индекс. Индексы начинают отсчет с нуля, поэтому первый элемент в массиве имеет индекс `0`.

Добавьте инструкцию для статического импорта типа `System.Console`. Напишите инструкцию для вывода количества аргументов, переданных приложению. Удалите ненужные инструкции `using`. Теперь ваш код должен выглядеть так:

```
using static System.Console;

namespace Arguments
{
  class Program
  {
    static void Main(string[] args)
    {
      WriteLine($"There are {args.Length} arguments.");
    }
  }
}
```

Не забудьте статически импортировать тип System.Console в грядущих проектах, чтобы упростить код, поскольку я не стану повторять эти инструкции.

Запустите консольное приложение и проанализируйте результат вывода.

```
There are 0 arguments.
```

Передача аргументов в Visual Studio 2017

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте Arguments и в контекстном меню выберите пункт Properties (Свойства).

На открывшейся панели перейдите на вкладку Debug (Отладка) и в группе элементов управления Application arguments (Аргументы приложения) укажите четыре аргумента, разделяя их пробелами, как показано в коде и на рис. 2.19:

```
firstarg second-arg third:arg "fourth arg"
```

Вы можете использовать практически любые символы в именах аргументов, включая дефисы и двоеточия. Если нужно указать пробел в имени аргумента, то оборачивайте его в двойные кавычки.

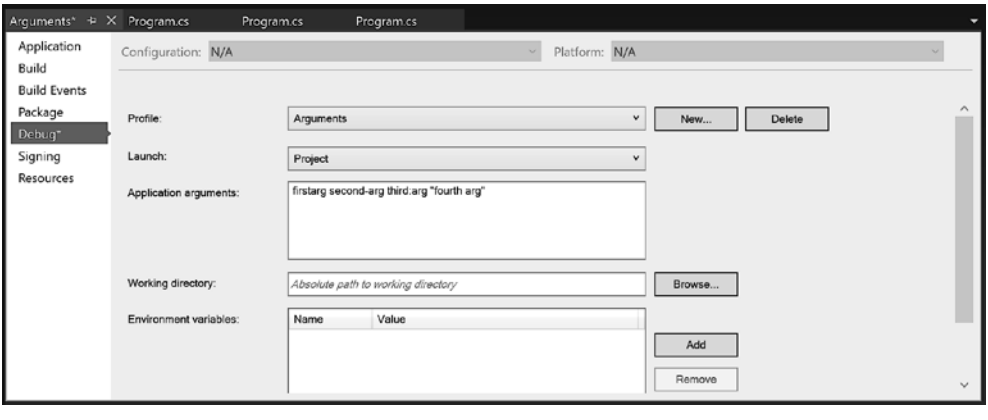


Рис. 2.19

Передача аргументов в Visual Studio Code

Укажите аргументы после команды `dotnet run`, как показано в следующем примере кода:

```
dotnet run firstarg second-arg third:arg "fourth arg"
```

Результат вывода

Запустите консольное приложение с переданными аргументами и просмотрите результат вывода:

```
There are 4 arguments.
```

Перечисление аргументов

Для перечисления или перебора (то есть циклической обработки) значений этих четырех аргументов добавьте следующие выделенные полужирным строки кода после вывода длины массива:

```
WriteLine($"There are {args.Length} arguments.");
foreach (string arg in args)
{
  WriteLine(arg);
}
```

Теперь мы будем применять эти аргументы, чтобы пользователь мог выбрать цвет шрифта и маркера текста, а также ширину и высоту окна консоли.

Измените значения аргументов так:

```
red yellow 50 10
```

Импортируйте пространство имен `System`, добавив следующую строку кода в верхнюю часть файла, если она отсутствует:

```
using System;
```

Пространство имен System нужно импортировать, чтобы компилятор смог обрабатывать типы ConsoleColor и Enum. Отсутствие этих типов в меню IntelliSense говорит о том, что в коде файла отсутствует инструкция using System;.

Добавьте код, выделенный полужирным шрифтом, выше существующего таким образом:

```
ForegroundColor = (ConsoleColor)
  Enum.Parse(typeof(ConsoleColor), args[0], true);
BackgroundColor = (ConsoleColor)
  Enum.Parse(typeof(ConsoleColor), args[1], true);
WindowWidth = int.Parse(args[2]);
WindowHeight = int.Parse(args[3]);

WriteLine($"There are {args.Length} arguments.");
foreach (var arg in args)
{
  WriteLine(arg);
}
```

Запуск в Windows. В Visual Studio 2017 нажмите сочетание клавиш Ctrl+F5. Консольное окно теперь имеет другой размер и иной цвет шрифта и маркера текста (рис. 2.20).

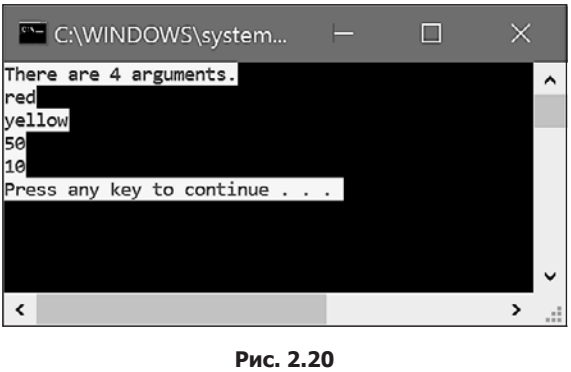


Рис. 2.20

Запуск в macOS. В Visual Studio Code введите следующую команду в области TERMINAL (Терминал):

```
dotnet run red yellow 50 10
```

Вы увидите окно с сообщением об ошибке и подробные сведения о ней (рис. 2.21).

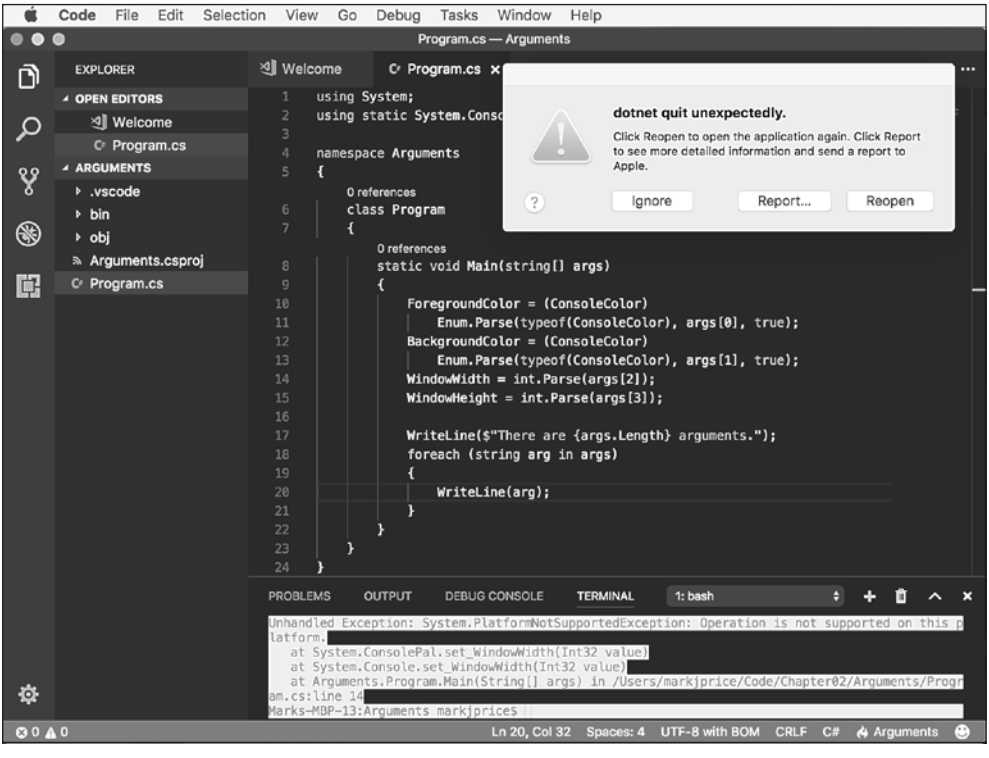


Рис. 2.21

Нажмите кнопку Ignore (Игнорировать).

Хотя компилятор не выдал ошибку или предупреждение, во время выполнения некоторые API-вызовы могут вызывать ошибку на ряде платформ. Консольное приложение, запущенное в операционной системе Windows, способно изменять свой размер, однако в macOS такой возможности нет.

Работа с платформами без поддержки API

Взаимодействовать с платформами, не прибегая к поддержке API, позволяет обработчик исключений.

Измените код, чтобы обернуть строки кода, меняющие высоту и ширину окна, инструкцией `try`, как показано ниже:

```
try
{
  WindowWidth = int.Parse(args[2]);
  WindowHeight = int.Parse(args[3]);
}
catch(PlatformNotSupportedException)
{
  WriteLine("The current platform does not support changing the size of
             a console window.");
}
```

Если вы перезапустите консольное приложение, то увидите, что исключение перехвачено и пользователю выведено соответствующее сообщение.

Работа с переменными

Операторы выполняют простые операции, подобные сложению и умножению операндов, например чисел. Обычно они возвращают новое значение, являющееся результатом операции.

Большинство операторов — *бинарные*; это значит, что они работают с двумя операндами, как показано в следующем псевдокоде:

```
var результатОперации = ПервыйОперанд оператор ВторойОперанд;
```

Из оставшихся некоторые операторы являются *унарными*, то есть работают с одним операндом, как показано в псевдокоде ниже:

```
var результатОперации = Операнд оператор;
```

А *тернарные* операторы принимают три операнда, как показано в этом псевдокоде:

```
var результатОперации = ПервыйОперанд ПервыйОператор ВторойОперанд ВторойОператор
ТретийОперанд;
```

Экспериментируем с унарными операторами

К двум самым распространенным унарным операторам относятся операторы инкремента (`++`) и декремента (`--`) числа.

В Visual Studio 2017 откройте меню View (Вид), раскройте в нем пункт Other Windows (Другие окна), а затем выберите пункт C# Interactive (Интерактивный C#).

В Visual Studio Code создайте новое консольное приложение и напишите собственные инструкции для вывода результатов с помощью метода Console.WriteLine().

Введите следующий код:

```
> int i = 3;
> i
3
```

Обратите внимание: когда вы вводите полную инструкцию, заканчивающуюся точкой с запятой, она выполняется после нажатия клавиши Enter.

В первой инструкции используется оператор присваивания = для присвоения значения 3 переменной i. Когда вы вводите имя переменной в окне консоли, возвращается текущее значение переменной.

Введите следующие инструкции и, прежде чем нажмете клавишу Enter, попытайтесь угадать, чему будет равно значение переменных x и y:

```
> int x = 3;
> int y = x++;
```

Теперь проверьте значения этих переменных. Вы удивитесь, но значение y равно 3:

```
> x
4
> y
3
```

Переменной y присвоено значение 3, поскольку оператор ++ выполняется уже после присваивания. Это так называемый *постфикс*. При необходимости выполнить инкрементирование перед присваиванием используйте *префикс*, как показано в коде, представленном ниже:

```
> int x = 3;
> int y = ++x;
> x
4
> y
4
```

Декрементировать значение можно с помощью оператора --.

Из-за путаницы с префиксами и постфиксами операторов инкремента и декремента в процессе присваивания разработчики языка программирования Swift планируют отказаться от поддержки данного оператора в версии 3. Я рекомендую программистам на языке C# никогда не сочетать операторы ++ и — с оператором присваивания =. Лучше выполнять эти операции как отдельные инструкции.

Экспериментируем с арифметическими операторами

Арифметические операторы используются для выполнения арифметических операций с числами. Введите следующие команды на панели C# Interactive (Интерактивный C#):

```
> 11 + 3
14
> 11 - 3
8
> 11 * 3
```

```
33
> 11 / 3
3
> 11 % 3
2
> 11.0 / 3
3.6666666666666665
```

Чтобы понять, как работают операторы деления (`/`) и остатка от деления (деления по модулю) (`%`) при применении к целым числам (натуральным числам), вам нужно вспомнить уроки из средней школы.

Представьте, что у вас есть 11 леденцов и три друга. Как разделить леденцы поровну между друзьями? Можно дать по три леденца каждому другу, после чего останется два леденца. Они представляют собой остаток от деления, также известный как модуль. Если же у вас 12 леденцов, каждый друг получает четыре леденца и ничего не остается. Таким образом, остаток равен 0.

Если брать вещественные числа (скажем, `11,0`), то оператор деления возвращает значение с плавающей запятой (например, `3,6666666666665`), а не целое число.

Логические операторы и операторы сравнения

Логические операторы и операторы сравнения возвращают в результате значение `true` или `false`. В следующей главе вы примените операторы сравнения в инструкциях `if` и `while`, чтобы проверить условия.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенное упражнение и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

Какой тип нужно выбрать для каждого указанного ниже числа?

1. Телефонный номер.
2. Рост.
3. Возраст.
4. Размер оклада.
5. ISBN книги.
6. Цена книги.
7. Вес книги.

8. Размер населения страны.
9. Количество звезд во Вселенной.
10. Количество сотрудников на каждом из предприятий малого и среднего бизнеса (примерно до 50 000 сотрудников на каждом предприятии).

Упражнение 2.1. Числовые размеры и диапазоны

Создайте проект `Exercise02`, который выводит количество байтов в памяти для каждого из следующих числовых типов, а также минимальное и максимальное допустимые значения: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`.

Посмотрите в Интернете документацию «Составное форматирование» на сайте MSDN по адресу https://msdn.microsoft.com/en-us/library/txafckwd(v=vs.110).aspx, чтобы разобраться, как выравнивать текст в окне консольного приложения.

Результат вывода вашего приложения должен выглядеть примерно так (рис. 2.22).

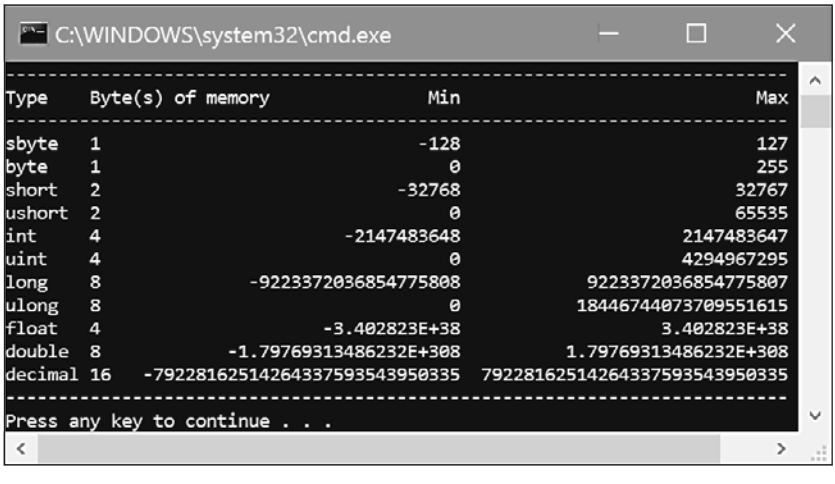


Рис. 2.22

Дополнительные ресурсы

- Ключевые слова C#: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/index.
- Метод `Main()` и аргументы командной строки (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/mainand-command-args/.

- Типы (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/.
- Инструкции, выражения и операторы (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statementsexpressions-operators/.
- Строки (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/.
- Типы, допускающие значения `NULL` (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/nullable-types/.
- Класс `Console`: https://msdn.microsoft.com/en-us/library/system.console(v=vs.110).aspx.
- Операторы C#: https://msdn.microsoft.com/en-us/library/6a71f45d.aspx.

Резюме

В этой главе вы научились объявлять переменные конкретного типа или с использованием ключевого слова `var`; мы обсудили встроенные числовые, текстовые и логические типы. Кроме того, разобрались, как выбирать тот или иной тип; а также поэкспериментировали с некоторыми операторами.

В следующей главе вы разберетесь в принципах ветвления, перебора, преобразования типов и обработки исключений.

3 Управление потоком выполнения и преобразование типов

Эта глава посвящена написанию кода, который выполняет операции, повторяет блоки инструкций, преобразует типы данных, обрабатывает исключения и проверяет переполнение числовых переменных.

В этой главе:

- инструкции выбора;
- инструкции перебора;
- приведение и преобразование типов;
- обработка исключений;
- проверка переполнения.

Инструкции выбора

Код каждого приложения должен обеспечивать возможность выбора одного из нескольких вариантов и ветвиться в соответствии с выбранным вариантом. Две инструкции выбора в языке C# называются `if` и `switch`. Первая применима для любых приложений, а вторая позволяет упростить код в некоторых стандартных ситуациях.

Visual Studio 2017

Запустите Microsoft Visual Studio 2017. Нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) выберите пункт Visual C#. В центре диалогового окна выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)), присвойте ему имя

SelectionStatements, укажите расположение по адресу C:\Code, введите имя решения Chapter03, а затем нажмите кнопку OK.

Visual Studio Code в среде macOS, Linux или Windows

Если вы выполняли упражнения из предыдущих глав, то у вас уже должен быть создан каталог Code в пользовательской папке. В противном случае создайте его, а затем вложите подкаталог Chapter03, в который поместите еще один подкаталог SelectionStatements.

Запустите Visual Studio Code и откройте каталог /Chapter03/SelectionStatements/.

В Visual Studio Code выполните команду View ▸ Integrated Terminal (Вид ▸ Интегрированный терминал), а затем введите следующую команду в области TERMINAL (Терминал):

```
dotnet new console
```

Инструкция if

Данная инструкция определяет, по какой ветви кода необходимо следовать после оценки логического выражения. Блок else — необязателен. Инструкции, содержащие инструкцию if, можно вкладывать друг в друга и комбинировать. Каждое логическое выражение может быть независимым от других.

Использование в коде

Добавьте следующие инструкции в метод Main для проверки, имеет ли это консольное приложение какие-либо аргументы, переданные ему:

```
if (args.Length == 0)
{
  WriteLine("There are no arguments.");
}
else
{
  WriteLine("There is at least one argument.");
}
```

Ввиду того что в каждом блоке указывается только одна инструкция, данный код *можно* переписать без фигурных скобок, как показано в коде, приведенном ниже:

```
if (args.Length == 0)
  WriteLine("There are no arguments.");
else
  WriteLine("There is at least one argument.");
```

Такой стиль использования инструкции if не рекомендуется, поскольку может содержать серьезные ошибки, например печально известный баг *#gotofail* в операционной системе iOS на смартфонах Apple iPhone. На протяжении 18 месяцев после релиза версии iOS 6 в ней присутствовала ошибка в коде протокола *Secure*

Sockets Layer (SSL). Из-за этого любой пользователь, подключающийся через браузер Safari к защищенным сайтам, скажем к сервису интернет-банкинга, не был должным образом защищен, так как важная проверка была случайно пропущена. См. сайт https://gotofail.com/.

Только потому, что вы *можете* опустить фигурные скобки, не стоит так делать. Ваш код не станет без них «более эффективным», вместо этого он станет более сложным в сопровождении и потенциально более уязвимым (твит на рис. 3.1).

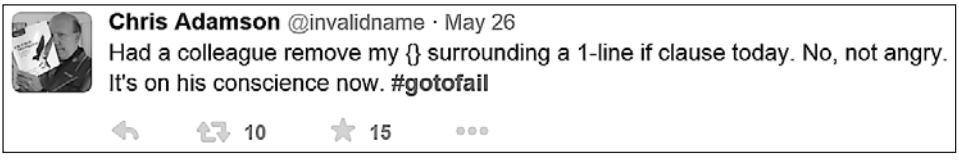


Рис. 3.1

Сопоставление шаблонов с помощью инструкции if

Новая возможность версии C# 7 — это *сопоставление шаблонов*. Синтаксис инструкции `if` позволяет использовать ключевое слово `is` в сочетании с объявлением локальной переменной, чтобы сделать код более безопасным.

Добавьте в конец метода `Main` приведенные ниже инструкции. Если значение, хранящееся в переменной `o`, является `int`, то значение присваивается локальной переменной `i`, которая затем может применяться внутри блока `if`. Это безопаснее, чем использование переменной `o`, поскольку мы точно знаем: `i` есть `int`, а не что-то другое:

```
object o = "3";
int j = 4;

if(o is int i)
{
  WriteLine($"{i} x {j} = {i * j}");
}
else
{
  WriteLine("o is not an int so it cannot multiply!");
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
o is not an int so it cannot multiply!
```

Удалите двойные кавычки вокруг значения `"3"`, чтобы значение, хранящееся в переменной `o`, стало `int` вместо `string`, а затем перезапустите консольное приложение и проанализируйте результат вывода:

```
3 x 4 = 12
```

Инструкция switch

Инструкция `switch` отличается от `if` тем, что проверяет одно выражение на соответствие одному из множества условий (`case`). Каждое условие относится к единственному выражению. Каждое условие должно заканчиваться ключевым словом `break` (например, `case 1` в коде ниже), ключевыми словами `goto case` (например, `case 2` в коде ниже) или не иметь никаких инструкций (`case 3` в коде ниже)[1].

Использование в коде

Введите указанный ниже код после инструкции `if`, написанной вами ранее. Обратите внимание, что первая строка — метка, к которой можно перейти, а вторая строка генерирует случайное число. Инструкция `switch` выбирает ветвь, основываясь на значении этого случайного числа.

```
A_label:
  var number = (new Random()).Next(1, 7);
  WriteLine($"My random number is {number}");
  switch (number)
  {
    case 1:
      WriteLine("One");
      break; // переход в конец инструкции switch
    case 2:
      WriteLine("Two");
      goto case 1;
    case 3:
    case 4:
      WriteLine("Three or four");
      goto case 1;
    case 5:
    // заснуть на полсекунды
    System.Threading.Thread.Sleep(500);
    goto A_label;
  default:
    WriteLine("Default");
    break;
} // конец инструкции switch
```

Ключевое слово goto можно добавить для перехода к другому условию или метке. Такой подход не одобряется большинством программистов, но может стать хорошим решением при кодировании логики в некоторых сценариях. Не используйте его без необходимости.

[1] Утверждение верно не до конца. Как минимум из case еще можно возвращать результат (return) или выбрасывать исключение. Возможно, автор умолчал об этом сознательно (эти темы еще не рассмотрены). — *Примеч. науч. ред.*

В Visual Studio 2017 запустите приложение, нажав сочетание клавиш Ctrl+F5.

В Visual Studio Code запустите приложение вводом следующей команды на панели Integrated Terminal (Интегрированный терминал):

```
dotnet run
```

Запустите приложение несколько раз, чтобы посмотреть, что происходит при различных значениях случайного числа, как показано в приведенном ниже выводе Visual Studio Code:

```
bash-3.2$ dotnet run
My random number is 4
Three or four
One
bash-3.2$ dotnet run
My random number is 2
Two
One
bash-3.2$ dotnet run
My random number is 1
One
```

Сопоставление шаблонов с помощью инструкции switch

Как и `if`, инструкция `switch` поддерживает сопоставление шаблонов в C# 7. Значения условий больше не должны быть литеральными. Они могут быть шаблонами.

Добавьте следующую инструкцию в начало файла:

```
using System.IO;
```

Добавьте инструкции, приведенные после примечания, в конец метода `Main`.

Если вы работаете с операционной системой macOS, то раскомментируйте соответствующую инструкцию, которая задает переменную PATH, и закомментируйте инструкцию для Windows, а также замените мое имя пользовательской папки своим.

```
// string path = "/Users/markjprice/Code/Chapter03"; // macOS
string path = @"C:\Code\Chapter03"; // Windows
Stream s = File.Open(
  Path.Combine(path, "file.txt"),
  FileMode.OpenOrCreate);

switch(s)
{
  case FileStream writeableFile when s.CanWrite:
    WriteLine("The stream is to a file that I can write to.");
    break;
  case FileStream readOnlyFile:
    WriteLine("The stream is to a read-only file.");
```

```
    break;
  case MemoryStream ms:
    WriteLine("The stream is to a memory address.");
    break;
  default: // всегда оценивается последним,
           // несмотря на его текущее положение
    WriteLine("The stream is some other type.");
    break;
  case null:
    WriteLine("The stream is null.");
    break;
}
```

Обратите внимание: переменная `s` объявлена с типом `Stream`.

Подробнее пространство имен System.IO и тип Stream мы разберем в главе 9. Получить дополнительную информацию о сопоставлении шаблонов можно, перейдя по ссылке https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching.

В платформе .NET доступно несколько подтипов `Stream`, включая `FileStream` и `MemoryStream`. В версии C# 7 и более поздней ваш код может быть более лаконичным благодаря ветвлению на основе подтипа потока и объявлению/назначению локальной переменной для безопасного использования.

Кроме того, обратите внимание, что инструкции `case` могут содержать ключевое слово `when` для выполнения более специфичного сопоставления шаблонов. В первом блоке `case` в предыдущем коде `s` считалось бы совпадением только в том случае, если бы поток являлся `FileStream`, а его свойство `CanWrite` было бы истинно.

Инструкции перебора

Инструкции перебора повторяют блок кода, пока условие истинно для каждого элемента группы. Выбор того, какую инструкцию следует использовать, основывается на сочетании простоты понимания решения логической задачи и личных предпочтений.

Запустите Visual Studio 2017 или Visual Studio Code и создайте новый проект консольного приложения с именем `IterationStatements`.

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства) или нажмите сочетание клавиш Alt+Enter. В категории Startup Project (Запускаемый проект) установите переключатель в положение Current selection (Текущий проект). С этого момента вы можете просто щелкнуть на проекте на панели Solution Explorer (Обозреватель решений), а затем нажать сочетание клавиш Ctrl+F5 для запуска текущего проекта.

Инструкция while

Данная инструкция оценивает логическое выражение и продолжает цикл, пока оно остается истинным.

Напечатайте код, показанный ниже, в методе `Main`:

```
int x = 0;
while (x < 10)
{
  WriteLine(x);
  x++;
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
0
1
2
3
4
5
6
7
8
9
```

Инструкция do

Похожа на `while`, за исключением того, что логическое выражение проверяется в конце блока кода, а не в начале. Это подразумевает обязательное выполнение кода хотя бы один раз.

Напечатайте код, показанный ниже, в конце метода `Main` и запустите приложение:

```
string password = string.Empty;
do
{
  Write("Enter your password: ");
  password = ReadLine();
} while (password != "secret");
WriteLine("Correct!");
```

Вы увидите запрос на ввод пароля, который будет появляться, пока вы не введете корректный пароль, о чем сообщается в выводе, показанном ниже:

```
Enter your password: password
Enter your password: 12345678
Enter your password: ninja
Enter your password: asdfghjkl
Enter your password: secret
Correct!
```

Чтобы потренироваться, добавьте инструкции, ограничивающие количество попыток ввода пароля десятью, после чего, если корректный пароль так и не был введен, выводится сообщение об ошибке.

Инструкция for

Аналогична `while`, за исключением более лаконичного синтаксиса. Комбинирует инструкцию инициализатора, выполняемого однократно при запуске цикла, логическое выражение, результат проверки которого определяет продолжение цикла, и инкремент, выполняемый в конце цикла. Инструкция `for` часто используется с целочисленным счетчиком, как показано в листинге ниже:

```
for (int y = 1; y <= 10; y++)
{
  WriteLine(y);
}
```

Запустите консольное приложение и проанализируйте результат вывода, который должен быть представлен числами в диапазоне от 1 до 10.

Инструкция foreach

Данная инструкция несколько отличается от предыдущих трех инструкций перебора. Используется для выполнения блока инструкций для каждого элемента в последовательности, скажем в массиве или коллекции. Каждый элемент доступен только для чтения, и если во время перебора последовательность изменяется, например, путем добавления или удаления элемента, то будет вызвано исключение.

В методе `Main` напечатайте код, показанный ниже. Этот код создает массив строковых переменных, а затем выводит длину каждой из них:

```
string[] names = { "Adam", "Barry", "Charlie" };
foreach (string name in names)
{
  WriteLine($"{name} has {name.Length} characters.");
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Adam has 4 characters.
Barry has 5 characters.
Charlie has 7 characters.
```

Технически инструкция `foreach` применима к любому типу, который реализует интерфейс под названием `IEnumerable`. На данный момент вам не нужно беспокоиться о том, какой интерфейс используется. Вы изучите интерфейсы в главе 6.

Компилятор превратит инструкцию `foreach` из предыдущего кода в нечто похожее на это:

```
IEnumerator e = names.GetEnumerator();
while (e.MoveNext())
{
  string name = (string)e.Current;    // Current – только для чтения!
  WriteLine($"{name} has {name.Length} characters.");
}
```

Из-за использования итератора переменная, объявленная в инструкции foreach, не может применяться для изменения значения текущего элемента.

Приведение и преобразование типов

Конвертировать различные типы данных вам придется довольно часто. Например, данные, вводимые в текстовое поле, изначально сохраняются в переменную типа `string`, но затем должны быть преобразованы в дату, время, число или некий другой тип, в зависимости от того, как эти данные следует хранить и обрабатывать.

Приведение типов может быть *неявным* и *явным*. Первое происходит автоматически и является безопасным, то есть вы не потеряете информацию. Второе же должно производиться вручную, поскольку можно потерять информацию, например точность числа. Совершая явное приведение типов, вы сообщаете компилятору языка C#, что понимаете и принимаете на себя эти риски.

Создайте новый проект консольного приложения с именем `CastingConverting`.

Приведение от числа к числу

Переменную `int` можно неявно привести к типу `double`.

Неявное приведение

Добавьте в метод `Main` следующие инструкции:

```
int a = 10;
double b = a; // тип int может быть сохранен как double
WriteLine(b);
```

Вы не сможете неявно привести переменную `double` к типу `int`, поскольку это потенциально небезопасно и способно вызвать потерю данных.

Добавьте в метод `Main` эти инструкции:

```
double c = 9.8;
int d = c; // компилятор выдаст ошибку на данной строке
WriteLine(d);
```

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+W или Ctrl+E для открытия панели Error List (Список ошибок) (рис. 3.2).

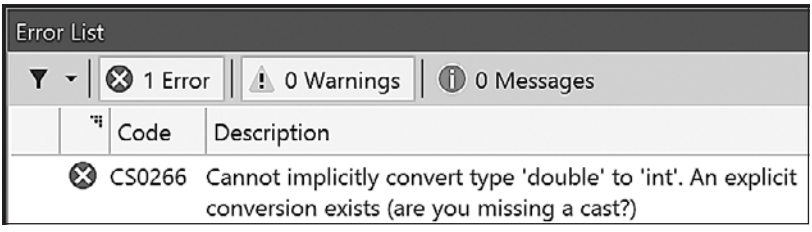


Рис. 3.2

В Visual Studio Code или откройте панель PROBLEMS (Проблемы), или, использовав команду `dotnet run`, выведите результат вывода, показанный ниже:

```
Compiling CastingConverting for .NETCoreApp,Version=v1.1
/usr/local/share/dotnet/dotnet compile-csc
@/Users/markjprice/Code/Chapter03/CastingConverting/obj/Debug/netcoreapp1.1/dotnet-
compile.rsp returned Exit Code 1
/Users/markjprice/Code/Chapter03/CastingConverting/Program.cs(14,21): error CS0266:
Cannot implicitly convert type 'double' to 'int'.An explicit conversion exists (are
you missing a cast?)
Compilation failed.
0 Warning(s)
1 Error(s)
Time elapsed 00:00:01.0461813
```

Явное приведение

Необходимо явно привести тип `double` к типу `int`, заключив тип, к которому вы хотите привести переменную `double`, в круглые скобки. Парой круглых скобок обозначается *оператор приведения*. И даже в этом случае нужно иметь в виду, что часть после десятичной запятой может быть без предупреждения отброшена.

Измените инструкцию присваивания переменной `d`, как показано в коде, представленном ниже:

```
double c = 9.8;
int d = (int)c;
WriteLine(d); // d равняется 9 с потерей части .8
```

Запустите консольное приложение и проанализируйте результат вывода:

```
10
9
```

Аналогичную операцию следует выполнять при перемещении значений между большими и меньшими целыми числами. Обратите внимание: можно потерять данные, поскольку слишком крупное значение будет установлено как `-1`!

Введите этот код:

```
long e = 10;
int f = (int)e;
WriteLine($"e is {e} and f is {f}");
e = long.MaxValue;
f = (int)e;
WriteLine($"e is {e} and f is {f}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
e is 10 and f is 10
e is 9223372036854775807 and f is -1
```

Использование типа System.Convert

Альтернативой применению оператора приведения является использование типа `System.Convert`.

В начале файла `Program.cs` напечатайте код, показанный ниже:

```
using static System.Convert;
```

Добавьте следующие инструкции в нижнюю часть метода `Main`:

```
double g = 9.8;
int h = ToInt32(g);
WriteLine($"g is {g} and h is {h}");
```

Запустите консольное приложение и проанализируйте результат вывода.

```
g is 9.8 and h is 10
```

Одно из различий между приведением и преобразованием заключается в том, что при преобразовании значение double округляется до 10 вместо отбрасывания части после десятичной запятой.

Тип `System.Convert` можно преобразовывать во все числовые типы языка C# и из них, а также в логические значения, строки и значения даты/времени.

Округление чисел

Вы убедились, что оператор приведения отбрасывает десятичную часть вещественного числа и при преобразовании число округляется в большую или меньшую сторону. Но каковы правила округления?

В средней школе дети учатся округлять числа следующим образом: если десятичная часть больше или равна 0,5, то число округляется в *бóльшую* сторону, а если десятичная меньше — то в *меньшую*.

Введите следующий код:

```
double i = 9.49;
double j = 9.5;
```

```
double k = 10.49;
double l = 10.5;
WriteLine($"i is {i}, ToInt(i) is {ToInt32(i)}");
WriteLine($"j is {j}, ToInt(j) is {ToInt32(j)}");
WriteLine($"k is {k}, ToInt(k) is {ToInt32(k)}");
WriteLine($"l is {l}, ToInt(l) is {ToInt32(l)}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
i is 9.49, ToInt(i) is 9
j is 9.5, ToInt(j) is 10
k is 10.49, ToInt(k) is 10
l is 10.5, ToInt(l) is 10
```

Число округляется все по тем же правилам, что и в школе, если дробная часть не равна `0,5`. В противном случае округление осуществляется к ближайшему четному числу: в бóльшую сторону, если целая часть числа нечетная, и в меньшую, если четная.

Эта схема известна как *банковское округление*, и она предпочтительнее, поскольку уменьшает вероятность ошибки. К сожалению, другие языки, например JavaScript, используют правила математического округления.

Проверяйте правила округления в каждом языке программирования, который используете. Код может работать не так, как вы ожидаете!

Преобразование любого типа в строку

Наиболее распространено преобразование в тип `string`, поэтому все типы включают метод `ToString`, наследуемый ими от класса `System.Object`.

Метод `ToString` преобразует текущее значение любой переменной в текстовое представление. Некоторые типы не могут быть представлены в виде текста, поэтому возвращают свое пространство имен и название типа.

Добавьте следующие инструкции в нижнюю часть метода `Main`:

```
int number = 12;
WriteLine(number.ToString());
bool boolean = true;
WriteLine(boolean.ToString());
DateTime now = DateTime.Now;
WriteLine(now.ToString());
object me = new object();
WriteLine(me.ToString());
```

Запустите консольное приложение и проанализируйте результат вывода:

```
12
True
27/01/2017 13:48:54
System.Object
```

Конвертация бинарного объекта в строку

При необходимости сохранить или передать бинарный объект лучше не отправлять его в виде необработанных битов, ведь вы не знаете, какие ошибки могут возникнуть при интерпретации битов, например, сетевым протоколом, ответственным за их передачу, или другой операционной системой, считывающей и сохраняющей объект.

Самый безопасный способ — конвертировать бинарный объект в строку безопасных символов. Программисты называют этот подход кодировкой *Base64*.

Для типа `Convert` реализовано два метода: `ToBase64String` и `FromBase64String`, которые выполнят конвертацию за вас.

Добавьте следующие инструкции в конец метода `Main`:

```
// выделить массив из 128 байт
byte[] binaryObject = new byte[128];

// заполнить массив случайными байтами
(new Random()).NextBytes(binaryObject);

WriteLine("Binary Object as bytes:");
for(int index = 0; index < binaryObject.Length; index++)
{
  Write($"{binaryObject[index]:X} ");
}
WriteLine();

// преобразовать в строку Base64
string encoded = Convert.ToBase64String(binaryObject);

WriteLine($"Binary Object as Base64: {encoded}");
```

По умолчанию целочисленное значение int будет выведено с учетом десятеричного формата записи, то есть base10. Вы можете использовать коды форматирования (например, index:X) для представления значений в шестнадцатеричном формате.

Запустите консольное приложение и проанализируйте результат вывода:

```
Binary Object as bytes:
B3 4D 55 DE 2D E BB CF BE 4D E6 53 C3 C2 9B 67 3 45 F9 E5 20 61 7E 4F 7A 81 EC 49
F0 49 1D 8E D4 F7 DB 54 AF A0 81 5 B8 BE CE F8 36 90 7A D4 36 42 4 75 81 1B AB 51
CE 5 63 AC 22 72 DE 74 2F 57 7F CB E7 47 B7 62 C3 F4 2D 61 93 85 18 EA 6 17 12 AE
44 A8 D B8 4C 89 85 A9 3C D5 E2 46 E0 59 C9 DF 10 AF ED EF 8AA1 B1 8D EE 4A BE 48
EC 79 A5 A 5F 2F 30 87 4A C7 7F 5D C1 D 26 EE
Binary Object as Base64:
s01V3i0Ou8++TeZTw8KbZwNF+eUgYX5PeoHsSfBJHY7U99tUr6CBBbi+zvg2kHrUNkIEdYEbq1HOBWOsIn
LedC9Xf8vnR7diw/QtYZOFGOoGFxKuRKgNuEyJhak81eJG4FnJ3xCv7e+KobGN7kq+SOx5pQpfLzCHSsd/
XcENJu4=
```

Разбор строк для преобразования в числа или значения даты и времени

Вторым по популярности считается преобразование из строковых переменных в числа или значения даты и времени. Здесь вместо метода `ToString` выступает метод `Parse`. Последний поддерживается только несколькими типами, включая числовые типы и `DateTime`.

Добавьте следующие инструкции в метод `Main`:

```
int age = int.Parse("38");
DateTime birthday = DateTime.Parse("4 July 1980");
WriteLine($"I was born {age} years ago.");
WriteLine($"My birthday is {birthday}.");
WriteLine($"My birthday is {birthday:D}.");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
I was born 38 years ago.
My birthday is 04/07/1980 00:00:00.
My birthday is 04 July 1980.
```

По умолчанию значение даты и времени выводится в коротком формате. Вы можете использовать описатели формата типа D (для вывода только даты в полном формате). Есть множество других описателей формата для распространенных сценариев.

С методом `Parse` связана одна проблема, которая заключается в том, что он выдает ошибку, если строка не может быть преобразована.

Добавьте следующие инструкции в нижнюю часть метода `Main`:

```
int count = int.Parse("abc");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Unhandled Exception: System.FormatException: Input string was not in a correct format.
```

Избежать ошибки можно, использовав вместо `Parse` метод `TryParse`. Последний пробует преобразовать исходную строку и возвращает `true`, если конвертация возможна, и `false` в противном случае. Ключевое слово `out` требуется для того, чтобы разрешить методу `TryParse` устанавливать переменную `count` при выполнении преобразования.

Замените объявление `int count` следующими инструкциями:

```
Write("How many eggs are there? ");
int count;
string input = Console.ReadLine();
if (int.TryParse(input, out count))
{
  WriteLine($"There are {count} eggs.");
```

```
}
else
{
  WriteLine("I could not parse the input.");
}
```

Запустите приложение дважды. В первый раз введите число `12`. Вы увидите результат вывода, показанный ниже:

```
How many eggs are there? 12
There are 12 eggs.
```

Во второй раз введите слово `twelve`. Вы увидите такой результат вывода:

```
How many eggs are there? twelve
I could not parse the count.
```

Можно использовать и тип Convert. Но, как и метод Parse, он выдает ошибку, если преобразование невозможно.

Обработка исключений при преобразовании типов

Вы видели несколько сценариев, при выполнении которых возникали ошибки, инструкций, при выполнении которых «выбрасывались» исключения.

Разумеется, рекомендуется писать код, не выбрасывающий исключений, выполнив проверку инструкций `if`, но это не всегда возможно. В таких ситуациях необходимо перехватить (поймать) исключение и обработать его.

Как вы видели, в случае ошибки консольное приложение по умолчанию выводит информацию об исключении и прекращает работу.

Обработку исключений можно взять под контроль, используя инструкцию `try`.

Инструкция try

Создайте новый проект консольного приложения `HandlingExceptions`.

При возникновении предположения о том, что та или иная инструкция может вызвать ошибку, вы должны обернуть эту инструкцию кодом блока `try`. Например, анализ строки для преобразования в число способен повлечь ошибку. Нам не нужно ничего делать в блоке `catch`. Когда код, показанный ниже, выполнится, ошибка будет перехвачена и не отобразится, а консольное приложение продолжит работу.

Добавьте в метод `Main` следующие инструкции:

```
WriteLine("Before parsing");
Write("What is your age? ");
string input = Console.ReadLine();
```

```
try
{
  int age = int.Parse(input);
  WriteLine($"You are {age} years old.");
}
catch
{

}
WriteLine("After parsing");
```

Запустите консольное приложение и введите допустимое значение возраста, к примеру 43:

```
Before parsing
What is your age? 43
You are 43 years old.
After parsing
```

Запустите консольное приложение вновь и введите некорректное значение возраста, например kermit:

```
Before parsing
What is your age? kermit
After parsing
```

Исключение было перехвачено, но хорошо бы увидеть тип ошибки, которая произошла.

Перехват всех исключений

Измените инструкцию catch, как показано в коде ниже:

```
catch(Exception ex)
{
  WriteLine($"{ex.GetType()} says {ex.Message}");
}
```

Запустите консольное приложение и вновь введите некорректное значение возраста, скажем kermit:

```
Before parsing
What is your age? kermit
System.FormatException says Input string was not in a correct format.
After parsing
```

Перехват определенных исключений

Теперь, когда известно, какого типа ошибка возникла, можно улучшить код, перехватывая ошибку только данного типа и изменив сообщение, которое выводится пользователю.

Не трогая существующий блок `catch`, выше него добавьте следующий код:

```
catch (FormatException)
{
  WriteLine("The age you entered is not a valid number format.");
}
catch (Exception ex)
{
  WriteLine($"{ex.GetType()} says {ex.Message}");
}
```

Запустите программу и вновь введите некорректное значение возраста, например `kermit`:

```
Before parsing
What is your age? kermit
The age you entered is not a valid number format.
After parsing
```

Причина, по которой мы не должны изменять универсальный блок `catch`, состоит в том, что могут возникнуть исключения другого типа. К примеру, запустите программу и введите очень большое целое число, допустим `9876543210`:

```
Before parsing
What is your age? 9876543210
System.OverflowException says Value was either too large or too small for an Int32.
After parsing
```

Добавим перехватчик для этого нового типа исключения:

```
catch(OverflowException)
{
  WriteLine("Your age is a valid number format but it is either too big
             or small.");
}
catch (FormatException)
{
  WriteLine("The age you entered is not a valid number format.");
}
```

Перезапустите программу и вновь введите очень большое целое число:

```
Before parsing
What is your age? 9876543210
Your age is a valid number format but it is either too big or small.
After parsing
```

Порядок, в котором вы перехватываете исключения, важен. Правильный порядок связан с иерархией их наследования. С наследованием вы познакомитесь в главе 5, а пока не беспокойтесь об этом — компилятор выдаст ошибку сборки, если вы будете обрабатывать исключения в неправильном порядке.

Проверка переполнения

Ранее мы убедились, что в момент приведения числовых типов можно потерять данные, к примеру при выполнении приведения из переменной `long` к типу `int`. Если значение, хранящееся в типе, слишком велико, то возникнет переполнение.

Создайте новый проект консольного приложения с именем `CheckingForOverflow`.

Инструкция checked

При переполнении инструкция `checked` передает .NET команду вызова исключения вместо отсутствия реакции на переполнение.

Мы установим начальное значение переменной `int` на максимальное значение минус один. Затем увеличим его несколько раз, выводя каждый раз значение переменной. Обратите внимание: как только переменная достигнет своего максимального значения, произойдет переполнение и значение переменной уже будет равняться минимальному значению данного типа, после чего значение продолжит увеличиваться от этого минимума.

Напечатайте код, показанный ниже, в методе `Main` и запустите приложение:

```
int x = int.MaxValue - 1;
WriteLine(x);
x++;
WriteLine(x);
x++;
WriteLine(x);
x++;
WriteLine(x);
```

Запустите консольное приложение и проанализируйте результат вывода:

```
2147483646
2147483647
-2147483648
-2147483647
```

Теперь с помощью инструкции `checked` научим компилятор оповещать нас о переполнении:

```
checked
{
  int x = int.MaxValue - 1;
  WriteLine(x);
  x++;
  WriteLine(x);
  x++;
  WriteLine(x);
  x++;
  WriteLine(x);
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
2147483646
2147483647
Unhandled Exception: System.OverflowException: Arithmetic operation resulted in an
overflow.
```

Как и любое другое исключение, нужно поместить эти инструкции в блок `try` и отобразить пользователю поясняющее сообщение об ошибке:

```
try
{
  // сюда помещается предыдущий код
}
catch(OverflowException)
{
  WriteLine("The code overflowed but I caught the exception.");
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
2147483646
2147483647
The code overflowed but I caught the exception.
```

Инструкция unchecked

Запретить проверку переполнения можно с помощью ключевого слова `unchecked`.

Напечатайте данную ниже инструкцию в конце предыдущей. Компилятор ее не выполнит, поскольку она вызовет переполнение:

```
int y = int.MaxValue + 1;
```

Нажмите клавишу `F6` или введите команду `dotnet run` для сборки и обратите внимание на ошибку (рис. 3.3, снимок экрана со средой разработки Visual Studio 2017).

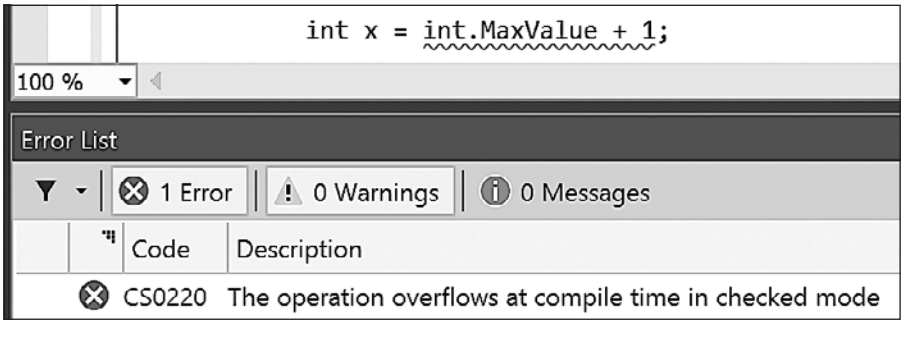


Рис. 3.3

Обратите внимание: проверка выполнялась *во время компиляции*. Чтобы запретить такие проверки, можно поместить инструкцию в блок `unchecked`, как показано в коде, приведенном ниже:

```
unchecked
{
  int y = int.MaxValue + 1;
  WriteLine(y); // выведет -2147483648
  y--;
  WriteLine(y); // выведет 2147483647
  y--;
  WriteLine(y); // выведет 2147483646
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
2147483646
2147483647
The code overflowed but I caught the exception.
-2147483648
2147483647
2147483646
```

Разумеется, вряд ли возникнет ситуация, когда вы захотите явно отключить проверку, поскольку будет допускаться переполнение. Но, возможно, в каком-то из ваших сценариев пригодится такое поведение компилятора.

Поиск документации

Этот раздел главы посвящен тому, как найти достоверную информацию о программировании во Всемирной паутине.

Сайты Microsoft Docs и MSDN

Главным ресурсом, позволяющим получить справочные сведения об инструментах и платформах Microsoft, ранее был Microsoft Developer Network (MSDN). Теперь это Microsoft Docs — https://docs.microsoft.com/.

В Visual Studio 2017 интегрирована поддержка ресурсов MSDN и Docs, поэтому вы можете, поместив указатель мыши внутрь ключевого слова C#, нажать клавишу F1, чтобы открыть браузер на соответствующей странице официальной документации.

В Visual Studio Code нажатие клавиши F1 отображает панель Command Palette (Палитра команд). Эта среда разработки не поддерживает контекстную справочную систему.

Переход к определению

Другой полезной клавишей в обеих средах, Visual Studio 2017 и Visual Studio Code, является F12. После ее нажатия вам будет показано, как выглядит определение типа, полученное с помощью чтения метаданных из скомпилированной сборки. Некоторые инструменты способны даже полностью восстановить C#-код из метаданных и IL-кода.

Введите показанный ниже код, щелкните внутри слова `int`, а затем нажмите клавишу F12 (или щелкните правой кнопкой мыши на слове `int` и в контекстном меню выберите Go To Definition (Перейти к определению)):

```
int z;
```

На появившейся новой вкладке с кодом вы увидите, что `int` находится в сборке `mscorlib.dll`; его имя — `Int32`; он расположен в пространстве имен `System`; и таким образом `int` — псевдоним для `System.Int32` (рис. 3.4).

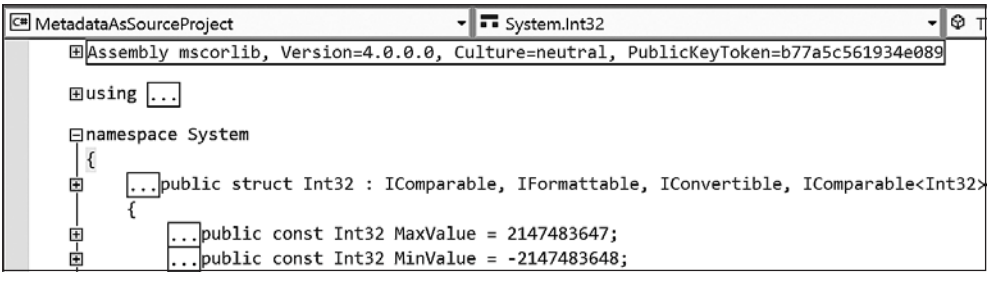


Рис. 3.4

Microsoft определяет `int`, используя ключевое слово `struct`; это значит, что `int` — тип значения, хранящийся в стеке. Вы также можете видеть, что `int` реализует такие интерфейсы, как `IComparable`, и содержит константы для своих максимальных и минимальных значений.

В окне редактора кода в Visual Studio 2017 прокрутите содержимое вкладки вниз и найдите методы `Parse`. Щелкните на маленьком квадратике с символом +, чтобы развернуть код (рис. 3.5).

В комментарии вы увидите, что сотрудники корпорации Microsoft задокументировали, какие исключения могут возникнуть, если вы вызываете этот метод (`ArgumentNullException`, `FormatException` и `OverflowException`).

Благодаря данной информации мы знаем, что вызов этого метода имеет смысл помещать в инструкцию `try` и какие исключения следует перехватывать.

Сайт StackOverflow

StackOverflow — самый популярный сторонний сайт, на котором можно найти ответы на сложные вопросы по программированию. Он настолько популярен, что поисковые системы, такие как DuckDuckGo, поддерживают специальный режим поиска в пределах этого сайта.

```
//
// Summary:
//     Converts the string representation of a number to its 32-bit signed integer equivalent.
//
// Parameters:
//   s:
//     A string containing a number to convert.
//
// Returns:
//     A 32-bit signed integer equivalent to the number contained in s.
//
// Exceptions:
//   T:System.ArgumentNullException:
//     s is null.
//
//   T:System.FormatException:
//     s is not in the correct format.
//
//   T:System.OverflowException:
//     s represents a number less than System.Int32.MinValue or greater than System.Int32.MaxValue.
public static Int32 Parse(string s);
```

Рис. 3.5

Перейдите на сайт DuckDuckGo.com и введите следующий запрос:

```
!so securestring
```

Вы увидите такой результат (рис. 3.6).

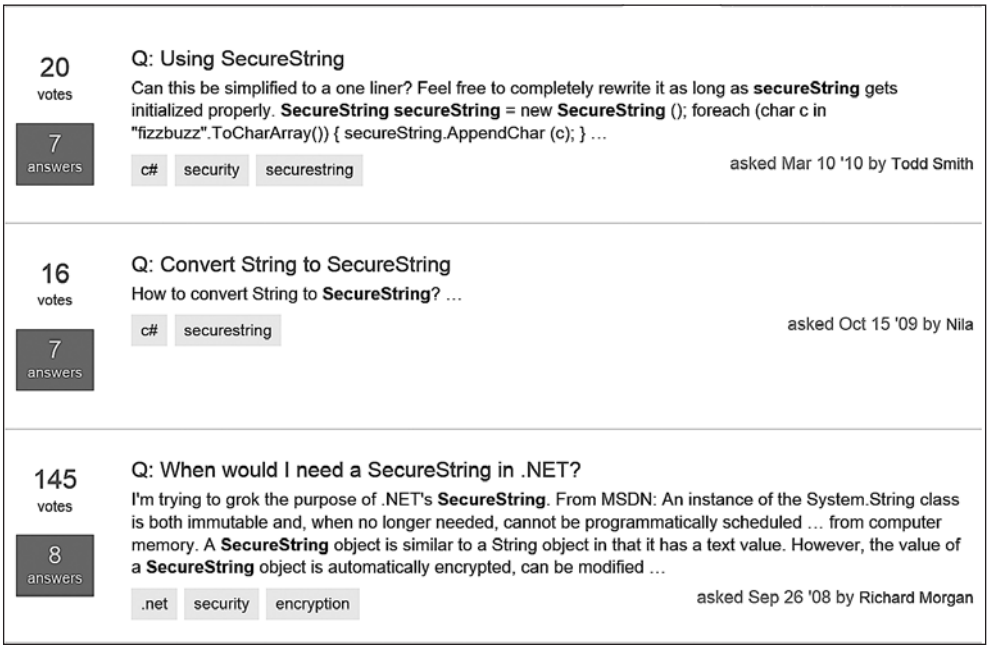


Рис. 3.6

Поисковая система Google

Вы можете выполнять поиск на сайте Google с дополнительными настройками, чтобы увеличить вероятность нахождения нужной информации.

Например, если вы ищете в Google информацию о `garbage collection` с помощью обычного запроса, то увидите ссылку на термин в «Википедии», а также список сервисов по вызову мусора в вашем районе (рис. 3.7).

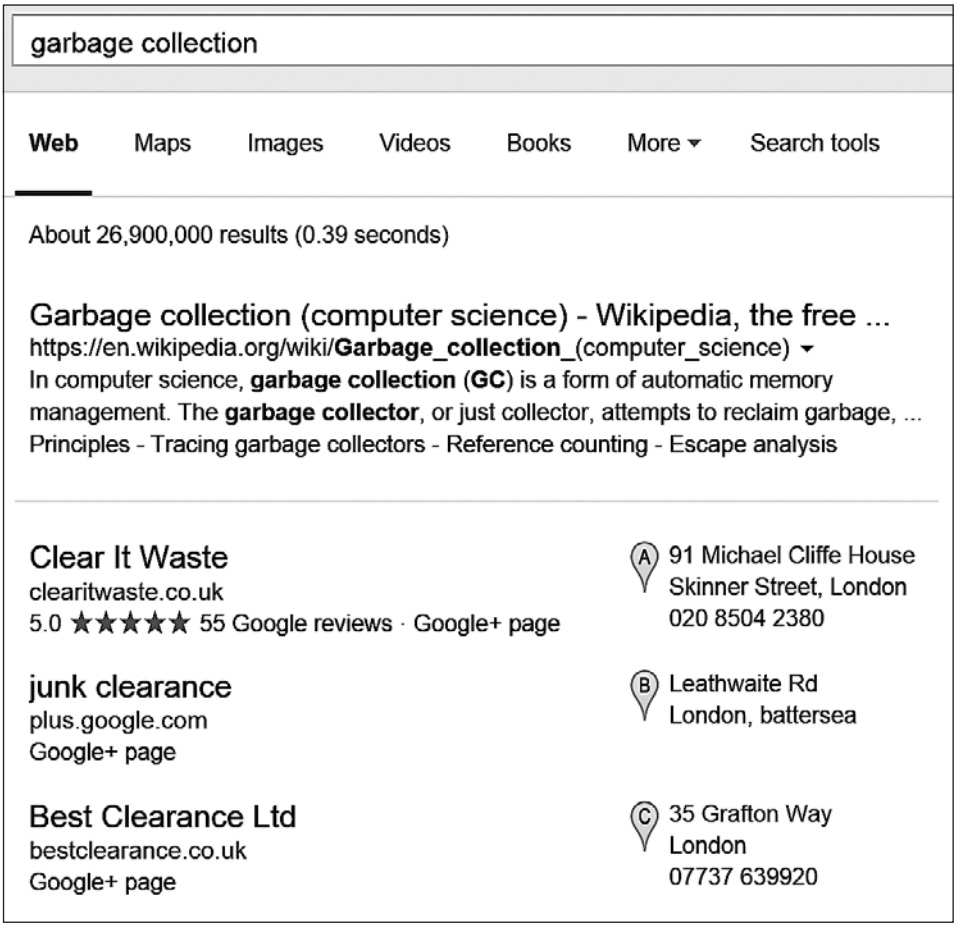


Рис. 3.7

Можно повысить эффективность поиска, ограничив его полезным сайтом, подобным StackOverflow (рис. 3.8).

Можно еще больше уточнить поиск, запретив отображать ссылки на языки, которые не нужны (например, C++) (рис. 3.9).

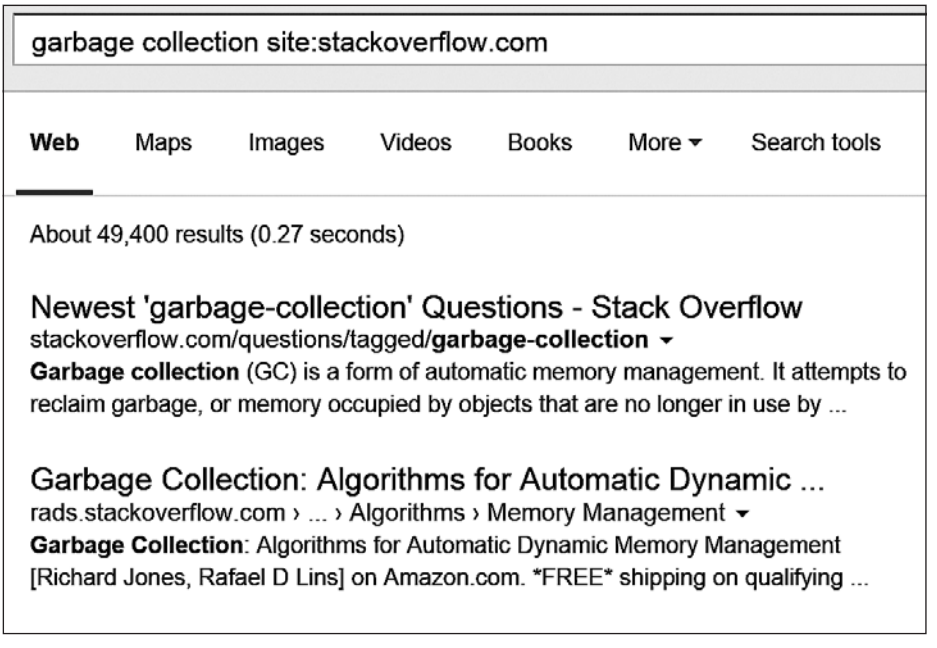

garbage collection site:stackoverflow.com
Web
Maps
Images
Videos
Books
More
Search tools
About 49,400 results (0.27 seconds)
Newest 'garbage-collection' Questions - Stack Overflow
stackoverflow.com/questions/tagged/garbage-collection
Garbage collection (GC) is a form of automatic memory management. It attempts to reclaim garbage, or memory occupied by objects that are no longer in use by ...
Garbage Collection: Algorithms for Automatic Dynamic ...
rads.stackoverflow.com › ... › Algorithms › Memory Management
Garbage Collection: Algorithms for Automatic Dynamic Memory Management [Richard Jones, Rafael D Lins] on Amazon.com. *FREE* shipping on qualifying ...

Рис. 3.8


garbage collection site:stackoverflow.com -c++ -java
Web
Maps
Images
Videos
Books
More
Search tools
About 19,100 results (0.30 seconds)
c# - Garbage Collection not happening even when needed ...
stackoverflow.com/.../garbage-collection-not-happening-even-when-nee...
4 Apr 2012 - As a sanity check, I have a button to force GC. When I push that, I quickly get 6GB back. Doesn't that prove my 6 arrays were not being referenced and ...
How expensive is it to call the Garbage Collector manually?
stackoverflow.com/.../how-expensive-is-it-to-call-the-garbage-collector-...
4 Feb 2014 - Yes, there are some other drawbacks. Even if you call GC.Collect, you can not ensure that objects that you believe are gone, are actually gone.

Рис. 3.9

Подписка на блоги

Быть в курсе новостей платформы .NET поможет ее официальный блог, который ведут группы разработчиков (рекомендую подписаться). В нем используется тег `Week in .NET`, благодаря которому можно быстро получить доступ к интересным событиям в мире .NET за предыдущую неделю. Блог доступен по адресу https://blogs.msdn.microsoft.com/dotnet/.

Паттерны проектирования

Паттерны проектирования — универсальное решение распространенных проблем. Программисты снова и снова решают одни и те же задачи. Когда в сообществе обнаруживают удачное решение, допускающее многократное использование, его называют паттерном проектирования. На протяжении последних лет было задокументировано множество таких паттернов.

Перейдите по ссылке https://en.wikipedia.org/wiki/Software_design_pattern#Classification_and_list, чтобы узнать больше информации о распространенных паттернах проектирования.

Группа Microsoft *patterns & practices* специализируется на документировании и продвижении паттернов проектирования для продуктов данной компании.

Прежде чем писать код с нуля, поищите, не решил ли кто-нибудь уже вашу проблему в общем случае и не опубликовал ли решение.

Паттерн Singleton. Это один из самых распространенных паттернов проектирования. Его примерами в .NET служат типы `Console` и `Math`.

Дополнительную информацию о данном паттерне можно получить по адресу https://en.wikipedia.org/wiki/Singleton_pattern.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

1. Где нужно искать справку по ключевому слову языка C#?
2. Где следует искать решения распространенных задач программирования?
3. Что случится, если разделить переменную `int` на `0`?

4. Что произойдет в случае деления переменной `double` на `0`?
5. Что происходит при переполнении переменной `int`, то есть когда ей присваивается значение, выходящее за пределы допустимого диапазона?
6. Чем отличаются инструкции `x = y++;` и `x = ++y;`?
7. В чем разница между ключевыми словами `break`, `continue` и `return` при использовании в инструкции цикла?
8. Из каких трех частей состоит инструкция `for` и какие из них обязательны?
9. В чем разница между операторами `=` и `==`?
10. Будет ли скомпилирована такая инструкция: `for ( ; true; ) ;`?

Упражнение 3.1. Циклы и переполнение

Что произойдет при выполнении кода, приведенного ниже?

```
int max = 500;
for (byte i = 0; i < max; i++)
{
  WriteLine(i);
}
```

В папке `Chapter03` создайте консольное приложение с именем `Exercise02` и введите код из листинга, приведенного выше. Запустите консольное приложение и проанализируйте результат вывода. Что произошло?

Какой код вы могли бы добавить (не изменяя строки предыдущего кода), чтобы получать предупреждение о проблеме?

Упражнение 3.2. Циклы и операторы

FizzBuzz — командная детская игра, обучающая делению. Игроки по очереди называют натуральные числа, заменяя те, что делятся на 3, словом *fizz*, а те, которые делятся на 5, словом *buzz*. Если встречается число, делящееся и на 3, и на 5, то говорят *fizzbuzz*.

Некоторые менеджеры персонала задают кандидатам простые задачки в духе этой игры. Нормальный программист должен написать такую программу на бумажке или маркерной доске за пару минут.

Но хотите узнать нечто, заставляющее поволноваться? Многие люди с профильным образованием вообще не могут справиться с данной задачей. Были даже случаи, когда кандидаты, подававшие резюме на вакансию «Старший разработчик», тратили на эту программу больше 10–15 минут.

> 199 из 200 претендентов на должность
> программиста вообще не могут писать код.
> Повторяю: они вообще не могут писать код.
>
> *Реджинальд Брейтуэйт*
> *(Reginald Braithwaite)*

Цитата взята с сайта http://blog.codinghorror.com/why-cant-programmers-program/.

Дополнительную информацию вы найдете по адресу http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/.

Создайте в `Chapter03` консольное приложение с именем `Exercise03`, вывод которого похож на имитацию игры FizzBuzz, выполняющей счет до 100. Результат должен выглядеть примерно так:

```
1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16, 17,
Fizz, 19, Buzz, Fizz, 22, 23, Fizz, Buzz, 26, Fizz, 28, 29, FizzBuzz, 31, 32,
Fizz, 34, Buzz, Fizz, 37, 38, Fizz, Buzz, 41, Fizz, 43, 44, FizzBuzz, 46, 47,
Fizz, 49, Buzz, Fizz, 52, 53, Fizz, Buzz, 56, Fizz, 58, 59, FizzBuzz, 61, 62,
Fizz, 64, Buzz, Fizz, 67, 68, Fizz, Buzz, 71, Fizz, 73, 74, FizzBuzz, 76, 77,
Fizz, 79, Buzz, Fizz, 82, 83, Fizz, Buzz, 86, Fizz, 88, 89, FizzBuzz, 91, 92,
Fizz, 94, Buzz, Fizz, 97, 98, Fizz, Buzz
```

Упражнение 3.3. Обработка исключений

Создайте в `Chapter03` консольное приложение с именем `Exercise04`, которое запрашивает у пользователя два числа в диапазоне от 0 до 255, а затем делит первое на второе. Вывод показан ниже:

```
Enter a number between 0 and 255: 100
Enter another number between 0 and 255: 8
100 divided by 8 is 12
```

Реализуйте обработчики исключений, перехватывающие любые произошедшие ошибки:

```
Enter a number between 0 and 255: apples
Enter another number between 0 and 255: bananas
FormatException: Input string was not in a correct format.
```

Дополнительные ресурсы

- Инструкции выбора (справочник по C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/selection-statements.
- Операторы перебора (справочник по C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/iteration-statements.
- Операторы перехода (справочник по C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/jump-statements.
- Приведение и преобразование типов (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/types/casting-and-type-conversions.
- Операторы обработки исключений (справочник по C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/exception-handling-statements.
- Переполнение стека: http://stackoverflow.com/.
- Расширенный поиск Google: http://www.google.com/advanced_search.

- Виртуальная академия Microsoft: https://mva.microsoft.com/.
- Microsoft Channel 9 — видеоканал для разработчиков: https://channel9.msdn.com/.
- Паттерны проектирования: https://msdn.microsoft.com/en-us/library/ff649977.aspx.
- Паттерны и их использование: https://msdn.microsoft.com/en-us/library/ff921345.aspx.

Резюме

Из этой главы вы узнали, как разветвлять и зацикливать код, выполнять преобразование типов, обрабатывать исключения и, самое главное, как найти документацию!

Теперь вы готовы научиться отлавливать баги в коде и исправлять их!

4 Создание, отладка и тестирование функций

Эта глава посвящена написанию функций для повторного использования кода, отладке логических ошибок во время разработки, протоколированию исключений во время выполнения и модульному тестированию вашего кода для избавления от ошибок и с целью гарантии стабильности и надежности.

В этой главе:

- написание функций;
- отладка во время разработки;
- ведение журнала во время выполнения;
- модульное тестирование.

Написание функций

Фундаментальный принцип программирования гласит: *«Не повторяйте самостоятельно»* (Don't repeat yourself, DRY). Если во время программирования вы обнаруживаете, что приходится писать одни и те же инструкции вновь и вновь, то превратите их в функцию. Функции как маленькие программы, выполняющие только одно маленькое задание. Например, можно написать функцию для исчисления налога с продаж и затем использовать ее много раз в финансовом приложении.

Как и у программ, у функций обычно есть ввод и вывод. Иногда их называют черными ящиками: вы поставляете сырье на входе и получаете готовое изделие на выходе. Когда функция создана, вам не нужно думать о том, как она работает.

Предположим, вы хотите помочь ребенку выучить таблицу умножения, поэтому нужно упростить задачу по генерации таблицы для указанного числа, например 12:

```
1 x 12 = 12
2 x 12 = 24
...
12 x 12 = 144
```

Ранее вы уже изучили инструкцию `for` и знаете, что она может использоваться для генерации повторяющихся строк вывода, когда существует некая регулярная закономерность, подобно случаю с таблицей умножения на 12, как показано в следующем листинге:

```
for (int row = 1; row <= 12; row++)
{
  Console.WriteLine($"{row} x 12 = {row * 12}");
}
```

Однако вместо выведения таблицы умножения на 12 мы хотим сделать программу более гибкой, так, чтобы она выводила таблицу умножения для любого числа. Это можно сделать, создав функцию, также называемую *методом*.

Написание функции таблицы умножения

Создайте решение и каталог `Chapter04` и добавьте новый проект консольного приложения под названием `WritingFunctions`.

Измените файл шаблона, как показано в следующем коде:

```
using static System.Console;

namespace WritingFunctions
{
  class Program
  {
    static void TimesTable(byte number)
    {
      WriteLine($"This is the {number} times table");
      for (int row = 1; row <= 12; row++)
      {
        WriteLine(
          $"{row} x {number} = {row * number}");
      }
    }

    static void RunTimesTable()
    {
      Write("Enter a number between 0 and 255: ");
      if (byte.TryParse(ReadLine(), out byte number))
      {
        TimesTable(number);
      }
      else
      {
        WriteLine("You did not enter a valid number!");
      }
    }

    static void Main(string[] args)
    {
```

```
      RunTimesTable();
    }
  }
}
```

Обратите внимание на следующие аспекты.

- Мы статически импортировали тип `Console`, таким образом можем упростить вызовы методов этого типа, таких как `WriteLine`.
- Мы написали функцию `TimesTable`, которой может быть передано значение типа `byte` с именем `number`.
- Функция `TimesTable` использует инструкцию `for` для вывода таблицы умножения на переданное в функцию число.
- Мы написали функцию `RunTimesTable`, приглашающую пользователя ввести число и впоследствии вызывающую метод `TimesTable`, при этом передавая в него введенное число. Эта функция также включает в себя код для работы с ситуациями, когда пользователь вводит неприемлемое число.
- Мы вызываем функцию `RunTimesTable` в методе `Main`.

У функции TimesTable только один ввод: параметр number, который должен быть типа byte. Функция TimesTable не возвращает в вызывающий ее код никаких значений, поэтому объявлена с ключевым словом void.

Запустите консольное приложение, введите число, например, `6` и проанализируйте результат вывода:

```
Enter a number between 0 and 255: 6
This is the 6 times table:
1 x 6 = 6
2 x 6 = 12
3 x 6 = 18
4 x 6 = 24
5 x 6 = 30
6 x 6 = 36
7 x 6 = 42
8 x 6 = 48
9 x 6 = 54
10 x 6 = 60
11 x 6 = 66
12 x 6 = 72
```

Написание функции, возвращающей значение

Предыдущая функция выполняла действия (циклический проход и вывод текста в консоль), однако не возвращала никаких значений.

Предположим, вам нужно исчислить налог с продаж или *налог на добавленную себестоимость (НДС)*. В Европе НДС варьируется от 8 % в Швейцарии до 27 %

в Венгрии. В США налог с продаж варьируется от 0 % в штате Орегон до 8,25 % в Калифорнии.

Добавьте в класс `Program` новую функцию `SalesTax`, а также функцию для запуска этой функции, как показано в листинге ниже, и обратите внимание на следующее:

- у функции `SalesTax` два входных параметра — `amount`, представляющий сумму потраченных денег, и `twoLetterRegionCode`, который станет указывать регион совершения покупки;
- функция `SalesTax` будет производить расчет с использованием инструкции `switch` и возвращать размер подлежащего оплате налога с продаж в виде значения типа `decimal`, поэтому перед именем функции мы объявили тип возвращаемого значения;
- функция `RunSalesTax` приглашает пользователя ввести сумму и код региона, а затем вызывает функцию `SalesTax` и выводит результат:

```
static decimal SalesTax(
  decimal amount, string twoLetterRegionCode)
{
  decimal rate = 0.0M;
  switch (twoLetterRegionCode)
  {
    case "CH": // Швейцария
      rate = 0.08M;
      break;
    case "DK": // Дания
    case "NO": // Норвегия
      rate = 0.25M;
      break;
    case "GB": // Великобритания
    case "FR": // Франция
      rate = 0.2M;
      break;
    case "HU": // Венгрия
      rate = 0.27M;
      break;
    case "OR": // Орегон
    case "AK": // Аляска
    case "MT": // Монтана
      rate = 0.0M;
      break;
    case "ND": // Северная Дакота
    case "WI": // Висконсин
    case "ME": // Мэриленд
    case "VA": // Вирджиния
      rate = 0.05M;
      break;
    case "CA": // Калифорния
      rate = 0.0825M;
      break;
```

```
    default: // большинство штатов США
      rate = 0.06M;
      break;
  }
  return amount * rate;
}

static void RunSalesTax()
{
  Write("Enter an amount: ");
  string amountInText = ReadLine();
  Write("Enter a two letter region code: ");
  string region = ReadLine();
  if (decimal.TryParse(amountInText, out decimal amount))
  {
    decimal taxToPay = SalesTax(amount, region);
    WriteLine($"You must pay {taxToPay} in sales tax.");
  }
  else
  {
    WriteLine("You did not enter a valid amount!");
  }
}
```

В методе `Main` закомментируйте вызов метода `RunTimesTable` и вызовите метод `RunSalesTax`, как показано в следующем листинге:

```
// RunTimesTable();

RunSalesTax();
```

Запустите консольное приложение, введите сумму и код региона и проанализируйте результат вывода:

```
Enter an amount: 149
Enter a two letter region code: FR
You must pay 29.8 in sales tax.
```

Можете ли вы представить какие-либо проблемы с функцией SalesTax в том виде, в котором мы ее написали? Что случится, если пользователь введет код региона Великобритании? Как бы вы переписали функцию для улучшения программы в этом аспекте?

Написание математических функций

Несмотря на то что, возможно, вам никогда не потребуется писать приложение, в котором будут использоваться математические функции, все изучают математику в школе, поэтому применение математики — общепринятый подход для изучения функций.

Форматирование выводимых чисел

Числа, используемые для вычислений, называются *количественными*, например 1, 2 и 3.

Числа, применяемые для нумерации, называются *порядковыми*, к примеру 1-й, 2-й и 3-й.

Мы напишем функцию `CardinalToOrdinal`, которая будет преобразовывать целочисленное значение `int` в строковое порядковое числительное `string`. Скажем, эта функция преобразует `1` в `1st`, `2` в `2nd` и т. д., как показано в следующем листинге:

```
static string CardinalToOrdinal(int number)
{
  switch (number)
  {
    case 11:
    case 12:
    case 13:
      return $"{number}th";
    default:
      string numberAsText = number.ToString();
      char lastDigit =
        numberAsText[numberAsText.Length - 1];
      string suffix = string.Empty;
      switch (lastDigit)
      {
        case '1':
          suffix = "st";
          break;
        case '2':
          suffix = "nd";
          break;
        case '3':
          suffix = "rd";
          break;
        default:
          suffix = "th";
          break;
      }
      return $"{number}{suffix}";
  }
}

static void RunCardinalToOrdinal()
{
  for (int number = 1; number <= 40; number++)
  {
    Write($"{CardinalToOrdinal(number)} ");
  }
}
```

Обратите внимание на следующие аспекты.

- У функции `CardinalToOrdinal` только один входной параметр `number`, который должен быть типа `int`, — и один выходной: возвращаемое значение типа `string`.
- Инструкция `switch` используется для обработки особых случаев чисел 11, 12 и 13.
- Вложенная инструкция `switch` затем обрабатывает все остальные случаи: если последняя цифра числа — 1, то подставляется суффикс `st`, если 2 — то `nd`, если 3 — то `rd`; во всех остальных случаях используется суффикс `th`.
- Функция `RunCardinalToOrdinal` использует инструкцию `for`, чтобы выполнить циклический проход от `1` до `40`, вызывая функцию `CardinalToOrdinal` для каждого числа и записывая возвращенную строку в консоль, разделяя полученные порядковые числа пробелом.

В методе `Main` закомментируйте вызов метода `RunSalesTax` и вызовите метод `RunCardinalToOrdinal`, как показано в следующем листинге:

```
// RunTimesTable();
// RunSalesTax();

RunCardinalToOrdinal();
```

Запустите консольное приложение и проанализируйте результат вывода:

```
1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th 17th 18th
19th 20th 21st 22nd 23rd 24th 25th 26th 27th 28th 29th 30th 31st 32nd 33rd 34th
35th 36th 37th 38th 39th 40th
```

Вычисление факториалов с помощью рекурсии

Факториал 5 равняется 120, поскольку вычисляется путем умножения изначального числа на число, которое меньше его на 1, затем на число, меньше второго числа на 1, и т. д., пока следующий множитель не будет равен 1, например, вот так: $5 \times 4 \times \times 3 \times 2 \times 1 = 120$.

Мы напишем функцию и назовем ее `Factorial`. Она будет вычислять факториал целого числа типа `int`, переданного ей в качестве параметра. Мы также воспользуемся умной методикой под названием *«рекурсия»*, означающей, что наша функция станет вызывать саму себя.

Рекурсия — умная методика, но ее использование может привести к проблемам, например к переполнению стека из-за слишком большого количества вызовов функции. В таких языках, как C#, перебор — более практичное решение, при условии лаконичности. Более подробную информацию можно узнать, перейдя по следующей ссылке: https://en.wikipedia.org/wiki/Recursion_(computer_science)#Recursion_versus_iteration.

Добавьте функцию `Factorial`, как показано в следующем листинге:

```
static int Factorial(int number)
{
  if (number < 1)
```

```
  {
    return 0;
  }
  else if (number == 1)
  {
    return 1;
  }
  else
  {
    return number * Factorial(number - 1);
  }
}

static void RunFactorial()
{
  Write("Enter a number: ");
  if (int.TryParse(ReadLine(), out int number))
  {
    WriteLine(
      $"{number:N0}! = {Factorial(number):N0}");
  }
  else
  {
    WriteLine("You did not enter a valid number!");
  }
}
```

Обратите внимание на следующие моменты.

- Если введенное число равно нулю или меньше его, то функция `Factorial` вернет `0`.
- В случае введения числа `1` функция `Factorial` вернет `1` и, таким образом, прекратит вызывать себя.
- Если введенное число больше единицы, то функция `Factorial` умножает его на результат вызова самой себя при передаче параметра, меньшего на единицу. Такой подход делает функцию рекурсивной.
- Функция `RunFactorial` приглашает пользователя ввести число, вызывает функцию `Factorial` и затем выводит результат, отформатированный с помощью кода `N0`. Данный формат подразумевает вывод чисел без дробных разрядов с отделением тысяч запятыми.

В методе `Main` закомментируйте вызов метода `RunCardinalToOrdinal` и вызовите метод `RunFactorial`.

Запустите консольное приложение несколько раз, вводя разные числа, и проанализируйте результат вывода:

```
Enter a number: 3
3! = 6
Enter a number: 5
5! = 120
Enter a number: 31
31! = 738,197,504
Enter a number: 32
32! = -2,147,483,648
```

Факториалы принято записывать так: 5!; таким образом, выражение 5! = 120 читается как факториал пяти равен ста двадцати. Факториал — хорошее название для данной операции, поскольку факториалы увеличивают число просто в промышленных масштабах. Как видно по предыдущему выводу, факториал числа 32 и больше вызовет переполнение типа int ввиду того, что эти числа слишком велики.

Отладка приложений в процессе разработки

В этом разделе вы узнаете, как отлаживать проблемы, возникающие во время разработки.

Создание приложения с умышленной ошибкой

Создайте новый проект консольного приложения `Debugging`.

Измените шаблонный код так, как показано ниже:

```
using static System.Console;

namespace Debugging
{
  class Program
  {
    static double Add(double a, double b)
    {
      return a * b; // преднамеренная ошибка!
    }

    static void Main(string[] args)
    {
      double a = 4.5; // или используйте var
      double b = 2.5;
      double answer = Add(a, b);
      WriteLine($"{a} + {b} = {answer}");
      ReadLine(); // ожидает от пользователя нажатия ENTER
    }
  }
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
4.5 + 2.5 = 11.25
```

В этом коде обнаружен следующий баг: 4,5 плюс 2,5 должно в результате давать 7, но никак не 11,25!

Мы воспользуемся инструментами отладки в Visual Studio 2017 и Visual Studio Code, чтобы справиться с этой ошибкой.

Установка точек останова

Точки останова (breakpoints) позволяют помечать строки кода, на которых следует приостановить выполнение программы, чтобы найти ошибки. Щелкните на открывающей фигурной скобке в начале метода `Main` и нажмите клавишу F9.

В левой части окна на полях появится красный кружок, указывающий на то, что была установлена точка останова (рис. 4.1).

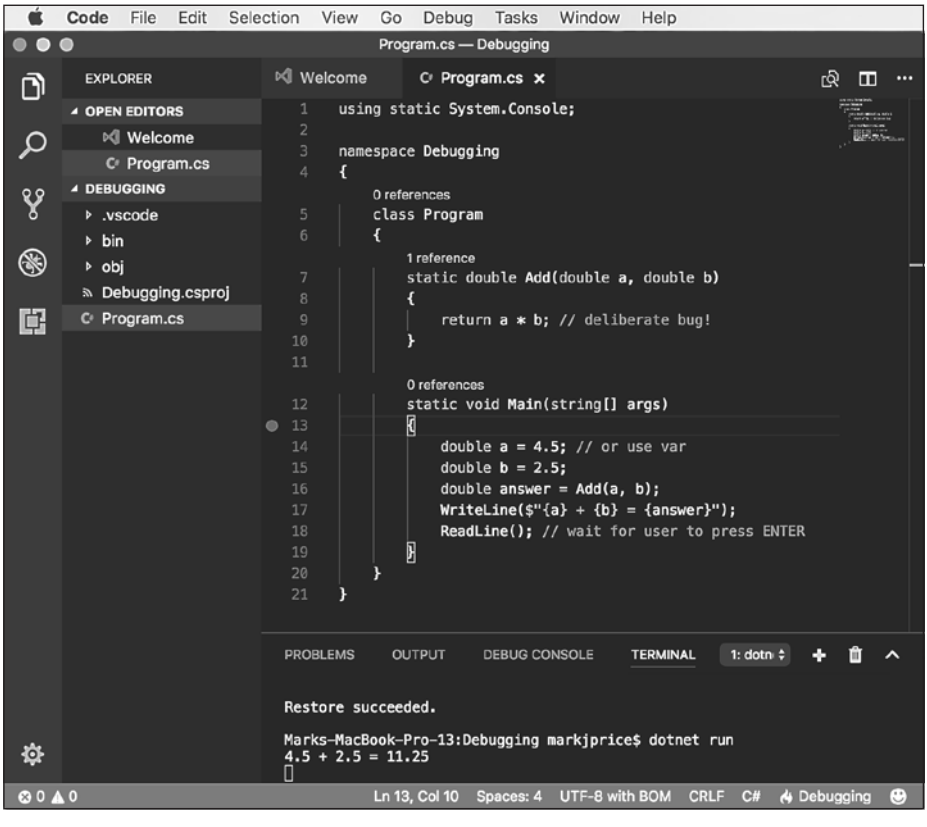


Рис. 4.1

Точки останова можно циклично устанавливать/удалять нажатием клавиши F9. Для тех же целей можно щелкнуть на левом поле. А щелчок на точке останова правой кнопкой мыши вызовет дополнительные команды, помимо удаления, например выключение или изменение параметров данной точки.

В Visual Studio 2017 выполните команду Debug ▸ Start Debugging (Отладка ▸ Начать отладку) или нажмите клавишу F5.

В Visual Studio Code выполните команду View ▸ Debug (Вид ▸ Отладка) или нажмите сочетание клавиш Shift+Cmd+D, далее нажмите кнопку Start Debugging (Начать отладку) или клавишу F5.

Visual Studio запустит консольное приложение, а затем приостановит выполнение при обнаружении точки останова. Это так называемый *режим прерывания*. Строка кода, которая будет выполнена следующей, подсвечивается желтым цветом, и на нее указывает стрелка такого же цвета, размещенная на поле (рис. 4.2).

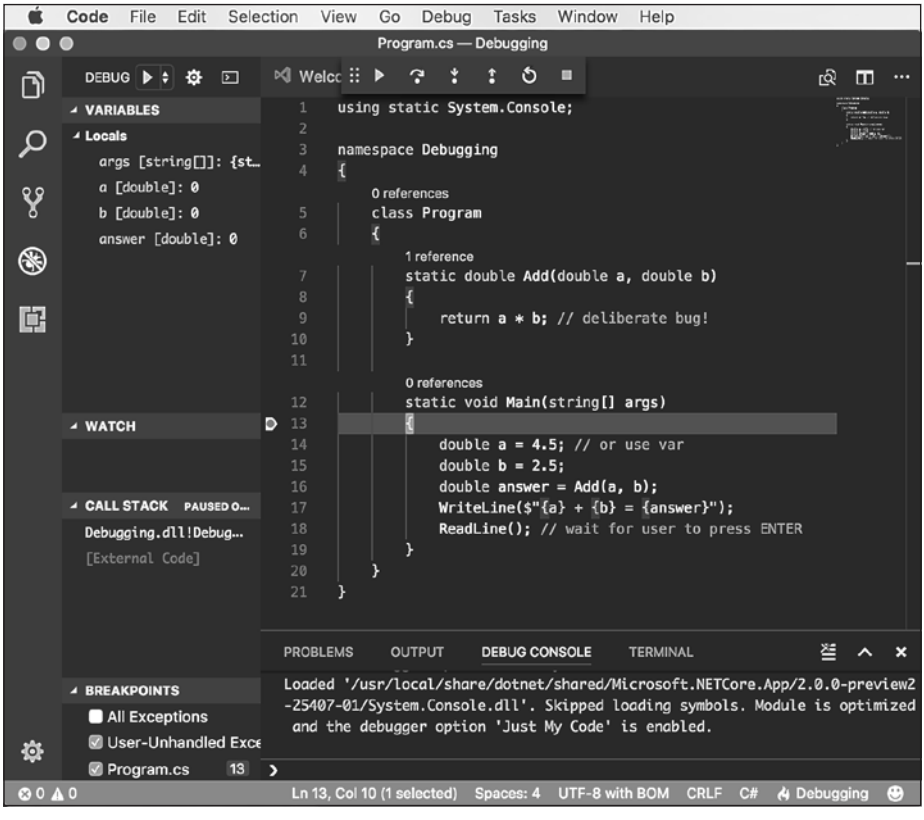


Рис. 4.2

Панель Debugging

Visual Studio 2017 содержит специальную панель с дополнительными элементами управления, упрощающими доступ к средствам отладки. Ниже перечислены некоторые из этих элементов.

- Continue (Продолжить) (F5) (зеленый треугольник) — запускает код с текущей позиции на полной скорости.
- Stop Debugging (Остановить отладку) (Shift+F5) (красный квадрат) — прекращает выполнение программы.
- Restart (Перезапустить) (Ctrl или Cmd+Shift+F5) (круглая черная стрелка) — завершает выполнение и сразу вновь запускает программу.

- Step Into (Шаг с заходом) (F11), Step Over (Шаг с обходом) (F10) и Step Out (Шаг с выходом) (Shift+F11) (синяя стрелка над точкой) — различными способами пошагово выполняют код.

  На рис. 4.3 показана панель Debugging (Отладка) в Visual Studio 2017.

Рис. 4.3

На рис. 4.4 показана панель Debugging (Отладка) в Visual Studio Code.

Рис. 4.4

Дополнительные панели отладки

Visual Studio 2017 во время отладки отображает дополнительные панели, с помощью которых вы можете отслеживать полезную информацию, например переменные, при пошаговом выполнении кода. Если вы не видите нужную панель, то в Visual Studio 2017 раскройте меню Debug ▸ Windows (Отладка ▸ Окна) и выберите необходимую панель (рис. 4.5).

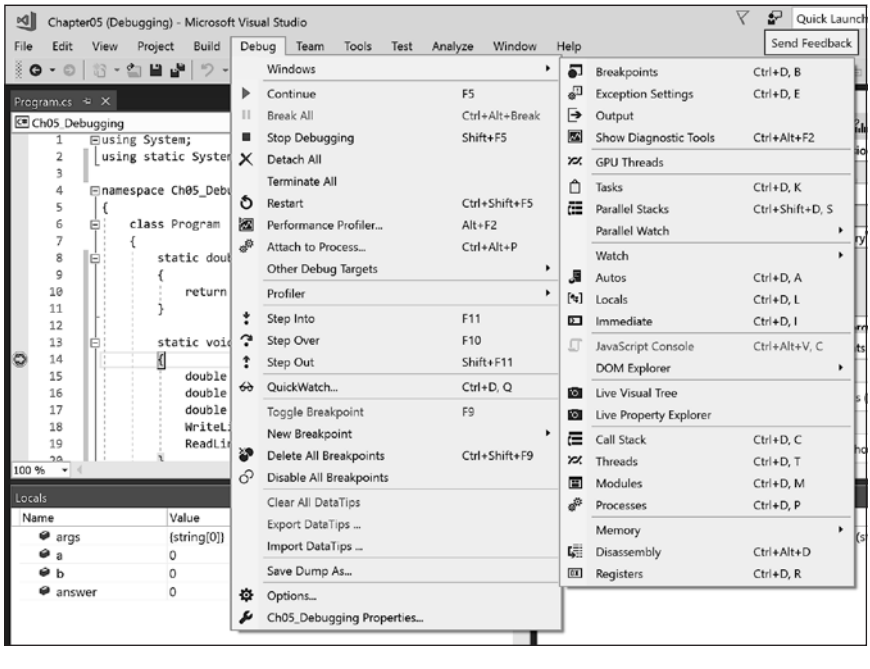


Рис. 4.5

Многие из панелей отладки доступны только в режиме прерывания.

В Visual Studio Code аналогичные панели в режиме отладки отображаются в левой части окна программы, как показано на рис. 4.4 выше.

На панели Locals (Локальные) в Visual Studio 2017 и Visual Studio Code отображаются имена, значения и типы всех используемых локальных переменных (рис. 4.6). Отслеживайте содержимое этой панели, когда отлаживаете свой код.

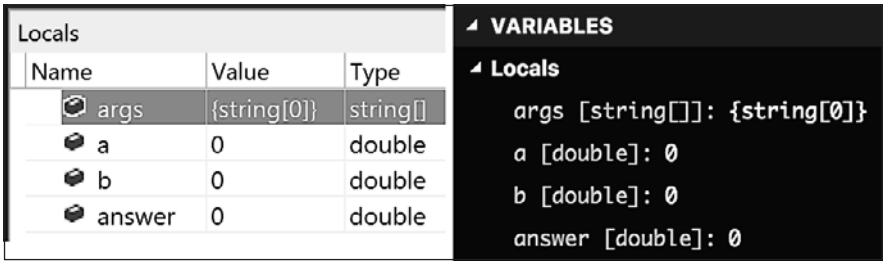


Рис. 4.6

В главе 1 я познакомил вас с панелью C# Interactive (Интерактивный C#). Аналогичные, но упрощенные панели Immediate (Интерпретация) в Visual Studio 2017 и DEBUG CONSOLE (Консоль отладки) в Visual Studio Code также позволяют работать с кодом в реальном времени.

На этих панелях можно задать вопрос и получить ответ. К примеру, чтобы спросить: «Сколько будет 1 + 2?» — надо набрать `1+2` и нажать клавишу Enter (рис. 4.7).

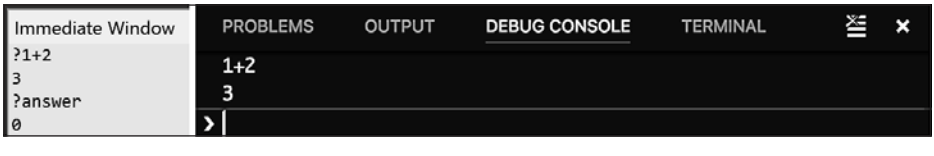


Рис. 4.7

Пошаговое выполнение кода

В Visual Studio 2017 выполните команду Debug ▸ Step Into (Отладка ▸ Шаг с заходом) или в обеих средах нажмите кнопку Step Into (Шаг с заходом) на панели или клавишу F11.

Желтая подсветка переместится на одну строку кода (рис. 4.8).

Выполните команду Debug ▸ Step Over (Отладка ▸ Шаг с обходом) или нажмите клавишу F10. Желтая подсветка также переместится на одну строку кода. На данный момент вы не видите разницы между командами Step Into (Шаг с заходом) или Step Over (Шаг с обходом).

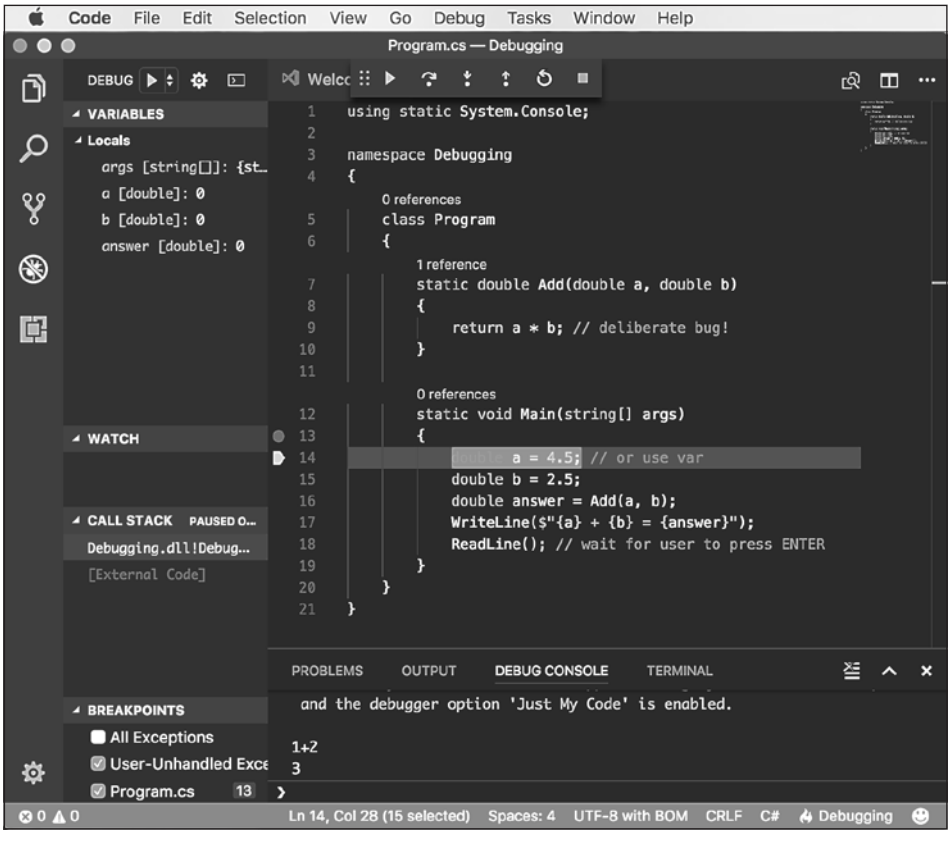


Рис. 4.8

Вновь нажмите клавишу F10, чтобы желтая подсветка оказалась на строке вызова метода Add (рис. 4.9).

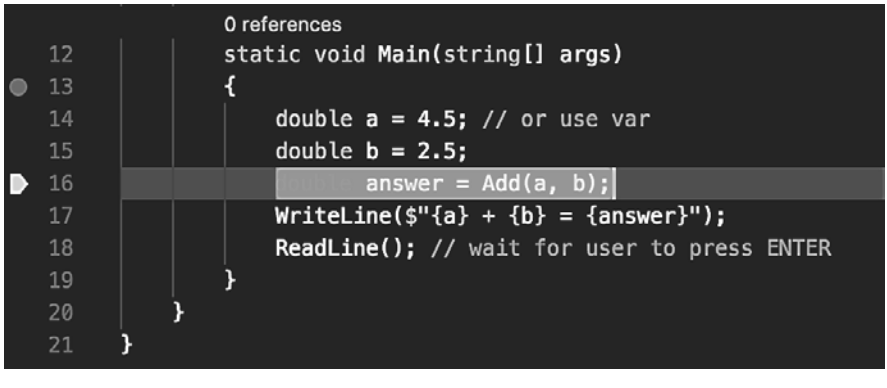


Рис. 4.9

Разницу между командами Step Into (Шаг с заходом) и Step Over (Шаг с обходом) можно увидеть, если следующая инструкция выполняет вызов метода. После нажатия кнопки Step Into (Шаг с заходом) отладчик выполнит *только* сам вызов, а затем остановит выполнение в первой строке кода метода. В случае нажатия кнопки Step Over (Шаг с обходом) отладчик выполнит метод *целиком*, а затем остановит выполнение в первой строке, расположенной вне метода!

Нажмите кнопку Step Into (Шаг с заходом), чтобы перейти к методу. Если вы пользуетесь средой разработки Visual Studio 2017, то установите указатель мыши на оператор умножения (*). Появится всплывающая подсказка, в которой указано: этот оператор перемножает a и b, чтобы выдать результат 11,25. Итак, ошибка обнаружена. Можно закрепить всплывающую подсказку, нажав кнопку в виде булавки (рис. 4.10).

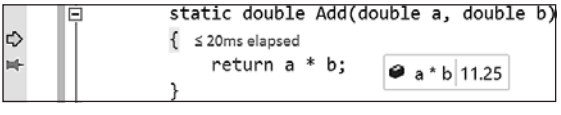


Рис. 4.10

В Visual Studio Code отсутствуют функции отображения и закрепления подсказок при наведении указателя мыши, но мышь можно наводить на переменные. Если навести указатель мыши на параметр a или b, то появится всплывающая подсказка с текущим значением.

Исправьте баг, сменив оператор * на +.

Теперь нужно остановить, перекомпилировать и перезапустить приложение, поэтому нажмите кнопку остановки или сочетание клавиш Shift+F5.

Если вы перезапустите консольное приложение, то обнаружите, что теперь вычисление производится корректно.

Настройка точек останова

Щелчок на точке останова правой кнопкой мыши позволит получить доступ к дополнительным командам, например Conditions (Условия) (рис. 4.11).

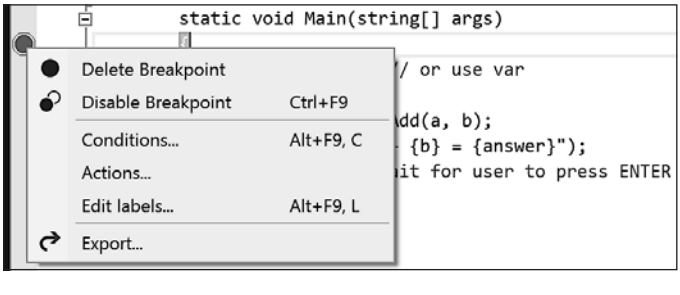


Рис. 4.11

Условия для точки останова включают выражение, которое должно быть истинным, и количество попаданий для применения точки останова.

В примере (рис. 4.12) я определил использование точки останова только в том случае, если оба условия выполнены: значение переменной `answer` превышает `9` и точка останова попалась три раза.

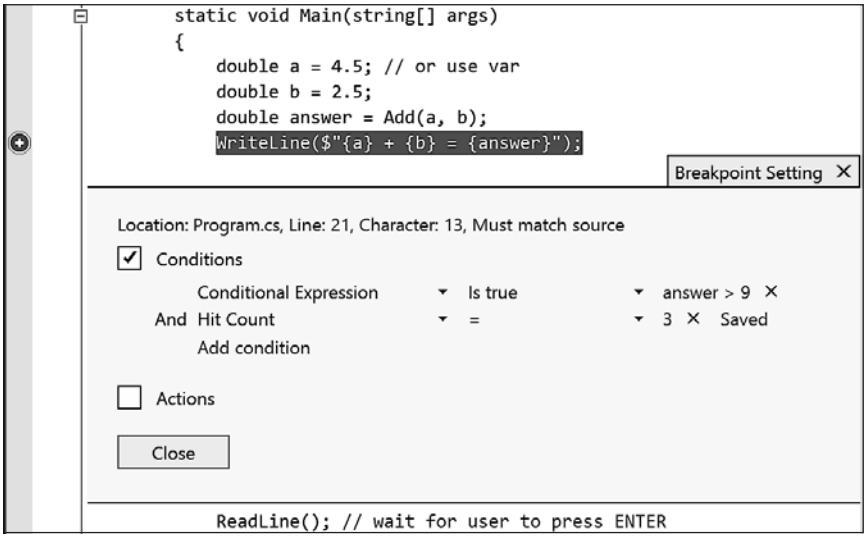


Рис. 4.12

Среда разработки Visual Studio имеет похожие, но более ограниченные возможности настройки.

Теперь вы исправили ошибку, используя некоторые средства отладки среды разработки Visual Studio.

Ведение журнала во время разработки и выполнения

Когда будете уверены, что избавились от всех ошибок в вашем коде, можно будет скомпилировать релизную версию и распространить приложение, чтобы люди могли начать им пользоваться. Однако не бывает так, чтобы код не содержал ошибок в принципе, и во время выполнения могут возникать неожиданные ошибки.

Пользователей объединяет одно заметное сходство: они очень плохо запоминают действия, которые выполняли, когда произошла ошибка, поэтому не стоит надеяться на то, что они предоставят вам точную информацию, которая поможет исправить проблему.

Как следствие, рекомендуется добавлять на разные участки приложения код, который бы протоколировал, какие действия выполняются, особенно когда происходят ошибки, ведь таким образом можно будет просмотреть эти журналы и воспользоваться ими для поиска причины и устранения проблемы.

Существует два типа, позволяющих добавить в программу простое протоколирование: `Debug` и `Trace`. Первый используется для записей в журнал во время разработки, тогда как второй применяется для записей в журнал и при разработке, и при выполнении.

Реализация прослушивателей Debug и Trace

Вы уже видели использование типа `Console` и его метода `WriteLine` для печати вывода в окно консоли. У нас также есть два типа, `Debug` и `Trace`, которые отличает бóльшая гибкость в плане того, куда они могут печатать свой вывод.

Классы `Debug` и `Trace` способны записывать в любой *прослушиватель трассировки*. Таковым является особый тип, который можно настроить так, чтобы он при вызове метода `Trace.WriteLine` направлял свой вывод в удобное для вас место. В .NET Core реализовано несколько прослушивателей трассировки, кроме того, вы можете создать и собственный при условии, что он будет наследовать от типа `TraceListener`.

Запись в прослушиватель трассировки по умолчанию

Один прослушиватель трассировки — класс `DefaultTraceListener` — настроен автоматически и ведет запись в область Output (Вывод) среды Visual Studio 2017 или в панель Debug (Отладка) среды Visual Studio Code. Другие прослушиватели можно настроить вручную с помощью кода.

Создайте новый проект консольного приложения с именем `Instrumenting`.

Измените код шаблона, как показано в листинге ниже:

```
using System.Diagnostics;

namespace Instrumenting
{
  class Program
  {
    static void Main(string[] args)
    {
      Debug.WriteLine("Debug says, I am watching!");
      Trace.WriteLine("Trace says, I am watching!");
    }
  }
}
```

Запустите консольное приложение с прикрепленным отладчиком.

В области Output (Вывод) в Visual Studio 2017 вы увидите два сообщения. Если вы ее не видите, то нажмите сочетание клавиш Ctrl+W+O или выполните команду

View ▸ Output (Вид ▸ Вывод). Убедитесь в том, что в раскрывающемся списке Show output from (Показать выходные данные из) у вас выбрано значение Debug (Отладка) (рис. 4.13).

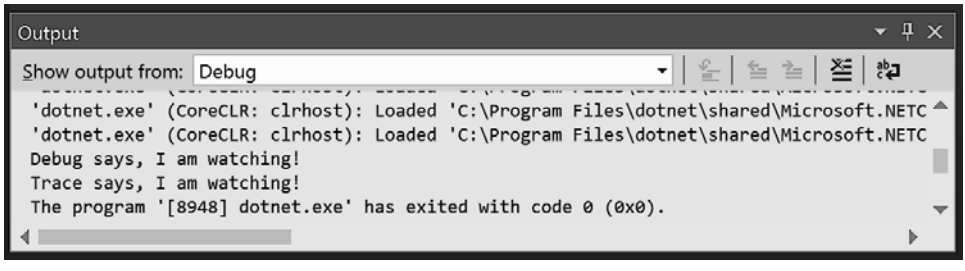


Рис. 4.13

В Visual Studio Code на панели DEBUG CONSOLE (Консоль отладки) будут выведены те же самые два сообщения, а также другая информация отладки, например загруженные DLL сборки (рис. 4.14).

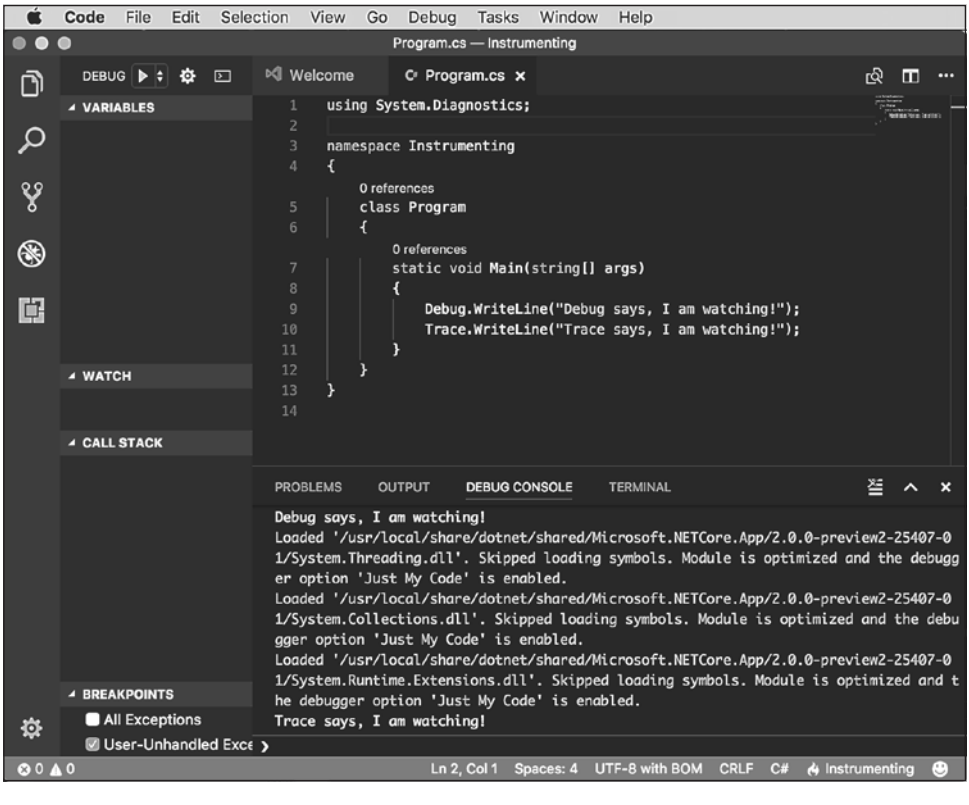


Рис. 4.14

Настройка прослушивателей трассировки

Теперь мы настроим другой прослушиватель трассировки, который будет выполнять запись в текстовый файл.

Измените код шаблона следующим образом:

```
using System.Diagnostics;
using System.IO;

namespace Instrumenting
{
  class Program
  {
    static void Main(string[] args)
    {
      // запись в текстовый файл в папке проекта
      Trace.Listeners.Add(new TextWriterTraceListener(
      File.CreateText("log.txt")));

      // модуль записи текста буферизован, поэтому данная опция вызывает
      // Flush() для всех прослушивателей после записи
      Trace.AutoFlush = true;
      Debug.WriteLine("Debug says, I am watching!");
      Trace.WriteLine("Trace says, I am watching!");
    }
  }
}
```

Запустите консольное приложение без отладки и откройте файл `log.txt` (рис. 4.15).

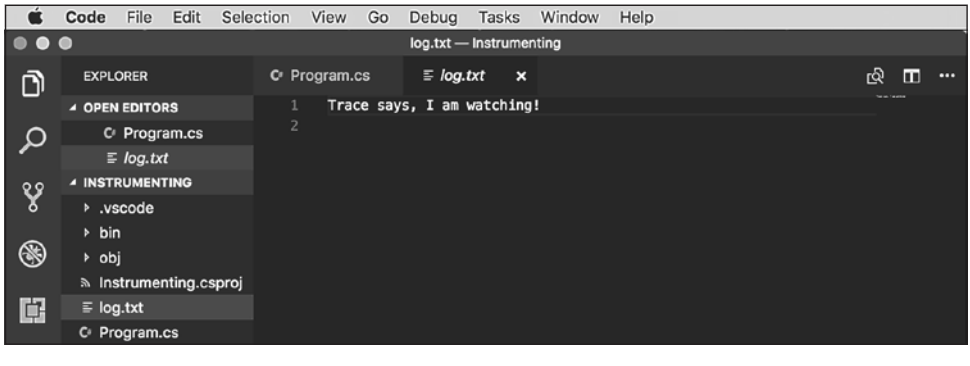


Рис. 4.15

При отладке активизируются оба класса Debug и Trace и отображают свой вывод в области OUTPUT (Вывод) или DEBUG CONSOLE (Консоль отладки). При запуске без отладки отображается только вывод класса Trace. Таким образом, вы свободно можете вызывать метод Debug.WriteLine по всему коду, зная, что он будет удален при компиляции релизной версии вашего приложения.

Переключение уровней трассировки

Вызовы метода `Trace.WriteLine` остаются в коде вашей программы даже после релиза. Так что необходимо найти способ контролировать момент, когда они отображают свой вывод. Это можно сделать с помощью переключателя трассировки.

Значение переключателя трассировки может быть установлено с использованием цифры или слова. Например, число `3` заменяется словом `Info`, как показано в табл. 4.1.

Таблица 4.1

Число	Слово	Описание
0	Off	Ничего не выводится
1	Error	Выводятся только ошибки
2	Warning	Выводятся ошибки и предупреждения
3	Info	Выводятся ошибки, предупреждения и информация
4	Verbose	Выводится вся информация

Добавьте несколько инструкций в конец метода `Main`, чтобы создать переключатель трассировки, установите его уровень, используя переданный параметр командной строки, а затем выведите четыре уровня переключателя трассировки, как показано в следующем листинге:

```
var ts = new TraceSwitch("PacktSwitch", "This switch is set via a command line
argument.");

if (args.Length > 0)
{
  if (System.Enum.TryParse<TraceLevel>(args[0],
    ignoreCase: true, result: out TraceLevel level))
  {
    ts.Level = level;
  }
}

Trace.WriteLineIf(ts.TraceError, "Trace error");
Trace.WriteLineIf(ts.TraceWarning, "Trace warning");
Trace.WriteLineIf(ts.TraceInfo, "Trace information");
Trace.WriteLineIf(ts.TraceVerbose, "Trace verbose");
```

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) введите команду `dotnet run info` для запуска консольного приложения и запроса вывода вплоть до уровня `Info (3)`, а затем откройте файл `log.txt` (рис. 4.16).

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) введите команду `dotnet run 1` для запуска консольного приложения и запроса вывода вплоть до уровня `Error (1)`, а затем откройте файл `log.txt` и обратите внимание, что в этот раз из всех четырех уровней распечатан только вывод `Trace error`.

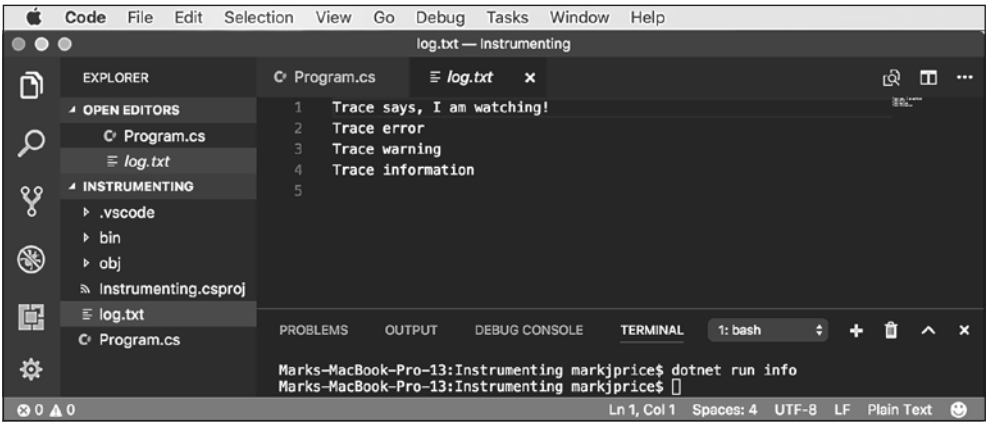


Рис. 4.16

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте Instrumenting и в контекстном меню выберите пункт Properties (Свойства). Перейдите на вкладку Debug (Отладка) и введите слово `info` в поле Application arguments (Аргументы приложения) (рис. 4.17).

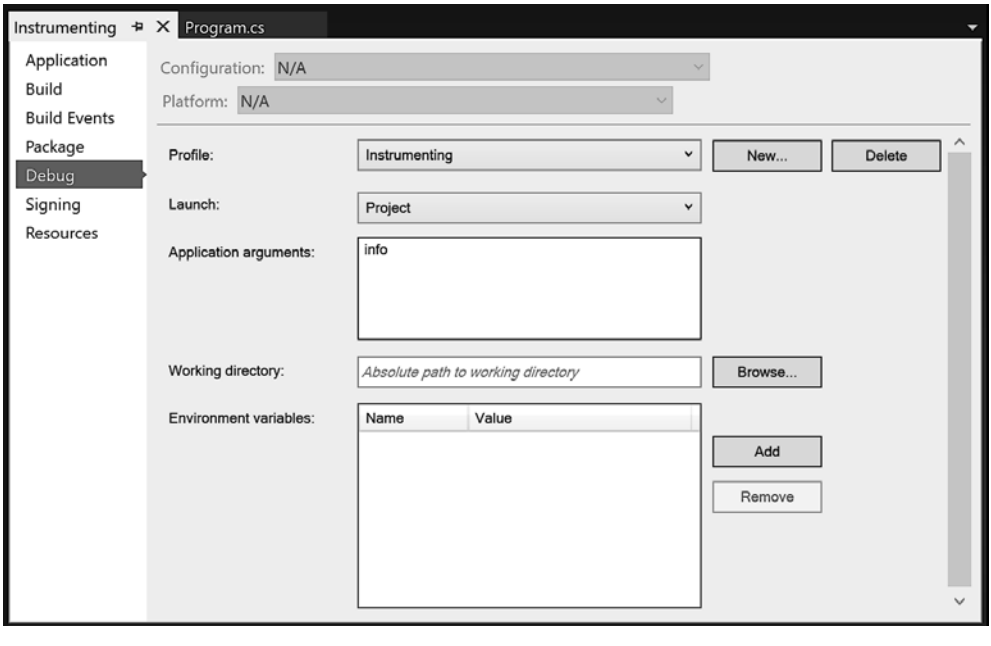


Рис. 4.17

Нажмите сочетание клавиш Ctrl+F5 для запуска консольного приложения и откройте файл `log.txt` (рис. 4.18).

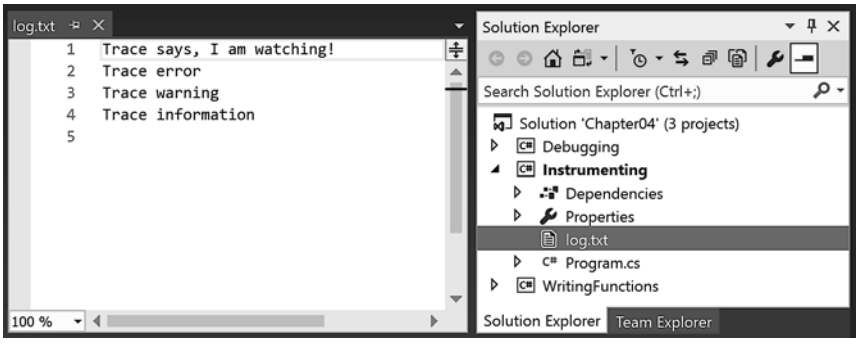


Рис. 4.18

Если не передано ни одного аргумента, то уровень переключателя трассировки по умолчанию установлен на Off (0), а это значит, что ничего не выводится.

Функции модульного тестирования

Исправление ошибок в коде стоит немалых денег, поэтому чем раньше ошибка будет найдена, тем дешевле обойдется ее исправление. Модульное тестирование — отличный способ поиска ошибок еще на этапе проектирования. Некоторые разработчики даже следуют такому принципу: программисты должны создать тесты для модулей прежде, чем эти модули будут написаны. Данный подход называется *разработкой через тестирование* (test driven development, TDD).

Более подробную информацию о модульном тестировании можно узнать, перейдя по ссылке https://docs.microsoft.com/en-us/dotnet/core/testing/. Детальная информация о TDD доступна на сайте https://en.wikipedia.org/wiki/Test-driven_development.

Microsoft использует собственный фреймворк модульного тестирования, известный как MS Test, который тесно интегрирован в среду разработки Visual Studio. Но в целях модульного тестирования мы будем использовать фреймворк сторонних разработчиков, совместимый с .NET Core, — xUnit.net.

Создание библиотеки классов для тестирования в Visual Studio 2017

В Visual Studio 2017 в Class Library (.NET Standard) (Библиотека классов (.NET Standard)) создайте проект `CalculatorLib` (рис. 4.19).

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на файле `Class1.cs` и в контекстном меню выберите пункт

Rename (Переименовать). Смените имя файла на `Calculator`. Появится уведомление о переименовании связанных зависимостей. Нажмите кнопку Yes (Да).

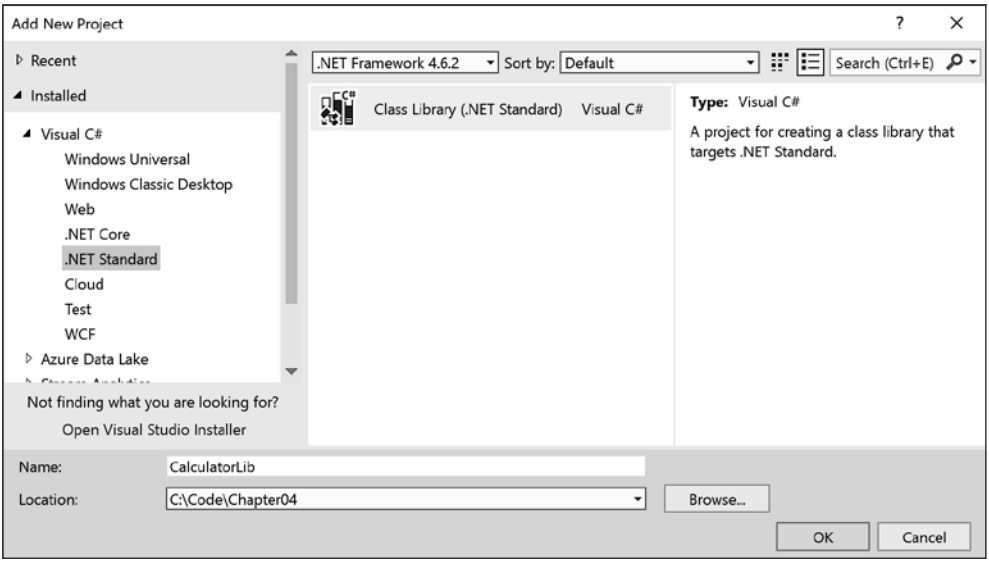


Рис. 4.19

Измените код, как показано ниже (обратите внимание на преднамеренную ошибку!):

```
namespace Packt.CS7
{
  public class Calculator
  {
    public double Add(double a, double b)
    {
      return a * b;
    }
  }
}
```

Создание проекта модульного тестирования в Visual Studio 2017

В Visual Studio 2017 в xUnit Test Project (.NET Core) (Тестовый проект xUnit (.NET Core)) создайте проект `CalculatorLibUnitTests` (рис. 4.20).

На панели Solution Explorer (Обозреватель решений) в проекте CalculatorLibUnitTests щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Add Reference (Добавить ссылку). В появившемся окне Reference Manager (Дис-

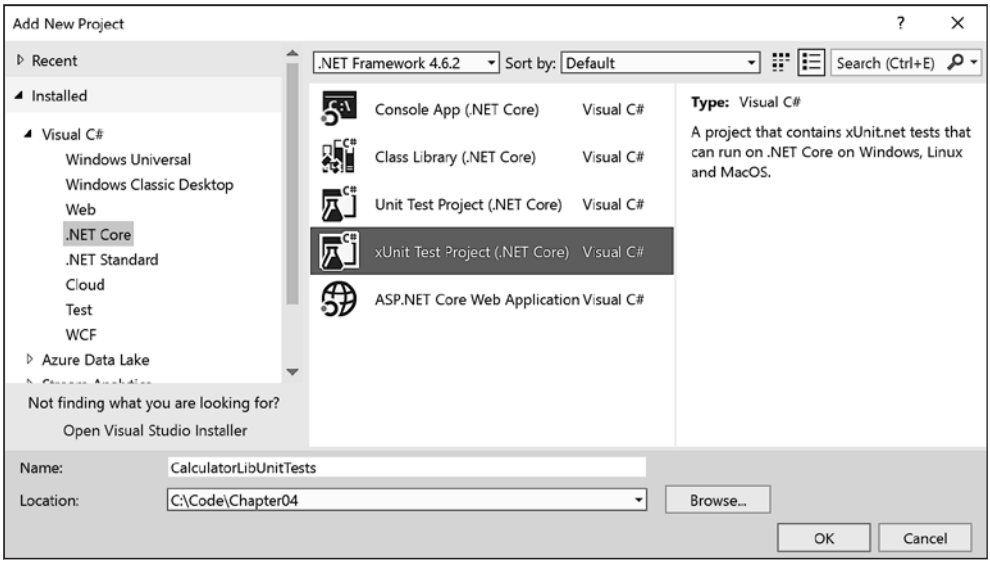


Рис. 4.20

петчер ссылок) установите флажок для `CalculatorLib`, а затем нажмите кнопку OK (рис. 4.21).

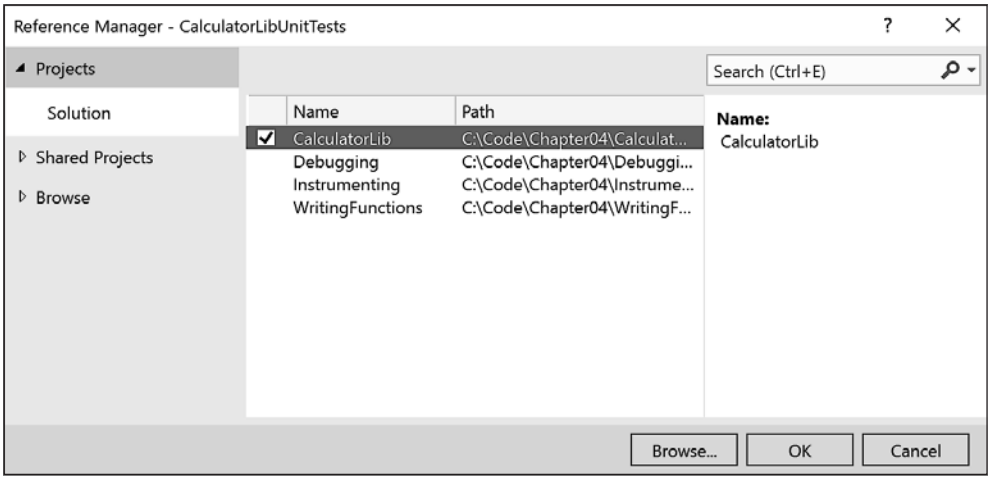


Рис. 4.21

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на файле `UnitTest1.cs` и в контекстном меню выберите пункт Rename (Переименовать). Укажите новое имя — `CalculatorUnitTests`. Нажмите кнопку Yes (Да), когда появится предупреждение.

Создание библиотеки классов для тестирования в Visual Studio Code

В каталоге `Chapter04` создайте подкаталог `CalculatorLib` и вложите в него подкаталог `CalculatorLibUnitTests`.

В Visual Studio Code откройте каталог `CalculatorLib` и введите следующую команду на панели Integrated Terminal (Интегрированный терминал):

```
dotnet new classlib
```

Переименуйте файл `Class1.cs`, присвоив ему имя `Calculator.cs`, и измените код, как показано ниже (обратите внимание на преднамеренную ошибку!):

```
namespace Packt.CS7
{
  public class Calculator
  {
    public double Add(double a, double b)
    {
      return a * b;
    }
  }
}
```

Введите следующие команды на панели Integrated Terminal (Интегрированный терминал):

```
dotnet build
```

Откройте каталог `CalculatorLibUnitTests` и введите такую команду на панели Integrated Terminal (Интегрированный терминал):

```
dotnet new xunit
```

Щелкните на файле `CalculatorLibUnitTests.csproj` и измените код так, чтобы добавить группу элементов с ссылкой на проект CalculatorLib, как показано полужирным шрифтом в листинге ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk"
                      Version="15.5.0-preview-20171012-09" />
    <PackageReference Include="xunit" Version="2.3.0" />
    <PackageReference Include="xunit.runner.visualstudio"
                      Version="2.3.0" />
</ItemGroup>

<ItemGroup>
```

```
  <ProjectReference Include="..\CalculatorLib\CalculatorLib.csproj" />
  </ItemGroup>

</Project>
```

На сайте NuGet можно найти последнюю версию пакета Microsoft.NET.Test.Sdk: https://www.nuget.org/packages?q=Microsoft.NET.Test.Sdk.

Переименуйте файл `UnitTest1.cs` в `CalculatorUnitTests.cs`, а классу присвойте имя `CalculatorUnitTests`.

Разработка модульных тестов

В Visual Studio 2017 или Visual Studio Code откройте файл `CalculatorUnitTests.cs` и измените код так, как показано в листинге ниже:

```
using Packt.CS7;
using Xunit;

namespace CalculatorLibUnitTests
{
  public class CalculatorUnitTests
  {
    [Fact]
    public void TestAdding2And2()
    {
      // подготовка
      double a = 2;
      double b = 2;
      double expected = 4;
      var calc = new Calculator();
      // действие
      double actual = calc.Add(a, b);
      // проверка
      Assert.Equal(expected, actual);
    }
    [Fact]
    public void TestAdding2And3()
    {
      // подготовка
      double a = 2;
      double b = 3;
      double expected = 5;
      var calc = new Calculator();
      // действие
      double actual = calc.Add(a, b);
      // проверка
      Assert.Equal(expected, actual);
    }
  }
}
```

Правильно разработанный модульный тест по модели AAA (arrange, act, assert — «подготовка, действие, проверка») состоит из трех частей:

- *подготовка* — объявляет и создает экземпляры переменных для ввода и вывода;
- *действие* — выполняет тестируемый модуль;
- *проверка* — производит одну или несколько проверок результатов работы модуля.

Выполнение модульных тестов в Visual Studio 2017

Выполните команду Test ▸ Windows ▸ Test Explorer (Тест ▸ Окна ▸ Обозреватель тестов). Выполните команду Build ▸ Build Solution (Сборка ▸ Построить решение) или нажмите клавишу F6. Обратите внимание: обнаружены, но не выполнены два теста (рис. 4.22).

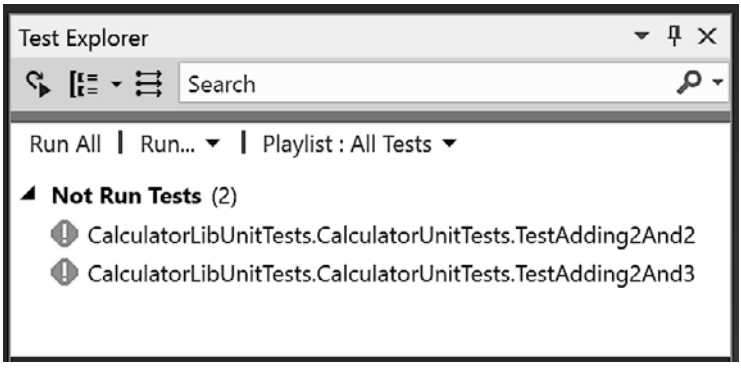


Рис. 4.22

На панели Test Explorer (Обозреватель тестов) щелкните на ссылке Run All (Запустить все).

Подождите несколько секунд, пока тесты не завершатся (рис. 4.23). Обратите внимание, что один тест был пройден успешно, а второй провален. По этой причине рекомендуется писать несколько тестов для каждого модуля.

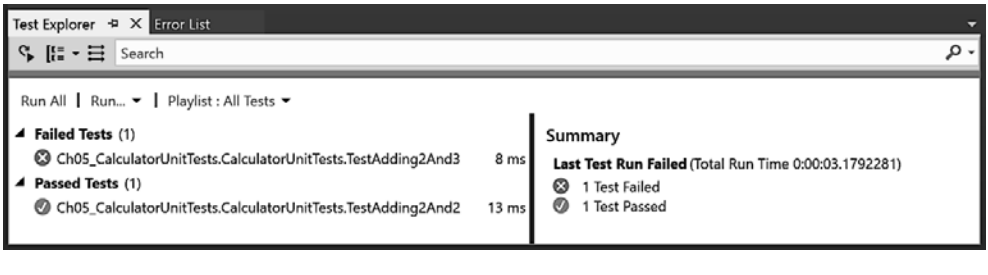


Рис. 4.23

Щелкнув на строке с тестом, вы увидите более подробную информацию, на основе которой следует попытаться диагностировать ошибку и исправить ее (рис. 4.24).

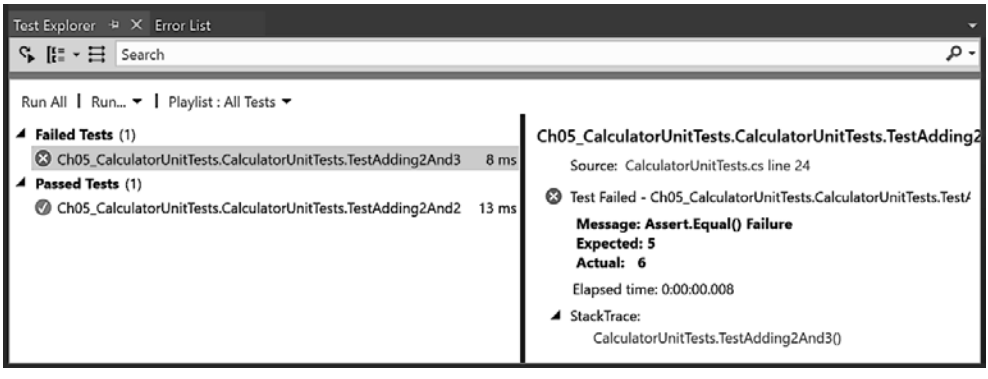


Рис. 4.24

Исправьте баг в методе `Add`, а затем перезапустите модульные тесты, чтобы убедиться в исправлении ошибки (рис. 4.25).

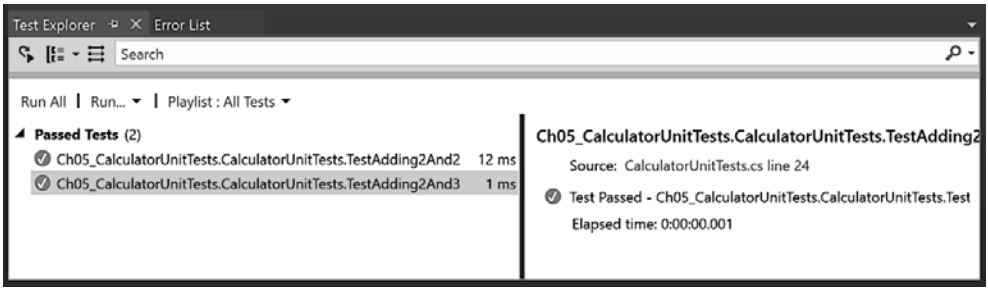


Рис. 4.25

Выполнение модульных тестов в Visual Studio Code

В Visual Studio Code откройте каталог `Chapter04`.

На панели Integrated Terminal (Интегрированный терминал) введите следующую команду.

```
cd CalculatorLibUnitTest
dotnet test
```

Вы должны увидеть следующий результат (рис. 4.26).

Исправьте баг в методе `Add`, а затем перезапустите модульные тесты, чтобы убедиться в исправлении ошибки (рис. 4.27).

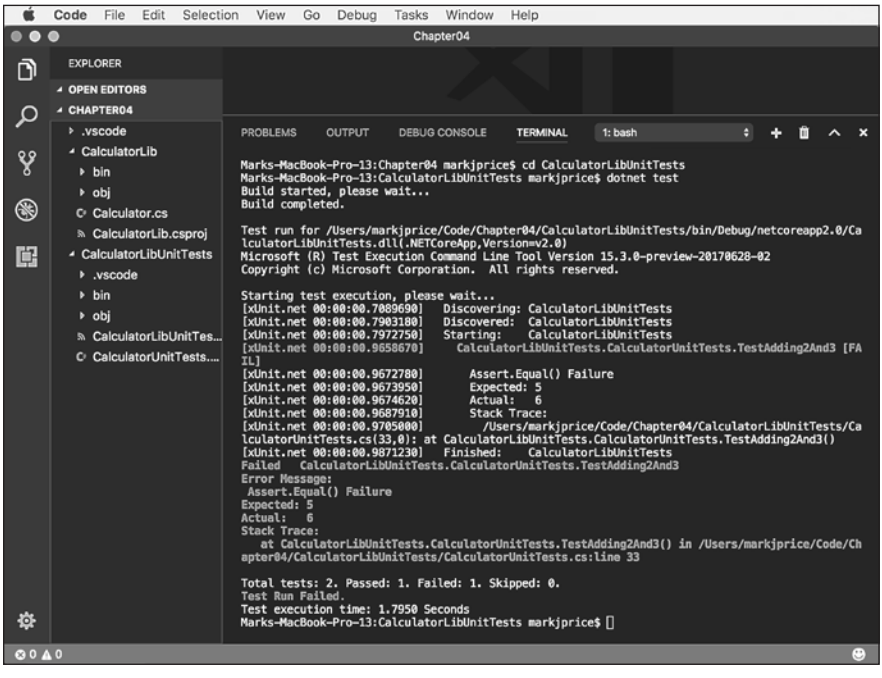


Рис. 4.26

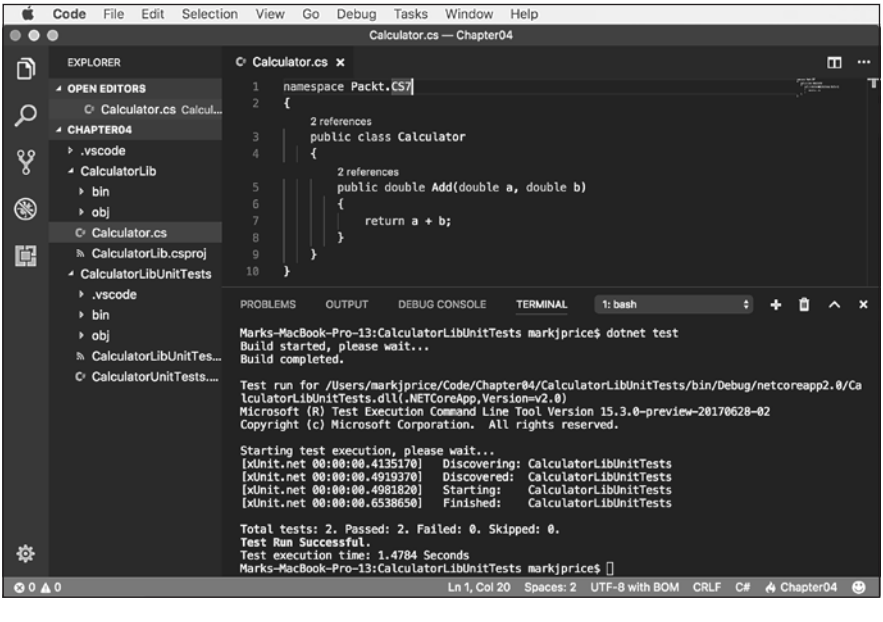


Рис. 4.27

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенное упражнение и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

1. Для чего в языке C# используется ключевое слово `void`?
2. Сколько параметров может быть у метода в языке C#?
3. Чем отличается назначение клавиши F5 и сочетаний клавиш Ctrl+F5, Shift+F5 и Ctrl+Shift+F5 при использовании Visual Studio 2017?
4. Куда производится запись вывода метода `Trace.WriteLine`?
5. Каковы пять уровней трассировки?
6. Чем отличаются классы `Debug` и `Trace`?
7. Как расшифровывается принцип AAA в модульном тестировании?
8. Каким атрибутом следует сопроводить методы тестирования при написании теста с помощью xUnit?
9. Какая команда `dotnet` отвечает за запуск тестов xUnit?
10. Что такое TDD?

Упражнение 4.1. Отладка и модульное тестирование

Простые множители — сочетание наименьших простых чисел, которые при умножении вместе произведут исходное число. Рассмотрим следующие примеры:

- простые множители 4: 2×2;
- простые множители 30: $2 \times 3 \times 5$;
- простые множители 40: $2 \times 2 \times 2 \times 5$;
- простые множители 50: $2 \times 5 \times 5$.

Создайте консольное приложение с именем `Exercise02`, которое с помощью метода `PrimeFactors` при передаче целочисленной переменной в качестве параметра возвращает строку с указанием ее простых множителей. Используйте средства отладки и модульного тестирования, чтобы убедиться, что ваша функция работает правильно с различными числами и возвращает корректный вывод.

Дополнительные ресурсы

- Отладка в Visual Studio Code: https://code.visualstudio.com/docs/editor/debugging.
- Пространство имен `System.Diagnostics`: https://docs.microsoft.com/en-us/dotnet/core/api/system.diagnostics.
- Основы отладки: https://docs.microsoft.com/en-us/visualstudio/debugger/debugger-basics.
- xUnit.net: http://xunit.github.io/.

Резюме

В этой главе вы изучили, как в среде разработки Visual Studio применять средства отладки, диагностики и модульного тестирования кода.

Из следующей главы вы узнаете, как создавать пользовательские типы с помощью технологий объектно-ориентированного программирования.

5 Создание пользовательских типов с помощью объектно-ориентированного программирования

Эта глава посвящена созданию ваших собственных типов с применением *объектно-ориентированного программирования (ООП)*. Вы узнаете обо всех возможных видах членов типа, включая поля для хранения данных и методы для выполнения действий. Вы задействуете на практике концепции ООП, в частности агрегирование и инкапсуляцию. Помимо этого, вы узнаете о свойствах языка C# 7, таких как поддержка синтаксиса кортежей и переменные `out`, и о возможностях языка C# 7.1, включая вывод имен кортежей и литералы по умолчанию.

В данной главе:

- вкратце об ООП;
- сборка библиотек классов;
- хранение данных в полях;
- запись и вызов методов;
- управление передачей параметров;
- разделение классов с помощью ключевого слова partial;
- управление доступом с помощью свойств и индексаторов.

Вкратце об объектно-ориентированном программировании

Объект в реальном мире — это предмет, например, автомобиль или человек. Объект в программировании часто представляет нечто в реальном мире, скажем товар или банковский счет, но может быть и чем-то более абстрактным.

В C# используются `class` (обычно) или `struct` (редко) для определения каждого типа объекта. Тип можно представить как копию или шаблон объекта.

Концепции объектно-ориентированного программирования кратко описаны ниже.

- *Инкапсуляция* — комбинация данных и действий, связанных с объектом. Например, тип `BankAccount` может иметь такие данные, как `Balance` и `AccountName`, а также действия, скажем, `Deposit` и `Withdraw`. При инкапсуляции часто возникает необходимость контролировать то, что может получить доступ к этим действиям и данным.
- *Композиция* — то, из чего состоит объект. Например, автомобиль состоит из разных частей, таких как четыре колеса, несколько сидений, двигатель и т. д.
- *Агрегирование* определяет все, что связано с объектом. Скажем, какой-нибудь человек может сидеть на водительском сиденье, а затем стать водителем автомобиля.
- *Наследование* — многократное использование кода с помощью подклассов, производных от *базовых* классов (или *суперклассов*). Все функциональные возможности базового класса становятся доступными в производном.
- *Абстракция* — передача основной идеи объекта и игнорирование его деталей или особенностей. Абстракция — сложный баланс. Если вы сделаете класс слишком абстрактным, то его сможет наследовать большее количество классов, но возможностей использования общей функциональности станет меньше.
- *Полиморфизм* заключается в переопределении производным классом унаследованных методов для реализации пользовательского поведения.

Сборка библиотек классов

Сборки библиотек классов группируют типы в легко развертываемые модули (DLL-файлы). В главе 4 при изучении модульного тестирования вы создавали консольные приложения, содержащие ваш код. Но, чтобы он стал доступен для других проектов, его следует помещать в сборки библиотеки классов, как это делают сотрудники корпорации Microsoft.

Поместите типы, которые необходимо использовать многократно, в библиотеку классов, поддерживающую «чистый» .NET Standard. После этого данные типы станут доступны для многократного применения в проектах .NET Core, .NET Framework и Xamarin.

Создание библиотек классов в Visual Studio 2017

Запустите Microsoft Visual Studio 2017. Нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Standard. В центре диалогового окна

выберите пункт Class Library (.NET Standard) (Библиотека классов (.NET Standard)), присвойте ему имя `PacktLibrary`, укажите расположение по адресу `C:\Code`, введите имя решения `Chapter05`, а затем нажмите кнопку OK (рис. 5.1).

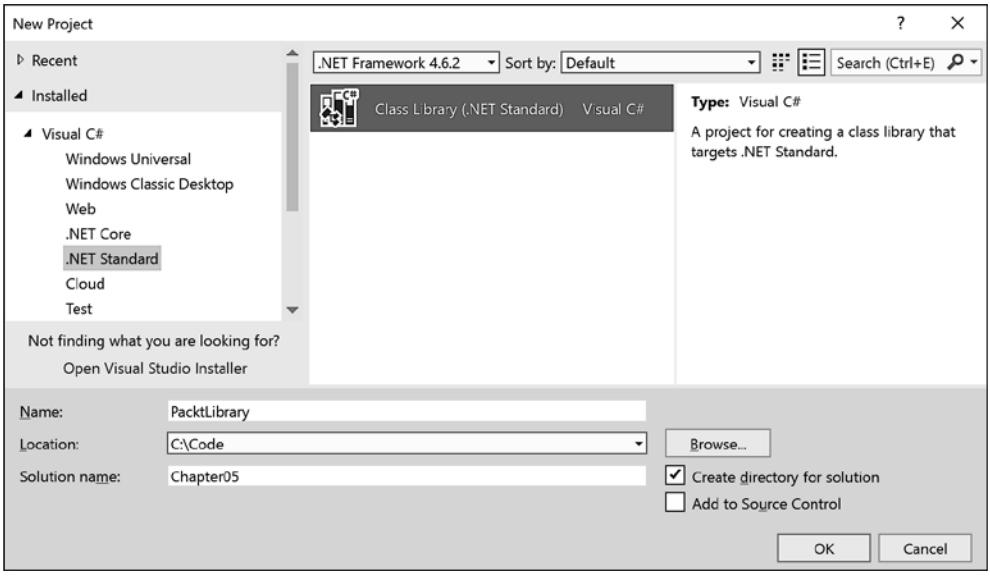


Рис. 5.1

Убедитесь, что выбрали пункт Class Library (.NET Standard) (Библиотека классов (.NET Standard)), а не Console App (.NET Core) (Консольное приложение (.NET Core))!

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на файле `Class1.cs` и в контекстном меню выберите пункт Rename (Переименовать). Присвойте ему имя `Person`. Когда появится предупреждение о том, что будут переименованы все ссылки в классе, нажмите кнопку Yes (Да).

Создание библиотек классов в Visual Studio Code

Для создания библиотеки классов в данной среде выполните следующие действия.

1. Создайте каталог `Chapter05` и добавьте в него подкаталог `PacktLibrary`.
2. Запустите Visual Studio Code и откройте каталог `PacktLibrary`.
3. На панели Integrated Terminal (Интегрированный терминал) введите такую команду:

```
dotnet new classlib
```

4. На панели EXPLORER (Проводник) переименуйте файл `Class1.cs`, присвоив ему имя `Person.cs`.
5. Щелкните на файле `Person.cs`, чтобы открыть его, восстановите пакеты и измените имя класса на `Person`.

Определение классов

В Visual Studio 2017 или Visual Studio Code измените пространство имен на `Packt.CS7`, поскольку классы важно помещать в пространства с логичными именами. В этой и следующей главах вы получите больше информации об ООП и большинстве новых возможностей языка C# версий 7.0 и 7.1.

Код вашего класса должен теперь выглядеть так, как показано ниже:

```
using System;

namespace Packt.CS7
{
  public class Person
  {
  }
}
```

Обратите внимание: ключевое слово `public` указывается перед словом `class`. Это ключевое слово называется *модификатором доступа* и позволяет всему коду получить доступ к данному классу. Если вы явно не определили доступ к нему с помощью ключевого слова `public`, то класс будет доступен только в определяющей его сборке. Нам же нужно, чтобы класс был доступен за ее пределами. У этого типа еще нет членов, инкапсулированных в него. Скоро мы их создадим.

Членами могут быть поля, методы или специализированные версии их обоих. См. описание ниже.

- *Поля* используются для хранения данных. Существует три специализированных поля:
  - *константы*, данные в которых никогда не меняются;
  - *поля, доступные только для чтения*, — данные в таких полях не могут измениться после создания экземпляра класса;
  - *события* — с их помощью определяются методы, вызываемые автоматически при возникновении определенной ситуации, например при нажатии кнопки.
- *Методы* применяются для выполнения инструкций. Примеры вы видели в главе 4. Ниже показаны четыре специализированных метода:
  - *конструкторы* выполняются при использовании ключевого слова `new` для выделения памяти и создания экземпляра класса;
  - *свойства* выполняются при необходимости получить доступ к данным;
  - *индексаторы* тоже выполняются, когда необходимо получить доступ к данным;
  - *операторы* выполняются, когда нужно применить оператор.

Создание экземпляров классов

В этом подразделе мы создадим экземпляр класса `Person`. Подобное действие известно под названием *«инстанцирование* класса».

Обращение к сборке в Visual Studio 2017

В Visual Studio 2017 создайте новый проект консольного приложения PeopleApp в существующем решении Chapter05.

Убедитесь, что выбрали пункт Console App (.NET Core) (Консольное приложение (.NET Core)), а не Class Library (.NET Standard) (Библиотека классов (.NET Standard))!

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства) или нажмите сочетание клавиш Alt+Enter. В категории Startup Project (Запускаемый проект) установите переключатель в положение Single startup project (Один запускаемый проект) и в раскрывающемся списке выберите пункт PeopleApp.

Этот проект нуждается в ссылке на библиотеку классов, которую мы только что создали.

На панели Solution Explorer (Обозреватель решений) раскройте проект PeopleApp, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Add Reference (Добавить ссылку).

В левой части появившегося диалогового окна Reference Manager — PeopleApp (Диспетчер ссылок) выберите раздел Projects ▸ Solution (Проекты ▸ Решение), выберите сборку PacktLibrary, а затем нажмите кнопку OK (рис. 5.2).

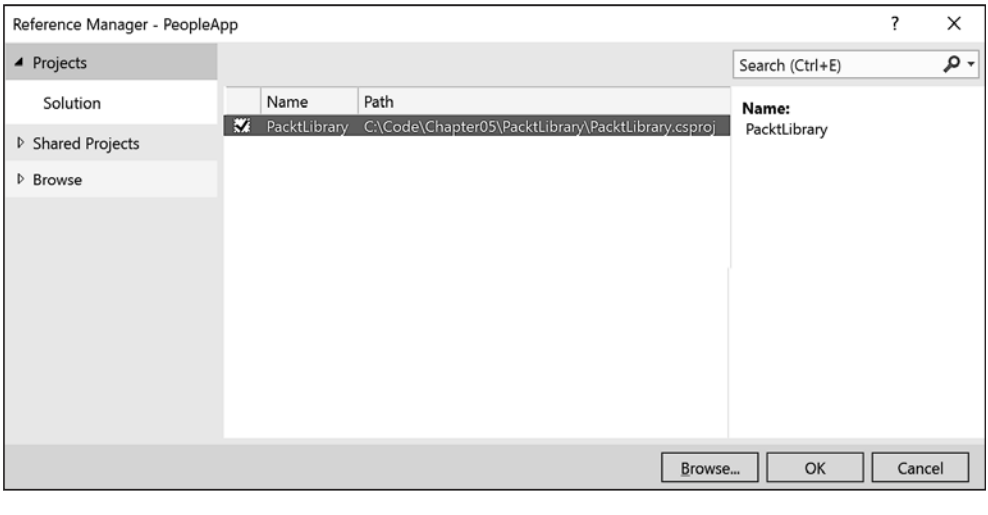


Рис. 5.2

На панели Solution Explorer (Обозреватель решений) раскройте раздел Dependencies (Зависимости), чтобы отобразить зависимость библиотеки классов от .NET Standard и зависимость консольного приложения от PacktLibrary и .NET Core (рис. 5.3).

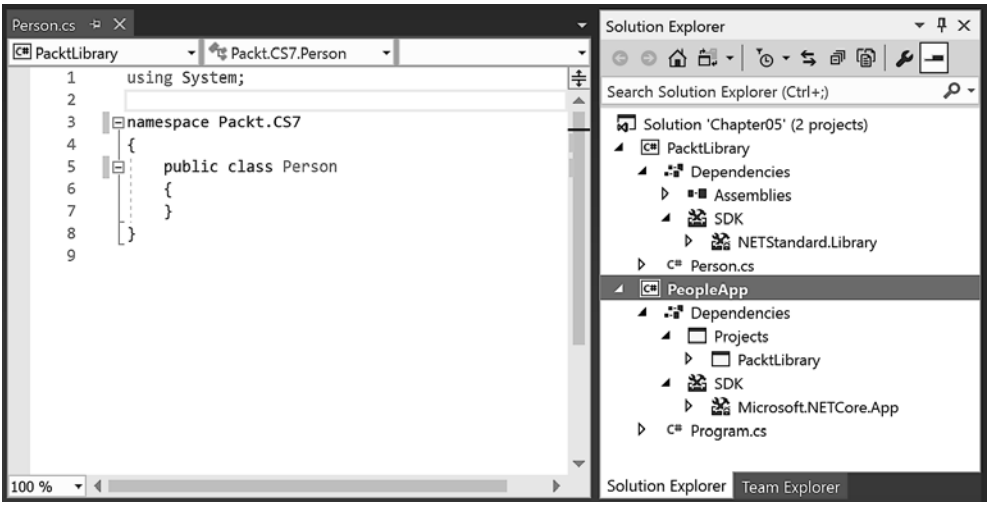


Рис. 5.3

Обращение к сборке в Visual Studio Code

Создайте подкаталог `PeopleApp` в каталоге `Chapter05`.

Откройте каталог `PeopleApp` в Visual Studio Code.

На панели Integrated Terminal (Интегрированный терминал) введите следующую команду:

```
dotnet new console
```

На панели EXPLORER (Проводник) щелкните на файле `PeopleApp.csproj` и добавьте ссылку на проект PacktLibrary, как показано в следующем листинге (добавленный код выделен полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk>

  <PropertyGroup>
    <OutputType>Exe</OutputType>
      <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference
    Include="../PacktLibrary/PacktLibrary.csproj" />
  </ItemGroup>

</Project>
```

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) введите следующие команды:

```
dotnet restore
dotnet build
```

Оба проекта, PacktLibrary и PeopleApp, будут скомпилированы в сборки DLL (рис. 5.4).

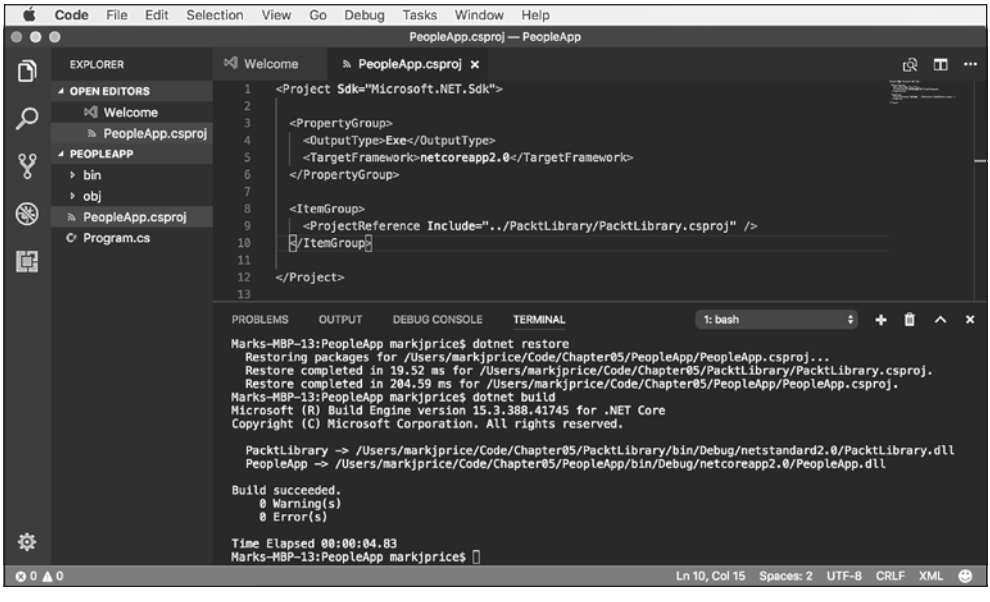


Рис. 5.4

Импорт пространства имен

В Visual Studio 2017 или Visual Studio Code добавьте в начало файла `Program.cs` код, показанный ниже. Он необходим для импорта пространства имен для нашего класса и статического импорта класса `Console`.

```
using Packt.CS7;
using static System.Console;
```

В методе `Main` напечатайте код, показанный ниже, чтобы с помощью ключевого слова `new` создать экземпляр типа `Person`. Это ключевое слово выделяет память для объекта и инициализирует любые данные в нем. Мы могли бы использовать `Person` вместо ключевого слова `var`, но благодаря последнему объем кода уменьшается при одинаковом результате.

```
var p1 = new Person();
WriteLine(p1.ToString());
```

Возможно, вы зададитесь вопросом: «Почему переменная p1 имеет метод ToString? Класс Person же пуст!» Вскоре вы на него ответите.

Запустите консольное приложение, используя сочетание клавиш Ctrl+F5 в Visual Studio 2017 или запустив `dotnet run` в Visual Studio Code, и проанализируйте результат вывода.

```
Packt.CS7.Person
```

Управление проектами в Visual Studio Code

Если требуется одновременная работа с несколькими проектами, то либо откройте новое окно, выбрав команду File ▸ New Window (Файл ▸ Новое окно) или нажав сочетание клавиш Ctrl+Cmd+N, либо откройте родительский каталог, содержащий папки проектов, с которыми хотите работать.

Решив открыть родительский каталог, будьте осторожны при выполнении команд в области TERMINAL (Терминал), поскольку они будут применяться ко всему текущему каталогу.

В Visual Studio Code откройте каталог `Chapter05`, а затем в области TERMINAL (Терминал) введите следующую команду для перехода к каталогу консольного приложения (рис. 5.5).

```
cd PeopleApp
```

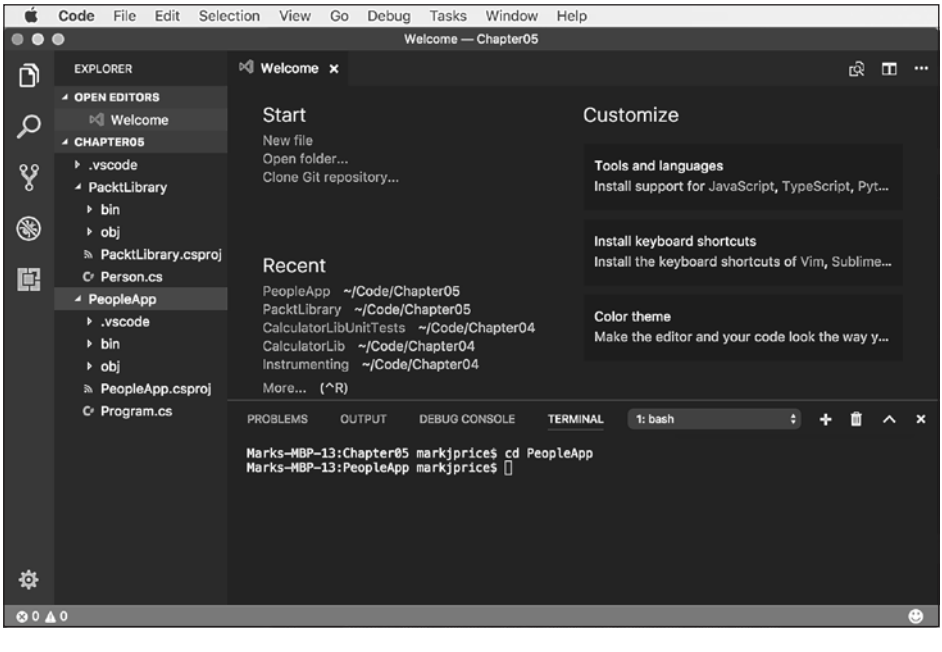


Рис. 5.5

Наследование System.Object

Хотя наш класс `Person` явно не наследуется от какого-либо типа, все типы неявно наследуются от специального типа `System.Object`. Реализация метода `ToString` в типе `System.Object` выводит полные имена пространства имен и типа, что и видно по итогам выполнения примера, который мы рассмотрели выше.

Возвращаясь к исходному классу `Person`, мы могли бы явно сообщить компилятору, что `Person` наследуется от типа `System.Object`:

```
public class Person : System.Object
```

Когда класс Б наследуется от класса А, мы говорим, что А является базовым классом, или суперклассом, а Б — производным. В нашем случае System.Object — базовый класс (суперкласс), а Person — производный.

Мы также можем использовать псевдоним типа — ключевое слово `object`:

```
public class Person : object
```

Измените класс `Person`, чтобы явно наследовать `object`. Затем установите указатель мыши внутри ключевого слова и нажмите клавишу F12 или щелкните правой кнопкой мыши на слове `object` и в контекстном меню выберите пункт Go To Definition (Перейти к определению).

Вы увидите Microsoft-определение типа `System.Object` и его членов. Вам не нужно разбираться во всем этом определении, но обратите внимание на метод `ToString` (рис. 5.6).

```
namespace System
{
    ...public class Object
    {
        ...public Object();

        ...~Object();

        ...public static bool Equals(Object objA, Object objB);
        ...public static bool ReferenceEquals(Object objA, Object objB);
        ...public virtual bool Equals(Object obj);
        ...public virtual int GetHashCode();
        ...public Type GetType();
        ...public virtual string ToString();
        ...protected Object MemberwiseClone();
    }
}
```

Рис. 5.6

Предположим, другие программисты знают, что если наследование не указано, то класс наследуется от System.Object.

Хранение данных в полях

Теперь мы определим в классе несколько полей для хранения информации о человеке.

Определение полей

В классе `Person` напишите код, показанный ниже. На данный момент информация о человеке включает имя и дату рождения. Мы инкапсулировали эти два значения в классе `Person`. Кроме того, сделали поля общедоступными, чтобы доступ к ним мог осуществляться вне самого класса.

```
public class Person : object
{
  // поля
  public string Name;
  public DateTime DateOfBirth;
}
```

Для полей можно использовать любые типы, включая массивы и коллекции, если нужно сохранить несколько значений в одном именованном поле.

В Visual Studio 2017 с помощью мыши можно перетащить вкладки открытых файлов, расположив их так, чтобы одновременно видеть оба файла, `Person.cs` и `Program.cs` (рис. 5.7).

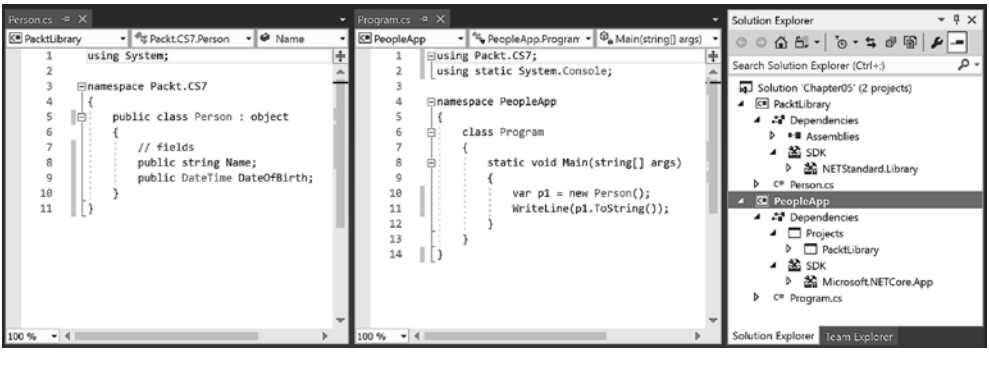


Рис. 5.7

В Visual Studio Code нажмите кнопку Split Editor (Разделить редактор) или сочетание клавиш Cmd+\ и закройте копию одного окна редактора, чтобы разместить два окна файла рядом друг с другом (рис. 5.8).

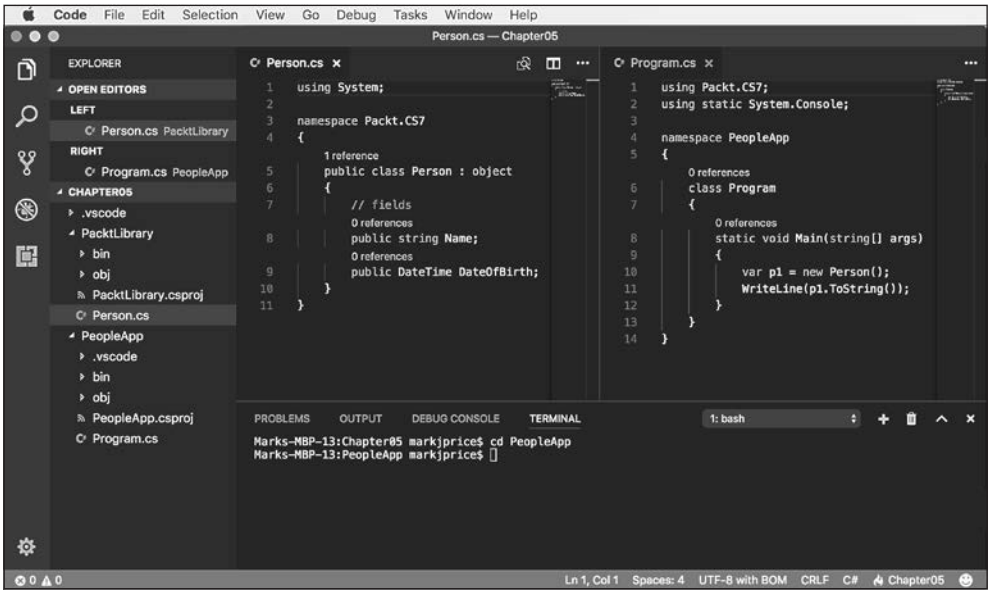


Рис. 5.8

Модификаторы доступа. Вы обратили внимание, что когда мы работали с классом, то использовали ключевое слово `public` по отношению к созданным полям? Если бы мы этого не сделали, то поля были бы закрытыми (`private`), то есть доступными только в пределах класса.

Доступны четыре ключевых слова для определения модификаторов доступа, каждое из которых можно применить к члену класса, например к полю или методу (табл. 5.1). Эта часть операции инкапсуляции отвечает за область видимости членов класса.

Таблица 5.1

Модификатор доступа	Описание
private	Доступ ограничен содержащим типом. Используется по умолчанию
internal	Доступ ограничен содержащим типом и любым другим типом в текущей сборке
protected	Доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса
internal protected	Доступ ограничен содержащим типом и любым другим типом в текущей сборке, а также типами, которые являются производными от содержащего класса
public	Неограниченный доступ

Следует явно применять один из модификаторов доступа ко всем членам типа, даже если будет использоваться модификатор по умолчанию, private. Кроме того, поля, как правило, должны сопровождаться модификатором private или protected, а для получения или установки значений полей нужно создавать свойства с модификатором public. Так код будет более управляемым.

В начало файла `Program.cs` импортируйте пространство имен `System`, если это еще не было сделано, как показано в листинге ниже:

```
using System;
```

Измените код метода `Main`, как показано в следующем листинге:

```
var p1 = new Person();
p1.Name = "Bob Smith";
p1.DateOfBirth = new System.DateTime(1965, 12, 22);
WriteLine($"{p1.Name} was born on {p1.DateOfBirth:dddd, d MMMM yyyy}");
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith was born on Wednesday, 22 December 1965
```

Инициализировать поля можно с помощью краткого синтаксиса с фигурными скобками.

Под существующим кодом добавьте код, показанный ниже, чтобы создать запись еще об одном человеке. Обратите внимание на иной формат даты рождения:

```
var p2 = new Person
{
  Name = "Alice Jones",
  DateOfBirth = new DateTime(1998, 3, 17)
};
WriteLine($"{p2.Name} was born on {p2.DateOfBirth:d MMM yy}");
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith was born on Wednesday, 22 December 1965
Alice Jones was born on 17 Mar 98
```

Хранение значения с помощью ключевого слова enum

Иногда значение должно соответствовать одному из вариантов ограниченного списка. Например, у человека может быть любимое чудо света — одно из семи. А иногда значение должно соответствовать нескольким вариантам подобного списка. Допустим, человек может хотеть посетить несколько определенных чудес света из числа тех семи. Сохранять такие данные позволяет перечисление — тип `enum`.

Перечисление — весьма эффективный способ хранить одно или несколько значений из списка допустимых, поскольку в нем используются значения типа `int` в сочетании с таблицей поиска соответствующих строковых описаний.

В Visual Studio 2017 создайте в проекте PacktLibrary класс WondersOfTheAncientWorld, нажав сочетание клавиш Shift+Alt+C или выполнив команду Project ▸ Add Class (Проект ▸ Добавить класс).

В Visual Studio Code класс создается путем выбора проекта PacktLibrary, нажатия кнопки New File (Новый файл) на небольшой панели инструментов и ввода имени WondersOfTheAncientWorld.cs (рис. 5.9).

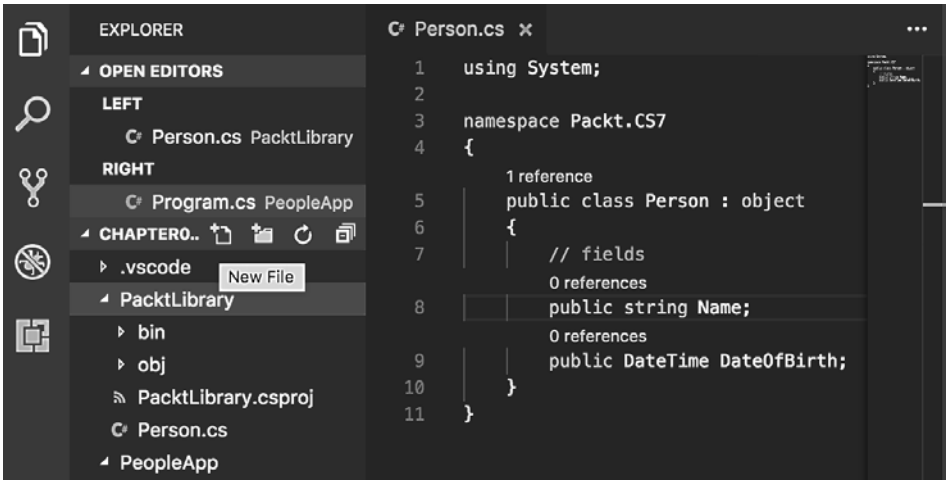


Рис. 5.9

Измените код в файле WondersOfTheAncientWorld.cs, как показано в листинге ниже:

```
namespace Packt.CS7
{
  public enum WondersOfTheAncientWorld
  {
    GreatPyramidOfGiza,
    HangingGardensOfBabylon,
    StatueOfZeusAtOlympia,
    TempleOfArtemisAtEphesus,
    MausoleumAtHalicarnassus,
    ColossusOfRhodes,
    LighthouseOfAlexandria
  }
}
```

Добавьте в классе Person следующую инструкцию к списку полей:

```
public WondersOfTheAncientWorld FavouriteAncientWonder;
```

Вернитесь к методу Main файла Program.cs и добавьте в него такие инструкции:

```
p1.FavouriteAncientWonder = WondersOfTheAncientWorld.StatueOfZeusAtOlympia;
WriteLine($"{p1.Name}'s favourite wonder is {p1.FavouriteAncientWonder}");
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith's favourite wonder is StatueOfZeusAtOlympia
```

Для хранения списка значений мы могли бы создать коллекцию экземпляров `enum`, но есть лучший способ. Можно объединить несколько вариантов в одно значение с помощью *флагов*.

Измените код перечисления, как показано в листинге, приведенном ниже. Обратите внимание: я использовал оператор сдвига битов влево (`<<`), чтобы установить отдельные биты внутри флага. Я мог бы также установить значения 1, 2, 4, 8, 16, 32 и т. д.:

```
namespace Packt.CS7
{
  [System.Flags]
  public enum WondersOfTheAncientWorld : byte
  {
    None = 0,
    GreatPyramidOfGiza = 1,
    HangingGardensOfBabylon = 1 << 1, // то есть 2
    StatueOfZeusAtOlympia = 1 << 2, // то есть 4
    TempleOfArtemisAtEphesus = 1 << 3, // то есть 8
    MausoleumAtHalicarnassus = 1 << 4, // то есть 16
    ColossusOfRhodes = 1 << 5, // то есть 32
    LighthouseOfAlexandria = 1 << 6 // то есть 64
  }
}
```

Мы явно присваиваем значения каждому варианту так, чтобы они не перекрывались при просмотре битов, хранящихся в памяти. Кроме того, следует пометить перечисление атрибутом System.Flags. Обычно перечислению соответствует внутренняя переменная типа int, но, поскольку такие большие значения не нужны, код можно улучшить, указав, что надо использовать тип byte.

Если требуется указать, что наш список значений включает в себя *Висячие сады* и *Мавзолей в Галикарнасе*, то биты 16 и 2 должны быть установлены равными `1`. Другими словами, мы сохранили бы значение `18` (табл. 5.2):

Таблица 5.2

64	32	16	8	4	2	1	0
0	0	1	0	0	1	0	0

Добавьте в классе `Person` такую инструкцию к списку полей:

```
public WondersOfTheAncientWorld BucketList;
```

Вернитесь к методу `Main` проекта PeopleApp и добавьте в него следующие инструкции, применив к списку перечислений оператор | (логическое ИЛИ) для

комбинирования значений enum. Мы также могли бы установить значение, приведя число 18 к типу enum, как показано в комментарии:

```
p1.BucketList = WondersOfTheAncientWorld.HangingGardensOfBabylon |
WondersOfTheAncientWorld.MausoleumAtHalicarnassus;
// p1.BucketList = (WondersOfTheAncientWorld)18;
WriteLine($"{p1.Name}'s bucket list is {p1.BucketList}");
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith's bucket list is HangingGardensOfBabylon,
MausoleumAtHalicarnassus
```

Используйте значения enum для сохранения комбинаций вариантов, четко отделимых друг от друга. Произведите enum из byte, если вариантов не более 8, из short, если их не более 16, из int, если не более 32, и из long, если не более 64.

Хранение группы значений с помощью коллекций

Добавим поле для хранения информации о детях человека. Перед вами пример агрегирования, поскольку дети — экземпляры класса, связанного с этим человеком, но не часть самого человека.

Мы будем использовать универсальный тип коллекции List<T>, поэтому нужно импортировать пространство имен System.Collections.Generic в верхней части файла класса Person.cs:

```
using System.Collections.Generic;
```

Угловые скобки после типа List<T> — особенность языка C# под названием универсальные типы (generics), которая была представлена в 2005 году в C# 2.0. На самом деле это лишь изящный термин для механизма, делающего коллекции строго типизированными, то есть компилятор знает, какой конкретно тип объекта может быть сохранен в коллекции. Универсальные типы повышают производительность и правильность вашего кода. Строгая типизация отличается от статической. Старые типы System.Collection типизированы статически и содержат слабо типизированные экземпляры System.Object. Новые типы System.Collection.Generic, представленные в 2005 году, типизированы статически, но включают строго типизированные экземпляры <T>. Как это ни парадоксально, термин «универсальный тип» означает более специфичный статический тип!

Затем можно объявить новое поле в классе Person:

```
public List<Person> Children = new List<Person>();
```

Убедимся, что коллекция инициализируется новым экземпляром коллекции, прежде чем сможем добавить элементы в коллекцию.

Добавьте в метод `Main` код, приведенный ниже:

```
p1.Children.Add(new Person { Name = "Alfred" });
p1.Children.Add(new Person { Name = "Zoe" });
WriteLine(
  $"{p1.Name} has {p1.Children.Count} children:");
for (int child = 0; child < p1.Children.Count; child++)
{
  WriteLine($" {p1.Children[child].Name}");
}
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith has 2 children:
  Alfred
  Zoe
```

Создание статического поля

Поля, которые мы создавали до сих пор, были членами, принадлежащими экземпляру; это значит, что для каждого созданного экземпляра класса существует собственная копия каждого поля.

Иногда требуется определить поле, для которого существует только одна копия, общая для всех экземпляров. Это так называемые *статические* члены.

В проекте PacktLibrary создайте класс `BankAccount`. Измените код класса, как показано в листинге, приведенном ниже:

```
namespace Packt.CS7
{
  public class BankAccount
  {
    public string AccountName;
    public decimal Balance;
    public static decimal InterestRate;
  }
}
```

Каждый экземпляр класса BankAccount будет иметь собственные поля AccountName и Balance, но все экземпляры будут содержать общее значение InterestRate.

Добавьте в метод `Main` файла `Program.cs` код, показанный ниже. С помощью него мы установим общую процентную ставку и создадим два экземпляра типа `BankAccount`:

```
BankAccount.InterestRate = 0.012M;
var ba1 = new BankAccount();
ba1.AccountName = "Mrs. Jones";
ba1.Balance = 2400;
WriteLine($"{ba1.AccountName} earned {ba1.Balance * BankAccount.InterestRate:C}
interest.");
var ba2 = new BankAccount();
```

```
ba2.AccountName = "Ms. Gerrier";
ba2.Balance = 98;
WriteLine($"{ba2.AccountName} earned {ba2.Balance * BankAccount.InterestRate:C}
interest.");
```

Запустите приложение и проанализируйте результат вывода:

```
Mrs. Jones earned £28.80 interest.
Ms. Gerrier earned £1.18 interest.
```

:C — это код, который определяет формат валюты для чисел на платформе .NET. В главе 8 вы изучите, как управлять региональными настройками, определяющими символ валюты. На данный момент в настройках вашей операционной системы применяется значение по умолчанию. Я живу в Лондоне, поэтому в коде используются британские фунты стерлингов (£).

Создание константного поля

Если значение поля никогда *не* должно меняться, можно использовать ключевое слово `const` и присваивать значение во время компиляции.

Добавьте в класс `Person` код, приведенный ниже:

```
// константы
public const string Species = "Homo Sapien";
```

Измените в методе `Main` код так, как показано ниже. Обратите внимание: чтобы прочитать константное поле, вы должны написать имя класса, а не экземпляра класса.

```
WriteLine($"{p1.Name} is a {Person.Species}");
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith is a Homo Sapien
```

Примеры полей `const` в типах Microsoft включают `System.Int32.MaxValue` и `System.Math.PI`, поскольку ни одно из этих значений никогда не изменится (рис. 5.10).

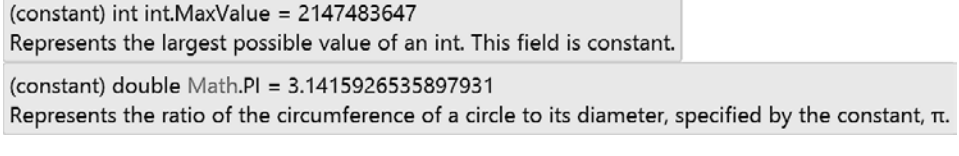


Рис. 5.10

Констант следует избегать по двум важным причинам: значение должно быть известно во время компиляции и оно должно быть выражено как литеральная строка, логическое или числовое значение. Каждая ссылка в поле const во время компиляции заменяется литеральным значением, что, следовательно, не будет учтено, если значение изменится в будущей версии.

Создание поля только для чтения

Лучший способ создания полей, значения которых не должны меняться, заключается в пометке их как «только для чтения».

В классе `Person` напишите код, приведенный ниже:

```
// поля только для чтения
public readonly string HomePlanet = "Earth";
```

Добавьте в метод `Main` инструкцию, показанную ниже. Обратите внимание: для создания поля только для чтения вы должны указать имя экземпляра класса, а не имя типа, в отличие от `const`:

```
WriteLine($"{p1.Name} was born on {p1.HomePlanet}");
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith was born on Earth
```

Используйте поля только для чтения вместо константных полей по двум важным причинам: значение может быть вычислено или загружено во время выполнения и выражено с помощью любой исполняемой инструкции. Таким образом, поле только для чтения может быть установлено с применением конструктора. Каждая ссылка в поле — живая, поэтому любые грядущие изменения будут правильно отражены при вызове кода.

Инициализация полей с помощью конструкторов

Поля часто необходимо инициализировать во время выполнения. Это делается в конструкторе, который будет вызываться при создании экземпляра класса с помощью ключевого слова `new`. Конструкторы выполняются до того, как код, использующий этот тип, установит любые поля.

Добавьте в класс `Person` следующий выделенный полужирным шрифтом код после существующего поля только для чтения `HomePlanet`:

```
// поля только для чтения
public readonly string HomePlanet = "Earth";
public readonly DateTime Instantiated;

// конструкторы
public Person()
{
  // установка дефолтных значений полей,
  // включая поля только для чтения
  Name = "Unknown";
  Instantiated = DateTime.Now;
}
```

Добавьте в метод `Main` код, приведенный ниже.

```
var p3 = new Person();
WriteLine($"{p3.Name} was instantiated at {p3.Instantiated:hh:mm:ss} on {p3.
Instantiated:dddd, d MMMM yyyy}");
```

Запустите приложение и проанализируйте результат вывода:

```
Unknown was instantiated at 11:58:12 on Sunday, 12 March 2017
```

Можно использовать несколько конструкторов в типе.

Добавьте в класс `Person` следующий код:

```
public Person(string initialName)
{
  Name = initialName;
  Instantiated = DateTime.Now;
}
```

Добавьте в метод `Main` код, указанный ниже:

```
var p4 = new Person("Aziz");
WriteLine($"{p4.Name} was instantiated at
{p4.Instantiated:hh:mm:ss} on {p4.Instantiated:dddd, d MMMM yyyy}");
```

Запустите приложение и проанализируйте результат вывода:

```
Aziz was instantiated at 11:59:25 on Sunday, 4 June 2017
```

Настройка полей через символьные константы default

Новая языковая особенность, представленная в версии C# 7.1, — символьные константы `default`. В главе 2 вы уже познакомились с ключевым словом `default(тип)`.

Например, при наличии в вашем классе нескольких полей, которые вы хотели бы инициализировать в конструкторе со значениями по умолчанию для типа, вы могли бы воспользоваться ключевым словом `default(тип)`, как показано в следующем листинге:

```
using System;
using System.Collections.Generic;
using Packt.CS7;

public class ThingOfDefaults
{
  public int Population;
  public DateTime When;
  public string Name;
  public List<Person> People;

  public ThingOfDefaults()
  {
    Population = default(int);        // C# 2.0 и выше
    When = default(DateTime);
    Name = default(string);
    People = default(List<Person>);
  }
}
```

Вы можете подумать, что компилятор должен быть в состоянии выяснить, какой тип мы имеем в виду без необходимости сообщать компилятору об этом явно, и были бы правы, но на протяжении 15 лет своей жизни компилятор C# этого не делал.

Наконец, с появлением компилятора C# 7.1 это стало возможным, как показано в следующем листинге:

```
using System;
using System.Collections.Generic;
using Packt.CS7;

public class ThingOfDefaults
{
  public int Population;
  public DateTime When;
  public string Name;
  public List<Person> People;

  public ThingOfDefaults()
  {
    Population = default;        // C# 7.1 и выше
    When = default;
    Name = default;
    People = default;
  }
}
```

Но если вы попробуете воспользоваться этим новым ключевым словом языка C# 7.1, то Visual Studio 2017 и Visual Studio Code сообщат об ошибке компиляции (рис. 5.11).

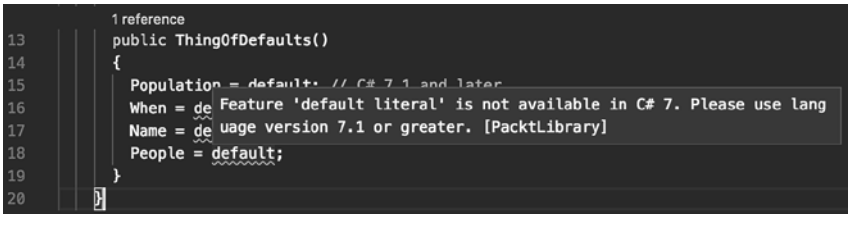


Рис. 5.11

Чтобы сообщить Visual Studio 2017 и Visual Studio Code о необходимости использования компилятора C# 7.1, откройте файл `PacktLibrary.csproj`, добавьте пару элементов `<PropertyGroup>` для конфигураций `Release` и `Debug`, как показано в листинге ниже:

```
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|AnyCPU'">
  <LangVersion>7.1</LangVersion>
</PropertyGroup>

<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
  <LangVersion>7.1</LangVersion>
</PropertyGroup>
```

Тег <LangVersion>7.1</LangVersion> можно заменить на <LangVersion>latest</LangVersion>, если есть желание всегда использовать самый новый компилятор C# в своем проекте. Например, в начале 2018 года корпорация Microsoft планирует выпустить C# 7.2.

Конструкторы — это особая категория методов. Рассмотрим последние более подробно.

Запись и вызов методов

Методы — это члены типа, которые выполняют блок инструкций.

Методы возвращают значение в зависимости от указанного типа возврата. Если указать тип возврата `void` перед именем метода, то метод выполнит определенные действия, но не вернет значение. Если же вместо ключевого слова `void` перед именем метода указать тип значения, то метод выполнит определенные действия и вернет значение.

К примеру, вы создаете два метода:

- `WriteToConsole` — будет выполнено действие (запись строки), но ничего не возвращено из метода, определенного со словом `void`;
- `GetOrigin` — вернет строковое значение, так как перед именем метода указано ключевое слово `string`.

В класс `Person` статически импортируйте тип `System.Console`, а затем добавьте код, приведенный ниже:

```
// методы
public void WriteToConsole()
{
  WriteLine($"{Name} was born on {DateOfBirth:dddd, d MMMM yyyy}");
}

public string GetOrigin()
{
  return $"{Name} was born on {HomePlanet}";
}
```

Добавьте в метод `Main` следующий код:

```
p1.WriteToConsole();
WriteLine(p1.GetOrigin());
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith was born on Wednesday, 22 December 1965
Bob Smith was born on Earth
```

Комбинация нескольких значений с помощью кортежей

Методы способны возвращать только одно значение одного типа. Этот тип может быть простым (`string` в предыдущем примере), сложным (`Person`) или коллекцией (подобно `List<Person>`).

Представьте, что нужно определить метод `GetTheData`, который возвращает как значение `string`, так и `int`. Мы могли бы определить новый класс `TextAndNumber`

с полем `string` и полем `int` и вернуть экземпляр этого сложного типа, как показано в листинге ниже:

```
public class TextAndNumber
{
  public string Text;
  public int Number;
}

public class Processor
{
  public TextAndNumber GetTheData()
  {
    return new TextAndNumber
    {
      Text = "What's the mean of life?",
      Number = 42
    };
  }
}
```

В качестве альтернативы мы могли бы применить кортежи.

Кортежи давно используются в некоторых языках, таких как F# (с первой версии), но на платформе .NET поддержка была реализована только с выпуском версии 4.0 (был добавлен тип `System.Tuple`), а в языке C# синтаксис кортежей стал поддерживаться, начиная с версии 7.

Наряду с началом поддержки кортежей в языке C# 7 на платформе .NET стал доступен новый тип `System.ValueTuple`, более эффективный в некоторых распространенных ситуациях, чем старый тип `System.Tuple` из версии .NET 4.0.

Тип System.ValueTuple не входит в состав .NET Standard 1.6 и поэтому по умолчанию недоступен в проектах .NET Core 1.0 и 1.1. Тип System.ValueTuple встроен в .NET Standard 2.0 и, следовательно, .NET Core 2.0.

Определение методов с помощью кортежей

Сначала мы определим метод, который будет работать в версии C# 4 и выше. Затем добавим поддержку нового типа из языка C# 7.

Добавьте в класс `Person` код, показанный ниже, чтобы определить два метода: первый с типом `System.Tuple<string, int>` и второй — с возвращаемым типом с использованием синтаксиса C# 7:

```
// старый синтаксис C# 4 и тип .NET 4.0 System.Tuple
public Tuple<string, int> GetFruitCS4()
{
  return Tuple.Create("Apples", 5);
}

// новый синтаксис C# 7 и новый тип System.ValueTuple
public (string, int) GetFruitCS7()
{
  return ("Apples", 5);
}
```

Добавьте в метод `Main` код, приведенный ниже:

```
Tuple<string, int> fruit4 = p1.GetFruitCS4();
WriteLine($"There are {fruit4.Item2} {fruit4.Item1}.");

(string, int) fruit7 = p1.GetFruitCS7();
WriteLine($"{fruit7.Item1}, {fruit7.Item2} there are.");
```

Запустите приложение и проанализируйте результат вывода:

```
There are 5 Apples.
Apples, 5 there are.
```

Присвоение имен полям кортежа

Чтобы получить доступ к полям кортежа, используются имена по умолчанию: `Item1`, `Item2` и т. д.

Можно явно определить имена полей. Добавьте в класс `Person` код, показанный ниже, чтобы определить метод:

```
public (string Name, int Number) GetNamedFruit()
{
  return (Name: "Apples", Number: 5);
}
```

Добавьте в метод `Main` следующий код:

```
var fruitNamed = p1.GetNamedFruit();
WriteLine($"Are there {fruitNamed.Number} {fruitNamed.Name}?");
```

Запустите приложение и проанализируйте результат вывода:

```
Are there 5 Apples?
```

Выведение имен элементов кортежа

Если вы создаете кортеж из другого объекта, то можете воспользоваться новой функцией, представленной в версии языка C# 7.1 и названной *выведением имен элементов кортежа*.

Создадим два элемента кортежа, каждый из которых будет состоять из значений `string` и `int`, как показано в следующем листинге:

```
var thing1 = ("Neville", 4);
WriteLine(
  $"{thing1.Item1} has {thing1.Item2} children.");
var thing2 = (p1.Name, p1.Children.Count);
WriteLine(
  $"{thing2.Item1} has {thing2.Item2} children.");
```

В C# 7 для обоих элементов реализована схема именования `Item1` и `Item2`. В C# 7.1 для второго элемента будут выведены имена `Name` и `Count`, как показано в следующем листинге:

```
var thing2 = (p1.Name, p1.Children.Count);
WriteLine(
  $"{thing2.Name} has {thing2.Count} children.");
```

Как и в случае с остальными нововведениями языка C# 7.1, потребуется отредактировать файл PeopleApp.csproj для использования компилятора версии 7.1 или новее.

Деконструкция кортежей

Вы также можете деконструировать кортежи в отдельные переменные. Объявление деконструкции имеет тот же синтаксис, что и именование полей, но без имени переменной для самого кортежа. Это приводит к разделению кортежа на части и назначению их новым переменным.

Добавьте в метод `Main` код, приведенный ниже:

```
(string fruitName, int fruitNumber) = p1.GetFruitCS7();
WriteLine($"Deconstructed: {fruitName}, {fruitNumber}");
```

Запустите приложение и проанализируйте результат вывода:

```
Deconstructed: Apples, 5
```

Операция деконструкции применима не только к кортежам. Любой тип может быть деконструирован, если содержит метод Deconstruct. Прочитать о деконструкции можно на сайте https://docs.microsoft.com/en-us/dotnet/csharp/tuples#deconstruction.

Определение и передача параметров в методы

У методов могут быть параметры, переданные им для изменения их поведения. Параметры определяются по аналогии с объявлением переменных, только внутри круглых скобок при объявлении метода.

Добавьте в класс `Person` код, показанный ниже, чтобы определить два метода: первый без параметров и второй — с одним параметром в скобках:

```
public string SayHello()
{
  return $"{Name} says 'Hello!'";
}
public string SayHelloTo(string name)
{
  return $"{Name} says 'Hello {name}!'";
}
```

Добавьте в метод `Main` следующий код:

```
WriteLine(p1.SayHello());
WriteLine(p1.SayHelloTo("Emily"));
```

Запустите приложение и проанализируйте результат вывода:

```
Bob Smith says 'Hello!'
Bob Smith says 'Hello Emily!'
```

Перегрузка методов

В Visual Studio 2017 и Visual Studio Code с установленным соответствующим языковым расширением при вводе инструкции, вызывающей метод, отображается меню IntelliSense, в котором приводятся полезные сведения.

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+K, I или выполните команду Edit ▸ IntelliSense ▸ Quick Info (Редактировать ▸ IntelliSense ▸ Краткие сведения), чтобы посмотреть краткую информацию о методе (рис. 5.12).

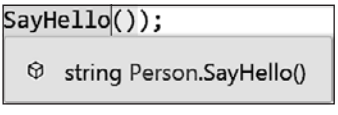


Рис. 5.12

На рис. 5.13 показана краткая информация о методе `SayHelloTo`.

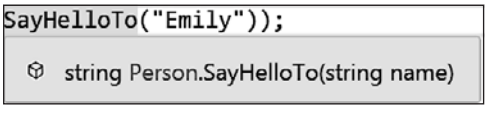


Рис. 5.13

Вместо двух методов с разными именами мы могли бы присвоить обоим методам одно и то же имя. Это допустимо, поскольку для каждого метода определена собственная сигнатура. *Сигнатура метода* — список типов параметров, которые могут быть переданы при вызове метода[1].

В классе `Person` измените имя `SayHelloTo` метода на `SayHello`. Теперь, если отобразить краткие сведения о методе, будет видно, что метод имеет одну дополнительную перегрузку (рис. 5.14).

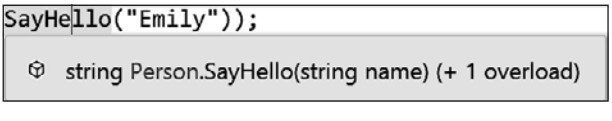


Рис. 5.14

Используйте перегруженные методы — это позволяет упростить классы так, чтобы казалось, будто у них меньше методов.

[1] Не совсем так. Полная сигнатура (используется на уровне CLR) — это имя + список типов параметров + возвращаемый тип. Краткая сигнатура (используется на уровне языка C#) — то же без возвращаемого типа. С точки зрения перегрузки методов в C# важно различать краткие сигнатуры при совпадении имен (на самом деле все еще сложнее: https://stackoverflow.com/questions/8808703/method-signature-in-c-sharp). — *Примеч. науч. ред.*

Необязательные параметры и именованные аргументы

Другой способ упрощения методов заключается в использовании необязательных параметров. Вы определяете параметр как необязательный, назначая в списке параметров метода значение по умолчанию. Необязательные параметры всегда должны указываться последними в списке параметров.

Есть исключение, касающееся необязательных параметров, всегда указываемых последними. В языке C# имеется ключевое слово params, позволяющее передавать список параметров, разделенных запятыми, любой длины в виде массива. Информацию о данном ключевом слове см. на сайте https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params.

Сейчас вы создадите метод с тремя необязательными параметрами.

Добавьте в класс `Person` код, приведенный ниже:

```
public string OptionalParameters(string command = "Run!",
  double number = 0.0, bool active = true)
{
  return $"command is {command}, number is {number}, active is {active}";
}
```

Добавьте в метод `Main` следующий код:

```
WriteLine(p1.OptionalParameters());
```

При вводе кода появится меню IntelliSense с краткими сведениями, в которых указаны три необязательных параметра со значениями по умолчанию (рис. 5.15).

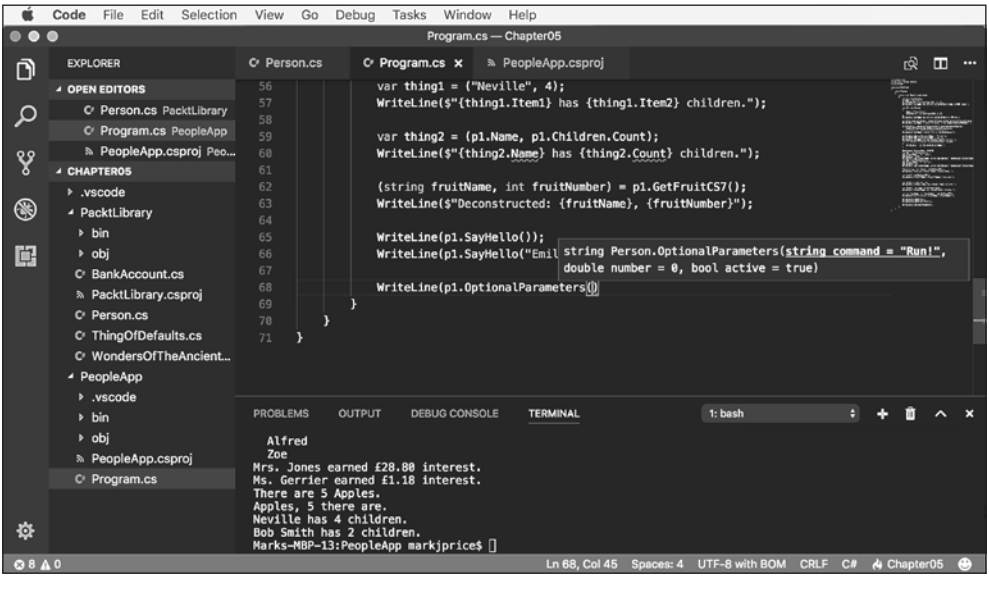


Рис. 5.15

Когда вы запустите приложение, увидите результат вывода, показанный ниже:

```
command is Run!, number is 0, active is True
```

Добавьте в метод `Main` следующую строку кода, который передает значение `string` для параметра `command` и `double` — для параметра `number`:

```
WriteLine(p1.OptionalParameters("Jump!", 98.5));
```

Запустите приложение и проанализируйте результат вывода:

```
command is Jump!, number is 98.5, active is True
```

Значения по умолчанию для параметров `command` и `number` были заменены, но значение по умолчанию для параметра `active` по-прежнему `true`.

Необязательные параметры при вызове метода нередко сочетаются с именованными параметрами, так как именование параметра позволяет передавать значения в порядке, отличном от того, в котором они были объявлены.

Добавьте в метод `Main` следующую строку кода, которая также передает значение `string` для параметра `command` и `double` — для параметра `number`, но с использованием именованных параметров, поэтому порядок их передачи может быть заменен:

```
WriteLine(p1.OptionalParameters(number: 52.7, command: "Hide!"));
```

Запустите приложение и проанализируйте результат вывода:

```
command is Hide!, number is 52.7, active is True
```

Вы даже можете использовать именованные параметры, чтобы пропустить необязательные параметры.

Добавьте в метод `Main` следующую строку кода, которая передает значение `string` для параметра `command` в соответствии с порядком, пропускает параметр `number` и включает именованный параметр `active`:

```
WriteLine(p1.OptionalParameters("Poke!", active: false));
```

Запустите приложение и проанализируйте результат вывода:

```
command is Poke!, number is 0, active is False
```

Управление передачей параметров

Параметр в метод передается одним из трех способов:

- по *значению* (по умолчанию) — можно представить как режим «только вход»;
- по *ссылке* в качестве параметра `ref` — можно представить как режим «вход-выход»;
- как *параметр* `out` — можно представить как режим «только выход».

Добавьте в класс `Person` следующий метод:

```
public void PassingParameters(int x, ref int y, out int z)
{
```

```
  // параметры out не могут иметь значение по умолчанию и должны быть
  // инициализированы в методе
  z = 99;

  // инкрементирование каждого параметра
  x++;
  y++;
  z++;
}
```

Добавьте в метод `Main` эти инструкции для объявления нескольких переменных `int` и передачи их в метод:

```
int a = 10;
int b = 20;
int c = 30;
WriteLine($"Before: a = {a}, b = {b}, c = {c}");
p1.PassingParameters(a, ref b, out c);
WriteLine($"After: a = {a}, b = {b}, c = {c}");
```

Запустите приложение и проанализируйте результат вывода:

```
Before: a = 10, b = 20, c = 30
After: a = 10, b = 21, c = 100
```

При передаче переменной как параметра обычным способом передается текущее значение, а не сама переменная. Следовательно, x — это копия переменной a. Последняя сохраняет свое первоначальное значение, 10. При передаче переменной в качестве параметра ref в метод передается ссылка на переменную. Таким образом, y является ссылкой на b. Переменная b инкрементируется при инкрементировании параметра y. При передаче переменной в качестве параметра out в метод передается ссылка на переменную. Следовательно, z является ссылкой на c. Что код метода выполнит, тем переменная c и заменится. Можно было бы упростить код в методе Main, не присваивая значение 30 переменной c, поскольку оно будет заменено в любом случае.

В языке C# 7 код можно упростить с помощью переменных `out`.

Добавьте следующие инструкции в метод `Main`:

```
// упрощенный синтаксис параметров out в C# 7
int d = 10;
int e = 20;
WriteLine($"Before: d = {d}, e = {e}, f doesn't exist yet!");
p1.PassingParameters(d, ref e, out int f);
WriteLine($"After: d = {d}, e = {e}, f = {f}");
```

В версии C# 7 ключевое слово ref используется не только для передачи параметров в метод, но и для возвращаемого значения. Это позволяет внешней переменной ссылаться на внутреннюю и изменять ее значение после вызова метода. Такая особенность может быть полезна в сложных сценариях, например для передачи временно незаполненных «пустышек» в большие структуры данных, и выходит за рамки этой книги.

Разделение классов с помощью ключевого слова partial

При командной работе над большими проектами удобно разделять определения сложных классов на несколько файлов с помощью ключевого слова `partial`.

Допустим, нам нужно создать метод в классе `Person`, не прибегая к необходимости просить другого программиста закрыть файл `Person.cs`. Если класс определен как `partial`, то мы можем разбить его на нужное количество отдельных файлов.

Добавьте в код класса `Person` ключевое слово `partial` (выделено полужирным шрифтом), как показано в листинге, приведенном ниже:

```
namespace Packt.CS7
{
  public partial class Person
  {
```

В Visual Studio 2017 выполните команду Project ▸ Add Class (Проект ▸ Добавить класс) или нажмите сочетание клавиш Shift+Alt+C. Присвойте создаваемому классу имя `Person2`. Мы не можем указать имя `Person`, поскольку Visual Studio 2017 не способна понять, что мы собрались сделать. Вместо этого теперь нужно переименовать созданный класс в `Person`, изменить пространство имен и добавить ключевые слова `public partial`, как показано в следующем коде:

```
namespace Packt.CS7
{
  public partial class Person
  {
```

В Visual Studio Code класс создается путем выбора проекта PacktLibrary на панели EXPLORER (Проводник), нажатия кнопки New File (Новый файл) на небольшой панели инструментов и ввода имени `Person2.cs`. Добавьте следующие инструкции в созданный файл:

```
namespace Packt.CS7
{
  public partial class Person
  {
  }
}
```

Остальная часть кода, который мы напишем в этой главе, будет находиться в файле Person2.cs.

Управление доступом с помощью свойств и индексаторов

Ранее мы создали метод `GetOrigin`, который возвращал строковое значение, содержащее имя и происхождение человека. В таких языках, как Java, этот способ используется часто. В C# доступен лучший подход: *свойства*.

Свойство — это обычный метод (или пара методов), который действует и выглядит как поле, когда требуется получить или установить значение, тем самым упрощая синтаксис.

Определение свойств только для чтения

Добавьте в класс `Person` в файле `Person2.cs` код, показанный ниже, чтобы определить три свойства:

- первое играет ту же роль, что и метод `GetOrigin`, и основано на синтаксисе `property`, допустимом во всех версиях языка C# (хотя используется синтаксис интерполяции строк из версии C# 6 и более поздней);
- второе возвращает приветственное сообщение, применяя синтаксис лямбда-выражения (`=>`) из версии C# 6 и более поздней;
- третье вычисляет возраст человека.

Код представлен ниже.

```
// свойство, определенное с синтаксисом из C# 1 - 5
public string Origin
{
  get
  {
    return $"{Name} was born on {HomePlanet}";
  }
}

// два свойства, определенные с помощью синтаксиса лямбда-выражения
// из C# 6 и выше
public string Greeting => $"{Name} says 'Hello!'";

public int Age => (int)(System.DateTime.Today.Subtract(DateOfBirth).TotalDays /
365.25);
```

Добавьте в метод `Main` следующий код. Можно увидеть, что получение свойства выполняется так, будто это поле.

```
var sam = new Person
{
  Name = "Sam",
  DateOfBirth = new DateTime(1972, 1, 27)
};
WriteLine(sam.Origin);
WriteLine(sam.Greeting);
WriteLine(sam.Age);
```

Запустите приложение и проанализируйте результат вывода:

```
Sam was born on Earth
Sam says 'Hello!'
46 // если запущено в период с 27 января 2018 года по 27 января 2019 года
```

Определение настраиваемых свойств

Чтобы создать такое свойство, вы должны использовать старый синтаксис и предоставить пару методов: не только часть `get`, но и фрагмент `set`.

Добавьте в файл `Person2.cs` следующий код, чтобы определить свойство `string`, которое имеет методы `get` и `set` (известные как *геттер* и *сеттер*). Несмотря на то что вы вручную не создали поле для хранения названия любимого мороженого человека, компилятор создает его автоматически.

```
public string FavoriteIceCream { get; set; } // автосинтаксис
```

Иногда требуется дополнительный контроль над установкой свойств. В этом случае следует использовать более детализированный синтаксис и вручную создать закрытое поле для хранения значения свойства:

```
private string favoritePrimaryColor;
public string FavoritePrimaryColor
{
  get
  {
    return favoritePrimaryColor;
  }
  set
  {
    switch (value.ToLower())
    {
      case "red":
      case "green":
      case "blue":
        favoritePrimaryColor = value;
        break;
      default:
        throw new System.ArgumentException($"{value} is not a
        primary color. Choose from: red, green, blue.");
    }
  }
}
```

Добавьте в метод `Main` код, приведенный ниже:

```
sam.FavoriteIceCream = "Chocolate Fudge";
WriteLine($"Sam's favorite ice-cream flavor is {sam.FavoriteIceCream}.");
sam.FavoritePrimaryColor = "Red";
WriteLine($"Sam's favorite primary color is {sam.FavoritePrimaryColor}.");
```

Запустите приложение и проанализируйте результат вывода:

```
Sam's favorite ice-cream flavor is Chocolate Fudge.
Sam's favorite primary color is Red.
```

Если вы попытаетесь присвоить в качестве цвета любое значение, отличное от красного, зеленого или синего, то код вызовет исключение. Затем вызывающий код может воспользоваться инструкцией `try` для отображения сообщения об ошибке.

Используйте свойства вместо полей при необходимости проверить, какое значение может быть сохранено, если хотите привязать данные в XAML (эта тема рассматривается в главе 17) и если хотите считывать и записывать поля без методов. Инкапсуляция полей с помощью свойств описывается на сайте https://www.microsoft.com/net/tutorials/csharp/getting-started/encapsulation-oop.

Определение индексаторов

Индексаторы позволяют вызывающему коду использовать синтаксис массива для доступа к свойству. К примеру, тип `string` определяет *индексатор*, поэтому вызывающий код может обращаться к отдельным символам в строке индивидуально. Мы определим индексатор, чтобы упростить доступ к информации о детях человека.

Добавьте в файл `Person2.cs` код, показанный ниже, чтобы определить индексатор, получающий и устанавливающий дочерний элемент на основе индекса (позиции) данного элемента:

```
// индексаторы
public Person this[int index]
{
  get
  {
    return Children[index];
  }
  set
  {
    Children[index] = value;
  }
}
```

Индексаторы можно перегружать, чтобы использовать их для разных типов. Например, по принципу передачи типа int можно передавать и string.

Добавьте в метод `Main` код, показанный ниже. После этого мы получим доступ к первому и второму дочерним элементам, используя более длинное поле `Children` и укороченный синтаксис индексатора:

```
sam.Children.Add(new Person { Name = "Charlie" });
sam.Children.Add(new Person { Name = "Ella" });
WriteLine($"Sam's first child is {sam.Children[0].Name}");
WriteLine($"Sam's second child is {sam.Children[1].Name}");
WriteLine($"Sam's first child is {sam[0].Name}");
WriteLine($"Sam's second child is {sam[1].Name}");
```

Запустите приложение и проанализируйте результат вывода:

```
Sam's first child is Charlie
Sam's second child is Ella
```

```
Sam's first child is Charlie
Sam's second child is Ella
```

Используйте индексаторы только в тех случаях, если действительно необходимо применять синтаксис квадратных скобок, как для массивов. Предыдущий пример — демонстрация того, что индексаторы редко имеют высокую ценность.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам, приведенным в данной главе.

Проверочные вопросы

1. Какие четыре модификатора доступа вы знаете и для чего они используются?
2. В чем разница между ключевыми словами `static`, `const` и `readonly`?
3. Для чего служит конструктор?
4. Зачем с ключевым словом `enum` используется атрибут `[Flags]`, если требуется хранить комбинированные значения?
5. В чем польза ключевого слова `partial`?
6. Что такое кортеж?
7. Для чего в C# используется ключевое слово `ref`?
8. Что такое перегрузка?
9. Чем поле отличается от свойства?
10. Как сделать параметр метода необязательным?

Дополнительные ресурсы

- Поля (руководство по программированию в C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/classes-and-structs/fields.
- Модификаторы доступа (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/access-modifiers.
- Конструкторы (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/classes-and-structs/constructors.
- Методы (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/methods.
- Свойства (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/properties.

Резюме

В этой главе вы освоили создание пользовательских типов с помощью объектно-ориентированного программирования. Узнали о различных категориях членов, которые могут иметь тип, включая поля для хранения данных и методов для выполнения действий. Рассмотрели концепции объектно-ориентированного программирования, такие как агрегирование и инкапсуляция, и изучили ряд новых особенностей синтаксиса языка C# 7.0 и 7.1.

В следующей главе вы примените эти концепции для определения делегатов и событий, реализации интерфейсов и наследования существующих классов.

6 Реализация интерфейсов и наследование классов

Эта глава посвящена созданию новых типов из существующих с использованием *объектно-ориентированного программирования (ООП)*. Вы узнаете, как определять операторы и локальные функции C# 7 для выполнения простых операций, делегаты и события для обмена данными между типами, а также реализовывать интерфейсы для достижения общей функциональности. Кроме того, мы обсудим, как наследовать базовые классы, чтобы получить производные классы и многократную реализацию функциональности, переопределять член типа, применять полиморфизм, создавать методы расширения и выполнять приведение классов в иерархии наследования.

В данной главе:

- настройка библиотеки классов и консольного приложения;
- упрощение методов с помощью операторов;
- определение локальных функций;
- вызов и обработка событий;
- реализация интерфейсов;
- многократное использование типов с помощью универсальных шаблонов;
- управление памятью с применением ссылочных типов и типов значений;
- наследование классов;
- приведение в иерархиях наследования;
- наследование и расширение типов .NET.

Настройка библиотеки классов и консольного приложения

Начнем с того, что введем решение/проект, подобные созданному в главе 5. Если вы выполнили все упражнения в ней, то можете открыть данный проект и продолжить работу. В противном случае следуйте инструкциям, приведенным ниже, для выбранной вами среды разработки.

Visual Studio 2017

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Standard. В центре диалогового окна выберите пункт Class Library (.NET Standard) (Библиотека классов (.NET Standard)), присвойте ему имя `PacktLibrary`, укажите расположение по адресу `C:\Code`, введите имя решения `Chapter06`, а затем нажмите кнопку OK.

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на файле `Class1.cs` и в контекстном меню выберите пункт Rename (Переименовать). Присвойте решению имя `Person`. Затем измените его содержимое следующим образом:

```
namespace Packt.CS7
{
  public class Person
  {
  }
}
```

Создайте новый проект консольного приложения `PeopleApp`.

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства) или нажмите сочетание клавиш Alt+Enter. В категории Startup Project (Запускаемый проект) выберите проект PeopleApp.

На панели Solution Explorer (Обозреватель решений) раскройте проект PeopleApp, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Add Reference (Добавить ссылку).

В левой части появившегося окна Reference Manager (Диспетчер ссылок) выберите раздел Projects (Проекты), выделите сборку PacktLibrary, а затем нажмите кнопку OK.

Visual Studio Code

Создайте каталог `Chapter06` с двумя подкаталогами — `PacktLibrary` и `PeopleApp`, как показано ниже:

- `Chapter06`:
  - `PacktLibrary`;
  - `PeopleApp`.

Запустите Visual Studio Code и откройте каталог `Chapter06`.

На панели Integrated Terminal (Интегрированный терминал) введите следующие команды:

```
cd PacktLibrary
dotnet new classlib
cd ..
cd PeopleApp
dotnet new console
```

На панели EXPLORER (Проводник) раскройте проект PacktLibrary, переименуйте файл `Class1.cs`, присвоив ему имя `Person.cs`.

Измените содержимое этого файла таким образом:

```
namespace Packt.CS7
{
  public class Person
  {
  }
}
```

На панели EXPLORER (Проводник) раскройте каталог `PeopleApp` и щелкните на файле `PeopleApp.csproj`.

Добавьте ссылку на проект PacktLibrary, как показано в следующем листинге:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\PacktLibrary\PacktLibrary.csproj" />
  </ItemGroup>

</Project>
```

На панели Integrated Terminal (Интегрированный терминал) убедитесь, что находитесь в каталоге `PeopleApp`, а затем введите команду `dotnet build` и обратите внимание на результат, указывающий успешную сборку обоих проектов (рис. 6.1):

Когда появится предупреждение о том, что необходимо добавить требуемые ресурсы, нажмите кнопку Yes (Да).

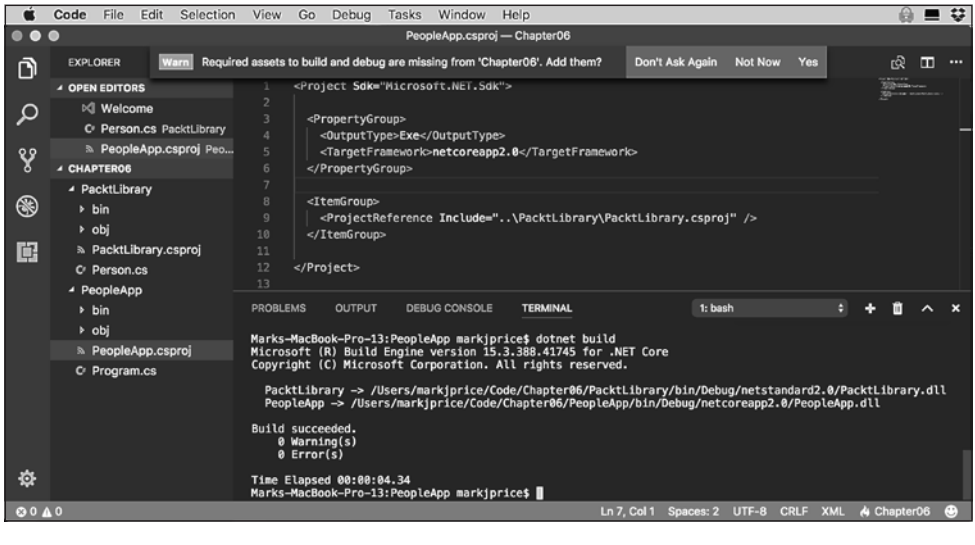


Рис. 6.1

Определение классов

В Visual Studio 2017 или Visual Studio Code добавьте код, показанный ниже, в класс `Person` библиотеки классов `PacktLibrary`:

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace Packt.CS7
{
  public partial class Person
  {
    // поля
    public string Name;
    public DateTime DateOfBirth;
    public List<Person> Children = new List<Person>();

    // методы
    public void WriteToConsole()
    {
      WriteLine($"{Name} was born on {DateOfBirth:dddd, d MMMM yyyy}");
    }
  }
}
```

Упрощение методов с помощью операторов

Иногда необходимо, чтобы два экземпляра класса `Person` могли давать потомство. Это можно реализовать путем написания методов. Методы экземпляра — действия, выполняемые объектом над самим собой, статические методы — действия, производимые типом. Ваш выбор должен строиться на действии, которое вы хотите совершить.

Иногда имеет смысл создавать и экземплярный, и статический метод, выполняющие похожие действия. Например, для класса string реализован статический метод Compare и метод экземпляра CompareTo. Такой подход делает функциональность более наглядной для программистов, использующих этот тип.

Реализация функционала с применением метода

Добавьте в класс `Person` следующие два метода, чтобы создать два объекта `Person`, как показано в листинге ниже:

```
// методы "размножения"
public static Person Procreate(Person p1, Person p2)
{
  var baby = new Person
```

```
  {
    Name = $"Baby of {p1.Name} and {p2.Name}"
  };
  p1.Children.Add(baby);
  p2.Children.Add(baby);
  return baby;
}
public Person ProcreateWith(Person partner)
{
  return Procreate(this, partner);
}
```

Обратите внимание на следующие аспекты:

- в статическом методе `Procreate` объекты `Person`, использующиеся для продолжения рода, передаются в качестве параметров `p1` и `p2`;
- создается новый экземпляр класса `Person` с именем `baby`, причем его имя — сочетание имен двух людей, которые стали его родителями;
- объект `baby` добавляется в коллекцию `Children` обоих родителей, после чего выполняется его возврат;
- в случае с экземплярным методом `ProcreateWith` объект класса `Person`, с которым нужно завести ребенка, передается в качестве параметра `partner`, после чего вместе с `this` передается в статический метод `Procreate`.

Метод, создающий новый объект или изменяющий уже имеющийся, должен возвращать этот объект, чтобы код, вызвавший данный метод, мог видеть результат.

В начало файла `Program.cs` из проекта `PeopleApp` поместите код, показанный ниже, чтобы импортировать пространство имен для нашего класса и выполнить статический импорт типа `Console`:

```
using System;
using Packt.CS7;
using static System.Console;
```

Теперь можно создать трех человек и произвести потомство, добавив следующий код в метод `Main`:

```
var harry = new Person { Name = "Harry" };
var mary = new Person { Name = "Mary" };
var jill = new Person { Name = "Jill" };

// метод вызова экземпляра
var baby1 = mary.ProcreateWith(harry);

// метод статического вызова
var baby2 = Person.Procreate(harry, jill);

WriteLine($"{mary.Name} has {mary.Children.Count} children.");
```

```
WriteLine($"{harry.Name} has {harry.Children.Count} children.");
WriteLine($"{jill.Name} has {jill.Children.Count} children.");
WriteLine($"{mary.Name}'s first child is named
\"{mary.Children[0].Name}\".");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Mary has 1 children.
Harry has 2 children.
Jill has 1 children.
Mary's first child is named "Baby of Harry and Mary".
```

Реализация функционала с помощью оператора

Для класса `System.String` реализован статический метод `Concat`, выполняющий конкатенацию двух строковых значений, как показано в следующем листинге:

```
string s1 = "Hello ";
string s2 = "World!";
string s3 = string.Concat(s1, s2);
WriteLine(s3); // => Hello World!
```

Вызов таких методов, как `Concat`, вполне позволяет решить задачу, однако для программиста было бы более естественно использовать символ +, чтобы *сложить* два строковых значения, как показано в листинге ниже:

```
string s1 = "Hello ";
string s2 = "World!";
string s3 = s1 + s2;
WriteLine(s3); // => Hello World!
```

Известная фраза «Плодитесь и размножайтесь» поощряет рождение детей. Поэтому напишем код, в котором символ * (умножение) позволит двум объектам `Person` производить потомство.

Мы сделаем это, определив статический оператор (с модификатором `static`) для символа *. Синтаксис больше похож на синтаксис метода, ведь, по сути, оператор есть метод, только вместо имени метода используется выбранный символ.

Список символов, которые ваши типы могут использовать в качестве операторов, приведен по ссылке https://docs.microsoft.com/en-us/dotnet/csharp/programmingguide/statements-expressions-operators/overloadable-operators.

В проекте PacktLibrary для класса `Person` создайте статический оператор для символа *, как показано в следующем листинге:

```
// оператор для «умножения»
public static Person operator *(Person p1, Person p2)
{
  return Person.Procreate(p1, p2);
}
```

В отличие от методов операторы не отображаются в списках IntelliSense для типа. Для каждого определяемого оператора создавайте метод, так как сторонний программист может не понять, что ему доступен определенный вами оператор. Реализация оператора может вызывать этот метод, повторно используя уже написанный вами код. Вторая причина для предоставления метода — тот факт, что операторы способны работать медленнее, чем вызовы методов. Если производительность в приоритете, то программист может пожертвовать удобочитаемостью и вызвать ваш метод.

В методе `Main` после вызова статического метода `Procreate` используйте оператор * для создания еще одного потомка, как выделено в следующем коде:

```
// вызов статического метода
var baby2 = Person.Procreate(harry, jill);

// вызов оператора
var baby3 = harry * mary
```

Запустите приложение и проанализируйте результат вывода:

```
Mary has 2 children.
Harry has 3 children.
Jill has 1 children.
Mary's first child is named "Baby of Harry and Mary".
```

Определение локальных функций

К новым возможностям языка C# 7 относятся локальные функции — способ программирования, эквивалентный локальным переменным. Другими словами, локальные функции — это методы, которые видимы и могут быть вызваны только в пределах метода, где были определены. В других языках программирования такие функции иногда называются *вложенными* или *внутренними*.

Мы воспользуемся локальной функцией для реализации вычисления факториала.

Добавьте в класс `Person` код, показанный ниже:

```
// метод с локальной функцией
public static int Factorial(int number)
{
  if (number < 0)
  {
    throw new ArgumentException(
      $"{nameof(number)} cannot be less than zero.");
  }
  return localFactorial(number);

  int localFactorial(int localNumber)
  {
    if (localNumber < 1) return 1;
    return localNumber * localFactorial(localNumber - 1);
  }
}
```

Добавьте в метод `Main` файла `Program.cs` следующую инструкцию:

```
WriteLine($"5! is {Person.Factorial(5)}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
5! is 120
```

Вызов и обработка событий

Методы часто описываются как *действия, которые может выполнять объект*. К примеру, класс `List` может добавить в себя элемент или очистить самого себя.

События нередко описываются как *действия, которые происходят с объектом*. К примеру, в пользовательском интерфейсе Button есть событие Click, определяющее, что происходит с кнопкой при ее нажатии.

События можно охарактеризовать и как способ обмена сообщениями между двумя объектами.

Вызов методов с помощью делегатов

Вы уже знакомы с наиболее распространенным способом вызова или выполнения метода; он заключается в использовании синтаксиса с *точкой* для доступа к методу по его имени. К примеру, метод `Console.WriteLine` дает указание типу `Console` выводить сообщения в окно консоли или в область TERMINAL (Терминал).

Как вариант, для вызова или выполнения метода можно применить *делегат*. Если вы использовали языки, поддерживающие указатели на функции, то делегат можно представить как типобезопасный указатель на метод. Другими словами, делегат — это адрес метода в памяти, имеющего ту же сигнатуру, что и у делегата, что позволяет безопасно его вызывать.

Для примера представьте, что существует метод, который должен иметь строку, переданную как единственный параметр, и возвращать `int`:

```
public int MethodIWantToCall(string input)
{
  return input.Length; // неважно, что выполняется здесь
}
```

Я мог бы вызвать этот метод непосредственно следующим образом:

```
int answer = p1.MethodIWantToCall("Frog");
```

Как вариант, я мог бы определить делегат с совпадающей сигнатурой для косвенного вызова метода. Обратите внимание: именам параметров совпадать не обязательно. Должны соответствовать только типы параметров и возвращаемые значения.

```
delegate int DelegateWithMatchingSignature(string s);
```

Теперь я могу создать экземпляр делегата, связать его с методом и, наконец, вызвать делегат (который вызывает метод!):

```
var d = new DelegateWithMatchingSignature(p1.MethodIWantToCall);
int answer2 = d("Frog");
```

Вероятно, вы задались вопросом: «А в чем смысл?» Отвечу лаконично — в гибкости.

Мы могли бы использовать делегаты для создания очереди методов, которые нужно вызвать по порядку. У делегатов имеется встроенная поддержка асинхронных операций, выполняемых в другом потоке для повышения производительности. Самое главное — делегаты позволяют создавать события.

Делегаты и события — одни из самых продвинутых функций языка C# и обучение им может занять некоторое время, поэтому не беспокойтесь, если сразу не поняли принцип их работы!

Определение событий

Microsoft ввел в использование два предопределенных делегата для использования в качестве событий. Они выглядят следующим образом:

```
public delegate void EventHandler(object sender, EventArgs e);
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Если хотите определить событие в собственном типе, то должны использовать один из этих двух предопределенных делегатов.

Добавьте код, показанный ниже, в класс `Person`. Код определяет событие `Shout`. Он также определяет поле `AngerLevel` для хранения данных и метод `Poke`. Каждый раз, когда человека толкают (`Poke`), уровень его раздражения (`AngerLevel`) растет. Как только данный уровень достигает 3, поднимается событие `Shout` (возмущенный возглас), но только если делегат события указывает на метод, определенный где-либо еще в коде, то есть не `null`.

```
// событие
public event EventHandler Shout;

// поле
public int AngerLevel;

// метод
public void Poke()
{
  AngerLevel++;
  if (AngerLevel >= 3)
  {
    // если кто-то нас слушает...
    if (Shout != null)
    {
      // ...поднимается событие
      Shout(this, EventArgs.Empty);
    }
  }
}
```

Проверка, является ли объект null до вызова одного из его методов, выполняется очень часто. C# позволяет упростить эти инструкции: Shout?.Invoke(this, EventArgs.Empty);.

Visual Studio 2017

В Visual Studio 2017 в методе `Main` начните вводить код, показанный ниже, чтобы назначить обработчик события:

```
harry.Shout +=
```

Обратите внимание на меню IntelliSense, которое появляется при вводе символов оператора `+=` (рис. 6.2).

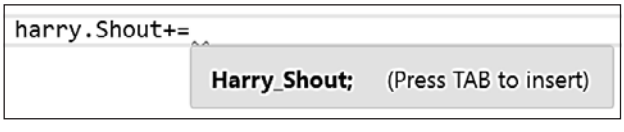


Рис. 6.2

Нажмите клавишу Tab. Вы увидите панель с предложениями вставки кода.

Нажмите клавишу Enter, чтобы принять имя метода.

Visual Studio 2017 добавит метод, который соответствует сигнатуре делегата события. Этот метод будет вызван автоматически при поднятии события.

Прокрутите код вниз, чтобы найти метод, созданный Visual Studio 2017, и удалите инструкцию, которая вызывает исключение `NotImplementedException`.

Visual Studio Code

В Visual Studio Code вам нужно написать метод и назначить ему имя самостоятельно. Имя может быть любым, но логичнее использовать текст `Harry_Shout`.

Добавьте в класс `Program` метод, как показано в этом коде:

```
private static void Harry_Shout(object sender, EventArgs e)
{
}
```

В метод `Main` добавьте данную инструкцию для назначения метода событию:

```
harry.Shout += Harry_Shout;
```

Visual Studio 2017 или Visual Studio Code

В Visual Studio 2017 или Visual Studio Code добавьте в метод `Harry_Shout` следующие инструкции для получения ссылки на объект `Person` и вывода некой информации, как показано в коде, приведенном ниже:

```
private static void Harry_Shout(object sender, EventArgs e)
{
  Person p = (Person)sender;
  WriteLine($"{p.Name} is this angry: {p.AngerLevel}.");
}
```

Вернитесь к методу `Main` и добавьте следующие инструкции для вызова метода `Poke` четыре раза после назначения метода событию `Shout`:

```
harry.Shout += harry_Shout;
harry.Poke();
harry.Poke();
harry.Poke();
harry.Poke();
```

Запустите приложение. Обратите внимание, что Гарри первое время только злится и начинает кричать после того, как его толкнули не менее трех раз:

```
Harry is this angry: 3.
Harry is this angry: 4.
```

Вы можете определять собственные типы EventArgs, чтобы передавать дополнительные данные методу обработчика событий. Более подробная информация доступна на сайте https://docs.microsoft.com/en-us/dotnet/standard/events/how-toraise-and-consume-events.

Реализация интерфейсов

Интерфейсы — это способ соединять разные типы с целью создать что-нибудь новое. В качестве примеров интерфейсов можно привести шипы на кирпичиках «Лего», которые позволяют деталькам удерживаться вместе, или электрические стандарты для вилок и розеток.

Если тип реализует интерфейс, то гарантирует остальной части .NET, что поддерживает определенный функционал.

Универсальные интерфейсы

В табл. 6.1 приведен список универсальных интерфейсов, которые могут реализовать ваши типы.

Таблица 6.1

Интерфейс	Метод (-ы)	Описание
IComparable	CompareTo(прочие)	Определяет метод сравнения, который тип реализует для упорядочения или сортировки своих экземпляров
IComparer	Compare(первый, второй)	Определяет метод сравнения, который вторичный тип реализует для упорядочения или сортировки экземпляров первичного типа
IDisposable	Dispose()	Предоставляет механизм для освобождения неуправляемых ресурсов, что более эффективно, чем ожидание финализатора

Продолжение ⇩

Таблица 6.1 *(продолжение)*

Интерфейс	Метод (-ы)	Описание
IFormattable	ToString(формат, культура)	Определяет метод, поддерживающий региональные параметры, для форматирования значения объекта в строковое представление
IFormatter	Serialize(поток, объект), Deserialize(поток)	Определяет методы преобразования объекта в поток байтов и из него для хранения или передачи
IFormatProvider	GetFormat(тип)	Определяет метод форматирования входных данных на основе языка и региона

Сравнение объектов при сортировке

Один из наиболее распространенных интерфейсов, который вы можете реализовать, — это `IComparable`. Он позволяет сортировать массивы и коллекции вашего типа.

Сортировка объектов без метода сравнения

Добавьте в метод `Main` показанный ниже код, который создает массив экземпляров `Person`, выводит массив, пытается его сортировать и затем выводит отсортированный массив:

```
Person[] people =
{
  new Person { Name = "Simon" },
  new Person { Name = "Jenny" },
  new Person { Name = "Adam" },
  new Person { Name = "Richard" }
};

WriteLine("Initial list of people:");
foreach (var person in people)
{
  WriteLine($"{person.Name}");
}

WriteLine("Use Person's IComparable implementation to sort:");
Array.Sort(people);
foreach (var person in people)
{
  WriteLine($"{person.Name}");
}
```

Запустите приложение и увидите следующую ошибку выполнения:

```
Unhandled Exception: System.InvalidOperationException: Failed to compare two
elements in the array. ---> System.ArgumentException: At least one object must
implement IComparable.
```

Как следует из текста ошибки, для устранения проблемы наш тип должен реализовать интерфейс `IComparable`.

Определение метода сравнения

Добавьте в проект PacktLibrary, в конец описания класса `Person`, код, показанный ниже:

```
public partial class Person : IComparable<Person>
```

Visual Studio 2017 и Visual Studio Code подчеркнут новый код, предупреждая о том, что вы еще не реализовали обещанный метод.

Эти среды могут сгенерировать каркас реализации, если вы установите указатель мыши на кнопку в виде лампочки и в контекстном меню выберете пункт Implement interface (Реализовать интерфейс) (рис. 6.3).

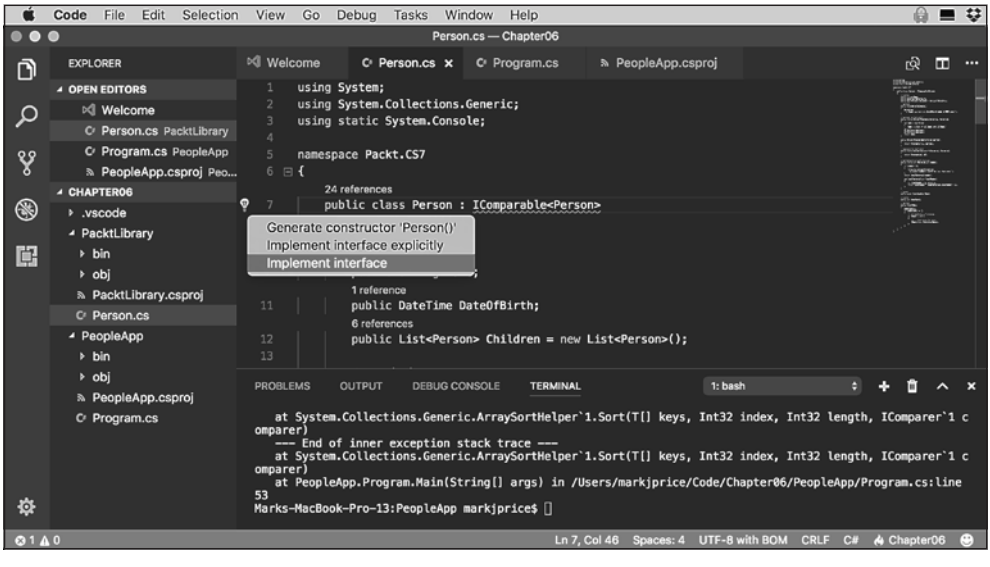


Рис. 6.3

Интерфейсы могут реализовываться неявно и явно. Неявные реализации проще. Явные необходимы только в том случае, если тип должен содержать несколько методов с одинаковым именем. Например, IGamePlayerк и IKeyHolder могут включать метод Lose. В типе, который должен реализовывать оба интерфейса, только одна реализация метода Lose может быть выполнена неявно. Другой метод Lose нужно реализовать и вызвать явно. Дополнительная информация приведена на сайте https://docs.microsoft.com/en-us/dotnet/csharp/programmingguide/interfaces/explicit-interface-implementation.

Прокрутите код вниз, чтобы найти сгенерированный метод, и удалите инструкцию, вызывающую ошибку `NotImplementedException`. Добавьте инструкцию для

вызова метода CompareTo поля Name, использующего реализацию CompareTo типа string, как показано в листинге ниже:

```
public int CompareTo(Person other)
{
  return Name.CompareTo(other.Name);
}
```

Теперь сравним два экземпляра Person по полям Name. Это значит, что люди будут сортироваться в алфавитном порядке по имени.

Запустите приложение. На этот раз все работает:

```
Initial list of people:
Simon
Jenny
Adam
Richard
Use Person's IComparable implementation to sort:
Adam
Jenny
Richard
Simon
```

Определение отдельного компаратора

В некоторых случаях доступа к исходному коду типа нет, и он может не реализовывать интерфейс IComparable. К счастью, есть еще один способ сортировки экземпляров типа. Можно создать вторичный тип, реализующий несколько иной интерфейс под названием IComparer.

Добавьте в проект PacktLibrary новый класс PersonComparer, который реализует интерфейс IComparer, как показано в следующем блоке кода. Он будет сравнивать двух людей, сравнивая длину их полей Name или имена в алфавитном порядке, если последние одинаковой длины:

```
using System.Collections.Generic;

namespace Packt.CS7
{
  public class PersonComparer : IComparer<Person>
  {
    public int Compare(Person x, Person y)
    {
      // Сравнение длины полей Name...
      int temp = x.Name.Length.CompareTo(y.Name.Length);

      // ...если они одинаковы...
      if (temp == 0)
      {
        // ...выполнить сортировку по именам...
```

```
        return x.Name.CompareTo(y.Name);
      }
      else
      {
        // ...иначе сортировать по длине.
        return temp;
      }
    }
  }
}
```

Добавьте в метод `Main` класса `Program` проекта PeopleApp код, приведенный ниже:

```
WriteLine("Use PersonComparer's IComparer implementation to sort:");
Array.Sort(people, new PersonComparer());
foreach (var person in people)
{
  WriteLine($"{person.Name}");
}
```

Запустите приложение.

На этот раз при сортировке массива людей мы явно просим алгоритм сортировки использовать тип `PersonComparer`, так что сначала люди сортируются по длине имен, а если длины двух или более имен равны, то по алфавиту:

```
Use Person's IComparable implementation to sort:
Adam
Jenny
Richard
Simon
Use PersonComparer's IComparer implementation to sort:
Adam
Jenny
Simon
Richard
```

Если кому-либо потребуется сортировать массив или коллекцию экземпляров вашего типа, то реализуйте интерфейс IComparable.

Многократное использование типов с помощью универсальных шаблонов

В 2005 году с появлением C# и .NET Framework 2.0 корпорация Microsoft представила нововведение, названное *универсальными типами*. Оно позволяет повысить повторное использование типов, благодаря тому что программисты теперь могут передавать типы как параметры, наподобие того, как вы передаете объекты в качестве параметров.

Создание универсальных типов

В первую очередь рассмотрим пример неуниверсального типа, чтобы вам было легче понять, какую проблему призваны решить универсальные типы.

Добавьте в проект PacktLibrary новый класс `Thing`, как показано в листинге ниже, и обратите внимание на следующее:

- в классе `Thing` есть поле `Data` типа `object`;
- в классе `Thing` есть метод `Process`, принимающий входящий параметр типа `string` и возвращающий значение типа `string`.

Если бы мы захотели, чтобы тип Thing был гибким в .NET Framework 1.0, то пришлось бы использовать тип object для поля.

```
using System;

namespace Packt.CS7
{
  public class Thing
  {
    public object Data = default(object);

    public string Process(string input)
    {
      if (Data == input)
      {
        return Data.ToString() + Data.ToString();
      }
      else
      {
        return Data.ToString();
      }
    }
  }
}
```

Добавьте в проект PeopleApp несколько инструкций в конец метода `Main`, как показано в следующем листинге:

```
var t = new Thing();
t.Data = 42;
WriteLine($"Thing: {t.Process("42")}");
```

Запустите консольное приложение, проанализируйте результат вывода и обратите внимание на предупреждение:

```
Thing.cs(11,17): warning CS0252: Possible unintended reference comparison;
to get a value comparison, cast the left hand side to type 'string'
[/Users/markjprice/Code/Chapter06/PacktLibrary/PacktLibrary.csproj]
```

Класс Thing является гибким, поскольку полю Data может быть присвоен любой тип, поэтому в методе Process мы не можем безопасно выполнять какие-либо операции, за исключением вызова метода ToString.

Добавьте в проект PacktLibrary новый класс GenericThing, как показано в следующем листинге:

```
using System;

namespace Packt.CS7
{
  public class GenericThing<T> where T : IComparable, IFormattable
  {
    public T Data = default(T);

    public string Process(string input)
    {
      if (Data.ToString().CompareTo(input) == 0)
      {
        return Data.ToString() + Data.ToString();
      }
      else
      {
        return Data.ToString();
      }
    }
  }
}
```

Обратите внимание, что в классе GenericThing есть:

- параметр T; его может иметь любой тип, для которого реализованы интерфейсы IComparable и IFormattable, то есть в этом классе должны быть реализованы методы CompareTo и ToString. По соглашению используйте имя параметра T при наличии только одного параметра-типа;
- поле Data и тип T;
- метод Process, принимающий входящий параметр типа string и возвращающий значение типа string.

Добавьте в проект PeopleApp несколько инструкций в конец метода Main, как показано в листинге ниже, и обратите внимание на следующее:

- при создании экземпляра универсального типа разработчик должен передать в него параметр-тип. В этом примере мы передаем int в качестве типа, поэтому всякий раз при появлении в классе GenericClass тип T автоматически заменяется на int;
- при установке значения поля Data мы должны использовать значение типа int, например 42:

```
var gt = new GenericThing<int>();
gt.Data = 42;
WriteLine($"GenericThing: {gt.Process("42")}");
```

Запустите консольное приложение и проанализируйте результат вывода; обратите внимание, что логика метода Process работает корректно для объектов класса GenericThing, но не для Thing:

```
Thing: 42
GenericThing: 4242
```

Создание универсального метода

Универсальные методы могут быть реализованы даже внутри неуниверсального типа.

Добавьте в проект PacktLibrary класс `Squarer` с универсальным методом `Square`, как показано в листинге ниже:

```
using System;
using System.Threading;

namespace Packt.CS7
{
  public static class Squarer
  {
    public static double Square<T>(T input)
    where T : IConvertible
    {
      double d = input.ToDouble(
      Thread.CurrentThread.CurrentCulture);
      return d * d;
    }
  }
}
```

Обратите внимание на следующие аспекты.

- Статический класс `Squarer` не является универсальным.
- Статический метод `Square` — универсальный, а для его параметра-типа `T` должен быть реализован интерфейс `IConvertible`, чтобы мы знали о доступности метода `ToDouble`. Тип `T` используется как тип входящего параметра.
- Метод `ToDouble` требует параметр, для которого реализован интерфейс `IFormatProvider`; благодаря ему код понимает формат чисел для заданного языка или региона. Можно передать в данный поток свойство `CurrentCulture`, чтобы задать регион, используемый в настоящее время вашим компьютером. Более подробно определение регионов вы изучите в главе 8.
- Возвращаемое значение — это входящее значение, умноженное на само себя.

В нижнюю часть метода `Main` класса `Program` проекта PeopleApp добавьте следующий код. Обратите внимание, что при вызове универсального метода можно указать параметр-тип для повышения удобочитаемости кода, хотя обычно компилятор справится с этой задачей, не прибегая к указанию типа с вашей стороны:

```
string number1 = "4";
WriteLine($"{number1} squared is {Squarer.Square<string>(number1)}");

byte number2 = 3;
WriteLine($"{number2} squared is {Squarer.Square<byte>(number2)}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
4 squared is 16
3 squared is 9
```

Управление памятью с применением ссылочных типов и типов значений

Существует две категории памяти: *стек* и *куча*. Первый быстр, но лимитирован, а вторая медленна, однако увеличивается динамически.

Для создания объектов можно использовать два ключевых слова C#: `class` и `struct`. Оба могут иметь одинаковые члены. Разница между ними заключается в распределении памяти.

При использовании `class` вы определяете ссылочный тип. Это значит, что память для самого объекта выделяется в куче и в стеке хранится только адрес объекта в памяти (и некоторые служебные данные).

Задействуя `struct`, вы определяете тип значения. Это значит, что память для самого объекта выделяется в стеке.

Если в структуре используются типы, которые не относятся к типу struct для любого из своих полей, то эти поля будут храниться в куче!

На платформе .NET Core к наиболее распространенным типам `struct` относятся следующие:

- *числа* — `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`;
- *разное* — `char`, `bool`;
- *system.Drawing* — `Color`, `Point`, `Rectangle`.

Почти все остальные типы в .NET Core являются классами, включая `string`.

Вы не можете наследовать структуры.

Определение структур

Добавьте файл класса `DisplacementVector.cs` в проект PacktLibrary.

В Visual Studio 2017 нет шаблона структуры, поэтому нужно добавить класс, а затем сменить тип вручную.

Измените код в файле так, как показано в листинге ниже, и учитывайте следующее:

- используется значимый тип `struct`, а не ссылочный тип `class`;
- у него есть два поля типа `int` с именами `X` и `Y`;
- у типа есть конструктор для установки начальных значений полей `X` и `Y`;

- для типа реализован оператор, чтобы сложить два экземпляра; результатом выполнения данного оператора является возврат нового экземпляра, при этом X складывается с X, а Y — с Y.

```
namespace Packt.CS7
{
  public struct DisplacementVector
  {
    public int X;
    public int Y;

    public DisplacementVector(int initialX, int initialY)
    {
      X = initialX;
      Y = initialY;
    }

    public static DisplacementVector operator
      +(DisplacementVector vector1, DisplacementVector vector2)
    {
      return new DisplacementVector(vector1.X + vector2.X,vector1.Y + vector2.Y);
    }
  }
}
```

Открыв проект PeopleApp, в метод `Main` класса `Program` добавьте код, приведенный ниже, чтобы создать два новых экземпляра `DisplacementVector`, добавить их вместе и вывести результат:

```
var dv1 = new DisplacementVector(3, 5);
var dv2 = new DisplacementVector(-2, 7);
var dv3 = dv1 + dv2;
WriteLine($"({dv1.X}, {dv1.Y}) + ({dv2.X}, {dv2.Y}) = ({dv3.X},{dv3.Y})");
```

Запустите приложение и проанализируйте результат вывода:

```
(3, 5) + (-2, 7) = (1, 12)
```

Если все поля вашего типа используют не более 16 байт стековой памяти, в нем в качестве полей служат только структуры и вы не планируете создавать наследников своего типа, то сотрудники корпорации Microsoft рекомендуют применять структуру. Если ваш тип задействует более 16 байт стековой памяти, или в нем в качестве полей используются классы, или вы планируете наследовать его, то применяйте класс.

Освобождение неуправляемых ресурсов

Из предыдущей главы вы узнали, что конструкторы могут использоваться для инициализации полей и тип может иметь несколько конструкторов.

Представьте, что конструктор выделяет неуправляемый ресурс, то есть что-то, неподконтрольное .NET. Такой ресурс должен освобождаться вручную, поскольку платформа не способна сделать это автоматически.

В рамках данной темы я продемонстрирую несколько примеров кода, которые не нужно создавать в вашем текущем проекте.

Каждый тип может иметь один *деструктор* (финализатор), который общеязыковая исполняющая среда (CLR) будет вызывать при необходимости освободить ресурсы. Деструктор имеет то же имя, что и конструктор, то есть имя типа, но с префиксом в виде символа тильды (~), как показано в следующем примере:

```
public class Animal
{
  public Animal()
  {
    // выделение неуправляемого ресурса
  }
  ~Animal() // деструктор (финализатор)
  {
    // освобождение неуправляемого ресурса
  }
}
```

Не путайте деструктор (финализатор) с деконструктором. Первый освобождает ресурсы, то есть уничтожает объект. Второй возвращает объект, разбитый на составные части, и использует синтаксис деконструкции, появившийся в версии C# 7.

Это минимум, необходимый для текущего сценария. Проблема с предоставлением деструктора заключается в следующем: сборщику мусора .NET потребуется две сборки мусора, чтобы полностью освободить выделенные ресурсы для данного типа.

Хотя это не обязательно, рекомендуется также предоставить метод, позволяющий разработчику, использующему ваш тип, явно освобождать ресурсы, чтобы сборщик мусора мог освободить объект в течение одной сборки.

В .NET есть стандартный механизм, позволяющий выполнить указанное действие, — реализация интерфейса `IDisposable`, показанная в следующем примере:

```
public class Animal : IDisposable
{
  public Animal()
  {
    // выделение неуправляемого ресурса
  }

  ~Animal() // деструктор
  {
    if (disposed) return;
    Dispose(false);
  }

  bool disposed = false; // освобождены ли ресурсы?

  public void Dispose()
  {
```

```
    Dispose(true);
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing)
  {
    if (disposed) return;
    // освобождение *неуправляемого* ресурса
    // ...
    if (disposing)
    {
      // освобождение любых других *управляемых* ресурсов
      // ...
    }
    disposed = true;
  }
}
```

Существует два метода Dispose. Открытый метод (public) разработчик должен вызывать при использовании вашего типа. Защищенный (protected) метод Dispose (с параметром bool) используется для реализации удаления ресурсов, как неуправляемых, так и управляемых. При вызове открытого метода Dispose необходимо освободить как неуправляемые, так и управляемые ресурсы, а при выполнении деструктора — только неуправляемые.

Кроме того, обратите внимание на вызов метода `GC.SuppressFinalize(this)` — он уведомляет сборщик мусора о том, что больше не нужно запускать деструктор, и устраняет необходимость во второй сборке.

Обеспечение вызова метода Dispose

Когда используется тип, реализующий интерфейс `IDisposable`, гарантировать вызов открытого метода `Dispose` можно с помощью инструкции `using`, как показано в коде ниже:

```
using(Animal a = new Animal())
{
  // код с экземпляром Animal
}
```

Компилятор преобразует ваш код в нечто подобное показанному ниже, гарантируя: метод `Dispose` будет вызван даже в случае вызова исключения:

```
Animal a = new Animal();
try
{
  // код с экземпляром Animal
}
finally
{
  if (a != null) a.Dispose();
}
```

Практические примеры освобождения неуправляемых ресурсов с помощью метода IDisposable, инструкций using и блоков try ... finally приведены в главе 9.

Наследование классов

Тип `Person`, созданный нами ранее, неявно произведен (унаследован) от типа `System.Object`. Теперь мы создадим новый класс, наследуемый от класса `Person`.

Создайте класс `Employee.cs` (сотрудник) в проекте PacktLibrary.

Измените код этого класса так, как показано в приведенном ниже листинге:

```
using System;

namespace Packt.CS7
{
  public class Employee : Person
  {
  }
}
```

Добавьте следующие инструкции в метод `Main`, чтобы создать экземпляр класса `Employee`.

```
Employee e1 = new Employee
{
  Name = "John Jones",
  DateOfBirth = new DateTime(1990, 7, 28)
};
e1.WriteToConsole();
```

Запустите консольное приложение и проанализируйте результат вывода:

```
John Jones was born on Saturday, 28 July 1990
```

Обратите внимание, что класс `Employee` унаследовал все члены класса `Person`.

Расширение классов

Теперь мы добавим несколько членов, специфичных для сотрудников (`Employee`), тем самым расширив класс.

Добавьте в класс `Employee` код, показанный ниже, чтобы определить два свойства:

```
public string EmployeeCode { get; set; }
public DateTime HireDate { get; set; }
```

Вернитесь к методу `Main` и добавьте такой код:

```
e1.EmployeeCode = "JJ001";
e1.HireDate = new DateTime(2014, 11, 23);
WriteLine($"{e1.Name} was hired on {e1.HireDate:dd/MM/yy}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
John Jones was hired on 23/11/14
```

Скрытие членов класса

До сих пор метод `WriteToConsole` наследовался от класса `Person` и выводил только имя сотрудника и дату его рождения. Вам может понадобиться изменить поведение этого метода для сотрудника.

Добавьте в класс `Employee` код, показанный ниже, чтобы переопределить метод `WriteToConsole`.

Обратите внимание: вам нужно статически импортировать метод System.Console.

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

namespace Packt.CS6
{
  public class Employee : Person
  {
    public string EmployeeCode { get; set; }
    public DateTime HireDate { get; set; }

    public void WriteToConsole()
    {
      WriteLine($"{Name}'s birth date is {DateOfBirth:dd/MM/yy} and hire
          date was {HireDate:dd/MM/yy}");
    }
  }
}
```

Запустите приложение и проанализируйте результат вывода:

```
John Jones's birth date is 28/07/90 and hire date was 01/01/01
John Jones was hired on 23/11/14
```

Обе среды: и Visual Studio 2017, и Visual Studio Code — уведомят вас о том, что ваш метод скрывает одноименный метод, унаследованный от класса `Person`, с помощью зеленой волнистой черты под именем вашего метода (рис. 6.4).

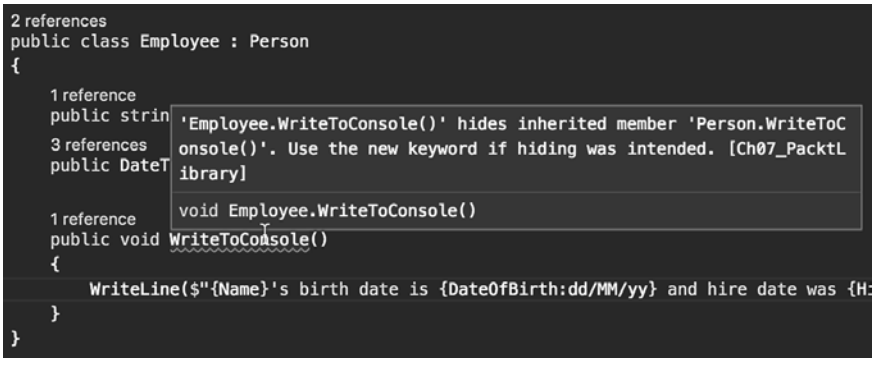


Рис. 6.4

Избавиться от предупреждения можно, добавив к описанию метода ключевое слово `new`, указывающее на то, что вы намеренно замещаете старый метод, как показано в коде, приведенном ниже:

```
public new void WriteToConsole()
```

Переопределение членов

Вместо скрытия метода чаще лучше переопределить его. Вы можете переопределять члены только в том случае, если базовый класс допускает переопределение с помощью ключевого слова `virtual`.

Добавьте в метод `Main` следующую инструкцию:

```
WriteLine(e1.ToString());
```

Запустите приложение. Метод `ToString` наследуется от типа `System.Object`. При реализации выводится пространство имен и имя типа, как показано ниже:

```
Packt.CS7.Employee
```

Теперь переопределим поведение класса `Person`.

Изменения вносятся в класс Person, а не Employee.

Visual Studio 2017

В Visual Studio 2017 откройте файл `Person.cs` и в его конце (внутри скобок класса) напечатайте ключевое слово `override`, добавив после него пробел. Visual Studio отобразит список виртуальных методов, которые можно переопределить (рис. 6.5).

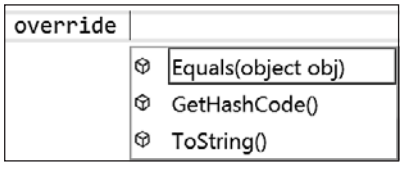


Рис. 6.5

Используя клавиши со стрелками, выберите пункт ToString, а затем нажмите клавишу Enter.

Visual Studio 2017 или Visual Studio Code

В Visual Studio 2017 добавьте в метод инструкцию `return`, а в Visual Studio Code напишите код метода полностью, как показано в листинге ниже:

```
// переопределенные методы
public override string ToString()
{
  return $"{Name} is a {base.ToString()}";
}
```

Запустите консольное приложение и проанализируйте результат вывода. Теперь в момент вызова метод `ToString` выводит имя человека, а также реализацию `ToString` из базового класса, как показано в следующем выводе:

```
John Jones is a Packt.CS7.Employee
```

Многие реальные API, например Microsoft Entity Framework, прокси Castle Windsor и модели контента Episerver, требуют, чтобы свойства, определяемые программистами в своих классах, определялись как виртуальные. Если у вас нет веских оснований поступить иначе, то определяйте свои методы и свойства как виртуальные.

Предотвращение наследования и переопределения

Вы можете предотвратить наследование своего класса, указав в его определении ключевое слово `sealed`. Никто не сможет наследовать *запечатанный* класс `ScroogeMcDuck`:

```
public sealed class ScroogeMcDuck
{
}
```

Примером запечатанного класса может служить string. Корпорация Microsoft внедрила в данный класс ряд критических оптимизаций, на которые наследование может повлиять негативно, и поэтому запечатала его.

Вы можете предотвратить переопределение метода в своем классе, указав имя метода с ключевым словом `sealed`. Никто не сможет переопределить, как поет `LadyGaga`:

```
public class LadyGaga
{
  public sealed void Sing()
  {
  }
}
```

Полиморфизм

Теперь вы знаете два способа изменения поведения унаследованного метода. Его можно скрыть с помощью ключевого слова `new` (то есть выполнить *неполиморфное наследование*) или переопределить (так называемое *полиморфное наследование*).

Оба способа могут вызывать базовый класс с помощью ключевого слова `base`, так в чем же разница?

Все зависит от типа переменной, содержащей ссылку на объект. Например, переменная типа `Person` может включать ссылку на класс `Person` или *любой тип, производный* от класса `Person`.

Добавьте в класс `Employee` код, приведенный ниже:

```
public override string ToString()
{
  return $"{Name}'s code is {EmployeeCode}";
}
```

В методе `Main` напишите такой код:

```
Employee aliceInEmployee = new Employee { Name = "Alice", EmployeeCode = "AA123" };
Person aliceInPerson = aliceInEmployee;
aliceInEmployee.WriteToConsole();
aliceInPerson.WriteToConsole();
WriteLine(aliceInEmployee.ToString());
WriteLine(aliceInPerson.ToString());
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Alice's birth date is 01/01/01 and hire date was 01/01/01
Alice was born on Monday, 1 January 0001
Alice's code is AA123
Alice's code is AA123
```

Обратите внимание: когда метод скрывается с помощью ключевого слова `new`, компилятор «недостаточно умен», чтобы знать, что объект является сотрудником (`Employee`), поэтому вызывает метод `WriteToConsole` класса `Person`.

Если метод переопределяется с помощью ключевых слов `virtual` и `override`, то компилятор «понимает», что, хотя переменная объявлена как класс `Person`, сам объект является `Employee` и поэтому вызывается реализация `Employee` метода `ToString`.

Модификаторы доступа и их влияние перечислены в табл. 6.2.

Таблица 6.2

Тип переменной	Модификатор доступа	Выполненный метод	В классе
Person		WriteToConsole	Person
Employee	new	WriteToConsole	Employee
Person	virtual	ToString	Employee
Employee	override	ToString	Employee

Большинству программистов парадигма полиморфизма кажется малоперспективной с практической точки зрения. Если вы освоите данную концепцию — прекрасно; а если нет — не волнуйтесь. Некоторым программистам нравится заставлять других чувствовать себя ущербными, говоря, что понимание полиморфизма важно, хотя, на мой взгляд, это не так. Вы можете построить блестящую карьеру программиста на C#, будучи неспособным объяснить концепцию полиморфизма, точно так же, как успешный автогонщик не знает, как работает система впрыска.

Приведение в иерархиях наследования

Приведение отличается от преобразования типов.

Неявное приведение

В предыдущем примере показано, как экземпляр производного типа может быть сохранен в переменной базового типа (или базового базового типа и т. д.). Этот процесс называется *неявным приведением*.

Явное приведение

Иногда нужно пойти другим путем и применить *явное приведение*, указав в коде круглые скобки.

Добавьте в метод `Main` следующий код:

```
Employee e2 = aliceInPerson;
```

Обе среды: и Visual Studio 2017, и Visual Studio Code — подчеркнут код красной волнистой чертой и отобразят ошибку компиляции на панели Error List (Список ошибок) (рис. 6.6) и в области PROBLEMS (Проблемы).

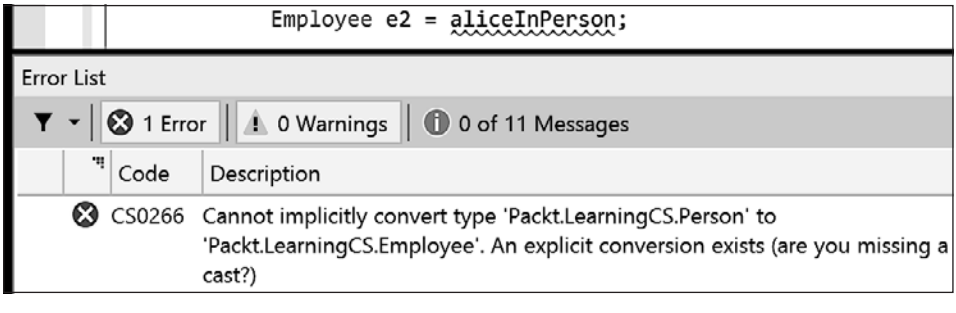


Рис. 6.6

Измените код так, как показано в листинге ниже:

```
Employee e2 = (Employee)aliceInPerson;
```

Обработка исключений приведения

Компилятор теперь не выдает ошибок, *но*, поскольку `aliceInPerson` может быть другим производным типом (например, `Student`, а не `Employee`), нужно соблюдать осторожность. Эта инструкция способна вызвать ошибку `InvalidCastException`.

Справиться с ней можно, написав инструкцию `try`, но существует лучший способ: проверить текущий тип объекта с помощью ключевого слова `is`.

Оберните инструкцию явного приведения инструкцией `if` следующим образом:

```
if (aliceInPerson is Employee)
{
  WriteLine($"{nameof(aliceInPerson)} IS an Employee");
  Employee e2 = (Employee)aliceInPerson;
  // действия с e2
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
aliceInPerson IS an Employee
```

Кроме того, для приведения можно использовать ключевое слово `as`. Вместо вызова исключения ключевое слово `as` возвращает `null`, если тип не может быть приведен.

Добавьте эти инструкции в конец метода `Main`:

```
Employee e3 = aliceInPerson as Employee;
if (e3 != null)
{
  WriteLine($"{nameof(aliceInPerson)} AS an Employee");
  // действия с e3
}
```

Поскольку доступ к переменной `null` может вызвать ошибку `NullReferenceException`, то вы должны всегда проверять значение `null` перед использованием результата.

Запустите консольное приложение и проанализируйте результат вывода:

```
aliceInPerson AS an Employee
```

Используйте ключевые слова is и as, чтобы избежать вызова исключений при приведении типов.

Наследование и расширение типов .NET

Платформа .NET включает готовые библиотеки классов, содержащие сотни тысяч типов. Вместо того чтобы создавать собственные, совершенно новые типы, зачастую достаточно наследовать один из предустановленных типов корпорации Microsoft.

Наследование исключений

В проекте PacktLibrary создайте класс `PersonException`, как показано в коде, приведенном ниже:

```
using System;

namespace Packt.CS7
```

```
{
  public class PersonException : Exception
  {
    public PersonException() : base() { }
    public PersonException(string message) : base(message) { }
    public PersonException(string message, Exception innerException) :
    base(message, innerException) { }
  }
}
```

В отличие от обычных методов конструкторы не наследуются, поэтому мы должны явно объявлять и явно вызывать реализации базового конструктора в System.Exception, чтобы сделать их доступными для программистов, которые смогут использовать эти конструкторы в нашем исключении.

Добавьте в класс `Person` следующий метод:

```
public void TimeTravel(DateTime when)
{
  if (when <= DateOfBirth)
  {
    throw new PersonException("If you travel back in time
    to a date earlier than your own birth then the universe
    will explode!");
  }
  else
  {
    WriteLine($"Welcome to {when:yyyy}!");
  }
}
```

Добавьте в метод `Main` следующие инструкции для определения того, что произойдет, если переместиться в машине времени слишком далеко:

```
try
{
  e1.TimeTravel(new DateTime(1999, 12, 31));
  e1.TimeTravel(new DateTime(1950, 12, 25));
}
catch (PersonException ex)
{
  WriteLine(ex.Message);
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Welcome to 1999!
If you travel back in time to a date earlier than your own birth then the universe
will explode!
```

При определении собственных исключений снабжайте их такими же тремя конструкторами.

Расширение типов при невозможности наследования

Ранее вы узнали, как использовать модификатор `sealed` для предотвращения наследования.

Корпорация Microsoft применила запечатанное ключевое слово `sealed` к классу `System.String`, чтобы никто не смог наследовать его и потенциально нарушить поведение строк.

Можем ли мы добавить в него новые методы? Да, если воспользуемся *методами расширения*, впервые появившимися в версии C# 3.

Использование статических методов для многократного использования

Начиная с первой версии языка C#, можно создавать статические методы для многократного использования, например, возможность проверки того, что строка содержит адрес электронной почты.

В проекте PacktLibrary создайте класс `MyExtensions.cs`, как показано в коде, приведенном ниже, и учтите следующее:

- класс импортирует пространство имен, чтобы обработать регулярные выражения;
- в статическом методе `IsValidEmail` используется тип `Regex` для проверки совпадений с простым шаблоном адреса электронной почты путем поиска допустимых символов перед символом `@` и после него.

О регулярных выражениях вы узнаете из главы 8.

```
using System.Text.RegularExpressions;
namespace Packt.CS7
{
  public class StringExtensions
  {
    public static bool IsValidEmail(string input)
    {
      // используйте простое регулярное выражение для
      // проверки, что строка содержит допустимый email
      return Regex.IsMatch(input, @"[a-zA-Z0-9\.-_]+@[a-zA-Z0-9\.-_]+");
    }
  }
}
```

Добавьте следующие инструкции в нижней части метода `Main`, чтобы проверить два примера адресов электронной почты:

```
string email1 = "pamela@test.com";
string email2 = "ian&test.com";

WriteLine($"{email1} is a valid e-mail address:
```

```
{StringExtensions.IsValidEmail(email1)}.");
WriteLine($"{email2} is a valid e-mail address:
{StringExtensions.IsValidEmail(email2)}.");
```

Запустите приложение и проанализируйте результат вывода:

```
pamela@test.com is a valid e-mail address: True.
ian&test.com is a valid e-mail address: False.
```

Способ работает, но методы расширения могут уменьшить объем кода, который требуется набирать, и упростить использование этой функции.

Использование методов расширения для многократного использования

В класс `StringExtensions` перед его именем добавьте модификатор `static`, а перед типом `string` — модификатор `this`, как показано ниже:

```
public static class StringExtensions
{
  public static bool IsValidEmail(this string input)
  {
```

Эти две поправки сообщают компилятору, что он должен рассматривать код как метод, расширяющий тип `System.String`.

Вернитесь к классу `Program` и добавьте несколько дополнительных инструкций для использования метода в качестве метода расширения:

```
WriteLine($"{email1} is a valid e-mail address:{email1.IsValidEmail()}.");
WriteLine($"{email2} is a valid e-mail address:{email2.IsValidEmail()}.");
```

Обратите внимание на небольшое изменение в синтаксисе. Теперь метод `IsValidEmail` кажется членом экземпляра типа `string` (рис. 6.7).

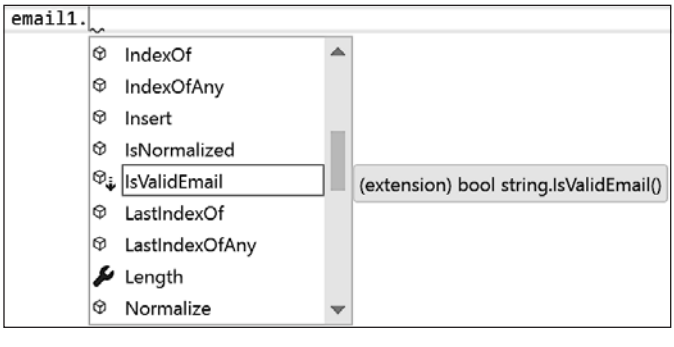


Рис. 6.7

Методы расширения не могут заменить или переопределить используемые методы экземпляра, поэтому нельзя, к примеру, переопределить метод Insert переменной string. Метод расширения будет проявляться как перегрузка, но метод экземпляра будет вызван с приоритетом к методу расширения с тем же именем и сигнатурой.

Хотя методы расширения, на первый взгляд, не дают большого преимущества по сравнению со статическими, в главе 12 вы увидите некоторые чрезвычайно мощные способы их использования.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенное упражнение и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

1. Что такое делегат?
2. Что такое событие?
3. Как связаны базовый и производный классы?
4. В чем разница между ключевыми словами `is` и `as`?
5. Какое ключевое слово используется для предотвращения наследования класса и переопределения метода?
6. Какое ключевое слово служит для предотвращения создания экземпляра класса с помощью нового ключевого слова `new`?
7. Какое ключевое слово применяется для переопределения члена класса?
8. Чем деструктор отличается от деконструктора?
9. Как выглядят сигнатуры конструкторов, которые должны иметь все исключения?
10. Что такое метод расширения и как его определить?

Упражнение 6.1. Создание иерархии наследования

Создайте консольное приложение с именем `Exercise02`.

Создайте класс `Shape` со свойствами `Height`, `Width` и `Area`.

Добавьте три унаследованных класса — `Rectangle`, `Square` и `Circle` — с любыми дополнительными членами, которые, по вашему мнению, подходят и правильно переопределяют и реализуют свойство `Area`.

Дополнительные ресурсы

- Ключевое слово `operator` (справочник по C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/operator.
- Делегаты: https://docs.microsoft.com/en-us/dotnet/articles/csharp/tour-of-csharp/delegates.
- Ключевое слово `event` (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/event.

- Интерфейсы: https://docs.microsoft.com/en-us/dotnet/articles/csharp/tour-of-csharp/interfaces.
- Универсальные типы (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics.
- Ссылочные типы (справочник по C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/reference-types.
- Типы значений (справочник по C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/value-types.
- Наследование (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/classes-and-structs/inheritance.
- Деструкторы (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/classes-and-structs/destructors.

Резюме

В этой главе вы познакомились с делегатами и событиями, научились реализовывать интерфейсы и наследовать типы с помощью объектно-ориентированного программирования. Вы также научились различать базовые и производные классы, переопределять члены типа, использовать полиморфизм и приведение типов.

В следующей части вы узнаете о .NET Core 2.1 и .NET Standard 2.0 и типах, которые они предоставляют для реализации общих функций, таких как обработка файлов, доступ к базам данных, шифрование и многозадачность.

Часть II
.NET Core 2.0
и .NET Standard 2.0

Эта часть книги посвящена функциям API, предоставляемым .NET Core 2.0, и тому, как их кросс-платформенно использовать повторно с помощью .NET Standard 2.0.

Поддержка платформой .NET Core 2.0 стандарта .NET Standard 2.0 важна, поскольку этот стандарт предоставляет большое количество программных интерфейсов .NET Framework API, отсутствовавших в предыдущих версиях .NET Core. Библиотеки и приложения, создаваемые на протяжении 15 лет разработчиками .NET Framework для Windows, теперь могут быть перенесены на .NET Core 2.0 и запущены кросс-платформенно и в версиях для macOS и Linux.

Поддержка API увеличилась на 142 % с появлением .NET Core 2.0: количество доступных интерфейсов API сейчас составляет 32 638 по сравнению с 13 501 в версии .NET Core 1.1!

Полный набор программных интерфейсов .NET Standard 2.0 задокументирован по ссылке https://github.com/dotnet/standard/blob/master/docs/versions/netstandard2.0.md.

Для поиска и просмотра всех .NET API используйте ссылку https://docs.microsoft.com/en- us/dotnet/api/ (рис. II.1).

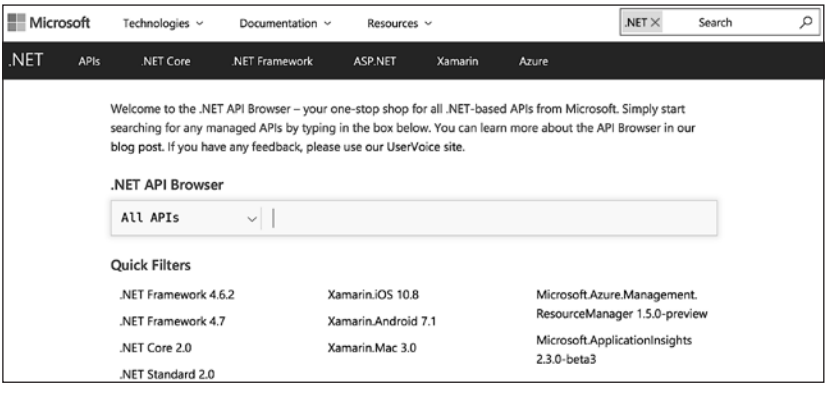


Рис. II.1

В главах, представленных ниже, вы изучите следующий материал.

1. Использование типов .NET Standard 2.0, а также создание и упаковка собственных типов.
2. Применение некоторых часто используемых в приложениях типов .NET Standard 2.0.
3. Работа с файлами и потоками байтов в них.
4. Защита данных с помощью шифрования и других методик.
5. Работа с базами данных.
6. Создание запросов и управление данными.
7. Улучшение производительности, масштабируемости и ресурсного использования вашего кода.

7 Обзор и упаковка типов .NET Standard

Эта глава посвящена платформе .NET Core 2.0 и тому, как в ней реализованы типы, определенные в .NET Standard 2.0. Вы узнаете, каким образом ключевые слова языка C# связаны с типами .NET, а также о взаимоотношениях пространств имен и сборок. Мы обсудим, как упаковать и опубликовать ваши приложения и библиотеки .NET Core для кросс-платформенного использования, как применять существующие библиотеки .NET Framework в библиотеках .NET Standard. Кроме того, изучим возможность переноса кодовых основ .NET Framework в .NET Core.

В данной главе:

- использование сборок и пространств имен;
- кросс-платформенное применение кода с помощью библиотек классов .NET Standard;
- использование пакетов NuGet;
- публикация приложений для развертывания;
- упаковка библиотек для распространения в пакетах NuGet;
- перенос с .NET Framework в .NET Core.

Использование сборок и пространств имен

Платформа .NET Core состоит из следующих частей.

- *Компиляторы языка* превращают ваш исходный код (написанный на языке C#, F#, Visual Basic и др.) в код *промежуточного языка (IL)*, сохраняющийся в сборках

(приложениях и библиотеках классов). В C# 6 был добавлен полностью переписанный компилятор, известный под названием Roslyn.

- *Общеязыковая исполняющая среда (CoreCLR)* загружает сборки, компилирует IL-код, хранящийся в них, в инструкции машинного кода для процессора вашего компьютера и выполняет код в среде с управлением такими ресурсами, как потоки и память.
- *Базовые библиотеки классов и NuGet-пакеты (CoreFX)* — это готовые сборки типов для выполнения универсальных задач при разработке приложений. Вы можете использовать их для быстрого создания всех атрибутов приложений, как если бы конструировали из деталей Lego. Платформа .NET Core 2.0 основана на версии .NET Standard 2.0, включающей функции всех предыдущих версий .NET Standard и поднимающей .NET Core до уровня современных версий .NET Framework и Xamarin.

Стандартные библиотеки классов и CoreFX

Библиотеки предварительно собранного кода, BCL в .NET Framework и CoreFX в .NET Core, состоят из сборок и пространств имен, упрощающих управление десятками тысяч доступных типов. Важно понимать разницу между сборкой и пространством имен.

Сборки, NuGet-пакеты и платформы

Сборки используются для хранения типов в файловой системе. По сути, это механизм для развертывания кода. Например, сборка `System.Data.dll` содержит типы для управления данными. Чтобы использовать типы в других сборках, на них нужно сослаться.

Сборки часто распространяются в виде *NuGet-пакетов*, которые могут содержать несколько сборок и других ресурсов. Вы также услышите разговоры о *метапакетах* и *платформах*, представляющих собой комбинации NuGet-пакетов.

Пространства имен

Пространство имен — это адрес типа. Пространство имен — это механизм уникальной идентификации типа через его полный адрес, а не просто короткое имя.

В реальном мире *Серега* из дома номер *34 по улице Абрикосовой* отличается от *Сереги* из дома номер *12 по улице Виноградной*.

Говоря о .NET Core, интерфейс `IActionFilter` пространства имен `System.Web.Mvc` отличается от интерфейса `IActionFilter` пространства имен `System.Web.Http.Filters`.

Добавление ссылок на зависимые сборки

Если сборка компилируется в виде *библиотеки классов* (предоставляет типы другим сборкам), то получает расширение `.dll` (dynamic link library — библиотека динамической компоновки) и не может выполняться автономно, а только через команду `dotnet run`.

Если сборка компилируется как *приложение*, то получает расширение `.exe` (executable — исполняемый файл) и может выполняться автономно.

Любые сборки (как приложения, так и библиотеки классов) могут ссылаться на одну или несколько сборок, содержащих библиотеку классов, определяя эти связи как *зависимости*, но вы не можете использовать циклические ссылки. То есть сборка Б не может ссылаться на сборку А, если та уже ссылается на сборку Б. Среда разработки Visual Studio предупредит вас о том, что вы пытаетесь добавить ссылку зависимости, создавая при этом циклическую ссылку.

Циклические ссылки нередко являются признаком плохо написанного кода. Если вы уверены, что вам необходимы именно циклические ссылки, то воспользуйтесь советом, описанным на сайте https://stackoverflow.com/questions/6928387/how-to-solve-circular-reference.

Любое приложение, созданное средствами .NET Core, зависит от *платформы приложений Microsoft .NET Core*. Эта специальная платформа содержит тысячи типов, доступных в виде NuGet-пакетов, которые требуются при разработке практически всех приложений.

Visual Studio 2017

Запустите Microsoft Visual Studio 2017. Нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Core. В центре диалогового окна выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)), присвойте ему имя `Assemblies`, укажите расположение по адресу `C:\Code`, введите имя решения `Chapter07`, а затем нажмите кнопку OK.

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте Assemblies и выберите пункт Edit Assemblies.csproj (Редактировать Assemblies.csproj).

Visual Studio Code

В Visual Studio Code используйте панель Integrated Terminal (Интегрированный терминал), чтобы создать каталог `Chapter07` с подкаталогом `Assemblies`.

Выполните команду `dotnet new console` для создания консольного приложения.

Откройте файл `Assemblies.csproj`.

Visual Studio 2017 и Visual Studio Code

При работе с .NET Core вы ссылаетесь на зависимые сборки, NuGet-пакеты и платформы, которые необходимы для работы вашего приложения, в файле проекта.

Изначально файлы проектов.NET Core 1.0 имели формат JSON и имя project.json. На платформе .NET Core 1.1 и более поздних версий стали использоваться обновленные файлы формата XML с расширением .csproj. Я сказал «обновленные», так как, по сути, это еще более старый формат, который применялся с момента выпуска .NET в 2002 году. Корпорация Microsoft изменила формат после выхода .NET Core 1.0, а затем вновь вернулась к старому!

Файл `Assemblies.csproj` — типичный представитель файла проекта приложения .NET Core, что продемонстрировано следующей разметкой:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Связанные сборки и пространства имен

Чтобы разобраться во взаимосвязях между сборками и пространствами имен, мы воспользуемся возможностями Visual Studio 2017. Если вы работаете в Visual Studio Code, то просто изучите приведенные далее рисунки.

Просмотр сборок в Visual Studio 2017

В Visual Studio 2017 выполните команду View ▸ Object Browser (Вид ▸ Обозреватель объектов) (или нажмите сочетание клавиш Ctrl+W, J). Вы увидите, что ваше решение зависит от разных сборок, таких как `System.Collections` и `System.Console` (используется во всех упражнениях) (рис. 7.1).

Панель Object Browser (Обозреватель объектов) может применяться для получения информации о сборках и пространствах имен, которые .NET Core использует для логической и физической группировки типов.

Для типов, применяемых только в ряде ситуаций (примером послужит тип `Console`, используемый только в консольных приложениях, но не в веб- или мобильных приложениях), существует сборка только для этого типа, а также поддерживаемых им типов. Сборка `System.Console.dll` находится в файловой системе (рис. 7.2).

Сборка `System.Console.dll` содержит только восемь типов, и все они находятся в пространстве имен `System` и поддерживают тип `Console` (рис. 7.3).

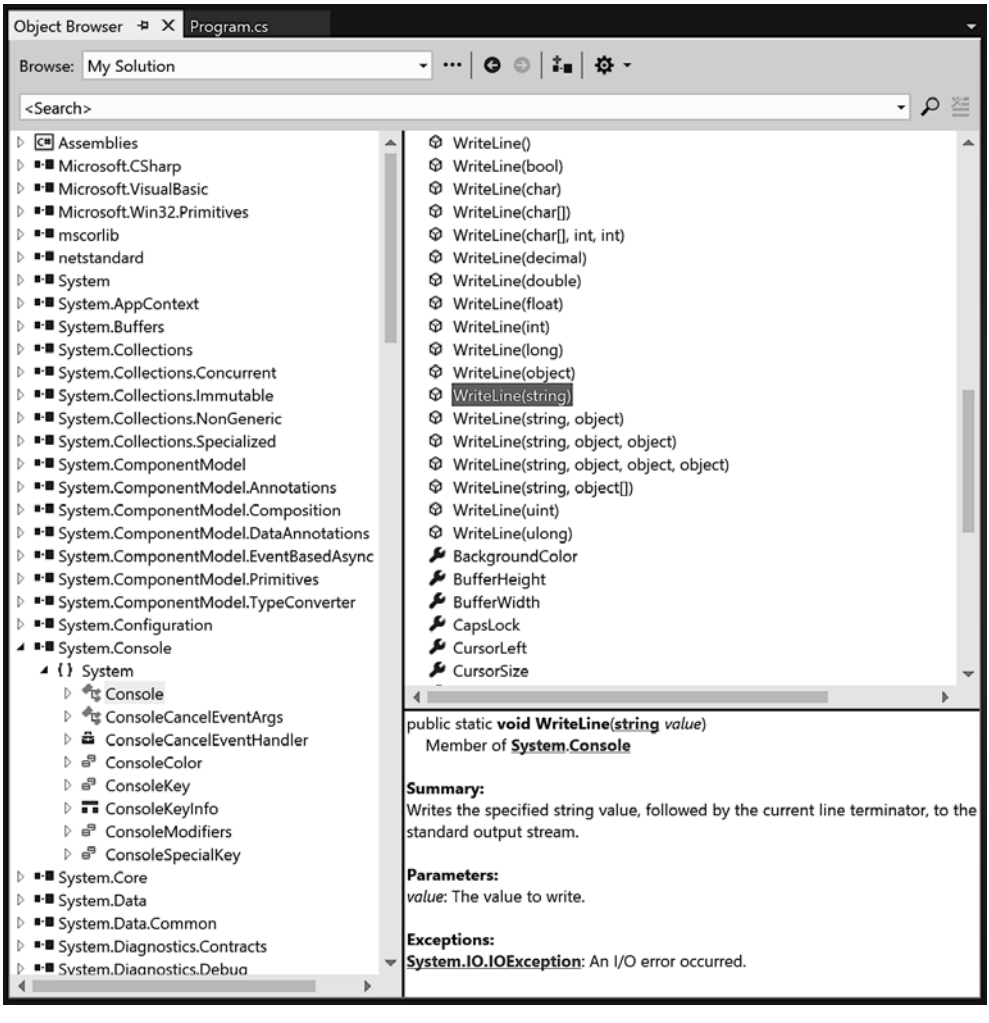

Object Browser
Program.cs
Browse: My Solution
<Search>
Assemblies
Microsoft.CSharp
Microsoft.VisualBasic
Microsoft.Win32.Primitives
mscorlib
netstandard
System
System.AppContext
System.Buffers
System.Collections
System.Collections.Concurrent
System.Collections.Immutable
System.Collections.NonGeneric
System.Collections.Specialized
System.ComponentModel
System.ComponentModel.Annotations
System.ComponentModel.Composition
System.ComponentModel.DataAnnotations
System.ComponentModel.EventBasedAsync
System.ComponentModel.Primitives
System.ComponentModel.TypeConverter
System.Configuration
System.Console
System
Console
ConsoleCancelEventArgs
ConsoleCancelEventHandler
ConsoleColor
ConsoleKey
ConsoleKeyInfo
ConsoleModifiers
ConsoleSpecialKey
System.Core
System.Data
System.Data.Common
System.Diagnostics.Contracts
System.Diagnostics.Debug
WriteLine()
WriteLine(bool)
WriteLine(char)
WriteLine(char[])
WriteLine(char[], int, int)
WriteLine(decimal)
WriteLine(double)
WriteLine(float)
WriteLine(int)
WriteLine(long)
WriteLine(object)
WriteLine(string)
WriteLine(string, object)
WriteLine(string, object, object)
WriteLine(string, object, object, object)
WriteLine(string, object[])
WriteLine(uint)
WriteLine(ulong)
BackgroundColor
BufferHeight
BufferWidth
CapsLock
CursorLeft
CursorSize
public static void WriteLine(string value)
Member of System.Console
Summary:
Writes the specified string value, followed by the current line terminator, to the standard output stream.
Parameters:
value: The value to write.
Exceptions:
System.IO.IOException: An I/O error occurred.

Рис. 7.1

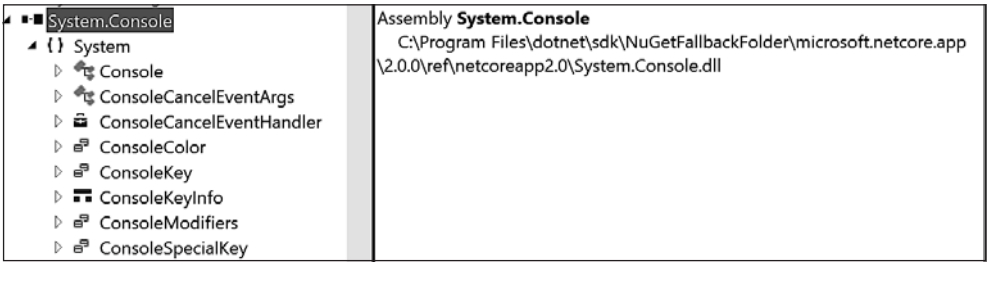

System.Console
System
Console
ConsoleCancelEventArgs
ConsoleCancelEventHandler
ConsoleColor
ConsoleKey
ConsoleKeyInfo
ConsoleModifiers
ConsoleSpecialKey
Assembly System.Console
C:\Program Files\dotnet\sdk\NuGetFallbackFolder\microsoft.netcore.app\2.0.0\ref\netcoreapp2.0\System.Console.dll

Рис. 7.2

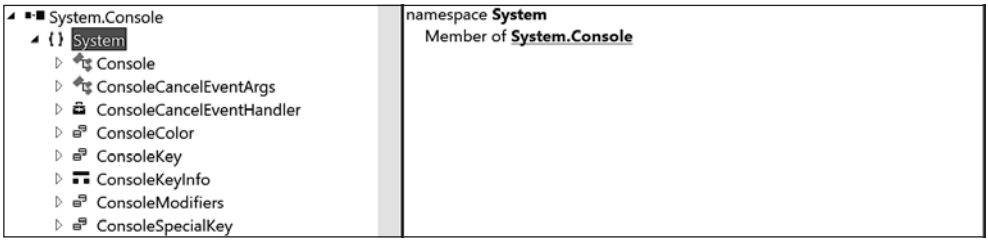


Рис. 7.3

По умолчанию панель Object Browser (Обозреватель объектов) отображает типы, сгруппированные по сборкам, то есть файл, *содержащий* все пространства имен и типы в файловой системе. Иногда полезно не обращать внимания на *физическое* местоположение типа и сконцентрировать внимание на его месте в *логической* группировке, то есть в пространстве имени.

На панели Object Browser (Обозреватель объектов) нажмите кнопку Object Browser Settings (Параметры обозревателя объектов) (на панели это последняя кнопка со значком шестеренки) и выберите пункт View Namespaces (Просмотреть пространства имен) (рис. 7.4).

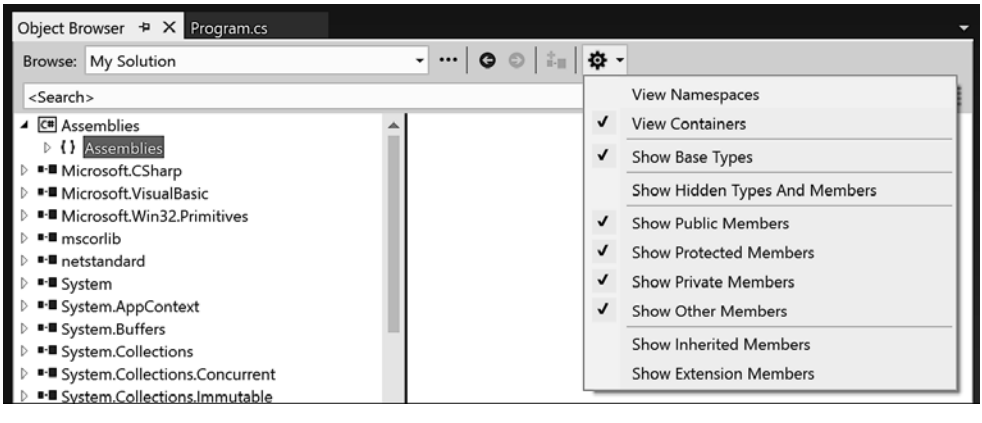


Рис. 7.4

Теперь панель Object Browser (Обозреватель объектов) отображает типы, сгруппированные по сборкам. Теперь при выборе нужного пространства имен (например, `System`) в обозревателе отобразится список сборок, в которых реализованы типы, попадающие в выбранное пространство имен (рис. 7.5).

Наиболее часто используемые типы .NET Core находятся в сборке `System.Runtime.dll`. Вы можете просмотреть взаимоотношения отдельных сборок и пространств имен, для которых эти сборки предоставляют типы. Обратите также внимание, что между сборками и пространствами соотношение всегда один к одному, как показано в табл. 7.1.

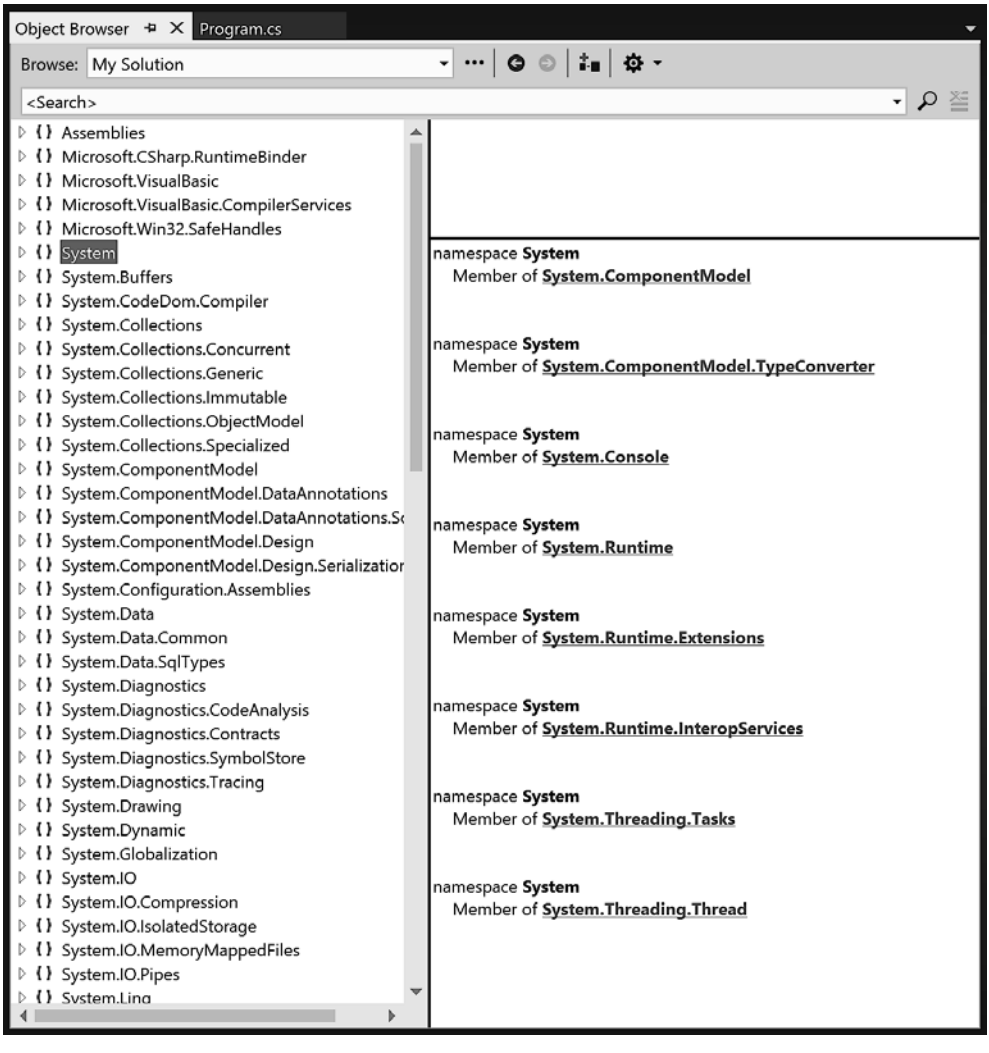


Рис. 7.5

Таблица 7.1

Сборка	Пример пространства имен	Пример типа
System.Runtime.dll	System, System.Collections, System.Collections.Generics	Int32, String, List<T>
System.Console.dll	System	Console
System.Threading.dll	System.Threading	Interlocked, Monitor, Mutex
System.Xml.XDocument.dll	System.Xml.Linq	XDocument, XElement, XNode

Visual Studio 2017 или Visual Studio Code

В Visual Studio 2017 или Visual Studio Code в методе `Main` наберите код, приведенный ниже:

```
var doc = new XDocument();
```

Тип `XDocument` не будет распознан, поскольку мы не сообщили компилятору пространство имен этого типа. Хотя в данном проекте уже есть ссылка на сборку, содержащую указанный тип, также нужно либо указать пространство имен перед именем типа, либо импортировать это пространство.

Импорт пространства имен

Установите указатель мыши внутри имени класса `XDocument`. И Visual Studio 2017, и Visual Studio Code отобразит лампочку, указывающую, что тип распознан и проблему можно решить автоматически средствами программы.

Нажмите кнопку в виде светящейся лампочки или сочетание клавиш Ctrl+. (точка) (в Visual Studio 2017) или Cmd+. (точка) (в Visual Studio Code).

Среда разработки Visual Studio 2017 отобразит более подробные сведения о доступных вариантах решения проблемы с предварительным просмотром предлагаемых изменений (рис. 7.6).

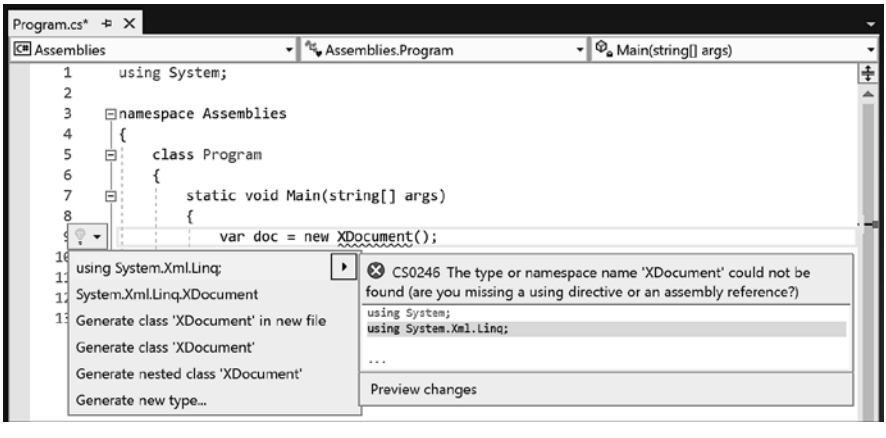


Рис. 7.6

Среда разработки Visual Studio Code не отобразит подробные сведения, но предложит те же варианты решения проблемы (рис. 7.7).

Выберите в контекстном меню пункт using System.Xml.Linq;. Так вы *импортируете пространство имен* с помощью инструкции `using`, которая появится в начале файла.

После того как пространство имен будет импортировано, а соответствующий код появится в начале файла, все типы, имеющиеся в пространстве, станут доступны для использования в данном файле. Вы сможете применять их, указывая имена этих типов.

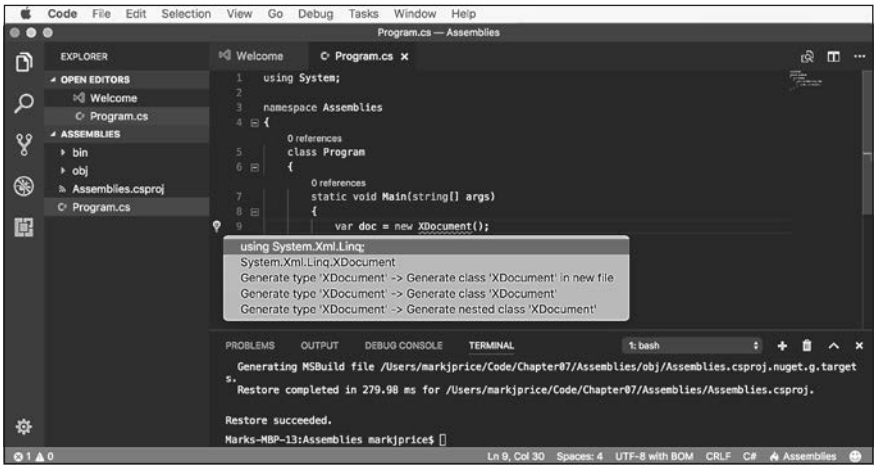


Рис. 7.7

Связывание ключевых слов C# с типами .NET

Один из самых распространенных вопросов начинающих программистов на C# звучит так: *«Есть ли разница между словом string со строчной буквы и словом String с прописной?»*

Краткий ответ: нет.

Более развернутый ответ: все ключевые слова типов в C# являются псевдонимами для типов .NET в сборке библиотеки классов.

Использованное в коде ключевое слово `string` компилятор преобразует в тип `System.String`, а тип `int` преобразует в тип `System.Int32`. Вы даже можете убедиться в этом собственными глазами, наведя указатель мыши на тип `int` (рис. 7.8).

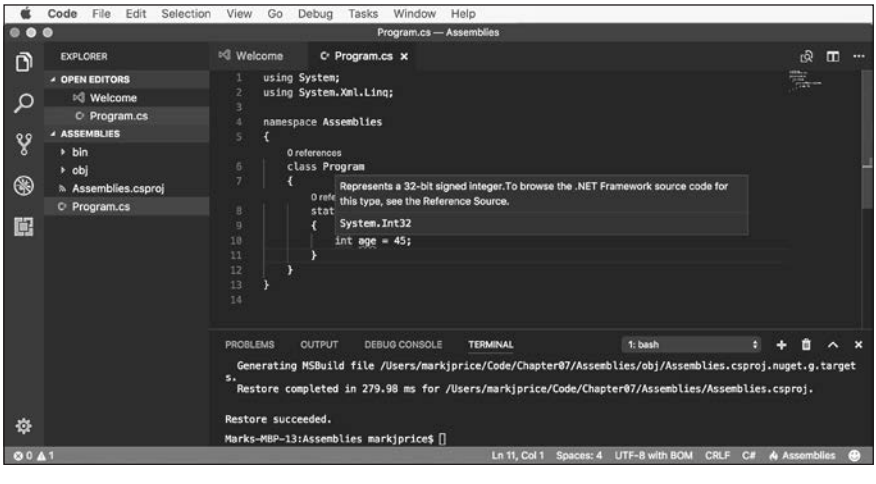


Рис. 7.8

Применяйте ключевые слова C# вместо явного указания типа, поскольку в этом случае не нужно импортировать пространство имен.

В табл. 7.2 перечислены 16 ключевых слов языка C# и их соответствующие типы .NET.

Таблица 7.2

Ключевое слово	Тип .NET	Ключевое слово	Тип .NET
string	System.String	char	System.Char
sbyte	System.SByte	byte	System.Byte
short	System.Int16	ushort	System.UInt16
int	System.Int32	uint	System.UInt32
long	System.Int64	ulong	System.UInt64
float	System.Single	double	System.Double
decimal	System.Decimal	bool	System.Boolean
object	System.Object	dynamic	System.Dynamic.DynamicObject

Компиляторы других языков программирования .NET способны на то же самое. Например, язык Visual Basic .NET содержит тип Integer, который является псевдонимом для типа System.Int32.

Совместное кросс-платформенное использование кода с помощью библиотек классов .NET Standard 2.0

Прежде чем появился .NET Standard 2.0, в ходу были *портативные библиотеки классов* (Portable Class Libraries, PCL). С их помощью можно было создать библиотеку кода и явно указать, на каких платформах ей требуется поддержка, например Xamarin, Silverlight, Windows 8 и т. д. Затем ваша библиотека могла использовать пересечение возможностей API, поддерживаемых указанными платформами.

Сотрудники корпорации Microsoft поняли, что это неустойчивая ветвь развития, вследствие чего было решено начать работу над .NET Standard 2.0 — единым API, который будет поддерживаться всеми будущими платформами .NET. Существуют более ранние версии .NET Standard, но они не поддерживаются сразу несколькими платформами .NET.

.NET Standard 2.0 схож с HTML5, поскольку они оба представляют собой стандарты, которые платформа должна поддерживать. Так же как браузеры Google Chrome и Microsoft Edge поддерживают стандарт HTML5, .NET Core и Xamarin поддерживают .NET Standard 2.0.

Создать библиотеку типов, которые будут поддерживаться и .NET Framework (Windows), и .NET Core (Windows, macOS и Linux), и Xamarin (iOS, Android и Windows Mobile), проще всего с помощью .NET Standard 2.0.

В табл. 7.3 представлены версии .NET Standard и платформы, поддерживаемые каждой версией.

Таблица 7.3

Платформа	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	→	→	→	→	→	1.0, 1.1	2.0
.NET Framework	4.5	4.5.1	4.6	→	→	→	4.6.1
Xamarin/Mono	→	→	→	→	→	4.6	5.4
UWP	→	→	→	10	→	→	6.0

Создание библиотеки классов .NET Standard 2.0

Мы создадим библиотеку классов .NET Standard 2.0, чтобы ее можно было использовать независимо от платформы в операционной системе Windows, macOS и Linux.

Visual Studio 2017

Запустите Microsoft Visual Studio 2017.

Нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ Add ▸ New project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Standard. В центре диалогового окна выберите пункт Class Library (.NET Standard) (Библиотека классов (.NET Standard)), присвойте ему имя `SharedLibrary`, укажите расположение по адресу `C:\Code\Chapter07`, а затем нажмите кнопку OK (рис. 7.9).

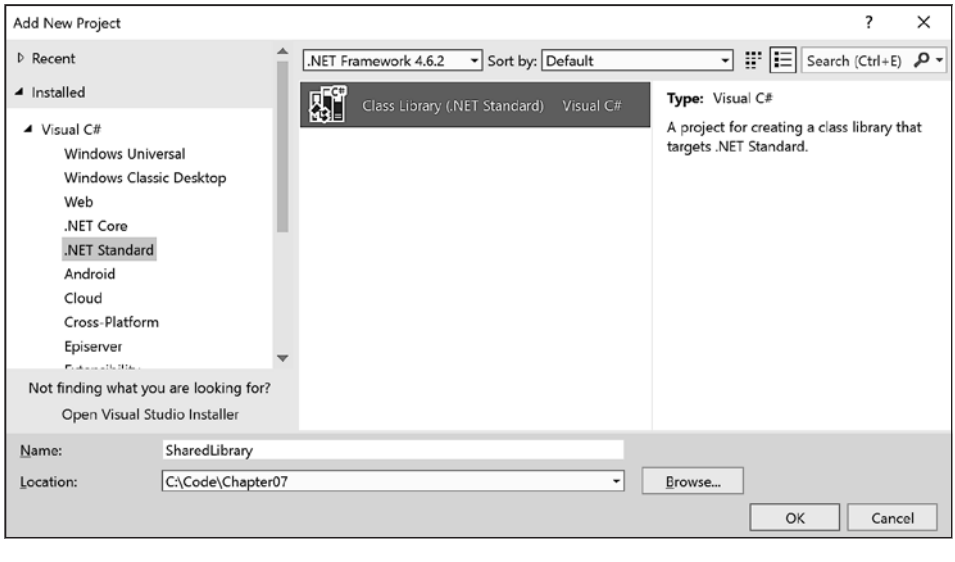


Рис. 7.9

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте SharedLibrary и выберите пункт Edit SharedLibrary.csproj (Редактировать SharedLibrary.csproj).

Обратите внимание, что библиотека классов .NET Standard по умолчанию ссылается на версию 2.0, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Visual Studio Code

В каталоге `Code/Chapter07` создайте подкаталог `SharedLibrary`.

Запустите Visual Studio Code и откройте каталог `SharedLibrary`.

Выполните команду View ▸ Integrated Terminal (Вид ▸ Интегрированный терминал), а затем введите в области TERMINAL (Терминал) такую команду:

```
dotnet new classlib
```

Выберите файл `SharedLibrary.csproj` и обратите внимание, что библиотека классов по умолчанию сгенерирована с помощью CLI-инструмента `dotnet` версии 2.0, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Использование NuGet-пакетов

Платформа .NET Core разделена на несколько пакетов, распределенных с помощью технологии управления пакетами Microsoft под названием NuGet. Каждый из этих пакетов представляет собой отдельную сборку с тем же именем, что и у пакета. Например, пакет `System.Collections` содержит сборку `System.Collections.dll`.

Рассмотрим преимущества использования пакетов:

- производитель может распространять их по собственному расписанию;
- каждый пакет можно тестировать независимо от других пакетов;
- пакеты могут поддерживать различные операционные системы и процессорные архитектуры;
- пакеты могут иметь зависимости, характерные только для определенной библиотеки;

- приложения имеют меньший размер, поскольку неиспользуемые пакеты исключаются из дистрибутива.

В табл. 7.4 перечислены некоторые наиболее важные пакеты.

Таблица 7.4

Пакет	Важные типы
System.Runtime	Object, String, Array
System.Collections	List<T>, Dictionary<TKey, TValue>
System.Net.Http	HttpClient, HttpResponseMessage
System.IO.FileSystem	File, Directory
System.Reflection	Assembly, TypeInfo, MethodInfo

Метапакеты

Метапакеты представляют собой наборы совместно использующихся пакетов. На метапакеты ссылаются так же, как и на любой другой NuGet-пакет. Ссылаясь на метапакет, вы фактически добавляете ссылки на каждый из вложенных пакетов.

Среда разработки Visual Studio 2017 отлично демонстрирует взаимосвязь между метапакетами, пакетами и сборками (рис. 7.10).

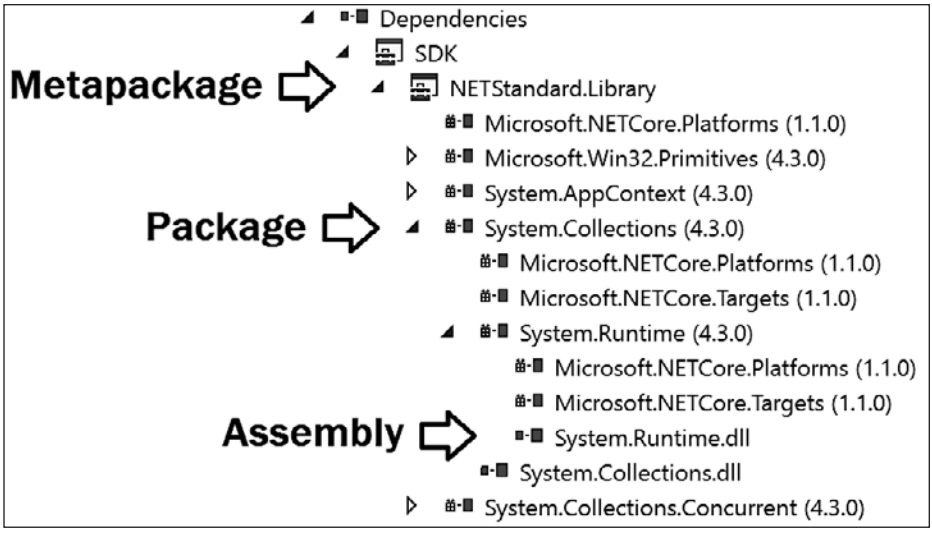


Рис. 7.10

Как вы можете убедиться, в документации Microsoft метапакеты часто упоминаются под названием «пакеты».

Список, приведенный ниже, содержит ссылки на некоторые универсальные метапакеты и пакеты, в том числе официальный список их зависимостей:

- https://www.nuget.org/packages/Microsoft.NETCore.App;
- https://www.nuget.org/packages/NETStandard.Library;
- https://www.nuget.org/packages/Microsoft.NETCore.Runtime.CoreCLR;
- https://www.nuget.org/packages/System.IO;
- https://www.nuget.org/packages/System.Collections;
- https://www.nuget.org/packages/System.Runtime.

Перейдя по первой ссылке, вы увидите информацию, показанную на рис. 7.11 (включая сведения о процессе установки, зависимостях, истории версий, а также о количестве загрузок).

Раскрыв пункт Dependencies (Зависимости), вы увидите список зависимостей метапакета (рис. 7.12).

Обратите внимание: у пакета Microsoft.NETCore.App версии 2.0.0 есть зависимость от пакета NETStandard.Library версии 2.0.0.

Платформы

Между платформами и пакетами существует двусторонняя связь. Пакеты определяют API, а платформы группируют пакеты. Платформа без пакетов не сможет определить никакой API.

Если вы глубоко разбираетесь в интерфейсах и реализующих их типах, можете посетить сайт https://gist.github.com/davidfowl/8939f305567e1755412d6dc0b8baf1b7, чтобы разобраться, каким образом пакеты и соответствующие API связаны с такими платформами, как различные версии .NET Standard.

Каждый из пакетов .NET Core поддерживает определенный набор платформ. К примеру, пакет `System.IO.FileSystem` поддерживает следующие платформы (рис. 7.13):

- .NET Standard версии 1,3;
- .NET Framework версии 4,6;
- шесть платформ Xamarin (к примеру, Xamarin.iOS 10).

Используйте NETStandard.Library, если создаете библиотеку классов, на которую должны ссылаться несколько платформ, таких как .NET Framework и Xamarin, а также .NET Core.

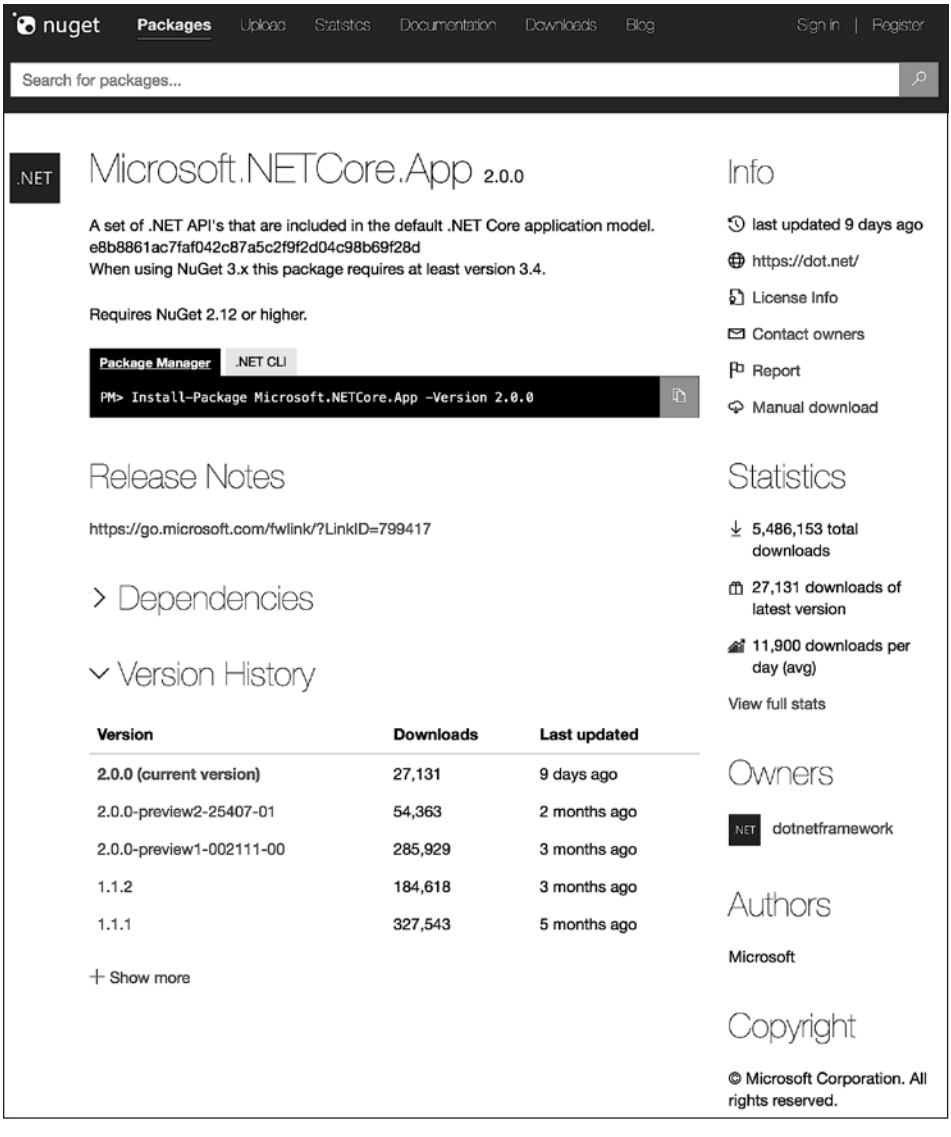


Рис. 7.11

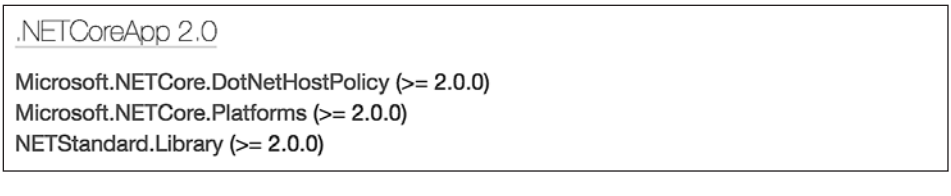


Рис. 7.12

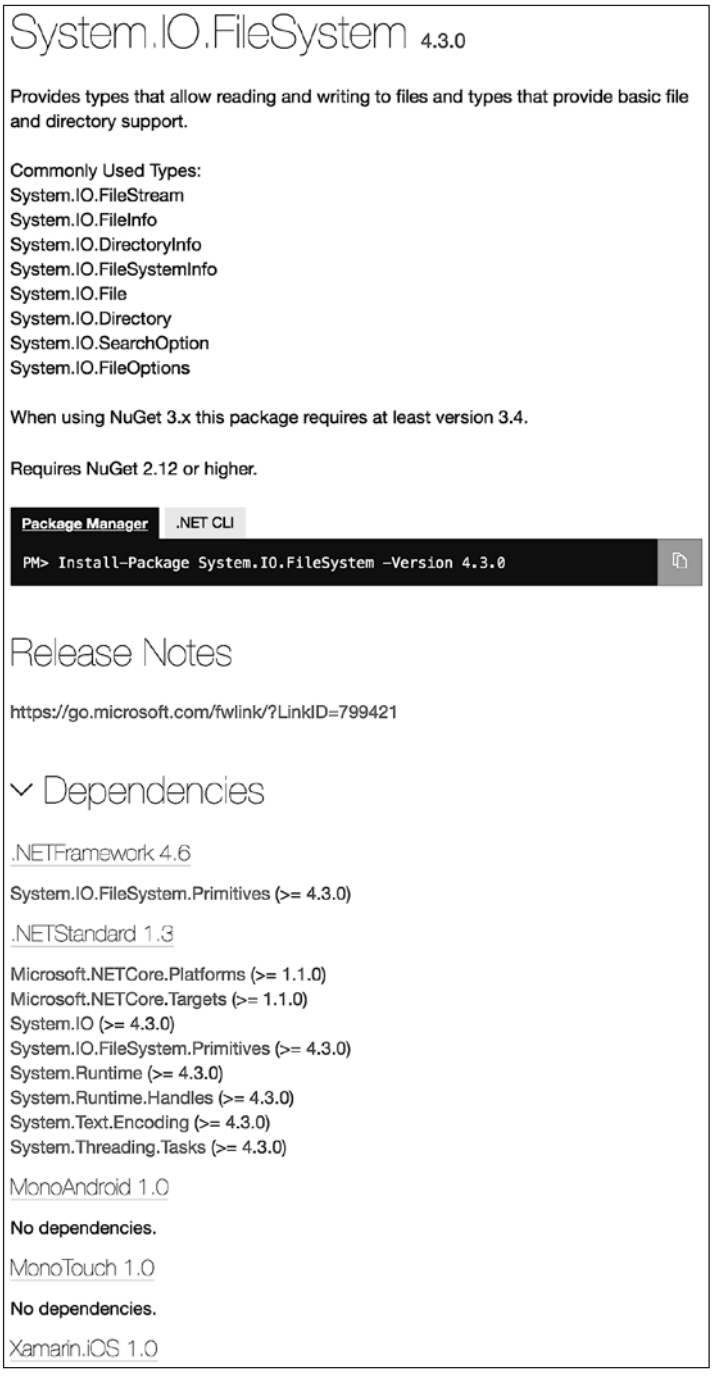

System.IO.FileSystem 4.3.0
Provides types that allow reading and writing to files and types that provide basic file and directory support.
Commonly Used Types:
System.IO.FileStream
System.IO.FileInfo
System.IO.DirectoryInfo
System.IO.FileSystemInfo
System.IO.File
System.IO.Directory
System.IO.SearchOption
System.IO.FileOptions
When using NuGet 3.x this package requires at least version 3.4.
Requires NuGet 2.12 or higher.
Package Manager
.NET CLI
PM> Install-Package System.IO.FileSystem -Version 4.3.0
Release Notes
https://go.microsoft.com/fwlink/?LinkID=799421
Dependencies
.NETFramework 4.6
System.IO.FileSystem.Primitives (>= 4.3.0)
.NETStandard 1.3
Microsoft.NETCore.Platforms (>= 1.1.0)
Microsoft.NETCore.Targets (>= 1.1.0)
System.IO (>= 4.3.0)
System.IO.FileSystem.Primitives (>= 4.3.0)
System.Runtime (>= 4.3.0)
System.Runtime.Handles (>= 4.3.0)
System.Text.Encoding (>= 4.3.0)
System.Threading.Tasks (>= 4.3.0)
MonoAndroid 1.0
No dependencies.
MonoTouch 1.0
No dependencies.
Xamarin.iOS 1.0

Рис. 7.13

Исправление зависимостей

Последовательно восстанавливать пакеты и писать надежный код поможет исправление зависимостей. Оно подразумевает использование одного и того же семейства пакетов, выпущенных для определенной версии .NET Core, к примеру 1.0.

Чтобы исправить зависимости, каждый пакет должен иметь одну версию без дополнительных квалификаторов. К таковым относятся кандидаты на выпуск (`rc4`) и подстановочные символы (`*`). Эти знаки позволяют автоматически ссылаться на будущие версии и использовать их, поскольку они всегда указывают на самую последнюю версию. Подстановочные символы особенно опасны, так как могут привести к восстановлению несовместимых пакетов и сделать код неработоспособным.

Следующие зависимости *не* исправлены, и их нужно избегать:

```
<PackageReference Include="System.Net.Http" Version="4.1.0-*" />
<PackageReference Include="Microsoft.NETCore.App" Version="1.0.0-rc4-00454-00" />
```

Сотрудники корпорации Microsoft гарантируют совместную работу пакетов, если вы исправили свою зависимость на ту, что поставляется с определенной версией .NET Core, к примеру 2.0. Всегда исправляйте свои зависимости.

Публикация приложений

Существует два способа публикации и развертывания приложений .NET Core:

- платформозависимый;
- автономный.

При развертывании приложения с соответствующими зависимостями, но не самой платформы .NET Core вам придется полагаться на то, что она уже развернута на целевом компьютере. Способ подойдет для веб-приложений, разворачиваемых на сервере, поскольку эта платформа и множество других веб-приложений, вероятно, уже установлены на сервере.

Иногда может понадобиться передать клиенту USB-диск с приложением и быть уверенными, что он сумеет запустить его на своем компьютере. Для этого желательно выполнить автономное развертывание. Дистрибутив будет больше, но вы сможете быть уверены в успешном запуске.

Подготовка консольного приложения к публикации

Создайте новый проект консольного приложения с именем `DotNetCoreEverywhere`.

Измените его код следующим образом:

```
using static System.Console;

namespace DotNetCoreEverywhere
```

```
{
  class Program
  {
    static void Main(string[] args)
    {
      WriteLine("I can run everywhere!");
    }
  }
}
```

Откройте файл DotNetCoreEverywhere.csproj и внутри элемента <PropertyGroup> добавьте идентификаторы уведомлений среды выполнения для четырех целевых операционных систем, как показано в коде ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <RuntimeIdentifiers>
      win10-x64;osx.10.12-x64;rhel.7-x64;ubuntu.14.04-x64
    </RuntimeIdentifiers>
  </PropertyGroup>

</Project>
```

Идентификатор win10-64 означает операционную систему Windows 10 или Windows Server 2016. Идентификатор osx.10.12-x64 означает операционную систему macOS Sierra. Полный список поддерживаемых идентификаторов уведомлений среды выполнения доступен по адресу https://docs.microsoft.com/en-us/dotnet/articles/core/rid-catalog.

Публикация в Visual Studio 2017

В Visual Studio 2017 щелкните правой кнопкой мыши на проекте DotNetCoreEverywhere и в контекстном меню выберите пункт Folder (Папка) и нажмите кнопку Publish (Опубликовать) (рис. 7.14).

По умолчанию вы готовы опубликовать версию приложения для 64-разрядной операционной системы Windows 10 (рис. 7.15).

Щелкните на ссылке Settings (Настройки) и измените значение в поле Target Runtime (Целевая среда выполнения) на osx.10.12-x64 (рис. 7.16). Нажмите кнопку Save (Сохранить).

Нажмите кнопку Publish (Опубликовать).

На панели Solution Explorer (Обозреватель решений) нажмите кнопку Show All Files (Показать все файлы), разверните папки bin ▸ Release ▸ netcoreapp2.0 ▸ osx.10.12-x64 и bin ▸ Release ▸ netcoreapp2.0 ▸ win10-x64 (рис. 7.17) и обратите внимание на файлы приложений.

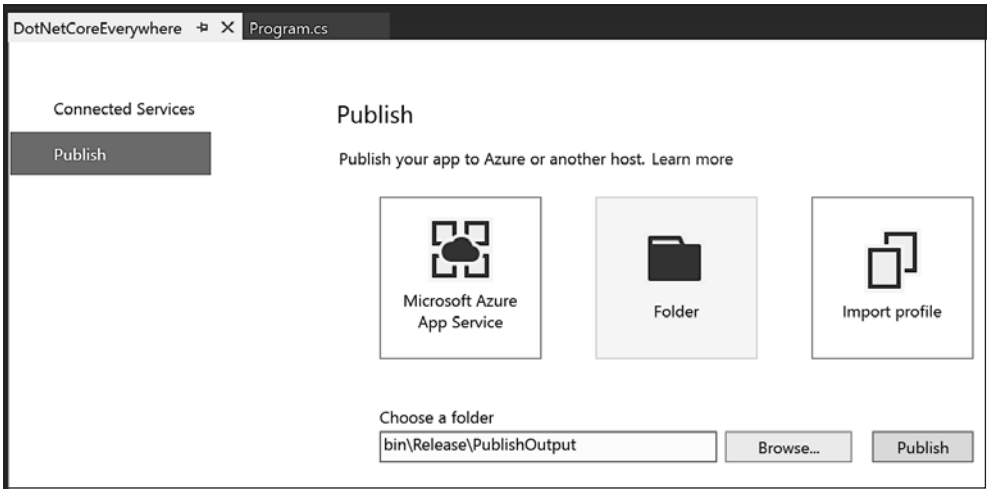


Рис. 7.14

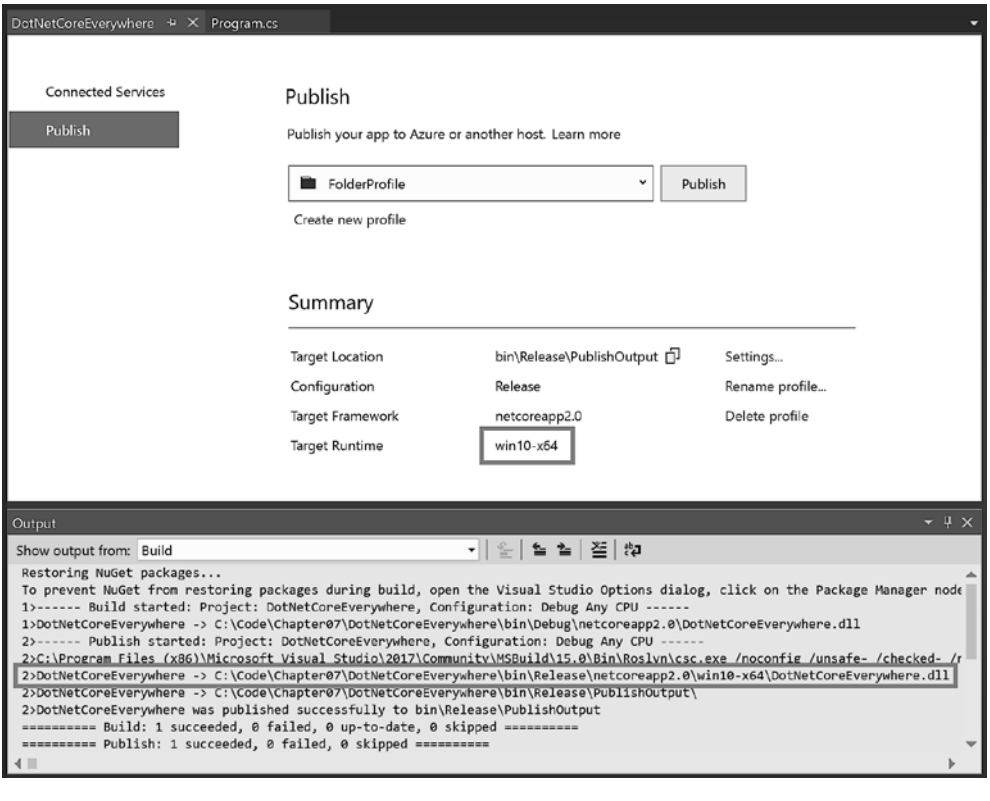


Рис. 7.15

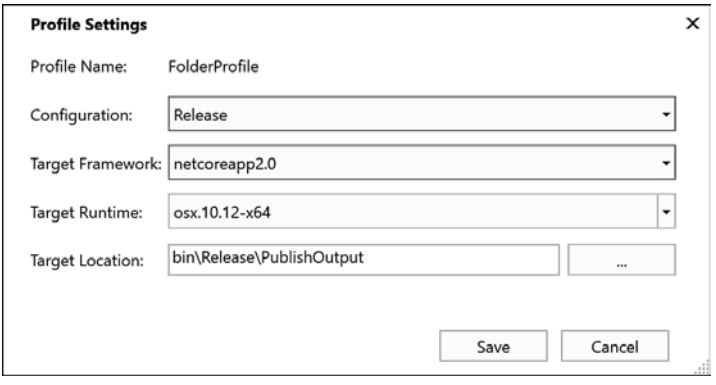


Рис. 7.16

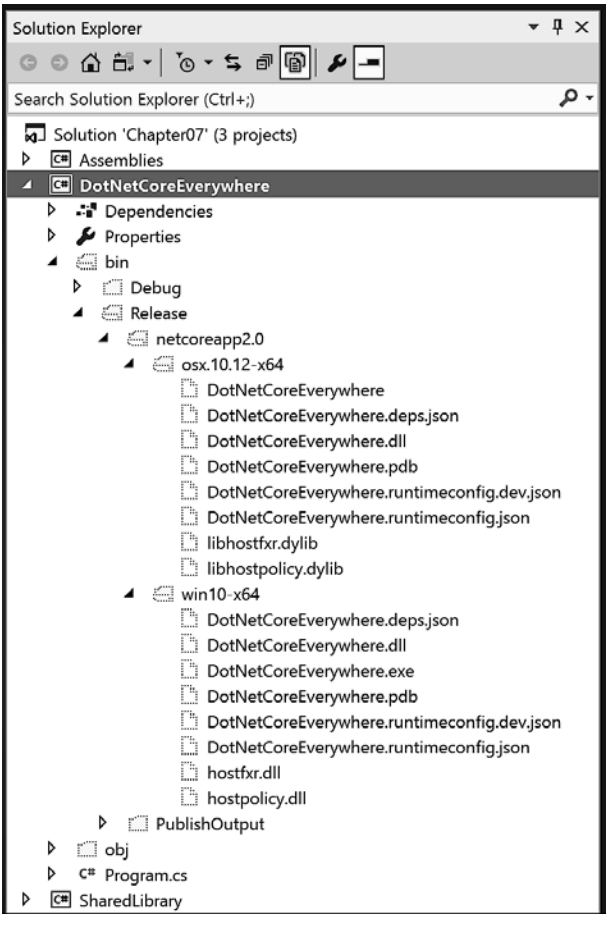


Рис. 7.17

Публикация в Visual Studio Code

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) введите указанную ниже команду, чтобы опубликовать релиз приложения для операционной системы Windows 10:

```
dotnet publish -c Release -r win10-x64
```

Движок Microsoft Build Engine скомпилирует и опубликует консольное приложение (рис. 7.18).

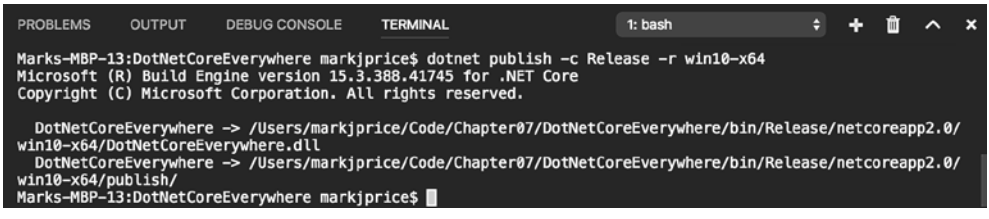


Рис. 7.18

На панели Integrated Terminal (Интегрированный терминал) введите указанные ниже команды, чтобы опубликовать версии приложения для операционных систем macOS, Red Hat Enterprise Linux (RHEL) и Ubuntu Linux:

```
dotnet publish -c Release -r osx.10.12-x64
dotnet publish -c Release -r rhel.7-x64
dotnet publish -c Release -r ubuntu.14.04-x64
```

Откройте окно macOS-приложения Finder, перейдите к каталогу `DotNetCoreEverywhere\bin\Release\netcoreapp2.0` и обратите внимание на созданные каталоги для четырех операционных систем и файлы, в том числе исполняемый файл Windows `DotNetCoreEverywhere.exe` (рис. 7.19).

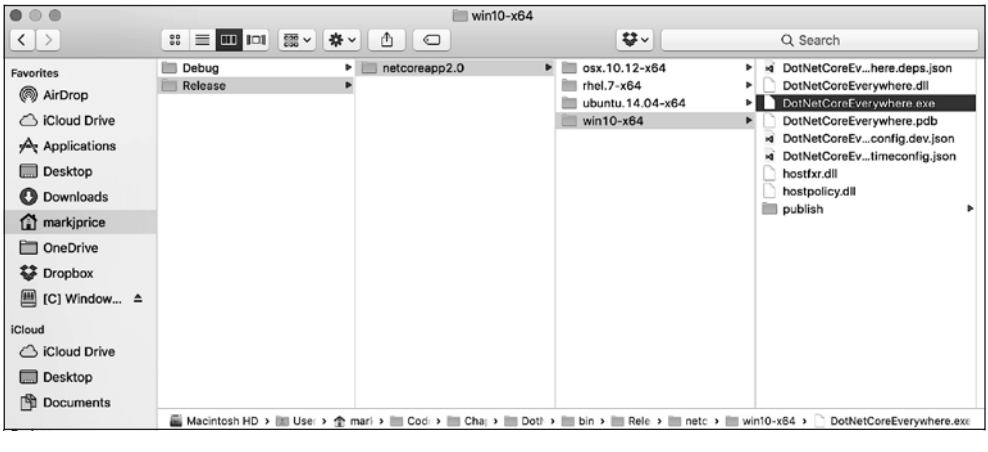


Рис. 7.19

Если вы скопируете каждый из этих каталогов в соответствующую операционную систему, то консольное приложение можно будет запустить, так как оно представляет собой автономное развертываемое приложение .NET Core.

Упаковка библиотек для распространения с помощью NuGet

Установленный пакет .NET Core SDK включает цепочку инструментов *интерфейса командной строки* (Command Line Interface, CLI) под названием `dotnet`.

Команды dotnet

Интерфейс командной строки `dotnet` позволяет выполнять различные команды в целях создания с помощью шаблонов новых проектов в текущем каталоге. Некоторые из этих команд перечислены ниже:

- `dotnet new console` — создает проект нового консольного приложения;
- `dotnet new classlib` — создает проект новой библиотеки классов;
- `dotnet new web` — создает новый пустой проект ASP.NET Core;
- `dotnet new mvc` — создает новый проект веб-приложения ASP.NET Core MVC;
- `dotnet new razor` — создает новый проект ASP.NET Core MVC с поддержкой технологии Razor Pages;
- `dotnet new angular` — создает новый проект ASP.NET Core MVC с поддержкой технологии Angular Single Page Application в качестве фронтенда;
- `dotnet new react` — создает новый проект ASP.NET Core MVC с поддержкой технологии React Single Page Application (SPA) в качестве фронтенда;
- `dotnet new webapi` — создает новый проект ASP.NET Core Web API.

Дополнительные шаблоны можно установить, перейдя по ссылке https://github.com/dotnet/templating/wiki/Available-templatesfor-dotnet-new.

Вы можете определять собственные шаблоны проектов, как сказано в официальной документации по команде dotnet new на сайте https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-new?tabs=netcore2x.

Выполните команду `dotnet new -l` для вывода списка текущих установленных шаблонов (рис. 7.20).

Для управления проектом в текущем каталоге CLI `dotnet` содержит следующие команды:

- `dotnet restore` — восстанавливает зависимости для проекта;
- `dotnet build` — компилирует проект;
- `dotnet test` — выполняет модульное тестирование проекта;

```
markjprice — -bash — 104×39
[Marks-MBP-13:~ markjprice$ dotnet new -l
Getting ready...
Usage: new [options]

Options:
  -h, --help          Displays help for this command.
  -l, --list          Lists templates containing the specified name. If no name is specified, lists all
templates.
  -n, --name          The name for the output being created. If no name is specified, the name of the cu
rrent directory is used.
  -o, --output        Location to place the generated output.
  -i, --install       Installs a source or a template pack.
  -u, --uninstall     Uninstalls a source or a template pack.
  --type              Filters templates based on available types. Predefined values are "project", "item
" or "other".
  --force             Forces content to be generated even if it would change existing files.
  -lang, --language   Specifies the language of the template to create.


Templates                                         Short Name       Language          Tags
----------------------------------------------------------------------------------------------------------
Console Application                               console          [C#], F#, VB      Common/Console
Class library                                     classlib         [C#], F#, VB      Common/Library
Unit Test Project                                 mstest           [C#], F#, VB      Test/MSTest
xUnit Test Project                                xunit            [C#], F#, VB      Test/xUnit
ASP.NET Core Empty                                web              [C#], F#          Web/Empty
ASP.NET Core Web App (Model-View-Controller)      mvc              [C#], F#          Web/MVC
ASP.NET Core Web App                              razor            [C#]              Web/MVC/Razor Pages
ASP.NET Core with Angular                         angular          [C#]              Web/MVC/SPA
ASP.NET Core with React.js                        react            [C#]              Web/MVC/SPA
ASP.NET Core with React.js and Redux              reactredux       [C#]              Web/MVC/SPA
ASP.NET Core Web API                              webapi           [C#], F#          Web/WebAPI
global.json file                                  globaljson                         Config
Nuget Config                                      nugetconfig                        Config
Web Config                                        webconfig                          Config
Solution File                                     sln                                Solution
Razor Page                                        page                               Web/ASP.NET
MVC ViewImports                                   viewimports                        Web/ASP.NET
MVC ViewStart                                     viewstart                          Web/ASP.NET
```

Рис. 7.20

- `dotnet run` — запускает проект;
- `dotnet migrate` — переносит проект .NET Core, созданный с помощью CLI-инструментов предварительной версии, в формат MS Build CLI-инструмента текущей версии;
- `dotnet pack` — создает NuGet-пакет из проекта;
- `dotnet publish` — компилирует и публикует проект как платформозависимое или автономное приложение;
- `add` — добавляет ссылку на пакет в проект;
- `remove` — удаляет ссылку на пакет из проекта;
- `list` — перечисляет ссылки на пакеты в проекте.

Добавление ссылки на пакет

Предположим, вы хотите добавить пакет, созданный сторонним разработчиком, например, Newtonsoft.Json — популярный пакет для работы с форматом сериализации *JavaScript Object Notation (JSON)*.

Visual Studio Code

В Visual Studio Code откройте каталог `Chapter07/Assemblies`, созданный вами ранее, а затем введите следующую команду на панели Integrated Terminal (Интегрированный терминал):

```
dotnet add package newtonsoft.json
```

Visual Studio Code выведет данные о добавлении ссылки, как показано в листинге ниже:

```
info : Adding PackageReference for package 'newtonsoft.json' into project '/Users/
markjprice/Code/Chapter07/Assemblies/Assemblies.csproj'.
log : Restoring packages for /Users/markjprice/Code/Chapter07/Assemblies/
Assemblies.csproj...
info : GET
https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json
info : OK
https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json 485ms
info : GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/10.0.3/
newtonsoft.js on.10.0.3.nupkg
info : OK
https://api.nuget.org/v3-flatcontainer/newtonsoft.json/10.0.3/newtonsoft.js
on.10.0.3.nupkg 602ms
log : Installing Newtonsoft.Json 10.0.3.
info : Package 'newtonsoft.json' is compatible with all the specified frameworks in
project '/Users/markjprice/Code/Chapter07/Assemblies/Assemblies.csproj'.
info : PackageReference for package 'newtonsoft.json' version '10.0.3'
added to file '/Users/markjprice/Code/Chapter07/Assemblies/Assemblies.csproj'.
```

Открыв файл `Assemblies.csproj`, вы увидите добавленную ссылку на пакет, как показано в листинге ниже:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="newtonsoft.json" Version="10.0.3" />
  </ItemGroup>
</Project>
```

Visual Studio 2017

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте, выберите пункт Manage NuGet Packages (Управление пакетами NuGet). На открывшейся панели перейдите на вкладку Browse (Обзор), выполните поиск пакета Newtonsoft.Json, выберите его и нажмите кнопку Install (Установить) (рис. 7.21).

Нажмите кнопку Install (Установить) и примите лицензионное соглашение.

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте Assemblies и выберите пункт Edit Assemblies.csproj (Редактировать

Assemblies.csproj). Обратите внимание на изменения в файле, показанные выше для Visual Studio Code.

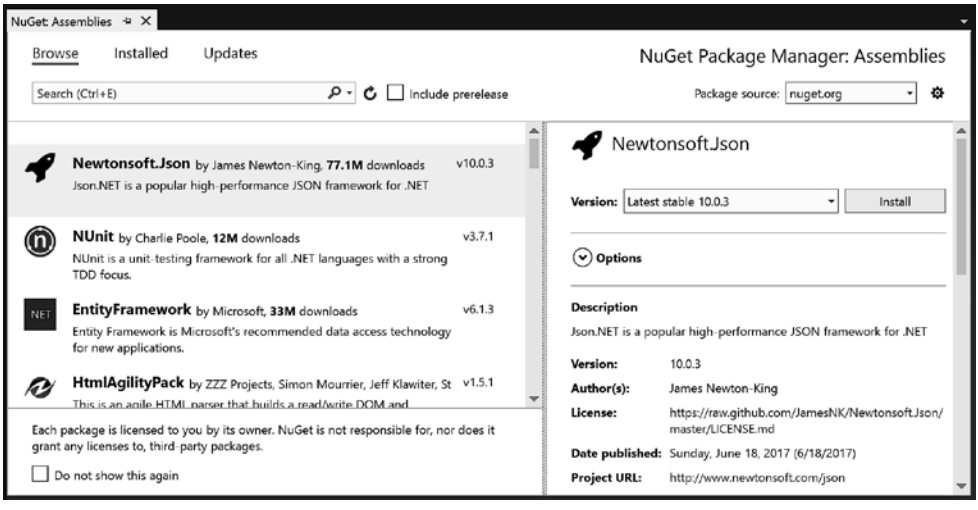


Рис. 7.21

В Visual Studio 2017 вы можете выполнить команду Tools ▸ NuGet Package Manager ▸ Package Manager Console (Средства ▸ Диспетчер пакетов NuGet ▸ Консоль диспетчера пакетов), чтобы использовать командную строку для добавления, изменения и удаления ссылок на пакеты так же, как делали это на панели Integrated Terminal (Интегрированный терминал) Visual Studio Code.

Упаковка библиотеки для распространения с помощью NuGet

Теперь упакуем созданный ранее проект SharedLibrary. В нем переименуйте файл `Class1.cs` в `StringExtensions.cs` и измените его содержимое так, как показано в листинге ниже:

```
using System.Text.RegularExpressions;

namespace Packt.CS7
{
  public static class StringExtensions
  {
    public static bool IsValidXmlTag(this string input)
    {
      return Regex.IsMatch(input, @"^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$");
    }

    public static bool IsValidPassword(this string input)
    {
```

```
      // Не менее восьми допустимых символов
      return Regex.IsMatch(input, "^[a-zA-Z0-9_-]{8,}$");
    }

    public static bool IsValidHex(this string input)
    {
      // Три или шесть допустимых символов шестнадцатеричного числа
      return Regex.IsMatch(input, "^#?([a-fA-F0-9]{3}|[a-fA-F0-9]{6})$");
    }
  }
}
```

В этих методах расширения используются регулярные выражения для проверки строковых значений. Вы научитесь писать такие выражения в главе 8.

Откройте файл `SharedLibrary.csproj` и измените его содержимое в соответствии с листингом, приведеным ниже, при этом обратите внимание на следующее:

- значение `PackageId` должно быть абсолютно уникальным, если хотите опубликовать свой NuGet-пакет на сайте https://www.nuget.org/, чтобы другие пользователи могли его просматривать и скачивать;
- прочие элементы не требуют пояснений:

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
  <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
  <PackageId>Packt.CS7.SharedLibrary</PackageId>
  <PackageVersion>1.0.0.0</PackageVersion>
  <Authors>Mark J Price</Authors>
  <PackageLicenseUrl>
    http://opensource.org/licenses/MS-PL
  </PackageLicenseUrl>
  <PackageProjectUrl>
    http://github.com/markjprice/cs7dotnetcore2
  </PackageProjectUrl>
  <PackageIconUrl>
    http://github.com/markjprice/cs7dotnetcore2/nuget.png
  </PackageIconUrl>
  <PackageRequireLicenseAcceptance>true</PackageRequireLicenseAcceptance>
  <PackageReleaseNotes>
    Example shared library packaged for NuGet.
  </PackageReleaseNotes>
  <Description>
    Three extension methods to validate a string value.
  </Description>
  <Copyright>
    Copyright c2017 Packt Publishing Limited
  </Copyright>
```

```
  <PackageTags>string extension packt cs7</PackageTags>
  </PropertyGroup>

</Project>
```

В Visual Studio 2017 щелкните правой кнопкой мыши на проекте SharedLibrary и выберите пункт Pack (Упаковать).

В Visual Studio Code введите следующую команду на панели Integrated Terminal (Интегрированный терминал), чтобы сгенерировать NuGet-пакет:

```
dotnet pack -c Release
```

Обе среды: и Visual Studio 2017, и Visual Studio Code — выведут сообщение об успешном завершении операции (рис. 7.22).

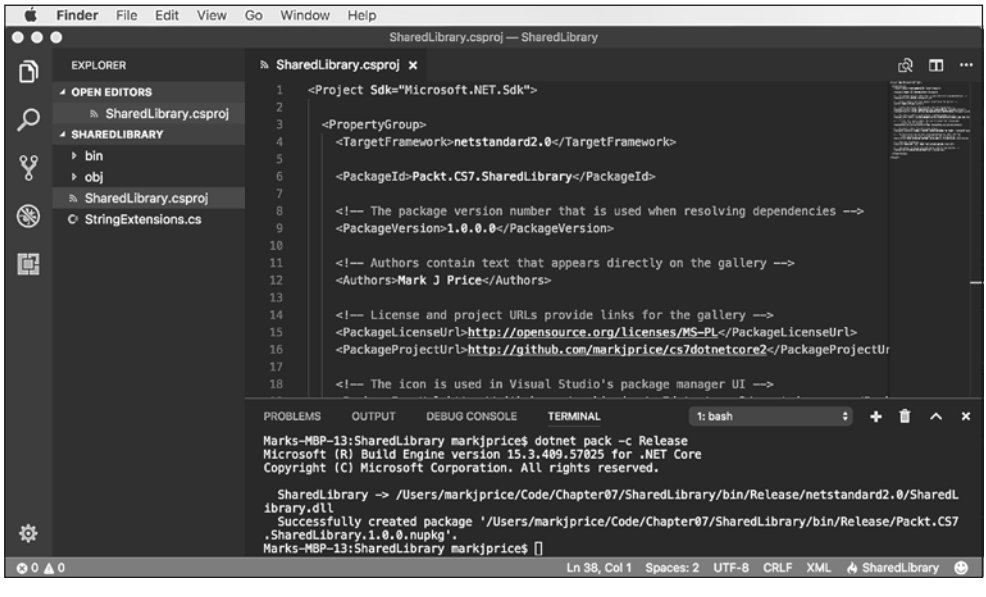


Рис. 7.22

Запустите свой любимый браузер и перейдите по следующей ссылке: https://www.nuget.org/packages/manage/upload.

Вам нужно будет зарегистрироваться на сайте https://www.nuget.org/, чтобы другие разработчики могли ссылаться на ваш NuGet-пакет.

Нажмите кнопку Browse (Обзор) и выберите файл с расширением `.nupkg`, созданный с помощью команды `pack` (рис. 7.23).

Убедитесь, что информация, добавленная в файл `SharedLibrary.csproj`, корректна, а затем нажмите кнопку Submit (Отправить).

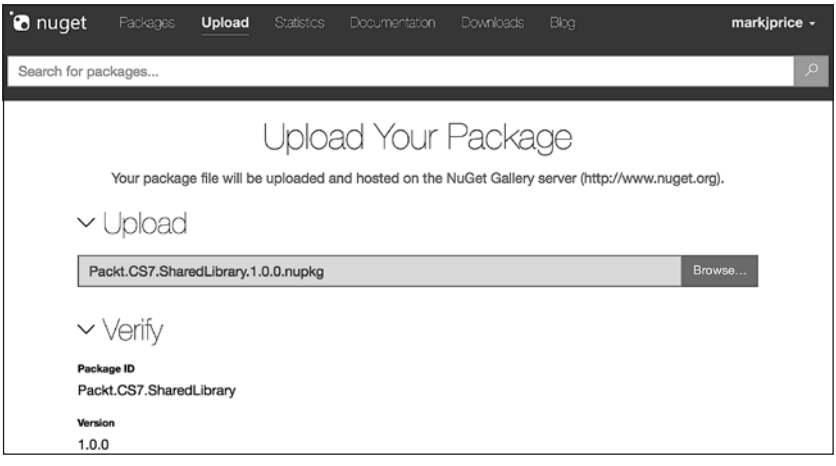


Рис. 7.23

Через несколько секунд вы увидите сообщение об успешном завершении процесса загрузки и увидите свой пакет (рис. 7.24).

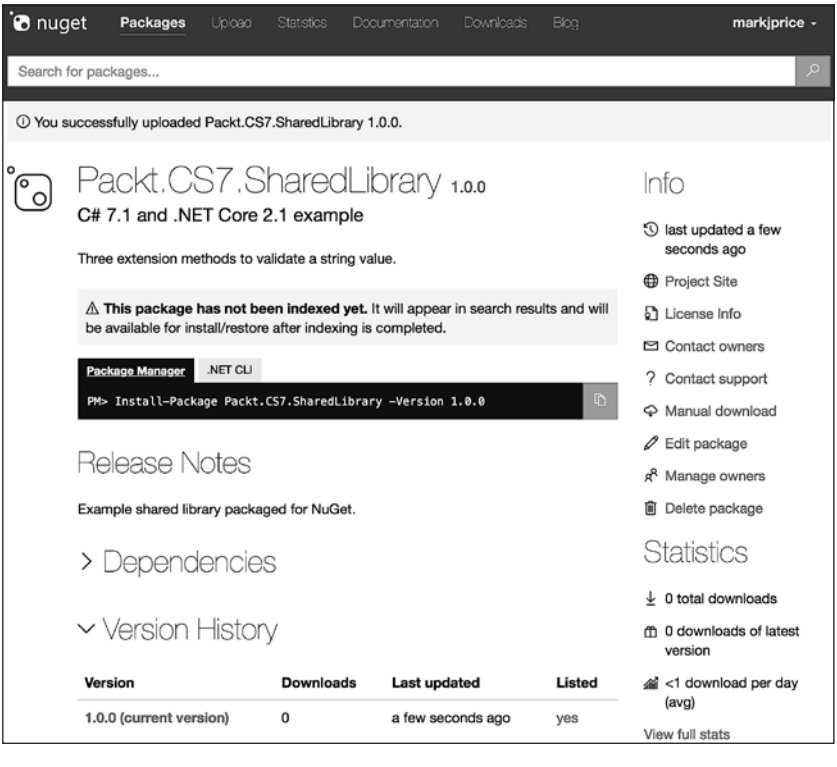


Рис. 7.24

Тестирование пакетов

Теперь вы опробуете свой загруженный пакет, указав ссылку на него в проекте Assemblies.

Visual Studio Code

В Visual Studio Code откройте проект Assemblies и измените код файла проекта, добавив ссылку на свой пакет, как показано в листинге ниже:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="newtonsoft.json" Version="10.0.3" />
    <PackageReference Include="packt.cs7.sharedlibrary" Version="1.0.0" />
  </ItemGroup>
</Project>
```

Visual Studio 2017

В Visual Studio 2017 откройте панель NuGet Package Manager (Диспетчер пакетов NuGet) и выполните поиск и установку своего пакета (рис. 7.25).

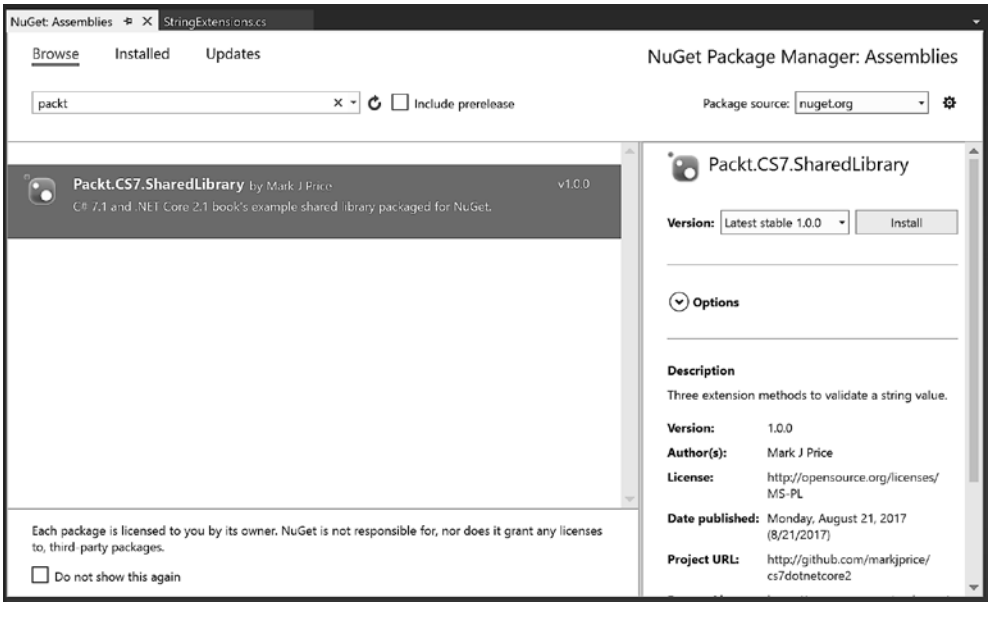


Рис. 7.25

Visual Studio 2017 и Visual Studio Code

Отредактируйте файл `Program.cs`, указав строку для импорта пространства имен `Packt.CS7` и в методе `Main` предложив пользователю ввести некие строковые значения. Добавьте код для проверки введенных значений с помощью методов расширения в пакете, как показано в листинге ниже:

```
using static System.Console;
using Packt.CS7;

namespace Assemblies
{
  class Program
  {
    static void Main(string[] args)
    {
      Write("Enter a valid color value in hex: ");
      string hex = ReadLine();
      WriteLine($"Is {hex} a valid color value:
      {hex.IsValidHex()}");

      Write("Enter a valid XML tag: ");
      string xmlTag = ReadLine();
      WriteLine($"Is {xmlTag} a valid XML tag:
      {xmlTag.IsValidXmlTag()}");

      Write("Enter a valid password: ");
      string password = ReadLine();
      WriteLine($"Is {password} a valid password:
      {password.IsValidPassword()}");
    }
  }
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Enter a valid color value in hex: 00ffc8
Is 00ffc8 a valid color value: True
Enter a valid XML tag: <h1 class="<" />
Is <h1 class="<" /> a valid XML tag: False
Enter a valid password: secretsauce
Is secretsauce a valid password: True
```

Портирование кода в .NET Core

Если у вас уже есть опыт программирования на .NET и вы разработали несколько приложений под старые платформы, такие как .NET Framework, то можете портировать (перенести) их на .NET Core. Однако внимательно проанализируйте ситуацию и разберитесь, так ли необходимо заниматься портированием. Иногда от него стоит отказаться.

Можно ли портировать?

Платформа .NET Core предоставляет великолепную поддержку следующих типов приложений:

- веб-приложения *ASP.NET Core MVC*;
- сервисы *ASP.NET Core Web API* (REST/HTTP);
- приложения для *универсальной платформы Windows (UWP)*;
- *консольные* приложения.

Платформа .NET Core не поддерживает следующие типы приложений:

- веб-приложения *ASP.NET Web Forms*;
- настольные приложения *Windows Forms*;
- настольные приложения *Windows Presentation Foundation (WPF)*;
- приложения *Silverlight*.

К счастью, WPF- и Silverlight-приложения основаны на диалекте языка XAML, аналогичном используемому на платформах UWP и Xamarin.Forms.

Нужно ли портировать?

Даже если портирование *возможно*, так ли оно *необходимо*? Какие преимущества вы получите? К некоторым общим преимуществам относятся:

- *развертывание в Linux или Docker* — эти операционные системы легковесны и малозатратны в случае использования их для развертывания веб-приложений и веб-сервисов, особенно если сравнивать с Windows Server;
- *удаление зависимостей от IIS и System.Web.dll* —даже если вы решите разворачивать приложения на Windows Server, платформа ASP.NET Core допускает размещение на легковесных, высокопроизводительных веб-серверах Kestrel (и других);
- *инструменты командной строки*, позволяющие разработчикам и администраторам автоматизировать задачи при написании кода консольных приложений. Доступна очень полезная возможность кросс-платформенного запуска одного и того же инструмента.

Анализатор портируемости .NET

Сотрудники корпорации Microsoft создали полезный инструмент, который служит для оценки возможности портирования существующих приложений и подготовки отчета. Демонстрация процесса работы с инструментом .NET Portability Analyzer доступна по адресу https://channel9.msdn.com/Blogs/Seth-Juarez/A-Brief-Look-at-the-NET-Portability-Analyzer.

Сравнение .NET Framework и .NET Core

Заметны три ключевых отличия, как показано в табл. 7.5.

Таблица 7.5

.NET Core	.NET Framework
Распространяется в виде NuGet-пакетов, поэтому каждое приложение может быть развернуто с помощью отдельной локальной версии платформы .NET Core, которая необходима для запуска	Распространяется в виде общесистемного набора сборок (если буквально, то глобального кэша сборок)
Разделена на небольшие разноуровневые компоненты, вследствие чего нужно выполнить минимальное развертывание	Требует полноценного развертывания
Удалены устаревшие компоненты, такие как Windows Forms и Web Forms, и функции, привязанные к определенной платформе, например AppDomains, .NET Remoting и бинарная сериализация	Сохранены некоторые старые технологии, такие как Windows Forms, WPF и ASP.NET Web Forms

Использование библиотек, не относящихся к .NET Standard

Семьдесят процентов доступных NuGet-пакетов можно использовать на платформе .NET Core 2.0, даже если они не были скомпилированы для .NET Standard 2.0.

Для поиска подходящих NuGet-пакетов перейдите по ссылке https://www.nuget.org/packages.

Так, существует набор пользовательских коллекций для обработки матриц, созданных компанией Dialect Software LLC. Документация доступна на сайте https://www.nuget.org/packages/DialectSoftware.Collections.Matrix/.

Этот пакет был в последний раз обновлен в 2013 году (рис. 7.26).

Рис. 7.26

Таким образом, пакет был выпущен задолго до релиза .NET Core и предназначался для .NET Framework. До тех пор пока пакет сборки наподобие этого использует API, доступные в .NET Standard 2.0, его можно применять в проектах .NET Core 2.0.

Откройте файл `Assemblies.csproj` и добавьте элемент `<PackageReference>` со ссылкой на пакет компании Dialect Software, как показано в листинге ниже:

```
<PackageReference Include="dialectsoftware.collections.matrix" Version="1.0.0" />
```

Откройте файл `Program.cs`, добавьте инструкции для импорта пространств имен `DialectSoftware.Collections` и `DialectSoftware.Collections.Generics` и для создания экземпляров `Axis` и `Matrix<T>`, укажите их значения и выведите их, как показано в листинге ниже:

```
var x = new Axis("x", 0, 10, 1);
var y = new Axis("y", 0, 4, 1);

var matrix = new Matrix<long>(new[] { x, y });
int i = 0;
for (; i < matrix.Axes[0].Points.Length; i++)
{
  matrix.Axes[0].Points[i].Label = "x" + i.ToString();
}
i = 0;

for (; i < matrix.Axes[1].Points.Length; i++)
{
  matrix.Axes[1].Points[i].Label = "y" + i.ToString();
}

foreach (long[] c in matrix)
{
  matrix[c] = c[0] + c[1];
}

foreach (long[] c in matrix)
{
  WriteLine("{0},{1} ({2},{3}) = {4}", matrix.Axes[0].Points[c[0]].Label, matrix.
Axes[1].Points[c[1]].Label, c[0], c[1], matrix[c]);
}
```

Запустите консольное приложение, проанализируйте результат вывода и обратите внимание на предупреждающее сообщение:

```
/Users/markjprice/Code/Chapter07/Assemblies/Assemblies.csproj : warning
NU1701: Package 'DialectSoftware.Collections.Matrix 1.0.0' was restored
using '.NETFramework,Version=v4.6.1' instead of the project target framework
'.NETCoreApp,Version=v2.0'. This package may not be fully compatible with your
project.
x0,y0 (0,0) = 0
x0,y1 (0,1) = 1
x0,y2 (0,2) = 2
x0,y3 (0,3) = 3
...и так далее.
```

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам, приведенным в данной главе.

Проверочные вопросы

1. Чем пространства имен отличаются от сборок?
2. Как сослаться на другой проект в файле `.csproj`?
3. Чем пакет отличается от метапакета?
4. Какой тип .NET представлен в C# псевдонимом `float`?
5. Какова разница между пакетами `NETStandard.Library` и `Microsoft.NETCore.App`?
6. Чем отличаются платформозависимые и автономные приложения .NET Core?
7. Что такое RID?
8. В чем разница между командами `dotnet pack` и `dotnet publish`?
9. Какие типы приложений, написанных для .NET Framework, можно портировать в .NET Core?
10. Можно ли в проектах .NET Core использовать пакеты, написанные для .NET Framework?

Дополнительные ресурсы

- Перенос кода в .NET Core из .NET Framework: https://docs.microsoft.com/en-us/dotnet/articles/core/porting/.
- Пакеты, метапакеты и платформы: https://docs.microsoft.com/en-us/dotnet/articles/core/packages.
- Блог, посвященный .NET: https://blogs.msdn.microsoft.com/dotnet/.
- Документация CoreFX README.md: https://github.com/dotnet/corefx/blob/master/Documentation/README.md.
- Развертывание приложений .NET Core: https://docs.microsoft.com/en-us/dotnet/articles/core/deploying/.
- Справочная система по API .NET Standard: https://github.com/dotnet/standard.

Резюме

В этой главе мы обсудили взаимосвязи между сборками и пространствами имен, возможности переноса существующих кодовых баз .NET Framework, способы публикации ваших приложений и библиотек и развертывания вашего кода независимо от платформы.

Из следующей главы вы узнаете о некоторых распространенных типах .NET Standard 2.0, входящих в состав .NET Core 2.0.

8 Использование распространенных типов .NET Standard

Эта глава посвящена распространенным типам .NET Standard 2.0, которые входят в состав платформы .NET Core 2.0. К ним относятся типы для управления числами, текстом, коллекциями, сетевым доступом, отражением, атрибутами, изображениями, а также настройками глобализации.

В данной главе:

- операции над числами;
- работа с текстом;
- операции над коллекциями;
- работа с сетевыми ресурсами;
- работа с типами и атрибутами;
- глобализация кода.

Работа с числами

Числа — один из наиболее употребительных типов данных. Самые популярные в .NET Standard 2.0 типы данных для работы с числами приведены в табл. 8.1.

Таблица 8.1

Пространство имен	Примеры типов	Описание
System	SByte, Int16, Int32, Int64	Положительные и отрицательные целые числа
System	Byte, UInt16, UInt32, UInt64	Натуральные числа, то есть положительные целые числа

Продолжение ⇩

Таблица 8.1 *(продолжение)*

Пространство имен	Примеры типов	Описание
System	Single, Double	Вещественные числа, то есть числа с плавающей точкой
System	Decimal	Точные вещественные числа, используются для научных, инженерных или финансовых нужд
System.Numerics	BigInteger, Complex, Quaternion	Условно большие целые числа, комплексные и гиперкомплексные числа

Более подробную информацию можно получить, перейдя по ссылке https://docs.microsoft.com/en-us/dotnet/standard/numerics.

В решении `Chapter08` создайте новое консольное приложение и присвойте ему имя `WorkingWithNumbers`.

Крупные целые числа

Наибольшее целое число, которое может быть сохранено в типах .NET Standard, имеющих псевдонимы в языке C#, равняется примерно 18,5 квинтиллиона. Это число сохраняется как длинное число без знака.

Добавьте в файл `Program.cs` инструкцию импорта пространства имен `System.Numerics`, как показано в следующем коде:

```
using System.Numerics;
```

Добавьте в метод `Main` инструкции для вывода наибольшего значения типа `ulong` и 30-значного числа с помощью типа `BigInteger`, как показано в листинге ниже:

```
var largestLong = ulong.MaxValue;
WriteLine($"{largestLong,40:N0}");

var atomsInTheUniverse =
BigInteger.Parse("123456789012345678901234567890");
WriteLine($"{atomsInTheUniverse,40:N0}");
```

Код форматирования ,40 предполагает выравнивание 40 символов по правому краю, таким образом, оба числа выровнены по правой границе окна.

Запустите приложение и проанализируйте результат вывода:

```
 18,446,744,073,709,551,615
123,456,789,012,345,678,901,234,567,890
```

Работа с комплексными числами

Комплексное число может быть представлено как $a + bi$, где a и b — вещественные числа, а i — мнимая единица, причем $i2 = -1$. Если вещественная часть равна нулю, то перед нами чистое мнимое число, если мнимая — то вещественное. Комплексные числа используются во многих областях науки, технологии, инженерии и математики.

Комплексные числа складываются путем раздельного суммирования вещественных и мнимых частей слагаемых. Рассмотрим пример:

$$(a + bi) + (c + di) = (a + c) + (b + d)i.$$

В метод `Main` пропишите инструкции для добавления двух комплексных чисел, как показано в следующем листинге:

```
var c1 = new Complex(4, 2);
var c2 = new Complex(3, 7);
var c3 = c1 + c2;
WriteLine($"{c1} added to {c2} is {c3}");
```

Запустите приложение и проанализируйте результат вывода:

```
(4, 2) added to (3, 7) is (7, 9)
```

Кватернионы — система чисел, расширяющая комплексные числа. Они формируют четырехмерную ассоциативную нормированную алгебру с делением вещественных чисел и, соответственно, область значений.

А? Да, знаю... Я этого тоже не понимаю. Не переживайте, мы не станем писать программы с использованием таких чисел! Достаточно будет сказать, что эти числа хороши для описания пространственных вращений, как следствие, применяются в игровых движках, компьютерных симуляторах и системах управления летательными аппаратами.

Работа с текстом

Другой распространенный тип данных для переменных — это текст. Самые применяемые типы .NET для работы с текстом перечислены в табл. 8.2.

Таблица 8.2

Пространство имен	Тип	Описание
System	Char	Хранение одного текстового символа
System	String	Хранение нескольких текстовых символов
System.Text	StringBuilder	Эффективное управление строками
System.Text.RegularExpressions	Regex	Эффективное управление строками, соответствующими шаблонам

Извлечение длины строки

Создайте новый проект консольного приложения с именем `WorkingWithText`.

В Visual Studio 2017 щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства). В категории Startup Project (Запускаемый проект) установите переключатель в положение Current selection (Текущий проект).

В некоторых случаях может понадобиться определить длину фрагмента текста, хранящегося в классе `string`.

Добавьте в метод `Main` инструкции для определения переменной, хранящей название города Лондон, и вывода его названия и длины, как показано ниже:

```
string city = "London";
WriteLine($"{city} is {city.Length} characters long.");
```

Извлечение символов строки

Класс `string` для хранения текста использует внутренний массив `char`. Он тоже индексируется, так что можно применить синтаксис массива для чтения хранящихся в нем символов.

Добавьте указанную ниже инструкцию, а затем запустите консольное приложение:

```
WriteLine($"First char is {city[0]} and third is {city[2]}.");
```

Разделение строк

Иногда требуется разделить текст во всех местах, где используется определенный символ, например запятая.

Добавьте несколько строк кода, определяющих одну строку с названиями городов, разделенными запятыми. Вы можете задействовать метод `Split` и указать символ, который будет применяться в качестве разделителя. Затем создается массив строк; его можно перечислить с помощью инструкции `foreach`.

```
string cities = "Paris,Berlin,Madrid,New York";
string[] citiesArray = cities.Split(',');
foreach (string item in citiesArray)
{
  WriteLine(item);
}
```

Извлечение фрагмента строки

В некоторых случаях может потребоваться извлечь часть текста. Например, если в коде используется полное имя человека, хранящееся в строке с пробелом между именем и фамилией, то вы могли бы обнаружить позицию пробела и извлечь имя и фамилию в виде двух частей, скажем, так:

```
string fullname = "Alan Jones";
int indexOfTheSpace = fullname.IndexOf(' ');
```

```
string firstname = fullname.Substring(0, indexOfTheSpace);
string lastname = fullname.Substring(indexOfTheSpace + 1);
WriteLine($"{lastname}, {firstname}");
```

Если бы формат исходного полного имени был иным, к примеру Lastname, Firstname, то код немного бы отличался. В качестве дополнительного упражнения попробуйте написать несколько инструкций, которые изменят ввод Jones, Alan на Alan Jones.

Проверка содержимого строк

Иногда необходимо проверить наличие определенных символов в начале, составе или конце строки. Это делается следующим образом:

```
string company = "Microsoft";
bool startsWithM = company.StartsWith("M");
bool containsN = company.Contains("N");
WriteLine($"Starts with M: {startsWithM}, contains an N:{containsN}");
```

Другие члены класса string

В табл. 8.3 перечислены некоторые другие члены типа `string`.

Таблица 8.3

Член	Описание
Trim, TrimStart и TrimEnd	Удаляют пробельные символы в начале и/или конце строки
ToUpper и ToLower	Преобразуют символы строки в прописные или строчные
Insert и Remove	Добавляют или удаляют указанный текст в переменной типа string
Replace	Замещает указанный текст
String.Concat	Конкатенирует две переменные типа string. Оператор + вызывает этот метод, если используется между переменными типа string
String.Join	Конкатенирует одну или несколько переменных типа string с указанным символом между ними
string.IsNullOrEmpty	Проверяет, хранит ли переменная типа string значение null или она пустая ("")
string.IsNullOrWhitespace	Проверяет, является ли переменная типа string значением null, пустой строкой или строкой, состоящей только из пробельных символов (например, табуляции, пробела, возврата каретки, перевода строки и т. д.)
String.Empty	Можно задействовать вместо выделения памяти каждый раз, когда вы применяете литеральное значение string, используя пару двойных кавычек без содержимого ("")
string.Format	Устаревший альтернативный метод вывода форматированных строк, применяющий позиционированные параметры вместо именованных

Обратите внимание: некоторые из предыдущих методов являются *статическими*. Это значит, что метод может быть вызван только из типа, но не из экземпляра переменной.

Приведу пример: если мне понадобится взять массив строк и объединить их обратно в одну строку с разделителем, то я могу использовать метод `Join` следующим образом:

```
string recombined = string.Join(" => ", citiesArray);
WriteLine(recombined);
```

При желании использовать позиционированные параметры вместо синтаксиса интерполяции строк можно задействовать метод `Format`, примерно так:

```
string fruit = "Apples";
decimal price = 0.39M;
DateTime when = DateTime.Today;

WriteLine($"{fruit} cost {price:C} on {when:dddd}s.");

WriteLine(string.Format("{0} cost {1:C} on {2:dddd}s.",
fruit, price, when));
```

Позиционированные параметры начинаются с нуля. Иногда их легче форматировать в коде по сравнению с синтаксисом интерполяции строк, как вы можете видеть в предыдущем примере кода.

Если вы запустите консольное приложение и посмотрите на результат вывода, то он будет следующим:

```
London is 6 characters long.
First char is L and third is n.
Paris
Berlin
Madrid
New York
Jones, Alan
Starts with M: True, contains an N: False
Paris => Berlin => Madrid => New York
Apples cost £0.39 on Mondays.
Apples cost £0.39 on Mondays.
```

Эффективное оперирование строками

Вы можете конкатенировать (сцепить) две строки в новую с помощью метода `String.Concat` или просто применив оператор `+`. Но делать так не рекомендуется, поскольку .NET нужно будет создать совершенно новую строку в памяти. При объединении только двух переменных типа `string` последствий может и не быть. Но если конкатенацию проводить в цикле, то операция способна оказать значительное негативное влияние на производительность и использование памяти.

В главе 13 вы узнаете, как эффективно конкатенировать переменные типа string с помощью типа StringBuilder.

Сопоставление шаблонов с регулярными выражениями

Регулярные выражения полезны для проверки на допустимость ввода пользователя. Они очень мощные и могут становиться крайне сложными. Почти все языки программирования поддерживают регулярные выражения и применяют универсальный набор специальных символов для их определения.

Создайте новый проект консольного приложения с именем `WorkingWithRegularExpressions`.

В начало файла импортируйте следующие пространства имен и статические типы:

```
using System.Text.RegularExpressions;
using static System.Console;
```

Добавьте в метод `Main` такие инструкции:

```
Write("Enter your age: ");
string input = ReadLine();
var ageChecker = new Regex(@"\d");
if (ageChecker.IsMatch(input))
{
  WriteLine("Thank you!");
}
else
{
  WriteLine($"This is not a valid age: {input}");
}
```

Символ @ перед строкой отключает возможность применения управляющих символов в string. Эти символы предваряются префиксом в виде обратного слеша (\). К примеру, управляющий символ \t обозначает горизонтальный отступ (табуляцию), а \n — новую строку. При использовании регулярных выражений нужно отключить эту функцию.

Запустите консольное приложение и проанализируйте результат вывода.

Указав целочисленное значение возраста, вы увидите надпись `Thank you!`:

```
Enter your age: 34
Thank you!
```

Если вместо числа укажете слово `carrots`, то увидите сообщение об ошибке:

```
Enter your age: carrots
This is not a valid age: carrots
```

Но использовав значение типа `bob30smith`, вы вновь увидите `Thank you!`:

```
Enter your age: bob30smith
Thank you!
```

Регулярное выражение, которое мы использовали, — `\d`, обозначает одну цифру. Однако оно не ограничивает ввод значения *до* и *после* цифры. Это регулярное выражение на русском языке может быть объяснено так: «Введите хотя бы одну цифру».

Измените регулярное выражение на `^\d$`, например, так:

```
var ageChecker = new Regex(@"^\d$");
```

Перезапустите приложение. Теперь не допускаются любые значения, кроме одной цифры.

Мы же хотим, чтобы можно было указать *одну цифру или более*. В этом случае нужно добавить символ `+` (плюс) после выражения `\d`. Измените регулярное выражение так, как показано ниже:

```
var ageChecker = new Regex(@"^\d+$");
```

Запустите приложение и посмотрите, как регулярное выражение теперь допускает ввод только положительных целых чисел любой длины.

Синтаксис регулярных выражений

В табл. 8.4 приведено несколько универсальных комбинаций символов, которые можно использовать в регулярных выражениях.

Таблица 8.4

Символ	Значение	Символ	Значение
^	Начало ввода	$	Конец ввода
\d	Одна цифра	\D	Любой нецифровой символ
\w	Пробельный символ	\W	Любой символ, кроме пробельного
[A-Za-z0-9]	Диапазон символов	\^	Символ ^ (каретки)
[aeiou]	Набор символов	[^aeiou]	Любой символ, кроме входящего в набор
.	Один символ	\.	Символ . (точка)

В табл. 8.5 приведены некоторые символы, влияющие на предыдущий символ в регулярном выражении.

Таблица 8.5

Символ	Значение	Символ	Значение
+	Один или больше	?	Один или ни одного
{3}	Точно три	{3,5}	От трех до пяти
{3,}	Три или больше	{,3}	До трех

Примеры регулярных выражений

В табл. 8.6 я привел примеры некоторых регулярных выражений.

Таблица 8.6

Выражение	Значение
\d	Одна цифра где-либо в вводе
a	Символ где-либо в вводе
Bob	Слово Bob где-либо в вводе
^Bob	Слово Bob в начале ввода
Bob$	Слово Bob в конце ввода
^\d{2}$	Точно две цифры
^[0-9]{2}$	Точно две цифры
^[A-Z]{4,}$	Не менее четырех прописных букв
^[A-Za-z]{4,}$	Не менее четырех прописных или строчных букв
^[A-Z]{2}\d{3}$	Точно две прописные буквы и три цифры
^d.g$	Буква d, далее любой символ, а затем буква g, так что это выражение совпадет со словами типа dig, dog и др. с любым символом между буквами d и g
^d\.g$	Буква d, далее точка (.), а затем буква g, поэтому данное выражение совпадает только с последовательностью d.g

Применяйте регулярные выражения для проверки пользовательского ввода. Такие же регулярные выражения можно задействовать и в других языках, например в JavaScript.

Работа с коллекциями

Еще один распространенный тип данных — коллекции. Если в переменной нужно сохранить несколько значений, то можно использовать коллекцию.

Коллекция — это структура данных в памяти, позволяющая управлять несколькими элементами различными способами, хотя все коллекции имеют общие функции.

Наиболее распространенные типы в .NET Standard 2.0 для работы с коллекциями приведены в табл. 8.7.

Таблица 8.7

Пространство имен	Примеры типов	Описание
System.Collections	IEnumerable, IEnumerable<T>	Интерфейсы и базовые классы, используемые коллекциями

Продолжение ⇩

Таблица 8.7 *(продолжение)*

Пространство имен	Примеры типов	Описание
System.Collections.Generic	List<T>, Dictionary<T>, Queue<T>, Stack<T>	Стали применяться в версии C# 2 с .NET 2.0 и являются более предпочтительными, так как позволяют указать тип, который будет использован при сохранении (а это безопаснее, быстрее и эффективнее)
System.Collections.Concurrent	BlockingCollection, ConcurrentDictionary, ConcurrentQueue	Эти коллекции безопасны для применения в многопоточных приложениях
System.Collections.Immutable	ImmutableArray, ImmutableDictionary, ImmutableList, ImmutableQueue	Предназначены для сценариев, в которых содержимое коллекции никогда не должно изменяться

Дополнительная информация доступна на сайте https://docs.microsoft.com/en-us/dotnet/standard/collections.

Общие характеристики всех коллекций

Все коллекции реализуют интерфейс `ICollection`; это значит, что они должны иметь свойство `Count`, сообщающее, сколько в коллекции элементов.

К примеру, если бы у нас была коллекция `passengers`, то мы могли бы сделать следующее:

```
int howMany = passengers.Count;
```

Все коллекции реализуют интерфейс `IEnumerable`; это значит, что они должны содержать метод `GetEnumerator`, который возвращает объект, реализующий `IEnumerator`. Таким образом, они должны включать метод `MoveNext` и свойство `Value`, чтобы коллекции можно было перебирать с помощью инструкции `foreach`.

Выполнить определенное действие со всеми элементами в коллекции `passengers` поможет следующий код:

```
foreach (var passenger in passengers)
{
  // операции с каждым пассажиром
}
```

Понять принципы коллекций может помочь схема с наиболее распространенными интерфейсами, реализующими коллекции (рис. 8.1).

Списки, то есть тип, реализующий `IList`, представляют собой *упорядоченные коллекции*. Это значит следующее: они реализуют `ICollection` и поэтому должны содержать свойство `Count` и методы `Add` и `Insert` (первый позволяет помещать эле-

мент в конец коллекции, второй — в список, в указанную позицию), а также метод `RemoveAt` (позволяет удалить элемент в указанной позиции).

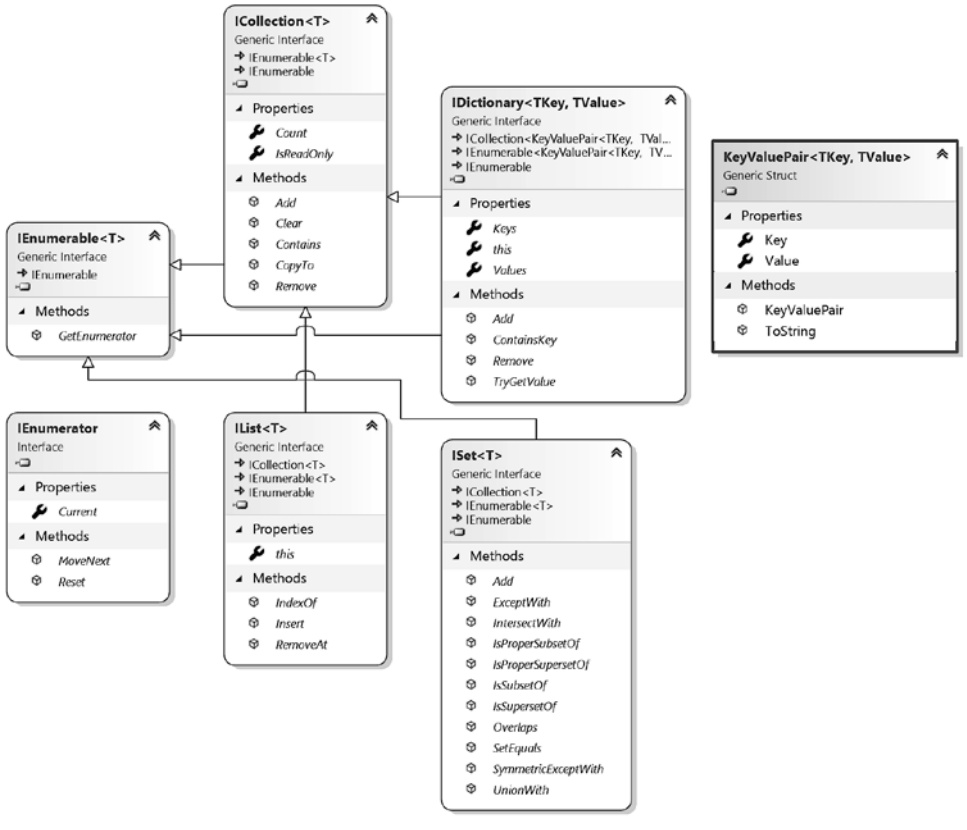


Рис. 8.1

Понятие коллекции

Существует несколько различных категорий коллекции: списки, словари, стеки, очереди, множества и многие другие узкоспециализированные коллекции.

Списки

Списки отлично подойдут, если нужно вручную управлять порядком элементов в коллекции. Каждому элементу в списке автоматически присваивается уникальный индекс (позиция). Элементы могут быть любого типа (хотя в одном списке все они должны быть одного типа) и дублироваться. Нумеруются элементы в списке, начиная с 0 целыми числами, поэтому первый элемент в списке, как и в массиве, имеет индекс 0, как показано в табл. 8.8.

Таблица 8.8

Индекс	Элемент
0	London
1	Paris
2	London
3	Sydney

Если новый элемент (к примеру, `Santiago`) добавить между элементами `London` и `Sydney`, то индекс элемента `Sydney` автоматически увеличится. Исходя из этого, следует учитывать, что индекс объекта может измениться после добавления или удаления элементов, как показано в табл. 8.9.

Таблица 8.9

Индекс	Элемент
0	London
1	Paris
2	London
3	Santiago
4	Sydney

Словари

Словари будут удобны, если каждое значение (или элемент) имеет уникальное подзначение (или специально введенное значение), которое в дальнейшем можно использовать в качестве ключа для быстрого поиска значения в коллекции. Ключ должен быть уникальным. Так, при сохранении списка людей в качестве ключа могут послужить номера паспортов.

Представьте, что ключ — это своего рода запись в алфавитном указателе в словаре в реальном мире. Он облегчает поиск определения слова, поскольку слова (то есть ключи) сортируются, и если мы ищем определение слова `Манго`, то открыли бы словарь в середине, ведь буква `M` находится в середине алфавита. Словари в программировании так же умны, когда дело доходит до поиска элемента по ключу.

И ключ, и значение могут быть любого типа. В табл. 8.10 используются строковые ключи и значения.

Таблица 8.10

Ключ	Значение
BSA	Bob Smith
MW	Max Williams
BSB	Bob Smith
AM	Amir Mohammed

Стеки

Стеки удобно использовать в тех случаях, когда нужно реализовать поведение *«последним пришел — первым вышел»* (last-in, first-out, LIFO). Стек позволяет напрямую получить доступ только к одному элементу в начале стека, хотя, конечно, можно перечислить элементы один за другим, чтобы прочитать их все. Нельзя получить прямой доступ, скажем, ко второму элементу в стеке.

К примеру, текстовые редакторы задействуют стек для хранения последовательности действий, которые пользователь выполнил недавно, а затем, когда он нажимает сочетание клавиш Ctrl+Z, программа отменяет последнее действие в стеке, затем предпоследнее и т. д.

Очереди

Очереди удобны, если нужно реализовать поведение *«первым пришел — первым вышел»* (first-in, first-out, FIFO). Очередь позволяет напрямую получить доступ только к одному элементу в начале очереди, хотя, конечно, можно перечислить элементы один за другим, чтобы прочитать их все. Нельзя получить прямой доступ, скажем, ко второму элементу в очереди.

Например, фоновые процессы используют очередь для обработки заданий в том порядке, в котором те поступают, — точно так же, как получают услугу люди, стоящие в очереди в почтовом отделении.

Множества

Множества — прекрасный выбор при необходимости выполнять операции над множествами элементов двух коллекций. К примеру, есть две коллекции с названиями городов и нужно выяснить, какие названия присутствуют в обеих (так называемое *пересечение* множеств).

Работа со списками

Создайте новый проект консольного приложения с именем `WorkingWithLists`.

В начало файла импортируйте следующие пространства имен и статические типы:

```
using System;
using System.Collections.Generic;
using static System.Console;
```

В методе `Main` напечатайте код, показанный ниже. Он демонстрирует некоторые из распространенных способов работы со списками:

```
var cities = new List<string>();
cities.Add("London");
cities.Add("Paris");
cities.Add("Milan");
WriteLine("Initial list");
foreach (string city in cities)
{
  WriteLine($" {city}");
```

```
}
WriteLine($"The first city is {cities[0]}.");
WriteLine($"The last city is {cities[cities.Count - 1]}.");
cities.Insert(0, "Sydney");
WriteLine("After inserting Sydney at index 0");
foreach (string city in cities)
{
  WriteLine($" {city}");
}
cities.RemoveAt(1);
cities.Remove("Milan");
WriteLine("After removing two cities");
foreach (string city in cities)
{
  WriteLine($" {city}");
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Initial list
  London
  Paris
  Milan
The first city is London.
The last city is Milan.
After inserting Sydney at index 0
  Sydney
  London
  Paris
  Milan
After removing two cities
  Sydney
  Paris
```

Работа со словарями

Создайте новый проект консольного приложения с именем `WorkingWithDictionaries`.

Импортируйте те же пространства имен, которые импортировали раньше.

В методе `Main` напечатайте код, показанный ниже. Он демонстрирует некоторые из распространенных способов работы со словарями:

```
var keywords = new Dictionary<string, string>();
keywords.Add("int", "32-bit integer data type");
keywords.Add("long", "64-bit integer data type");
keywords.Add("float", "Single precision floating point number");
WriteLine("Keywords and their definitions");
foreach (KeyValuePair<string, string> item in keywords)
{
  WriteLine($" {item.Key}: {item.Value}");
}
WriteLine($"The definition of long is {keywords["long"]}");
```

Запустите приложение и проанализируйте результат вывода:

```
Keywords and their definitions
  int: 32-bit integer data type
```

```
  long: 64-bit integer data type
  float: Single precision floating point number
The definition of long is 64-bit integer data type
```

Сортировка коллекций

Класс `List<T>` можно отсортировать, вызвав его метод `Sort` (но помните, что индексы всех элементов будут изменены!).

Сортировка списка строк или других встроенных типов выполняется автоматически, но если вы создаете коллекцию собственного типа, то данный тип должен реализовать интерфейс IComparable. Как это сделать, вы узнали в главе 6.

Классы `Dictionary<T>`, `Stack<T>` и `Queue<T>` не могут быть отсортированы, поскольку обычно это не требуется. Например, вы никогда не будете сортировать очередь из посетителей гостиницы. Но в некоторых случаях может понадобиться отсортировать словарь или множество.

Различия между этими отсортированными коллекциями часто неуловимы, но могут влиять на загруженность памяти и производительность вашего приложения, так что рекомендую выбирать коллекции, наиболее подходящие под ваши требования.

Несколько распространенных отсортированных коллекций показаны в табл. 8.11.

Таблица 8.11

Коллекция	Описание
SortedDictionary<TKey, TValue>	Представляет собой коллекцию пар «ключ — значение», которые сортируются по ключу
SortedList<TKey, TValue>	Представляет собой коллекцию пар «ключ — значение», сортируемых по ключу, на основе связанной реализации IComparer<T>
SortedSet<T>	Представляет собой коллекцию объектов, хранящихся в отсортированном порядке

Использование специализированных коллекций

Существует несколько других коллекций для особых случаев (табл. 8.12).

Таблица 8.12

Коллекция	Описание
System.Collections.BitArray	Управляет компактным массивом двоичных значений, представленных логическими значениями, где true соответствует включенному биту (1), а false — отключенному (0)
System.Collections.Generics.LinkedList<T>	Представляет собой двусвязный список, в котором каждый элемент имеет ссылку на свой предыдущий и следующий элементы

Использование неизменяемых коллекций

Иногда необходимо сделать коллекцию неизменяемой, то есть ни один из ее членов не может быть изменен, как, впрочем, удален или добавлен.

Если вы импортируете пространство имен `System.Collections.Immutable`, то любая коллекция, реализующая интерфейс `IEnumerable<T>`, получит шесть методов расширения для преобразования этой коллекции в неизменяемый список, словарь, набор хеш-функций и т. д.

Откройте проект WorkingWithLists, импортируйте пространство имен `System.Collections.Immutable` и добавьте следующие инструкции в конец метода `Main`, как показано в листинге ниже:

```
var immutableCities = cities.ToImmutableList();

var newList = immutableCities.Add("Rio");

Write("Immutable cities:");
foreach (string city in immutableCities)
{
  Write($" {city}");
}
WriteLine();

Write("New cities:");
foreach (string city in newList)
{
  Write($" {city}");
}
WriteLine();
```

Запустите консольное приложение, проанализируйте результат вывода и обратите внимание: неизменяемый список городов остается без изменений, если с ним используется метод `Add`. Вместо этого программа возвращает новый список с только что добавленным городом.

Работа с сетевыми ресурсами

Иногда нужно взаимодействовать с сетевыми ресурсами. Наиболее подходящие для этого типы .NET Standard приведены в табл. 8.13.

Таблица 8.13

Пространство имен	Тип (пример)	Описание
System.Net	Dns, Uri, Cookie, WebClient, IPAddress	Для работы с DNS-серверами, идентификаторами URI, IP-адресами и т. д.
System.Net	FtpStatusCode, FtpWebRequest, FtpWebResponse	Для работы с FTP-серверами

Пространство имен	Тип (пример)	Описание
System.Net	HttpStatusCode, HttpWebRequest, HttpWebResponse	Для работы с HTTP-серверами, то есть с сайтами
System.Net .Mail	Attachment, MailAddress, MailMessage, SmtpClient	Для работы с SMTP-серверами, то есть для отправки сообщений электронной почты
System.Net .NetworkInformation	IPStatus, NetworkChange, Ping, TcpStatistics	Для работы с низкоуровневыми сетевыми протоколами

Работа с идентификаторами URI, DNS и IP-адресами

Создайте новый проект консольного приложения с именем `WorkingWithNetworkResources`.

В верхнюю часть файла импортируйте следующие пространства имен:

```
using System;
using System.Net;
using static System.Console;
```

Добавьте в метод `Main` инструкции, приглашающие пользователя ввести адрес сайта, а затем задействуйте тип `Uri`, чтобы разбить введенный адрес на несколько частей, в том числе схему (HTTP, FTP и т. д.), номер порта и хост, как показано в следующем листинге:

```
Write("Enter a valid web address: ");
string url = ReadLine();
if (string.IsNullOrWhiteSpace(url))
{
  url = "http://world.episerver.com/cms/?q=pagetype";
}

var uri = new Uri(url);
WriteLine($"Scheme: {uri.Scheme}");
WriteLine($"Port: {uri.Port}");
WriteLine($"Host: {uri.Host}");
WriteLine($"Path: {uri.AbsolutePath}");
WriteLine($"Query: {uri.Query}");
```

Запустите консольное приложение, введите действительный адрес сайта, нажмите клавишу Enter и проанализируйте результат вывода:

```
Enter a valid web address:
Scheme: http
Port: 80
Host: world.episerver.com
Path: /cms/
Query: ?q=pagetype
```

Вставьте в метод Main следующие инструкции для получения IP-адреса введенного сайта, как показано в листинге ниже:

```
IPHostEntry entry = Dns.GetHostEntry(uri.Host);
WriteLine($"{entry.HostName} has the following IP addresses:");
foreach (IPAddress address in entry.AddressList)
{
  WriteLine($" {address}");
}
```

Запустите консольное приложение, введите действительный адрес сайта, нажмите клавишу Enter и проанализируйте результат вывода:

```
world.episerver.com has the following IP addresses: 217.114.90.249
```

Опрос сервера

Добавьте в файл Program.cs инструкцию импорта пространства имен System.Net.NetworkInformation, как показано в следующем листинге:

```
using System.Net.NetworkInformation;
```

Вставьте следующие инструкции в метод Main для получения IP-адреса введенного сайта:

```
var ping = new Ping();
PingReply reply = ping.Send(uri.Host);
WriteLine($"{uri.Host} was pinged, and replied: {reply.Status}.");
if (reply.Status == IPStatus.Success)
{
  WriteLine($"Reply from {reply.Address} took
  {reply.RoundtripTime:N0}ms");
}
```

Запустите консольное приложение, введите действительный адрес сайта, нажмите клавишу Enter, проанализируйте результат вывода и обратите внимание, что сайт episerver.com не отвечает на запросы ping (зачастую это делается для предотвращения DDoS-атак):

```
world.episerver.com was pinged, and replied: TimedOut.
```

Запустите консольное приложение еще раз, введите адрес http://google.com, как показано в следующем выводе:

```
Enter a valid web address: http://google.com
Scheme: http
Port: 80
Host: google.com
Path: /
Query:
google.com has the following IP addresses:
  216.58.206.78
  2a00:1450:4009:804::200e
google.com was pinged, and replied: Success.
Reply from 216.58.206.78 took 9ms
```

Работа с типами и атрибутами

Reflection — технология программирования, позволяющая коду понимать самого себя и управлять им. Сборка состоит из четырех частей (максимум):

- *метаданные и манифест сборки* — имя, сборка, версия файла, сборки, на которые производится ссылка, и т. д.;
- *метаданные о типах* — информация о типах, их членах и т. д.;
- *IL-код* — реализация методов, свойств, конструкторов и т. д.;
- *встроенные ресурсы (не обязательно)* — изображения, строки, JavaScript и т. д.

Метаданные состоят из информации о коде. Применяются к коду с помощью атрибутов. Последние могут быть использованы на нескольких уровнях — к сборкам, к типам и их членам, как показано в следующем листинге:

```
// атрибут уровня сборки
[assembly: AssemblyTitle("Working with Reflection")]

[Serializable] // атрибут уровня типа

public class Person
// атрибут уровня члена
[Obsolete("Deprecated: use Run instead.")]
public void Walk()
{
  // ...
}
```

Указание версий сборок

В .NET номера версий — это комбинация из трех чисел с двумя необязательными добавочными номерами.

По правилам семантического указания номеров версий:

- *основной* — фундаментальные изменения;
- *дополнительный* —незначительные изменения, в том числе новые функции и исправление ошибок;
- *патч* — незначительные исправления ошибок.

Как вариант, номер версии может также включать:

- *предрелиз* — неподдерживаемые релизы для предварительного ознакомления;
- *номер сборки* — ночные сборки.

Следуйте правилам семантического указания номеров версий, которые описаны на сайте http://semver.org.

Чтение метаданных о сборке

Создайте новый проект консольного приложения с именем `WorkingWithReflection`.

В верхнюю часть файла импортируйте следующие типы и пространства имен:

```
using static System.Console;
using System;
using System.Reflection;
```

Добавьте следующие инструкции в метод `Main` для получения сборки консольных приложений, вывода ее имени и местоположения, а также всех атрибутов уровня сборки и вывода их типов, как показано в листинге ниже:

```
WriteLine("Assembly metadata:");

Assembly assembly = Assembly.GetEntryAssembly();

WriteLine($" Full name: {assembly.FullName}");
WriteLine($" Location: {assembly.Location}");

var attributes = assembly.GetCustomAttributes();

WriteLine($" Attributes:");
foreach (Attribute a in attributes)
{
  WriteLine($" {a.GetType()}");
}
```

Запустите консольное приложение и проанализируйте результат вывода.

```
Assembly metadata:
  Full name: WorkingWithReflection, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null
  Location:
/Users/markjprice/Code/Chapter08/WorkingWithReflection/bin/Debug/netcoreapp
2.0/WorkingWithReflection.dll
  Attributes:
    System.Runtime.CompilerServices.CompilationRelaxationsAttribute
    System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
    System.Diagnostics.DebuggableAttribute
    System.Runtime.Versioning.TargetFrameworkAttribute
    System.Reflection.AssemblyCompanyAttribute
    System.Reflection.AssemblyConfigurationAttribute
    System.Reflection.AssemblyDescriptionAttribute
    System.Reflection.AssemblyFileVersionAttribute
    System.Reflection.AssemblyInformationalVersionAttribute
    System.Reflection.AssemblyProductAttribute
  мSystem.Reflection.AssemblyTitleAttribute
```

Теперь, когда мы знаем, какие атрибуты присвоены сборке, можем запрашивать только нужные.

Добавьте следующие инструкции в метод `Main` для получения классов `AssemblyInformationalVersionAttribute` и `AssemblyCompanyAttribute`, как показано в листинге:

```
var version =
assembly.GetCustomAttribute<AssemblyInformationalVersionAttribute>();
```

```
WriteLine($" Version: {version.InformationalVersion}");

var company = assembly.GetCustomAttribute<AssemblyCompanyAttribute>();
WriteLine($" Company: {company.Company}");
```

Запустите консольное приложение и проанализируйте результат вывода.

```
Version: 1.0.0
Company: WorkingWithReflection
```

Хм-м, установим эту информацию явно. В .NET Framework эти значения устанавливаются путем добавления атрибутов в файл исходного кода на языке C#, как показано в следующем листинге:

```
[assembly: AssemblyCompany("Packt Publishing")]
[assembly: AssemblyInformationalVersion("1.0.0")]
```

Компилятор Roslyn устанавливает указанные атрибуты автоматически, так что мы не можем пользоваться старым способом. Теперь эти атрибуты могут быть установлены в файле проекта.

Измените файл `WorkingWithReflection.csproj`, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <Version>1.3.0</Version>
    <Company>Packt Publishing</Company>
  </PropertyGroup>

</Project>
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Version: 1.3.0
Company: Packt Publishing
```

Создание собственных атрибутов

Вы можете определить собственные атрибуты, унаследовав их от класса `Attribute`.

Добавьте класс `CoderAttribute`, как показано в следующем листинге:

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
                AllowMultiple = true)]
public class CoderAttribute : Attribute
{
  public string Coder { get; set; }
  public DateTime LastModified { get; set; }
  public CoderAttribute(string coder, string lastModified)
  {
    Coder = coder;
    LastModified = DateTime.Parse(lastModified);
  }
}
```

Добавьте в класс `Program` метод `DoStuff` и создайте для него атрибут `Coder`, как показано в листинге ниже:

```
[Coder("Mark Price", "22 August 2017")]
[Coder("Johnni Rasmussen", "13 September 2017")]
public static void DoStuff()
{

}
```

В файл `Program.cs` импортируйте пространство имен `System.Linq`, как показано в следующем листинге:

```
using System.Linq;
```

Добавьте в метод `Main` программный код для получения типов, перечисления их членов, считывания любых атрибутов `Coder` этих членов и вывода данной информации, как показано ниже:

```
WriteLine($"Types:");
Type[] types = assembly.GetTypes();

foreach (Type type in types)
{
  WriteLine($" Name: {type.FullName}");

  MemberInfo[] members = type.GetMembers();

  foreach (MemberInfo member in members)
  {
    WriteLine($" {member.MemberType}: {member.Name}
     ({member.DeclaringType.Name})");

    var coders = member.GetCustomAttributes<CoderAttribute>()
    .OrderByDescending(c => c.LastModified);
    foreach (CoderAttribute coder in coders)
    {
      WriteLine($" Modified by {coder.Coder} on
      {coder.LastModified.ToShortDateString()}");
    }
  }
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Types:
  Name: CoderAttribute
    Method: get_Coder (CoderAttribute)
    Method: set_Coder (CoderAttribute)
    Method: get_LastModified (CoderAttribute)
    Method: set_LastModified (CoderAttribute)
    Method: Equals (Attribute)
    Method: GetHashCode (Attribute)
    Method: get_TypeId (Attribute)
    Method: Match (Attribute)
    Method: IsDefaultAttribute (Attribute)
```

```
    Method: ToString (Object)
    Method: GetType (Object)
    Constructor: .ctor (CoderAttribute)
    Property: Coder (CoderAttribute)
    Property: LastModified (CoderAttribute)
    Property: TypeId (Attribute)
  Name: WorkingWithReflection.Program
    Method: DoStuff (Program)
      Modified by Johnni Rasmussen on 13/09/2017
      Modified by Mark Price on 22/08/2017
    Method: ToString (Object)
    Method: Equals (Object)
    Method: GetHashCode (Object)
    Method: GetType (Object)
    Constructor: .ctor (Program)
  Name: WorkingWithReflection.Program+<>c
    Method: ToString (Object)
    Method: Equals (Object)
    Method: GetHashCode (Object)
    Method: GetType (Object)
    Constructor: .ctor (<>c)
    Field: <>9 (<>c)
    Field: <>9__0_0 (<>c)
```

Другие возможности Reflection

Это лишь небольшой пример того, чего можно добиться с помощью технологии Reflection. Мы использовали ее только для считывания метаданных из кода. Данная технология также позволяет динамически:

- загружать сборки, на которые в данный момент не установлены ссылки;
- выполнять код;
- генерировать новый код и сборки.

Глобализация кода

Глобализация — это процесс, позволяющий вашему приложению выполняться правильно во всем мире. Он состоит из двух частей: *интернационализации* и *локализации*.

Интернационализация заключается в написании кода для поддержки разных языковых и региональных параметров. При разработке приложений важно учитывать язык и регион, поскольку форматы даты и валюты отличаются, например, в Квебеке и Париже, несмотря на то, что в обоих городах говорят по-французски.

Существуют коды *Международной организации по стандартизации* (International Standards Organization, ISO) для всех языковых и региональных параметров. Так, в коде `da-DK` символы `da` указывают на датский язык, а `DK` определяют страну — Данию; в коде `fr-CA` символы `fr` указывают на французский язык, а `CA` определяют страну — Канаду.

ISO — это не аббревиатура, а отсылка к греческому слову isos (что означает «равный»).

Локализация — настройка пользовательского интерфейса для реализации поддержки языка, например изменения метки кнопки Close (en) или Fermer (fr). Поскольку локализация — это всего лишь язык, то ей не нужно учитывать регион, хотя по иронии судьбы стандартизации (en-US) и (en-GB) указывают на обратное.

Глобализация — это огромная тема, по которой написаны целые книги. В этом разделе вы изучите самые основы, используя тип `CultureInfo` в пространстве имен `System.Globalization`.

Интернационализация приложений. Создайте новый проект консольного приложения с именем `Internationalization`.

В начало файла импортируйте следующие типы и пространства имен:

```
using static System.Console;
using System;
using System.Globalization;
```

Добавьте в метод `Main` такие инструкции:

```
CultureInfo globalization = CultureInfo.CurrentCulture;
CultureInfo localization = CultureInfo.CurrentUICulture;
WriteLine($"The current globalization culture is
{globalization.Name}: {globalization.DisplayName}");
WriteLine($"The current localization culture is
{localization.Name}: {localization.DisplayName}");
WriteLine();
WriteLine("en-US: English (United States)");
WriteLine("da-DK: Danish (Denmark)");
WriteLine("fr-CA: French (Canada)");
Write("Enter an ISO culture code: ");
string newculture = ReadLine();
if (!string.IsNullOrEmpty(newculture))
{
  var ci = new CultureInfo(newculture);
  CultureInfo.CurrentCulture = ci;
  CultureInfo.CurrentUICulture = ci;
}
Write("Enter your name: ");
string name = ReadLine();
Write("Enter your date of birth: ");
string dob = ReadLine();
Write("Enter your salary: ");
string salary = ReadLine();
DateTime date = DateTime.Parse(dob);
int minutes = (int)DateTime.Today.Subtract(date).TotalMinutes;
decimal earns = decimal.Parse(salary);
WriteLine($"{name} was born on a {date:dddd} and is {minutes:N0}
minutes old and earns {earns:C}.");
```

При запуске приложения поток автоматически устанавливается на применение языковых и региональных настроек в соответствии с операционной системой. Я за-

пускаю свой код в Лондоне, поэтому поток уже установлен на английский язык (Великобритания).

Код уведомит пользователя, что нужно указать другой ISO-код. Благодаря этому приложения смогут менять языковые и региональные настройки по умолчанию во время выполнения.

Затем приложение задействует стандартные коды формата для вывода дня недели, `dddd`; количества минут с разделителями тысяч, `N0`; и денежные единицы с символом валюты, `C`. Настройка происходит автоматически на основе языковых и региональных настроек потока.

Запустите консольное приложение и проанализируйте результат вывода. Введите значение `en-GB` в качестве ISO-кода, а затем любые данные. Вам нужно будет ввести дату в формате, допустимом для британского варианта английского языка.

```
Enter an ISO culture code: en-GB
Enter your name: Alice
Enter your date of birth: 30/3/1967
Enter your salary: 23500
Alice was born on a Thursday, is 25,469,280 minutes old and earns £23,500.00.
```

Перезапустите приложение и используйте другие языковые и региональные настройки, например датский язык в Дании (`da-DK`):

```
Enter an ISO culture code: da-DK
Enter your name: Mikkel
Enter your date of birth: 12/3/1980
Enter your salary: 34000
Mikkel was born on a onsdag, is 18.656.640 minutes old and earns kr. 34.000,00.
```

Подумайте, нужна ли глобализация вашему приложению, и если да, то запланируйте ее, прежде чем начинать кодирование! Запишите весь текст, применяемый в пользовательском интерфейсе, который придется локализовать. Подумайте обо всех данных, которые необходимо будет локализовать (форматы даты и чисел, сортировка текста).

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

Пользуясь ресурсами в Интернете, ответьте на следующие вопросы.

1. Какое максимальное количество символов может быть сохранено в переменной типа `string`?
2. В каких случаях и почему нужно использовать класс `SecureString`?

3. Когда целесообразно применять тип `StringBuilder`?
4. В каких ситуациях нужно задействовать `LinkedList`?
5. В каких случаях следует использовать класс `SortedDictionary` вместо класса `SortedList`?
6. Каков ISO-код языковых и региональных параметров ISO для валлийского языка?
7. В чем разница между локализацией, глобализацией и интернационализацией?
8. Что означает символ `$` в регулярных выражениях?
9. Как в регулярных выражениях представить цифры?
10. Почему *нельзя* применять официальный стандарт для адресов электронной почты при создании регулярного выражения, цель которого — проверка адреса электронной почты пользователя?

Упражнение 8.1. Регулярные выражения

Создайте консольное приложение с именем `Exercise02`, которое предлагает пользователю ввести сначала регулярное выражение, а затем еще некий ввод и сравнивает его на соответствие выражению. Процесс повторяется, пока пользователь не нажмет клавишу Esc.

```
The default regular expression checks for at least one digit.
Enter a regular expression (or press ENTER to use the default): ^[a- z]+$
Enter some input: apples
apples matches ^[a-z]+$? True
Press ESC to end or any key to try again.
Enter a regular expression (or press ENTER to use the default): ^[a- z]+$
Enter some input: abc123xyz
abc123xyz matches ^[a-z]+$? False
Press ESC to end or any key to try again.
```

Упражнение 8.2. Методы расширения

Создайте методы расширения, выполняющие следующую задачу.

Расширьте тип `BigInteger` методом `ToWords`, который станет возвращать введенное цифрами число прописью: например, 18 000 000 будет возвращено как *«восемнадцать миллионов»*, а 18 456 000 000 000 000 000 — как *«восемнадцать с половиной квинтиллионов»*. Воспользуйтесь ссылкой https://en.wikipedia.org/wiki/Names_of_large_numbers.

Дополнительные ресурсы

- Справочник .NET Core API: https://docs.microsoft.com/en-us/dotnet/core/api/index.
- Класс `String`: https://docs.microsoft.com/en-us/dotnet/core/api/system.string.
- Класс `Regex`: https://docs.microsoft.com/en-us/dotnet/core/api/system.text.regularexpressions.regex.

- Регулярные выражения в .NET: https://docs.microsoft.com/en-us/dotnet/articles/standard/base-types/regular-expressions.
- Язык регулярных выражений — быстрое руководство: https://docs.microsoft.com/en-us/dotnet/articles/standard/base-types/quick-ref.
- RegExr: изучение, конструирование и проверка регулярных выражений: http://regexr.com/.
- Коллекции (C# и Visual Basic): https://docs.microsoft.com/en-us/dotnet/core/api/system.collections.
- Атрибуты: https://docs.microsoft.com/en-us/dotnet/standard/attributes/.
- Глобализация: https://docs.microsoft.com/en-us/dotnet/standard/globalization-localization/.

Резюме

В этой главе вы узнали об эффективных способах использования типов для хранения и управления текстом и механизмах применения коллекций для хранения групп элементов, а также научились глобализировать код.

В следующей главе вы научитесь управлять файлами и потоками, кодировать/декодировать текст и выполнять сериализацию.

9 Работа с файлами, потоками и сериализация

Эта глава посвящена чтению и записи в файлы и потоки, кодированию текста и сериализации.

В данной главе:

- управление файловой системой;
- чтение и запись с помощью потоков;
- кодирование текста;
- сериализация графов объектов.

Управление файловой системой

Ваши приложения часто должны совершать операции ввода и вывода с файлами и каталогами. Пространства имен `System` и `System.IO` содержат классы, позволяющие выполнять эти задачи.

Работа в кросс-платформенных средах и файловых системах

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) выберите пункт .NET Core. В центре диалогового окна выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)), присвойте ему имя `WorkingWithFileSystems`, укажите расположение по адресу `C:\Code`, введите имя решения `Chapter09`, а затем нажмите кнопку OK.

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) создайте новый каталог `Chapter09` и подкаталог `WorkingWithFileSystems`. Откройте созданный каталог и выполните команду `dotnet new console`.

Добавьте в начало файла `Program.cs` следующие инструкции импорта. Обратите внимание: мы статически импортируем типы `Directory`, `Path` и `Environment`, чтобы упростить код:

```
using static System.Console;
using System.IO;
using static System.IO.Directory;
using static System.IO.Path;
using static System.Environment;
```

Файловые пути различаются в Windows, macOS и Linux, поэтому начнем с рассмотрения того, как с этим справляется платформа .NET Core.

Создайте статический метод `OutputFileSystemInfo` и напишите инструкции, которые выводили бы:

- символы разделения пути и каталога;
- путь к текущему каталогу;
- ряд специальных путей к системным файлам, временным файлам и документам.

```
static void OutputFileSystemInfo()
{
  WriteLine($"Path.PathSeparator: {PathSeparator}");
  WriteLine($"Path.DirectorySeparatorChar: {DirectorySeparatorChar}");
  WriteLine($"Directory.GetCurrentDirectory(): {GetCurrentDirectory()}");
  WriteLine($"Environment.CurrentDirectory: {CurrentDirectory}");
  WriteLine($"Environment.SystemDirectory: {SystemDirectory}");
  WriteLine($"Path.GetTempPath(): {GetTempPath()}");
  WriteLine($"GetFolderPath(SpecialFolder):");
  WriteLine($" System: {GetFolderPath(SpecialFolder.System)}");
  WriteLine($" ApplicationData:
  {GetFolderPath(SpecialFolder.ApplicationData)}");
  WriteLine($" MyDocuments: {GetFolderPath(SpecialFolder.MyDocuments)}");
  WriteLine($" Personal: {GetFolderPath(SpecialFolder.Personal)}");
}
```

В типе Environment реализовано много других полезных членов, в том числе метод GetEnvironmentVariables, а также свойства OSVersion и ProcessorCount.

В методе `Main` вызовите метод `OutputFileSystemInfo`, как показано в следующем листинге:

```
static void Main(string[] args)
{
  OutputFileSystemInfo();
}
```

Windows 10

Запустите консольное приложение и проанализируйте результат вывода (рис. 9.1).

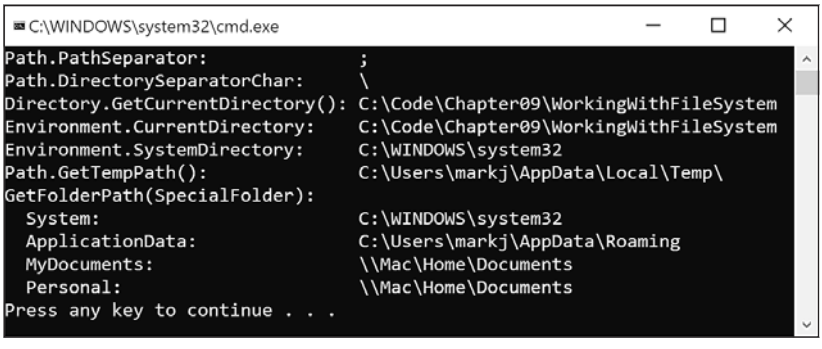


Рис. 9.1

В Windows в качестве разделителя используется обратный слеш.

macOS

Запустите консольное приложение и проанализируйте результат вывода (рис. 9.2).

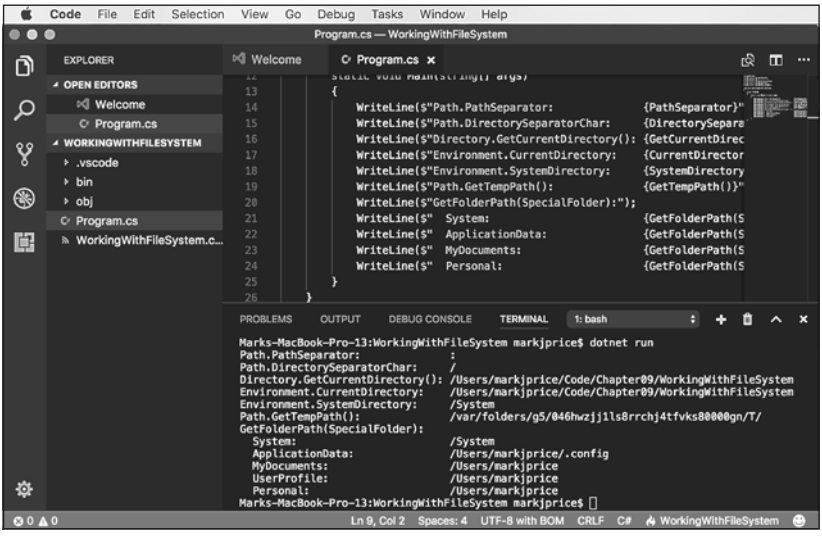


Рис. 9.2

В macOS в качестве разделителя используется (прямой) слеш.

Работа с дисками

Для работы с дисками воспользуйтесь статическим методом `DriveInfo`, возвращающим информацию обо всех дисках, подключенных к вашему компьютеру. У каждого диска свой тип. Методы `DriveInfo` и `DriveType` показаны на рис. 9.3.

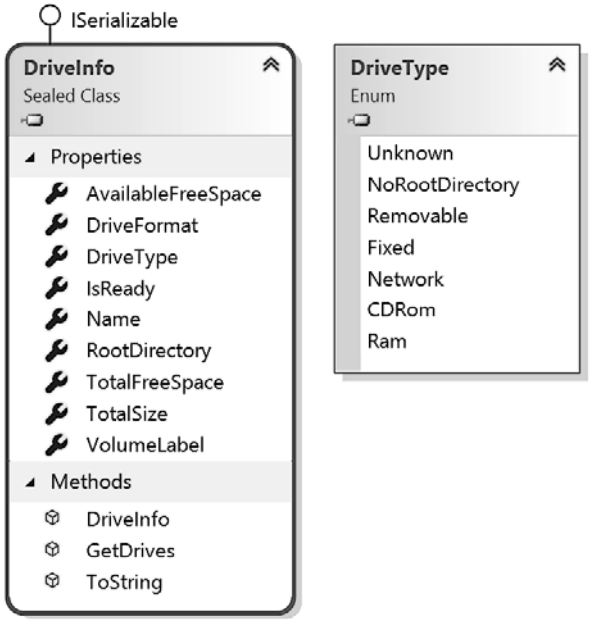


Рис. 9.3

Создайте метод `WorkWithDrives` и напишите инструкции для получения всех дисков и вывода их имен, типов, размеров, количества доступного свободного места, а также формата, но только если диск готов к использованию, как показано в следующем листинге:

```
static void WorkWithDrives()
{
  WriteLine($"|-------------------------------|------------|
  ---------|-------------------|-------------------|");
  WriteLine($"| Name | Type | Format | Size | Free space |");
  WriteLine($"|-------------------------------|------------|
  ---------|-------------------|-------------------|");

  foreach (DriveInfo drive in DriveInfo.GetDrives())
  {
    if (drive.IsReady)
    {
      WriteLine($"| {drive.Name,-30} |
      {drive.DriveType,-10} | {drive.DriveFormat, -7} |
      {drive.TotalSize,18:N0} |
```

```
      {drive.AvailableFreeSpace,18:N0} |");
    }
    else
    {
      WriteLine($"| {drive.Name,-30} | {drive.DriveType,-10} |");
    }
  }
  WriteLine($"|--------------------------------|------------|
  ---------|--------------------|--------------------|");
}
```

Проверяйте готовность диска перед считыванием таких свойств, как TotalSize, или в случае со съемными дисками ваша программа выведет сообщение о возникшем исключении.

В методе `Main` закомментируйте предыдущий вызов метода и добавьте новый вызов метода `WorkWithDrives`, как показано в следующем листинге:

```
static void Main(string[] args)
{
  // OutputFileSystemInfo();

  WorkWithDrives();
}
```

Запустите консольное приложение и проанализируйте результат вывода (рис. 9.4).

Name	Type	Format	Size	Free space
/	Fixed	hfs	498,954,403,840	135,917,678,592
/dev	Ram	devfs	191,488	0
/net	Network	autofs	0	0
/home	Network	autofs	0	0
/Volumes/LaCie	Fixed	hfs	4,000,443,056,128	3,775,136,669,696
/Volumes/[C] Windows 10.hidden	Network	smbfs	136,844,406,784	43,311,140,864

Рис. 9.4

Работа с каталогами

Для работы с каталогами используйте статические классы `Directory`, `Path` и `Environment` (рис. 9.5).

При компоновке собственных файловых путей вы должны соблюдать осторожность и писать код так, чтобы не допускать в нем предположений о том, на какой платформе запущено ваше приложение, например, какой символ использовать в качестве разделителя каталогов.

Создайте метод `WorkWithDirectories` и напишите инструкции, которые выполняли бы следующие действия.

- Определение собственного файлового пути в домашнем каталоге пользователя путем создания строкового массива для хранения имен каталогов. После этого

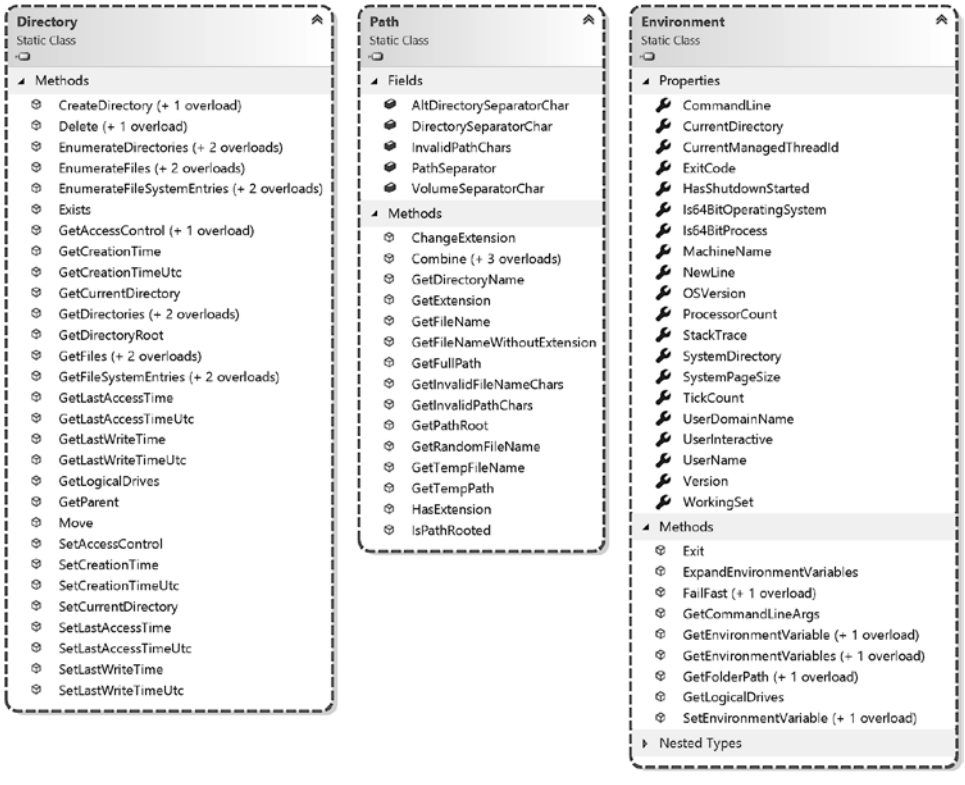


Рис. 9.5

элементы массива должны будут соединиться в правильный путь с помощью метода `Combine` типа `Path`.

- Проверка наличия создаваемого каталога `Create`.
- Удаление этого каталога, а также всех папок и файлов, хранящихся в нем:

```
static void WorkWithDirectories()
{
  // определение собственного пути к папке
  string userFolder = GetFolderPath(SpecialFolder.Personal);

  var customFolder = new string[]
  { userFolder, "Code", "Chapter09", "NewFolder" };
  string dir = Combine(customFolder);

  WriteLine($"Working with: {dir}");

  // проверка существования папки
  WriteLine($"Does it exist? {Exists(dir)}");

  // создание каталога
  WriteLine("Creating it...");
  CreateDirectory(dir);
```

```
  WriteLine($"Does it exist? {Exists(dir)}");
  Write("Confirm the directory exists, and then press ENTER: ");
  ReadLine();

  // удаление каталога
  WriteLine("Deleting it...");
  Delete(dir, recursive: true);
  WriteLine($"Does it exist? {Exists(dir)}");
}
```

В методе `Main` закомментируйте предыдущий вызов метода и добавьте новый вызов метода `WorkWithDirectories`, как показано в следующем листинге:

```
static void Main(string[] args)
{
  // OutputFileSystemInfo();
  // WorkWithDrives();

  WorkWithDirectories();
}
```

Запустите консольное приложение и проанализируйте результат вывода. Затем, прежде чем нажмете клавишу Enter и удалите созданный каталог, воспользуйтесь своим файловым менеджером и убедитесь в том, что каталог существует:

```
Working with: /Users/markjprice/Code/Chapter09/NewFolder
Does it exist? False
Creating it...
Does it exist? True
Confirm the directory exists, and then press ENTER:
Deleting it...
Does it exist? False
```

Управление файлами

При работе с файлами вы можете статически импортировать тип `File`, как и в случае типа `Directory`, но в следующем примере мы этого не сделаем, поскольку он содержит некоторые методы, что и тип `Directory`, и они конфликтуют. В этом случае тип `File` имеет достаточно короткое имя, поэтому проблема невелика.

Создайте метод `WorkWithFiles` и добавьте в него инструкции для следующих действий:

- проверка существования файла;
- создание текстового файла;
- запись текстовой строки в файл;
- резервное копирование файла;
- удаление оригинального файла;
- чтение содержимого файла из резервной копии.

```
static void WorkWithFiles()
{
  // определение пути к каталогу
```

```
  string userFolder = GetFolderPath(SpecialFolder.Personal);

  var customFolder = new string[]
  { userFolder, "Code", "Chapter09", "OutputFiles" };

  string dir = Combine(customFolder);
  CreateDirectory(dir);

  // определение путей к файлам
  string textFile = Combine(dir, "Dummy.txt");
  string backupFile = Combine(dir, "Dummy.bak");

  WriteLine($"Working with: {textFile}");

  // проверка существования файла
  WriteLine($"Does it exist? {File.Exists(textFile)}");

  // создание текстового файла и запись текстовой строки
  StreamWriter textWriter = File.CreateText(textFile);
  textWriter.WriteLine("Hello, C#!");
  textWriter.Close(); // закрытие файла и высвобождение ресурсов

  WriteLine($"Does it exist? {File.Exists(textFile)}");

  // копирование файла с перезаписью (если существует)
  File.Copy(
    sourceFileName: textFile,
    destFileName: backupFile,
    overwrite: true);

  WriteLine($"Does {backupFile} exist? {File.Exists(backupFile)}");

  Write("Confirm the files exist, and then press ENTER: ");
  ReadLine();

  // удаление файла
  File.Delete(textFile);

  WriteLine($"Does it exist? {File.Exists(textFile)}");

  // чтение содержимого текстового файла
  WriteLine($"Reading contents of {backupFile}:");
  StreamReader textReader = File.OpenText(backupFile);
  WriteLine(textReader.ReadToEnd());
  textReader.Close();
}
```

На платформе .NET Standard 2.0 после завершения работы с методами StreamReader и StreamWriter можно использовать метод Close или Dispose. А на платформе .NET Core 1.x доступен только метод Dispose, поскольку корпорация Microsoft упростила API.

В методе `Main` закомментируйте предыдущий вызов метода и добавьте вызов `WorkWithFiles`.

Запустите консольное приложение и проанализируйте результат вывода.

```
Working with: /Users/markjprice/Code/Chapter09/OutputFiles/Dummy.txt
Does it exist? False
Does it exist? True
Does /Users/markjprice/Code/Chapter09/OutputFiles/Dummy.bak exist? True
Confirm the files exist, and then press ENTER:
Does it exist? False
Reading contents of /Users/markjprice/Code/Chapter09/OutputFiles/Dummy.bak:
Hello, C#!
```

Управление путями

Порой нужно работать с путями, к примеру, извлечь только имя каталога, имя файла или расширение. Иногда понадобится создавать имена временных каталогов и файлов. Все это выполняется с помощью класса `Path`.

Добавьте следующие инструкции в конец метода `WorkWithFiles`:

```
WriteLine($"File Name: {GetFileName(textFile)}");
WriteLine($"File Name without Extension: {GetFileNameWithoutExtension(textFile)}");
WriteLine($"File Extension: {GetExtension(textFile)}");
WriteLine($"Random File Name: {GetRandomFileName()}");
WriteLine($"Temporary File Name: {GetTempFileName()}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
File Name: Dummy.txt
File Name without Extension: Dummy
File Extension: .txt
Random File Name: u45w1zki.co3
Temporary File Name:
/var/folders/tz/xx0y_wld5sx0nv0fjtq4tnpc0000gn/T/tmpyqrepP.tmp
```

Метод GetTempFileName создает файл с нулевым размером и возвращает его имя, готовое к использованию. А метод GetRandomFileName просто возвращает имя файла, не создавая сам файл.

Извлечение информации о файле

Чтобы получить дополнительную информацию о файле или каталоге (например, каков его размер или когда к нему обращались в последний раз), можно создать экземпляр класса `FileInfo` или `DirectoryInfo` (рис. 9.6).

Оба класса, и FileInfo и DirectoryInfo, наследуются от FileSystemInfo и потому содержат такие члены, как LastAccessTime и Delete.

Добавьте следующие инструкции в конец метода `WorkWithFiles`:

```
var info = new FileInfo(backupFile);
WriteLine($"{backupFile}:");
```

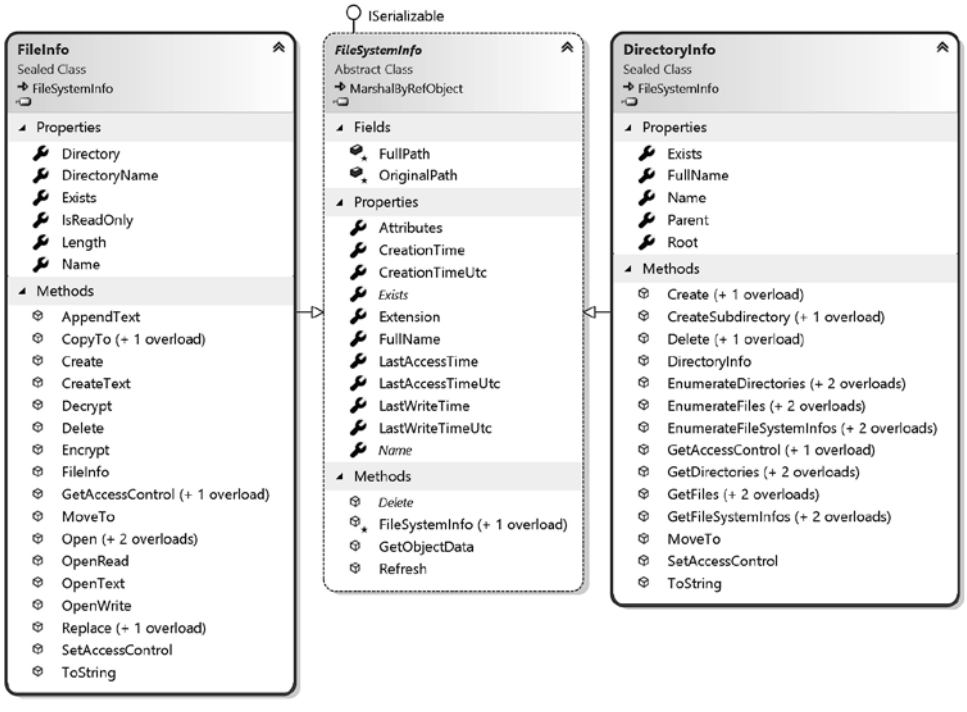


Рис. 9.6

```
WriteLine($" Contains {info.Length} bytes");
WriteLine($" Last accessed {info.LastAccessTime}");
WriteLine($" Has readonly set to {info.IsReadOnly}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
/Users/markjprice/Code/Chapter09/OutputFiles/Dummy.bak:
  Contains 11 bytes
  Last accessed 26/08/2017 09:08:26
  Has readonly set to False
```

Управление файлами

При работе с файлами зачастую требуется управлять дополнительными опциями. Для этого подойдет метод `File.Open`, поскольку в нем реализованы определенные перегрузки для указания дополнительных опций с помощью значений `enum` (рис. 9.7).

Доступны следующие свойства типа `enum`:

- `FileMode` — указывает, что вы хотите сделать с файлом;
- `FileAccess` — отслеживает требуемый уровень доступа;
- `FileShare` — контролирует блокировки файла, что позволяет установить указанный уровень доступа для других процессов;

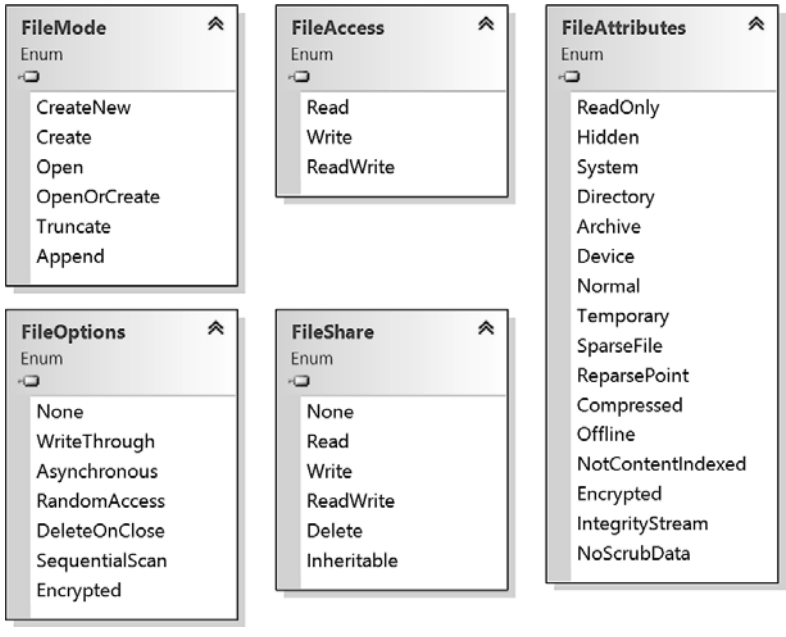


Рис. 9.7

- `FileOptions` — служит для установки дополнительных опций;
- `FileAttributes` — применяется для типа, наследующего от `FileSystemInfo`, воспользуйтесь этим типом `enum` для проверки свойства `Attributes`.

Иногда требуется открыть файл, прочитать его, а также позволить другим процессам считывать этот файл, как показано в следующем листинге:

```
FileStream file = File.Open(pathToFile, FileMode.Open, FileAccess.Read,
FileShare.Read);
```

Порой нужно проверить атрибуты файла или каталога, как показано в следующем листинге:

```
var info = new FileInfo(backupFile);
WriteLine($"Compressed?
{info.Attributes.HasFlag(FileAttributes.Compressed)}");
```

Чтение и запись с помощью потоков

Поток — это последовательность байтов, которую можно считать или в которую можно записать некие данные. Несмотря на то что файлы могут быть обработаны как массивы (что позволяет реализовать случайный доступ, если известно положение нужного бита в файле), обработка файла как потока, в котором доступ к байтам осуществляется последовательно, также может оказаться полезной.

Потоки могут использоваться и для обработки ввода и вывода терминала и сетевых ресурсов, таких как сокеты и порты, не предоставляющие случайный доступ

и не имеющие возможности выполнять поиск требуемой позиции. Таким образом, вы можете написать код для обработки произвольного набора байтов, не обращая внимания на их источник или не зная его. Ваш код просто считывает или записывает в поток, а другой код обрабатывает хранение этих байтов.

Существует абстрактный класс `Stream`, представляющий собой поток. Есть много классов, которые наследуются от этого базового класса, вследствие чего все они работают одинаково (рис. 9.8).

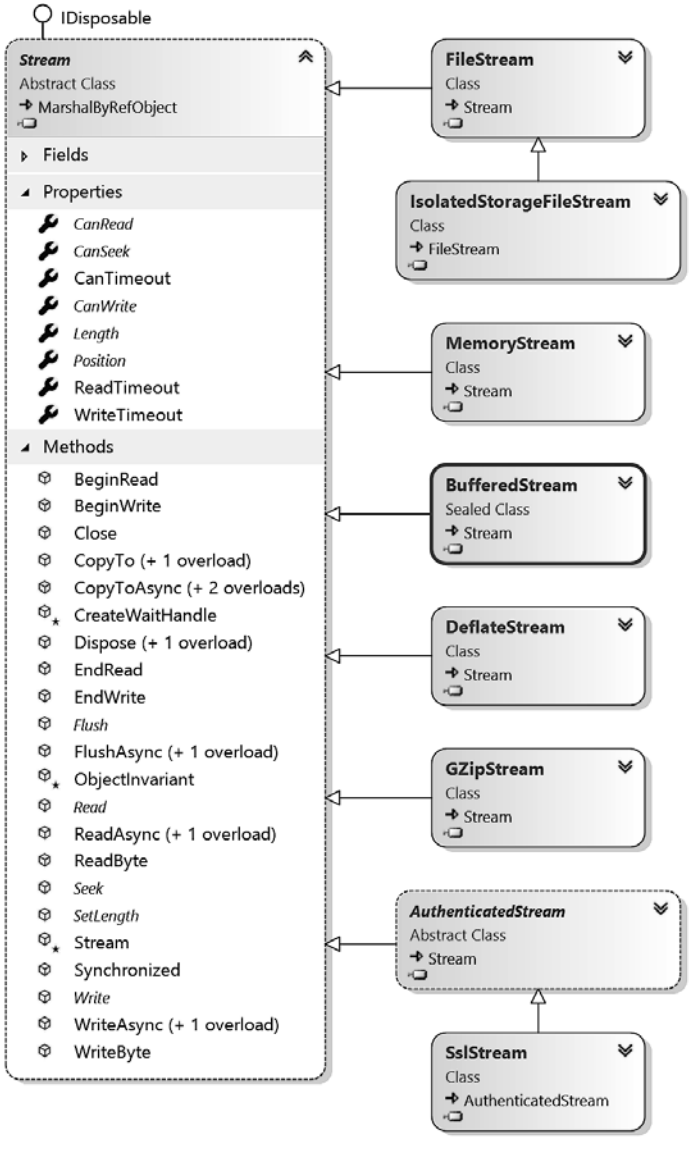


Рис. 9.8

Все потоки реализуют метод IDisposable, поэтому содержат метод Dispose для освобождения неуправляемых ресурсов.

В табл. 9.1 приведены некоторые из универсальных членов класса `Stream`.

Таблица 9.1

Член	Описание
CanRead, CanWrite	Определяет, поддерживает ли текущий поток возможность чтения и записи соответственно
Length, Position	Определяет длину потока в байтах и текущую позицию в потоке
Dispose()	Закрывает поток и освобождает его ресурсы
Flush()	Если поток имеет буфер, он очищается и записывается в основной поток
Read(), ReadAsync()	Считывает определенное количество байтов из потока в байтовый массив и перемещает позицию синхронно и асинхронно
ReadByte()	Считывает байт из потока и перемещает позицию
Seek()	Задает позицию в текущем потоке (если значение CanSeek истинно)
Write(), WriteAsync()	Записывает последовательность байтов в текущий поток, синхронно и асинхронно соответственно
WriteByte()	Записывает байт в поток

Запоминающие потоки можно прочитать и записать, а байты сохранить в данной позиции (табл. 9.2).

Таблица 9.2

Пространство имен	Класс	Описание
System.IO	FileStream	Создает поток, хранилищем которого является файловая система
System.IO	Memory Stream	Создает поток, хранилищем которого выступает память
System.Net.Sockets	NetworkStream	Создает поток, хранилищем которого служит сеть

Функциональные потоки можно «подключить» к другим потокам, чтобы расширить их функциональность (табл. 9.3).

Таблица 9.3

Пространство имен	Класс	Описание
System.Security.Cryptography	CryptoStream	Шифрует и дешифрует поток
System.IO.Compression	GZipStream, DeflateStream	Сжимает и распаковывает поток
System.Net.Security	AuthenticatedStream	Передает учетные данные через поток

Иногда действительно приходится работать с потоками на низком уровне. Но чаще можно собрать цепочку вспомогательных классов, чтобы упростить решение типичных задач (рис. 9.9).

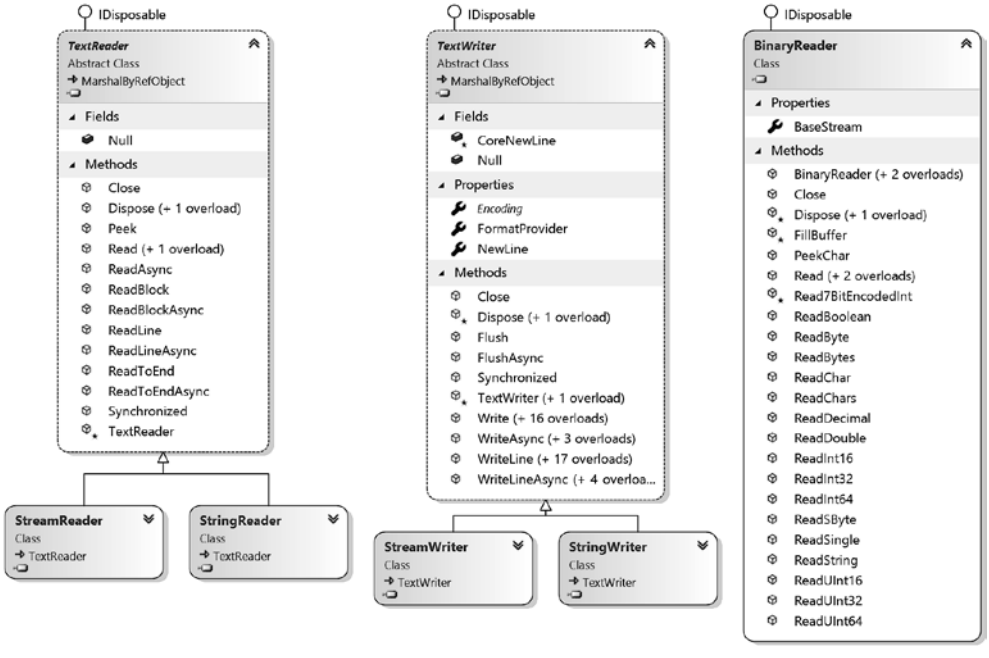


Рис. 9.9

Все вспомогательные типы реализуют метод IDisposable, поэтому содержат метод Dispose для освобождения неуправляемых ресурсов.

В табл. 9.4 представлены некоторые вспомогательные классы.

Таблица 9.4

Пространство имен	Класс	Описание
System.IO	StreamReader	Считывает данные из потока в текстовом формате
System.IO	StreamWriter	Записывает данные в поток в текстовом формате
System.IO	BinaryReader	Считывает данные из потока в виде типов .NET
System.IO	BinaryWriter	Записывает данные в поток в виде типов .NET
System.Xml	XmlReader	Считывает данные из потока в XML-формате
System.Xml	XmlWriter	Записывает данные в поток в XML-формате

Запись в текстовые и XML-потоки

Создайте новый проект консольного приложения с именем `WorkingWithStreams`.

На панели Solution Explorer (Обозреватель решений) в Visual Studio 2017 щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства). В категории Startup Project (Запускаемый проект) установите переключатель в положение Current selection (Текущий проект).

Запись в текстовые потоки

Импортируйте пространства имен `System.IO` и `System.Xml` и статически импортируйте типы `System.Console`, `System.Environment` и `System.IO.Path`.

Определите массив позывных пилота вертолета Viper и создайте метод `WorkWithText`, перечисляющий позывные, записывая каждый из них в текстовый файл, как показано в следующем листинге:

```
// определение массива позывных пилота Viper
static string[] callsigns = new string[] { "Husker", "Starbuck",
"Apollo", "Boomer", "Bulldog", "Athena", "Helo", "Racetrack" };

static void WorkWithText()
{
  // определение файла для записи
  string textFile = Combine(CurrentDirectory, "streams.txt");

  // создание текстового файла и возвращение помощника записи
  StreamWriter text = File.CreateText(textFile);

  // перечисление строк с записью каждой из них в поток в отдельной строке
  foreach (string item in callsigns)
  {
    text.WriteLine(item);
  }
  text.Close(); // release resources

  // вывод содержимого файла в консоль
  WriteLine($"{textFile} contains
  {new FileInfo(textFile).Length} bytes.");
  WriteLine(File.ReadAllText(textFile));
}
```

В методе `Main` вызовите метод `WorkWithText`, как показано в листинге ниже:

```
static void Main(string[] args)
{
  WorkWithText();
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.txt contains 60
bytes.
Husker
```

```
Starbuck
Apollo
Boomer
Bulldog
Athena
Helo
Racetrack
```

Откройте созданный файл и удостоверьтесь, что он содержит список позывных.

Запись в потоки XML

Создайте метод `WorkWithXml`, перечисляющий все позывные, записывая каждый из них в файл XML, как показано в следующем листинге:

```
static void WorkWithXml()
{
  // определение файла для записи
  string xmlFile = Combine(CurrentDirectory, "streams.xml");

  // создание файловых потоков
  FileStream xmlFileStream = File.Create(xmlFile);

  // оборачивание файлового потока в помощник записи XML
  // и автоматическое добавление отступов для вложенных элементов
  XmlWriter xml = XmlWriter.Create(xmlFileStream,
  new XmlWriterSettings { Indent = true });

  // запись объявления XML
  xml.WriteStartDocument();

  // запись корневого элемента
  xml.WriteStartElement("callsigns");

  // перечисление строк и запись каждой в поток
  foreach (string item in callsigns)
  {
    xml.WriteElementString("callsign", item);
  }

  // запись закрывающего корневого элемента
  xml.WriteEndElement();

  // закрытие помощника и потока
  xml.Close();
  xmlFileStream.Close();

  // вывод содержимого файла в консоль
  WriteLine($"{xmlFile} contains {new FileInfo(xmlFile).Length} bytes.");
  WriteLine(File.ReadAllText(xmlFile));
}
```

В методе `Main` закомментируйте предыдущий вызов метода и добавьте новый вызов метода `WorkWithXml`.

Запустите консольное приложение и проанализируйте результат вывода:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.xml contains
310 bytes.
<?xml version="1.0" encoding="utf-8"?>
<callsigns>
  <callsign>Husker</callsign>
  <callsign>Starbuck</callsign>
  <callsign>Apollo</callsign>
  <callsign>Boomer</callsign>
  <callsign>Bulldog</callsign>
  <callsign>Athena</callsign>
  <callsign>Helo</callsign>
  <callsign>Racetrack</callsign>
</callsigns>
```

Освобождение файловых ресурсов

При открытии файла для чтения или записи вы используете ресурсы вне .NET. Эти ресурсы называются неуправляемыми и должны быть освобождены по окончании работы с ними. Чтобы гарантировать их освобождение, можно вызывать метод `Dispose` внутри блока `finally`.

Освобождение ресурсов с помощью инструкции try

Измените метод `WorkWithXml`, как показано в следующем листинге (выделено полужирным шрифтом):

```
static void WorkWithXml()
{
  FileStream xmlFileStream = null;
  XmlWriter xml = null;
  try
  {
    // определение файла для записи
    string xmlFile = Combine(CurrentDirectory, "streams.xml");

    // создание файловых потоков
    xmlFileStream = File.Create(xmlFile);

    // оборачивание файлового потока в помощник записи XML
    // и автоматическое добавление отступов для вложенных элементов
    xml = XmlWriter.Create(xmlFileStream,
    new XmlWriterSettings { Indent = true });

    // запись объявления XML
    xml.WriteStartDocument();

    // запись корневого элемента
    xml.WriteStartElement("callsigns");

    // перечисление строк и запись каждой в поток
```

```
    foreach (string item in callsigns)
    {
      xml.WriteElementString("callsign", item);
    }

    // запись закрывающего корневого элемента
    xml.WriteEndElement();

    // закрытие помощника и потока
    xml.Close();
    xmlFileStream.Close();

    // вывод содержимого файла в консоль
    WriteLine($"{xmlFile} contains
    {new FileInfo(xmlFile).Length} bytes.");
    WriteLine(File.ReadAllText(xmlFile));
  }
  catch(Exception ex)
  {
    // если путь не существует, то исключение будет перехвачено
    WriteLine($"{ex.GetType()} says {ex.Message}");
  }
  finally
  {
    if (xml != null)
    {
      xml.Dispose();
      WriteLine("The XML writer's unmanaged
      resources have been disposed.");
    }
    if (xmlFileStream != null)
    {
      xmlFileStream.Dispose();
      WriteLine("The file stream's unmanaged
      resources have been disposed.");
    }
  }
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
The XML writer's unmanaged resources have been disposed.
The file stream's unmanaged resources have been disposed.
```

Перед вызовом метода Dispose проверьте, что объект не равен null.

Упрощение освобождения ресурсов с помощью инструкции using

Если не нужно перехватывать исключения, то можно упростить код, которому потребуется лишь проверить, что объект не равен `null`, а затем вызвать метод `Dispose` с помощью инструкции `using`.

Это может запутать, но у инструкции using две функции: импорт пространства имен и генерация инструкции finally, освобождающей объект.

Компилятор заменит ваш вариант на полный код инструкции `try` и `finally`, но без использования инструкции `catch`. Вы можете применить вложенные инструкции `try`, таким образом, если не хотите перехватывать исключения, можете написать код, как показано в следующем листинге:

```
using (FileStream file2 = File.OpenWrite(Path.Combine(path, "file2.txt")))
{
  using (StreamWriter writer2 = new StreamWriter(file2))
  {
    try
    {
      writer2.WriteLine("Welcome, .NET Core!");
    }
    catch(Exception ex)
    {
      WriteLine($"{ex.GetType()} says {ex.Message}");
    }
  } // автоматически вызывает Dispose, если объект не равен null
} // автоматически вызывает Dispose, если объект не равен null
```

Многие типы, в том числе и упомянутые ранее FileStream и StreamWriter, предоставляют как метод Close, так и метод Dispose. В .NET Standard 2.0 можно применять любой из них, поскольку эти методы выполняют одинаковые операции, в буквальном смысле вызывая друг друга. В .NET Core 1.1 разработчики Microsoft слишком сильно упростили соответствующие интерфейсы API, вследствие чего приходилось безальтернативно использовать метод Dispose.

Сжатие потоков

Формат XML довольно объемный, поэтому занимает больше памяти в байтах, чем обычный текст. Можно сжать XML-данные, воспользовавшись универсальным алгоритмом сжатия, известным как GZIP.

Импортируйте следующее пространство имен:

```
using System.IO.Compression;
```

Добавьте метод `WorkWithCompression`, как показано ниже:

```
static void WorkWithCompression()
{
  // сжатие XML-вывода
  string gzipFilePath = Combine(CurrentDirectory, "streams.gzip");

  FileStream gzipFile = File.Create(gzipFilePath);
  using (GZipStream compressor =
  new GZipStream(gzipFile, CompressionMode.Compress))
```

```
  {
    using (XmlWriter xmlGzip = XmlWriter.Create(compressor))
    {
      xmlGzip.WriteStartDocument();
      xmlGzip.WriteStartElement("callsigns");
      foreach (string item in callsigns)
      {
        xmlGzip.WriteElementString("callsign", item);
      }
    }
  } // также закрывает базовый поток

  // выводит все содержимое сжатого файла в консоль
  WriteLine($"{gzipFilePath} contains
  {new FileInfo(gzipFilePath).Length} bytes.");
  WriteLine(File.ReadAllText(gzipFilePath));

  // чтение сжатого файла
  WriteLine("Reading the compressed XML file:");
  gzipFile = File.Open(gzipFilePath, FileMode.Open);
  using (GZipStream decompressor = new GZipStream(gzipFile,
  CompressionMode.Decompress))
  {
    using (XmlReader reader = XmlReader.Create(decompressor))
    {
      while (reader.Read())
      {
        // проверка, находимся ли мы в данный момент на узле
        // элемента с именем callsign
        if ((reader.NodeType == XmlNodeType.Element) &&
            (reader.Name == "callsign"))
        {
          reader.Read(); // переход к текстовому узлу внутри элемента
          WriteLine($"{reader.Value}"); // считывание значения
        }
      }
    }
  }
}
```

В методе `Main` оставьте вызов `WorkWithXml` и добавьте вызов `WorkWithCompression`, как показано в листинге ниже:

```
static void Main(string[] args)
{
  // WorkWithText();
  WorkWithXml();
  WorkWithCompression();
}
```

Перезапустите приложение и обратите внимание, что сжатые XML-данные занимают вполовину меньше объема памяти по сравнению с таким же количеством несжатых XML-данных.

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.xml contains 310 bytes.
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.gzip contains 150 bytes.
```

Кодирование текста

Текстовые символы можно представить разными способами. Например, алфавит может быть закодирован с помощью азбуки Морзе в серии точек и тире для передачи по телеграфной линии.

Аналогичным образом текст в памяти компьютера сохраняется в виде битов (единиц и нулей). Платформа .NET Core задействует стандарт, называемый *Юникодом*, для внутреннего кодирования текста. В некоторых случаях понадобится выводить текст за пределы платформы .NET, чтобы им могли пользоваться системы, не поддерживающие Юникод или применяющие вариации этого стандарта.

В табл. 9.5 перечислен ряд альтернативных кодировок текста, которые обычно используются на компьютерах.

Таблица 9.5

Кодировка	Описание
ASCII	Кодирует ограниченный диапазон символов, используя семь младших битов байта
UTF-8	Представляет каждую кодовую точку Юникода в виде последовательности от одного до четырех байт
UTF-16	Представляет каждую кодовую точку Юникода в виде последовательности из одного или двух 16-разрядных целых чисел
Кодировки ANSI и ISO	Обеспечивает поддержку ряда кодовых страниц, которые используются для поддержки конкретного языка или группы языков

Преобразование строк в последовательности байтов

Создайте новый проект консольного приложения с именем `WorkingWithEncodings`.

Импортируйте пространство имен `System.Text`, статически импортируйте `Console` и добавьте следующие инструкции в метод `Main`. Приложение кодирует строку, используя выбранную кодировку, перебирает каждый байт, а затем декодирует код обратно в строку и выводит ее.

```
WriteLine("Encodings");
WriteLine("[1] ASCII");
WriteLine("[2] UTF-7");
WriteLine("[3] UTF-8");
WriteLine("[4] UTF-16 (Unicode)");
WriteLine("[5] UTF-32");
WriteLine("[any other key] Default");

// выбор кодировки
Write("Press a number to choose an encoding: ");
ConsoleKey number = ReadKey(false).Key;
WriteLine();
WriteLine();

Encoding encoder;
switch (number)
```

```
{
  case ConsoleKey.D1:
    encoder = Encoding.ASCII;
    break;
  case ConsoleKey.D2:
    encoder = Encoding.UTF7;
    break;
  case ConsoleKey.D3:
    encoder = Encoding.UTF8;
    break;
  case ConsoleKey.D4:
    encoder = Encoding.Unicode;
    break;
  case ConsoleKey.D5:
    encoder = Encoding.UTF32;
    break;
  default:
    encoder = Encoding.Default;
    break;
}

// определение строки для кодирования
string message = "A pint of milk is £1.99";

// кодирование строки в последовательность байтов
byte[] encoded = encoder.GetBytes(message);

// проверка количества байтов, необходимого для кодирования
WriteLine($"{encoder.GetType().Name} uses {encoded.Length} bytes.");

// перечисление каждого байта
WriteLine($"Byte Hex Char");
foreach (byte b in encoded)
{
  WriteLine($"{b,4} {b.ToString("X"),4} {(char)b,5}");
}

// декодирование последовательности байтов обратно в строку и ее вывод
string decoded = encoder.GetString(encoded);
WriteLine(decoded);
```

Версии .NET Core 1.0 и 1.1 не поддерживают Encoding.Default, вместо него используется код GetEncoding(0).

Запустите приложение и нажмите клавишу 1, чтобы выбрать кодировку ASCII. Обратите внимание: при выводе байтов в данной кодировке не может быть представлен символ фунта стерлингов (£), поэтому вместо него используется знак вопроса (?).

```
Encodings
[1] ASCII
[2] UTF-7
[3] UTF-8
[4] UTF-16 (Unicode)
[5] UTF-32
```

```
[any other key] Default
Press a number to choose an encoding: 1
ASCIIEncoding uses 23 bytes.
Byte    Hex   Char
  65     41     A
  32     20
 112     70     p
 105     69     i
 110     6E     n
 116     74     t
  32     20
 111     6F     o
 102     66     f
  32     20
 109     6D     m
 105     69     i
 108     6C     l
 107     6B     k
  32     20
 105     69     i
 115     73     s
  32     20
  63     3F     ?
  49     31     1
  46     2E     .
  57     39     9
  57     39     9
A pint of milk is ?1.99
```

Перезапустите приложение и нажмите клавишу 3, чтобы выбрать кодировку UTF-8. Обратите внимание: UTF-8 требует дополнительный байт (24 байта вместо 23 байт) для хранения данных, но корректно сохраняет символ £.

```
UTF8Encoding uses 24 bytes.
Byte    Hex   Char
  65     41     A
  32     20
 112     70     p
 105     69     i
 110     6E     n
 116     74     t
  32     20
 111     6F     o
 102     66     f
  32     20
 109     6D     m
 105     69     i
 108     6C     l
 107     6B     k
  32     20
 105     69     i
 115     73     s
  32     20
 194     C2     Â
 163     A3     £
  49     31     1
```

```
  46     2E     .
  57     39     9
  57     39     9
A pint of milk is £1.99
```

Перезапустите приложение и нажмите клавишу 4, чтобы выбрать кодировку UTF-16. Обратите внимание: UTF-16 требует два байта для хранения каждого символа, но корректно сохраняет символ £.

```
UnicodeEncoding uses 46 bytes.
```

Кодирование/декодирование текста в файлах

При использовании вспомогательных классов потоков, таких как `StreamReader` и `StreamWriter`, можно указать предпочтительную кодировку. При записи во вспомогательный поток строки будут кодироваться автоматически, а при чтении из него — автоматически декодироваться. Ниже показано, как указать кодировку:

```
var reader = new StreamReader(stream, Encoding.UTF7);
var writer = new StreamWriter(stream, Encoding.UTF7);
```

Зачастую вы не сможете выбирать кодировку, поскольку будете генерировать файл для использования в другой системе. Но если возможность есть, то выбирайте такую кодировку, которая занимает наименьший объем памяти (количество байтов), но поддерживает все необходимые символы.

Сериализация графов объектов

Сериализация — это процесс преобразования объекта в поток байтов в выбранном формате. Обратный процесс называется *десериализацией*.

Существуют десятки форматов, доступных для выбора; среди них два наиболее распространенных — *расширяемый язык разметки* (eXtensible Markup Language, XML) и *объектная нотация JavaScript* (JavaScript Object Notation, JSON).

Формат JSON компактнее и лучше подходит для веб- и мобильных приложений. Формат XML более объемный, но лучше поддерживается старыми системами.

Платформа .NET Standard содержит несколько классов для обеспечения сериализации в форматы XML и JSON (и из них). Мы начнем с классов `XmlSerializer` и `JsonSerializer`.

XML-сериализация

Создайте новый проект консольного приложения с именем `WorkingWithSerialization`.

В качестве универсального примера мы определим пользовательский класс для хранения информации о человеке, а затем создадим граф объектов, применяя список экземпляров `Person` с вложением.

Добавьте класс `Person` с указанным ниже определением. Обратите внимание, что свойство `Salary` закрыто, то есть доступно только внутри себя и производных классов. Для указания зарплаты класс содержит конструктор с единственным параметром, устанавливающий начальный оклад.

```
using System;
using System.Collections.Generic;

namespace Packt.CS7
{
  public class Person
  {
    public Person(decimal initialSalary)
    {
      Salary = initialSalary;
    }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public HashSet<Person> Children { get; set; }
    protected decimal Salary { get; set; }
  }
}
```

Вернитесь к файлу `Program.cs` и импортируйте следующие пространства имен и статические типы:

```
using System;                           // DateTime
using System.Collections.Generic;       // List<T>, HashSet<T>
using System.Xml.Serialization;         // XmlSerializer
using System.IO;                        // FileStream
using Packt.CS7;                        // Person
using static System.Console;
using static System.Environment;
using static System.IO.Path;
```

Добавьте следующие инструкции в метод `Main`:

```
// создание графа объектов
var people = new List<Person>
{
  new Person(30000M) { FirstName = "Alice", LastName = "Smith",
  DateOfBirth = new DateTime(1974, 3, 14) },
  new Person(40000M) { FirstName = "Bob", LastName = "Jones",
  DateOfBirth = new DateTime(1969, 11, 23) },
  new Person(20000M) { FirstName = "Charlie", LastName = "Rose",
  DateOfBirth = new DateTime(1964, 5, 4),
  Children = new HashSet<Person>
  { new Person(0M) { FirstName = "Sally", LastName = "Rose",
    DateOfBirth = new DateTime(1990, 7, 12) } } }
  };

  // создание файла для записи
```

```
string path = Combine(CurrentDirectory, "people.xml");

FileStream stream = File.Create(path);

// создание объекта, который будет отформатирован в виде
// списка людей в формате XML
var xs = new XmlSerializer(typeof(List<Person>));

// сериализация графа объектов в поток
xs.Serialize(stream, people);

// необходимо закрыть поток, чтобы разблокировать файл
stream.Close();

WriteLine($"Written {new FileInfo(path).Length} bytes of XML to {path}");
WriteLine();

// отображение сериализованного графа объектов
WriteLine(File.ReadAllText(path));
```

Запустите консольное приложение и проанализируйте результат вывода. Обратите внимание на вызываемое исключение:

```
Unhandled Exception: System.InvalidOperationException: Packt.CS7.Person cannot be
serialized because it does not have a parameterless constructor.
```

Вернитесь к файлу `Person.cs` и добавьте следующую инструкцию для определения конструктора без параметров. Обратите внимание: конструктору ничего не требуется делать, но он должен присутствовать, чтобы класс `XmlSerializer` мог его вызвать для создания новых экземпляров `Person` в процессе десериализации.

```
public Person() { }
```

Перезапустите консольное приложение и проанализируйте результат вывода.

Обратите внимание, что граф объектов сериализуется в формате XML и свойство `Salary` не используется:

```
Written 754 bytes of XML to
/Users/markjprice/Code/Chapter09/WorkingWithSerialization/people.xml
<?xml version="1.0"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Person>
    <FirstName>Alice</FirstName>
    <LastName>Smith</LastName>
    <DateOfBirth>1974-03-14T00:00:00</DateOfBirth>
  </Person>
  <Person>
    <FirstName>Bob</FirstName>
    <LastName>Jones</LastName>
    <DateOfBirth>1969-11-23T00:00:00</DateOfBirth>
  </Person>
  <Person>
```

```
    <FirstName>Charlie</FirstName>
    <LastName>Rose</LastName>
    <DateOfBirth>1964-05-04T00:00:00</DateOfBirth>
    <Children>
      <Person>
        <FirstName>Sally</FirstName>
        <LastName>Rose</LastName>
        <DateOfBirth>1990-07-12T00:00:00</DateOfBirth>
      </Person>
    </Children>
  </Person>
</ArrayOfPerson>
```

Формат XML можно использовать более эффективно, если для некоторых полей вместо элементов применить атрибуты.

В файле `Person.cs` импортируйте пространство имен `System.Xml.Serialization` и измените все свойства, кроме `Children`, добавив атрибут `[XmlAttribute]` и указав короткое имя, как показано в листинге ниже:

```
[XmlAttribute("fname")]
public string FirstName { get; set; }
[XmlAttribute("lname")]
public string LastName { get; set; }
[XmlAttribute("dob")]
public DateTime DateOfBirth { get; set; }
```

Перезапустите приложение и обратите внимание на более эффективное использование формата XML:

```
Written 464 bytes of XML to C:\Code\Ch10_People.xml
<?xml version="1.0"?>
  <ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Person fname="Alice" lname="Smith" dob="1974-03-14T00:00:00" />
    <Person fname="Bob" lname="Jones" dob="1969-11-23T00:00:00" />
    <Person fname="Charlie" lname="Rose" dob="1964-05-04T00:00:00">
      <Children>
        <Person fname="Sally" lname="Rose" dob="1990-07-12T00:00:00" />
      </Children>
    </Person>
  </ArrayOfPerson>
```

Размер файла был уменьшен с 754 до 464 байт, то есть на 38 %.

XML-десериализация

Добавьте следующие инструкции в конец метода `Main`:

```
FileStream xmlLoad = File.Open(path, FileMode.Open);
// десериализация и приведение графа объектов к списку людей
var loadedPeople = (List<Person>)xs.Deserialize(xmlLoad);

foreach (var item in loadedPeople)
```

```
{
  WriteLine($"{item.LastName} has {item.Children.Count}
  children.");
}
xmlLoad.Close();
```

Перезапустите приложение и обратите внимание, что информация о людях успешно загружается из XML-файла:

```
Smith has 0 children.
Jones has 0 children.
Rose has 1 children.
```

Контроль результата XML-сериализации

Существует множество других атрибутов, которые можно использовать при работе с форматом XML. Для получения дополнительной информации см. ссылки в конце этой главы.

Во время применения класса XmlSerializer не забывайте о том, что учитываются только открытые (public) поля и свойства, а тип должен содержать конструктор без параметров. Кроме того, с помощью атрибутов можно настроить вывод.

JSON-сериализация

Лучшей библиотекой для работы с форматом сериализации JSON считается *Newtonsoft.Json*.

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) раскройте проект WorkingWithSerialization, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Manage NuGet Packages (Управление пакетами NuGet). На открывшейся панели перейдите на вкладку Browse (Обзор), выполните поиск пакета Newtonsoft.Json, выберите его и нажмите кнопку Install (Установить).

В Visual Studio Code отредактируйте файл `WorkingWithSerialization.csproj`, добавив ссылку на пакет Newtonsoft.Json, как показано в листинге ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.3" />
  </ItemGroup>

</Project>
```

Выполняйте поиск NuGet-пакетов в новостной ленте Microsoft NuGet, чтобы узнать информацию о последних поддерживаемых версиях (см. ссылку https://www.nuget.org/packages/Newtonsoft.Json/).

В начало файла `Program.cs` импортируйте следующее пространство имен:

```
using Newtonsoft.Json;
```

Добавьте эти инструкции в конец метода `Main`:

```
// создание файла для записи
string jsonPath = Combine(CurrentDirectory, "people.json");

StreamWriter jsonStream = File.CreateText(jsonPath);

// создание объекта для форматирования в JSON
var jss = new JsonSerializer();

// сериализация графа объектов в строку
jss.Serialize(jsonStream, people);
jsonStream.Close(); // разблокировать файл

WriteLine();
WriteLine($"Written {new FileInfo(jsonPath).Length} bytes of
JSON to: {jsonPath}");

// отображение сериализованного графа объектов
WriteLine(File.ReadAllText(jsonPath));
```

Перезапустите приложение и обратите внимание, что формат JSON занимает более чем вполовину меньше байтов памяти по сравнению с форматом XML. Его размер даже меньше, чем XML с атрибутами:

```
Written 368 bytes of JSON to:
/Users/markjprice/Code/Chapter09/WorkingWithSerialization/people.json
[{"FirstName":"Alice","LastName":"Smith","DateOfBirth":"\/Date(132451200000)\
/","Children":null},{"FirstName":"Bob","LastName":"Jones","DateOfBirth":"\/Date
(-3369600000)\/","Children":null},{"FirstName":"Charlie","LastName":"Rose",
"DateOfBirth":"\/Date(-178678800000)\/","Children":[{"FirstName":"Sally",
"LastName":"Rose","DateOfBirth":"\/Date(647737200000)\/","Children":null}]}]
```

Используйте формат JSON для минимизации размера сериализованных графов объектов. Он также отлично подойдет при отправке графов объектов в веб- и мобильные приложения, поскольку JSON — собственный формат сериализации языка JavaScript.

Сериализация в другие форматы

Существует много других форматов, доступных в виде NuGet-пакетов, которые можно использовать для сериализации. Чаще всего применяются следующие: `DataContractSerializer` (для XML) и `DataContractJsonSerializer` (для JSON), находящиеся в пространстве имен `System.Runtime.Serialization`.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенное упражнение и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

1. Чем отличается применение классов `File` и `FileInfo`?
2. В чем разница между методом `ReadByte` и методом `Read` потока?
3. В каких случаях используются классы `StringReader`, `TextReader` и `StreamReader`?
4. Для чего предназначен тип `DeflateStream`?
5. Сколько байтов затрачивается на символ при использовании кодировки UTF-8?
6. Что такое граф объектов?
7. Какой формат сериализации лучше всего подходит для наименьших затрат памяти?
8. Какой формат сериализации наиболее пригоден для кросс-платформенной совместимости?
9. Какая библиотека лучше всего подходит для работы с форматом сериализации JSON?
10. Сколько пакетов для сериализации доступно на сайте NuGet.org?

Упражнение 9.1. XML-сериализация

Создайте консольное приложение с именем `Exercise02`, генерирующее список фигур, выполняющее XML-сериализацию, для сохранения его в файловой системе, а затем десериализующее его обратно:

```
// создание списка фигур для сериализации
var listOfShapes = new List<Shape>
{
  new Circle { Colour = "Red", Radius = 2.5 },
  new Rectangle { Colour = "Blue", Height = 20.0, Width = 10.0 },
  new Circle { Colour = "Green", Radius = 8 },
  new Circle { Colour = "Purple", Radius = 12.3 },
  new Rectangle { Colour = "Blue", Height = 45.0, Width = 18.0 }
};
```

Фигуры должны содержать свойство только для чтения с именем `Area`, чтобы при десериализации можно было выводить список фигур, включая их площадь, как показано ниже:

```
List<Shape> loadedShapesXml = serializerXml.Deserialize(fileXml)
as List<Shape>;
```

```
foreach (Shape item in loadedShapesXml)
{
  WriteLine($"{item.GetType().Name} is {item.Colour} and has an area of {item.Area}");
}
```

После запуска приложения вывод должен выглядеть примерно так:

```
Loading shapes from XML:
Circle is Red and has an area of 19.6349540849362
Rectangle is Blue and has an area of 200
Circle is Green and has an area of 201.061929829747
Circle is Purple and has an area of 475.2915525616
Rectangle is Blue and has an area of 810
```

Дополнительные ресурсы

- Файловая система и реестр (руководство по программированию на C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/file-system/.
- Кодировка символов на платформе .NET: https://docs.microsoft.com/en-us/dotnet/articles/standard/base-types/character-encoding.
- Сериализация (C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/concepts/serialization/.
- Сериализация в файлы и объекты `TextWriters` и `XmlWriters`: https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/concepts/linq/serializing-to-files-textwriters-and-xmlwriters.
- Newtonsoft Json.NET: http://www.newtonsoft.com/json.

Резюме

В этой главе вы узнали, как выполнять чтение и запись в текстовые файлы и формат XML, сжимать и распаковывать файлы, кодировать/декодировать текст и сериализовать объекты в форматы JSON и XML (и десериализовать их обратно).

В следующей главе вы займетесь защитой данных и файлов.

10 Защита данных и приложений

Эта глава посвящена защите данных с помощью шифрования, хеширования и цифровых подписей от постороннего доступа, изменения и повреждения.

В данной главе:

- терминология безопасности;
- шифрование и дешифрование данных;
- хеширование данных;
- подписывание данных;
- генерация случайных чисел;
- аутентификация и авторизация пользователей.

Терминология безопасности

Защитить данные призвано множество способов; некоторые из них перечислены ниже.

- *Шифрование и дешифрование*. Двухсторонний процесс преобразования обычного текста в шифр и обратно.
- *Хеши*. Односторонний процесс генерации хеш-значения для безопасного хранения паролей или обнаружения вредоносных изменений либо повреждений данных.
- *Цифровые подписи*. Метод проверки доверенного источника поступивших данных (то есть подтверждение, что данные пришли именно от этого человека) путем верификации цифровой подписи данных по открытому ключу.
- *Аутентификация*. Метод идентификации пользователя через проверку его учетных данных.

- *Авторизация*. Метод допуска на выполнение неких действий или работы с определенными данными путем проверки ролей или групп, к которым принадлежат пользователи.

Если безопасность имеет для вас значение (а так и должно быть!), то следует нанять опытного эксперта по безопасности, а не полагаться на советы из Интернета. Очень легко совершить незаметные ошибки и оставить свои приложения и данные уязвимыми до тех пор, пока не окажется слишком поздно!

Ключи и их размеры

В алгоритмах защиты часто используются *ключи*. Они могут быть симметричными и асимметричными. Первые известны еще и как общие или секретные, поскольку один и тот же ключ применяется для шифрования и дешифрования. Вторые представляют собой пару из открытого и закрытого ключей, в которых для шифрования служит открытый ключ, а для дешифрования — только закрытый. Последнее утверждение верно для случая, когда надо передать зашифрованные данные. В случае с цифровой подписью все с точностью до наоборот: закрытым ключом шифруются данные, а открытым — дешифруются.

Алгоритмы шифрования с использованием симметричных ключей быстры и позволяют шифровать большие объемы потоковых данных. Асимметричные алгоритмы шифрования ключей медленны и позволяют шифровать только небольшие массивы байтов. В своих проектах применяйте оба способа, используя симметричное шифрование для защиты самих данных и асимметричное — для распространения симметричного ключа. По такому принципу, к примеру, работает криптографический протокол Secure Sockets Layer (SSL) в Интернете.

Ключи в шифровании представлены массивами байтов разного размера.

Больший размер ключа обозначает более сильную защиту.

Векторы инициализации (IV) и размеры блоков

Вполне вероятно, что при шифровании больших объемов данных повторяется часть их фрагментов (последовательностей символов). Например, в английском тексте часто применяется последовательность символов `the`, которая каждый раз шифруется как `hQ2`. Умный хакер воспользовался бы этим и упростил бы себе работу по взлому шифра:

```
When the wind blew hard the umbrella broke.
5:s4&hQ2aj#D f9d1d£8fh"&hQ2s0)an DF8SFd#][1
```

Избежать повторения последовательностей можно, разделив данные на блоки. После шифрования блока из него генерируется значение массива байтов, которое передается в следующий блок с целью настроить алгоритм так, чтобы последовательность `the` шифровалась иначе. Зашифровать первый блок поможет массив байтов для выполнения задачи. Это так называемый *вектор инициализации* (initialization vector, IV).

Чем меньше размер блока, тем более сильным получается шифр.

Соли

Соль представляет собой случайный массив байтов, который используется как дополнительный ввод для односторонней хеш-функции. Допустим, вы не применяете соль при генерации хешей, и многие из ваших пользователей регистрируются с указанием значения `123456` в качестве пароля (примерно 8 % из них в 2016 году так и делали!). В таком случае все они имеют одно и то же хеш-значение и их учетная запись будет уязвима для внешнего доступа через подбор пароля по словарю.

Подробнее о словарной атаке: https://blog.codinghorror.com/dictionary-attacks-101/.

Когда пользователь регистрируется, соль должна генерироваться случайным образом и конкатенироваться с указанным пользователем паролем до того, как будет хеширована. Вывод (но не исходный пароль) сохраняется с солью в базе данных.

Когда пользователь авторизуется в системе и вводит пароль, вы просматриваете соль, объединяете ее с введенным паролем, восстанавливаете хеш и затем сравниваете значение с хешем, хранящимся в базе данных. Если значения совпадают, то пароль введен верно.

Генерация ключей и векторов инициализации

Ключи и векторы инициализации представляют собой массивы байтов. Обеим сторонам, которые хотят обмениваться зашифрованными данными, необходимы значения ключей и векторов инициализации, но защищенно обмениваться массивами байтов может быть сложно.

Надежно генерировать ключи и векторы инициализации позволяет *стандарт формирования ключа на основе пароля (PBKDF2)*. Прекрасно подойдет класс `Rfc2898DeriveBytes`, который принимает пароль, соль и счетчик итераций, а затем генерирует ключи и векторы инициализации, вызывая метод `GetBytes`.

Класс `Rfc2898DeriveBytes` и его родственный класс наследуются от класса `DeriveBytes` (рис. 10.1).

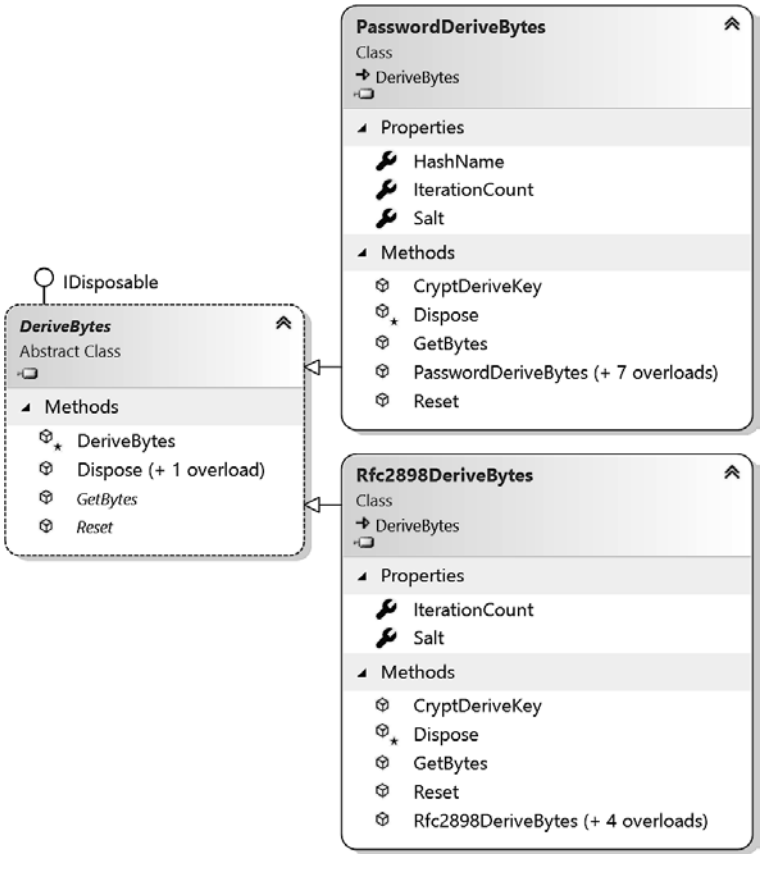


Рис. 10.1

Размер соли должен быть не меньше восьми байт, а счетчик итераций — больше нуля. Минимальное рекомендуемое количество итераций — 1000.

Шифрование и дешифрование данных

В платформе .NET Core на выбор доступно несколько алгоритмов шифрования. Одни алгоритмы реализуются операционной системой, и их имена предваряются именем класса `CryptoServiceProvider`, другие реализованы в .NET (и их имена предваряются именем класса `Managed`). Одни используют симметричные ключи, а другие — асимметричные.

Распространенные алгоритмы симметричного шифрования показаны на рис. 10.2.

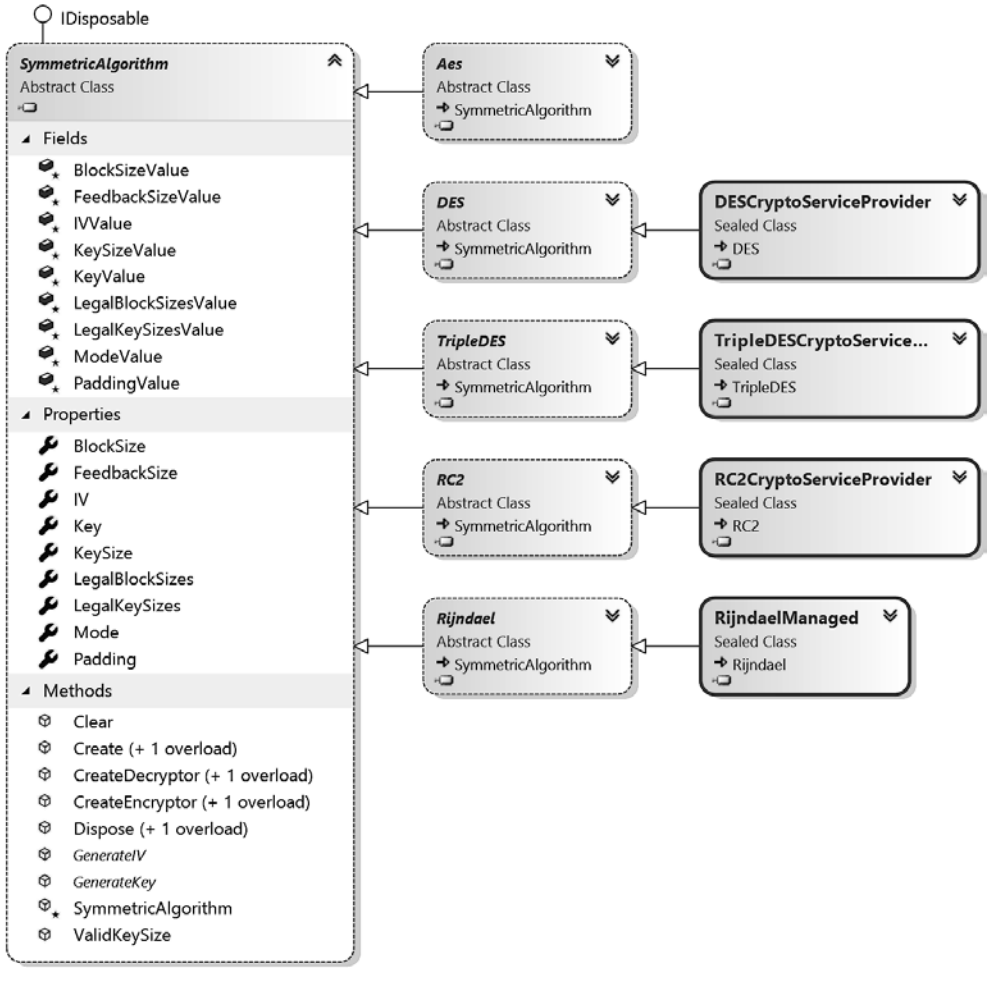


Рис. 10.2

Распространенные алгоритмы асимметричного шифрования показаны на рис. 10.3

Для симметричного шифрования используйте стандарт симметричного алгоритма блочного шифрования (AES — Advanced Encryption Standard), а для асимметричного — алгоритм RSA (Rivest — Shamir — Adleman). Не путайте алгоритм RSA с DSA, алгоритмом цифровой подписи (Digital Signature Algorithm). Последний не позволяет шифровать данные, а может лишь генерировать хеши и подписи.

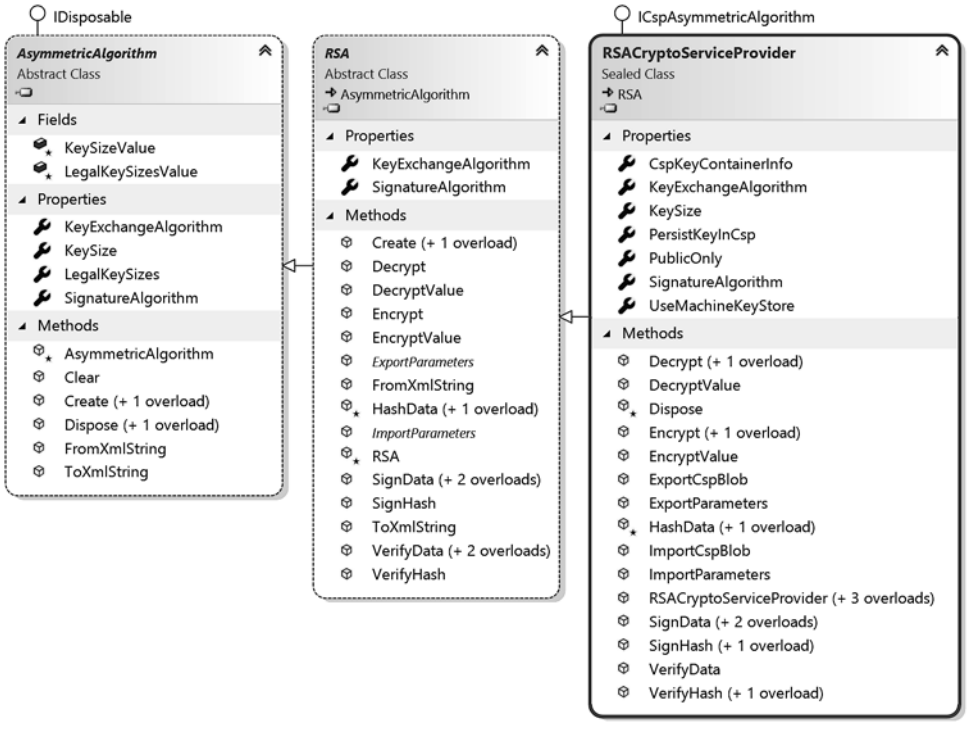


Рис. 10.3

Симметричное шифрование с помощью алгоритма AES

Чтобы упростить в будущем использование кода, обеспечивающего безопасность, мы создадим статический класс `Protector` в своей библиотеке классов.

Симметричные алгоритмы шифрования применяют класс `CryptoStream` для эффективного шифрования или дешифрования больших объемов байтов. Асимметричные алгоритмы могут обрабатывать только небольшие объемы, которые хранятся в массиве байтов, а не в потоке.

Visual Studio 2017

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Standard. В центре диалогового окна выберите пункт Class Library (.NET Standard) (Библиотека классов (.NET Standard)), присвойте проекту имя `CryptographyLib`, укажите расположение по адресу `C:\Code`, введите имя решения `Chapter10`, а затем нажмите кнопку OK. Переименуйте класс `Class1.cs` в `Protector.cs`.

В Visual Studio 2017 создайте новый проект консольного приложения с именем EncryptionApp.

На панели Solution Explorer (Обозреватель решений) в Visual Studio 2017 щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства). В категории Startup Project (Запускаемый проект) установите переключатель в положение Current selection (Текущий проект).

На панели Solution Explorer (Обозреватель решений) раскройте проект EncryptionApp, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Add Reference (Добавить ссылку). Выберите проект CryptographyLib, а затем нажмите кнопку OK.

Visual Studio Code

В операционной системе macOS откройте каталог Code и создайте в нем папку Chapter10, содержащую подкаталоги с именами CryptographyLib и EncryptionApp.

В Visual Studio Code откройте каталог CryptographyLib.

На панели Integrated Terminal (Интегрированный терминал) введите следующую команду:

```
dotnet new classlib
```

Откройте каталог EncryptionApp.

На панели Integrated Terminal (Интегрированный терминал) введите такую команду:

```
dotnet new console
```

Откройте каталог Chapter10.

На панели EXPLORER (Проводник) раскройте папку CryptographyLib и переименуйте файл Class1.cs в Protector.cs.

В папке EncryptionApp проекта найдите и откройте файл EncryptionApp.csproj. Добавьте в него ссылку на библиотеку CryptographyLib, как показано в листинге ниже (новый код выделен полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\CryptographyLib\CryptographyLib.csproj"/>
  </ItemGroup>

</Project>
```

На панели Integrated Terminal (Интегрированный терминал) введите эти команды:

```
cd EncryptionApp
dotnet build
```

Создание класса Protector

В Visual Studio 2017 или Visual Studio Code откройте файл `Protector.cs` и измените его содержимое, как показано в коде ниже:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Security.Cryptography;
using System.Text;
using System.Xml.Linq;

namespace Packt.CS7
{
  public static class Protector
  {
    // размер соли должен составлять не менее восьми байт,
    // мы будем использовать 16 байт
    private static readonly byte[] salt =
    Encoding.Unicode.GetBytes("7BANANAS");

    // число итераций должно быть не меньше 1000, мы будем использовать
    // 2000 итераций
    private static readonly int iterations = 2000;

    public static string Encrypt(string plainText,
    string password)
    {
      byte[] plainBytes = Encoding.Unicode.GetBytes(plainText);
      var aes = Aes.Create();
      var pbkdf2 = new Rfc2898DeriveBytes(password, salt,
      iterations);
      aes.Key = pbkdf2.GetBytes(32); // установить 256-битный ключ
      aes.IV = pbkdf2.GetBytes(16);  // установить 128-битный вектор
                                     // инициализации
      var ms = new MemoryStream();
      using (var cs = new CryptoStream(ms, aes.CreateEncryptor(),
      CryptoStreamMode.Write))
      {
        cs.Write(plainBytes, 0, plainBytes.Length);
      }
      return Convert.ToBase64String(ms.ToArray());
    }

    public static string Decrypt(string cryptoText,
    string password)
    {
      byte[] cryptoBytes = Convert.FromBase64String(cryptoText);
      var aes = Aes.Create();
      var pbkdf2 = new Rfc2898DeriveBytes(password, salt,
      iterations);
      aes.Key = pbkdf2.GetBytes(32);
      aes.IV = pbkdf2.GetBytes(16);
      var ms = new MemoryStream();
      using (var cs = new CryptoStream(ms, aes.CreateDecryptor(),
      CryptoStreamMode.Write))
```

```
      {
        cs.Write(cryptoBytes, 0, cryptoBytes.Length);
      }
      return Encoding.Unicode.GetString(ms.ToArray());
    }
  }
}
```

Обратите внимание на следующие моменты:

- в примере удвоены рекомендованный размер соли и количество итераций;
- хотя размер соли и количество итераций могут быть жестко закодированы, пароль *должен* передаваться во время выполнения при вызове методов `Encrypt` и `Decrypt`;
- в примере используется временный тип `MemoryStream` для хранения результатов шифрования и дешифрования, а затем вызывается метод `ToArray` для преобразования потока в массив байтов;
- в примере для конвертации зашифрованных массивов байтов применяется кодировка Base64, чтобы упростить обработку.

Пароль в исходном коде никогда не кодируйте жестко, поскольку даже после компиляции пароль в сборке можно прочитать с помощью дизассемблера.

В проекте EncryptionApp откройте файл `Program.cs` и импортируйте следующее пространство имен и тип:

```
using Packt.CS7;
using static System.Console;
```

Добавьте в метод `Main` указанные ниже инструкции, которые позволяют запросить у пользователя сообщение и пароль, а затем выполняют шифрование и дешифрование:

```
Write("Enter a message that you want to encrypt: ");
string message = ReadLine();
Write("Enter a password: ");
string password = ReadLine();
string cryptoText = Protector.Encrypt(message, password);
WriteLine($"Encrypted text: {cryptoText}");
Write("Enter the password: ");
string password2 = ReadLine();
try
{
  string clearText = Protector.Decrypt(cryptoText, password2);
  WriteLine($"Decrypted text: {clearText}");
}
catch
{
  WriteLine(
  "Enable to decrypt because you entered the wrong password!");
}
```

Запустите консольное приложение. Введите сообщение и пароль, а затем проанализируйте результат вывода.

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
Encrypted text: pV5qPDf1CCZmGzUMH2gapFSkn573lg7tMj5ajice3cQ=
Enter the password: secret
Decrypted text: Hello Bob
```

Перезапустите приложение и вновь введите сообщение и пароль, только на этот раз сделайте в пароле после шифрования намеренную ошибку. Проанализируйте результат вывода.

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
Encrypted text: pV5qPDf1CCZmGzUMH2gapFSkn573lg7tMj5ajice3cQ=
Enter the password: 123456
Enable to decrypt because you entered the wrong password!
```

Хеширование данных

На платформе .NET Core доступно несколько алгоритмов хеширования. Одни не требуют использования каких-либо ключей, другим необходимы симметричные ключи, а третьим — асимметричные. Распространенные алгоритмы хеширования показаны на рис. 10.4.

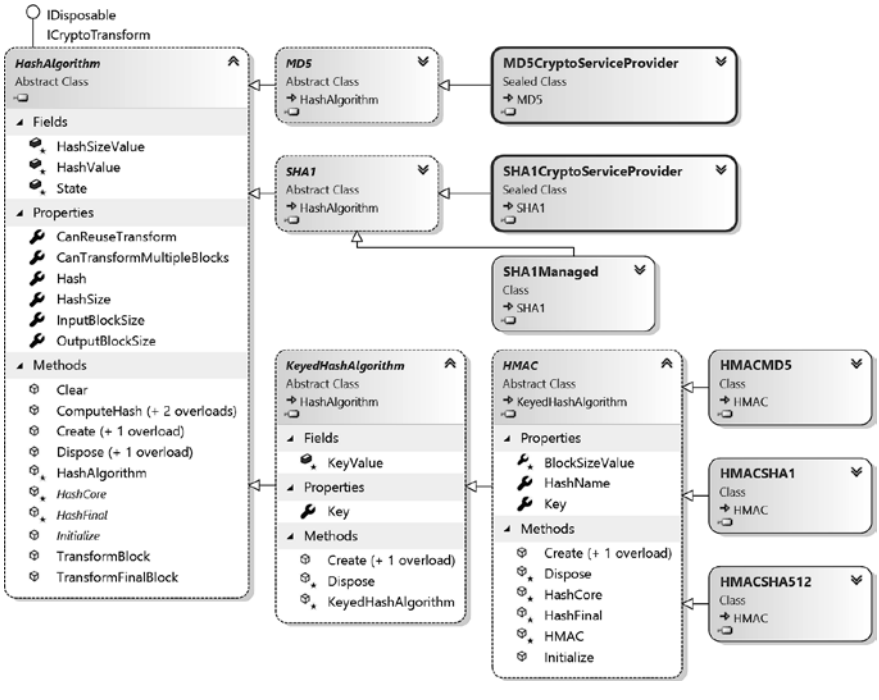


Рис. 10.4

При выборе алгоритма хеширования следует учитывать два важных фактора.

- *Устойчивость к коллизиям*: как часто два разных ввода могут иметь один и тот же хеш?
- *Устойчивость к нахождению прообраза*: в отношении хеша, насколько сложно обнаружить другой ввод, который имеет идентичный хеш?

В табл. 10.1 представлены некоторые универсальные алгоритмы хеширования.

Таблица 10.1

Алгоритм	Размер хеша	Описание
MD5	16 байт	Применяется чаще всего, поскольку быстр в работе, но не устойчив к коллизиям
SHA1	20 байт	Использование в Интернете не рекомендуется с 2011 года
SHA256 SHA384 SHA512	32 байт 48 байт 64 байт	Алгоритмы из семейства «Безопасный алгоритм хеширования второго поколения (SHA2, Secure Hashing Algorithm)» с разными размерами хешей

Старайтесь не использовать алгоритмы MD5 и SHA1, поскольку у них выявлены существенные недостатки. Выбирайте алгоритм с большим размером хеша, чтобы уменьшить вероятность повторения хешей. Первое упоминание о коллизии алгоритма SHA1 датируется 2017 годом. См. https://arstechnica.co.uk/information-technology/2017/02/atdeaths-door-for-years-widely-used-sha1-function-is-now-dead/.

Хеширование с помощью алгоритма SHA256. В проекте библиотеки классов `CryptographyLib` создайте класс `User`. Он будет представлять сведения о пользователе, хранящиеся в памяти, файле или базе данных.

```
namespace Packt.CS7
{
  public class User
  {
    public string Name { get; set; }
    public string Salt { get; set; }
    public string SaltedHashedPassword { get; set; }
  }
}
```

Добавьте код, показанный ниже, в класс `Protector`. Мы задействуем словарь для хранения в памяти информации о нескольких пользователях. Применяются два метода: для регистрации нового пользователя и проверки введенного пароля при последующей авторизации.

```
private static Dictionary<string, User> Users = new Dictionary<string, User>();

public static User Register(string username, string password)
{
  // генерация соли
  var rng = RandomNumberGenerator.Create();
  var saltBytes = new byte[16];
  rng.GetBytes(saltBytes);
```

```
  var saltText = Convert.ToBase64String(saltBytes);

  // генерация соленого и хешированного пароля
  var sha = SHA256.Create();
  var saltedPassword = password + saltText;
  var saltedhashedPassword = Convert.ToBase64String(
  sha.ComputeHash(Encoding.Unicode.GetBytes(saltedPassword)));

  var user = new User
  {
    Name = username,
    Salt = saltText,
    SaltedHashedPassword = saltedhashedPassword
  };
  Users.Add(user.Name, user);

  return user;
}

public static bool CheckPassword(string username, string password)
{
  if (!Users.ContainsKey(username))
  {
    return false;
  }
  var user = Users[username];

  // повторная генерация соленого и хешированного пароля
  var sha = SHA256.Create();
  var saltedPassword = password + user.Salt;
  var saltedhashedPassword = Convert.ToBase64String(
  sha.ComputeHash(Encoding.Unicode.GetBytes(saltedPassword)));

  return (saltedhashedPassword == user.SaltedHashedPassword);
}
```

Создайте новый проект консольного приложения с именем `HashingApp`. Добавьте в код ссылку на сборку `CryptographyLib`, как делали ранее, а затем импортируйте следующие пространство имен и тип:

```
using Packt.CS7;
using static System.Console;
```

Добавьте в метод `Main` эти инструкции для регистрации пользователя и запроса регистрации другого пользователя, а также запроса авторизации под одним из логинов и проверки пароля:

```
WriteLine("A user named Alice has been registered with Pa$$w0rd as her
password.");
var alice = Protector.Register("Alice", "Pa$$w0rd");
WriteLine($"Name: {alice.Name}");
WriteLine($"Salt: {alice.Salt}");
WriteLine($"Salted and hashed password: {alice.SaltedHashedPassword}");
WriteLine();
Write("Enter a different username to register: ");
```

```
string username = ReadLine();
Write("Enter a password to register: ");
string password = ReadLine();
var user = Protector.Register(username, password);
WriteLine($"Name: {user.Name}");
WriteLine($"Salt: {user.Salt}");
WriteLine($"Salted and hashed password: {user.SaltedHashedPassword}");

bool correctPassword = false;
while (!correctPassword)
{
  Write("Enter a username to log in: ");
  string loginUsername = ReadLine();
  Write("Enter a password to log in: ");
  string loginPassword = ReadLine();
  correctPassword = Protector.CheckPassword(loginUsername,
  loginPassword);
  if (correctPassword)
  {
    WriteLine($"Correct! {loginUsername} has been logged in.");
  }
  else
  {
    WriteLine("Invalid username or password. Try again.");
  }
}
```

Если вы работаете в Visual Studio Code и используете сразу несколько проектов, то не забывайте переходить к правильному каталогу консольного приложения с помощью команды cd HashingApp, прежде чем выполнять команду dotnet run.

Запустите консольное приложение и проанализируйте результат вывода:

```
A user named Alice has been registered with Pa$$w0rd as her password.
Name: Alice
Salt: tLn3gRn9DXmp2oeuvBSxTg==
Salted and hashed password:
  w8Ub2aH5NNQ8MJarYsUgm29bbbl0lV/9dlozjWs2Ipk=
Enter a different username to register: Bob
Enter a password to register: Pa$$w0rd
Name: Bob
Salt: zPU9YyFLaz0idhQkKpzY+g==
Salted and hashed password:
  8w14w8WNHoZddEeIx2+UJhpHQqSs4EmyoazqjbmmEz0=
Enter a username to log in: Bob
Enter a password to log in: secret
Invalid username or password. Try again.
Enter a username to log in: Alice
Enter a password to log in: secret
Invalid username or password. Try again.
Enter a username to log in: Bob
Enter a password to log in: Pa$$w0rd
Correct! Bob has been logged in.
```

Даже если оба пользователя зарегистрируются с одним и тем же паролем, соли будут генерироваться случайным образом, поэтому соленые и хешированные пароли пользователей будут различаться.

Подписывание данных

В качестве доказательства, что полученные данные на самом деле присланы доверенным отправителем, используются цифровые подписи. По сути, вы подписываете не сами данные, а только их хеш. С помощью алгоритма SHA256 мы сгенерируем хеш и подпишем его, применив алгоритм RSA.

Мы могли бы использовать алгоритм DSA для хеширования и подписывания. Он работает быстрее RSA при генерации подписи, но медленнее при ее проверке. Поскольку подпись генерируется однажды, но проверяется много раз, то преимущество состоит в более быстрой проверке, а не генерации.

Алгоритм RSA базируется на факторизации больших целых чисел по сравнению с алгоритмом DSA, который основан на вычислении дискретного логарифма. Прочитать об этом можно на сайте http://mathworld.wolfram.com/RSAEncryption.html.

Подписывание с помощью алгоритмов SHA256 и RSA

В проекте библиотеки классов `CryptographyLib` добавьте в класс `Protector` код, показанный ниже:

```
public static string PublicKey;

public static string ToXmlStringExt(this RSA rsa, bool
includePrivateParameters)
{
  var p = rsa.ExportParameters(includePrivateParameters);
  XElement xml;
  if (includePrivateParameters)
  {
    xml = new XElement("RSAKeyValue"
        , new XElement("Modulus",
        Convert.ToBase64String(p.Modulus))
        , new XElement("Exponent",
        Convert.ToBase64String(p.Exponent))
        , new XElement("P", Convert.ToBase64String(p.P))
        , new XElement("Q", Convert.ToBase64String(p.Q))
        , new XElement("DP", Convert.ToBase64String(p.DP))
        , new XElement("DQ", Convert.ToBase64String(p.DQ))
        , new XElement("InverseQ",
        Convert.ToBase64String(p.InverseQ)));
  }
  else
  {
    xml = new XElement("RSAKeyValue"
        , new XElement("Modulus",
        Convert.ToBase64String(p.Modulus))
```

```
        , new XElement("Exponent",
        Convert.ToBase64String(p.Exponent)));
  }
  return xml?.ToString();
}

public static void FromXmlStringExt(this RSA rsa,
string parametersAsXml)
{
  var xml = XDocument.Parse(parametersAsXml);
  var root = xml.Element("RSAKeyValue");
  var p = new RSAParameters
  {
    Modulus =
    Convert.FromBase64String(root.Element("Modulus").Value),
    Exponent =
    Convert.FromBase64String(root.Element("Exponent").Value)
  };
  if (root.Element("P") != null)
  {
    p.P = Convert.FromBase64String(root.Element("P").Value);
    p.Q = Convert.FromBase64String(root.Element("Q").Value);
    p.DP = Convert.FromBase64String(root.Element("DP").Value);
    p.DQ = Convert.FromBase64String(root.Element("DQ").Value);
    p.InverseQ =
    Convert.FromBase64String(root.Element("InverseQ").Value);
  }
  rsa.ImportParameters(p);
}

public static string GenerateSignature(string data)
{
  byte[] dataBytes = Encoding.Unicode.GetBytes(data);
  var sha = SHA256.Create();
  var hashedData = sha.ComputeHash(dataBytes);

  var rsa = RSA.Create();
  PublicKey = rsa.ToXmlStringExt(false); // исключение закрытого ключа

  return Convert.ToBase64String(rsa.SignHash(hashedData,
  HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1));
}

public static bool ValidateSignature(string data, string signature)
{
  byte[] dataBytes = Encoding.Unicode.GetBytes(data);
  var sha = SHA256.Create();
  var hashedData = sha.ComputeHash(dataBytes);

  byte[] signatureBytes = Convert.FromBase64String(signature);

  var rsa = RSA.Create();
  rsa.FromXmlStringExt(PublicKey);

  return rsa.VerifyHash(hashedData, signatureBytes,
  HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
}
```

Обратите внимание на следующие моменты.

- Я использовал два полезных метода класса `RSA` платформы .NET Framework: `ToXmlString` и `FromXmlString`. Они сериализуют/десериализуют структуру `RSAParameters`, которая содержит открытые и закрытые ключи. Тем не менее реализация .NET Core этих методов на macOS вызывает исключение `PlatformNotSupportedException`. Мне пришлось повторно реализовать их самостоятельно, используя типы, предоставляемые *LINQ to XML*, такие как `XDocument`, о которых вы узнаете в главе 12.
- Перед использованием кода проверки подписи должна быть доступна только открытая часть пары открытого/закрытого ключей, чтобы при вызове метода `ToXmlString` мы могли передать значение `false`. Закрытая часть требуется для подписи данных и должна храниться в секрете, поскольку любой, кто имеет к ней доступ, может подписать данные от вашего имени!
- Алгоритм хеширования, применяемый для генерации хеша из данных, должен соответствовать алгоритму хеширования, используемому при подписывании и проверке подписи. В вышеприведенном примере задействован алгоритм SHA256.

Проверка подписи и валидация

Создайте новый проект консольного приложения с именем `SigningApp`. Добавьте в код ссылку на сборку `CryptographyLib`, а затем в файле `Program.cs` импортируйте следующие пространства имен и статические типы:

```
using static System.Console;
using Packt.CS7;
```

Добавьте в метод `Main` код, приведенный ниже:

```
Write("Enter some text to sign: ");
string data = ReadLine();
var signature = Protector.GenerateSignature(data);
WriteLine($"Signature: {signature}");
WriteLine("Public key used to check signature:");
WriteLine(Protector.PublicKey);

if (Protector.ValidateSignature(data, signature))
{
  WriteLine("Correct! Signature is valid.");
}
else
{
  WriteLine("Invalid signature.");
}

// создаем поддельную подпись, заменив первый символ на X
var fakeSignature = signature.Replace(signature[0], 'X');
if (Protector.ValidateSignature(data, fakeSignature))
{
```

```
  WriteLine("Correct! Signature is valid.");
}
else
{
  WriteLine($"Invalid signature: {fakeSignature}");
}
```

Запустите консольное приложение и введите любой текст. Проанализируйте результат вывода (показан фрагмент):

```
Enter some text to sign: The cat sat on the mat.
Correct! Signature is valid.
Invalid signature:
X1uDRfCDXvOyhMtqXlxqzSljhADD/81E0UonuVs9VfZ7ceuyFWh4O7rwkdc1+l25DzGf64swtb
XZsukpSupFqvkAOIJ6XqMlD92vlG1nquereiWkshYnxxVts30QJIFKKyOTBTfN/VOljlZVMxT/
RA6pggPtESlv+urDJT4z/PEtR5jdx+CTZHQc9WiceFbpuybyf/vEdddtF0T7g8NeLKEPbT6b7CHGDM1HKbR
qnSecv456QNfHNmEXxRk9MpI0DgQLnXpOhHcVwEFc6+dY6kdNnWd6NIOY3qX6FT782t0lQ2swcWxF9fUcvW
VSeC84EgVK447X9Xewkrf6CF7jxg==
```

Генерация случайных чисел

Иногда необходимо сгенерировать случайные числа, возможно, при создании игры, симулирующей сброс игральных костей, или для дальнейшего использования с криптографией в шифровании или постановке электронной подписи.

В .NET Standard существует несколько классов, генерирующих случайные числа (рис. 10.5).

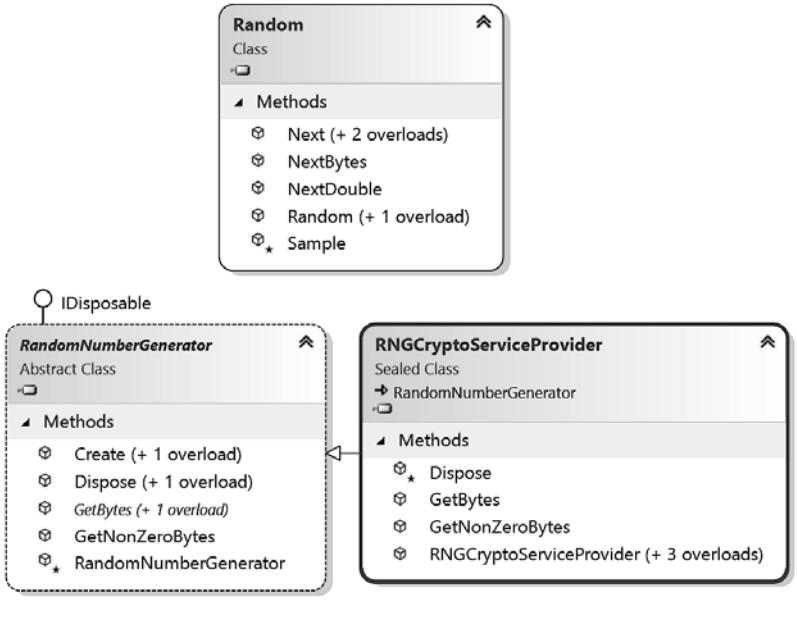


Рис. 10.5

Генерация случайных чисел для игр

В ситуациях, не требующих истинно случайных чисел, можно использовать класс `Random`, как показано в следующем примере:

```
var r = new Random();
```

В конструкторе класса `Random` есть параметр для указания начального значения, используемого для инициализации генератора псевдослучайных чисел, как показано в коде ниже:

```
var r = new Random(Seed: 12345);
```

Как вы помните из главы 2 Speaking C#, имена параметров нужно задавать с помощью так называемого верблюжьего регистра. Разработчик, определивший конструктор класса Random, нарушил это соглашение! Имя параметра следовало задать как seed, но не как Seed.

Общие начальные значения работают как секретные ключи, поэтому если вы воспользуетесь одинаковым алгоритмом генерации случайных чисел в двух приложениях, то эти приложения будут генерировать одинаковые последовательности «случайных» чисел. Иногда такая возможность полезна, например при синхронизации GPS-приемника со спутником. Но обычно нужно держать начальное значение в секрете.

Создав объект `Random`, вы можете вызвать его методы для генерации случайных чисел, как показано в следующих примерах кода:

```
int dieRoll = r.Next(minValue: 1, maxValue: 7); // возвращает числа от 1 до 6
double randomReal = r.NextDouble(); // возвращает числа от 0.0 до 1.0
var arrayOfBytes = new byte[256];
r.NextBytes(arrayOfBytes); // 256 случайных байтов в массиве
```

Метод Next принимает два параметра: minValue и maxValue. Параметр maxValue — НЕ максимальное значение, возвращаемое методом! Это эксклюзивная верхняя граница, то есть число на единицу больше максимального значения.

Генерация случайных чисел для криптографии

Класс `Random` генерирует псевдослучайные числа. Для криптографии его недостаточно! Для этих целей нужно использовать тип, наследующий от `RandomNumberGenerator`, например `RNGCryptoServiceProvider`.

В проекте библиотеки классов `CryptographyLib` добавьте инструкции в класс `Protector`, чтобы определить метод, генерирующий случайный ключ или проверку подлинности, пригодные для использования в шифровании, как показано в следующем листинге:

```
public static byte[] GetRandomKeyOrIV(int size)
```

```
{
  var r = RandomNumberGenerator.Create();
  var data = new byte[size];
  r.GetNonZeroBytes(data);    // криптографически наполняемый массив
                              // сильные случайные байты
  return data;
}
```

Тестирование генерации случайного числа или проверки подлинности

Создайте новый проект консольного приложения с именем `RandomizingApp`. Добавьте ссылку на сборку `CryptographyLib`, а затем в файле `Program.cs` импортируйте следующие пространства имен:

```
using static System.Console;
using Packt.CS7;
```

Добавьте этот код в метод `Main`:

```
static void Main(string[] args)
{
  Write("How big do you want the key (in bytes): ");
  string size = ReadLine();
  byte[] key = Protector.GetRandomKeyOrIV(int.Parse(size));
  WriteLine($"Key as byte array:");
  for (int b = 0; b < key.Length; b++)
  {
    Write($"{key[b]:x2} ");
    if (((b + 1) % 16) == 0) WriteLine();
  }
  WriteLine();
}
```

Запустите консольное приложение, введите размерность ключа, например `256`, и проанализируйте результат вывода:

```
How big do you want the key (in bytes): 256
Key as byte array:
8c 93 d8 d3 b2 0f 20 45 8e de d9 79 17 1a 78 47
ab 34 e9 e5 38 89 91 58 65 d5 fe 5f 17 18 e2 b8
a4 5c f0 48 65 60 ae f1 29 c0 c2 20 9d 1b a6 9d
17 14 aa d9 25 79 19 b4 3e bd 48 84 bc a9 a0 b4
4c 4d 7c cb 9d f6 12 15 08 a4 42 93 da 46 b6 b4
68 65 6d cc 5e 9e 92 7e 04 52 22 35 65 84 76 06
11 d1 be be 5b 1f de 8e 44 ea d4 d4 ca d4 bf b0
e6 50 6d d1 69 16 8c d1 14 68 35 43 6a ee d8 a9
63 2e 4b 54 38 ef 45 c1 8b c9 f8 f1 f8 d9 d7 80
e4 c8 d3 a3 1f 3f 24 9c 1e 97 e0 55 17 ab 6d a1
b2 2a d8 59 d6 e6 06 28 f1 5c 86 9f 5a 1f d5 01
18 d7 73 bd ae 8c a1 ef ab 18 75 ba 76 65 d0 17
71 ec 68 6f fd c3 6a 0b 23 e8 ee 65 99 b2 0a af
ff c9 09 c7 7f e9 41 a1 1b e6 b8 c7 b4 0e 91 26
26 e7 e6 d0 85 0e e5 f1 48 ca 4b f8 b8 71 19 ee
ba 3b bb 6b b8 e6 2d d9 ae b8 81 fb 71 fa 98 ae
```

Аутентификация и авторизация пользователей

Аутентификация — это процесс верификации личности пользователя путем сверки его учетных данных с данными, хранящимися в авторитетном источнике. Учетные данные включают в себя сочетание имени пользователя и пароля, или отпечатка пальца, или скана поверхности лица. После аутентификации пользователи могут создавать запросы, например узнавать, каков их адрес электронной почты, к каким группам или ролям они принадлежат.

Авторизация — процесс верификации принадлежности к группам или ролям перед предоставлением доступа к ресурсам, например к функциям приложения или данным. Авторизация может основываться на верификации личности. Однако, несмотря на это, хорошим методом обеспечения безопасности является авторизация на основе принадлежности к группе или роли: такой подход позволяет изменить принадлежность в будущем, не прибегая к переназначению прав доступа.

Механизмов аутентификации и авторизации очень много, однако все они реализуют два интерфейса из пространства имен `System.Security.Principal`: `IIdentity` и `IPrincipal`.

Интерфейс `IIdentity` представляет пользователя, так что для этого интерфейса реализованы свойства `Name` и `IsAuthenticated`, позволяющие указать, был ли аутентифицирован текущий пользователь, или он является анонимным.

Самый распространенный класс, который реализует данный интерфейс, — это `GenericIdentity`, наследующий от `ClaimsIdentity` (рис. 10.6).

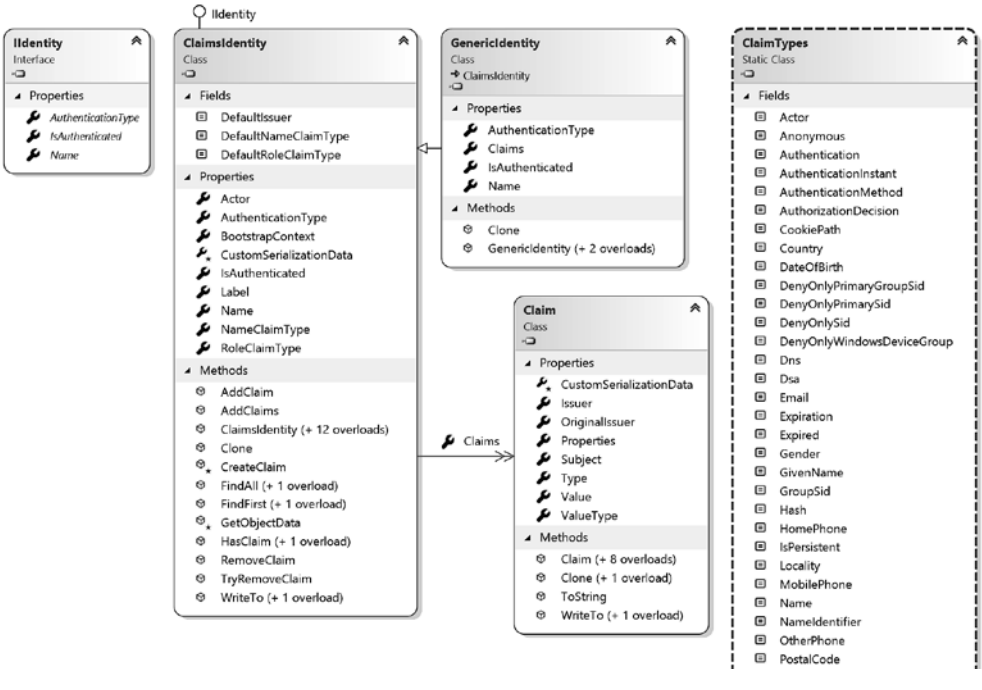


Рис. 10.6

У каждого класса `ClaimsIdentity` есть свойство `Claims`. У объектов `Claim` есть свойство `Type`, указывающее тип запроса: на имя, принадлежность к роли или группе, дату рождения и т. д.

Интерфейс `IPrincipal` используется для привязки личности к ролям и группам, к которым эта личность принадлежит, таким образом, он может применяться для авторизации. У текущего потока, выполняющего вашу программу, есть свойство `CurrentPrincipal`, которое можно установить для любого объекта, реализующего этот интерфейс. Позже данное свойство может быть проверено, когда потребуется получить разрешение на выполнение защищенного действия.

Самый распространенный класс, который реализует этот интерфейс, — это `GenericPrincipal`, наследующий от `ClaimsPrincipal` (рис. 10.7).

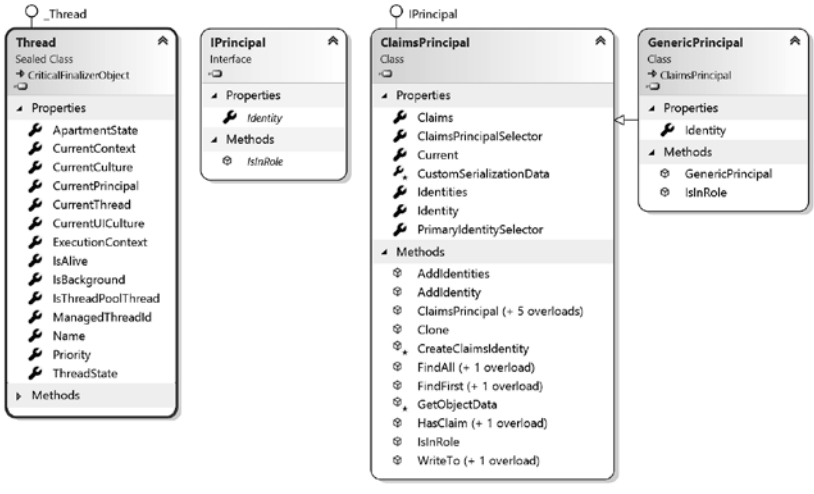


Рис. 10.7

Реализация аутентификации и авторизации

В проекте библиотеки классов `CryptographyLib` добавьте свойство в класс `User` для хранения массива ролей:

```
public string[] Roles { get; set; }
```

В проекте библиотеки классов `CryptographyLib` измените метод `Register` класса `Protector`, чтобы в данный метод можно было передавать массив ролей в качестве необязательного параметра, как показано выделением в следующем листинге:

```
public static User Register(string username, string password, string[] roles = null)
```

Измените метод `Register` класса `Protector`, чтобы установить массив ролей в объекте `User`, как показано в листинге ниже:

```
var user = new User
{
  Name = username,
```

```
  Salt = saltText,
  SaltedHashedPassword = saltedhashedPassword,
  Roles = roles
};
```

В проекте библиотеки классов `CryptographyLib` добавьте инструкции в класс `Protector` для определения метода, регистрирующего пользователей с именами Alice, Bob и Eve в различных ролях, как показано в следующем листинге:

```
public static void RegisterSomeUsers()
{
  Register("Alice", "Pa$$w0rd", new[] { "Admins" });
  Register("Bob", "Pa$$w0rd", new[] { "Sales", "TeamLeads" });
  Register("Eve", "Pa$$w0rd");
}
```

Чтобы сделать класс Protector пригодным для повторного использования с минимальными сложностями, было бы лучше определить RegisterSomeUsers в отдельном классе конкретной библиотеки.

В проекте библиотеки классов `CryptographyLib` добавьте следующий код в класс `Protector`, чтобы импортировать пространство имен `System.Security.Principal`, определите метод `LogIn` для пользовательского входа и воспользуйтесь обобщенными значениями идентификатора и его владельца, чтобы присвоить их актуальному потоку:

```
public static void LogIn(string username, string password)
{
  if (CheckPassword(username, password))
  {
    var identity = new GenericIdentity(username, "PacktAuth");
    var principal = new GenericPrincipal(identity,
    Users[username].Roles);
    System.Threading.Thread.CurrentPrincipal = principal;
  }
}
```

Тестирование аутентификации и авторизации

Создайте новый проект консольного приложения с именем `SecureApp`. Добавьте ссылку на сборку `CryptographyLib`, а затем в файле `Program.cs` импортируйте следующие пространства имен:

```
using static System.Console;
using Packt.CS7;
using System.Threading;
using System.Security;
using System.Security.Permissions;
using System.Security.Principal;
using System.Security.Claims;
```

Добавьте в метод `Main` инструкции для регистрации новых пользователей, приглашения их войти в систему, а затем вывода информации о них, как показано в следующем листинге:

```
Protector.RegisterSomeUsers();

Write($"Enter your user name: ");
string username = ReadLine();
Write($"Enter your password: ");
string password = ReadLine();

Protector.LogIn(username, password);
if (Thread.CurrentPrincipal == null)
{
  WriteLine("Log in failed.");
  return;
}

var p = Thread.CurrentPrincipal;

WriteLine($"IsAuthenticated: {p.Identity.IsAuthenticated}");
WriteLine($"AuthenticationType: {p.Identity.AuthenticationType}");
WriteLine($"Name: {p.Identity.Name}");
WriteLine($"IsInRole(\"Admins\"): {p.IsInRole("Admins")}");
WriteLine($"IsInRole(\"Sales\"): {p.IsInRole("Sales")}");

if (p is ClaimsPrincipal)
{
  WriteLine($"{p.Identity.Name} has the following claims:");
  foreach (Claim claim in (p as ClaimsPrincipal).Claims)
  {
    WriteLine($" {claim.Type}: {claim.Value}");
  }
}
```

Запустите консольное приложение, войдите в систему как `Alice` с паролем `Pa$$word` и проанализируйте результат вывода, показанный ниже:

```
Enter your user name: Alice
Enter your password: Pa$$w0rd
IsAuthenticated: True
AuthenticationType: PacktAuth
Name: Alice
IsInRole("Admins"): True
IsInRole("Sales"): False
Alice has the following claims:
  http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Alice
  http://schemas.microsoft.com/ws/2008/06/identity/claims/role: Admins
```

Запустите консольное приложение, войдите в систему как `Alice` с паролем `secret` и проанализируйте результат вывода, показанный ниже:

```
Enter your user name: Alice
Enter your password: secret
Log in failed.
```

Запустите консольное приложение, войдите в систему как Bob с паролем Pa$$word и проанализируйте результат вывода, показанный ниже:

```
Enter your user name: Bob
Enter your password: Pa$$w0rd
IsAuthenticated: True
AuthenticationType: PacktAuth
Name: Bob
IsInRole("Admins"): False
IsInRole("Sales"): True
Bob has the following claims:
  http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Bob
  http://schemas.microsoft.com/ws/2008/06/identity/claims/role: Sales
  http://schemas.microsoft.com/ws/2008/06/identity/claims/role: TeamLeads
```

Защита функций приложения

Добавьте в класс Program метод, защищенный проверкой разрешений внутри другого метода и выводящий сообщения о соответствующих исключениях при отказе в доступе, как показано в листинге ниже:

```
static void SecureFeature()
{
  if (Thread.CurrentPrincipal == null)
  {
    throw new SecurityException("Thread.CurrentPrincipal cannot be null.");
  }
  if (!Thread.CurrentPrincipal.IsInRole("Admins"))
  {
    throw new SecurityException("User must be a member of Admins
        to access this feature.");
  }

  WriteLine("You have access to this secure feature.");
}
```

Добавьте инструкции в конец метода Main для вызова метода SecureFeature в инструкции try...catch, как показано в следующем листинге:

```
try
{
  SecureFeature();
}
catch(System.Exception ex)
{
  WriteLine($"{ex.GetType()}: {ex.Message}");
}
```

Запустите консольное приложение, войдите в систему как Alice с паролем Pa$$word и проанализируйте результат вывода, показанный ниже:

```
You have access to this secure feature.
```

Запустите консольное приложение, войдите в систему как `Bob` с паролем `Pa$$word` и проанализируйте результат вывода, показанный ниже:

```
System.Security.SecurityException: User must be a member of Admins to access this
feature.
```

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

1. Какой из алгоритмов шифрования, доступных на платформе .NET, лучше всего подойдет для симметричного шифрования?
2. Какой из алгоритмов шифрования, доступных на платформе .NET, наиболее пригоден для асимметричного шифрования?
3. Что такое радужная атака?
4. При использовании алгоритмов шифрования лучше применять большой или маленький размер блока?
5. Что такое хеш?
6. Что такое подпись?
7. Чем отличаются симметричное и асимметричное шифрование?
8. Как расшифровывается RSA?
9. Почему пароли следует посолить перед сохранением?
10. SHA1 — это протокол хеширования, созданный в Агентстве национальной безопасности Соединенных Штатов. Почему вы никогда не должны его использовать?

Упражнение 10.1. Защита данных с помощью шифрования и хеширования

Создайте консольное приложение с именем `Exercise02`, предназначенное для защиты XML-файла (см. пример ниже). Обратите внимание: номер и ПИН-код (`Pa$$w0rd`) банковской карты клиента хранятся в открытом виде. Номер карты должен быть зашифрован, чтобы его можно было расшифровать и использовать позже, а пароль нужно посолить и хешировать.

```
<?xml version="1.0" encoding="utf-8" ?>
<customers>
  <customer>
```

```
    <name>Bob Smith</name>
    <creditcard>1234-5678-9012-3456</creditcard>
    <password>Pa$$w0rd</password>
  </customer>
</customers>
```

Упражнение 10.2. Дешифрование данных

Создайте консольное приложение с именем `Exercise03`, открывающее XML-файл, защитой которого вы занимались в предыдущем упражнении, и расшифровывающее номер банковской карты.

Дополнительные ресурсы

- Ключевые концепции безопасности: https://msdn.microsoft.com/en-us/library/z164t8hs(v=vs.110).aspx.
- Шифрование данных: https://msdn.microsoft.com/en-us/library/as0w18af(v=vs.110).aspx.
- Криптографические подписи: https://msdn.microsoft.com/en-us/library/hk8wx38z(v=vs.110).aspx.

Резюме

Из этой главы вы узнали, как зашифровать и расшифровывать данные, используя симметричное шифрование, как генерировать соленый хеш, подписывать данные и проверять цифровые подписи.

В следующей главе вы научитесь работать с базами данных, задействуя Entity Framework Core.

11 Работа с базами данных с помощью Entity Framework Core

Эта глава посвящена чтению и записи в такие базы данных, как Microsoft SQL Server и SQLite, с помощью технологии объектно-реляционного отображения данных *Entity Framework Core (EF Core)*.

В данной главе:

- современные базы данных;
- настройка EF Core;
- определение моделей EF Core;
- запрос данных из модели EF Core;
- управление данными с помощью EF Core.

Современные базы данных

Чаще всего данные принято хранить в *системе управления реляционными базами данных (СУРБД)*, среди них самыми популярными выступают Microsoft SQL Server, PostgreSQL, MySQL и SQLite, а также в *NoSQL*, примерами которых являются Microsoft Azure Cosmos DB, Redis, MongoDB и Apache Cassandra.

В этой главе мы рассмотрим СУРБД, такие как SQL Server и SQLite. Чтобы узнать больше о базах данных NoSQL, например, Cosmos и MongoDB, и о том, как их применять с EF Core, посетите следующие сайты.

- Добро пожаловать в базу данных Azure Cosmos DB: https://docs.microsoft.com/en-us/azure/cosmos-db/introduction.
- Использование баз данных NoSQL в EF Core: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrspatterns/nosql-database-persistence-infrastructure.

- Что такое MongoDB: https://www.mongodb.com/what-is-mongodb.
- EF Core и MongoDB: https://github.com/crhairr/EntityFrameworkCore.MongoDb.

Использование образца реляционной базы данных

Чтобы научиться управлять базой данных с помощью .NET Core, было бы полезно иметь под рукой образец средней сложности, но с достойным количеством заготовленных записей. Корпорация Microsoft предлагает несколько образцов баз данных, большинство из которых слишком сложны для наших целей, поэтому применим базу данных Northwind, впервые созданную в начале 1990 годов.

Перейдите по ссылке https://github.com/markjprice/cs7dotnetcore2/tree/master/sql-scripts/ и загрузите файл `Northwind4SQLServer.sql` для работы с Microsoft SQL Server или файл `Northwind4SQLite.sql` для использования SQLite.

На рис. 11.1 приведена схема базы данных Northwind, которую можно применять для справки, когда будем писать запросы.

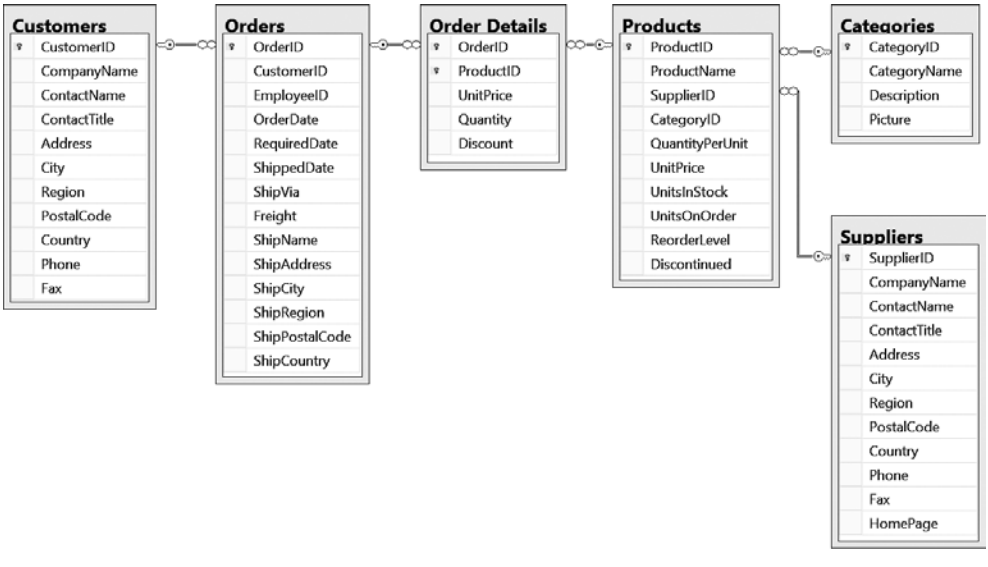


Рис. 11.1

Microsoft SQL Server

Корпорация Microsoft предлагает различные версии своего продукта SQL Server для платформ Windows, Linux и Docker, о чем вы можете узнать на сайте https://www.microsoft.com/en-us/sql-server/sql-server-2017.

Мы обратимся к бесплатной версии *SQL Server LocalDB*, которая может запускаться как отдельное приложение. Новейшая версия этой базы устанавливается в комплекте с другими инструментами Visual Studio 2017.

Подключение к Microsoft SQL Server LocalDb

При написании программного кода для подключения к базе данных SQL Server необходимо знать *имя сервера* этой базы. Оно зависит от версии, которую вы решите использовать. Ниже приведены несколько примеров.

- Visual Studio 2017 установка SQL Server 2016: `(localdb)\mssqllocaldb`.
- Visual Studio 2015 установка SQL Server 2014: `(localdb)\mssqllocaldb`.
- Visual Studio 2012/2013 установка SQL Server 2012: `(localdb)\v11.0`.
- Если устанавливаете SQL Server Express: `.\sqlexpress`.
- Если устанавливаете полную версию SQL Server: `.`.

Точка в последних двух строках подключения — сокращение имени локального компьютера. Имена серверов для SQL Server состоят из двух частей: имени компьютера и имени экземпляра SQL Server. Полная версия не требует имени экземпляра, хотя во время установки можно его указать.

Создание образца базы данных Northwind для SQL Server

Загрузите сценарий для создания базы данных Northwind для Microsoft SQL Server, перейдя по ссылке https://github.com/markjprice/cs7dotnetcore2/sql-scripts/Northwind4SQLServer.sql.

В Visual Studio 2017 выполните команду File ▸ Open ▸ File (Файл ▸ Открыть ▸ Файл) или нажмите сочетание клавиш Ctrl+O.

Найдите и выберите файл `Northwind4SQLServer.sql`, а затем нажмите кнопку Open (Открыть).

Щелкните правой кнопкой мыши на области редактора кода и выберите пункт Execute (Выполнить) либо нажмите сочетание клавиш Ctrl+Shift+E.

В открывшемся диалоговом окне в поле Server Name (Имя сервера) укажите значение `(localdb)\mssqllocaldb` (рис. 11.2) и нажмите кнопку Connect (Подключиться).

Сообщение Command(s) completed successfully (Запрос (-ы) успешно выполнен (-ы)) означает, что база данных Northwind была создана и можно подключиться к ней.

Иногда на первый запуск LocalDb уходит довольно много времени, вследствие чего вы можете увидеть ошибку истечения времени ожидания. В этом случае просто вновь нажмите кнопку Connect (Подключиться) — и все должно заработать.

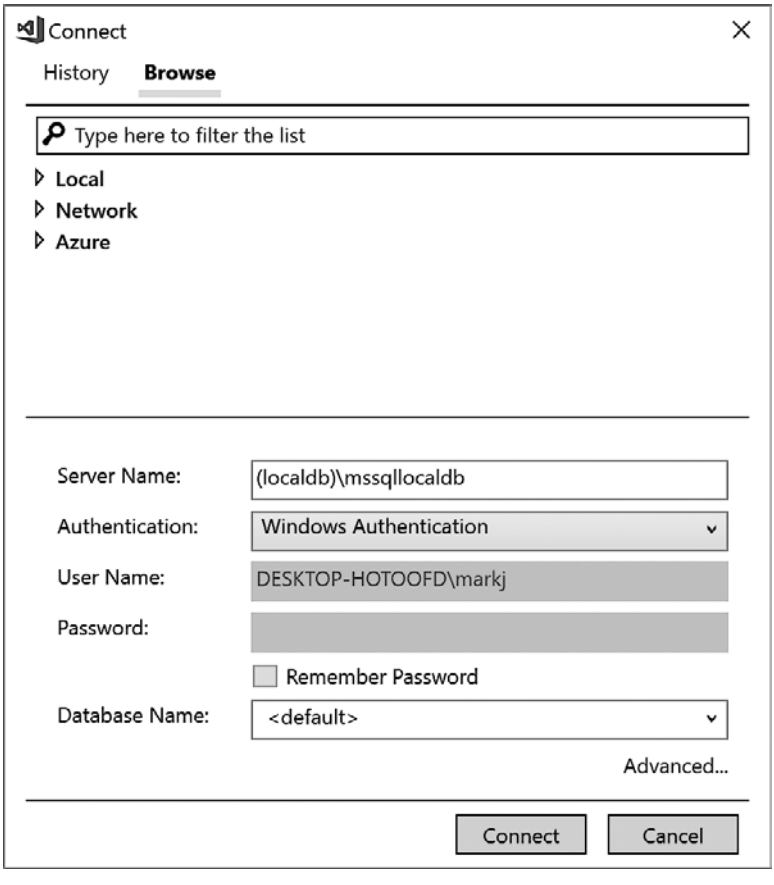


Рис. 11.2

Управление образцом базы данных Northwind в обозревателе серверов

В Visual Studio 2017 выполните команду View ▸ Server Explorer (Вид ▸ Обозреватель серверов) или нажмите сочетание клавиш Ctrl+W+L.

На панели Server Explorer (Обозреватель серверов) щелкните правой кнопкой мыши на пункте Data Connections (Подключения данных) и в контекстном меню выберите пункт Add Connection (Добавить подключение).

Если увидите диалоговое окно Choose Data Source (Выбор источника данных) (рис. 11.3), то выберите пункт Microsoft SQL Server и нажмите кнопку Continue (Продолжить).

В диалоговом окне Add Connection (Добавить подключение) в поле Server Name (Имя сервера) укажите значение `(localdb)\mssqllocaldb`, в поле Select or enter a database name (Выберите или введите имя базы данных) укажите значение `Northwind` (рис. 11.4) и нажмите кнопку OK.

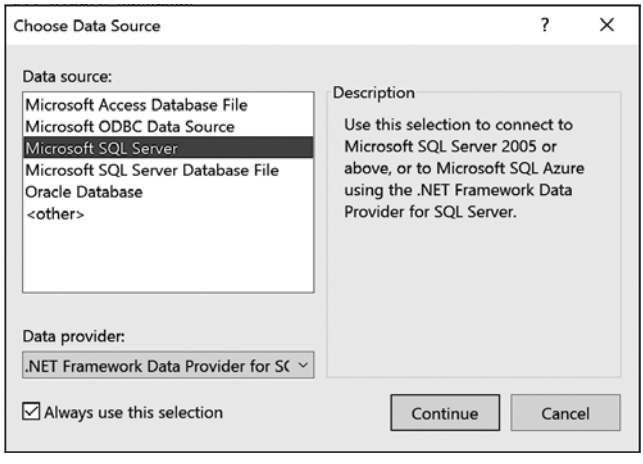


Рис. 11.3

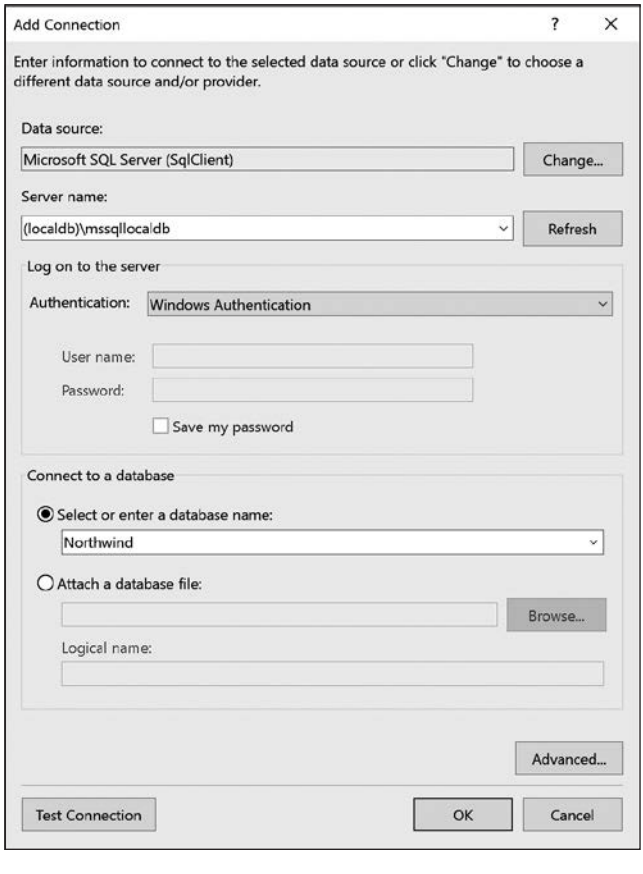


Рис. 11.4

На панели Server Explorer (Обозреватель серверов) раскройте дерево Data Connections (Подключения данных) и вложенное в него дерево Tables (Таблицы). В результате вы должны увидеть 13 таблиц, в том числе и таблицы Products и Categories (рис. 11.5).

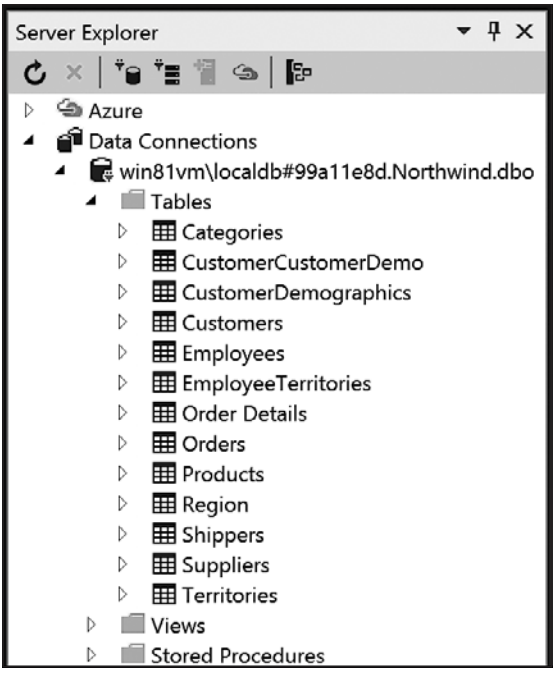
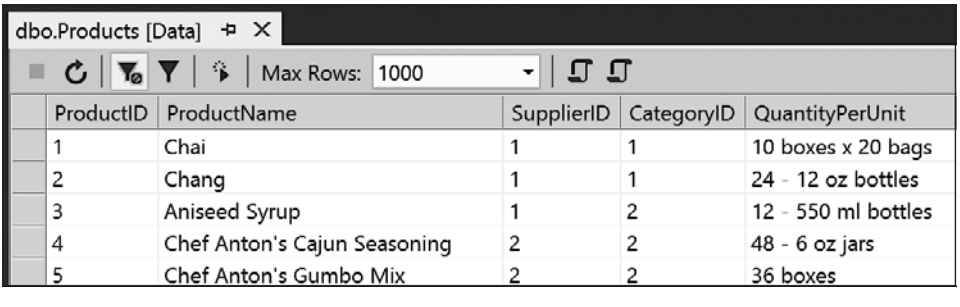


Рис. 11.5

Щелкните правой кнопкой мыши на таблице Products и выберите пункт Show Table Data (Показать таблицу данных) (рис. 11.6).



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit
1	Chai	1	1	10 boxes x 20 bags
2	Chang	1	1	24 - 12 oz bottles
3	Aniseed Syrup	1	2	12 - 550 ml bottles
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars
5	Chef Anton's Gumbo Mix	2	2	36 boxes

Рис. 11.6

Чтобы просмотреть подробную информацию о таблице Products, ее столбцах и типах, щелкните на ней правой кнопкой мыши и выберите пункт Open Table Definition (Открыть определение таблицы) (рис. 11.7).

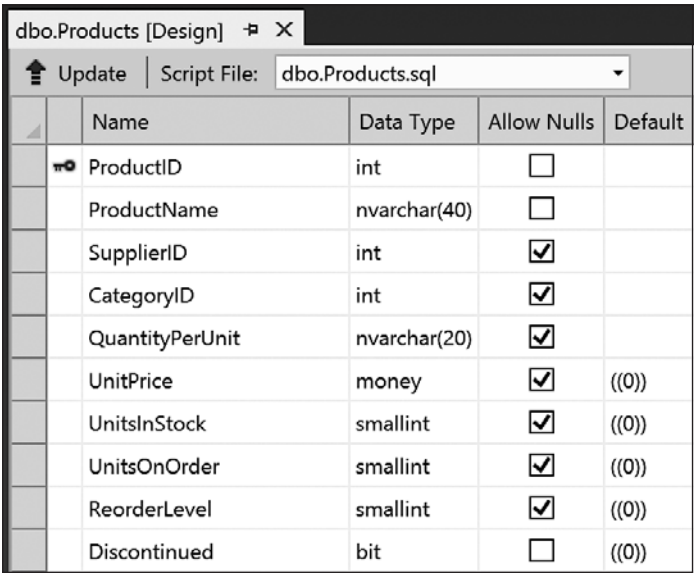


Рис. 11.7

SQLite

SQLite — небольшая кросс-платформенная самодостаточная СУРБД, доступная в публичном домене. Это самая распространенная база данных для мобильных платформ, таких как iOS (iPhone и iPad) и Android.

SQLite включена в каталог `/usr/bin/` операционной системы macOS в виде приложения командной строки с именем sqlite3.

Если вы используете Visual Studio Code в операционной системе Windows, то должны установить SQLite самостоятельно или задействовать SQL Server. Приведенные инструкции касаются Visual Studio Code для macOS.

Загрузить графический менеджер баз данных SQLiteStudio для SQLite можно, перейдя по ссылке http://sqlitestudio.pl.

Прочитать об инструкциях SQL, поддерживаемых SQLite, вы можете на сайте https://sqlite.org/lang.html.

Создание образца базы данных Northwind для SQLite

Создайте каталог `Chapter11` с подкаталогом `WorkingWithEFCore`.

Загрузите сценарий для создания базы данных `Northwind` для SQLite, перейдя по ссылке https://github.com/markjprice/cs7dotnetcore2/blob/master/sql-scripts/Northwind4SQLite.sql.

Скопируйте файл `Northwind4SQLite.sql` в каталог `Chapter11`.

Запустите приложение Terminal (Терминал) или используйте панель Integrated Terminal (Интегрированный терминал) в Visual Studio Code.

Введите команды для смены рабочего каталога сначала на каталог `Code`, потом на каталог `Chapter11`, а затем — на каталог `WorkingWithEFCore`, после чего запустите сценарий SQLite для создания базы данных `Northwind.db`, как показано ниже:

```
cd Code/Chapter11/WorkingWithEFCore
sqlite3 Northwind.db < Northwind4SQLite.sql
```

Управление образцом базы данных Northwind в SQLiteStudio

Запустите SQLiteStudio или другой GUI-инструмент для управления базами данных SQLite.

Если появится предупреждение о невозможности запуска приложения, нажмите и удерживайте клавишу Shift во время его запуска, а затем нажмите кнопку Open (Открыть).

В SQLiteStudio в меню Database (База данных) выберите пункт Add a database (Добавить базу данных) или воспользуйтесь сочетанием клавиш Cmd+O.

В диалоговом окне Database (База данных) щелкните на значке папки, чтобы перейти к существующему файлу базы данных на локальном компьютере. Выберите файл `Northwind.db`. Щелкните на команде Test connection (Проверить подключение), чтобы увидеть зеленый флажок (рис. 11.8), а затем нажмите кнопку OK.

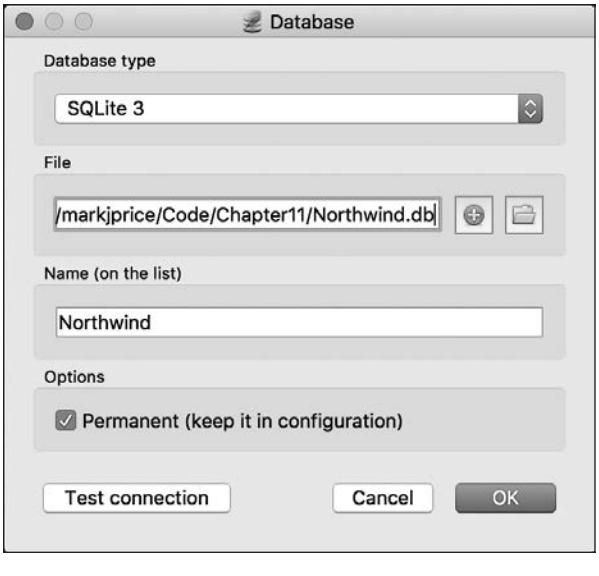


Рис. 11.8

Щелкните правой кнопкой мыши на базе данных Northwind и в контекстном меню выберите пункт Connect to the database (Подключиться к базе данных) (рис. 11.9).

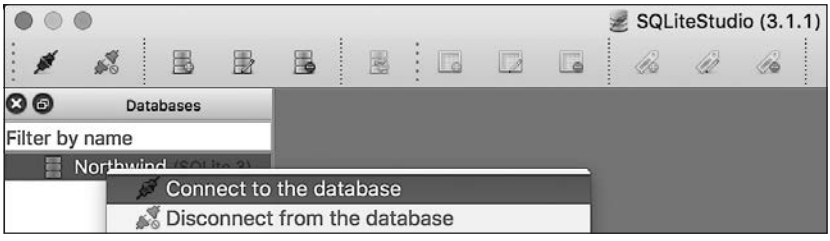


Рис. 11.9

Вы увидите десять таблиц, созданных сценарием (рис. 11.10).

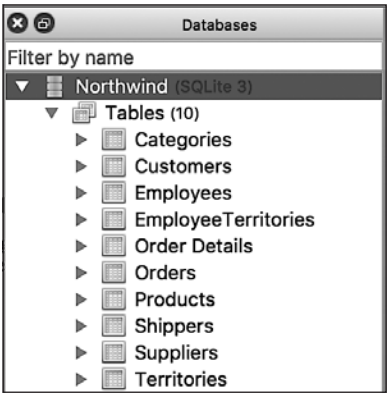


Рис. 11.10

Щелкните правой кнопкой мыши на таблице Products и в контекстном меню выберите пункт Edit the table (Редактировать таблицу) (рис. 11.11).

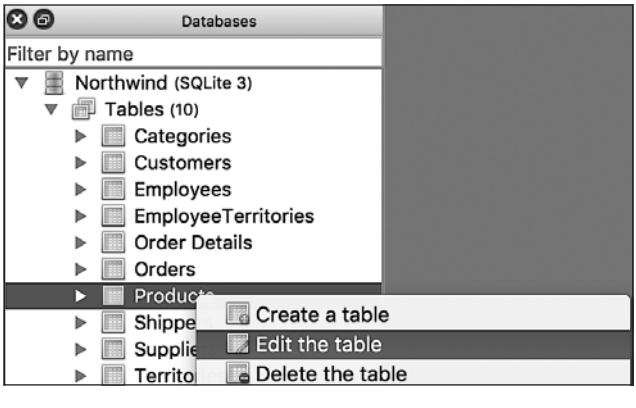
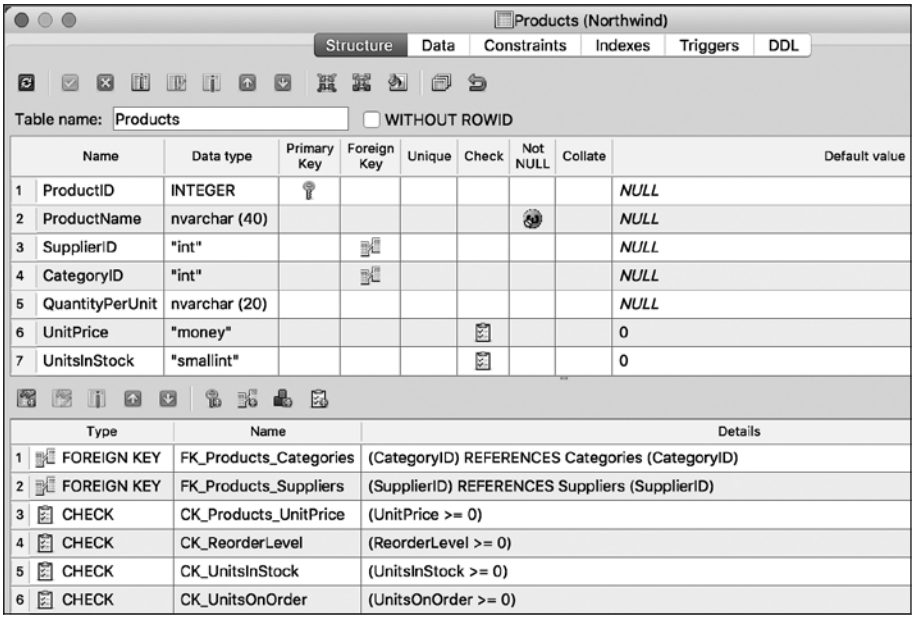


Рис. 11.11

В окне редактора таблицы вы увидите ее структуру, в том числе имена столбцов, типы данных, ключи и ограничения (рис. 11.12).

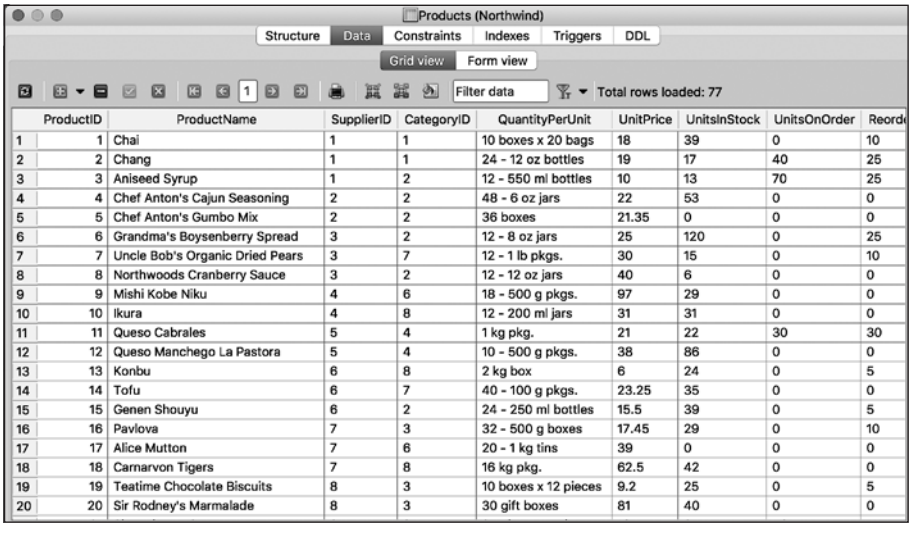


	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default value
1	ProductID	INTEGER							NULL
2	ProductName	nvarchar (40)							NULL
3	SupplierID	"int"							NULL
4	CategoryID	"int"							NULL
5	QuantityPerUnit	nvarchar (20)							NULL
6	UnitPrice	"money"							0
7	UnitsInStock	"smallint"							0

	Type	Name	Details
1	FOREIGN KEY	FK_Products_Categories	(CategoryID) REFERENCES Categories (CategoryID)
2	FOREIGN KEY	FK_Products_Suppliers	(SupplierID) REFERENCES Suppliers (SupplierID)
3	CHECK	CK_Products_UnitPrice	(UnitPrice >= 0)
4	CHECK	CK_ReorderLevel	(ReorderLevel >= 0)
5	CHECK	CK_UnitsInStock	(UnitsInStock >= 0)
6	CHECK	CK_UnitsOnOrder	(UnitsOnOrder >= 0)

Рис. 11.12

В окне редактора таблицы перейдите на вкладку Data (Данные). Вы увидите 77 строк с наименованиями товаров (рис. 11.13) (показаны первые 20 строк).



	ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	Reord
1	1	Chai	1	1	10 boxes x 20 bags	18	39	0	10
2	2	Chang	1	1	24 - 12 oz bottles	19	17	40	25
3	3	Aniseed Syrup	1	2	12 - 550 ml bottles	10	13	70	25
4	4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22	53	0	0
5	5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35	0	0	0
6	6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25	120	0	25
7	7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30	15	0	10
8	8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40	6	0	0
9	9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97	29	0	0
10	10	Ikura	4	8	12 - 200 ml jars	31	31	0	0
11	11	Queso Cabrales	5	4	1 kg pkg.	21	22	30	30
12	12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38	86	0	0
13	13	Konbu	6	8	2 kg box	6	24	0	5
14	14	Tofu	6	7	40 - 100 g pkgs.	23.25	35	0	0
15	15	Genen Shouyu	6	2	24 - 250 ml bottles	15.5	39	0	5
16	16	Pavlova	7	3	32 - 500 g boxes	17.45	29	0	10
17	17	Alice Mutton	7	6	20 - 1 kg tins	39	0	0	0
18	18	Carnarvon Tigers	7	8	16 kg pkg.	62.5	42	0	0
19	19	Teatime Chocolate Biscuits	8	3	10 boxes x 12 pieces	9.2	25	0	5
20	20	Sir Rodney's Marmalade	8	3	30 gift boxes	81	40	0	0

Рис. 11.13

Настройка Entity Framework Core

Впервые инструмент *Entity Framework (EF)* был выпущен в составе *.NET Framework 3.5 с Пакетом обновления 1* еще в конце 2008 года. С тех пор данный продукт эволюционировал по мере того, как корпорация Microsoft наблюдала за тем, как программисты используют инструменты *объектно-реляционного отображения данных* (object-relational mapping, ORM) в приложениях реального мира.

Принцип ORM основан на сопоставлении столбцов таблиц со свойствами классов. Таким образом, программист может взаимодействовать с объектами разных типов знакомым ему образом, вместо того чтобы разбираться с правилами сохранения значений в структуре таблицы.

Версия, входящая в .NET Framework, — *Entity Framework 6 (EF6)*. Она зрелая, стабильная и поддерживает старый метод определения модели во время проектирования EDMX, а также сложные модели наследования и другие дополнительные возможности. Однако EF6 поддерживается лишь .NET Framework, но не .NET Core.

Кросс-платформенная версия, *Entity Framework Core (EF Core)*, отличается от вышеописанных. Корпорация Microsoft назвала ее так, чтобы акцентировать внимание на том, что версия имеет новую функциональность. Несмотря на схожесть имен продуктов, EF Core отличается от EF6.

Взглянем на достоинства и недостатки EF Core.

- Достоинства:
  - доступна как для .NET Core, так и для .NET Framework, а это значит, что может использоваться кросс-платформенно в Linux, macOS, а также в Windows;
  - поддерживает современные облачные нереляционные бессхемные хранилища данных, такие как Microsoft Azure Table Storage и Redis.
- Недостатки:
  - никогда не будет поддерживать XML-формат файла времени проектирования EDMX;
  - (пока что) не поддерживает ленивую загрузку данных или модели сложного наследования и другие продвинутые функции EF6.

Выбор поставщика данных .NET

Прежде чем погружаться в практические вопросы управления данными в СУРБД, немного поговорим о выборе *поставщиков данных .NET*.

Хотя EF Core 2.0 и является частью .NET Standard 2.0, для управления данными нужны классы, которые знают, как эффективно *«общаться»* с базой. Поставщики данных .NET — это наборы классов, оптимизированные для работы с конкретной СУРБД. Они поставляются в пакетах NuGet.

Поставщики баз данных SQL Server, SQLite и in-memory включены в метапакет ASP.NET Core 2.0. Если вы разрабатываете веб-приложения или веб-сервисы с помощью ASP.NET Core, то не придется устанавливать пакеты, указанные в табл. 11.1. Поскольку мы создаем консольное приложение, то должны установить их вручную.

Таблица 11.1

Управляемая СУРБД	Требуемый NuGet-пакет
Microsoft SQL Server 2008 или новее	Microsoft.EntityFrameworkCore.SqlServer
SQLite 3.7 или новее	Microsoft.EntityFrameworkCore.SQLite
MySQL	MySQL.EntityFrameworkCore
In-memory (для модульного тестирования)	Microsoft.EntityFrameworkCore.InMemory

Devart — это независимый производитель, предлагающий поставщики EF Core для широкого спектра баз данных. Более подробную информацию можно получить, перейдя по ссылке https://www.devart.com/dotconnect/entityframework.html.

Подключение к базе данных

Для подключения к SQLite понадобится только имя файла базы.

Чтобы подключиться к Microsoft SQL Server, необходимо знать следующее:

- имя сервера, на котором запущена требуемая СУРБД;
- имя базы данных;
- данные для обеспечения безопасности, такие как имя пользователя и пароль, а также сведения о том, следует ли передавать учетные данные текущего пользователя автоматически.

Эта информация указывается в строке подключения. Для обеспечения обратной совместимости можно использовать ряд ключевых слов. Ниже приведено несколько примеров.

- `Data Source`, или `server`, или `addr`: это имя сервера (и дополнительно имя экземпляра сервера на данной машине).
- `Initial Catalog` или `database`: это имя базы данных.
- `Integrated Security` или `trusted_connection`: этому ключевому слову присваивают значение `true` или SSPI, чтобы передать учетные данные текущего пользователя в потоке, из которого производится подключение.

Visual Studio 2017

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Core. В центральной части диалогового окна выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)), присвойте ему имя `WorkingWithEFCore`, укажите расположение по адресу `C:\Code`, введите имя решения `Chapter11`, а затем нажмите кнопку OK.

На панели Solution Explorer (Обозреватель решений) раскройте проект WorkingWithEFCore, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Manage NuGet Packages (Управление пакетами NuGet).

На открывшейся панели перейдите на вкладку Browse (Обзор), выполните поиск пакета Microsoft.EntityFrameworkCore.SqlServer, выберите его и нажмите кнопку Install (Установить) (рис. 11.14).

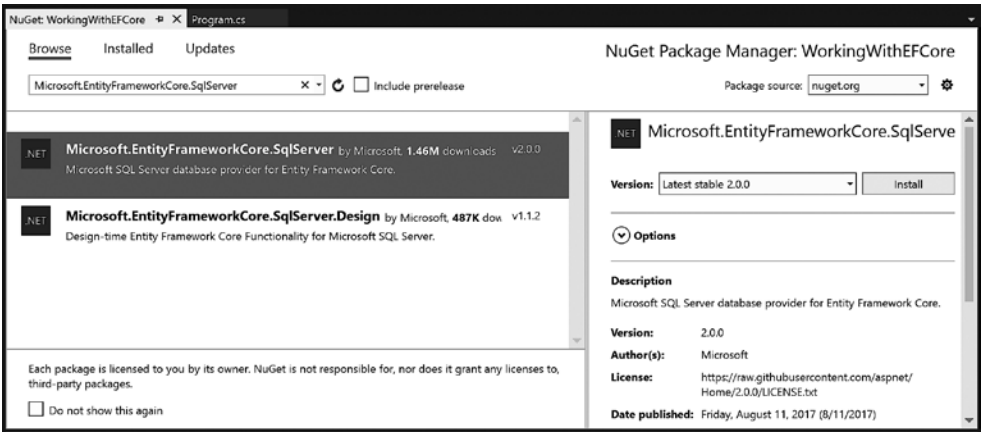


Рис. 11.14

В диалоговом окне, показанном на рис. 11.15, просмотрите изменения. Примите условия лицензионного соглашения.

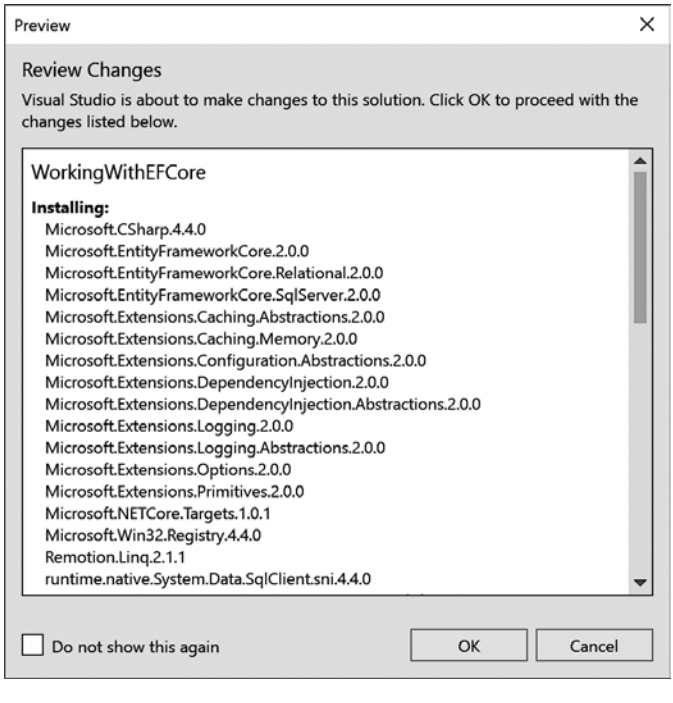


Рис. 11.15

Visual Studio Code

Запустите Visual Studio Code и откройте в ней каталог `WorkingWithEFCore`, созданный ранее.

На панели Integrated Terminal (Интегрированный терминал) выполните команду `dotnet new console`.

На панели EXPLORER (Проводник) щелкните на файле `EFCore.csproj`.

Добавьте ссылку на пакет EF Core для SQLite, как показано в листинге ниже (изменения выделены полужирным шрифтом).

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReferenceInclude=
    "Microsoft.EntityFrameworkCore.Sqlite" Version="2.0.0" />
  </ItemGroup>

</Project>
```

Информацию о новой версии Microsoft.EntityFrameworkCore.Sqlite можно найти на сайте https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Sqlite/.

Определение моделей Entity Framework Core

В EF Core для создания модели во время выполнения используется такое сочетание *соглашений*, *атрибутов аннотации* и инструкций *Fluent API*, что любое действие, произведенное над классами, позже можно транслировать в действие, совершаемое над действительной базой данных.

Соглашения EF Core

В программном коде, который мы напишем, будут использоваться такие соглашения:

- имя таблицы совпадает с именем свойства `DbSet<T>` в классе `DbContext`, например `Products`;
- имена столбцов совпадают с именами свойств в классе, например `ProductID`;
- строка .NET имеет тип `nvarchar` в базе данных;
- тип .NET `int` является типом `int` в базе данных;

- свойство `ID` либо имя класса с суффиксом `ID` выступает в качестве первичного ключа. Если это свойство имеет любой целочисленный тип или тип `Guid`, то также предполагается, что оно является `IDENTITY` (значение, присваиваемое автоматически при вводе).

Существует большое количество других соглашений. Вы можете определить и собственные, однако эта тема выходит за рамки данной книги. Более подробную информацию можно получить, перейдя по ссылке https://docs.microsoft.com/en-us/ef/core/modeling.

Атрибуты аннотации EF Core

Зачастую для сопоставления всех классов объектов базы данных соглашений недостаточно. Простой способ сделать модель более изящной — это применить атрибуты аннотации.

Например, в базе данных максимальная длина названия товара равна `40` и данное значение не может быть пустым (`null`). Для указания этих ограничений в классе `Product` можно использовать следующие атрибуты:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

Атрибуты могут применяться при отсутствии очевидного соответствия между типами .NET и типами базы данных. Например, в базе тип столбца UnitPrice таблицы Product — `money`. В .NET нет типа `money`, так что вместо него нужно воспользоваться типом `decimal`:

```
[Column(TypeName = "money")]
public decimal? UnitPrice { get; set; }
```

В таблице Category текст в столбце Description может иметь длину 8000 символов и больше. Такой текст не может быть сохранен в переменной типа `nvarchar`, поэтому тип данного столбца должен быть соотнесен с типом `ntext`:

```
[Column(TypeName = "ntext")]
public string Description { get; set; }
```

Существует еще много других атрибутов, однако их рассмотрение выходит за рамки данной книги.

EF Core Fluent API

И последний способ, с помощью которого можно определить модель, — это *Fluent API*. Данный способ может использоваться вместо атрибутов или вдобавок к ним. Например, взглянем на такие два атрибута в классе `Product`:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

Эти атрибуты можно удалить или заменить с помощью следующей инструкции Fluent API в методе `OnModelBuilding` класса `Northwind`:

```
modelBuilder.Entity<Product>()
  .Property(product => product.ProductName)
  .IsRequired()
  .HasMaxLength(40);
```

Создание модели EF Core

В Visual Studio 2017 или Visual Studio Code добавьте в проект три файла класса с именами `Northwind.cs`, `Category.cs` и `Product.cs`. Чтобы классы можно было использовать многократно, мы определим их в пространстве имен Packt.CS7.

Эти три класса будут ссылаться друг на друга, поэтому во избежание путаницы сначала создайте три класса без каких-либо членов, как показано в следующих трех файлах классов.

Создайте файл класса `Category.cs`, как показано в данном коде:

```
namespace Packt.CS7
{
  public class Category
  {
  }
}
```

Создайте файл класса `Product.cs`, как показано ниже:

```
namespace Packt.CS7
{
  public class Product
  {
  }
}
```

Создайте файл класса `Northwind.cs`, как показано ниже:

```
namespace Packt.CS7
{
  public class Northwind
  {
  }
}
```

Определение сущностного класса Category

Класс `Category` будет использоваться для представления строки в таблице Category, состоящей из четырех столбцов (рис. 11.16).

Определить четыре свойства, первичный ключ, отношение «один-ко-многим» в случае с таблицей Products поможет следование соглашениям. Однако, чтобы соотнести столбец Description с корректным типом базы данных, потребуется сопроводить свойство `string` атрибутом `Column`.

Позже интерфейс Fluent API позволит определить, что полю `CategoryName` не может быть присвоено значение `null`, а также что присваиваемое значение не должно содержать более 40 символов.

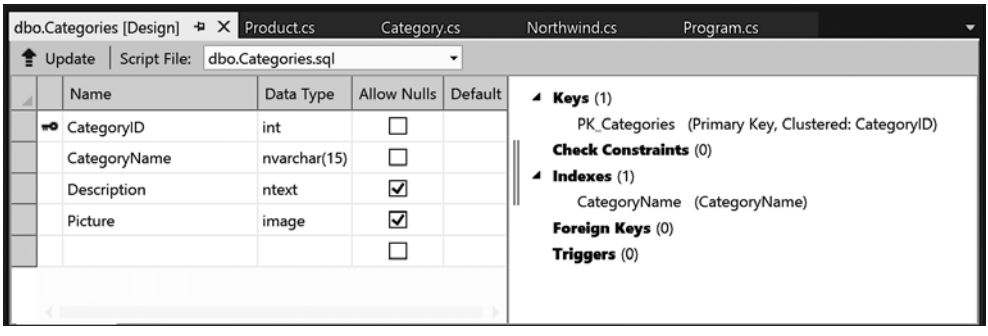


Рис. 11.16

Нам не нужно включать все столбцы таблицы в перечень свойств класса, и еще мы не будем соотносить столбец `Picture`.

Измените файл класса `Category.cs`, как показано в следующем листинге:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace Packt.CS7
{
  public class Category
  {
    // эти свойства соотносятся со столбцами в БД
    public int CategoryID { get; set; }

    public string CategoryName { get; set; }

     [Column(TypeName = "ntext")]
    public string Description { get; set; }

    // определяет свойство navigation для связанных строк
    public virtual ICollection<Product> Products { get; set; }

    public Category()
    {
      // чтобы позволить разработчикам добавлять товары в Category,
      // мы должны инициализировать свойства navigation пустым списком
      this.Products = new List<Product>();
    }
  }
}
```

Определение сущностного класса Product

Класс `Product` будет использоваться для представления строки в таблице Products, состоящей из десяти столбцов (рис. 11.17).

Мы соотнесем только шесть свойств: `ProductID`, `ProductName`, `UnitPrice`, `UnitsInStock`, `Discontinued` и `CategoryID`.

Свойство `CategoryID` связано со свойством `Category`, с помощью которого мы соотнесем каждый товар с его родительской категорией.

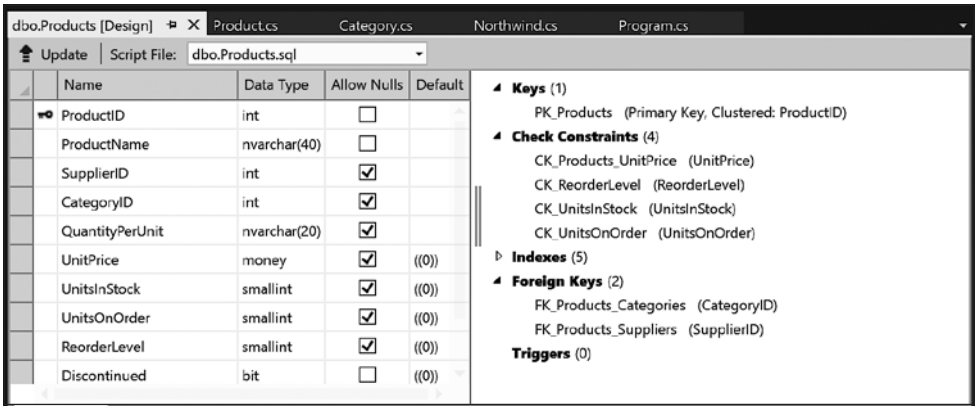


Рис. 11.17

Измените файл класса `Product.cs`, как показано в следующем листинге:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Packt.CS7
{
  public class Product
  {
    public int ProductID { get; set; }

    [Required]
    [StringLength(40)]
    public string ProductName { get; set; }

    [Column("UnitPrice", TypeName = "money")]
    public decimal? Cost { get; set; }

    [Column("UnitsInStock")]
    public short? Stock { get; set; }

    public bool Discontinued { get; set; }

    // эти два свойства определяют отношение
    // вторичного ключа к таблице Categories
    public int CategoryID { get; set; }
    public virtual Category Category { get; set; }
  }
}
```

Два свойства, применяемые для соотнесения двух сущностей, Category. Products и Product.Category, помечены как virtual. Это позволяет EF Core унаследовать и переопределить свойства, чтобы предоставить дополнительные функции, такие как «ленивая» загрузка. Данный тип загрузки не был реализован в .NET Core 2.0 и более ранних версиях. Надеемся, он будет реализован в .NET Core 2.1 или более поздних версиях.

Определение класса контекста базы данных Northwind

Для представления базы данных будет использоваться БД Northwind. Чтобы применить технологию EF Core, класс должен наследовать от `DbContext`. Этот класс понимает, как обмениваться сообщениями с базами данных и динамически генерировать инструкции SQL для запроса и управления данными.

В вашем классе, наследующем от `DbContext`, вы должны определить хотя бы одно свойство типа `DbSet<T>`. Эти свойства представляют таблицы. Чтобы сообщить EF Core, какие столбцы содержит каждая из них, тип `DbSet` использует универсальные типы для указания класса, представляющего строку таблицы, свойство этого класса представляют столбцы таблицы.

Ваш класс, наследующий от `DbContext`, должен содержать переопределенный метод `OnConfiguring`. Этот метод установит строку подключения к базе данных. Для использования SQL Server нужно раскомментировать вызов метода `UseSqlServer`, для применения SQLite — вызов метода `UseSqlite`.

В ваш класс, наследующий от `DbContext`, дополнительно может входить переопределенный метод `OnModelCreating`. В нем можно писать инструкции Fluent API как альтернативу сопровождения ваших сущностных классов атрибутами.

Измените код класса `Northwind.cs`, как показано ниже:

```
using Microsoft.EntityFrameworkCore;

namespace Packt.CS7
{
  // управление соединением с базой данных
  public class Northwind : DbContext
  {
    // свойства, сопоставляемые с таблицами в базе данных
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
      // для Microsoft SQL Server
      // optionsBuilder.UseSqlServer(
      //   @"Data Source=(localdb)\mssqllocaldb;" +
      //   "Initial Catalog=Northwind;" +
      //   "Integrated Security=true;" +
      //   "MultipleActiveResultSets=true;");

      // для SQLite
      // string path = System.IO.Path.Combine(
      //   System.Environment.CurrentDirectory, "Northwind.db");
      // optionsBuilder.UseSqlite($"Filename={path}");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      // пример использования Fluent API вместо атрибутов для ограничения
      // имени категории 40 символами
```

```
      modelBuilder.Entity<Category>()
        .Property(category => category.CategoryName)
        .IsRequired()
        .HasMaxLength(40);
    }
  }
}
```

Прежде чем продолжить, убедитесь, что раскомментировали подходящий метод для использования строки подключения либо к SQL Server, либо к SQLite.

Запрос данных из модели EF Core

Откройте файл `Program.cs` и импортируйте следующие пространства имен и статические типы:

```
using static System.Console;
using Packt.CS7;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

В классе `Program` определите метод `QueryingCategories` и добавьте инструкции для действий, указанных ниже:

- создание экземпляра класса `Northwind`, который будет управлять базой данных;
- создание запроса для всех категорий, включающих связанные товары;
- перечисление всех категорий с выводом названия и количества товаров в каждой из них.

```
static void QueryingCategories()
{
  using (var db = new Northwind())
  {
    WriteLine("Categories and how many products they have:");

    // запрос для всех категорий, включающих связанные товары
    IQueryable<Category> cats = db.Categories.Include(c => c.Products);

    foreach (Category c in cats)
    {
      WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
    }
  }
}
```

LINQ более подробно вы изучите в главе 12. А сейчас просто напишите код и проанализируйте результат.

В методе `Main` вызовите метод `QueryingCategories`, как показано в листинге ниже:

```
static void Main(string[] args)
{
  QueryingCategories();
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Categories and how many products they have:
Beverages has 12 products.
Condiments has 12 products.
Confections has 13 products.
Dairy Products has 10 products.
Grains/Cereals has 7 products.
Meat/Poultry has 6 products.
Produce has 5 products.
Seafood has 12 products.
```

В классе `Program` определите метод `QueryingProducts` и добавьте инструкции для следующих действий:

- создание экземпляра класса `Northwind`, который будет управлять базой данных;
- уведомление пользователя о стоимости товаров;
- создание с помощью LINQ запроса на товары, стоимость которых выше указанной цены;
- циклическая обработка результатов, вывод идентификатора, имени, стоимости (в долларах США) и количества единиц на складе:

```
static void QueryingProducts()
{
  using (var db = new Northwind())
  {
    WriteLine("Products that cost more than a price, and sorted.");
    string input;
    decimal price;
    do
    {
      Write("Enter a product price: ");
      input = ReadLine();
    } while(!decimal.TryParse(input, out price));

    IQueryable<Product> prods = db.Products
      .Where(product => product.Cost > price)
      .OrderByDescending(product => product.Cost);

    foreach (Product item in prods)
    {
      WriteLine($"{item.ProductID}: {item.ProductName}
      costs {item.Cost:$#,##0.00} and
      has {item.Stock} units in stock.");
    }
  }
}
```

В методе `Main` закомментируйте предыдущий вызов и вызовите метод, как показано в листинге ниже:

```
static void Main(string[] args)
{
  //QueryingCategories();
  QueryingProducts();
}
```

Запустите консольную программу и введите число `50`, когда она попросит указать стоимость товара:

```
Products that cost more than a price, and sorted.
Enter a product price: 50
38: Côte de Blaye costs $263.50 and has 17 units in stock.
29: Thüringer Rostbratwurst costs $123.79 and has 0 units in stock.
9: Mishi Kobe Niku costs $97.00 and has 29 units in stock.
20: Sir Rodney's Marmalade costs $81.00 and has 40 units in stock.
18: Carnarvon Tigers costs $62.50 and has 42 units in stock.
59: Raclette Courdavault costs $55.00 and has 79 units in stock.
51: Manjimup Dried Apples costs $53.00 and has 20 units in stock.
```

В консоли, поставляемой корпорацией Microsoft в версиях Windows, предшествующих Windows 10 Fall Creators Update, есть ограничения. По умолчанию оболочка командной строки этих версий ОС не может отображать символы Unicode. К сожалению, UTF-8 — гражданин второго сорта в старых версиях Windows. Можно временно изменить кодировку (набор символов) в консоли на Unicode UTF-8, введя следующую команду перед запуском приложения:

```
chcp65001
```

Логирование EF Core

Чтобы отслеживать взаимодействие между EF Core и базой данных, можно включить ведение журнала. Для этого потребуется:

- регистрация *поставщика логгеров* (logging provider);
- реализация *логгера*.

В Visual Studio 2017 или Visual Studio Code в свой проект добавьте класс `ConsoleLogger.cs`.

Измените код в файле, чтобы определить два класса: один для реализации интерфейса `ILoggerProvider` и второй — для реализации интерфейса `ILogger`, как показано далее в листинге, и обратите внимание на следующее.

- Класс `ConsoleLoggerProvider` возвращает экземпляр `ConsoleLogger`. Классу не требуются неуправляемые ресурсы, поэтому метод `Dispose` не выполняет никаких действий, но должен присутствовать.
- Класс `ConsoleLogger` отключен для уровней ведения журнала `None`, `Trace` и `Information`, но включен для всех остальных уровней.
- В классе `ConsoleLogger` реализован метод `Log`, выполняющий запись в `Console`:

```
using Microsoft.Extensions.Logging;
using System;
using static System.Console;

namespace Packt.CS7
{
  public class ConsoleLoggerProvider : ILoggerProvider
  {
    public ILogger CreateLogger(string categoryName)
    {
      return new ConsoleLogger();
    }

    // если логгер использует неуправляемые ресурсы,
    // то здесь можно освободить память
    public void Dispose() { }
  }

  public class ConsoleLogger : ILogger
  {
    // если логгер применяет неуправляемые ресурсы, то здесь
    // можно вернуть класс, реализующий IDisposable
    public IDisposable BeginScope<TState>(TState state)
    {
      return null;
    }

    public bool IsEnabled(LogLevel logLevel)
    {
      // во избежание ведения лишних журналов можно отфильтровать
      // по уровню логгирования
      switch (logLevel)
      {
        case LogLevel.Trace:
        case LogLevel.Information:
        case LogLevel.None:
          return false;
        case LogLevel.Debug:
        case LogLevel.Warning:
        case LogLevel.Error:
        case LogLevel.Critical:
        default:
          return true;
      };
    }

    public void Log<TState>(LogLevel logLevel,EventId eventId,
    TState state, Exception exception, Func<TState, Exception, string>
    formatter)
    {
      // занести в журнал уровень логгирования и идентификатор события
      Write($"Level: {logLevel}, Event ID: {eventId}");

      // вывод только состояния или исключения при наличии
      if (state != null)
      {
```

```
        Write($", State: {state}");
      }
      if (exception != null)
      {
        Write($", Exception: {exception.Message}");
      }
      WriteLine();
    }
  }
}
```

Добавьте в начало файла `Program.cs` следующие инструкции для импорта пространств имен:

```
using System;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
```

Для методов `QueryingCategories` и `QueryingProducts` добавьте эти инструкции в самом начале блока `using` для контекста базы данных `Northwind`:

```
using (var db = new Northwind())
{
  var loggerFactory = db.GetService<ILoggerFactory>();
  loggerFactory.AddProvider(new ConsoleLoggerProvider());
```

Запустите консольное приложение и проанализируйте результат вывода, который частично показан на рис. 11.18.

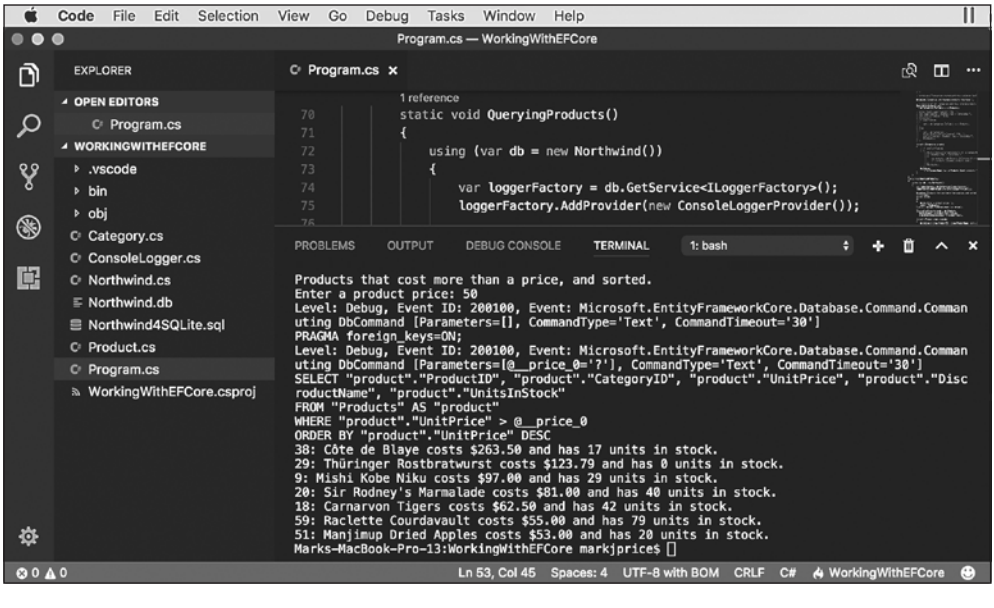


Рис. 11.18

Значения идентификаторов событий и их смысл будет отличаться в зависимости от поставщика данных .NET. Если нужно узнать, как запрос LINQ был транслирован в инструкции SQL, то значение свойства `Id` идентификатора события для вывода — `200100`, а имя — `Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting`.

Измените метод `Log` в классе `ConsoleLogger` для вывода только событий с идентификатором `200100`, как показано в листинге ниже (выделено полужирным шрифтом):

```
public void Log<TState>(LogLevel logLevel,EventId eventId,
TState state, Exception exception,Func<TState, Exception, string>
formatter)
{
  if (eventId.Id == 200100)
  {
    // занести в журнал уровень логгирования и идентификатор события
    Write($"Level: {logLevel}, Event ID: {eventId.Id},
    Event: {eventId.Name}");

    // вывод только состояния или исключения при наличии
    if (state != null)
    {
      Write($", State: {state}");
    }
    if (exception != null)
    {
      Write($", Exception: {exception.Message}");
    }
    WriteLine();
  }
}
```

В методе `Main` раскомментируйте метод `QueryingCategories`, чтобы можно было понаблюдать за генерируемыми инструкциями SQL.

Генерируемые инструкции SQL — одно из улучшений EF Core 2.0, они более эффективны и удобны по сравнению с EF Core 1.0 и 1.1.

Запустите консольное приложение и обратите внимание на следующие инструкции SQL, занесенные в журнал, как показано в этом отредактированном выводе:

```
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description"
FROM "Categories" AS "c"
ORDER BY "c"."CategoryID"

SELECT "c.Products"."ProductID", "c.Products"."CategoryID",
"c.Products"."UnitPrice", "c.Products"."Discontinued", "c.Products"."ProductName",
"c.Products"."UnitsInStock"
FROM "Products" AS "c.Products"
INNER JOIN (
  SELECT "c0"."CategoryID"
```

```
  FROM "Categories" AS "c0"
) AS "t" ON "c.Products"."CategoryID" = "t"."CategoryID"
ORDER BY "t"."CategoryID"

SELECT "product"."ProductID", "product"."CategoryID", "product"."UnitPrice",
"product"."Discontinued", "product"."ProductName", "product"."UnitsInStock"
FROM "Products" AS "product"
WHERE "product"."UnitPrice" > @__price_0
ORDER BY "product"."UnitPrice" DESC
```

Соотнесение шаблонов с помощью Like

Одна из новых функций EF Core 2.0 — поддержка инструкций SQL, в том числе `Like` для соотнесения шаблонов.

Добавьте в `Program` метод `QueryingWithLike`, как показано в следующем листинге:

```
static void QueryingWithLike()
{
  using (var db = new Northwind())
  {
    var loggerFactory = db.GetService<ILoggerFactory>();
    loggerFactory.AddProvider(new ConsoleLoggerProvider());

    Write("Enter part of a product name: ");
    string input = ReadLine();

    IQueryable<Product> prods = db.Products
    .Where(p => EF.Functions.Like(p.ProductName, $"%{input}%"));

    foreach (Product item in prods)
    {
      WriteLine($"{item.ProductName} has {item.Stock}
      units in stock. Discontinued? {item.Discontinued}");
    }
  }
}
```

В методе `Main` закомментируйте существующие методы и вызовите метод `QueryingWithLike`, как показано в листинге ниже:

```
static void Main(string[] args)
{
  // QueryingCategories();
  // QueryingProducts();
  QueryingWithLike();
}
```

Запустите консольное приложение, введите часть наименования товара, например `che`, и проанализируйте результат вывода:

```
Enter part of a product name: che
Level: Debug, Event ID: 200100, Event:
Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State:
```

```
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 200100, Event:
Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State:
Executing DbCommand [Parameters=[@__Format_1='?'], CommandType='Text',
CommandTimeout='30']
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE "p"."ProductName" LIKE @__Format_1
Chef Anton's Cajun Seasoning has 53 units in stock. Discontinued? False
Chef Anton's Gumbo Mix has 0 units in stock. Discontinued? True
Queso Manchego La Pastora has 86 units in stock. Discontinued? False
Gumbär Gummibärchen has 15 units in stock. Discontinued? False
```

Определение глобальных фильтров

Еще одно нововведение в EF Core 2.0 — глобальные фильтры.

Товары `Northwind` могут быть сняты с продажи, поэтому будет полезно удостовериться в том, что они не станут возвращаться в результатах, даже если программист забудет использовать фильтр `Where` для отсеивания таких товаров, как мы сделали в предыдущем примере (позиция Chef Anton's Gumbo Mix больше не доступна).

Измените метод `OnModelCreating` в классе `Northwind`, чтобы добавить глобальный фильтр и удалить снятые с продажи товары, как показано в следующем листинге:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  // пример использования Fluent API вместо атрибутов
  // для ограничения названия категории до менее 40
  modelBuilder.Entity<Category>()
    .Property(category => category.CategoryName)
    .IsRequired()
    .HasMaxLength(40);

  // глобальный фильтр для удаления снятых с продажи товаров
  modelBuilder.Entity<Product>().HasQueryFilter(p => !p.Discontinued);
}
```

Запустите консольное приложение, введите часть наименования товара, например `che`, проанализируйте результат вывода и обратите внимание, что позиция Chef Anton's Gumbo Mix больше не выводится, так как сгенерированная инструкция SQL уже включает в себя фильтр для колонки `Discontinued`:

```
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND "p"."ProductName" LIKE @__Format_1
Chef Anton's Cajun Seasoning has 53 units in stock. Discontinued? False
Queso Manchego La Pastora has 86 units in stock. Discontinued? False
Gumbär Gummibärchen has 15 units in stock. Discontinued? False
```

Схемы загрузки данных в EF Core

Entity Framework предоставляет три наиболее популярные *схемы загрузки данных*: *«ленивую»*, *«жадную»* и *явную загрузку*, но не все из них поддерживаются EF Core.

Ленивая и жадная загрузки элементов

В методе `QueryingCategories` программного кода в настоящий момент свойство `Categories` служит для циклического прохода по каждой категории и вывода имени категории и количества товаров в ней. Этот код работает только потому, что при написании запроса мы применили метод `Include`, позволяющий воспользоваться «жадной» загрузкой (то есть *ранней загрузкой*) релевантных товаров.

Измените запрос и закомментируйте вызов метода `Include`, как показано ниже:

```
IQueryable<Categories> cats = db.Categories; //.Include(c => c.Products);
```

В методе `Main` закомментируйте все методы, кроме `QueryingCategories`, как показано в следующем коде:

```
static void Main(string[] args)
{
  QueryingCategories();
  // QueryingProducts();
  // QueryingWithLike();
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Beverages has 0 products.
Condiments has 0 products.
Confections has 0 products.
Dairy Products has 0 products.
Grains/Cereals has 0 products.
Meat/Poultry has 0 products.
Produce has 0 products.
Seafood has 0 products.
```

Каждый элемент в цикле `foreach` — экземпляр класса `Category`, у которого есть свойство `Products`, представляющее, в свою очередь, список товаров в данной категории. Так как исходный запрос делает выборку только из таблицы Categories, значение этого свойства является пустым для каждой категории.

Когда «ленивая» загрузка будет наконец реализована в EF Core, при каждом циклическом перечислении и попытке считывания свойства `Products` система станет автоматически проверять, загружены ли элементы. Если нет, то EF Core «лениво» загрузит их для нас, выполнив инструкцию `SELECT` для подгрузки только набора товаров выбранной категории, после чего корректное количество товаров будет возвращено в программный вывод.

Проблема «ленивой» загрузки заключается в необходимости «путешествовать» к серверу базы данных и обратно с целью постепенно выбрать все данные (так на-

зываемый round trip). Именно поэтому реализация данной технологии до сих пор не была приоритетной задачей для команды разработчиков EF Core.

Явная загрузка элементов

Этот тип загрузки работает так же, как «ленивая» загрузка, с тем исключением, что вы контролируете, какие именно релевантные данные будут загружены и когда. Вы можете представить этот тип загрузки как выполняющийся автоматически для всех связанных объектов.

Измените код определения запроса, чтобы он выглядел следующим образом (так пользователь сможет выбрать обычную или явную загрузку):

```
IQueryable<Category> cats;
// = db.Categories;//.Include(c => c.Products);

Write("Enable eager loading? (Y/N): ");
bool eagerloading = (ReadKey().Key == ConsoleKey.Y);
bool explicitloading = false;
WriteLine();
if (eagerloading)
{
  cats = db.Categories.Include(c => c.Products);
}
else
{
  cats = db.Categories;
  Write("Enable explicit loading? (Y/N): ");
  explicitloading = (ReadKey().Key == ConsoleKey.Y);
  WriteLine();
}
```

Добавьте в цикл `foreach` перед вызовом метода `WriteLine` инструкции, проверяющие, разрешена ли явная загрузка, и в случае положительного ответа предлагающие пользователю явно загрузить каждую отдельную категорию, как показано в листинге ниже:

```
if (explicitloading)
{
  Write($"Explicitly load products for {c.CategoryName}? (Y/N):");
  if (ReadKey().Key == ConsoleKey.Y)
  {
    var products = db.Entry(c).Collection(c2 => c2.Products);
    if (!products.IsLoaded) products.Load();
  }
  WriteLine();
}
```

Запустите консольное приложение, отключите «жадную» загрузку (нажав клавишу `N`) и включите явную (нажав клавишу `Y`).

По желанию для каждой категории нажмите клавишу `Y` или `N` для загрузки товаров, содержащихся в данной категории. В следующем примере выходных

данных я решил загрузить товары только для двух из восьми категорий — Beverages и Seafood:

```
Categories and how many products they have:
Enable eager loading? (Y/N): n
Enable explicit loading? (Y/N): y
Level: Debug, Event ID: 200100, Event:
Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State:
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 200100, Event:
Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State:
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description"
FROM "Categories" AS "c"
Explicitly load products for Beverages? (Y/N): y
Level: Debug, Event ID: 200100, Event:
Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State:
Executing DbCommand [Parameters=[@__get_Item_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT "e"."ProductID", "e"."CategoryID", "e"."UnitPrice", "e"."Discontinued",
"e"."ProductName", "e"."UnitsInStock"
FROM "Products" AS "e"
WHERE "e"."CategoryID" = @__get_Item_0

Beverages has 12 products.
Explicitly load products for Condiments? (Y/N):n
Condiments has 0 products.
Explicitly load products for Confections? (Y/N):n
Confections has 0 products.
Explicitly load products for Dairy Products? (Y/N):n
Dairy Products has 0 products.
Explicitly load products for Grains/Cereals? (Y/N):n
Grains/Cereals has 0 products.
Explicitly load products for Meat/Poultry? (Y/N):n
Meat/Poultry has 0 products.
Explicitly load products for Produce? (Y/N):n
Produce has 0 products.
Explicitly load products for Seafood? (Y/N):yLevel: Debug, Event ID:
200100, Event:
Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State:
Executing DbCommand [Parameters=[@__get_Item_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT "e"."ProductID", "e"."CategoryID", "e"."UnitPrice", "e"."Discontinued",
"e"."ProductName", "e"."UnitsInStock"
FROM "Products" AS "e"
WHERE "e"."CategoryID" = @__get_Item_0

Seafood has 12 products.
```

Тщательно выбирайте схему загрузки данных, подходящую именно для вашего кода. В будущем использование по умолчанию «ленивой» загрузки сделает вас ленивым разработчиком баз данных!

Управление данными с помощью EF Core

EF Core позволяет очень легко добавить, обновить и удалить элементы.

Добавление элементов

В классе `Program` создайте новый метод `AddProduct`, как показано в листинге ниже:

```
static bool AddProduct(int categoryID, string productName, decimal? price)
{
  using (var db = new Northwind())
  {
    var newProduct = new Product
    {
      CategoryID = categoryID,
      ProductName = productName,
      Cost = price
    };

    //пометить товар как отслеживаемый на предмет изменений
    db.Products.Add(newProduct);
    // сохранить отслеживаемые изменения в базе данных
    int affected = db.SaveChanges();
    return (affected == 1);
  }
}
```

В классе `Program` создайте новый метод `ListProducts`, как показано в листинге ниже:

```
static void ListProducts()
{
  using (var db = new Northwind())
  {
    WriteLine("-------------------------------------------------------
    ---------------");
    WriteLine("| ID | Product Name | Cost | Stock | Disc. |");
    WriteLine("-------------------------------------------------------
    ---------------");
    foreach (var item in db.Products.OrderByDescending(p => p.Cost))
    {
      WriteLine($"| {item.ProductID:000} |
                  {item.ProductName,-35} |
                  {item.Cost,8:$#,##0.00} |
                  {item.Stock, 5} |
                  {item.Discontinued} |");
    }
    WriteLine("-------------------------------------------------------
    ---------------");
  }
}
```

В методе `Main` закомментируйте предыдущие вызовы методов, а затем вызовите методы `AddProduct` и `ListProducts`, как показано в листинге ниже:

```
static void Main(string[] args)
{
  // QueryingCategories();
  // QueryingProducts();
  // QueryingWithLike();

  AddProduct(6, "Bob's Burgers", 500M);
  ListProducts();
}
```

Запустите приложение и проанализируйте результат вывода. Обратите внимание: товар был добавлен (рис. 11.19).

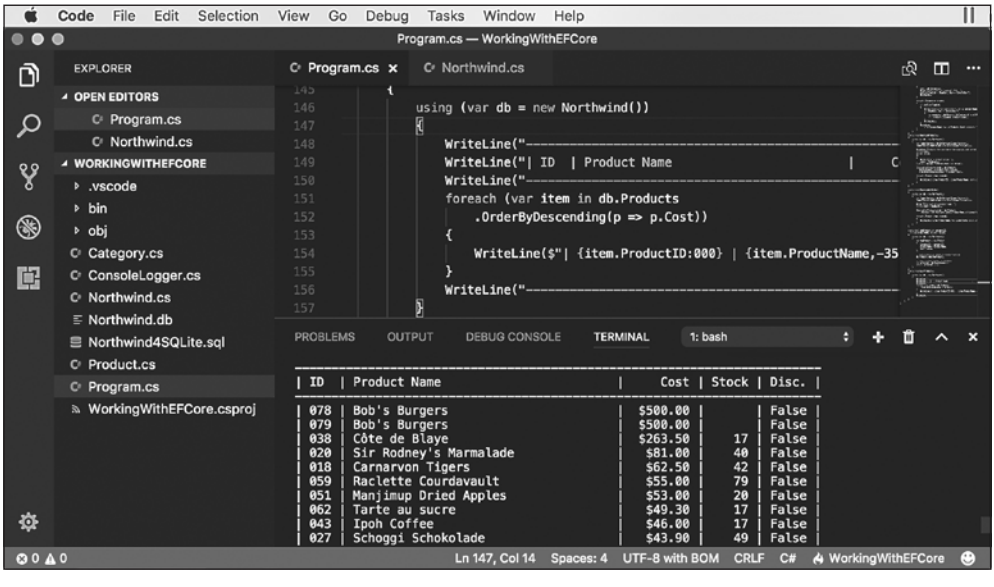


Рис. 11.19

Обновление элементов

Добавьте в класс `Program` метод, чтобы на 20 долларов увеличить цену первого товара, название которого начинается со слова `Bob`, как показано в листинге ниже:

```
static bool IncreaseProductPrice(string name, decimal amount)
{
  using (var db = new Northwind())
  {
    Product updateProduct = db.Products.First(
    p => p.ProductName.StartsWith(name));
```

```
    updateProduct.Cost += amount;
    int affected = db.SaveChanges();
    return (affected == 1);
  }
}
```

В методе `Main` закомментируйте вызов `AddProduct`, а затем вызовите метод `IncreaseProductPrice`, как показано в листинге ниже:

```
static void Main(string[] args)
{
  // QueryingCategories();
  // QueryingProducts();
  // QueryingWithLike();

  // AddProduct(6, "Bob's Burgers", 500M);

  IncreaseProductPrice("Bob", 20M);
  ListProducts();
}
```

Запустите консольное приложение, проанализируйте результат вывода и обратите внимание на то, что стоимость уже существующего товара Bob's Burgers увеличилась на 20 долларов.

```
-------------------------------------------------------
| ID  | Product Name                          |    Cost |
-------------------------------------------------------
| 078 | Bob's Burgers                         | $520.00 |
```

Удаление элементов

В класс `Program` импортируйте `System.Collections.Generic`, а затем добавьте метод для удаления всех продуктов с именем, которое начинается с символов `Bob`, как показано в листинге ниже:

```
static int DeleteProducts(string name)
{
  using (var db = new Northwind())
  {
    IEnumerable<Product> products = db.Products.Where(
    p => p.ProductName.StartsWith(name));
    db.Products.RemoveRange(products);
    int affected = db.SaveChanges();
    return affected;
  }
}
```

С помощью метода Remove можно удалить отдельные элементы. Метод RemoveRange эффективнее, если нужно удалить несколько элементов.

В методе `Main` закомментируйте вызов `AddProduct` и добавьте вызов `DeleteProducts`, как показано в листинге ниже:

```
static void Main(string[] args)
{
  // QueryingCategories();
  // QueryingProducts();
  // QueryingWithLike();

  // AddProduct(6, "Bob's Burgers", 500M);
  // IncreaseProductPrice("Bob", 20M);

  int deleted = DeleteProducts("Bob");
  WriteLine($"{deleted} product(s) were deleted.");
  ListProducts();
}
```

Запустите консольное приложение, проанализируйте результат вывода и обратите внимание на то, что продукты, имена которых начинаются с символов `Bob`, были удалены:

```
2 product(s) were deleted.
-----------------------------------------------------------------
| ID | Product Name                |    Cost |   Stock | Disc. |
-----------------------------------------------------------------
| 038 | Côte de Blaye              | $263.50 |      17 | False |
```

Группировка контекстов базы данных

Особенность платформы ASP.NET Core 2.0, связанная с EF Core 2.0, — оптимизация кода путем группировки в пул контекстов баз данных при создании веб-приложений и сервисов. Данная особенность позволяет создавать и освобождать любое количество объектов, наследующих от `DbContext`, и при этом быть уверенными в том, что код остается удобным и эффективным.

Транзакции

Каждый раз при вызове метода `SaveChanges` система запускает *неявную транзакцию*, таким образом, если что-то пойдет не так, система автоматически отменит все внесенные изменения. В случае успешного окончания операции транзакция считается совершенной.

Транзакции позволяют сохранить целостность базы данных с помощью блокировки чтения и записи до момента завершения последовательности операций.

В англоязычной литературе для характеристики транзакций принято использовать аббревиатуру ACID.

- *A* (atomic — «атомарный, неделимый»): либо совершаются все операции текущей транзакции, либо не выполняется ни одна из них.

- *C* (consistent — «согласованный»): база данных до и после совершения транзакции не содержит ошибок. Это зависит от логики программного кода.
- *I* (isolated — «изолированный»): во время выполнения транзакции вносимые изменения не видны другим процессам. Существует несколько уровней изолированности, которые можно установить (табл. 11.2). Чем выше уровень, тем выше целостность данных. Однако для такой установки требуется ввести множество блокировок, что негативно отразится на работе других процессов. Снимки состояния — особый случай, так как представляют собой копии строк таблицы, позволяющие избежать блокировок, однако это увеличивает размер базы данных при выполнении транзакций.
- *D* (durable — «устойчивый»): если при выполнении транзакции возникнет ошибка, то исходное состояние базы данных можно восстановить. В данном случае durable — антоним volatile (неустойчивый).

Таблица 11.2

Уровень изолированности	Блокировка (-и)	Допустимые проблемы целостности данных
ReadUncommitted	Нет	Грязное чтение, неповторяемое чтение, фантомные данные
ReadCommitted	При редактировании применяется блокировка, предотвращающая чтение записи (-ей) другими пользователями до завершения транзакции	Неповторяемое чтение и фантомные данные
RepeatableRead	При чтении применяется блокировка, предотвращающая редактирование записи (-ей) другими пользователями до завершения транзакции	Фантомные данные
Serializable	Применяются блокировки уровня диапазона ключа, предотвращающие любые действия, способные повлиять на результат, в том числе вставку и удаление данных	Нет
Snapshoot	Нет	Нет

Определение явной транзакции

С помощью свойства `Database` контекста базы данных можно управлять явными транзакциями.

Импортируйте в файл класса `Program.cs` указанное ниже пространство имен для подключения интерфейса `IDbContextTransaction`:

```
using Microsoft.EntityFrameworkCore.Storage;
```

После установки переменной db добавьте в метод DeleteProducts следующие инструкции для запуска явной транзакции и вывода уровня изолированности этой транзакции. В нижней части метода совершите транзакцию и закройте фигурную скобку, как показано ниже:

```
static int DeleteProducts(string name)
{
  using (var db = new Northwind())
  {
    using (IDbContextTransaction t =
    db.Database.BeginTransaction())
    {
      WriteLine($"Transaction started with
      this isolation level: {t.GetDbTransaction().IsolationLevel}");

      var products = db.Products.Where(
      p => p.ProductName.StartsWith(name));
      db.Products.RemoveRange(products);
      int affected = db.SaveChanges();
      t.Commit();
      return affected;
    }
  }
}
```

Запустите консольное приложение и проанализируйте результат вывода.

При использовании Microsoft SQL Server вы увидите следующий уровень изолированности:

```
Transaction started with this isolation level: ReadCommitted
```

Если работаете с SQLite, то увидите такой уровень изолированности:

```
Transaction started with this isolation level: Serializable
```

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенное упражнение и посетите указанный ресурс, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

1. Какой тип вы бы использовали для свойства, представляющего таблицу, например для свойства Products контекста БД Northwind?
2. Какой тип вы бы применили для свойства, представляющего соотношение «один-ко-многим», например для свойства Products сущности Category?
3. Какое соглашение, касающееся первичных ключей, действует в EF Core?
4. Когда бы вы применили атрибут аннотации в классе элемента?
5. Почему вы предпочли бы использовать Fluent API, а не атрибуты аннотации?

6. Что означает уровень изоляции транзакции `Serializable`?
7. Что возвращает метод `DbContext.SaveChanges()`?
8. В чем разница между «жадной» и явной загрузками?
9. Как бы вы определили класс сущности EF Core для этой таблицы?

```
CREATE TABLE Employees(
  EmpID INT IDENTITY,
  FirstName NVARCHAR(40) NOT NULL,
  Salary MONEY
)
```

10. Каковы преимущества объявления свойств сущности как `virtual`?

Упражнение 11.1. Практика экспорта данных с помощью различных форматов сериализации

Создайте консольное приложение с именем `Exercise02`, запрашивающее у базы данных Northwind все категории и продукты, а затем сериализующее полученные данные с помощью хотя бы трех форматов сериализации, доступных в .NET Core.

Какой из форматов сериализации использует наименьшее количество байтов?

Упражнение 11.2. Изучение документации EF Core

Перейдите на сайт https://docs.microsoft.com/en-us/ef/core/index и прочтите официальную документацию по Entity Framework Core. Воспользуйтесь учебными материалами, чтобы создать приложения и сервисы для настольных компьютеров под управлением операционной системы Windows, а также веб-приложений и сервисов. Если на вашем компьютере установлена операционная система macOS или Linux, то обратитесь к учебным материалам для этих альтернативных платформ.

Резюме

В этой главе вы научились подключаться к базе данных, выполнять простой запрос LINQ и обрабатывать результаты, а также создавать модели данных Code First для существующих баз данных, таких как Northwind.

В следующей главе вы научитесь писать более сложные запросы LINQ, позволяющие выбирать, фильтровать, сортировать, объединять и группировать данные.

12 Создание запросов и управление данными с помощью LINQ

Эта глава посвящена технологии *LINQ* (Language Integrated Query — внутриязыковой запрос). Она представляет собой набор языковых расширений, позволяющих работать с последовательностью элементов, а также фильтровать, сортировать и проецировать их в различные программные выводы.

В данной главе:

- написание запросов LINQ;
- работа с множествами;
- применение LINQ на платформе EF Core;
- подслащение синтаксиса с помощью синтаксического сахара;
- использование нескольких потоков и PLINQ;
- создание собственных методов расширения LINQ;
- работа с LINQ to XML.

Написание запросов LINQ

Несмотря на то что в главе 11 мы уже написали несколько запросов LINQ, я не объяснил подробно, как работает данная технология.

LINQ состоит из нескольких частей, одни из них обязательные, а другие — дополнительные.

- *Методы расширения (обязательные)*: набор методов включает в себя `Where`, `OrderBy`, `Select` и др. Эти методы как раз и предоставляют функциональность LINQ.
- *Поставщики данных LINQ (обязательные)*: набор включает в себя LINQ to Objects, LINQ to Entities, LINQ to XML, LINQ to OData, LINQ to Amazon, а также

другие поставщики данных LINQ. Они конвертируют стандартные операции LINQ в команды, характерные для различных видов данных.

- *Лямбда-выражения (дополнительные)*: могут применяться вместо именованных методов для упрощения вызовов методов расширения LINQ.
- *Понятный синтаксис запросов LINQ (дополнительный)*: этот синтаксис включает в себя такие ключевые слова, как `from`, `in`, `where`, `orderby`, `descending`, `select` и др. Вышеперечисленное — ключевые слова языка C#, являющиеся псевдонимами для некоторых методов расширения LINQ. Использование этих псевдонимов поможет упростить написание запросов, особенно при наличии опыта работы с другими языками написания запросов, такими как *Structured Query Language (SQL)*.

При первом знакомстве с LINQ многие программисты зачастую полагают, будто понятный синтаксис запросов и есть сама технология, но ирония в том, что этот синтаксис — одна из ее необязательных частей!

Расширение последовательностей с помощью перечислимого класса

Такие методы расширения, как `Where` и `Select`, подставляются к любому типу с помощью класса `Enumerable`, называемого *последовательностью* и реализующего интерфейс `IEnumerable<T>`. Класс `Enumerable` показан на рис. 12.1.

Например, массив любого типа автоматически реализует `IEnumerable<T>`, где `T` — это тип элемента массива, а значит, все массивы поддерживают создание запросов и управление с помощью LINQ.

Все универсальные коллекции, такие как `List<T>`, `Dictionary<IKey, IValue>`, `Stack<T>` и `Queue<T>`, реализуют `IEnumerable<T>` с тем, чтобы к этим коллекциям можно было создавать запросы и управлять ими, используя LINQ.

Коллекция `DbSet<T>` тоже реализует `IEnumerable<T>`, поэтому LINQ может применяться для запроса и обработки моделей сущностей, созданных для EF Core.

Фильтрация элементов с помощью метода Where

Наиболее распространенная причина применения LINQ — возможность фильтрации элементов в последовательности благодаря методу расширения `Where`.

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект). В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Core. В центре диалогового окна выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)), присвойте ему имя `LinqWithObjects`, укажите расположение по адресу `C:\Code`, введите имя решения `Chapter12`, а затем нажмите кнопку OK.

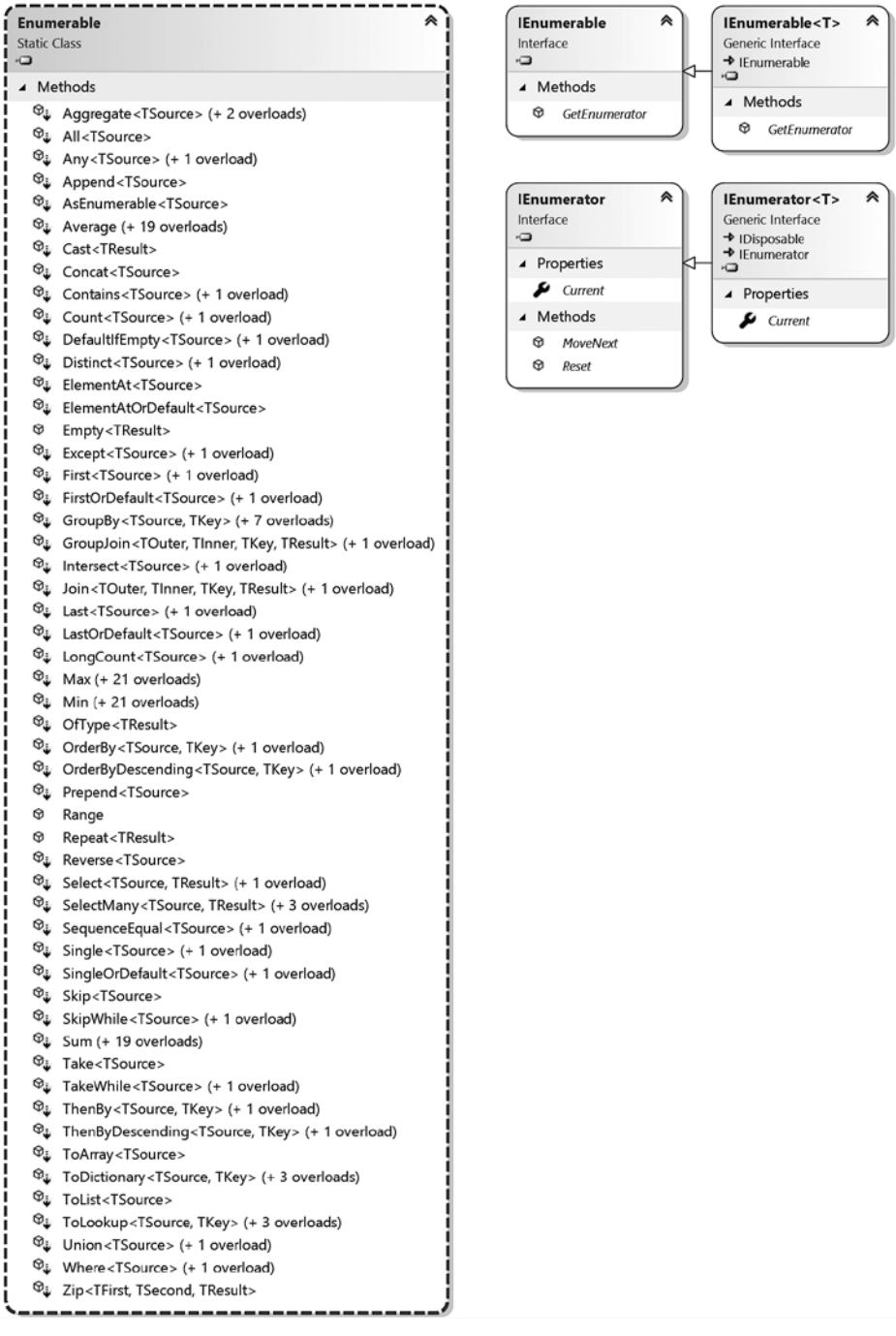

Enumerable
Static Class
Methods
Aggregate<TSource> (+ 2 overloads)
All<TSource>
Any<TSource> (+ 1 overload)
Append<TSource>
AsEnumerable<TSource>
Average (+ 19 overloads)
Cast<TResult>
Concat<TSource>
Contains<TSource> (+ 1 overload)
Count<TSource> (+ 1 overload)
DefaultIfEmpty<TSource> (+ 1 overload)
Distinct<TSource> (+ 1 overload)
ElementAt<TSource>
ElementAtOrDefault<TSource>
Empty<TResult>
Except<TSource> (+ 1 overload)
First<TSource> (+ 1 overload)
FirstOrDefault<TSource> (+ 1 overload)
GroupBy<TSource, TKey> (+ 7 overloads)
GroupJoin<TOuter, TInner, TKey, TResult> (+ 1 overload)
Intersect<TSource> (+ 1 overload)
Join<TOuter, TInner, TKey, TResult> (+ 1 overload)
Last<TSource> (+ 1 overload)
LastOrDefault<TSource> (+ 1 overload)
LongCount<TSource> (+ 1 overload)
Max (+ 21 overloads)
Min (+ 21 overloads)
OfType<TResult>
OrderBy<TSource, TKey> (+ 1 overload)
OrderByDescending<TSource, TKey> (+ 1 overload)
Prepend<TSource>
Range
Repeat<TResult>
Reverse<TSource>
Select<TSource, TResult> (+ 1 overload)
SelectMany<TSource, TResult> (+ 3 overloads)
SequenceEqual<TSource> (+ 1 overload)
Single<TSource> (+ 1 overload)
SingleOrDefault<TSource> (+ 1 overload)
Skip<TSource>
SkipWhile<TSource> (+ 1 overload)
Sum (+ 19 overloads)
Take<TSource>
TakeWhile<TSource> (+ 1 overload)
ThenBy<TSource, TKey> (+ 1 overload)
ThenByDescending<TSource, TKey> (+ 1 overload)
ToArray<TSource>
ToDictionary<TSource, TKey> (+ 3 overloads)
ToList<TSource>
ToLookup<TSource, TKey> (+ 3 overloads)
Union<TSource> (+ 1 overload)
Where<TSource> (+ 1 overload)
Zip<TFirst, TSecond, TResult>
IEnumerable
Interface
Methods
GetEnumerator
IEnumerable<T>
Generic Interface
IEnumerable
Methods
GetEnumerator
IEnumerator
Interface
Properties
Current
Methods
MoveNext
Reset
IEnumerator<T>
Generic Interface
IDisposable
IEnumerator
Properties
Current

Рис. 12.1

В Visual Studio Code создайте каталог `Chapter12` и подкаталог `LinqWithObjects`. Откройте последний и выполните команду `dotnet new console` в области TERMINAL (Терминал).

Добавьте в класс `Program` метод `LinqWithArrayOfStrings`, который определяет массив строк, а затем попытается вызвать на нем метод расширения `Where`, как показано в листинге ниже:

```
static void LinqWithArrayOfStrings()
{
  var names = new string[] { "Michael", "Pam", "Jim", "Dwight",
      "Angela", "Kevin", "Toby", "Creed" };
  var query = names.
```

Во время ввода метода `Where` обратите внимание, что в списке членов массива `string` в подсказке меню IntelliSense нет метода с таким именем (рис. 12.2).

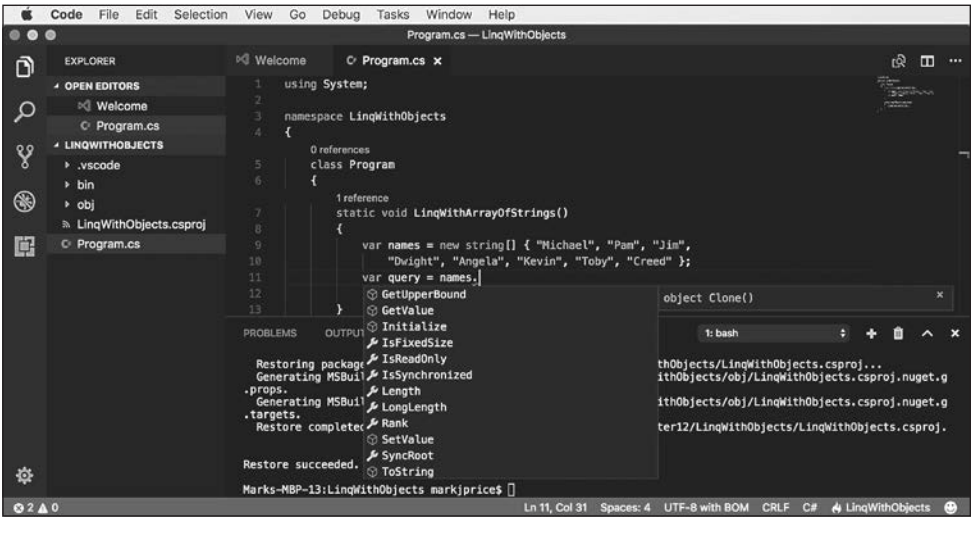


Рис. 12.2

Это происходит потому, что `Where` — *метод расширения*. Он не существует для типа массива. Этот метод существует в отдельной сборке и пространстве имен. Можно сделать метод расширения `Where` доступным, импортировав пространство имен `System.Linq`.

Добавьте следующую инструкцию в начало файла `Program.cs`:

```
using System.Linq;
```

Теперь при вводе скобок после имени метода `Where` обратите внимание на меню IntelliSense. Система подсказывает, что для вызова метода `Where` нужно передать в него экземпляр делегата `Func<string, bool>`. Этот делегат должен ссылаться на метод с подходящей сигнатурой (рис. 12.3).

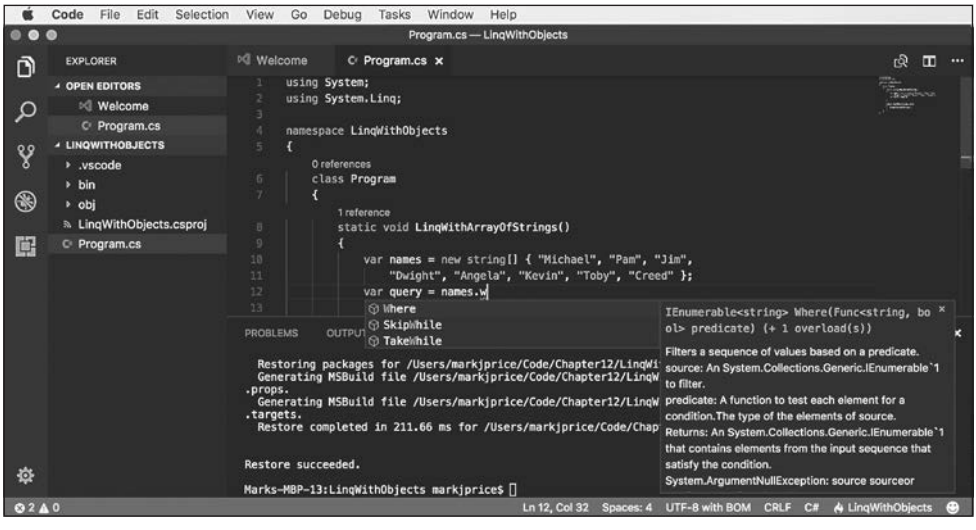


Рис. 12.3

Введите следующий код, чтобы создать новый экземпляр делегата:

```
var query = names.Where(new Func<string, bool>())
```

Обратите внимание, на подсказку меню IntelliSense, отображаемую в Visual Studio 2017 (но не в Visual Studio Code). Она сообщает нам, что метод `target` должен иметь один входящий параметр типа `string` и возвращать тип `bool` (рис. 12.4).

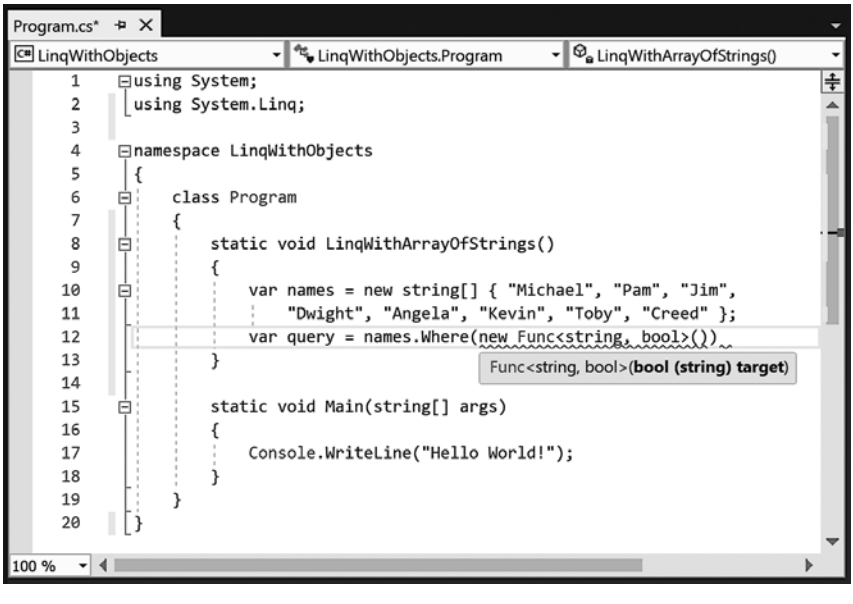


Рис. 12.4

Для каждой переменной типа `string`, передаваемой в метод, метод должен возвращать логическое значение. Если он возвращает `true`, то следует включить `string` в результаты, однако в случае возврата `false` нужно исключить данную строку из результатов.

Ссылка на именованные методы

Определим метод, который включает только имена длиннее четырех символов.

Статически импортируйте класс `Console`, а затем добавьте следующий метод в класс `Program`:

```
static bool NameLongerThanFour(string name)
{
  return name.Length > 4;
}
```

Передайте имя метода в делегат `Func<string, bool>`, а затем переберите элементы запроса, как показано в листинге ниже:

```
var query = names.Where(new Func<string, bool>(NameLongerThanFour));
foreach (string item in query)
{
  WriteLine(item);
}
```

В методе `Main` вызовите метод `LinqWithArrayOfStrings`, как показано в следующем листинге:

```
static void Main(string[] args)
{
  LinqWithArrayOfStrings();
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Michael
Dwight
Angela
Kevin
Creed
```

Упрощение кода путем удаления явного создания экземпляров делегатов

Можно упростить программный код, удалив явное создание экземпляра делегата `Func<string, bool>`. Компилятор C# создаст этот экземпляр самостоятельно, так что не нужно делать это явно.

Скопируйте и вставьте запрос, закомментируйте первый из них и измените второй, удалив явное создание экземпляра делегата `Func<string, bool>`, как показано в коде ниже:

```
// var query = names.Where(new Func<string, bool>(NameLongerThanFour));

var query = names.Where(NameLongerThanFour);
```

Перезапустите приложение и обратите внимание, что его поведение не изменилось.

Ссылка на лямбда-выражение

Можно упростить код еще больше, используя *лямбда-выражение* вместо именованного метода.

Несмотря на то что изначально это может казаться сложным, по сути, лямбда-выражение — *безымянная функция*. В таких выражениях используется символ `=>` (читается как «идет в») для обозначения возвращаемого значения.

Скопируйте и вставьте запрос, закомментируйте один из них и измените второй, чтобы он выглядел так:

```
var query = names.Where(name => name.Length > 4);
```

Обратите внимание, что синтаксис лямбда-выражения включает в себя все важнейшие части метода `NameLongerThanFour`, но ничего кроме них. Для лямбда-выражения требуется лишь определить следующее:

- имена входящих параметров;
- выражение возврата значения.

Тип входящего параметра `name` предполагается исходя из того, что последовательность содержит строковые значения. Притом для корректной работы метода `Where` тип возвращаемого значения должен быть `bool`, вследствие чего и выражению после символа `=>` следует возвращать это же значение.

Компилятор сделает большую часть работы за нас, благодаря чему наш код может быть максимально кратким.

Перезапустите приложение и обратите внимание на то, что его поведение не изменилось.

Сортировка элементов

Метод `Where` — лишь один из около 30 методов расширения, предоставляемых типом `Enumerable`. Из методов расширения может быть составлена цепочка, если предыдущий метод возвращает еще одну последовательность, иными словами, тип, реализующий интерфейс `IEnumerable<T>`.

Сортировка отдельных элементов с помощью OrderBy

Подставьте вызов метода `OrderBy` в конец уже написанного запроса, как показано ниже:

```
var query = names.Where(name => name.Length > 4)
    .OrderBy(name => name.Length);
```

Для повышения удобочитаемости форматируйте инструкцию LINQ таким образом, чтобы вызов каждого метода расширения происходил на отдельной строке.

Перезапустите приложение и обратите внимание, что теперь имена отсортированы по увеличению количества символов.

```
Kevin
Creed
Dwight
Angela
Michael
```

Чтобы поместить самое длинное имя в начало списка, воспользуйтесь методом OrderByDescending.

Сортировка по нескольким свойствам с помощью метода ThenBy

Иногда необходимо выполнить сортировку по более чем одному свойству.

Подставьте вызов метода `ThenBy` в конец уже написанного запроса, как показано далее:

```
var query = names.Where(name => name.Length > 4)
    .OrderBy(name => name.Length).ThenBy(name => name);
```

Перезапустите приложение и обратите внимание на небольшое изменение в порядке сортировки. В рамках группы имен с одинаковым количеством букв имена отсортированы по алфавиту по полному значению строки, таким образом имя `Creed` следует перед именем `Kevin`, а `Angela` — перед `Dwight`:

```
Creed
Kevin
Angela
Dwight
Michael
```

Фильтрация по типу

Метод расширения `Where` отлично работает для фильтрации по значению, таких как текст и числа. Но что, если последовательность содержит несколько типов, а вы хотите отфильтровать по какому-то конкретному из них и при этом сохранить иерархию наследования?

Представьте, что перед вами последовательность исключений, как показано в следующем примере кода:

```
var errors = new Exception[]
{
  new ArgumentException(),
```

```
  new SystemException(),
  new IndexOutOfRangeException(),
  new InvalidOperationException(),
  new NullReferenceException(),
  new InvalidCastException(),
  new OverflowException(),
  new DivideByZeroException(),
  new ApplicationException()
};
```

Исключения имеют сложную иерархию (рис. 12.5).

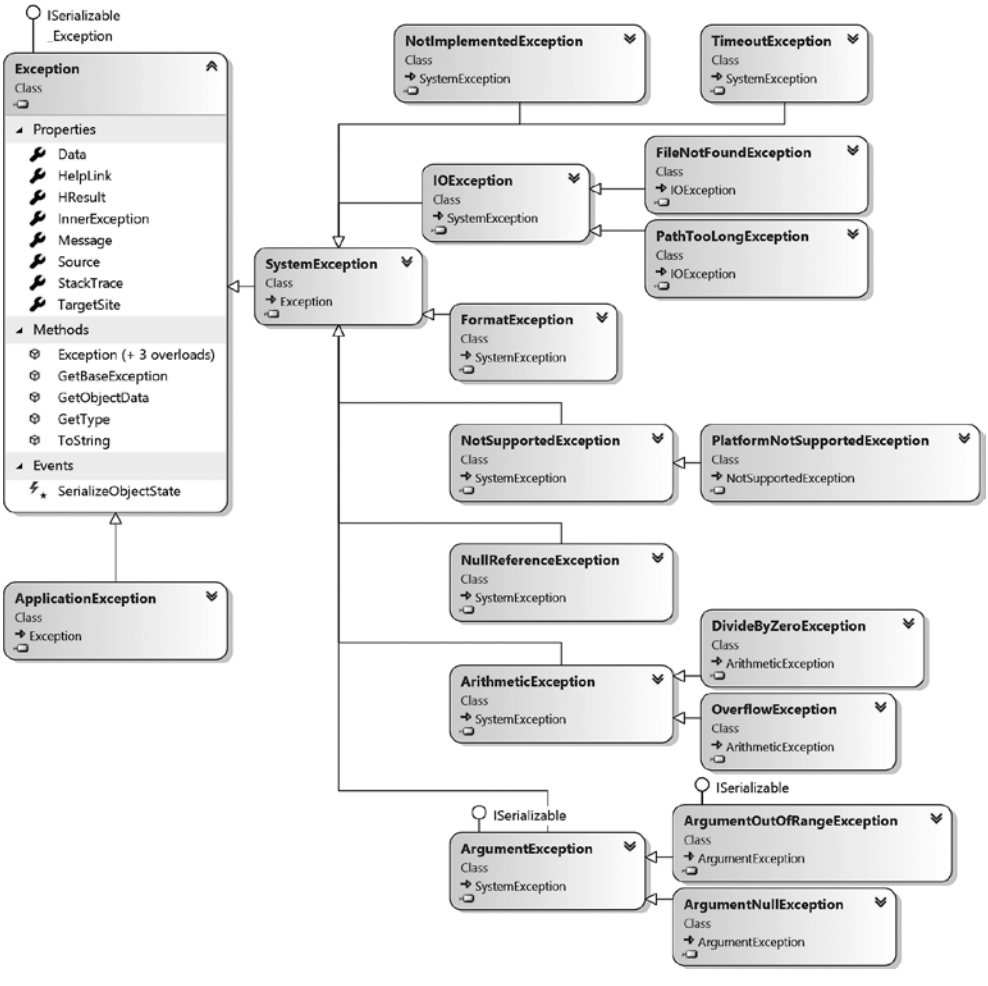


Рис. 12.5

Для фильтрации по типу можно написать инструкции с помощью метода расширения `OfType<T>`, как показано в примере ниже:

```
var numberErrors = errors.OfType<ArithmeticException>();

foreach (var error in numberErrors)
{
  WriteLine(error);
}
```

Результаты содержат только исключения типа `ArithmeticException` или наследующих от него типов, как показано в данном выводе:

```
System.OverflowException: Arithmetic operation resulted in an overflow.
System.DivideByZeroException: Attempted to divide by zero.
```

Работа с множествами

Множества — одно из наиболее фундаментальных понятий в математике. Это коллекция, состоящая из одного или нескольких объектов. Возможно, в свое время вы изучали диаграммы Венна. Универсальный набор операций включает в себя *пересечение* и *объединение* множеств.

Создайте новый проект консольного приложения с именем `LinqWithSets` в Visual Studio 2017 или Visual Studio Code.

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства) или нажмите сочетание клавиш Alt+Enter. В категории Startup Project (Запускаемый проект) установите переключатель в положение Current selection (Текущий проект).

В данном приложении мы определим три строковых массива для групп студентов, а затем выполним с этими массивами несколько общих операций.

Импортируйте такие дополнительные пространства имен:

```
using System.Collections.Generic;    // для IEnumerable<T>
using System.Linq;                   // для методов расширения LINQ
```

Добавьте в класс `Program` перед методом `Main` следующий метод для вывода в консоли последовательности переменных `string` в виде одной строки с запятыми в качестве разделителя и дополнительной строкой описания:

```
private static void Output(IEnumerable<string> cohort,
string description = "")
{
  if (!string.IsNullOrEmpty(description))
  {
    WriteLine(description);
  }
  Write(" ");
  WriteLine(string.Join(", ", cohort.ToArray()));
}
```

В методе `Main` напишите такие инструкции:

```
var cohort1 = new string[]
  { "Rachel", "Gareth", "Jonathan", "George" };
```

```
var cohort2 = new string[]
  { "Jack", "Stephen", "Daniel", "Jack", "Jared" };
var cohort3 = new string[]
  { "Declan", "Jack", "Jack", "Jasmine", "Conor" };

Output(cohort1, "Cohort 1");
Output(cohort2, "Cohort 2");
Output(cohort3, "Cohort 3");
WriteLine();

Output(cohort2.Distinct(),"cohort2.Distinct(): removes duplicates");

Output(cohort2.Union(cohort3),"cohort2.Union(cohort3):
combines and removes duplicates");

Output(cohort2.Concat(cohort3),"cohort2.Concat(cohort3):
combines but leaves duplicates");

Output(cohort2.Intersect(cohort3),"cohort2.Intersect(cohort3):
items that are in both sequences");
Output(cohort2.Except(cohort3),"cohort2.Except(cohort3):
removes items from the first sequence that are in the second sequence");
Output(cohort1.Zip(cohort2,(c1, c2) => $"{c1} matched with {c2}"),
      "cohort1.Zip(cohort2, (c1, c2) => $\"{c1} matched with {c2}\")" +
      ": matches items based on position in the sequence");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Cohort 1
  Rachel, Gareth, Jonathan, George
Cohort 2
  Jack, Stephen, Daniel, Jack, Jared
Cohort 3
  Declan, Jack, Jack, Jasmine, Conor
cohort2.Distinct(): removes duplicates
  Jack, Stephen, Daniel, Jared
cohort2.Union(cohort3): combines two sequences and removes any duplicates
  Jack, Stephen, Daniel, Jared, Declan, Jasmine, Conor
cohort2.Concat(cohort3): combines two sequences but leaves in any duplicates
  Jack, Stephen, Daniel, Jack, Jared, Declan, Jack, Jack, Jasmine, Conor
cohort2.Intersect(cohort3): returns items that are in both sequences
  Jack
cohort2.Except(cohort3): removes items from the first sequence that are in the
second sequence
  Stephen, Daniel, Jared
cohort1.Zip(cohort2, (c1, c2) => $"{c1} matched with {c2}"): matches items based on
position in the sequence
  Rachel matched with Jack, Gareth matched with Stephen, Jonathan matched with
Daniel, George matched with Jack
```

При использовании Zip неравное количество элементов в последовательностях приведет к тому, что некоторые из элементов останутся без подходящей пары.

Применение LINQ на платформе EF Core

Для изучения *проекции* лучше использовать более сложные последовательности, поэтому в проекте ниже мы задействуем образец базы данных `Northwind`.

Проекция элементов с помощью ключевого слова select

Создайте новый проект консольного приложения с именем `LinqWithEFCore`.

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) раскройте проект LinqWithEFCore, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Manage NuGet Packages (Управление пакетами NuGet). На открывшейся панели перейдите на вкладку Browse (Обзор), выполните поиск пакета Microsoft.EntityFrameworkCore.SqlServer, выберите его и нажмите кнопку Install (Установить).

Если вы не завершили работу с главой 11, то откройте файл `Northwind4SQLServer.sql`, щелкните правой кнопкой мыши и выберите пункт Execute (Выполнить) для создания базы данных `Northwind` на сервере `(localdb)\mssqllocaldb`.

В Visual Studio Code измените содержимое файла `LinqWithEFCore.csproj`, как показано ниже (выделено полужирным):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
    Include="Microsoft.EntityFrameworkCore.Sqlite"
    Version="2.0.0" />
  </ItemGroup>

</Project>
```

Скопируйте файл `Northwind4SQLite.sql` в папку `LinqWithEFCore`, а затем воспользуйтесь панелью Integrated Terminal (Интегрированный терминал), чтобы создать базу данных `Northwind` путем выполнения следующей команды:

```
sqlite3 Northwind.db < Northwind4SQLite.sql
```

Создание модели EF Core

В Visual Studio 2017 или Visual Studio Code добавьте в проект три файла: `Northwind.cs`, `Category.cs` и `Product.cs`.

Ваш класс, наследующийся от `DbContext`, должен содержать переопределенный метод `OnConfiguring`. Этот метод установит строку подключения к базе данных. Для использования SQL Server нужно раскомментировать вызов метода `UseSqlServer`, для применения SQLite — вызов метода `UseSqlite`.

Измените файл класса `Northwind.cs`, как показано в следующем листинге:

```
using Microsoft.EntityFrameworkCore;

namespace Packt.CS7
{
  // управление подключением к БД
  public class Northwind : DbContext
  {
    // эти свойства соотносятся с таблицами в базе
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
    {
      // для использования Microsoft SQL Server раскомментируйте код,
      // данный ниже
      // optionsBuilder.UseSqlServer(
      //   @"Data Source=(localdb)\mssqllocaldb;" +
      //   "Initial Catalog=Northwind;" +
      //   "Integrated Security=true;" +
      //   "MultipleActiveResultSets=true;");

      // чтобы использовать SQLite, раскомментируйте этот код
      // string path = System.IO.Path.Combine(
      //   System.Environment.CurrentDirectory, "Northwind.db");
      // optionsBuilder.UseSqlite($"Filename={path}");
    }
  }
}
```

Файл `Category.cs` должен выглядеть следующим образом:

```
using System.ComponentModel.DataAnnotations;

namespace Packt.CS7
{
  public class Category
  {
    public int CategoryID { get; set; }
    [Required]
    [StringLength(15)]
    public string CategoryName { get; set; }
    public string Description { get; set; }
  }
}
```

Файл `Product.cs` должен выглядеть так:

```
using System.ComponentModel.DataAnnotations;

namespace Packt.CS7
{
  public class Product
  {
    public int ProductID { get; set; }
```

```
    [Required]
    [StringLength(40)]
    public string ProductName { get; set; }
    public int? SupplierID { get; set; }
    public int? CategoryID { get; set; }
    [StringLength(20)]
    public string QuantityPerUnit { get; set; }
    public decimal? UnitPrice { get; set; }
    public short? UnitsInStock { get; set; }
    public short? UnitsOnOrder { get; set; }
    public short? ReorderLevel { get; set; }
    public bool Discontinued { get; set; }
  }
}
```

Мы не определили отношения между двумя классами сущностей. Это было сделано намеренно. Позднее вы примените LINQ для соединения двух наборов сущностей.

Откройте файл `Program.cs` и импортируйте следующие пространства имен:

```
using static System.Console;
using Packt.CS7;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

В методе `Main` напишите такие инструкции:

```
using (var db = new Northwind())
{
  var query = db.Products
    .Where(product => product.UnitPrice < 10M)
    .OrderByDescending(product => product.UnitPrice);

  WriteLine("Products that cost less than $10:");
  foreach (var item in query)
  {
    WriteLine($"{item.ProductID}: {item.ProductName}
    costs {item.UnitPrice:$#,##0.00}");
  }
  WriteLine();
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Products that cost less than $10:
41: Jack's New England Clam Chowder costs $9.65
45: Rogede sild costs $9.50
47: Zaanse koeken costs $9.50
19: Teatime Chocolate Biscuits costs $9.20
23: Tunnbröd costs $9.00
75: Rhönbräu Klosterbier costs $7.75
54: Tourtière costs $7.45
52: Filo Mix costs $7.00
13: Konbu costs $6.00
24: Guaraná Fantástica costs $4.50
33: Geitost costs $2.50
```

Хотя приведенный запрос и распечатывает необходимую информацию, он выполняет данное действие настолько неэффективно, что возвращает все столбцы таблицы Products вместо трех нужных столбцов. Это эквивалентно следующей инструкции SQL:

```
SELECT * FROM Products;
```

Вероятно, вы заметили, что последовательности реализуют интерфейсы IQueryable<T> и IOrderedQueryable<T> вместо интерфейсов IEnumerable<T> и IOrderedEnumerable<T>. Данный факт демонстрирует применение поставщика данных LINQ, который, в свою очередь, использует отложенное исполнение и создает запрос в памяти с помощью деревьев выражений. Запрос не станет выполняться до последнего момента и, только когда тот наступит, будет преобразован в другой язык запросов, такой как Transact-SQL для Microsoft SQL Server. Перечисление запроса с помощью foreach или вызов метода наподобие ToArray приведет к незамедлительному выполнению запроса.

В методе Main измените запрос LINQ, чтобы использовать метод Select и возвратить только три нужных свойства (столбцов таблицы), как показано ниже:

```
var query = db.Products
  .Where(product => product.UnitPrice < 10M)
  .OrderByDescending(product => product.UnitPrice)
  .Select(product => new
  {
    product.ProductID,
    product.ProductName,
    product.UnitPrice
  });
```

Запустите консольное приложение и убедитесь в том, что его вывод не изменился.

Присоединение и группировка

Существуют два метода расширения для присоединения и группировки.

- Join принимает четыре параметра: последовательность, к которой требуется присоединиться; свойство или свойства *левой* последовательности, с которыми нужно найти соответствие; свойство или свойства *правой* последовательности, с которыми следует найти соответствие; проекцию;
- GroupJoin принимает те же параметры, только объединяет совпадения в групповой объект, где Key используется для совпавшего значения, а интерфейс IEnumerable<T> — для нескольких совпадений.

Добавьте в метод Main эти инструкции:

```
// создаем две последовательности, которые хотим присоединить
var categories = db.Categories.Select(
```

```
  c => new { c.CategoryID, c.CategoryName }).ToArray();

var products = db.Products.Select(
  p => new { p.ProductID, p.ProductName,
  p.CategoryID }).ToArray();

// присоединяем каждый товар к своей категории, чтобы вернуть 77 совпадений
var queryJoin = categories.Join(products,
  category => category.CategoryID,
  product => product.CategoryID,
   (c, p) => new { c.CategoryName, p.ProductName,
  p.ProductID });

foreach (var item in queryJoin)
{
  WriteLine($"{item.ProductID}: {item.ProductName} is in
  {item.CategoryName}.");
}
```

Запустите консольное приложение и проанализируйте результат вывода.

Обратите внимание, что каждый из 77 товаров выводится в отдельной строке, а результаты сначала отображают все товары из категории `Beverages`, затем `Condiments` и т. д.:

```
1: Chai is in Beverages.
2: Chang is in Beverages.
24: Guaraná Fantástica is in Beverages.
34: Sasquatch Ale is in Beverages.
35: Steeleye Stout is in Beverages.
38: Côte de Blaye is in Beverages.
39: Chartreuse verte is in Beverages.
43: Ipoh Coffee is in Beverages.
67: Laughing Lumberjack Lager is in Beverages.
70: Outback Lager is in Beverages.
75: Rhönbräu Klosterbier is in Beverages.
76: Lakkalikööri is in Beverages.
3: Aniseed Syrup is in Condiments.
4: Chef Anton's Cajun Seasoning is in Condiments.
```

Измените запрос для выполнения сортировки по `ProductID`:

```
var queryJoin = categories.Join(products,
  category => category.CategoryID,
  product => product.CategoryID,
   (c, p) => new { c.CategoryName, p.ProductName,
  p.ProductID }).OrderBy(cp => cp.ProductID);
```

Перезапустите приложение и проанализируйте результат вывода:

```
1: Chai is in Beverages.
2: Chang is in Beverages.
3: Aniseed Syrup is in Condiments.
4: Chef Anton's Cajun Seasoning is in Condiments.
5: Chef Anton's Gumbo Mix is in Condiments.
6: Grandma's Boysenberry Spread is in Condiments.
```

```
7: Uncle Bob's Organic Dried Pears is in Produce.
8: Northwoods Cranberry Sauce is in Condiments.
9: Mishi Kobe Niku is in Meat/Poultry.
10: Ikura is in Seafood.
11: Queso Cabrales is in Dairy Products.
12: Queso Manchego La Pastora is in Dairy Products.
13: Konbu is in Seafood.
14: Tofu is in Produce.
15: Genen Shouyu is in Condiments.
```

Добавьте несколько инструкций, указанных в следующем листинге, в нижнюю часть метода `Main` для демонстрации использования метода `GropJoin`, и в выводе покажите название группы, а затем перечисление всех ее элементов:

```
// группируем все товары по соответствующим категориям,
// чтобы вернуть восемь совпадений
var queryGroup = categories.GroupJoin(products,
  category => category.CategoryID,
  product => product.CategoryID,
   (c, Products) => new { c.CategoryName,
  Products = Products.OrderBy(p => p.ProductName) });

foreach (var item in queryGroup)
{
  WriteLine($"{item.CategoryName} has
  {item.Products.Count()} products.");
  foreach (var product in item.Products)
  {
    WriteLine($" {product.ProductName}");
  }
}
```

Перезапустите консольное приложение и проанализируйте результат вывода.

Обратите внимание, что все товары в каждой категории были отсортированы по алфавиту, как и было задано в запросе:

```
Beverages has 12 products.
  Chai
  Chang
  Chartreuse verte
  Côte de Blaye
  Guaraná Fantástica
  Ipoh Coffee
  Lakkalikööri
  Laughing Lumberjack Lager
  Outback Lager
  Rhönbräu Klosterbier
  Sasquatch Ale
  Steeleye Stout
Condiments has 12 products.
  Aniseed Syrup
  Chef Anton's Cajun Seasoning
  Chef Anton's Gumbo Mix
```

Агрегирование последовательностей

Существуют методы расширения LINQ для выполнения функций агрегирования, например `Average` и `Sum`.

Добавьте в нижнюю часть метода `Main` несколько новых инструкций, иллюстрирующих применение методов расширения для агрегирования, как показано в следующем листинге:

```
WriteLine("Products");
WriteLine($" Count:
{db.Products.Count()}");
WriteLine($" Sum of units in stock:
{db.Products.Sum(p => p.UnitsInStock):N0}");
WriteLine($" Sum of units on order:
{db.Products.Sum(p => p.UnitsOnOrder):N0}");
WriteLine($" Average unit price:
{db.Products.Average(p => p.UnitPrice):$#,##0.00}");
WriteLine($" Value of units in stock:
{db.Products.Sum(p => p.UnitPrice * p.UnitsInStock):$#,##0.00}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Products
  Count: 77
  Sum of units in stock: 3,119
  Sum of units on order: 780
  Average unit price: $28.87
  Value of units in stock: $74,050.85
```

Подслащение синтаксиса с помощью синтаксического сахара

В C# 3 в 2008 году были представлены новые ключевые слова, которые упростили программистам работу с SQL при написании запросов LINQ. Иногда этот *синтаксический сахар* называется *понятным синтаксисом запросов LINQ.*

Функциональность понятного синтаксиса запросов LINQ весьма ограничена. Для доступа ко всем функциям LINQ нужно использовать методы расширения.

Рассмотрим следующий код:

```
var names = new string[] { "Michael", "Pam", "Jim", "Dwight", "Angela", "Kevin",
"Toby", "Creed" };

var query = names.Where(name => name.Length > 4).OrderBy(name => name.Length).
ThenBy(name => name);
```

Вместо того чтобы писать вышеприведенный код, используя *методы расширения* и *лямбда-выражения*, можно написать следующий код с помощью *понятного синтаксиса запросов*:

```
var query = from name in names where name.Length > 4 orderby name.Length, name
select name;
```

Компилятор самостоятельно преобразует понятный синтаксис запросов в методы расширения и лямбда-выражения.

Понятному синтаксису запросов LINQ всегда требуется ключевое слово select. Метод расширения Select необязателен при использовании методов расширений и лямбда-выражений.

Не у всех методов расширения есть эквивалентное ключевое слово на языке C#. В качестве таковых можно назвать методы `Skip` и `Take`, которые обычно применяются для постраничного просмотра большого объема данных. Следующий запрос не может быть написан только с помощью синтаксиса запросов:

```
var query = names.Where(name => name.Length > 4)
  .OrderBy(name => name.Length)
  .ThenBy(name => name).Skip(80).Take(10);
```

К счастью, можно заключить понятный синтаксис запроса в скобки и перейти к использованию методов расширения, как показано в коде, приведенном ниже:

```
var query = (from name in names
  where name.Length > 4
  orderby name.Length, name
  select name).Skip(80).Take(10);
```

Выучите не только методы расширения и лямбда-выражения, но и понятный синтаксис запросов для написания запросов LINQ, так как, скорее всего, вам придется поддерживать код, в котором задействовано все вышеперечисленное.

Использование нескольких потоков с помощью PLINQ

По умолчанию для выполнения одного запроса LINQ применяется только один поток. Применение *PLINQ (Parallel LINQ)* — это простой способ использовать несколько потоков для выполнения одного запроса LINQ.

Не следует полагать, что применение параллельных потоков улучшит производительность ваших приложений. Всегда измеряйте реальные временные показатели и использование ресурсов.

Чтобы проверить информацию раздела в действии, мы начнем с некоего кода, который будет применять только один поток для удвоения 200 миллионов целых чисел. Измерять изменения в производительности мы станем с помощью типа `StopWatch`. Для наблюдения за центральным процессором и использованием его ядра мы задействуем инструменты операционной системы.

В Visual Studio 2017 или Visual Studio Code создайте новый проект консольного приложения с именем `LINQingInParallel`.

Чтобы можно было использовать тип `StopWatch`, импортируйте пространство имен `System.Diagnostics`, а для применения типа `IEnumerable<T>` импортируйте пространство имен `System.Collections.Generic`, `System.Linq`, а также статически импортируйте тип `System.Console`.

Добавьте следующие инструкции в метод `Main`, чтобы создать секундомер, записывающий временны́е интервалы, запустить таймер только по нажатию клавиши, создать 200 миллионов целых чисел, возвести каждое из них в квадрат, остановить таймер и отобразить количество прошедших миллисекунд:

```
var watch = Stopwatch.StartNew();
Write("Press ENTER to start: ");
ReadLine();
watch.Start();

IEnumerable<int> numbers = Enumerable.Range(1, 200_000_000);
var squares = numbers.Select(number => number * 2).ToArray();
watch.Stop();
WriteLine(
$"{watch.ElapsedMilliseconds:#,##0} elapsed milliseconds.");
```

Запустите консольное приложение, но пока *не нажимайте* клавишу Enter.

В операционной системе Windows 10 щелкните правой кнопкой мыши на кнопке Start или нажмите сочетание клавиш Ctrl+Alt+Delete, а затем выберите пункт Task Manager (Диспетчер задач).

В нижней части окна Task Manager (Диспетчер задач) нажмите кнопку More Details (Подробнее). В верхней части этого окна перейдите на вкладку Performance (Производительность).

Щелкните правой кнопкой мыши на графике CPU Utilisation (Использование ЦП) и в контекстном меню выполните команду Change graph to ▸ Logical processors (Изменить график ▸ Логические процессоры) (рис. 12.6).

В операционной системе macOS запустите приложение Activity Monitor (Монитор активности) и увеличьте частоту измерений центрального процессора, выполнив команду View ▸ Update Frequency ▸ Very often (1 sec) (Просмотр ▸ Частота обновления ▸ Очень часто (1с)). Для просмотра графиков центрального процессора выполните команду Window ▸ CPU History (Окно ▸ История ЦП).

Если у вас только один ЦП, то вышеописанное изменение будет несущественным!

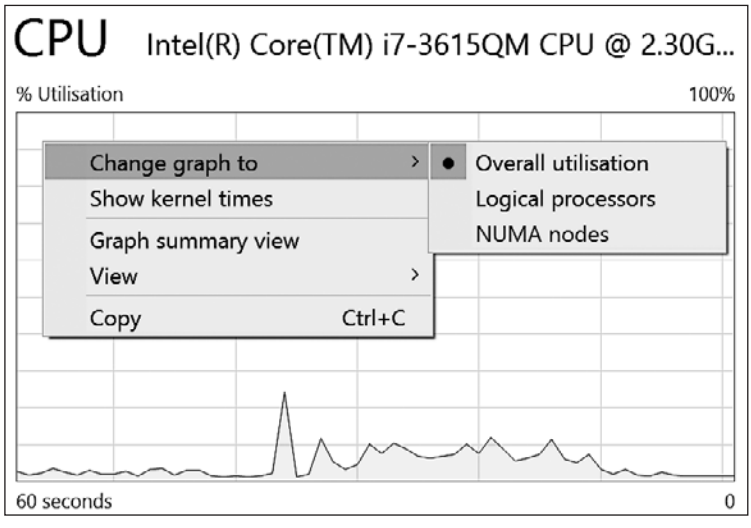


Рис. 12.6

Расположите окна Task Manager (Диспетчер задач) и вашего консольного приложения или окно CPU History (История ЦП) и панель Integrated Terminal (Интегрированный терминал) в среде разработки Visual Studio Code (рис. 12.7).

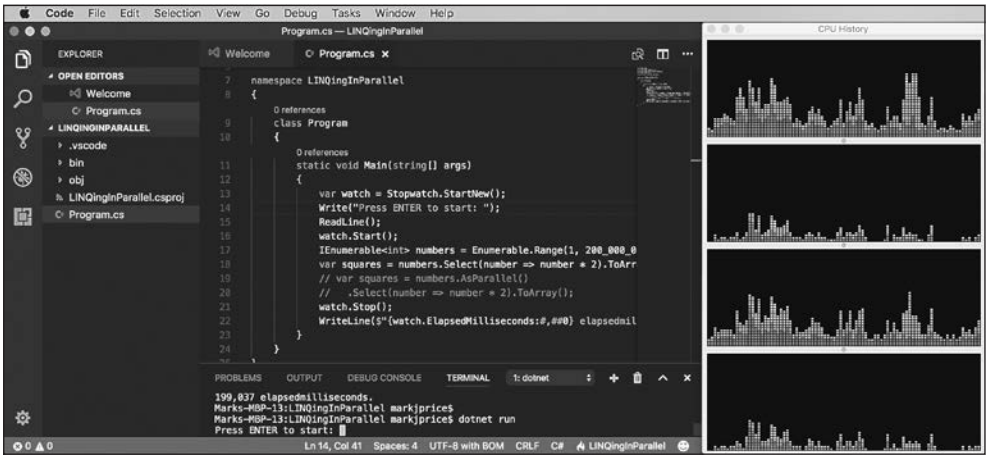


Рис. 12.7

Дождитесь стабилизации графиков центрального процессора и нажмите клавишу Enter для запуска секундомера и обработки запроса. Ваш программный вывод должен выглядеть примерно так (рис. 12.8):

```
Press ENTER to start.
31,230 elapsed milliseconds.
```

Окно Task Manager (Диспетчер задач) или CPU History (История ЦП) должно показать, что по большей части были использованы только один или два центральных процессора. Другие процессоры могут выполнять одновременно фоновые задачи, такие как сбор мусора, вследствие чего графики этих процессоров тоже не будут плоскими, но можно с уверенностью сказать, что работа не распределится поровну между всеми возможными центральными процессорами.

Вернитесь к методу `Main` и измените запрос так, чтобы вызвать метод расширения `AsParallel`:

```
var squares = numbers.AsParallel().Select(number => number * 2).ToArray();
```

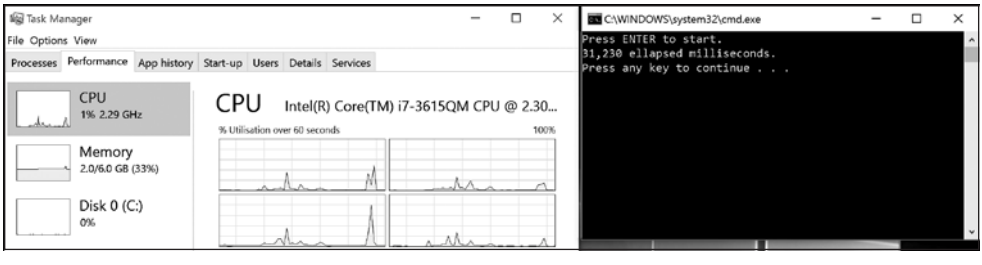


Рис. 12.8

Повторно запустите приложение. Дождитесь нормализации графиков в окне Task Manager (Диспетчер задач) или CPU History (История ЦП) и нажмите клавишу Enter для запуска секундомера и обработки запроса.

В этот раз приложение должно справиться с задачей быстрее (впрочем, ускорение может быть не столь большим, как вам хотелось бы: управление несколькими потоками требует дополнительных усилий!).

```
Press ENTER to start.
26,830 elapsed milliseconds.
```

Окно Task Manager (Диспетчер задач) или CPU History (История ЦП) должно показать, что для выполнения запроса LINQ были использованы все центральные процессоры в равной степени (рис. 12.9, снимок экрана из операционной системы Windows 10).

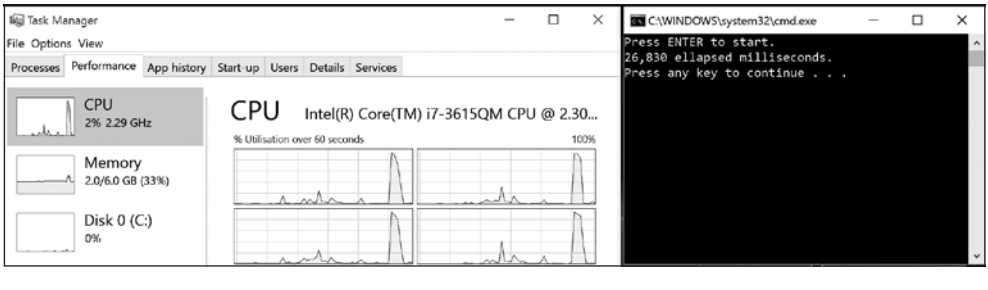


Рис. 12.9

На рис. 12.10 представлен снимок экрана из операционной системы macOS.

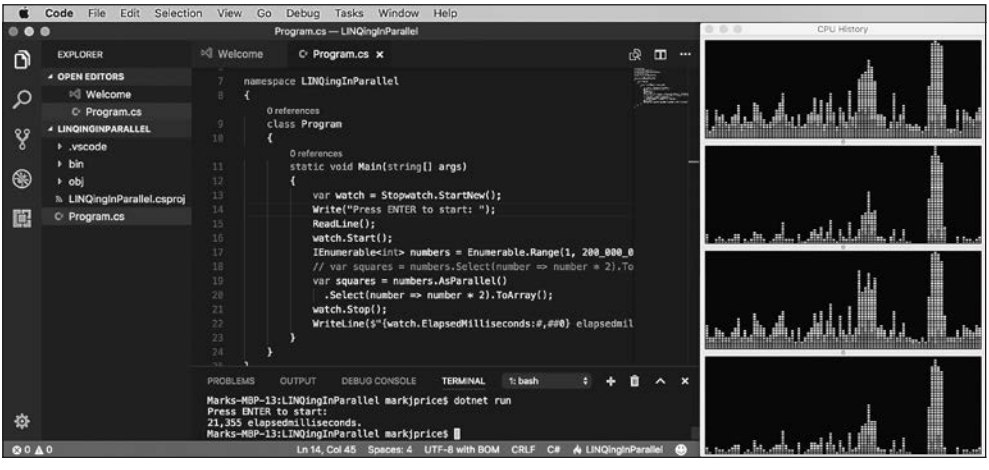


Рис. 12.10

В главе 13 вы более подробно изучите принципы управления несколькими потоками.

Создание собственных методов расширения LINQ

Из главы 6 вы узнали, как создавать пользовательские методы расширения. Для создания методов расширения LINQ нужно лишь расширить тип `IEnumerable<T>`.

Размещайте свои методы расширения в отдельной библиотеке классов, чтобы их можно было легко развернуть в виде сборки или NuGet-пакета.

В Visual Studio 2017 или Visual Studio Code откройте проект или каталог `LinqWithEFCore` и добавьте новый файл класса `MyLINQExtensions.cs`.

В качестве примера рассмотрим метод расширения `Average`. Любой школьник вам скажет, что *среднее* может означать три понятия:

- *среднее арифметическое* — сумма чисел, деленная на их количество;
- *мода* — наиболее часто встречающееся число;
- *медиана* — число в середине упорядоченной последовательности чисел.

Метод расширения `Average` позволяет вычислить среднее арифметическое. Может потребоваться определить собственные методы расширения для вычисления моды и медианы.

Измените класс, чтобы его код выглядел следующим образом:

```
using System.Collections.Generic;

namespace System.Linq
{
  public static class MyLINQExtensions
  {
    // это цепной метод расширения linq
    public static IEnumerable<T> ProcessSequence<T>(
    this IEnumerable<T> sequence)
    {
      return sequence;
    }

    // это скалярные методы расширения linq
    public static int? Median(this IEnumerable<int?> sequence)
    {
      var ordered = sequence.OrderBy(item => item);
      int middlePosition = ordered.Count() / 2;
      return ordered.ElementAt(middlePosition);
    }

    public static int? Median<T>(
    this IEnumerable<T> sequence, Func<T, int?> selector)
    {
      return sequence.Select(selector).Median();
    }

    public static decimal? Median(this IEnumerable<decimal?> sequence)
    {
      var ordered = sequence.OrderBy(item => item);
      int middlePosition = ordered.Count() / 2;
      return ordered.ElementAt(middlePosition);
    }

    public static decimal? Median<T>(
    this IEnumerable<T> sequence, Func<T, decimal?> selector)
    {
      return sequence.Select(selector).Median();
    }

    public static int? Mode(this IEnumerable<int?> sequence)
    {
      var grouped = sequence.GroupBy(item => item);
      var orderedGroups = grouped.OrderBy(group => group.Count());
      return orderedGroups.FirstOrDefault().Key;
    }

    public static int? Mode<T>(
    this IEnumerable<T> sequence, Func<T, int?> selector)
```

```
    {
      return sequence.Select(selector).Mode();
    }

    public static decimal? Mode(this IEnumerable<decimal?> sequence)
    {
      var grouped = sequence.GroupBy(item => item);
      var orderedGroups = grouped.OrderBy(group => group.Count());
    return orderedGroups.FirstOrDefault().Key;
    }

    public static decimal? Mode<T>(
    this IEnumerable<T> sequence, Func<T, decimal?> selector)
    {
      return sequence.Select(selector).Mode();
    }
  }
}
```

Обратите внимание: метод расширения `ProcessSequence` не изменяет последовательность. Я создал его только для использования в качестве примера. Вам нужно обработать последовательность так, как захотите.

Для использования ваших методов расширения LINQ вам понадобится лишь ссылаться на сборку библиотеки класса, поскольку пространство имен `System.Linq`, как правило, уже импортировано.

В методе `Main` измените запрос LINQ для вызова вашего цепного метода следующим образом:

```
static void Main(string[] args)
{
  using (var db = new Northwind())
  {
    var query = db.Products
      .ProcessSequence()
      .Where(product => product.UnitPrice < 10M)
      .OrderByDescending(product => product.UnitPrice)
      .Select(product => new
      {
        product.ProductID,
        product.ProductName,
        product.UnitPrice
      });
```

Запустив консольное приложение, вы увидите тот же самый программный вывод, что и раньше, поскольку ваш метод не изменяет последовательность. Но зато теперь вы знаете, как расширять LINQ с помощью созданных вами функций.

Добавьте в конец метода `Main` инструкции для вывода среднего арифметического, медианы и моды для `UnitsInStock` и `UnitPrice` для товаров, используйте собственные методы расширения и встроенный метод расширения `Average`, как показано в следующем листинге:

```
WriteLine("Custom LINQ extension methods:");
WriteLine($" Mean units in stock:
```

```
{db.Products.Average(p => p.UnitsInStock):N0}");
WriteLine($" Mean unit price:
{db.Products.Average(p => p.UnitPrice):$#,##0.00}");
WriteLine($" Median units in stock:
{db.Products.Median(p => p.UnitsInStock)}");
WriteLine($" Median unit price:
{db.Products.Median(p => p.UnitPrice):$#,##0.00}");
WriteLine($" Mode units in stock:
{db.Products.Mode(p => p.UnitsInStock)}");
WriteLine($" Mode unit price:
{db.Products.Mode(p => p.UnitPrice):$#,##0.00}");
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Custom LINQ extension methods:
  Mean units in stock: 41
  Mean unit price: $28.87
  Median units in stock: 26
  Median unit price: $19.50
  Mode units in stock: 13
  Mode unit price: $22.00
```

Работа с LINQ to XML

LINQ to XML — это поставщик данных LINQ, позволяющий создавать запросы и управлять XML.

Генерация XML с помощью LINQ to XML

Откройте проект консольного приложения или каталог `LinqWithEFCore`.

В файле `Program.cs` импортируйте пространство имен `System.Xml.Linq`.

В нижней части метода `Main` допишите следующие инструкции:

```
var productsForXml = db.Products.ToArray();

var xml = new XElement("products",
  from p in productsForXml
  select new XElement("product",
  new XAttribute("id", p.ProductID),
  new XAttribute("price", p.UnitPrice),
  new XElement("name", p.ProductName)));

WriteLine(xml.ToString());
```

Запустите консольное приложение и проанализируйте результат вывода.

Обратите внимание: сгенерированный XML-код совпадает с элементами и атрибутами, декларативно описанными с помощью инструкций LINQ to XML в предыдущем листинге.

```
<products>
  <product id="1" price="18.0000">
    <name>Chai</name>
```

```
  </product>
  <product id="2" price="19.0000">
    <name>Chang</name>
  </product>
  <product id="3" price="10.0000">
    <name>Aniseed Syrup</name>
  </product>
```

Чтение XML с применением LINQ to XML

Можно воспользоваться технологией LINQ to XML для легкого создания запросов к файлам XML.

Добавьте в проект LinqWithEFCore XML-файл `settings.xml`. Измените его содержимое таким образом:

```
<?xml version="1.0" encoding="utf-8" ?>
  <appSettings>
    <add key="color" value="red" />
    <add key="size" value="large" />
    <add key="price" value="23.99" />
  </appSettings>
```

Вернувшись к классу `Program`, добавьте в него следующие инструкции, чтобы:

- загрузить файл XML;
- воспользоваться LINQ to XML для поиска элемента `appSettings` и его потомков `add`;
- проецировать XML-код в массив анонимного типа со свойствами `Key` и `Value`:
- перебрать массив для отображения результатов.

```
XDocument doc = XDocument.Load("settings.xml");

var appSettings =
doc.Descendants("appSettings").Descendants("add").Select(node => new
{
  Key = node.Attribute("key").Value,
  Value = node.Attribute("value").Value
}).ToArray();

foreach (var item in appSettings)
{
  WriteLine($"{item.Key}: {item.Value}");
}
```

Запустите консольное приложение и проанализируйте результат вывода:

```
color: red
size: large
price: 23.99
```

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенное упражнение и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Проверочные вопросы

1. Каковы две обязательные составные части LINQ?
2. Какой метод расширения LINQ вы бы использовали для возврата из типа набора свойств?
3. Какой метод расширения LINQ вы бы применили для фильтрации последовательности?
4. Перечислите шесть методов расширения LINQ, выполняющих консолидацию (укрупнение) данных.
5. Чем отличаются методы расширения `Select` и `SelectMany`?

Упражнение 12.1. Создание запросов LINQ

Создайте консольное приложение с именем `Exercise02`, запрашивающее у пользователя название города, а затем перечисляющее названия компаний для клиентов Northwind в заданном городе:

```
Enter the name of a city: London
There are 6 customers in London:
Around the Horn
B's Beverages
Consolidated Holdings
Eastern Connection
North/South
Seven Seas Imports
```

Усовершенствуйте приложение так, чтобы оно отображало список всех уникальных городов, в которых уже находятся клиенты, прежде чем пользователь введет название предпочитаемого города:

```
Aachen, Albuquerque, Anchorage, Århus, Barcelona, Barquisimeto, Bergamo, Berlin,
Bern, Boise, Bräcke, Brandenburg, Bruxelles, Buenos Aires, Butte, Campinas,
Caracas, Charleroi, Cork, Cowes, Cunewalde, Elgin, Eugene, Frankfurt a.M., Genève,
Graz, Helsinki, I. de Margarita, Kirkland, Kobenhavn, Köln, Lander, Leipzig, Lille,
Lisboa, London, Luleå, Lyon, Madrid, Mannheim, Marseille, México D.F., Montréal,
München, Münster, Nantes, Oulu, Paris, Portland, Reggio Emilia, Reims, Resende,
Rio de Janeiro, Salzburg, San Cristóbal, San  Francisco, Sao Paulo, Seattle,
Sevilla, Stavern, Strasbourg, Stuttgart, Torino, Toulouse, Tsawassen, Vancouver,
Versailles, Walla Walla, Warszawa
```

Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе.

- LINQ в C#: https://docs.microsoft.com/en-us/dotnet/articles/csharp/linq/linq-in-csharp.
- 101 пример использования LINQ: https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b.
- PLINQ: https://msdn.microsoft.com/en-us/library/dd460688(v=vs.110).aspx.
- LINQ to XML (C#): https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/concepts/linq/linq-to-xml.
- LINQPad — площадка для программистов на .NET: https://www.linqpad.net/.

Резюме

В этой главе вы изучили, как писать запросы LINQ, чтобы выбрать, проецировать, фильтровать, сортировать, объединять и группировать данные в различных форматах, в том числе XML, что, в общем, является каждодневными задачами.

В следующей главе вы используете тип `Task` для повышения производительности приложений.

13 Улучшение производительности и масштабируемости с помощью многозадачности

Эта глава посвящена способам одновременного выполнения нескольких действий, позволяющим повысить производительность, масштабируемость и эффективность работы конечных пользователей.

В данной главе:

- мониторинг производительности и использования ресурсов;
- процессы, потоки и задачи;
- асинхронное выполнение задач;
- синхронизация доступа к общим ресурсам;
- методы `async` и `await`.

Мониторинг производительности и использования ресурсов

Прежде чем мы сможем улучшить производительность некоего кода, следует научиться отслеживать его код и эффективность, чтобы зафиксировать базовую черту, от которой можем начать измерения.

Оценка эффективности типов

Какой тип лучше всего подойдет в той или иной ситуации? Чтобы ответить на этот вопрос, мы должны хорошенько понять, что же подразумеваем под словами «лучше всего».

Следует принять во внимание такие факторы:

- *функциональность* — определяется путем проверки того, предоставляет ли данный тип нужную вам функциональность;
- *объем памяти* — определяется с помощью проверки того, сколько байтов памяти занимает выбранный тип;
- *производительность* — определяется после проверки того, насколько быстр данный тип;
- *потребности в будущем* — зависят от изменений требований и возможности поддержки.

Бывает так, что несколько типов имеют одинаковую функциональность, поэтому, прежде чем сделать выбор, следует сопоставить расход памяти и производительность. Для сохранения миллионов чисел наиболее подходящим типом будет тот, который занимает меньше всего места в памяти. Если нужно сохранить лишь несколько чисел, но выполнить с ними большое количество операций, то лучше выбрать тип, который быстрее всего обрабатывается ЦПУ.

Ранее я упоминал функцию `sizeof()`, предназначенную для отображения количества байтов, занимаемых в памяти одним экземпляром конкретного типа. При сохранении большого количества значений в более сложных структурах данных, таких как массивы и списки, для измерения использования памяти требуется способ получше.

В Интернете и книгах можно прочитать множество советов, но единственный способ узнать наверняка, какой тип хорошо подходит для вашей программы, — сравнить их самостоятельно. В следующем разделе вы научитесь писать код для мониторинга действительных требований к памяти и действительных значений производительности при использовании различных типов.

Сегодня оптимальным выбором может стать переменная `short`, но, вероятно, лучше будет задействовать переменную `int`, хотя она занимает в два раза больше места в памяти, так как, возможно, в будущем потребуется хранить вдвое больший диапазон значений.

Следует принять во внимание еще один показатель — обслуживание. Данная мера показывает, сколько другому программисту потребуется приложить усилий, чтобы понять и отредактировать ваш код. Если вы делаете неочевидный выбор типов, то это может запутать программиста, открывшего ваш код, чтобы исправить ошибку или добавить какую-либо функциональность. Существуют различные инструменты для анализа и отображения того, насколько легко поддерживать ваш код.

Мониторинг производительности и использования памяти

В пространстве имен `System.Diagnostics` реализовано большое количество полезных типов для мониторинга вашего кода. Однако в первую очередь следует рассмотреть тип `Stopwatch`.

Visual Studio 2017

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте категорию Visual C# и выберите .NET Standard. В списке в центральной части окна выберите пункт Class Library (.NET Standard) (Библиотека классов (.NET Standard)), введите имя `MonitoringLib`, измените место сохранения на `C:\Code`, введите имя решения `Chapter13` и затем нажмите кнопку OK. Переименуйте файл `Class1.cs` в `Recorder.cs`.

В Visual Studio 2017 создайте новый проект консольного приложения с именем `MonitoringApp`.

Задайте текущий выбор в качестве стартового проекта вашего решения.

На панели Solution Explorer (Обозреватель решений) в проекте MonitoringApp щелкните правой кнопкой мыши на категории Dependencies (Зависимости) и в контекстном меню выберите пункт Add Reference (Добавить ссылку). Выберите проект MonitoringLib и нажмите кнопку OK.

Visual Studio Code

В Visual Studio Code в каталоге `Code` создайте папку `Chapter13`, в ней создайте два подкаталога: `MonitoringLib` и `MonitoringApp`.

В Visual Studio Code откройте каталог `MonitoringLib`.

На панели Integrated Terminal (Интегрированный терминал) введите следующую команду:

```
dotnet new classlib
```

Откройте каталог `MonitoringApp`.

На панели Integrated Terminal (Интегрированный терминал) введите такую команду:

```
dotnet new console
```

Откройте каталог `Chapter13`.

На панели EXPLORER (Проводник) раскройте категорию MonitoringLib и переименуйте файл `Class1.cs` в `Recorder.cs`.

В папке `MonitoringApp` откройте файл `MonitoringApp.csproj` и добавьте в него пакетную ссылку на библиотеку MonitoringLib, как показано полужирным шрифтом в следующем листинге:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
```

```
    <ProjectReference Include="..\MonitoringLib\MonitoringLib.csproj" />
  </ItemGroup>

</Project>
```

На панели Integrated Terminal (Интегрированный терминал) введите следующие команды:

```
cd MonitoringApp
dotnet build
```

Создание класса Recorder

В Visual Studio 2017 и Visual Studio Code откройте файл `Recorder.cs` и измените его содержимое, чтобы он выглядел так:

```
using System;
using System.Diagnostics;
using static System.Console;
using static System.Diagnostics.Process;

namespace Packt.CS7
{
  public static class Recorder
  {
    static Stopwatch timer = new Stopwatch();
    static long bytesPhysicalBefore = 0;
    static long bytesVirtualBefore = 0;

    public static void Start()
    {
      GC.Collect();
      GC.WaitForPendingFinalizers();
      GC.Collect();
      bytesPhysicalBefore = GetCurrentProcess().WorkingSet64;
      bytesVirtualBefore = GetCurrentProcess().VirtualMemorySize64;
      timer.Restart();
    }

    public static void Stop()
    {
      timer.Stop();
      long bytesPhysicalAfter = GetCurrentProcess().WorkingSet64;
      long bytesVirtualAfter = GetCurrentProcess().VirtualMemorySize64;
      WriteLine("Stopped recording.");
      WriteLine($"{bytesPhysicalAfter - bytesPhysicalBefore:N0}
      physical bytes used.");
      WriteLine($"{bytesVirtualAfter - bytesVirtualBefore:N0}
      virtual bytes used.");
      WriteLine($"{timer.Elapsed} time span ellapsed.");
      WriteLine($"{timer.ElapsedMilliseconds:N0}
      total milliseconds ellapsed.");
    }
  }
}
```

В методе Start класса Recorder применяется тип «сборщик мусора» (garbage collector, GC), позволяющий нам быть уверенными в том, что вся выделенная в настоящий момент память будет собрана до записи количества использованной памяти. Это сложная техника, и ее стоит избегать при создании промышленного кода.

В методе `Main` класса `Program` напишите инструкции для запуска и остановки `Recorder` при генерации массива из 10 000 целых чисел, как показано в следующем листинге:

```
using System.Linq;
using Packt.CS7;
using static System.Console;

namespace MonitoringApp
{
  class Program
  {
    static void Main(string[] args)
    {
      Write("Press ENTER to start the timer: ");
      ReadLine();
      Recorder.Start();
      int[] largeArrayOfInts = Enumerable.Range(1, 10000).ToArray();
      Write("Press ENTER to stop the timer: ");
      ReadLine();
      Recorder.Stop();
      ReadLine();
    }
  }
}
```

Вы создали класс `Recorder` с двумя методами для начала и остановки записи времени и памяти, примененных любым запущенным вами кодом. Метод `Main` начинает запись, когда пользователь нажимает клавишу Enter, далее создает массив из 10 000 переменных типа `int`, а затем прекращает запись, когда пользователь вновь нажимает ту же клавишу.

Для типа `Stopwatch` реализовано несколько полезных членов, перечисленных в табл. 13.1.

Таблица 13.1

Член	Описание
Метод Restart	Сбрасывает количество прошедшего времени до нуля и запускает секундомер
Метод Stop	Останавливает секундомер
Свойство Elapsed	Количество прошедшего времени в формате TimeSpan (часы:минуты:секунды)
Свойство ElapsedMilliseconds	Затраченное время в миллисекундах, сохраненное как длинное целое число

Для типа `Process` реализовано несколько полезных членов, перечисленных в табл. 13.2.

Таблица 13.2

Член	Описание
VirtualMemorySize64	Отображает в байтах количество виртуальной памяти, выделенной для процесса
WorkingSet64	Отображает в байтах количество физической памяти, выделенной для процесса

Запустите консольное приложение без отладчика. Оно начнет запись времени и памяти, когда вы нажмете клавишу Enter, а затем остановит запись, когда нажмете эту же клавишу вновь. Подождите несколько секунд между нажатиями клавиши Enter, как это сделали мы для получения следующего вывода:

```
Press ENTER to start the timer:
Press ENTER to stop the timer:
Stopped recording.
942,080 physical bytes used.
0 virtual bytes used.
00:00:03.1166037 time span ellapsed.
3,116 total milliseconds ellapsed.
```

Замер эффективности обработки строк

Теперь, когда вы увидели, как применять типы `Stopwatch` и `Process` для мониторинга кода, воспользуемся ими, чтобы оценить наилучший способ обработки строковых переменных.

Закомментируйте предыдущий код в методе `Main`, заключив его между символами `/* */`.

Добавьте в метод `Main` следующий код. Он создает массив из 50 000 переменных типа `int` и конкатенирует их, используя в качестве разделителей запятые, с помощью классов `string` и `StringBuilder`:

```
int[] numbers = Enumerable.Range(1, 50000).ToArray();
Recorder.Start();
WriteLine("Using string");
string s = "";
for (int i = 0; i < numbers.Length; i++)
{
  s += numbers[i] + ", ";
}
Recorder.Stop();
Recorder.Start();
WriteLine("Using StringBuilder");
var builder = new System.Text.StringBuilder();
for (int i = 0; i < numbers.Length; i++)
{
```

```
  builder.Append(numbers[i]);
  builder.Append(", ");
}
Recorder.Stop();
ReadLine();
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Using string
Stopped recording.
12,447,744 physical bytes used.
1,347,584 virtual bytes used.
00:00:03.4700170 time span ellapsed.
3,470 total milliseconds ellapsed.

Using StringBuilder
Stopped recording.
12,288 physical bytes used.
0 virtual bytes used.
00:00:00.0051490 time span ellapsed.
5 total milliseconds ellapsed.
```

Подытожим результаты следующим образом:

- класс `string` использовал около 12,5 Мбайт памяти, его обработка заняла примерно 3,5 секунды;
- класс `StringBuilder` задействовал около 12,3 Кбайт памяти, его обработка заняла приблизительно пять миллисекунд.

В этой ситуации класс `StringBuilder` примерно в 700 раз быстрее и при конкатенации текста использует память примерно в 1000 раз эффективнее!

Избегайте использования метода String.Concat или оператора + внутри циклов, как и применения строковой интерполяции C# $ внутри циклов. Такой код занимает меньше физической, но в два раза больше виртуальной памяти, кроме того, обрабатывается 30 секунд! Вместо вышеперечисленного задействуйте класс StringBuilder.

Теперь, когда вы научились измерять производительность и ресурсную эффективность вашего кода, изучим процессы, потоки и задания.

Процессы, потоки и задачи

Процесс, как, например, любое из созданных нами консольных приложений, обладает выделенным специально для него набором ресурсов, таких как память и потоки. *Поток* выполняет написанный вами код, инструкцию за инструкцией. По умолчанию каждый процесс имеет только один поток, что может привести к проблемам, если нужно выполнять несколько *задач* одновременно.

Потоки также отвечают за отслеживание таких моментов, как авторизация пользователей и любые правила глобализации, которые должны соблюдаться для текущего языка и региона.

Windows и большинство других современных операционных систем используют режим вытесняющей многозадачности, которая имитирует параллельное выполнение задач. Данный режим делит процессорное время между потоками, выделяя *«интервал времени»* для каждого потока один за другим. Текущий поток приостанавливается, когда заканчивается его «интервал времени». Затем процессор позволяет запустить другой поток в «интервале времени».

При переключении с одного потока на другой Windows сохраняет контекст потока и перезагружает ранее сохраненный контекст следующего в очереди потока. На это требуются время и ресурсы.

У потоков есть свойства `Priority` и `ThreadState`, а также класс `ThreadPool`, необходимый для управления пулом фоновых рабочих потоков (рис. 13.1).

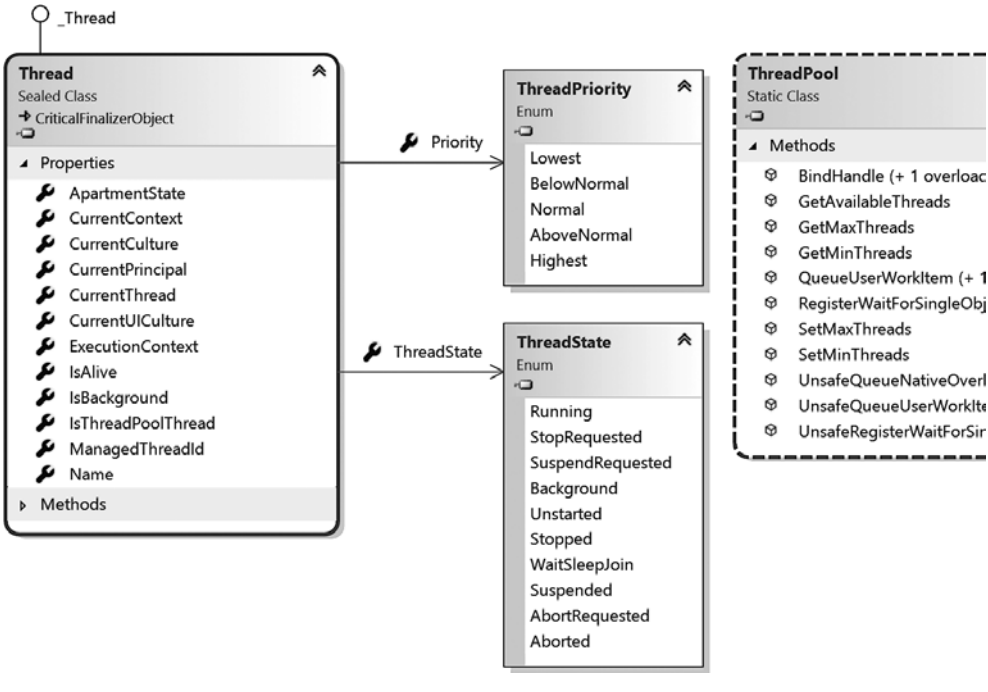


Рис. 13.1

Потоки могут конкурировать и ждать доступа к общим ресурсам, таким как переменные, файлы и объекты баз данных.

В зависимости от задачи удвоение количества потоков (рабочих) не уменьшает вдвое количество секунд, которое будет затрачено на выполнение задачи. Фактически это может даже *увеличить* длительность выполнения задачи (рис. 13.2).

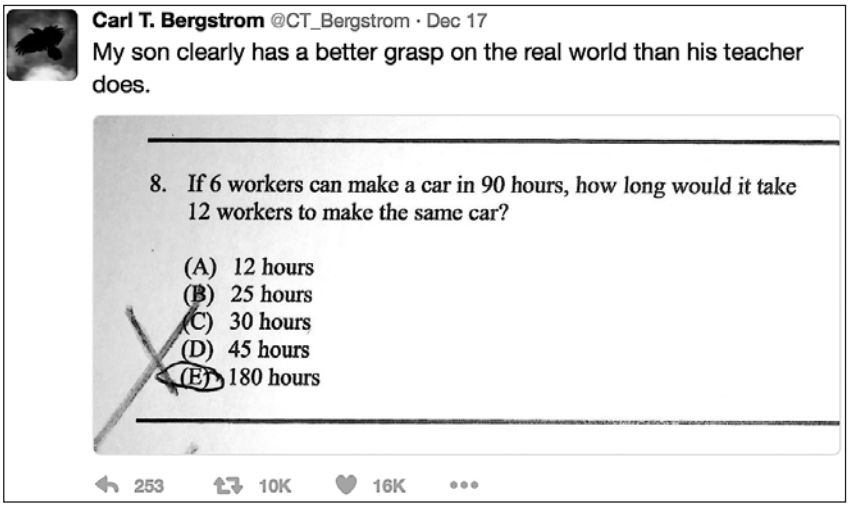


Рис. 13.2

Текст на картинке: «Мой сын, несомненно, имеет лучшее представление об окружающей действительности, чем его учитель. Задача: если 6 рабочих могут собрать машину за 90 часов, то как долго ту же машину будут собирать 12 рабочих? Варианты ответа: а) 12 часов; б) 25 часов; в) 30 часов; г) 45 часов; д) (выбран) 180 часов».

Никогда не полагайтесь на то, что увеличение количества потоков (рабочих) повысит производительность. Выполните тесты производительности сначала базового кода без реализации нескольких потоков, а затем — кода с несколькими потоками. Выполняйте эти тесты в промежуточном окружении, максимально приближенном к рабочей среде.

Асинхронное выполнение задач

Первым делом мы напишем простое консольное приложение, один за другим выполняющее три метода.

Синхронное выполнение нескольких действий

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ Add ▸ New project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Core. В центре диалогового окна выберите пункт Console App (.NET Core) (Консольное приложение (.NET Core)), присвойте ему имя `WorkingWithTasks`, укажите расположение по адресу `C:\Code`, введите имя решения `Chapter13`, а затем нажмите кнопку OK.

В Visual Studio Code создайте папку `Chapter13` с подкаталогом `WorkingWithTasks`, а затем откройте этот подкаталог. На панели Integrated Terminal (Интегрированный терминал) выполните команду `dotnet new console`.

В Visual Studio 2017 и Visual Studio Code убедитесь, что в коде импортируются следующие пространства имен и статические типы:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;
using static System.Console;
```

Как уже упоминалось, в программе будут использованы три метода, которые необходимо выполнить: первый занимает три секунды, второй — две, а третий — одну. Для имитации такого процесса можно применить класс `Thread`, позволяющий текущему потоку засыпать на заданное число миллисекунд.

Добавьте в класс `Program` код, приведенный ниже:

```
static void MethodA()
{
  WriteLine("Starting Method A...");
  Thread.Sleep(3000); // имитация работы в течение трех секунд
  WriteLine("Finished Method A.");
}

static void MethodB()
{
  WriteLine("Starting Method B...");
  Thread.Sleep(2000); // имитация работы в течение двух секунд
  WriteLine("Finished Method B.");
}

static void MethodC()
{
  WriteLine("Starting Method C...");
  Thread.Sleep(1000); // имитация работы в течение одной секунды
  WriteLine("Finished Method C.");
}
```

Добавьте в метод `Main` следующие инструкции:

```
static void Main(string[] args)
{
  var timer = Stopwatch.StartNew();

  WriteLine("Running methods synchronously on one thread.");
  MethodA();
  MethodB();
  MethodC();

  WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");

  WriteLine("Press ENTER to end.");
  ReadLine();
}
```

Вызовы WriteLine и ReadLine в конце метода Main, предлагающие пользователю нажать клавишу Enter, необходимы только в случае применения Visual Studio 2017 с отладчиком, поскольку эта среда автоматически прекратит выполнять приложение в конце метода Main! Вызывая ReadLine, консольное приложение функционирует до тех пор, пока пользователь не нажмет клавишу Enter. Если вы работаете с Visual Studio Code, то можете оставить обе инструкции.

Запустите консольное приложение, проанализируйте результат вывода и обратите внимание вот на что: так как используется только один поток, выполнение программы займет чуть более шести секунд:

```
Running methods synchronously on one thread.
Starting Method A...
Finished Method A.
Starting Method B...
Finished Method B.
Starting Method C...
Finished Method C.
6,047ms elapsed.
Press ENTER to end.
```

Асинхронное выполнение нескольких действий с помощью задач

Класс `Thread` был доступен уже в первой версии языка C# и годится для создания и управления новыми потоками, но с ним может быть сложно работать напрямую.

В версии C# 4 был добавлен класс `Task`, представляющий собой оболочку для потока, позволившую упростить создание и управление потоками. Обертывание нескольких потоков в задачи позволит коду выполняться асинхронно.

Класс `Task` имеет свойства `Status` и `CreationOptions`, а также метод `ContinueWith`, который можно настроить, прибегнув к перечислению `TaskContinuationOptions`, и управлять с помощью класса `TaskFactory` (рис. 13.3).

Рассмотрим три способа запуска методов с использованием экземпляров класса `Task`. Каждый из них несколько отличается синтаксисом, но все они определяют этот класс и запускают его.

Закомментируйте вызовы трех методов и связанное с ними консольное сообщение, а затем добавьте новые инструкции, как показано в следующем листинге:

```
static void Main(string[] args)
{
  var timer = Stopwatch.StartNew();
  //WriteLine("Running methods synchronously on one thread.");
  //MethodA();
  //MethodB();
  //MethodC();
  WriteLine("Running methods asynchronously on multiple threads.");
```

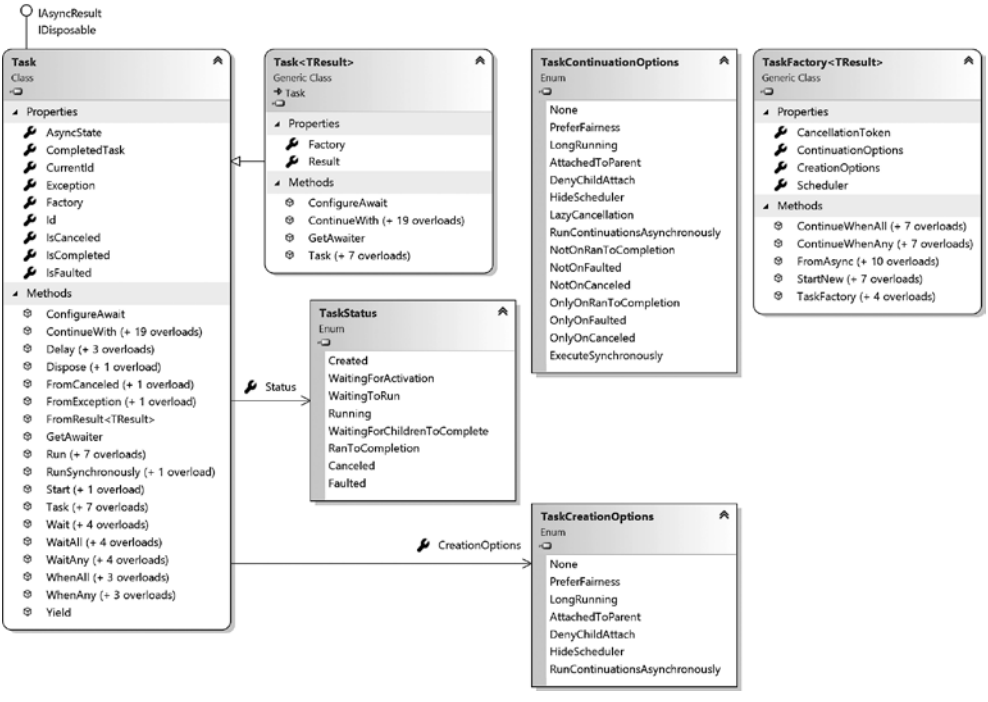


Рис. 13.3

```
  Task taskA = new Task(MethodA);
  taskA.Start();
  Task taskB = Task.Factory.StartNew(MethodB);
  Task taskC = Task.Run(new Action(MethodC));
  WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");
  WriteLine("Press ENTER to end.");
  ReadLine();
}
```

Запустите консольное приложение и проанализируйте результат вывода. Обратите внимание: количество миллисекунд, фактически затраченное на работу программы, зависит от производительности процессора в вашем компьютере, поэтому вы, вероятно, увидите другое значение, отличное от моего.

```
Running methods asynchronously on multiple threads.
Starting Method A...
Starting Method B...
Starting Method C...
23ms elapsed.
Press ENTER to end.
Finished Method C.
Finished Method B.
Finished Method A.
```

Значение затраченного времени выводится практически сразу, поскольку каждый из трех методов теперь выполняется тремя *новыми* потоками. *Исходный* поток продолжает выполняться до тех пор, пока не достигнет вызова `ReadLine` в конце метода `Main`.

Несмотря на то что три новых потока выполняют свой код одновременно, они запускаются в случайном порядке. Метод `MethodC` завершается первым, так как использует всего одну секунду процессорного времени, затем метод `MethodB` и, наконец, метод `MethodA`, поскольку затрачивает три секунды.

Тем не менее используемый центральный процессор существенно влияет на результат. Именно ЦП выделяет «интервалы времени» под каждый процесс, чтобы позволить им выполнять свои потоки. Вы не можете контролировать время работы или простоя того или иного метода.

Ожидание выполнения задач

Иногда требуется дождаться завершения задачи, прежде чем продолжать работу. Для этого используется метод `Wait` экземпляра класса `Task` или статические методы `WaitAll` и `WaitAny` для массива задач (табл. 13.3).

Таблица 13.3

Метод	Описание
t.Wait()	Ожидает завершения выполнения задачи t
Task.WaitAny(Task[])	Ожидает завершения выполнения любой задачи в массиве
Task.WaitAll(Task[])	Ожидает завершения выполнения всех задач в массиве

Добавьте следующие инструкции в метод `Main` сразу после кода создания трех задач и перед выводом затраченного времени. Так вы объедините в массив ссылки на три задачи и передадите его методу `WaitAll`. Теперь исходный поток будет останавливаться на этой инструкции, ожидая завершения всех трех задач, прежде чем выведет значение затраченного времени.

```
Task[] tasks = { taskA, taskB, taskC };
Task.WaitAll(tasks);
```

Перезапустите консольное приложение и проанализируйте результат вывода:

```
Running methods asynchronously on multiple threads.
Starting Method A...
Starting Method B...
Starting Method C...
Finished Method C.
Finished Method B.
Finished Method A.
3,024 milliseconds elapsed.
Press ENTER to end.
```

Обратите внимание: общее затраченное время теперь немного превышает время выполнения самого долгого метода. Если все три задачи могут выполняться одновременно, то это все, что нам нужно сделать.

Однако часто одна задача зависит от результата выполнения другой. Справиться с этой проблемой поможет определение *задач продолжения*.

Задачи продолжения

Добавьте следующие методы в класс `Program`:

```
static decimal CallWebService()
{
  WriteLine("Starting call to web service...");
  Thread.Sleep((new Random()).Next(2000, 4000));
  WriteLine("Finished call to web service.");
  return 89.99M;
}

static string CallStoredProcedure(decimal amount)
{
  WriteLine("Starting call to stored procedure...");
  Thread.Sleep((new Random()).Next(2000, 4000));
  WriteLine("Finished call to stored procedure.");
  return $"12 products cost more than {amount:C}.";
}
```

Эти методы имитируют вызов веб-сервиса, возвращающий денежную сумму, в дальнейшем применяемой для извлечения из базы данных количества товаров, цена которых превышает данную сумму. Результат, возвращаемый первым методом, должен быть передан в качестве входного параметра второго метода.

Чтобы имитировать работу, я воспользовался классом Random для ожидания каждого вызова метода со случайной длительностью в диапазоне от двух до четырех секунд.

В методе `Main` закомментируйте предыдущие задачи, обернув их код символами многострочного комментария `/* */`.

Затем добавьте следующие инструкции перед уже имеющейся инструкцией, которая выводит затраченное время, а затем вызывает метод `ReadLine` для ожидания нажатия пользователем клавиши Enter:

```
WriteLine("Passing the result of one task as an input into another.");

var taskCallWebServiceAndThenStoredProcedure =
Task.Factory.StartNew(CallWebService).
ContinueWith(previousTask => CallStoredProcedure(previousTask.Result));

WriteLine($"{taskCallWebServiceAndThenStoredProcedure.Result}");
```

Запустите консольное приложение и проанализируйте результат вывода.

```
Passing the result of one task as an input into another.
Starting call to web service...
Finished call to web service.
Starting call to stored procedure...
Finished call to stored procedure.
12 products cost more than £89.99.
5,971 milliseconds elapsed.
Press ENTER to end.
```

Вложенные и дочерние задачи

Создайте новый проект консольного приложения с именем `NestedAndChildTasks`.

На панели Solution Explorer (Обозреватель решений) в Visual Studio 2017 щелкните правой кнопкой мыши на решении и в контекстном меню выберите пункт Properties (Свойства). В категории Startup Project (Запускаемый проект) установите переключатель в положение Current selection (Текущий проект).

Убедитесь, что были импортированы следующие пространства имен и статические типы:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;
using static System.Console;
```

Добавьте в метод `Main` такие инструкции:

```
var outer = Task.Factory.StartNew(() =>
{
  WriteLine("Outer task starting...");
  var inner = Task.Factory.StartNew(() =>
  {
    WriteLine("Inner task starting...");
    Thread.Sleep(2000);
    WriteLine("Inner task finished.");
  });
});
outer.Wait();
WriteLine("Outer task finished.");
WriteLine("Press ENTER to end.");
ReadLine();
```

Запустите консольное приложение и проанализируйте результат вывода:

```
Outer task starting...
Outer task finished.
Inner task starting...
Inner task finished.
Press ENTER to end.
```

Обратите внимание: хотя мы ожидаем завершения внешней задачи, не должна заканчиваться и ее внутренняя задача. Связать две задачи поможет специальный параметр.

Измените существующий код, который определяет внутреннюю задачу, чтобы добавить перечисление `TaskCreationOption` с элементом `AttachedToParent`:

```
var inner = Task.Factory.StartNew(() =>
{
  WriteLine("Inner task starting...");
  Thread.Sleep(2000);
  WriteLine("Inner task finished.");
}, TaskCreationOptions.AttachedToParent);
```

Перезапустите консольное приложение и проанализируйте результат вывода. Обратите внимание: внутренняя задача должна завершиться прежде внешней:

```
Outer task starting...
Inner task starting...
Inner task finished.
Outer task finished.
Press ENTER to end.
```

Синхронизация доступа к общим ресурсам

При одновременном выполнении нескольких потоков существует вероятность того, что два и более потока могут одновременно обращаться к одной и той же переменной или к другому ресурсу и тем самым вызывать проблему.

По этой причине следует внимательно изучить способы превращения кода в *«потокобезопасный»*.

Простейший механизм обеспечения потоковой безопасности состоит в использовании в качестве *«флага»* или *«светофора»* объектной переменной, позволяющей указать, когда применяется общий ресурс с исключительной блокировкой.

В книге Уильяма Голдинга «Повелитель мух» (William Golding, Lord of the Flies) Хрюша (Piggy) и Ральф (Ralph) находят раковину и используют ее как рог, чтобы созывать детей на сбор. Мальчики принимают «правило рога» и договариваются о том, что говорить имеет право только тот, кто держит в руках рог. Мне пришлась по душе идея назвать объектную переменную тем самым «рогом», который нужно держать в руках при разговоре, — conch. Когда у потока есть «рог», ни один другой поток не может обращаться к общему ресурсу (-ам), представленному (-ым) этим «рогом».

Мы рассмотрим ряд типов, которые можно использовать для синхронизации доступа к ресурсам (рис. 13.4).

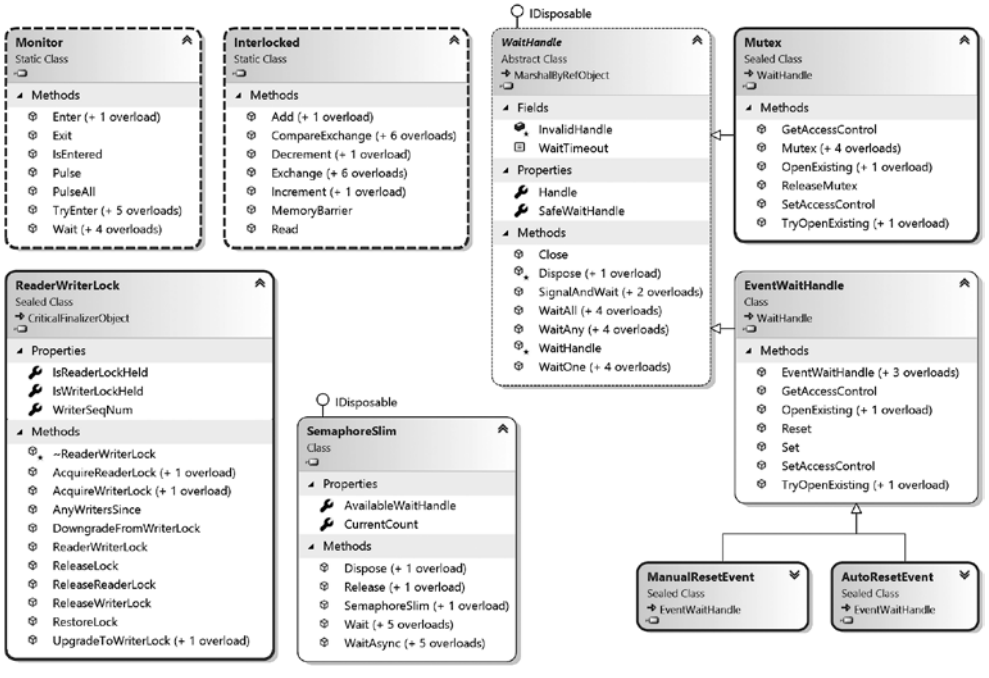


Рис. 13.4

Доступ к ресурсу из нескольких потоков

Создайте новый проект консольного приложения с именем `SynchronizingResourceAccess`.

Убедитесь, что импортируются следующие пространства имен и статические типы:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;
using static System.Console;
```

Добавьте в класс `Program` инструкции для реализации таких действий, как:

- объявление и создание экземпляра объекта для генерации случайного значения времени ожидания;
- объявление строковой переменной для хранения сообщения (общий ресурс);
- объявление двух методов, пять раз в цикле добавляющих букву `A` или `B` к общей строковой переменной и ожидающих каждую итерацию в течение случайного интервала времени в пределах до двух секунд;

- метод `Main`, который выполняет оба метода в отдельных потоках с использованием пары задач и ожидает их завершения, после чего выводит значение затраченного времени в миллисекундах.

```
static Random r = new Random();
static string Message; // общий ресурс

static void MethodA()
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(r.Next(2000));
    Message += "A";
    Write(".");
  }
}

static void MethodB()
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(r.Next(2000));
    Message += "B";
    Write(".");
  }
}

static void Main(string[] args)
{
  WriteLine("Please wait for the tasks to complete.");
  Stopwatch watch = Stopwatch.StartNew();

  Task a = Task.Factory.StartNew(MethodA);
  Task b = Task.Factory.StartNew(MethodB);

  Task.WaitAll(new Task[] { a, b });
  WriteLine();
  WriteLine($"Results: {Message}.");
  WriteLine($"{watch.ElapsedMilliseconds:#,##0} elapsed
  milliseconds.");
}
```

Запустите консольное приложение и проанализируйте результат вывода.

```
Please wait for the tasks to complete.
..........
Results: BABBABBAAA.
6,099 elapsed milliseconds.
```

Обратите внимание: в результате оба потока меняли сообщение одновременно. В реальном приложении это может стать проблемой. Предотвратить одновременный доступ можно с помощью взаимоисключающей блокировки.

Применение к ресурсу взаимоисключающей блокировки

Добавьте в код класса `Program` экземпляр объектной переменной, выступающей в роли *«рога»*:

```
static object conch = new object();
```

Добавьте в код обоих методов, `MethodA` и `MethodB`, инструкцию `lock`, окружив ею инструкцию `for`:

```
lock (conch)
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(r.Next(2000));
    Message += "A";
    Write(".");
  }
}
```

Перезапустите консольное приложение и проанализируйте результат вывода:

```
Please wait for the tasks to complete.
..........
Results: AAAAABBBBB.
9,751 elapsed milliseconds.
```

Хотя было затрачено довольно много времени, только один метод одновременно мог получить доступ к общему ресурсу. Только после того, как метод завершил работу с общим ресурсом, «рог освобождается» и другой метод получает возможность выполнять свою функцию.

Первым можно запустить как метод MethodA, так и MethodB.

Инструкция lock

У вас может возникнуть вопрос: как работает инструкция `lock`, когда блокирует переменную объекта? Компилятор заменит код, показанный ниже:

```
lock(conch)
{
  // доступ к общему ресурсу
}
```

на следующий код:

```
try
{
  Monitor.Enter(conch);
  // доступ к общему ресурсу
```

```
}
finally
{
  Monitor.Exit(conch);
}
```

Понимание принципа работы инструкции `lock` важно, поскольку ее использование может привести к взаимоблокировке.

Взаимоблокировка возникает при наличии двух или более общих ресурсов (и, соответственно, «рога»), после чего события происходят в таком порядке:

- поток X блокирует «рог» A;
- поток Y блокирует «рог» B;
- поток X пытается заблокировать «рог» B, но блокируется, поскольку тот уже есть у потока Y;
- поток Y пытается заблокировать «рог» A, но блокируется, поскольку тот уже есть у потока X.

Проверенный способ предотвратить взаимоблокировки заключается в указании тайм-аута при попытке получить блокировку. Чтобы реализовать такое поведение, нужно вручную использовать класс `Monitor` вместо инструкции `lock`.

Измените код, заменив инструкции `lock` командами входа в «рог» со временем ожидания, как показано ниже:

```
try
{
  Monitor.TryEnter(conch, TimeSpan.FromSeconds(15));
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(r.Next(2000));
    Message += "A";
    Write("."); }
  }
}
finally
{
  Monitor.Exit(conch);
}
```

Перезапустите консольное приложение и проанализируйте результат вывода. Вы должны увидеть те же результаты, что и раньше, и код при этом улучшится, поскольку сможет избежать потенциальных взаимоблокировок.

Никогда не применяйте ключевое слово lock. Всегда используйте метод Monitor.TryEnter в сочетании с инструкцией try, чтобы настроить время ожидания и избежать потенциальной взаимоблокировки.

Выполнение атомарных операций

Взгляните на данную операцию инкремента:

```
int x = 3;
x++; // это атомарная операция?
```

Атомарный: от греч. atomos — «неделимое».

Нет, не атомарная! Инкремент целого числа требует выполнения центральным процессором следующих трех операций.

1. Загрузка значения из экземпляра переменной в регистр.
2. Инкремент (увеличение) значения.
3. Сохранение значения в экземпляре переменной.

Поток может быть выгружен после выполнения первых двух шагов. Затем второй поток может выполнить все три шага. Когда первый поток возобновит выполнение, он перезапишет значение переменной, а результат инкремента или декремента, выполняемый вторым потоком, будет утерян!

Существует тип `Interlocked`, позволяющий выполнять атомарные действия для типов значений, таких как целые числа и числа с плавающей запятой.

Объявите другой общий ресурс, который будет подсчитывать количество выполненных операций:

```
static int Counter; // другой общий ресурс
```

В коде обоих методов в инструкции `for` после изменения значения `string` добавьте следующую инструкцию для безопасного инкремента счетчика:

```
Interlocked.Increment(ref Counter);
```

После вывода затраченного времени отобразите значение счетчика:

```
WriteLine($"{Counter} string modifications.");
```

Перезапустите консольное приложение и проанализируйте результат вывода:

```
10 string modifications.
```

Использование других типов синхронизации

Классы `Monitor` и `Interlocked` — взаимоисключающие блокировки, простые и эффективные; но иногда для синхронизации доступа к общим ресурсам требуются более сложные параметры (табл. 13.4).

Таблица 13.4

Тип	Описание
ReaderWriterLock и ReaderWriterLockSlim (рекомендуется)	Позволяют использовать несколько потоков в режиме чтения, при этом один поток должен находиться в режиме записи с исключительным правом блокировки, а другой поток с доступом для чтения должен быть в обновляемом режиме чтения, из которого поток может перейти в режим записи без необходимости отказываться от доступа для чтения
Mutex	Подобно Monitor, класс Mutex обеспечивает эксклюзивный доступ к общему ресурсу, за исключением того, что используется для синхронизации между процессами
Semaphore и SemaphoreSlim	Эти классы ограничивают количество потоков, которые могут одновременно обращаться к ресурсу или пулу ресурсов, определяя слоты
AutoResetEvent и ManualResetEvent	Обработчики ожидания событий позволяют потокам синхронизировать действия, сигнализируя друг другу и ожидая сигналов друг от друга

Методы async и await

В версии C# 5 были введены два ключевых слова для упрощения работы с типом `Task`. Они особенно полезны таких ситуациях, как:

- реализация многозадачности для *графического пользовательского интерфейса (GUI)*;
- улучшение масштабируемости веб-приложений и веб-сервисов.

Из глав 15 и 16 вы узнаете, как ключевые слова `async` и `await` позволяют улучшить масштабируемость сайтов, веб-сервисов и веб-приложений.

А в главах 17 и 18 мы изучим, как те же ключевые слова позволяют реализовать многозадачность в GUI-приложениях, запускаемых на универсальной платформе Windows и Xamarin.

В текущей главе изучим теорию: для чего были добавлены эти ключевые слова в C#, а затем попрактикуемся в работе с ними.

Увеличение скорости отклика консольных приложений

Одно из ограничений консольных приложений в том, что ключевое слово `await` можно использовать только в методах, помеченных как `async...`, при этом C# 7 и более ранние версии не позволяют методу `Main` быть `async`!

К счастью, одно из нововведений C# 7.1 — поддержка `async` для метода `Main`.

Создайте новое консольное приложение и назовите его `AsyncConsole`.

Импортируйте пространства имен `System.Net.Http` и `System.Net.Http`, а также статически импортируйте пространство имен `System.Console`, как показано в следующем листинге:

```
using System.Net.Http;
using System.Threading.Tasks;
using static System.Console;
```

Добавьте в метод `Main` инструкции для создания экземпляра `HttpClient`, запроса главной страницы Apple и вывода размера в байтах, как показано в коде ниже:

```
var client = new HttpClient();
HttpResponseMessage response = await
client.GetAsync("http://www.apple.com/");
WriteLine($"Apple's home page has
{response.Content.Headers.ContentLength:N0} bytes.");
```

Соберите проект и обратите внимание на сообщение об ошибке, как показано в этом выводе:

```
Program.cs(12,44): error CS4033: The 'await' operator can only be used within an
async method. Consider marking this method with the 'async' modifier and changing
its return type to 'Task'.
[/Users/markjprice/Code/Chapter13/AsyncConsole/AsyncConsole.csproj]
```

Добавьте ключевое слово `async` в метод `Main`, измените возвращаемый тип на `Task`, постройте проект и обратите внимание на сообщение об ошибке, показанное в выводе ниже:

```
Program.cs(10,22): error CS8107: Feature 'async main' is not available in C# 7.
Please use language version 7.1 or greater.
[/Users/markjprice/Code/Chapter13/AsyncConsole/AsyncConsole.csproj]
CSC : error CS5001: Program does not contain a static 'Main' method suitable for an
entry point
[/Users/markjprice/Code/Chapter13/AsyncConsole/AsyncConsole.csproj]
```

Измените файл `AsyncConsole.csproj`, чтобы указать используемую версию языка C# 7.1 (выделено полужирным шрифтом в следующем листинге):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <LangVersion>7.1</LangVersion>
  </PropertyGroup>

</Project>
```

Постройте проект и обратите внимание, что в этот раз все прошло успешно.

Запустите консольное приложение и проанализируйте результат вывода:

```
Apple's home page has 42,740 bytes.
```

Увеличение скорости отклика приложений с GUI

Пока что мы создавали только консольные приложения. Жизнь программиста усложняется, когда речь заходит о создании веб-приложений, веб-сервисов или приложений с графическим интерфейсом пользователя, например UWP или мобильных приложений.

Одна из причин трудностей заключается в том, что для приложений с GUI есть отдельный поток: поток *пользовательского интерфейса*.

При работе с GUI нужно соблюдать два правила:

- не решать задачи с длительным временем выполнения в потоке UI;
- не получать доступ к элементам пользовательского интерфейса ни в каком потоке, кроме потока UI.

Соблюдение этих правил требовало от программистов написания сложного кода, позволяющего быть уверенными в том, что задачи с длительным временем выполнения решаются не в потоке UI, но по завершении результаты выполнения этих задач безопасно передаются в этот поток и отображаются пользователю. Такой код мог очень быстро стать запутанным! К счастью, начиная с версии языка C# 5, можно применять ключевые слова `async` и `await`. Они позволяют писать код так, будто он синхронный, что обеспечивает создание чистого и понятного кода, но на заднем плане компилятор C# создает сложную машину состояний и отслеживает запущенные потоки. Это похоже на магию!

Улучшение масштабируемости веб-приложений и веб-сервисов

Ключевые слова `async` и `await` можно применять и на стороне сервера при создании сайтов, веб-приложений и сервисов. С точки зрения пользовательского приложения ничего не меняется (или же пользователи могут заметить небольшое увеличение времени возврата ответа на запрос). Таким образом, с точки зрения одного пользователя, эти ключевые слова, применяемые для реализации многозадачности на стороне сервера, результат использования!

На стороне сервера же дополнительные более дешевые потоки создаются затем, чтобы ожидать завершение задач с длительным временем выполнения, таким образом, что дорогие потоки ввода/вывода могут обрабатывать запросы других пользователей и не блокироваться. Это улучшает общую масштабируемость веб-приложений или сервисов, ведь одновременно можно поддерживать бóльшее количество клиентов.

Часто используемые типы, поддерживающие многозадачность

Далее приведено несколько распространенных типов с асинхронными методами, которые вы можете ожидать (табл. 13.5):

Таблица 13.5

Тип	Методы
DbContext<T>	AddAsync, AddRangeAsync, FindAsync и SaveChangesAsync
DbSet<T>	AddAsync, AddRangeAsync, ForEachAsync, SumAsync, ToListAsync, ToDictionaryAsync, AverageAsync и CountAsync
HttpClient	GetAsync, PostAsync, PutAsync, DeleteAsync и SendAsync
StreamReader	ReadAsync, ReadLineAsync и ReadToEndAsync
StreamWriter	WriteAsync, WriteLineAsync и FlushAsync

Каждый раз, когда встречаете метод, оканчивающийся суффиксом Async, проверяйте, возвращает ли такой метод Task или Task<T>. Если возвращает, то стоит использовать именно этот метод вместо синхронного метода без указанного выше суффикса. Однако не забудьте вызвать такой метод с помощью ключевого слова await и дополнить вызов ключевым словом async.

Ключевое слово await в блоках catch

В версии C# 5 ключевое слово `await` можно было использовать только в блоке обработки исключений `try`, но не в `catch`. В версии C# 6 и более поздней можно применять `await` как в блоках `try`, так и в `catch`.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам, приведенным в данной главе.

Проверочные вопросы

1. Какую информацию о процессе вы можете выяснить?
2. Насколько точен тип `Stopwatch`?
3. Какой суффикс по соглашению должен быть применен к методу, возвращаемому `Task` или `Task<T>`?
4. Какое ключевое слово следует применить к объявлению метода, чтобы в нем можно было использовать ключевое слово `await`?
5. Как создать дочернюю задачу?
6. Почему не стоит использовать ключевое слово `lock`?
7. В каких случаях нужно применить класс `Interlocked`?
8. Когда класс `Mutex` следует использовать вместо класса `Monitor`?

9. Позволяет ли применение ключевых слов `async` и `await` улучшить производительность веб-приложений? Если нет, то зачем их использовать?
10. Можно ли отменить задачу? Как?

Дополнительные ресурсы

- Потоки и поточность: https://docs.microsoft.com/en-us/dotnet/standard/threading/ threads-and-threading.
- Подробный обзор асинхронного программирования: https://docs.microsoft.com/en-us/dotnet/standard/asyncin-depth.
- Ключевое слово `await` (справочник по C#): https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/await.
- Параллельное программирование в .NET Framework: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/.
- Обзор примитивов синхронизации: https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives.

Резюме

В этой главе вы узнали, как определять и запускать задачи, дожидаться выполнения одной задачи или нескольких и управлять порядком завершения задач. Вы также узнали, для чего предназначены ключевые слова `async` и `await` и как синхронизировать доступ к общим ресурсам.

В следующей части вы научитесь создавать программы для моделей приложений, поддерживаемых .NET Core (веб-приложения, веб-сервисы, приложения Universal Windows Platform) и .NET Standard (Xamarin).

Часть III
Модели приложений

Эта часть книги посвящена *моделям приложений*: платформам для создания законченных приложений, таких как сайты, веб-сервисы, веб-приложения, Windows- и мобильные приложения. Поскольку данная книга посвящена языку C# 7.1 и .NET Core 2.0 (и .NET Standard 2.0), то мы сосредоточимся на моделях приложений, в которых вышеупомянутые технологии используются для реализации большей части решения.

Сайты состоят из нескольких веб-страниц, загружаемых статически из файловой системы или генерируемых динамически силами технологий на стороне сервера, например, ASP.NET Core. Браузер создает запросы `GET`, используя уникальные для каждой страницы URL, а также работает с данными, хранящимися на сервере, с помощью запросов `POST`, `PUT` и `DELETE` (рис. III.1).

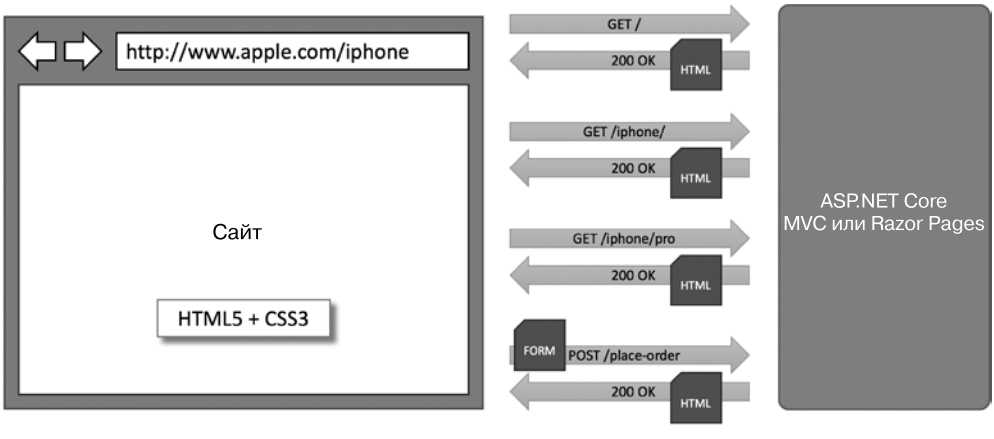


Рис. III.1

Применительно к *сайтам* браузер воспринимается как презентационный слой, притом что большая часть данных обрабатывается на стороне сервера. Небольшое количество кода на языке JavaScript можно использовать на стороне клиента для реализации функций представления, например показа слайдов.

Веб-приложения состоят из одной веб-страницы и веб-сервиса, к которым технологии стороны клиента, такие как Angular или React, могут совершать запросы для дальнейшего взаимодействия и обмена данными, используя общедоступные форматы сериализации, например XML и JSON (рис. III.2).

В случае с веб-приложениями, называемыми *одностраничными* (Single Page Application, SPA), на стороне клиента для реализации более сложных схем взаимодействия с пользователем применяются более сложные библиотеки JavaScript, такие как Angular или React. Однако все самые важные задачи обработки данных по-прежнему производятся на стороне сервера, поскольку браузеры имеют весьма ограниченный доступ к локальным системным ресурсам.

Начиная с 2018 года можно использовать сокеты ASP.NET Core Sockets (новое название для SignalR) для реализации веб-приложений с функцией обмена сообщениями в режиме реального времени, однако данная технология не была готова к моменту релиза платформы ASP.NET Core 2.0.

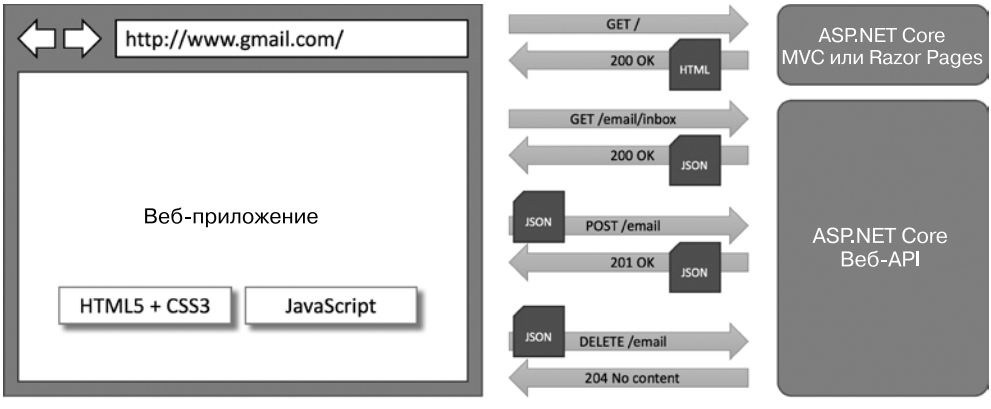


Рис. III.2

Приложения Windows могут быть запущены только на платформе Windows 10 с помощью C# и совместимы с различными устройствами от телефонов и планшетов до настольных и портативных компьютеров, консоли XBOX и устройств дополненной реальности. Такие приложения способны существовать сами по себе, но обычно вызывают различные веб-сервисы, чтобы обеспечить взаимодействие всех имеющихся у вас устройств (рис. III.3).

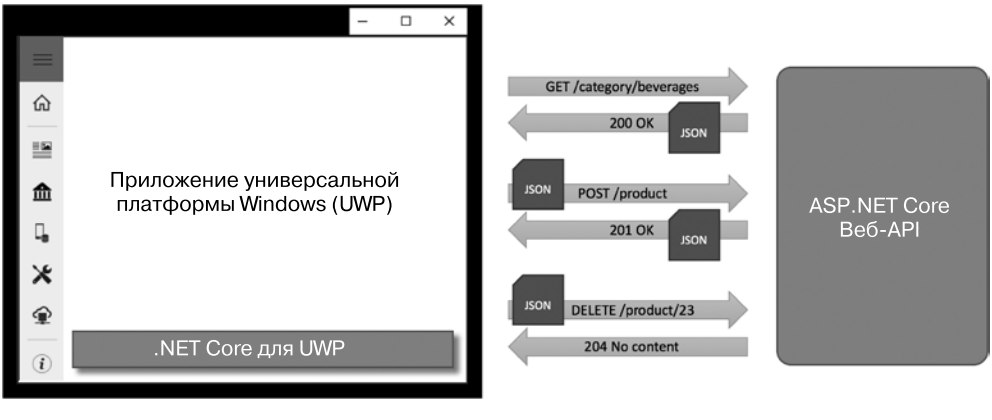


Рис. III.3

В случае с приложениями для Windows клиентская сторона может предоставлять очень сложные схемы взаимодействия с пользователем и имеет полный доступ к локальным ресурсам системы. Таким образом, приложению требуется серверная сторона, только если в нем нужно реализовать кросс-платформенную функциональность, например создание документа на планшете, продолжение работы с этим документом на настольном компьютере или сохранение прогресса в игре при смене устройства.

Мобильные приложения могут быть созданы на платформе Xamarin.Forms с помощью C#, а затем запускаться в iOS, Android и других операционных системах.

Такие приложения способны существовать сами по себе, но обычно вызывают различные веб-сервисы, чтобы обеспечить взаимодействие всех имеющихся у вас мобильных устройств (рис. III.4).

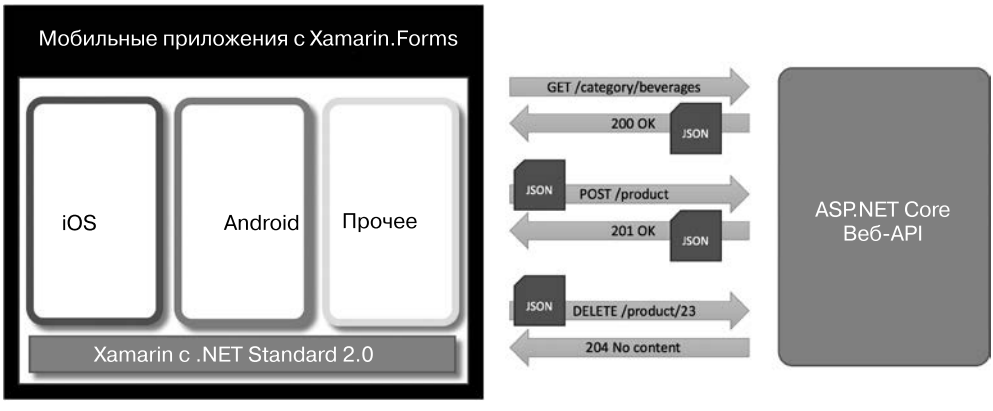


Рис. III.4

Мобильные приложения имеют те же преимущества, что и приложения для Windows, за одним исключением: первые не только кросс-устройственные, но и кросс-платформенные. Технологии для разметки пользовательского интерфейса, такие как XAML, могут применяться совместно с приложениями Windows.

Корпорация Microsoft разместила обширное руководство по реализации моделей приложений, таких как веб-приложения ASP.NET, мобильные приложения Xamarin и приложения UWP в документации .NET Application Architecture Guidance (Руководство по архитектуре приложений .NET), которое можно прочитать, перейдя по ссылке https://www.microsoft.com/net/learn/architecture.

В следующих главах вы научитесь создавать:

- простые сайты с помощью Razor Pages;
- сложные сайты, используя MVC;
- веб-сервисы и одностраничные приложения, применяя веб-API в первом случае и Angular во втором;
- приложения Windows с помощью XAML и Fluent Design;
- мобильные приложения, задействуя Xamarin.Forms.

14 Создание сайтов с помощью ASP.NET Core Razor Pages

Эта глава посвящена созданию сайтов с помощью современной HTTP-архитектуры на стороне сервера с применением Microsoft ASP.NET Core. Вы научитесь проектировать простые сайты, используя новую функцию ASP.NET Core 2.0 под названием Razor Pages.

В данной главе:

- веб-разработка;
- ASP.NET Core;
- технология Razor Pages;
- использование Entity Framework Core совместно с ASP.NET Core.

Веб-разработка

Разработка для Всемирной паутины — это разработка с HTTP.

Протокол передачи гипертекста

Для связи с веб-сервером клиент (он же *пользовательский агент*) совершает вызовы через Интернет, задействуя протокол, известный как *протокол передачи гипертекста* (Hypertext Transfer Protocol, HTTP). Это своего рода технический фундамент *Всемирной паутины*. Таким образом, говоря о веб-приложениях или веб-сервисах, мы имеем в виду, что они используют HTTP для связи между клиентом (обычно браузером) и сервером.

Клиент отправляет HTTP-запрос ресурса, например страницы, идентифицируемой по *URL* (Uniform Resource Locator, единый указатель ресурса), и сервер отправляет HTTP-ответ (рис. 14.1).

Для записи запросов и ответов можно использовать Google Chrome и другие браузеры.

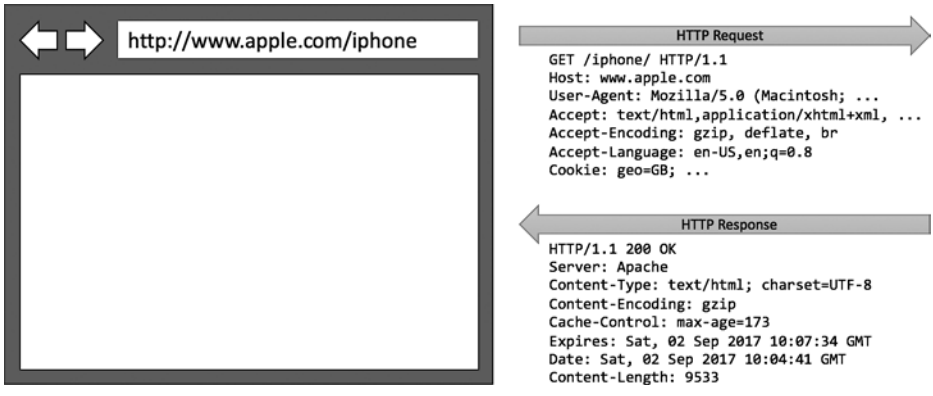


Рис. 14.1

Google Chrome доступен для большего числа операционных систем, чем любой другой браузер, и обладает мощными встроенными инструментами разработчика, поэтому лучше всего подойдет начинающему специалисту. Всегда тестируйте свое веб-приложение в Chrome и хотя бы в двух других браузерах, к примеру Firefox и Microsoft Edge для Windows 10 и Safari для macOS.

Запустите браузер *Google Chrome*. Чтобы отобразить инструменты разработчика, выполните следующие действия:

- в операционной системе macOS нажмите сочетание клавиш Alt+Cmd+I;
- в операционной системе Windows нажмите клавишу F12 или сочетание клавиш Ctrl+Shift+I.

Перейдите на вкладку Network (Сеть). Chrome должен немедленно начать запись сетевого трафика между вашим браузером и веб-серверами, к которым вы подключаетесь (рис. 14.2).

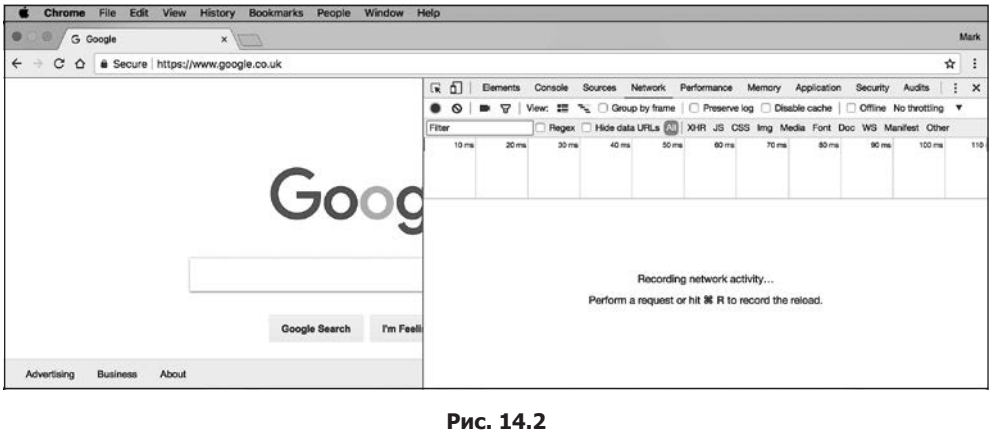


Рис. 14.2

В адресную строку браузера Chrome введите адрес https://www.asp.net/get-started и нажмите клавишу Enter.

На панели Developer Tools (Инструменты разработчика) в списке записанных запросов выделите первую запись (рис. 14.3).

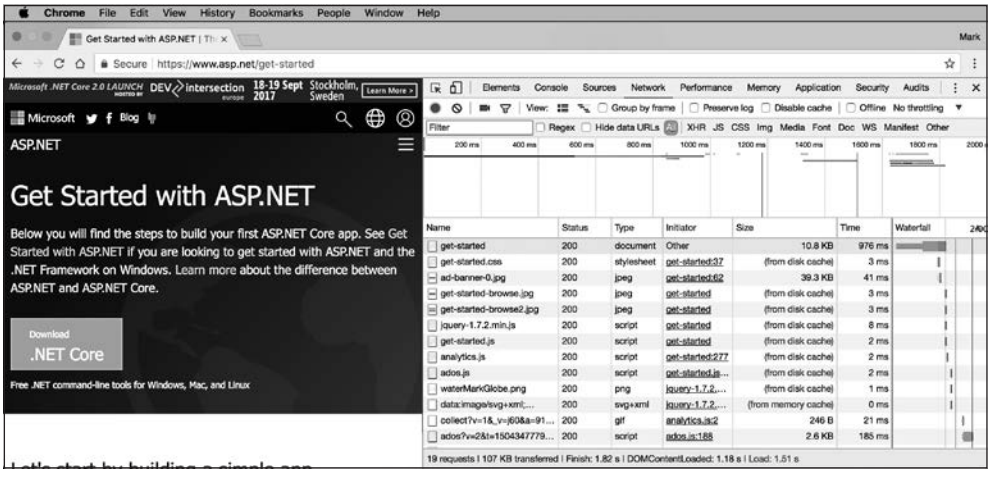


Рис. 14.3

В правой части панели щелкните на вкладке Headers (Заголовки), и вы увидите подробную информацию о запросе и ответе (рис. 14.4).

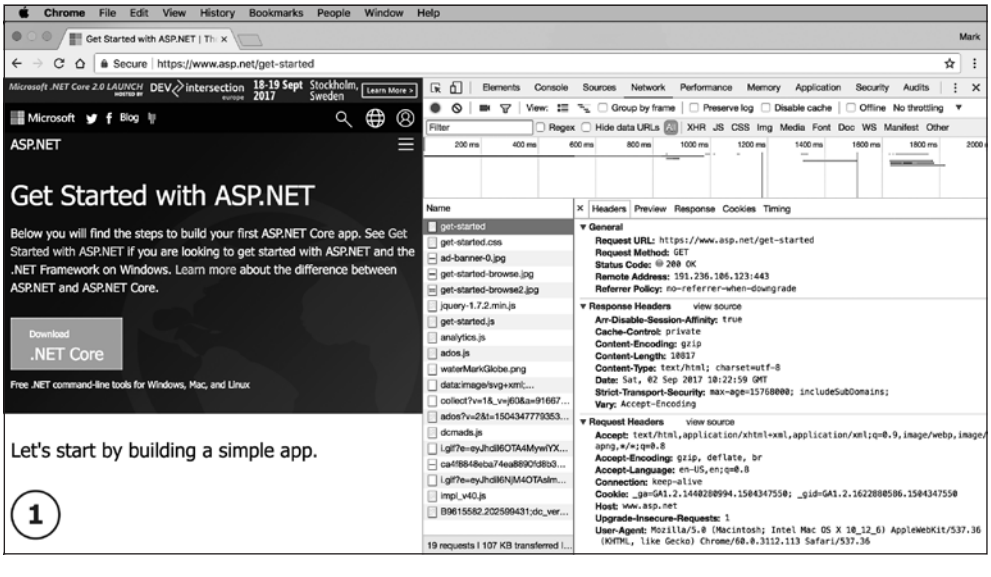


Рис. 14.4

Обратите внимание на следующие моменты.

- Применен *метод запроса* `GET`. Другие методы, используемые протоколом HTTP, включают `POST`, `PUT`, `DELETE` и `PATCH`.
- Получен *код состояния* `200 OK`. Это значит, что сервер нашел ресурс, который запросил браузер. Кроме того, встречается такой код, как `404 Missing`.
- *Заголовки запроса* включают параметр Accept, перечисляющий форматы ресурсов, принимаемые браузером. В этом случае браузер сообщает, что поддерживает форматы HTML, XHTML, XML и др.
- Заголовок Accept-Encoding указывает на следующее: браузер сообщил серверу, что поддерживает алгоритмы сжатия GZIP и DEFLATE.
- Браузер также сообщил серверу, какие человеческие языки он предпочитает: американский английский, а затем любой диалект английского языка (с качественным значением `0,8`).
- Я посещал данный сайт раньше, поэтому отображается строка Cookie сервиса Google Analytics со значением `_ga`, передаваемая на сервер, чтобы тот мог отслеживать мои действия.
- Сервер отправил обратно ответ, сжатый с помощью алгоритма GZIP, поскольку знает, что клиент способен распаковать этот формат.

Клиентская веб-разработка

При создании веб-приложений разработчику необходимо знать больше, чем просто C# и .NET Core. На стороне клиента (то есть в браузере) вы будете использовать следующую комбинацию технологий:

- *HTML5* для разметки структуры и контента веб-страницы;
- *CSS3* для настройки стилей форматирования и применения их к элементам веб-страницы;
- *JavaScript* для выполнения сценариев на веб-странице.

Несмотря на то что HTML5, CSS3 и JavaScript — фундаментальные компоненты клиентской веб-разработки, существуют много библиотек, которые делают ее более продуктивной, в том числе описанные ниже.

- *Bootstrap* — библиотека CSS3 для реализации отзывчивого дизайна веб-страниц. Дополнительно данная технология может использовать jQuery для добавления некоторых расширенных динамических функций.
- *jQuery* — популярная библиотека JavaScript.
- *TypeScript* — язык, созданный в Microsoft и используемый Google для добавления функциональности типов данных C# языку JavaScript. В процессе сборки файлы TypeScript (`*.ts`) могут быть скомпилированы в файлы JavaScript (`*.js`).

- *Angular* — популярная библиотека для создания *одностраничных приложений* (single page application, SPA) с помощью TypeScript.
- *React* — еще одна популярная библиотека для разработки SPA.
- *Redux* — библиотека для управления состоянием.

В качестве элемента процесса сборки и развертывания вы, скорее всего, будете использовать некое сочетание следующих технологий:

- *Node.js* — библиотека JavaScript, работающая на стороне сервера;
- *NPM* — это *менеджер узловых пакетов* (Node Package Manager), ставший де-факто менеджером пакетов для JavaScript и многих других модулей веб-разработки;
- *Bower* — клиентский менеджер пакетов для Всемирной паутины;
- *Gulp* — набор инструментов для автоматизации сложных и времязатратных задач;
- *Webpack* — популярный модульный упаковщик, инструмент для компиляции, трансформации и упаковки исходного кода приложения.

Данная книга посвящена C#, поэтому я рассмотрю лишь некоторые основы клиентской веб-разработки, а для получения более подробной информации рекомендую прочитать следующие книги: HTML5 Web Application Development By Example (https://www.packtpub.com/web-development/html5-web-application-development-example-beginners-guide) и ASP.NET Core and Angular 2 (https://www.packtpub.com/application-development/aspnet-core-and-angular-2).

Чтобы упростить работу с технологиями HTML5, CSS3 и JavaScript, среды Visual Studio 2017 и Visual Studio Code поддерживают следующие возможности:

- расширения Мадса Кристенсена (Mads Kristensen) для Visual Studio: https://marketplace.visualstudio.com/search?term=publisher%3A%22Mads%20Kristensen%22&target=VS&sortBy=Relevance;
- программирование на HTML в Visual Studio Code: https://code.visualstudio.com/Docs/languages/html;
- расширения корпорации Microsoft для Visual Studio Code: https://marketplace.visualstudio.com/search?term=publisher%3A%22Microsoft%22&target=VSCode&sortBy=Relevance.

Мадс Кристенсен написал одно из самых популярных расширений, Web Essentials, для пользователей среды разработки Visual Studio версии 2010 и выше. Последняя версия расширения раздроблена на небольшие модули, которые можно устанавливать по отдельности.

ASP.NET Core

Microsoft ASP.NET Core — составная часть технологий корпорации Microsoft, используемых для разработки веб-приложений и сервисов, которые эволюционировали на протяжении многих лет.

- *ASP* (Active Server Pages). Технология, увидевшая свет в 1996 году и ставшая первой попыткой корпорации Microsoft создать платформу для динамического выполнения кода веб-приложений на стороне сервера. ASP-файлы написаны на языке VBScript.
- *ASP.NET Web Forms*. Эта технология была представлена в 2002 году вместе с платформой .NET Framework. Она предназначена для того, чтобы разработчики, незнакомые с веб-программированием, но знающие такие языки, как Visual Basic, могли быстро создавать веб-приложения, перетаскивая и удаляя визуальные компоненты и записывая код событий в среде Visual Basic или C#. Хоть приложения, написанные с помощью Web Forms, могут размещаться только на серверах под управлением операционной системы Windows, они по-прежнему востребованы для таких продуктов, как Microsoft SharePoint. При разработке новых веб-проектов от данной технологии лучше отказаться в пользу платформы ASP.NET Core.
- *WCF* (Windows Communication Foundation). Платформа, выпущенная в 2006 году и позволяющая разработчикам создавать сервисы SOAP и REST. SOAP — технология мощная, но сложная, и поэтому ее следует избегать, если не нужны расширенные функции, такие как распределенные транзакции и сложные топологии обмена сообщениями.
- *ASP.NET MVC*. Платформа, выпущенная в 2009 году и реализующая модель четкого разделения задач веб-разработчиков между *моделями*, представляющими данные, *представлениями*, показывающими эти данные, и *контроллерами*, которые осуществляют выборку модели и передачу ее представлению. Такое разделение положительно влияет на возможность многократно использовать код и подвергать его модульному тестированию.
- *ASP.NET Web API*. Инструмент, выпущенный в 2012 году и позволяющий разработчикам создавать HTTP-сервисы REST, которые масштабируются проще и лучше, чем SOAP-сервисы.
- *ASP.NET SignalR*. Технология, выпущенная в 2013 году и реализующая методы коммуникации в реальном времени в веб-приложениях путем абстрагирования вспомогательных технологий и методов, таких как «веб-сокеты» и «длинные запросы».
- *ASP.NET Core*. Увидела свет в 2016 году и объединяет технологии MVC, Web API и SignalR на платформе .NET Core, благодаря чему является кроссплатформенной. В ASP.NET Core 2.0 было добавлено много шаблонов, чтобы упростить разработку кода как серверной части приложения с помощью .NET Core, так и клиентской части путем использования Angular, React или любой другой фронтенд-технологии.

Для разработки веб-приложений и сервисов выбирайте платформу ASP.NET Core, поскольку она включает в себя три веб-технологии, современные и кросс-платформенные.

Платформа ASP.NET Core совместима со стандартом .NET Standard 2.0, соответственно, может быть запущена на .NET Framework 4.6.1 и более новых версиях (только для Windows), а также на .NET Core 2.0 или более новых версиях (кросс-платформенно).

Классический ASP.NET и современный ASP.NET Core

Технологии ASP.NET стукнуло 15 лет в 2017 году. Это уже не дитя, а настоящий подросток!

До сих пор она строилась поверх крупной сборки `System.Web.dll` на платформе .NET Framework. За прошедшие годы в сборке скопилось множество функций, многие из которых не подходят для современной кросс-платформенной разработки.

ASP.NET Core — попытка глобального пересмотра платформы ASP.NET. Основной принцип заключается в удалении зависимостей от сборки `System.Web.dll` и реализации платформы в виде модульных легковесных пакетов, подобно остальным частям .NET Core.

Разрабатывать и запускать приложения ASP.NET Core можно независимо от платформы, в операционной системе Windows, macOS и Linux. Корпорация Microsoft даже создала кросс-платформенный, суперпроизводительный веб-сервер *Kestrel*. Весь стек имеет полностью открытый исходный код и предназначен для интеграции с различными инструментами и средами на стороне клиента, включая Bower, Gulp, Grunt, AngularJS, jQuery и Bootstrap.

Для удобства все проекты в части III будут располагаться в папке Part3, а не в отдельных папках по главам. Я бы рекомендовал вам прорабатывать примеры из глав части III последовательно, так как проекты из более поздних глав будут ссылаться на проекты из более ранних.

Создание проекта ASP.NET Core в Visual Studio 2017

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) раскройте категорию Visual C# и выберите пункт .NET Core. В центре диалогового окна выберите пункт ASP.NET Core Web Application (Веб-приложение ASP.NET Core), введите имя проекта `NorthwindWeb`, укажите расположение `C:\Code`, введите имя решения `Part3`, а затем нажмите кнопку OK.

В диалоговом окне New ASP.NET Core Web Application — NorthwindWeb (Создать веб-приложение ASP.NET Core — NorthwindWeb) выберите пункты раскрывающихся меню .NET Core и ASP.NET Core 2.0, после чего щелкните на шаблоне Empty (Пустой). Флажок Enable Docker Support (Включить поддержку Docker) должен быть снят. Нажмите кнопку OK (рис. 14.5).

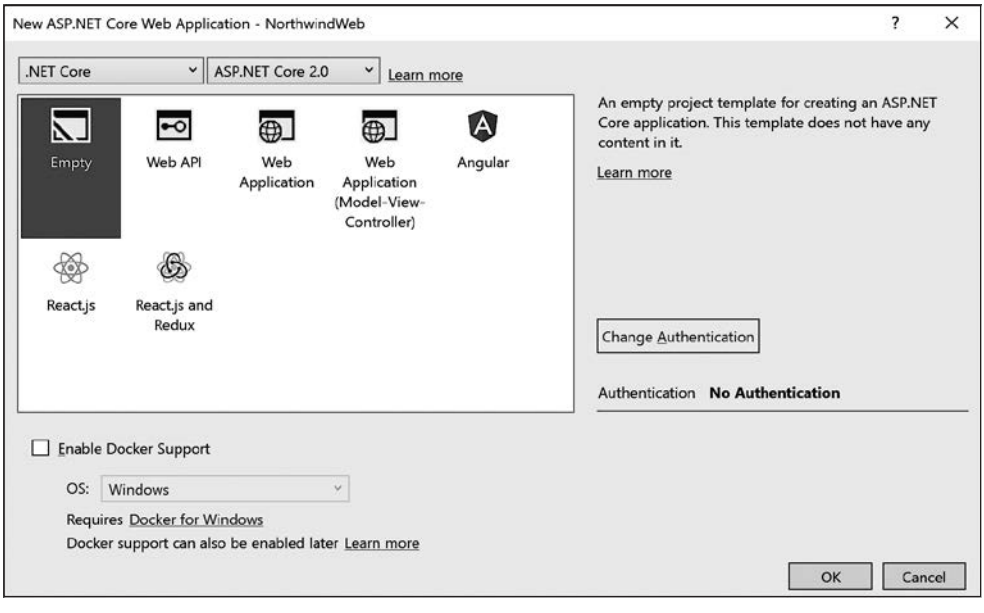


Рис. 14.5

Создание проекта ASP.NET Core в Visual Studio Code

Создайте каталог `Part3` и подкаталог `NorthwindWeb`.

В Visual Studio Code откройте каталог `NorthwindWeb`. На панели Integrated Terminal (Интегрированный терминал) введите следующую команду для создания пустого сайта ASP.NET Core:

```
dotnet new web
```

Обзор шаблона пустого проекта ASP.NET Core Empty

В Visual Studio 2017 и Visual Studio Code отредактируйте файл `NorthwindWeb.csproj` и обратите внимание на следующее:

- целевая среда — .NET Core 2.0;
- во время публикации будет включено содержимое каталога `wwwroot\`;

- единственная ссылка на пакет `Microsoft.AspNetCore.All`. В предыдущих версиях ASP.NET Core пришлось бы включать много ссылок. В ASP.NET Core 2.0 можно использовать эту специальную ссылку:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>

</Project>
```

Microsoft.AspNetCore.All — это метапакет, содержащий все пакеты ASP.NET Core, Entity Framework Core (EF Core), а также все зависимости ASP.NET Core и EF Core. Более подробную информацию о хранилище пакетов среды выполнения .NET Core можно получить, перейдя по ссылке https://docs.microsoft.com/en-us/dotnet/core/deploying/runtimestore.

Откройте файл `Program.cs` и обратите внимание вот на что:

- он похож на консольное приложение со статическим методом `Main`;
- `WebHost` указывает на класс запуска приложения. Это необходимо для настройки сайта:

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

  public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .Build();
}
```

Узнать, что именно делает метод CreateDefaultBuilder, можно, перейдя по ссылке https://github.com/aspnet/MetaPackages/blob/dev/src/Microsoft.AspNetCore/WebHost.cs.

Откройте файл `Startup.cs` и обратите внимание на следующие аспекты:

- метод `ConfigureServices` используется для подключения сервисов к основному контейнеру;
- метод `Configure` служит для настройки конвейера обработки запросов HTTP. На данный момент этот метод выполняет два действия: 1) в процессе создания кода все необрабатываемые исключения будут отображаться в браузере, чтобы создатель программы мог видеть более подробную информацию о них; 2) метод запускается, ожидает запросы и отвечает на все запросы отображением текстовой строки `Hello World!`:

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
  }

  public void Configure(
  IApplicationBuilder app, IHostingEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    app.Run(async (context) =>
    {
      await context.Response.WriteAsync("Hello World!");
    });
  }
}
```

Тестирование пустого веб-сайта

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+F5 или выполните команду Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки).

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) введите команду `dotnet run`.

В Visual Studio 2017 запустится браузер и создаст запрос к сайту (рис. 14.6).

В Visual Studio 2017 выполните команду View ▸ Output (Вид ▸ Вывод) для отображения вывода из ASP.NET Core Web Server и обратите внимание на обработку запросов (рис. 14.7).

В Visual Studio Code обратите внимание на сообщения на панели Integrated Terminal (Интегрированный терминал), приведенные далее:

```
Hosting environment: Production
Content root path: /Users/markjprice/Code/Chapter14/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

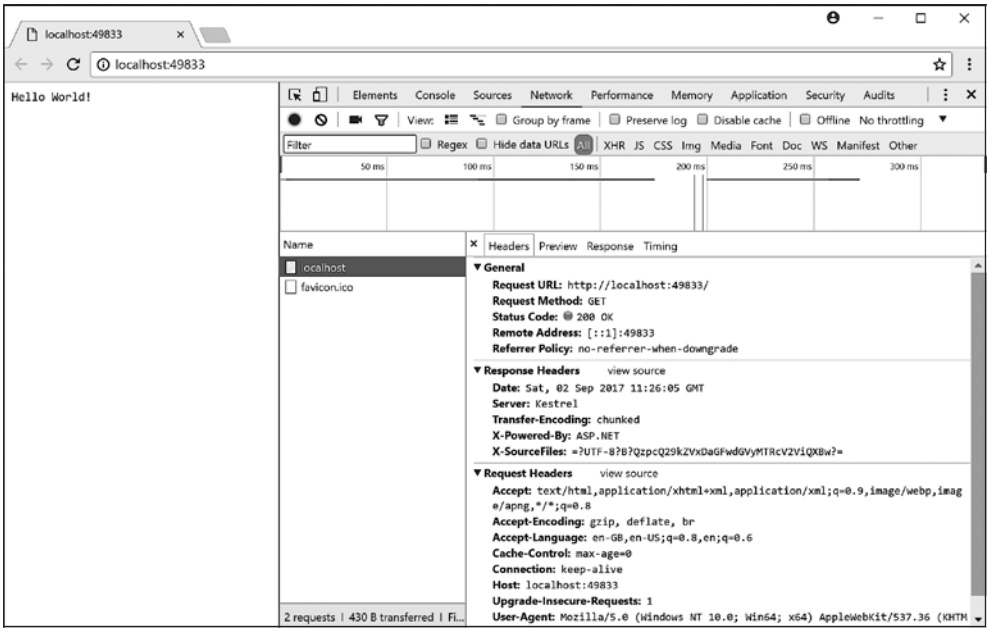


Рис. 14.6

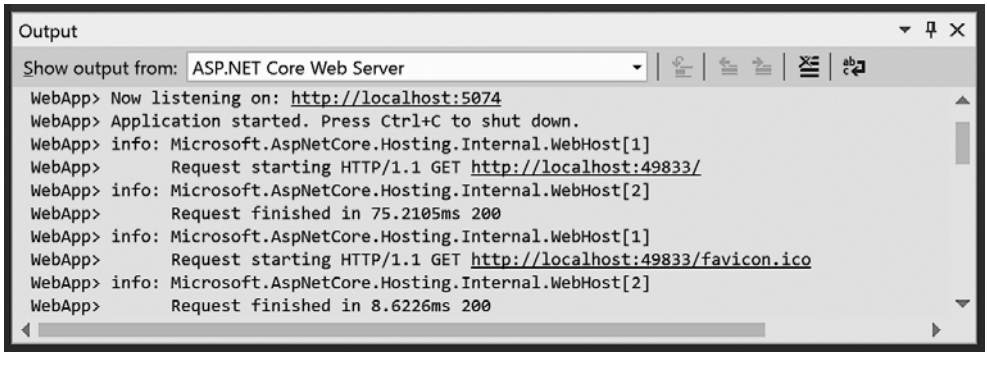


Рис. 14.7

Если используете Visual Studio Code, то должны запустить браузер самостоятельно.

Запустите Chrome, введите адрес http://localhost:5000/ и обратите внимание: вы получите тот же самый ответ, что показан на предыдущих снимках экрана для Visual Studio 2017.

В Visual Studio Code нажмите сочетание клавиш Ctrl+C для остановки веб-сервера. Не забывайте делать это каждый раз по окончании тестирования сайтов.

Закройте браузер.

Включение статических файлов

Сайт, который возвращает только текст без форматирования, на самом деле бесполезен! Как минимум сайт должен возвращать классические HTML-страницы, таблицы стилей CSS, используемые на веб-страницах, и любые другие статические ресурсы, например изображения и видео.

В Visual Studio 2017 и Visual Studio Code добавьте новый файл `index.html` в каталог `wwwroot` и измените его содержимое так, чтобы файл ссылался на Bootstrap, размещенный на CDN, и использущий современные практические рекомендации, например, настройки окна просмотра (viewport), как показано в листинге ниже.

Убедитесь в том, что поместили файл index.html в каталог wwwroot!

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-
fit=no" />
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css"
    integrity="sha384-
/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M"
    crossorigin="anonymous" />
  <title>Welcome ASP.NET Core!</title>
</head>
<body>
  <div class="container">
    <div class="jumbotron">
      <h1 class="display-3">Welcome to Northwind!</h1>
      <p class="lead">We supply products to our customers.</p>
      <hr />
      <p>Our customers include restaurants, hotels, and cruise lines.</p>
      <p>
      <a class="btn btn-primary"
        href="https://www.asp.net/">Learn more</a>
      </p>
    </div>
  </div>
</body>
</html>
```

Чтобы разобраться в классах Bootstrap, использованных мной для стилизации этой страницы, прочитайте документацию на сайте https://getbootstrap.com/docs/4.0/components/jumbotron/.

Если бы вы открыли сайт, введя в адресную строку `/index.html`, то сайт по-прежнему вернул бы простой текст `Hello World!`.

Чтобы сайт мог возвращать статические файлы, такие как `index.html`, нужно явно настроить эти функции.

В файле `Startup.cs` закомментируйте инструкции метода `Configure`, возвращающие ответ `Hello World!`, а также добавьте инструкцию для включения возможности возврата статических файлов, как показано в следующем листинге:

```
public void Configure(
  IApplicationBuilder app, IHostingEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }

    // app.Run(async (context) =>
    // {
    //     await context.Response.WriteAsync("Hello World!");
    // });

  app.UseStaticFiles();
}
```

Запустите сайт и обратите внимание на сообщение об ошибке `404` (рис. 14.8).

При использовании Visual Studio Code вы должны самостоятельно запустить браузер и ввести адрес http://localhost:5000.

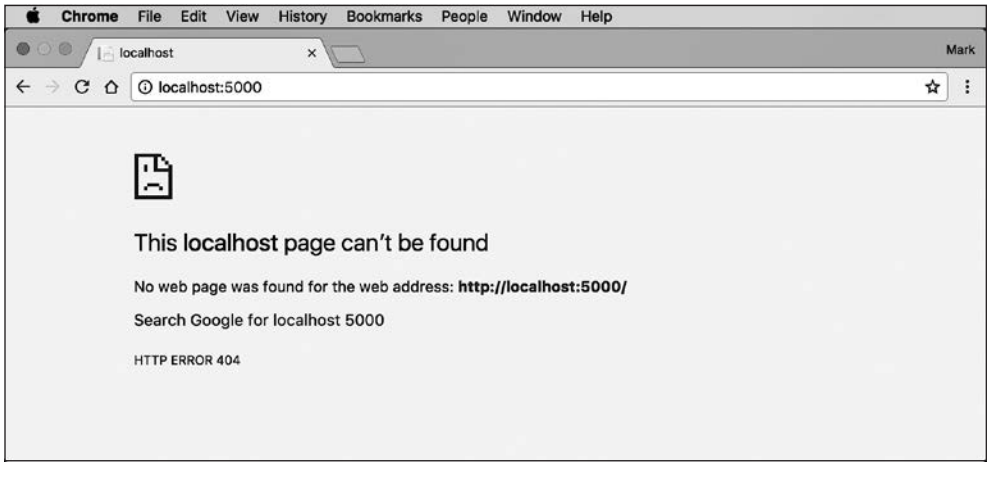


Рис. 14.8

В строке адреса введите `/index.html` и убедитесь, что браузер находит и возвращает страницу `index.html` (рис. 14.9).

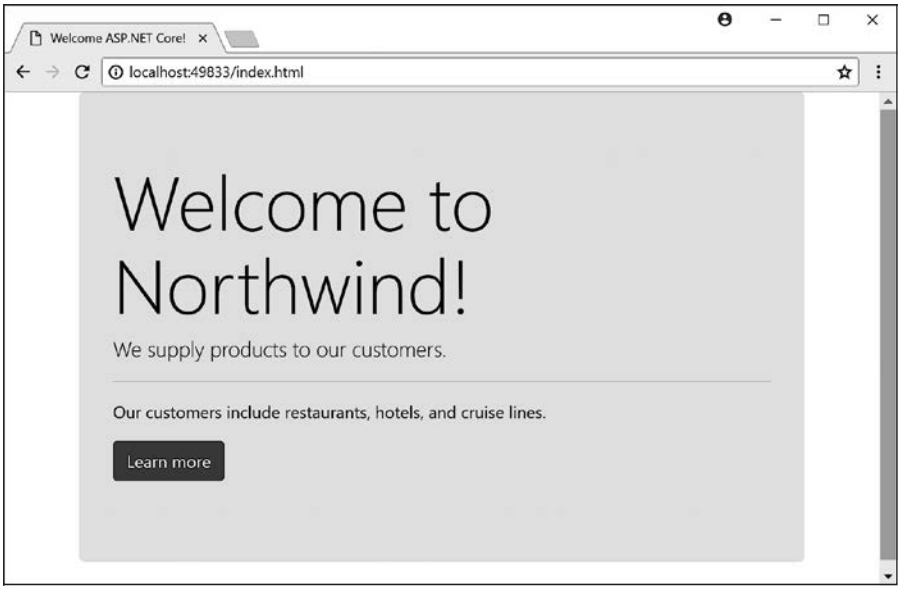


Рис. 14.9

Включение файлов по умолчанию

Было бы гораздо лучше, если бы файл `index.html` возвращался в качестве ответа по умолчанию и не приходилось бы вводить его имя.

Добавьте в файл `Startup.cs` инструкцию в метод `Configure`, позволяющую использовать файлы по умолчанию, как показано в коде ниже:

```
app.UseDefaultFiles(); // index.html, default.html и т. д.
```

Вызов метода UseDefaultFiles должен следовать перед вызовом UseStaticFiles, в противном случае метод не сработает!

Запустите сайт и обратите внимание, что файл `index.html` возвращался моментально, так как это страница по умолчанию для данного сайта.

Если все страницы сайта статические, то есть изменяются только вручную веб-редактором, значит, наша работа как веб-программистов завершена. Однако почти всем сайтам необходим динамический контент, например веб-страница, генерируемая во время работы сайта путем выполнения программного кода. Самый простой способ добиться этого результата — использовать новую функцию ASP.NET Core под названием *Razor Pages*.

Технология Razor Pages

Данная технология позволяет разработчику запросто смешивать разметку HTML с инструкциями программного кода на C#. Вот почему для страниц, созданных по этой технологии, применяется расширение `.cshtml`. Синтаксис Razor обозначается символом `@`.

В проекте NorthwindWeb создайте каталог `Pages`.

Среда ASP.NET Core по умолчанию ищет страницы Razor в каталоге Pages.

Переместите файл `index.html` в каталог `Pages` и замените его расширение с `.html` на `.cshtml`.

Включение страниц Razor

Чтобы включить страницы Razor, нужно сначала добавить и включить сервис MVC, так как Razor Pages — его составная часть.

Добавьте в файле `Startup.cs` в метод `ConfigureServices` инструкции для подключения сервиса MVC, как показано в следующем листинге:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
}
```

В файле `Startup.cs` в метод `Configure` сразу после уже прописанных инструкций, позволяющих использовать файлы по умолчанию и статические файлы, добавьте инструкцию для применения сервиса MVC, как показано в листинге ниже:

```
app.UseDefaultFiles(); // index.html, default.html и т.д.
app.UseStaticFiles();
app.UseMvc(); // подключение Razor Pages
```

Определение страницы Razor

Страницы Razor можно описать так:

- требуют директивы `@page` в начале файла;
- могут иметь раздел `@functions`, в котором можно определить следующее:
  - свойства для сохранения значений данных наподобие классов;
  - методы с именами `OnGet`, `OnPost`, `OnDelete` и т. п., которые выполняются при создании запросов HTTP, например `GET`, `POST` и `DELETE`.

Измените файл `index.cshtml`, добавив в его верхнюю часть инструкции на языке C#, чтобы определить данный файл как страницу Razor с помощью директивы `@page`.

Кроме того, определите свойство для хранения названия текущего дня недели и метод для установки этого свойства при передаче странице HTTP-запроса `GET`, как показано в следующем листинге:

```
@page
@functions {
  public string DayName { get; set; }
  public void OnGet()
  {
    Model.DayName = DateTime.Now.ToString("dddd");
  }
}
```

Измените файл `index.cshtml` для вывода названия дня недели в одном из абзацев, как показано в коде ниже:

```
<p>It's @Model.DayName! Our customers include restaurants, hotels, and cruise
lines.</p>
```

Запустите сайт и обратите внимание на выведенное название дня недели в нижней части страницы (рис. 14.10).

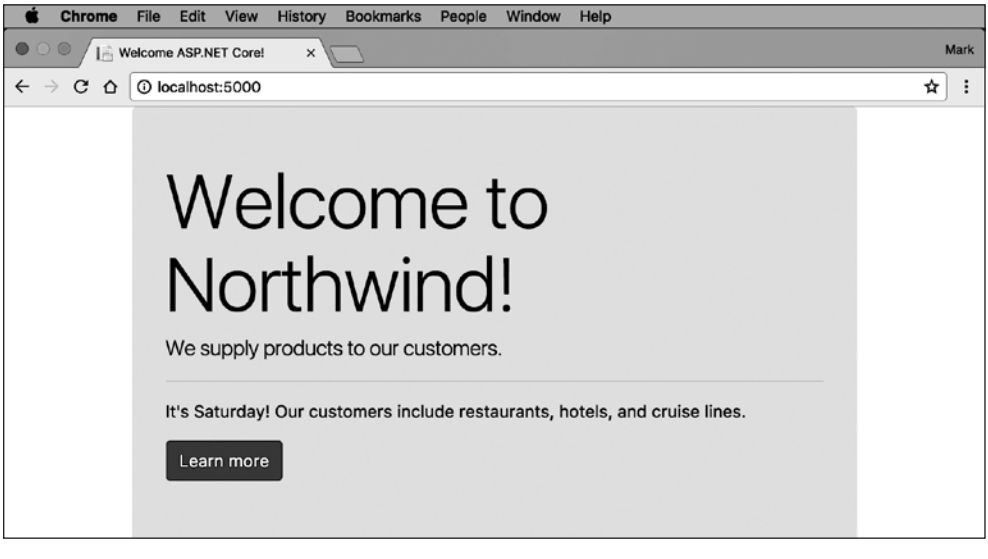


Рис. 14.10

Использование общих макетов с Razor Pages

Большинство сайтов имеют более одной страницы. Если бы каждая из страниц содержала шаблонный код разметки, который сейчас представлен в `index.cshtml`, то администрировать такие страницы было бы очень сложно. Поэтому в ASP.NET Core предусмотрены *макеты*.

Установка общего макета

Чтобы использовать макеты, сначала нужно создать специальный файл с необходимым именем, а затем установить его в качестве макета по умолчанию для всех страниц Razor (и всех видов MVC). Такому файлу необходимо присвоить имя `_ViewStart.cshtml`.

В Visual Studio 2017 добавьте новый элемент MVC View Start Page (Начальная страница представления MVC) в каталог `Pages` (рис. 14.11).

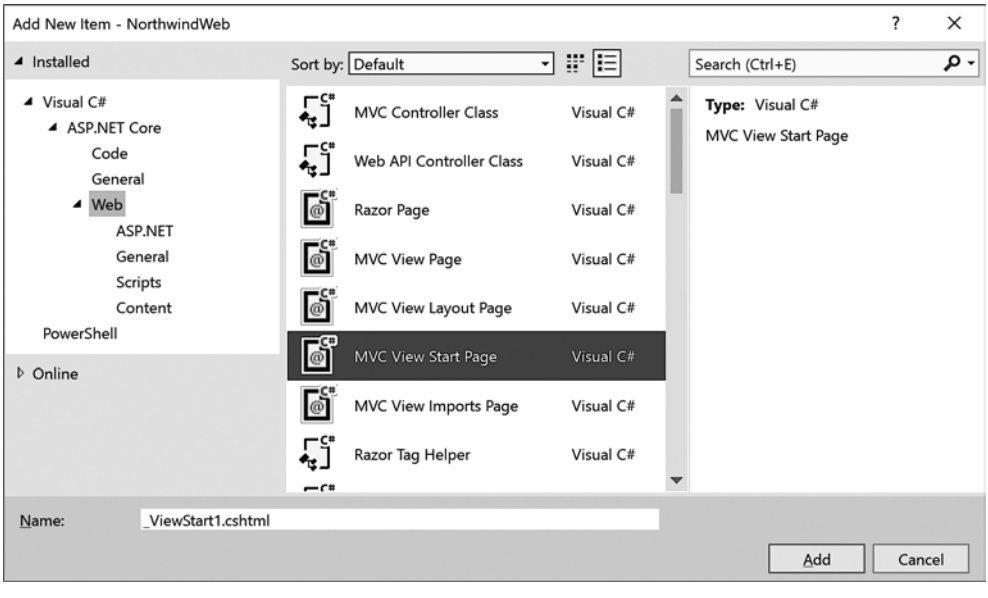


Рис. 14.11

В Visual Studio Code создайте файл `_ViewStart.cshtml` в каталоге `Pages` и измените его содержание, как показано в следующем листинге:

```
@{
  Layout = "_Layout";
}
```

Определение общего макета

Чтобы использовать макеты, теперь нужно создать файл `.cshtml` для определения макета по умолчанию для всех страниц Razor (и всех видов MVC). Такому файлу необходимо присвоить любое имя, но рекомендуется, как и в случае с файлом `_ViewStart.cshtml`, устанавливать стандартное имя `_Layout.cshtml`.

В Visual Studio 2017 добавьте новый элемент MVC View Start Page (Начальная страница представления MVC) в каталог `Pages`.

В Visual Studio Code создайте файл `_Layout.cshtml` в каталоге `Pages`.

В Visual Studio 2017 и Visual Studio Code измените содержимое файла `_Layout.cshtml`, как показано в листинге ниже (он аналогичен файлу `index.cshtml`, так что вы можете скопировать и вставить код этого файла):

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<meta name="viewport" content=
  "width=device-width, initial-scale=1, shrink-to-fit=no" />
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css"
    integrity="sha384-
/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M"
    crossorigin="anonymous" />
  <title>@ViewData["Title"]</title>
</head>
<body>
  <div class="container">
    @RenderBody()
    <hr />
    <footer>
      <p>Copyright &copy; 2017 - @ViewData["Title"]</p>
    </footer>
  </div>
  <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
    integrity="sha384-
KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN"
    crossorigin="anonymous"></script>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/umd/popper.min.js"
    integrity="sha384-
b/U6ypiBEHpOf/4+1nzFpr53nxSS+GLCkfwBdFNTxtclqqenISfwAzpKaMNFNmj4"
    crossorigin="anonymous"></script>
  <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/js/bootstrap.min.js"
    integrity="sha384-
h0AbiXch4ZDo7tp9hKZ4TsHbi047NrKGLO3SEJAg45jXxnGIfYzk4Si90RDIqNm1"
    crossorigin="anonymous"></script>
  @RenderSection("Scripts", required: false)
</body>
</html>
```

Для всех страниц мы будем применять аналогичный макет, изначально использованный нами в index.cshtml, но с добавлением некоторых сценариев в нижнюю часть файла, чтобы реализовать ряд динамических функций технологии Bootstrap.

Измените файл `index.cshtml`, удалив всю разметку HTML, за исключением элемента `<div class="jumbotron">` и его содержимого, добавьте инструкцию в метод `OnGet` и сохраните заголовок страницы в словаре `ViewData`. Затем измените кнопку так, чтобы ее нажатие вело на страницу поставщиков (которую мы создадим в следующей теме), как показано в следующем листинге:

```
@page
@functions {
  public string DayName { get; set; }

  public void OnGet()
  {
    ViewData["Title"] = "Northwind Web Site";
    Model.DayName = DateTime.Now.ToString("dddd");
  }
}
<div class="jumbotron">
  <h1 class="display-3">Welcome to Northwind!</h1>
  <p class="lead">We supply products to our customers.</p>
  <hr />
  <p>
    It's @Model.DayName! Our customers include restaurants, hotels, and cruise lines.
  </p>
  <p>
    <a class="btn btn-primary" href="suppliers">
      Learn more about our suppliers
    </a>
  </p>
</div>
```

ViewData — это словарь, в котором для ключей используются значения типа string. Очень удобно передавать значения между страницами Razor и общим макетом, не прибегая к необходимости определять свойство модели, как мы поступили для DayName. Обратите внимание на то, что ViewData["Title"] устанавливается на странице Razor, но при этом считывается в общем макете.

Запустите сайт и обратите внимание: его поведение аналогично тому, что мы видели ранее.

Работая в Visual Studio Code, не забудьте вот о чем: чтобы увидеть изменения, нужно останавливать веб-сервер нажатием сочетания клавиш Ctrl+C и перезапускать его с помощью ввода команды dotnet run.

Использование отдельных файлов кода программной части с технологией Razor Pages

Иногда лучше разделять разметку HTML от данных и исполняемого кода, поэтому технология Razor Pages позволяет создавать отдельные файлы с кодами классов.

В Visual Studio 2017 щелкните правой кнопкой мыши на каталоге `Pages` и в контекстном меню выполните команду Add ▸ New Item (Добавить ▸ Новый элемент). В открывшемся диалоговом окне Add New Item (Добавить новый элемент) выберите пункт Razor Page (Страница Razor), измените имя на `Suppliers.cshtml` и нажмите кнопку Add (Добавить).

В Visual Studio Code добавьте в каталог `Pages` два файла с именами `Suppliers.cshtml` и `Suppliers.cshtml.cs`.

Измените содержимое файла `Suppliers.cshtml.cs`, как показано в листинге ниже, и обратите внимание на эти аспекты:

- класс `SuppliersModel` наследует от `PageModel`, вследствие чего в нем присутствуют такие члены, как `ViewData`;
- класс `SuppliersModel` определяет свойство для хранения коллекции строковых значений;
- при создании HTTP-запроса GET на данную страницу Razor свойство `Suppliers` заполняется примерами названий поставщиков:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;

namespace NorthwindWeb.Pages
{
  public class SuppliersModel : PageModel
  {
    public IEnumerable<string> Suppliers { get; set; }

    public void OnGet()
    {
      ViewData["Title"] = "Northwind Web Site - Suppliers";
      Suppliers = new[]
      { "Alpha Co", "Beta Limited", "Gamma Corp" };
    }
  }
}
```

В этом примере мы сосредоточиваемся на изучении отдельных файлов с программным кодом. В следующей теме загрузим список поставщиков из базы данных, но на данный момент лишь эмулируем этот процесс с помощью жестко закодированного массива значений типа string.

Измените содержимое файла `Suppliers.cshtml`, как показано в следующем листинге, и обратите внимание на такие аспекты:

- класс `SuppliersModel` установлен в качестве модели страницы Razor;
- страница выводит на экран таблицу HTML со стилями Bootstrap;
- строки данных в таблице генерируются за счет цикличного прохода по свойству `Suppliers` класса `Model`.

```
@page
@model NorthwindWeb.Pages.SuppliersModel
<div class="row">
  <h1 class="display-2">Suppliers</h1>
  <table class="table">
    <thead class="thead-inverse">
      <tr><th>Company Name</th></tr>
    </thead>
```

```
        <tbody>
          @foreach(string name in Model.Suppliers)
          {
            <tr><td>@name</td></tr>
          }
        </tbody>
  </table>
</div>
```

Запустите сайт, нажмите кнопку для получения дополнительной информации о поставщиках и просмотрите таблицу поставщиков (рис. 14.12).

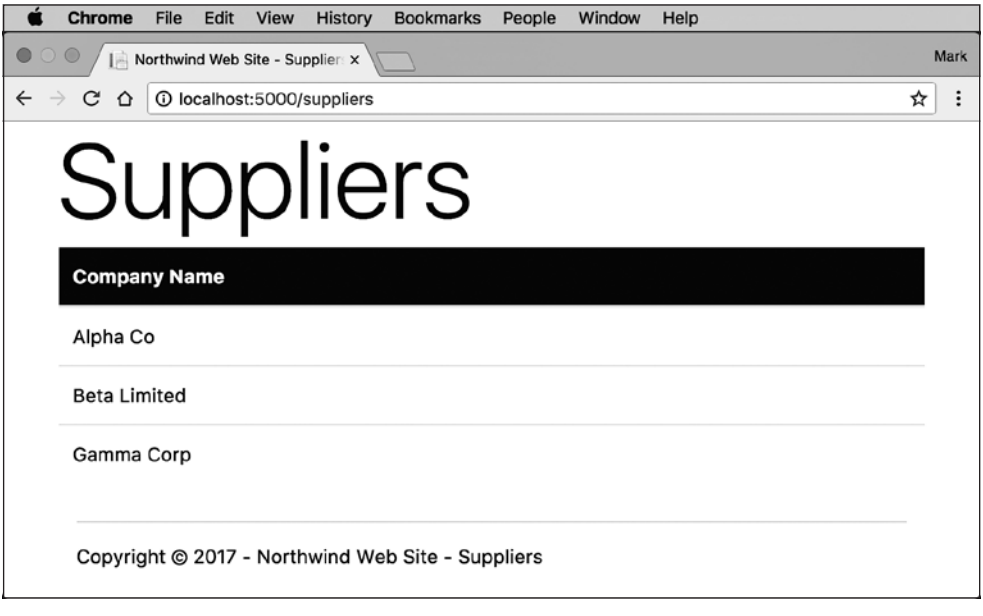


Рис. 14.12

Использование Entity Framework Core совместно с ASP.NET Core

Технология Entity Framework Core входит в состав платформы ASP.NET Core 2.0, поэтому вполне естественно подходит для выгрузки реальных данных на сайт.

Создание сущностных моделей для базы данных Northwind

Создание сущностных моделей данных в отдельных библиотеках классов, совместимых с .NET Standard 2.0, позволяет повторно использовать их в других проектах, что является хорошей практикой.

Создание библиотеки сущностных классов для базы данных Northwind

В Visual Studio 2017 выполните команду File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне Add New Project (Добавить новый проект) в списке Installed (Установленные) раскройте категорию Visual C# и выберите пункт .NET Standard. В центре диалогового окна выберите пункт Class Library (.NET Standard) (Библиотека классов (.NET Standard)), введите имя проекта `NorthwindEntitiesLib`, укажите расположение `C:\Code\Part3`, а затем нажмите кнопку OK (рис. 14.13).

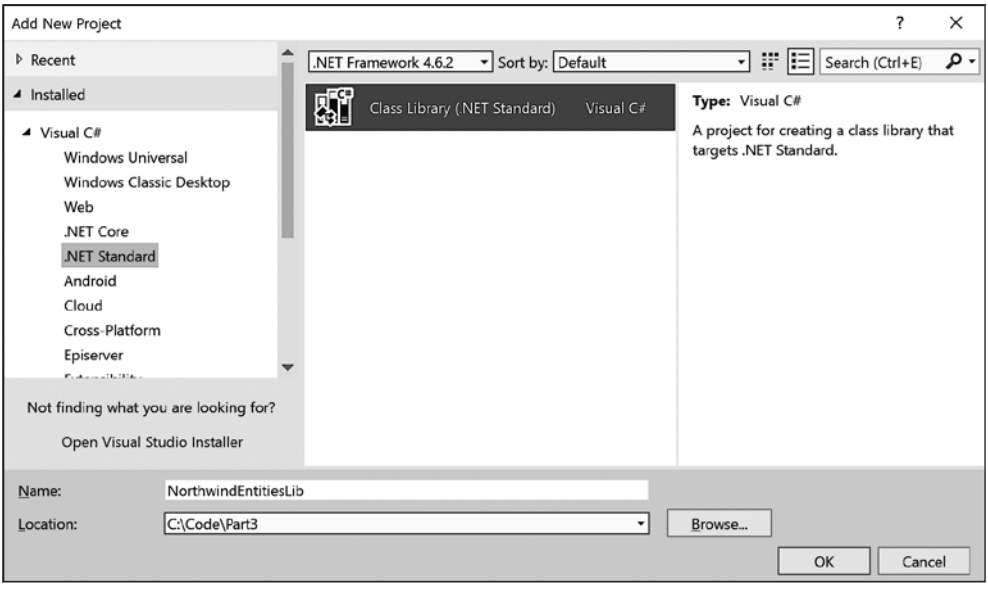


Рис. 14.13

В Visual Studio Code в каталоге `Part3` создайте подкаталог `NorthwindEntitiesLib` и откройте его.

На панели Integrated Terminal (Интегрированный терминал) введите команду `dotnet new classlib`.

В Visual Studio Code откройте каталог `Part3` и на панели Integrated Terminal (Интегрированный терминал) введите команду `cd NorthwindWeb`.

Определение сущностных классов

В Visual Studio 2017 и Visual Studio Code добавьте следующие файлы классов в проект NorthwindEntitiesLib: `Category.cs`, `Customer.cs`, `Employee.cs`, `Order.cs`, `OrderDetail.cs`, `Product.cs`, `Shipper.cs` и `Supplier.cs`.

Файл `Category.cs` должен выглядеть так:

```
using System.Collections.Generic;

namespace Packt.CS7
{
  public class Category
  {
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
    public string Description { get; set; }
    public ICollection<Product> Products { get; set; }
  }
}
```

Файл `Customer.cs` должен выглядеть следующим образом:

```
using System.Collections.Generic;

namespace Packt.CS7
{
  public class Customer
  {
    public string CustomerID { get; set; }
    public string CompanyName { get; set; }
    public string ContactName { get; set; }
    public string ContactTitle { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string Phone { get; set; }
    public string Fax { get; set; }
    public ICollection<Order> Orders { get; set; }
  }
}
```

Файл `Employee.cs` должен выглядеть так:

```
using System;
using System.Collections.Generic;

namespace Packt.CS7
{
  public class Employee
  {
    public int EmployeeID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public string Title { get; set; }
    public string TitleOfCourtesy { get; set; }
    public DateTime? BirthDate { get; set; }
```

```
    public DateTime? HireDate { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string HomePhone { get; set; }
    public string Extension { get; set; }
    public string Notes { get; set; }
    public int ReportsTo { get; set; }
    public Employee Manager { get; set; }
    public ICollection<Order> Orders { get; set; }
  }
}
```

Файл `Order.cs` должен выглядеть следующим образом:

```
using System;
using System.Collections.Generic;

namespace Packt.CS7
{
  public class Order
  {
    public int OrderID { get; set; }
    public string CustomerID { get; set; }
    public Customer Customer { get; set; }
    public int EmployeeID { get; set; }
    public Employee Employee { get; set; }
    public DateTime? OrderDate { get; set; }
    public DateTime? RequiredDate { get; set; }
    public DateTime? ShippedDate { get; set; }
    public int ShipVia { get; set; }
    public Shipper Shipper { get; set; }
    public decimal? Freight { get; set; } = 0;
    public ICollection<OrderDetail> OrderDetails { get; set; }
  }
}
```

Файл `OrderDetail.cs` должен выглядеть так:

```
namespace Packt.CS7
{
  public class OrderDetail
  {
    public int OrderID { get; set; }
    public Order Order { get; set; }
    public int ProductID { get; set; }
    public Product Product { get; set; }
    public decimal UnitPrice { get; set; } = 0;
    public short Quantity { get; set; } = 1;
    public double Discount { get; set; } = 0;
  }
}
```

Файл `Product.cs` должен выглядеть следующим образом:

```
namespace Packt.CS7
{
  public class Product
  {
    public int ProductID { get; set; }
    public string ProductName { get; set; }
    public int? SupplierID { get; set; }
    public Supplier Supplier { get; set; }
    public int? CategoryID { get; set; }
    public Category Category { get; set; }
    public string QuantityPerUnit { get; set; }
    public decimal? UnitPrice { get; set; } = 0;
    public short? UnitsInStock { get; set; } = 0;
    public short? UnitsOnOrder { get; set; } = 0;
    public short? ReorderLevel { get; set; } = 0;
    public bool Discontinued { get; set; } = false;
  }
}
```

Файл `Shipper.cs` должен выглядеть так:

```
using System.Collections.Generic;

namespace Packt.CS7
{
  public class Shipper
  {
    public int ShipperID { get; set; }
    public string ShipperName { get; set; }
    public string Phone { get; set; }
    public ICollection<Order> Orders { get; set; }
  }
}
```

Файл `Supplier.cs` должен выглядеть следующим образом:

```
using System.Collections.Generic;

namespace Packt.CS7
{
  public class Supplier
  {
    public int SupplierID { get; set; }
    public string CompanyName { get; set; }
    public string ContactName { get; set; }
    public string ContactTitle { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string Phone { get; set; }
```

```
    public string Fax { get; set; }
    public string HomePage { get; set; }
    public ICollection<Product> Products { get; set; }
  }
}
```

Вы должны создать отдельный проект библиотеки классов для моделей сущностей данных, которые не имеют зависимости ни от чего, кроме .NET Standard 2.0. Это позволяет упростить обмен данными между бэкенд-серверами и фронтенд-клиентами.

Создание библиотеки классов для контекста БД Northwind

В Visual Studio 2017 вы будете использовать поставщик данных EF Core для SQL Server, в Visual Studio Code — поставщик данных EF Core для SQLite.

Visual Studio 2017. Нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне Add New Project (Добавить новый проект) в списке Installed (Установленные) раскройте категорию Visual C# и выберите пункт .NET Standard. В центре диалогового окна выберите пункт Class Library (.NET Standard) (Библиотека классов (.NET Standard)), введите имя проекта `NorthwindContextLib`, укажите расположение `C:\Code\Part3`, а затем нажмите кнопку OK.

На панели Solution Explorer (Обозреватель решений) раскройте проект NorthwindContextLib, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Add Reference (Добавить ссылку).

В диалоговом окне Reference Manager — NorthwindContextLib (Диспетчер ссылок — NorthwindContextLib) выберите пункт NorthwindEntitiesLib и нажмите кнопку OK (рис. 14.14).

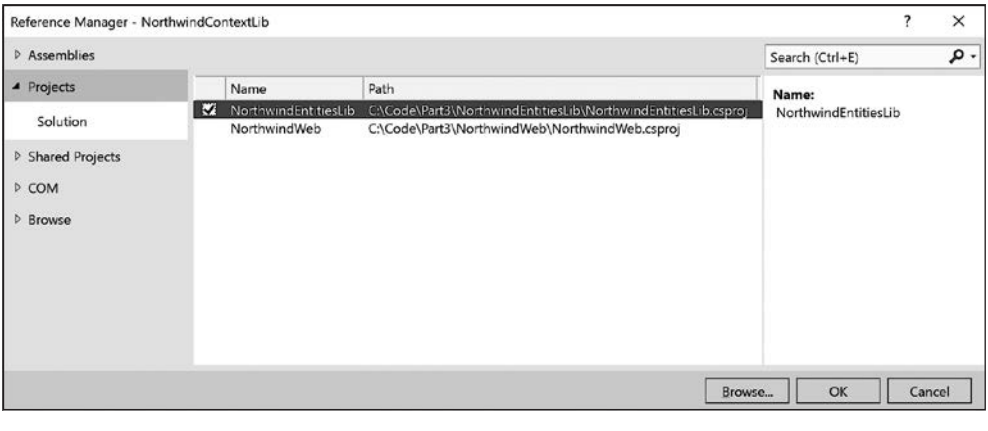


Рис. 14.14

На панели Solution Explorer (Обозреватель решений) раскройте проект NorthwindContextLib, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Manage NuGet Packages (Управление пакетами NuGet).

В окне NuGet Package Manager: NorthwindContextLib (Диспетчер пакетов NorthwindContextLib) перейдите на вкладку Browse (Обзор), выполните поиск и установите пакет Microsoft.EntityFrameworkCore.SqlServer, затем нажмите кнопку OK.

Visual Studio Code. В каталоге `Part3` создайте подкаталог `NorthwindContextLib` и откройте его.

На панели Integrated Terminal (Интегрированный терминал) введите команду: `dotnet new classlib`.

В Visual Studio Code измените содержимое файла `NorthwindContextLib.csproj`, чтобы добавить ссылку на проект NorthwindEntitiesLib и пакет Entity Framework Core 2.0 для SQLite, как показано в листинге ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\NorthwindEntitiesLib\NorthwindEntitiesLib.csproj" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SQLite"
    Version="2.0.0" />
  </ItemGroup>

</Project>
```

В Visual Studio Code откройте папку `Part3` и на панели Integrated Terminal (Интегрированный терминал) введите команду `cd NothwindWeb`.

Определение класса контекста базы данных

В Visual Studio 2017 и Visual Studio Code в проекте NorthwindContextLib переименуйте файл класса `Class1.cs` в `Northwind.cs`.

Файл `Northwind.cs` должен выглядеть следующим образом:

```
using Microsoft.EntityFrameworkCore;

namespace Packt.CS7
{
  public class Northwind : DbContext
  {
    public DbSet<Category> Categories { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderDetail> OrderDetails { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Shipper> Shippers { get; set; }
```

```
public DbSet<Supplier> Suppliers { get; set; }

public Northwind(DbContextOptions options)
    : base(options) { }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  base.OnModelCreating(modelBuilder);

  modelBuilder.Entity<Category>()
    .Property(c => c.CategoryName)
    .IsRequired()
    .HasMaxLength(15);

  // определить отношение "один-ко-многим"
  modelBuilder.Entity<Category>()
    .HasMany(c => c.Products)
    .WithOne(p => p.Category);

  modelBuilder.Entity<Customer>()
    .Property(c => c.CustomerID)
    .IsRequired()
    .HasMaxLength(5);

  modelBuilder.Entity<Customer>()
    .Property(c => c.CompanyName)
    .IsRequired()
    .HasMaxLength(40);

  modelBuilder.Entity<Customer>()
    .Property(c => c.ContactName)
    .HasMaxLength(30);

  modelBuilder.Entity<Customer>()
    .Property(c => c.Country)
    .HasMaxLength(15);

  modelBuilder.Entity<Employee>()
    .Property(c => c.LastName)
    .IsRequired()
    .HasMaxLength(20);

  modelBuilder.Entity<Employee>()
    .Property(c => c.FirstName)
    .IsRequired()
    .HasMaxLength(10);

  modelBuilder.Entity<Employee>()
    .Property(c => c.Country)
    .HasMaxLength(15);

  modelBuilder.Entity<Product>()
    .Property(c => c.ProductName)
```

```
        .IsRequired()
        .HasMaxLength(40);

      modelBuilder.Entity<Product>()
        .HasOne(p => p.Category)
        .WithMany(c => c.Products);

      modelBuilder.Entity<Product>()
        .HasOne(p => p.Supplier)
        .WithMany(s => s.Products);

      modelBuilder.Entity<OrderDetail>()
        .ToTable("Order Details");

      // определить первый ключ в нескольких столбцах
      // для таблицы Order Details
      modelBuilder.Entity<OrderDetail>()
        .HasKey(od => new { od.OrderID, od.ProductID });

      modelBuilder.Entity<Supplier>()
        .Property(c => c.CompanyName)
        .IsRequired()
        .HasMaxLength(40);

      modelBuilder.Entity<Supplier>()
        .HasMany(s => s.Products)
        .WithOne(p => p.Supplier);
    }
  }
}
```

Мы установим строку подключения базы данных в сценарии запуска ASP.NET Core, так что этого не потребуется делать в классе Northwind, но для класса, наследующего от DbContext, должен быть реализован конструктор с параметром DbContextOptions.

Создание базы данных Northwind на сайте

Выполните инструкции, описанные в главе 11, для создания базы данных `Northwind`. Ниже приведен краткий перечень необходимых шагов.

- Чтобы использовать SQL Server в Windows, создайте базу данных на сервере `(local)\mssqllocaldb`. Если вы выполнили задания из предыдущих глав, то у вас уже есть такая БД.
- Чтобы применять SQLite в macOS или других операционных системах, создайте файл `Northwind.db`, скопировав файл `Northwind4SQLite.sql` в каталог `Part3`, а затем в области TERMINAL (Терминал) введите следующую команду:

```
sqlite3 Northwind.db < Northwind4SQLite.sql
```

Настройка Entity Framework Core в качестве сервиса

Сервисы, такие как Entity Framework Core, необходимые для ASP.NET Core, должны быть зарегистрированы в качестве таковых во время запуска платформы.

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) раскройте проект NorthwindWeb, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Add Reference (Добавить ссылку).

В диалоговом окне Reference Manager — NorthwindWeb (Диспетчер ссылок — NorthwindContextLib) выберите пункт NorthwindContextLib и нажмите кнопку OK.

В Visual Studio Code в проекте NorthwindWeb измените файл `NorthwindWeb.csproj` и добавьте ссылку на проект NorthwindContextLib, как показано в следующем листинге (выделено полужирным):

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <ProjectReference Include="..\NorthwindContextLib\NorthwindContextLib.csproj" />
  </ItemGroup>
</Project>
```

В Visual Studio 2017 и Visual Studio Code откройте файл `Startup.cs` и импортируйте пространства имен `Microsoft.EntityFrameworkCore` и `Packt.CS7`, как показано в коде ниже:

```
using Microsoft.EntityFrameworkCore;
using Packt.CS7;
```

Добавьте следующие инструкции в метод `ConfigureServices`.

Для SQL Server LocalDB:

```
services.AddDbContext<Northwind>(options => options.UseSqlServer("Server=(localdb)\\
mssqllocaldb;Database=Northwind;
Trusted_Connection=True;MultipleActiveResultSets=true"));
```

Для SQLite:

```
services.AddDbContext<Northwind>(options => options.UseSqlite("DataSource=../
Northwind.db"));
```

В каталоге `Pages` проекта NorthwindWeb откройте файл `Suppliers.cshtml.cs` и импортируйте пространства имен `System.Linq` и `Packt.CS7`, как показано в листинге ниже:

```
private Northwind db;

public SuppliersModel(Northwind injectedContext)
{
  db = injectedContext;
}
```

В методе `OnGet` измените инструкции так, чтобы для получения названий поставщиков использовалась база данных `Northwind`, как показано в следующем листинге:

```
public void OnGet()
{
  ViewData["Title"] = "Northwind Web Site - Suppliers";

  Suppliers = db.Suppliers.Select(s => s.CompanyName).ToArray();
}
```

Запустите сайт и обратите внимание на то, что таблица поставщиков теперь загружается из базы (рис. 14.15).

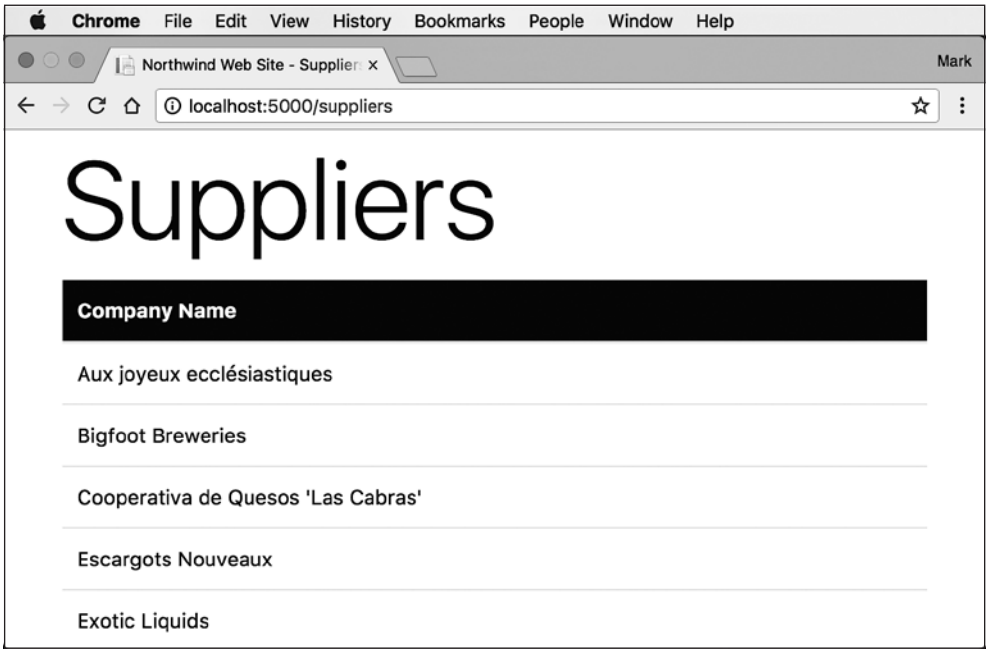


Рис. 14.15

Работа с данными

Реализуем функциональность, позволяющую добавлять нового поставщика.

Откройте файл `Suppliers.cshtml.cs` и импортируйте это пространство имен:

```
using Microsoft.AspNetCore.Mvc;
```

Добавьте в класс `SuppliersModel` свойство для сохранения поставщика, а также метод `OnPost`, добавляющий поставщика в случае, если его модель задана корректно, как показано в листинге ниже:

```
[BindProperty]
public Supplier Supplier { get; set; }

public IActionResult OnPost()
{
  if (ModelState.IsValid)
  {
    db.Suppliers.Add(Supplier);
    db.SaveChanges();
    return RedirectToPage("/suppliers");
  }
  return Page();
}
```

Откройте файл `Suppliers.cshtml` и добавьте вспомогательные функции тегов, как показано в следующем коде:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Добавьте в нижнюю часть файла `.cshtml` форму для ввода данных нового поставщика и воспользуйтесь вспомогательной функцией тега `asp-for`, чтобы подключить свойство `CompanyName` класса `Supplier` к полю для ввода текста, как показано в листинге ниже:

```
<div class="row">
  <p>Enter a name for a new supplier:</p>
  <form method="POST">
    <div><input asp-for="Supplier.CompanyName" /></div>
    <input type="submit" />
  </form>
</div>
```

Запустите сайт, выберите ссылку Learn more about out suppliers (Получить дополнительную информацию о наших поставщиках), пролистайте до формы для добавления нового поставщика, введите `Bob's Burgers` и нажмите кнопку Submit (Передать) (рис. 14.16).

Если бы вы создали точку останова внутри метода `OnPost`, а также добавили инструкцию наблюдения для свойства `Supplier`, то увидели бы, что его свойства заполнились автоматически (рис. 14.17).

После нажатия кнопки Submit (Передать) вас переадресуют обратно к списку Suppliers (Поставщики) (рис. 14.18).

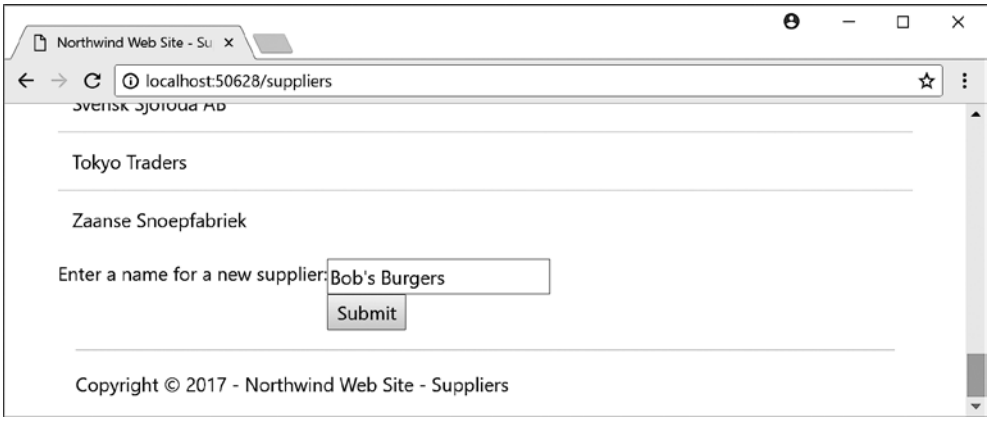


Рис. 14.16

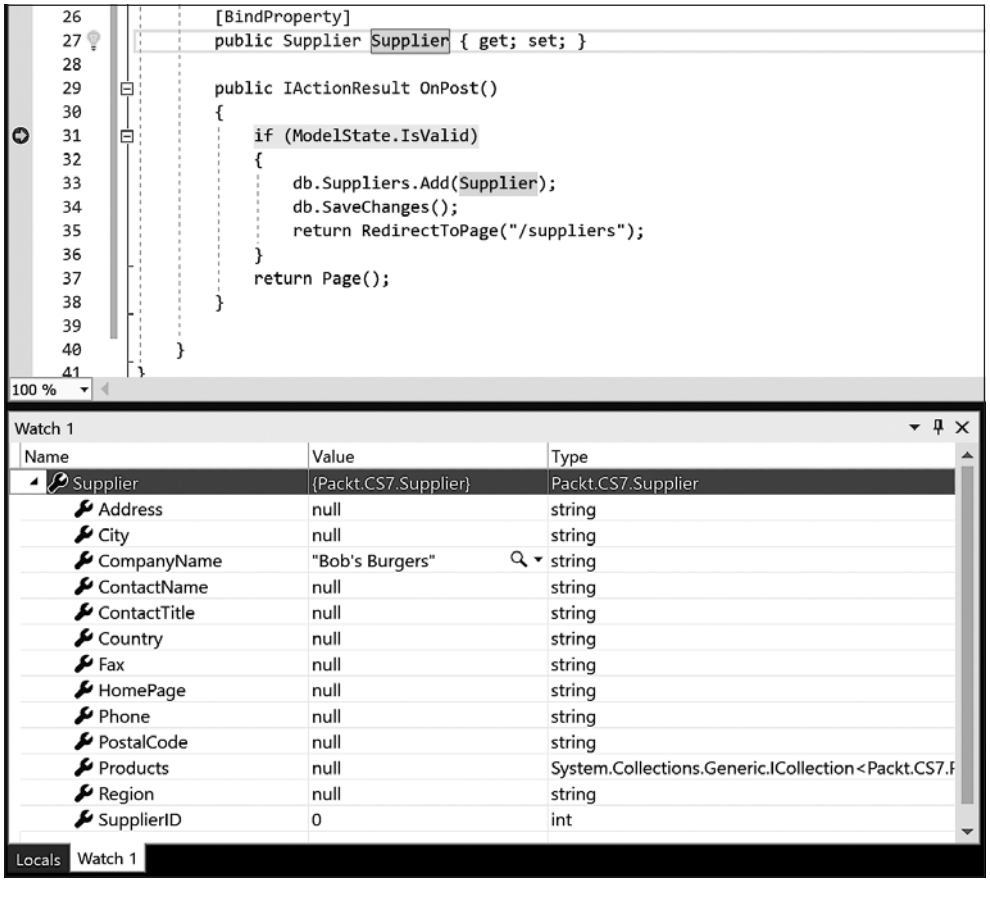


Рис. 14.17

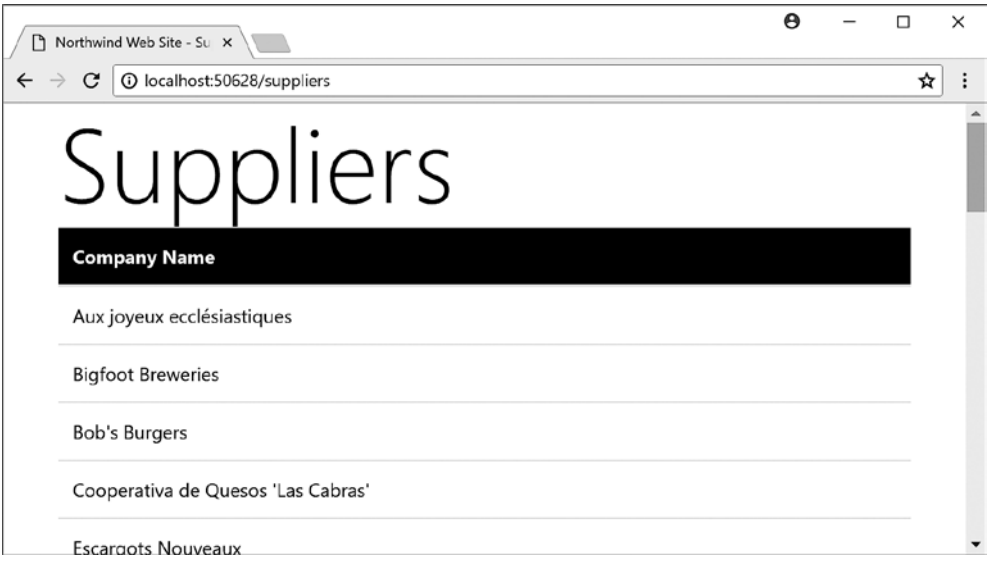


Рис. 14.18

Практические задания

Проверьте полученные знания. Для этого выполните приведенное упражнение и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Упражнение 14.1. Создание сайта, ориентированного на данные

Добавьте на сайт NorthwindWeb страницу Razor, которая позволяет посетителю просмотреть список клиентов, сгруппированный по странам. Щелкая на записи клиента, посетитель должен видеть страницу с полными контактными данными выбранного клиента, а также перечень его заказов.

Дополнительные ресурсы

- ASP.NET Core: https://www.asp.net/core.
- Работа со статическими файлами в ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files.
- Введение в Razor Pages в ASP.NET Core: https://docs.microsoft.com/en-us/aspnet/core/mvc/razor-pages/.

- Работа с данными в ASP.NET Core: https://docs.microsoft.com/en-gb/aspnet/core/data/.
- Расписание релизов и перспективы развития ASP.NET Core: https://github.com/aspnet/Home/wiki/Roadmap.

Резюме

В этой главе вы научились создавать простой сайт, возвращающий статические файлы, а также использовали технологию Razor Pages платформы ASP.NET Core и Entity Framework Core для создания веб-страниц, динамически генерируемых из базы данных.

В следующей главе вы научитесь создавать более сложные сайты, применяя ASP.NET Core MVC, разделяющую работу на модели, представления и контроллеры.

15 Разработка сайтов с помощью ASP.NET Core MVC

Эта глава посвящена разработке сайтов с современной серверной HTTP-архитектурой с помощью Microsoft ASP.NET Core MVC. Вы узнаете об основах конфигурации, аутентификации, авторизации, маршрутах, моделях, представлениях и контроллерах, которые составляют ASP.NET Core MVC.

В данной главе:

- создание и настройка сайта ASP.NET Core MVC;
- структура сайта ASP.NET Core MVC.

Создание и настройка сайта ASP.NET Core MVC

Технология ASP.NET Core Razor Pages великолепно подходит для создания простых сайтов. Разрабатывать сложные, комплексные сайты и управлять ими удобнее с помощью более формальной структуры.

Именно здесь находит применение паттерн проектирования «Модель — представление — контроллер». В нем используются технологии, подобные Razor Pages, но более тщательно разделяющие задачи, как продемонстрировано в этом перечне:

- `Models` — каталог, содержащий классы, которые представляют данные сайта;
- `Views` — каталог, включающий файлы Razor Pages с расширением `.cshtml`, которые преобразуют модели в HTML-страницы;
- `Controllers` — каталог, содержащий классы, выполняющие код при поступлении HTTP-запроса. Код обычно создает модель и передает ее в представление.

Лучший способ понять модель MVC — испытать ее на практике.

Создание сайта ASP.NET Core MVC

В Visual Studio 2017 сайт MVC создается графическим способом, а в Visual Studio Code — через интерфейс командной строки. Я рекомендую рассмотреть оба пути, чтобы оценить сходства и различия.

Visual Studio 2017

В Visual Studio 2017 откройте решение Part3 и нажмите сочетание клавиш Ctrl+Shift+N или выполните команду File ▸ Add ▸ New project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне Add New Project (Добавить новый проект) в списке Installed (Установленные) раскройте раздел Visual C# и выберите пункт .NET Core. В центре диалогового окна выберите пункт ASP.NET Core Web Application (Веб-приложение ASP.NET Core), присвойте ему имя `NorthwindMvc`, а затем нажмите кнопку OK.

В диалоговом окне New ASP.NET Core Web Application — NorthwindMvc (Создать веб-приложение ASP.NET Core — NorthwindMvc) выберите шаблон Web Application (Model-View-Controller) (Веб-приложение (Модель — представление — контроллер)). Убедитесь, что в качестве режима проверки подлинности в строке Authentication (Проверка подлинности) указано значение Individual User Accounts (Учетные записи отдельных пользователей). Если это не так, то измените режим, нажав кнопку Change Authentication (Изменить способ проверки подлинности). В раскрывающемся списке должен быть выбран пункт Store user accounts in-app (Хранить учетные записи пользователей в приложении) (рис. 15.1). Нажмите кнопку OK.

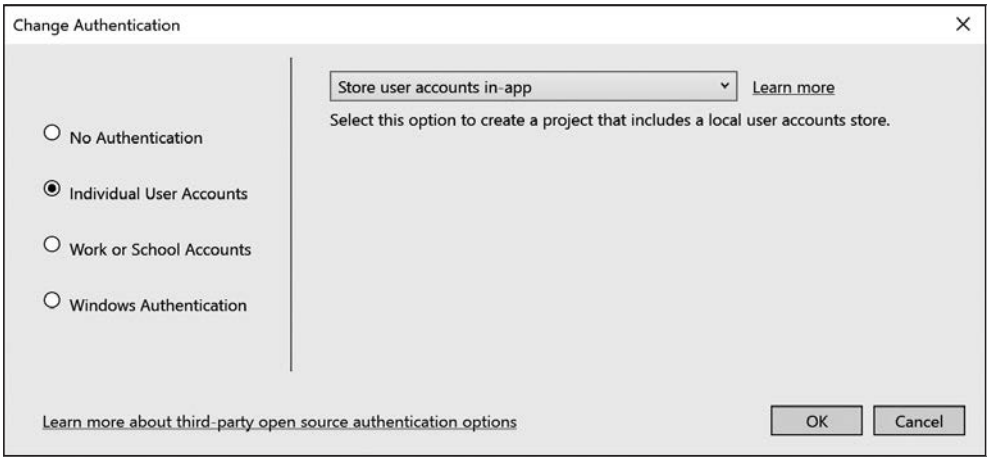


Рис. 15.1

В диалоговом окне New ASP.NET Core Web Application — NorthwindMvc (Создать веб-приложение ASP.NET Core — NorthwindMvc) флажок Enable Docker Support (Включить поддержку Docker) должен быть снят (рис. 15.2).

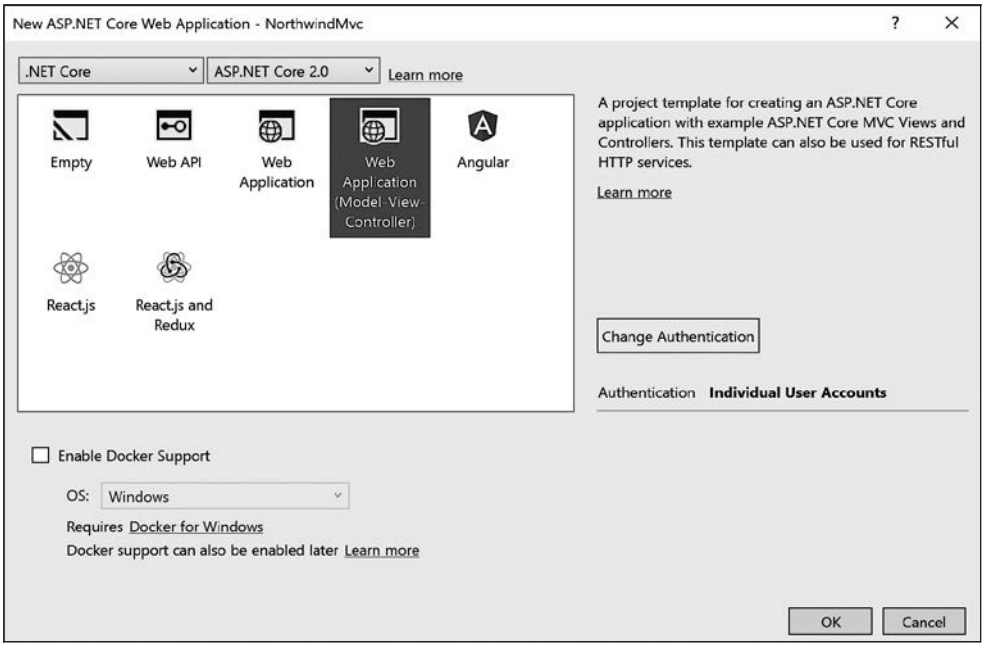


Рис. 15.2

Visual Studio Code

В каталоге `Part3` создайте подкаталог `NorthwindMvc` и откройте его в Visual Studio Code.

На панели Integrated Terminal (Интегрированный терминал) введите указанную ниже команду, чтобы создать приложение ASP.NET Core MVC с базой данных для аутентификации и авторизации пользователей.

```
dotnet new mvc --auth Individual
```

Проект MVC задействует расширение **Bower** для управления клиентскими пакетами, такими как Bootstrap и jQuery. По умолчанию, в Visual Studio Code оно не установлено, поэтому нужно установить его сейчас.

В Visual Studio Code выполните команду View ▸ Extensions (Вид ▸ Расширения) или нажмите сочетание клавиш Shift+Cmd+X.

Выполните поиск по запросу `bower`, чтобы найти популярное расширение Bower, и нажмите кнопку Install (Установить) (рис. 15.3).

После установки щелкните на ссылке Reload (Перезагрузка), а затем на ссылке Reload Window (Перезагрузка окна), чтобы перезагрузить Visual Studio Code.

Выполните команду View ▸ Command Palette (Вид ▸ Палитра команд) или нажмите сочетание клавиш Shift+Cmd+P.

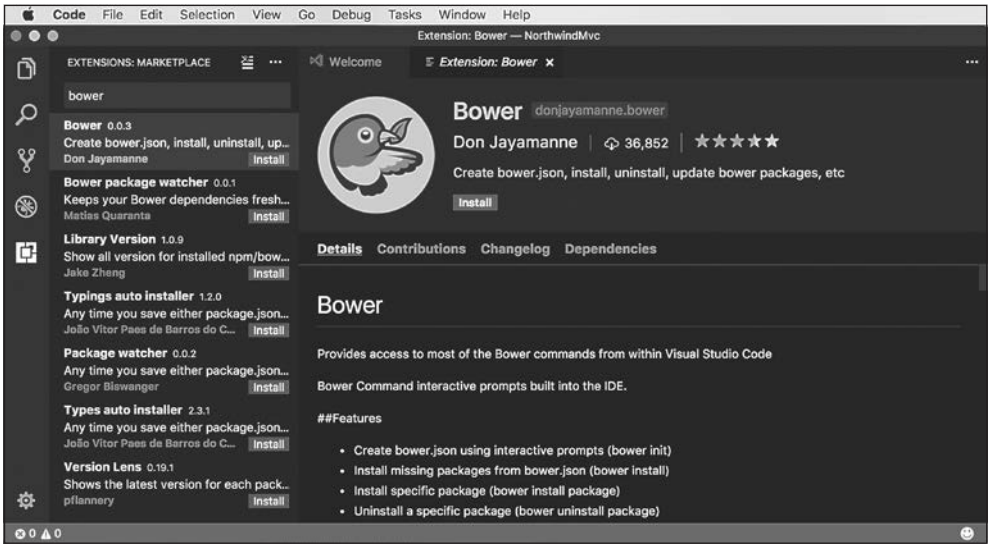


Рис. 15.3

Введите команду `Bower`, а затем выберите пункт Bower Install (Установить Bower) для восстановления клиентских пакетов, таких как Bootstrap (рис. 15.4).

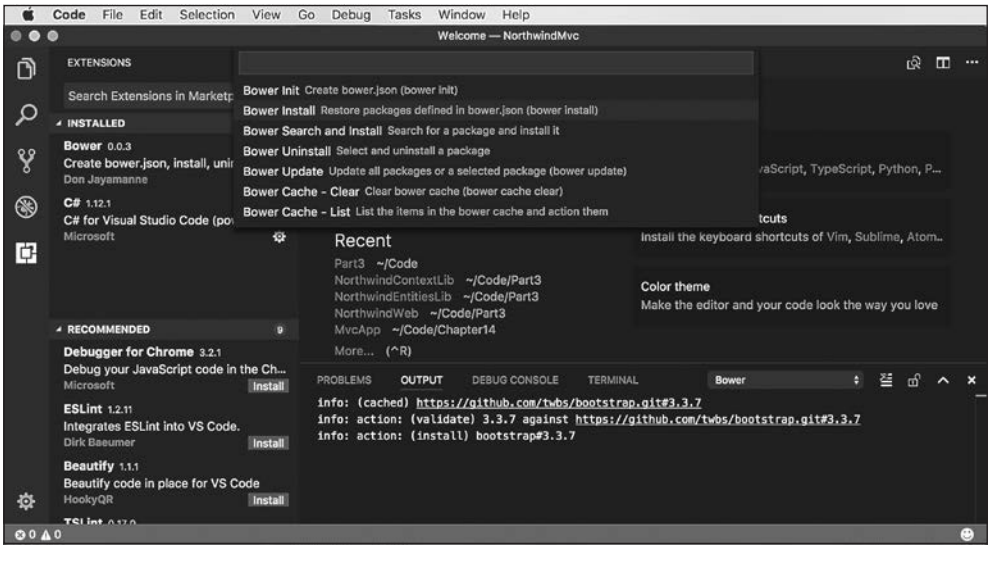


Рис. 15.4

Выполните команду View ▸ Explorer (Вид ▸ Проводник) или нажмите сочетание клавиш Shift+Cmd+E.

Раскройте каталог `wwwroot` и обратите внимание, что был создан каталог `lib` с четырьмя подкаталогами для пакетов, указанных в файле `bower.json`:

```
{
  "name": "webapplication",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.7",
    "jquery": "2.2.0",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Структура проекта веб-приложения ASP.NET Core MVC

В Visual Studio 2017 взгляните на панель Solution Explorer (Обозреватель решений), в Visual Studio Code — на панель EXPLORER (Проводник) (рис. 15.5).

Обратите внимание на следующие компоненты.

- Каталог `wwwroot` содержит статичный контент, такой как CSS-файлы для хранения стилей, изображения, JavaScript-сценарии и файл `favicon.ico`.
- Каталог `Data` включает классы Entity Framework Core, используемые системой *ASP.NET Identity* для обеспечения аутентификации и авторизации.
- Каталог `Dependencies` (Зависимости) (только в Visual Studio 2017) содержит графическое представление `NuGet`, предназначенное для удобного управления пакетами. Фактическими ссылками NuGet являются `bower.json` и `NorthwindMvc.csproj`. В Visual Studio 2017 можно отредактировать проект вручную, щелкнув правой кнопкой мыши на проекте и выбрав пункт Edit NorthwindMvc.csproj (Редактировать NorthwindMvc.csproj).
- Файл `NorthwindMvc.csproj` включает список NuGet-пакетов, таких как Core Entity Framework, которые необходимы для работы вашего проекта.
- Файлы `.vscode/launch.json` (только в Visual Studio Code) и `Properties/launchSettings.json` (только в Visual Studio 2017) определяют параметры запуска веб-приложения из вашей среды разработки.
- Каталог `Controllers` содержит классы C# с методами (известными как действия), которые осуществляют выборку *модели* и передачу ее *представлению*.
- Каталог `Models` включает классы C#, представляющие все данные, необходимые для ответа на HTTP-запрос.
- В каталог `Views` входят CSHTML-файлы, которые объединяют код на языках HTML и C#, реализующий динамическое генерирование HTML-ответа.
- Каталог `Services` содержит интерфейсы и классы C# для интеграции с внешними сервисами, такими как SMS (отправка коротких текстовых сообщений).
- Каталог `Extensions` включает методы расширения для проекта.

Рис. 15.5

- В файле `appsettings.json` доступны настройки, которые ваше веб-приложение может загружать во время выполнения, к примеру строка подключения базы данных для системы ASP.NET Identity.
- Файл `bower.json` содержит клиентские пакеты, которые объединяют ресурсы, такие как jQuery и Bootstrap.
- Файл `Program.cs` представляет собой консольное приложение, содержащее точку входа `Main` и выполняющее начальную конфигурацию, компиляцию и запуск веб-приложения. Он может вызывать метод `UseStartup<T>()`, чтобы указать класс, способный выполнить дополнительную конфигурацию.
- Дополнительный файл `Startup.cs` содержит конфигурационные данные для дополнительной настройки сервисов, к примеру ASP.NET Identity для аутентификации, SQLite для хранения данных и т. д., а также связи между компонентами в вашем приложении.

Миграция баз данных

Прежде чем тестировать веб-приложение, нужно убедиться в выполнении миграции базы данных. Это необходимо для создания таблиц, используемых системой аутентификации и авторизации ASP.NET Identity.

Visual Studio 2017

Откройте файл `appsettings.json` и обратите внимание на строку подключения базы данных. Она должна выглядеть примерно так:

```
Server=(localdb)\\mssqllocaldb;Database=aspnet-NorthwindMvcApp-584f323fa60e-4933-
9845-f67225753337;
Trusted_Connection=True;MultipleActiveResultSets=true
```

После выполнения миграции создается база данных с соответствующим именем в Microsoft SQL Server LocalDb.

Из меню Start (Пуск) операционной системы Windows запустите инструмент Developer Command Prompt for VS 2017.

Перейдите в каталог проекта и выполните миграцию базы данных, введя следующие команды:

```
cd C:\Code\Part3\NorthwindMvc
dotnet ef database update
```

Вы должны увидеть такой вывод (рис. 15.6).

Завершите работу инструмента Developer Command Prompt for VS 2017.

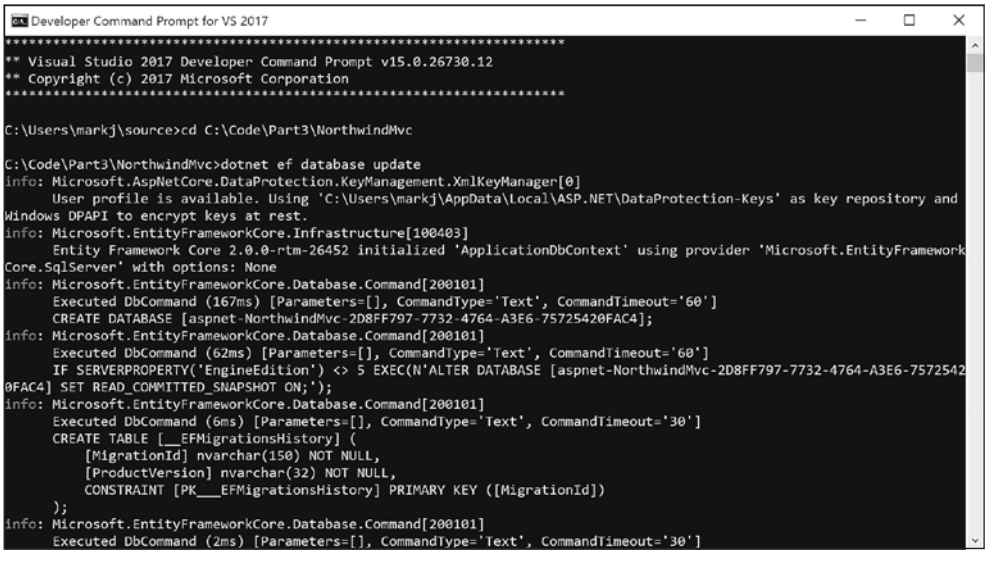


Рис. 15.6

Visual Studio Code

Откройте файл `appsettings.json` и обратите внимание на строку подключения базы данных. Она должна выглядеть примерно так:

```
Data Source=app.db
```

После завершения миграции в текущем каталоге будет создана база данных SQLite с соответствующим именем.

На панели Integrated Terminal (Интегрированный терминал) введите следующую команду для выполнения миграции базы данных:

```
dotnet ef database update
```

На панели EXPLORER (Проводник) обратите внимание на созданную базу данных `app.db`.

Если вы установили инструмент для работы с базами данных SQLite, такой как *SQLiteStudio*, то можете открыть базу данных и просмотреть таблицы, применяемые системой ASP.NET Identity для регистрации пользователей и ролей (рис. 15.7).

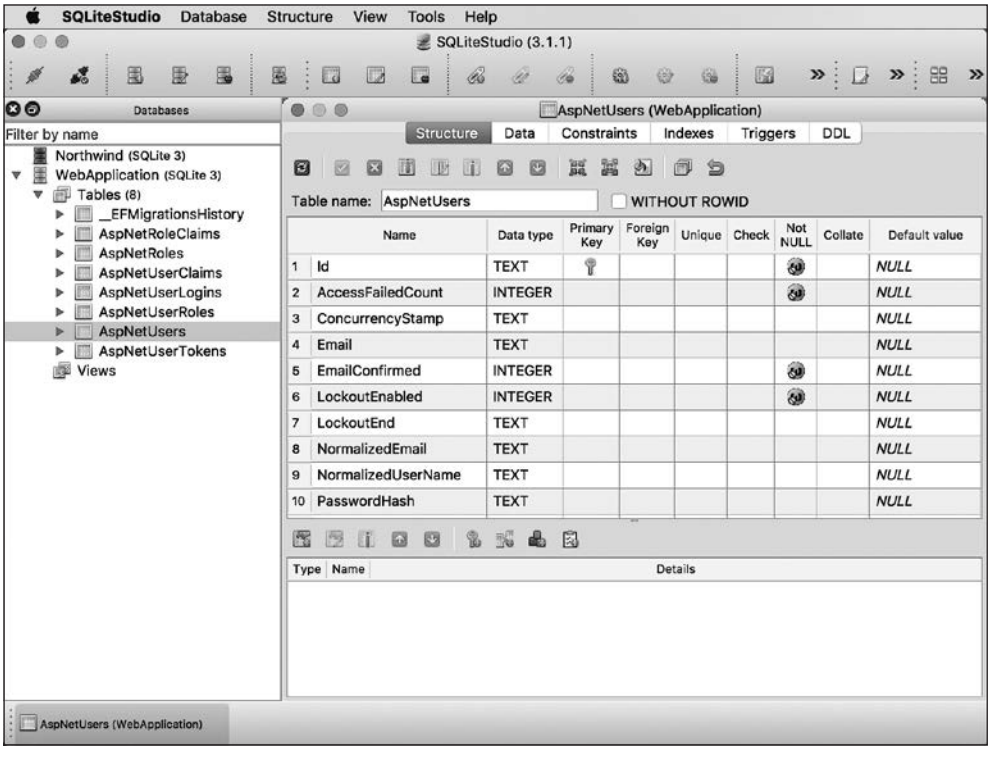


Рис. 15.7

Тестирование сайта ASP.NET MVC

Из Visual Studio 2017 или Visual Studio Code запустите сайт.

Если вы работаете в Visual Studio 2017, то браузер запустится автоматически; если в Visual Studio, то необходимо запустить его вручную и ввести адрес http://localhost:5000/.

При использовании Visual Studio 2017 появится запрос с подтверждением доверия самоподписанному сертификату, который был сгенерирован IIS Express для SSL. В этом случае нажмите кнопку Yes (Да).

Обратите внимание: ваш сайт ASP.NET Core выполняется на кросс-платформенном веб-сервере Kestrel (если используется Visual Studio 2017, то происходит интеграция с IIS Express) на случайном порте для локального тестирования. Кроме того, проект ASP.NET Core возвращает сайт с несколькими страницами, в том числе Home, About, Contact, Register и Log in (рис. 15.8).

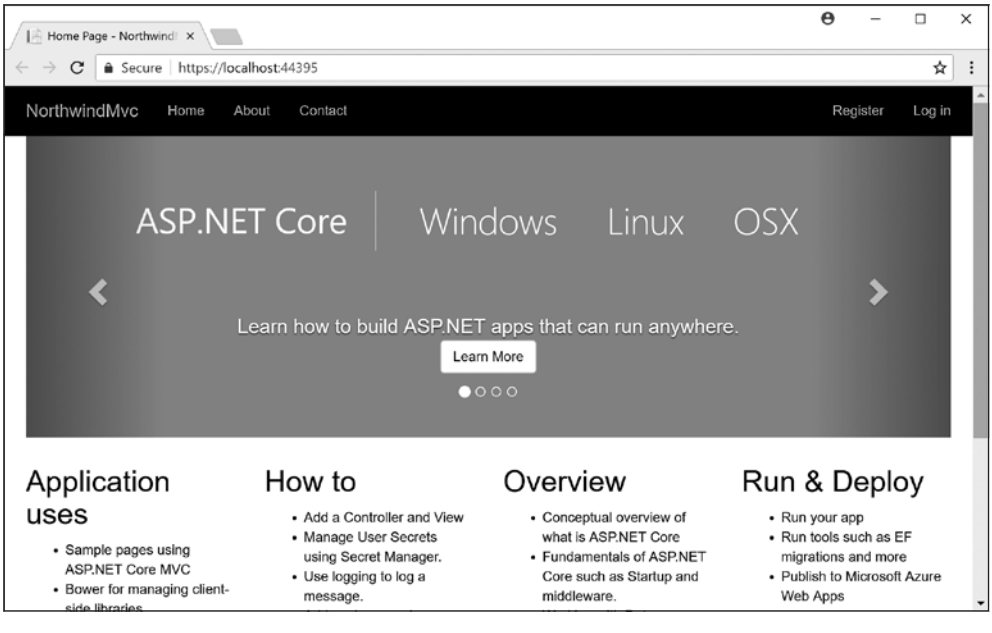


Рис. 15.8

Щелкните на ссылке Register (Регистрация) и заполните форму, чтобы ввести новую учетную запись в базе данных, созданной в процессе миграции (рис. 15.9).

Примите к сведению: если вы укажете недостаточно сложный пароль, то появится соответствующее уведомление.

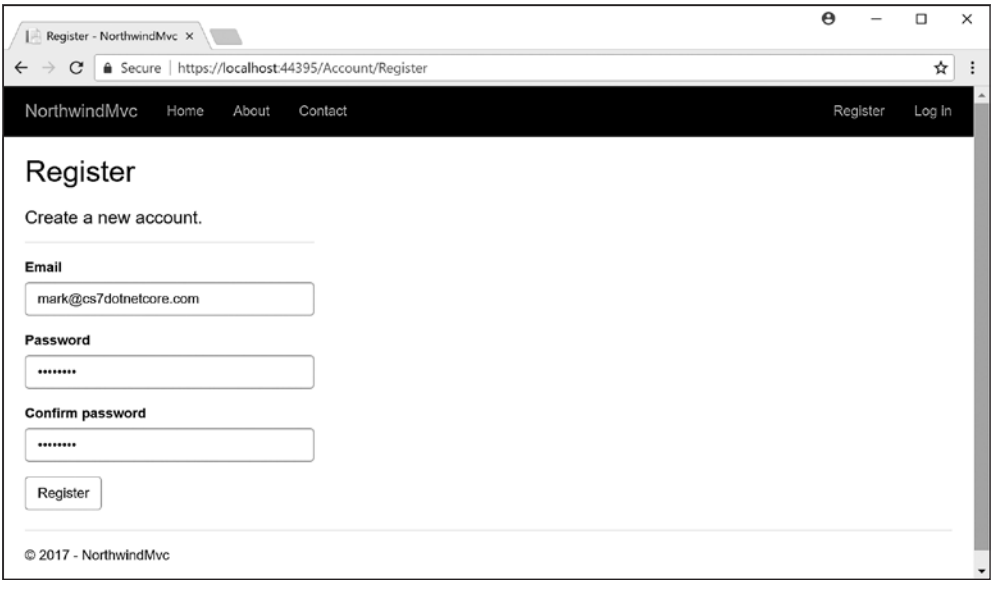


Рис. 15.9

Нажмите кнопку Register (Регистрация) и обратите внимание на то, что теперь вы зарегистрированы и авторизованы в системе (рис. 15.10).

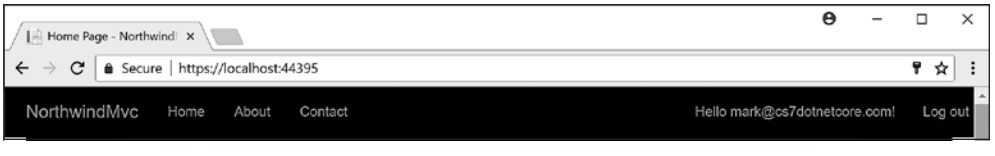


Рис. 15.10

Закройте браузер.

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) нажмите сочетание клавиш Ctrl+C, чтобы завершить работу консольного приложения и веб-сервера Kestrel, на котором размещен ваш сайт ASP.NET Core.

Проверка подлинности с помощью системы ASP.NET Identity

В Visual Studio 2017 выполните команду View ▸ Server Explorer (Вид ▸ Обозреватель серверов).

На панели Server Explorer (Обозреватель серверов) щелкните правой кнопкой мыши на пункте Data Connections (Подключения данных) и в контекстном меню выберите пункт Add Connection (Добавить подключение).

В диалоговом окне Add Connection (Добавить подключение) в поле Server Name (Имя сервера) укажите значение `(localdb)\mssqllocaldb`, а в раскрывающемся списке выберите пункт aspnet-NorthwindMvc-GUID (рис. 15.11).

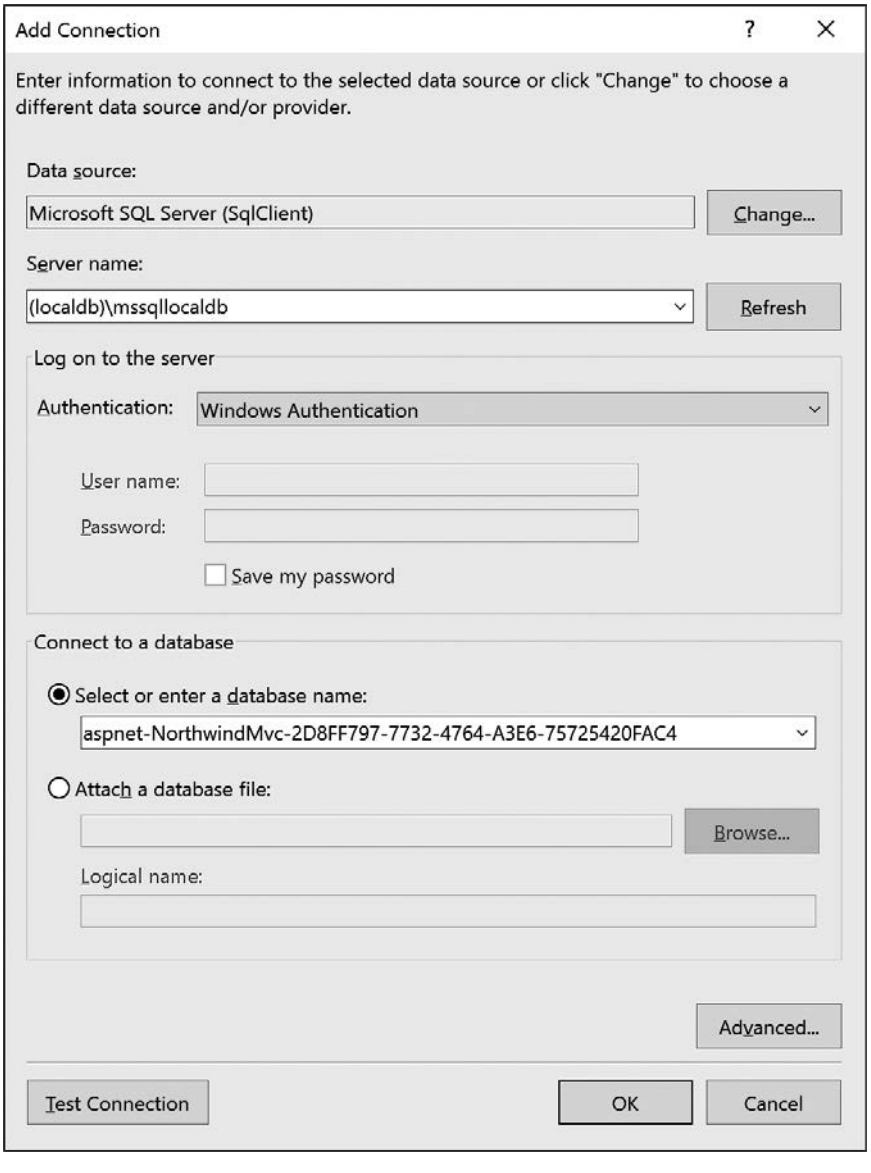


Рис. 15.11

На панели Server Explorer (Обозреватель серверов) раскройте дерево Tables (Таблицы), щелкните правой кнопкой мыши на таблице AspNetUsers и выберите

пункт Show Table Data (Показать таблицу данных). Обратите внимание на строку, добавленную в базу данных после того, как вы заполнили регистрационную форму (рис. 15.12).

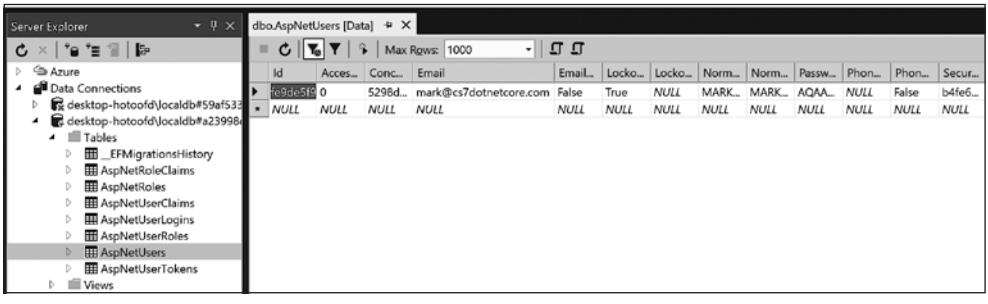


Рис. 15.12

Очень хорошо, что в проектах веб-приложений ASP.NET Core хранится только хеш паролей вместо них самих (об этом говорится в главе 10). Кроме того, в системе ASP.NET Core Identity можно реализовать процедуру двухфакторной аутентификации.

Закройте таблицу, а затем на панели Server Explorer (Обозреватель серверов) щелкните правой кнопкой мыши на подключении к базе данных и выберите пункт Close Connection (Завершить подключение).

Структура сайта ASP.NET Core MVC

Рассмотрим компоненты, составляющие современное приложение ASP.NET Core MVC.

Запуск ASP.NET Core

Откройте файл `Startup.cs`.

Обратите внимание на показанный в следующем листинге метод `ConfigureServices`, который добавляет поддержку MVC наряду с другими сервисами платформы и приложений, такими как ASP.NET Identity.

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration
    .GetConnectionString("DefaultConnection")));

  services.AddIdentity<ApplicationUser, IdentityRole>()
```

```
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

  // Добавление сервисов приложения.
  services.AddTransient<IEmailSender, EmailSender>();

  services.AddMvc();
}
```

Теперь у нас есть метод `Configure`, код которого показан ниже:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
    app.UseDatabaseErrorPage();
  }
  else
  {
    app.UseExceptionHandler("/Home/Error");
  }

  app.UseStaticFiles();

  app.UseAuthentication();

  app.UseMvc(routes =>
  {
    routes.MapRoute(
      name: "default",
      template: "{controller=Home}/{action=Index}/{id?}");
  });
}
```

Обратите внимание на следующие аспекты.

- Если сайт запущен из среды разработки, то:
  - при вызове исключения отображается страница с ошибкой, содержащая исходный код;
  - отображается страница с ошибкой базы данных.
- Если сайт запущен в рабочей среде, то посетитель перенаправляется в каталог `/Home/Error`.
- Включен доступ к статичным файлам, что позволяет использовать файлы CSS, JavaScript и т. д., хранящиеся в файловой системе.
- Активизирована система ASP.NET Identity для аутентификации и авторизации.
- Самая важная инструкция в данном примере отвечает за вызов `UseMvc` и сопоставление маршрута по умолчанию. Этот маршрут очень гибкий, поскольку будет сопоставляться практически с любым входящим URL, как вы увидите ниже.

Маршрутизация по умолчанию

Маршрутизация по умолчанию предусматривает анализ всех URL, введенных пользователем в адресной строке, и сопоставляет их для извлечения имен контроллера и действия и дополнительного значения `id` (таковым оно является благодаря символу `?`). Если пользователь не указал эти данные, то применяются значения по умолчанию: `Home` в качестве контроллера и `Index` в качестве действия (присвоение через символ `=` устанавливает значение по умолчанию для сегмента с именем).

Содержимое в фигурных скобках `{}` называется *сегментами* и играет роль именованных параметров метода. В качестве значения этих сегментов может выступать любая строка.

Маршрутизация отвечает за определение имен контроллера и действия.

В табл. 15.1 приведены примеры URL и определения имен MVC.

Таблица 15.1

URL-адрес	Контроллер	Действие	id
/	Home	Index	
/Muppet	Muppet	Index	
/Muppet/Kermit	Muppet	Kermit	
/Muppet/Kermit/Green	Muppet	Kermit	Green
/Products	Products	Index	
/Products/Detail	Products	Detail	
/Products/Detail/3	Products	Detail	3

Обратите внимание: если пользователь не укажет имя, то по умолчанию будут применяться значения `Home` и `Index`, как указано при регистрации маршрута. В случае необходимости можно изменить значения по умолчанию.

Контроллеры ASP.NET Core MVC

Теперь, когда ASP.NET Core MVC из сведений о маршруте и URL известны имена контроллера и действия, понадобится класс, способный реализовать интерфейс `IController`. Чтобы упростить задачу, Microsoft предоставляет класс `Controller`, который могут наследовать ваши классы.

Обязанности контроллера таковы:

- извлечение параметров из HTTP-запроса;
- использование параметров для выборки правильной модели и ее передачи в правильное представление, которое впоследствии передастся клиенту через HTTP-ответ;
- возвращение клиенту результата из представления в качестве HTTP-ответа.

Откройте каталог `Controllers` и дважды щелкните на файле `HomeController.cs`. Вы увидите его содержимое, показанное ниже:

```
public class HomeController : Controller
{
  public IActionResult Index()
  {
    return View();
  }

  public IActionResult About()
  {
    ViewData["Message"] = "Your application description page.";
    return View();
  }

  public IActionResult Contact()
  {
    ViewData["Message"] = "Your contact page.";
    return View();
  }

  public IActionResult Error()
  {
    return View(new ErrorViewModel {
      RequestId = Activity.Current?.Id ??
      HttpContext.TraceIdentifier });
  }
}
```

Введенные пользователем / или /Home равносильны /Home/Index, поскольку это значения по умолчанию.

Обратите внимание на следующие аспекты:

- в настоящее время ни один из методов-действий не использует модель;
- два метода-действия используют словарь `ViewData` для хранения элемента `string` с именем `Message`, который затем можно прочитать в представлении;
- все методы-действия вызывают метод `View()` и возвращают клиенту результаты в виде интерфейса `IActionResult`.

Модели ASP.NET Core MVC

В ASP.NET Core MVC модель представляет собой данные, необходимые для запроса. К примеру, GET-запрос через протокол HTTP ресурса по адресу http://www.example.com/products/details/3 может означать, что браузер запрашивает сведения о товаре под номером 3.

Контроллер должен будет использовать значение 3 идентификатора (ID), чтобы извлечь запись с этим товаром и передать ее в представление, которое затем может превратить модель в HTML-код для отображения в браузере.

Конфигурация модели данных с использованием сущностей EF Core

Мы будем ссылаться на модель данных Entity Framework Core для прямого доступа к базе данных Northwind, которую вы создали в главе 14.

В Visual Studio 2017 на панели Solution Explorer (Обозреватель решений) раскройте проект NorthwindMvc, щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Add Reference (Добавить ссылку). В открывшемся диалоговом окне выберите пункт NorthwindContextLib и нажмите кнопку OK.

В Visual Studio Code откройте проект NorthwindMvc и измените файл `Northwind-Mvc.csproj`, добавив ссылку на проект `NorthwindContextLib`.

Такие сервисы, как Entity Framework Core, необходимые контроллерам MVC, во время запуска должны быть зарегистрированы в виде сервисов.

В Visual Studio 2017 или Visual Studio Code измените файл `Startup.cs`, чтобы добавить в метод `ConfigureServices` следующие инструкции.

Для SQL Server LocalDb:

```
services.AddDbContext<Packt.CS7.Northwind>(options => options.
UseSqlServer("Server=(localdb)\\mssqllocaldb;" + "Database=Northwind;Trusted_
Connection=True;" + "MultipleActiveResultSets=true"));
```

Для SQLite:

```
services.AddDbContext<Packt.CS7.Northwind>(options => options.UseSqlite("Data
Source=../Northwind.db"));
```

Создание моделей представления для запросов

Представьте, что, когда пользователь заходит на наш сайт, мы хотим вывести ему список товаров и количество посетителей за текущий месяц. Все данные, которые мы хотим отобразить в ответ на запрос, — это модель MVC, иногда называемая *моделью представления*, поскольку являет собой *модель*, которая передается в *представление*.

Добавьте класс в каталог `Models` и присвойте ему имя `HomeIndexViewModel`.

Измените определение класса, как показано в листинге ниже:

```
using System.Collections.Generic;

namespace Packt.CS7
{
  public class HomeIndexViewModel
  {
    public int VisitorCount;
    public IList<Category> Categories { get; set; }
    public IList<Product> Products { get; set; }
  }
}
```

Выборка модели в контроллере

Откройте класс `HomeController`.

Импортируйте пространство имен `Packt.CS7`.

Добавьте поле для хранения ссылки на экземпляр `Northwind` и инициализируйте его в конструкторе таким образом:

```
private Northwind db;

public HomeController(Northwind injectedContext)
{
  db = injectedContext;
}
```

Измените содержимое метода действия `Index`, как показано в следующем листинге:

```
var model = new HomeIndexViewModel
{
  VisitorCount = (new Random()).Next(1, 1001),
  Categories = db.Categories.ToList(),
  Products = db.Products.ToList()
};
return View(model); // передача модели представлению
```

Мы смоделируем счетчик посетителей, используя класс Random для генерации числа в диапазоне от 1 до 1000.

Представления ASP.NET Core MVC

Представление несет ответственность за преобразование модели в HTML-код и другие форматы. Чтобы совершить преобразование, можно использовать несколько *обработчиков представления*. Такой обработчик по умолчанию на платформе ASP.NET MVC 3 и последующих версий называется *Razor* и применяет символ `@` для указания на выполнение кода на стороне сервера.

Новая функция Razor Pages в ASP.NET Core 2.0 задействует тот же обработчик представления и поэтому поддерживает схожий синтаксис Razor.

Отображение представлений контроллера Home

Откройте каталог `Views`, а затем каталог `Home`. Обратите внимание на три файла с расширением `.cshtml`.

Такое расширение обозначает, что документ содержит код на языках C# и HTML.

Когда метод `View()` вызывается в методе-действии контроллера, ASP.NET Core MVC сканирует каталог `Views` на наличие подкаталога с тем же именем, что и у текущего контроллера, то есть `Home`. Затем выполняется поиск файла с тем же именем, что и у текущего действия, то есть `Index`, `About` или `Contact`.

Откройте файл `Index.cshtml` и обратите внимание на блок кода C#, обернутый в `@{ }`. Он будет выполняться самым первым и может использоваться для хранения данных, которые необходимо передать в общий файл макета:

```
@{
  ViewData["Title"] = "Home Page";
}
```

Обратите внимание на статичный HTML-контент в нескольких элементах `<div>`, для форматирования которых используется библиотека Bootstrap.

При необходимости определить собственные стили старайтесь создавать их на основе универсальной общей библиотеки, такой как Bootstrap, реализующей принципы адаптивного дизайна. Чтобы узнать больше о CSS3 и адаптивном дизайне, прочитайте книгу Responsive Web Design with HTML5 and CSS3 — Second Edition (HTML5 и CSS3. Разработка сайтов для любых браузеров и устройств) Бена Фрейна (Ben Frain), перейдя по ссылке https://www.packtpub.com/web-development/responsive-web-design-html5-and-css3-second-edition.

Совместное использование макетов между представлениями

Так же как в случае со страницами Razor, файл `_ViewStart.cshtml` запускается с помощью метода `View()`. Он служит для установки значений по умолчанию, которые применяются ко всем представлениям.

К примеру, он настраивает свойство `Layout` всех представлений на отображение общего файла макета.

```
@{
  Layout = "_Layout";
}
```

Найдите в каталоге `Shared` файл `_Layout.cshtml` и откройте его. Обратите внимание, что название считывается из словаря `ViewData`, который был установлен ранее в представлении `Index.cshtml`.

Обратите внимание на присутствие инструкций для загрузки общих стилей для поддержки Bootstrap и двух разделов. Во время *разработки* будут использоваться полностью прокомментированные и хорошо отформатированные версии CSS-файлов. Для *промежуточной* и *финальной* версии применяются минимизированные версии.

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

```
<environment exclude="Development">
  <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/
bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
    asp-fallback-test-class="sr-only"
    asp-fallback-test-property="position"
    asp-fallback-test-value="absolute" />
  <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

Символом ~ обозначается каталог wwwroot.

Обратите внимание на отображение гиперссылок, позволяющих пользователям переходить между страницами с помощью панели навигации в верхней части каждой страницы. Элементы `<a>` задействуют специальные атрибуты (так называемые вспомогательные функции *тегов*) для указания имен контроллера и действия, применяемых при переходе по ссылке:

```
<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li><a asp-controller="Home" asp-action="Index">Home</a></li>
    <li><a asp-controller="Home" asp-action="About">About</a></li>
    <li><a asp-controller="Home" asp-action="Contact">Contact</a></li>
  </ul>
</div>
```

Так выглядит инструкция для отображения тела страницы:

```
@RenderBody()
```

Обратите внимание на инструкции для отображения блоков сценариев в конце страницы, призванные воспрепятствовать замедлению загрузки страницы:

```
<environment include="Development">
  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
  <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment exclude="Development">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
    asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
    asp-fallback-test="window.jQuery">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/bootstrap.min.js"
    asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.
fn.modal">
  </script>
    <script src="~/js/site.min.js" asp-append-version="true">
  </script>
</environment>
```

Вы можете добавить собственные блоки сценариев в дополнительный раздел `scripts`:

```
@RenderSection("scripts", required: false)
```

Определение пользовательских стилей

В каталоге `wwwroot\css` найдите и откройте файл `site.css`.

Создайте стиль, который будет применен к элементу с идентификатором `newspaper`, например, так:

```
#newspaper {
  column-count: 3;
}
```

В Visual Studio Code нужно будет добавить стиль и в файл site.min.css. Как правило, на этапе сборки можно сократить файл site.css до версии site.min.css, однако на данный момент это нужно сделать вручную.

Определение типизированного представления

Чтобы улучшить поведение IntelliSense при написании кода представления, с помощью директивы `@model` можно определить тип, который будет ожидаться этим представлением.

В папке `Views\Home` найдите и откройте представление `Index.cshtml` и добавьте в первой строке файла инструкцию, присваивающую типу модели значение `HomeIndexViewModel`, как показано ниже:

```
@model Packt.CS7.HomeIndexViewModel
```

Теперь каждый раз при вводе в данном представлении значения `@Model` IntelliSense будет знать, какой тип за этим скрывается, и сможет предоставлять соответствующие подсказки.

Для объявления типа для модели используйте ключевое слово @model (со строчной буквой m).

Для чтения из модели применяйте ключевое слово @Model (с прописной буквой M).

В файле `Index.cshtml` измените элемент `<div>` карусели, удалите все остальные такие элементы и замените их новой разметкой для вывода товаров в виде неупорядоченного списка, как показано в листинге ниже:

```
@model Packt.CS7.HomeIndexViewModel
@{
    ViewData["Title"] = "Home Page";
  }
  <div id="myCarousel" class="carousel slide" data-ride="carousel"
```

```
data-interval="6000">
 <ol class="carousel-indicators">
   @for (int c = 0; c < Model.Categories.Count; c++)
   {
     if (c == 0)
     {
       <li data-target="#myCarousel" data-slide-to="@c" class="active"></li>
     }
     else
     {
       <li data-target="#myCarousel" data-slide-to="@c"></li>
     }
   }
 </ol>
 <div class="carousel-inner" role="listbox">
   @for (int c = 0; c < Model.Categories.Count; c++)
   {
     if (c == 0)
     {
       <div class="item active">
         <img src="~/images/category@ (Model.Categories[c].CategoryID).jpeg"
         alt="@Model.Categories[c].CategoryName" class="img-responsive" />
         <div class="carousel-caption" role="option">
           <p>
             @Model.Categories[c].Description
             <a class="btn btn-default" href="/category/
             @Model.Categories[c].CategoryID">
               @Model.Categories[c].CategoryName
             </a>
           </p>
         </div>
       </div>
     }
     else
     {
       <div class="item">
         <img src="~/images/category@
            (Model.Categories[c].CategoryID).jpeg"
             alt="@Model.Categories[c].CategoryName"
             class="img-responsive" />
             <div class="carousel-caption" role="option">
               <p>
                 @Model.Categories[c].Description
                 <a class="btn btn-default" href="/category/
                 @Model.Categories[c].CategoryID">
                   @Model.Categories[c].CategoryName
                 </a>
               </p>
             </div>
           </div>
     }
   }
 </div>
 <a class="left carousel-control" href="#myCarousel" role="button"
```

```
     data-slide="prev">
      <span class="glyphicon glyphicon-chevron-left"
       aria-hidden="true"></span>
      <span class="sr-only">Previous</span>
    </a>
    <a class="right carousel-control" href="#myCarousel" role="button"
     data-slide="next">
      <span class="glyphicon glyphicon-chevron-right" aria-hidden="true">
      </span>
      <span class="sr-only">Next</span>
    </a>
  </div>
  <div class="row">
    <div class="col-md-12">
      <h1>Northwind</h1>
      <p class="lead">
        We have had @Model.VisitorCount visitors this month.
      </p>
      <h2>Products</h2>
      <div id="newspaper">
       <ul>
        @foreach (var item in @Model.Products)
        {
          <li>
            <a asp-controller="Home"
             asp-action="ProductDetail"
             asp-route-id="@item.ProductID">
             @item.ProductName costs
             @item.UnitPrice.Value.ToString("C")
            </a>
          </li>
      }
    </ul>
   </div>
 </div>
</div>
```

Обратите внимание, как просто смешивать статичные HTML-элементы, такие как `ul` и `li` с кодом на языке C# для вывода списка имен товаров.

Обратите внимание на элемент `<div>` с идентификатором `newspaper`. К нему применен пользовательский стиль, определенный нами ранее, поэтому все содержимое данного элемента будет выведено в три колонки.

В папку `wwwroot/images` добавьте восемь графических файлов с именами `category1.jpeg`, `category2.jpeg` и т. д. вплоть до `category8.jpeg`.

Изображения для выполнения примеров из этой книги можно загрузить из репозитория Github, перейдя по ссылке https://github.com/markjprice/cs7dotnetcore2/tree/master/Assets. Для поиска подходящих изображений для восьми категорий я задействовал поиск бесплатных и допустимых для коммерческого использования графических файлов на сайте https://www.pexels.com.

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+F5.

В Visual Studio Code выполните команду `dotnet run`, затем откройте браузер Chrome и перейдите по адресу http://localhost:5000/.

На главной странице будут представлены вращающаяся карусель с указанием категорий и список товаров в трех колонках (рис. 15.13).

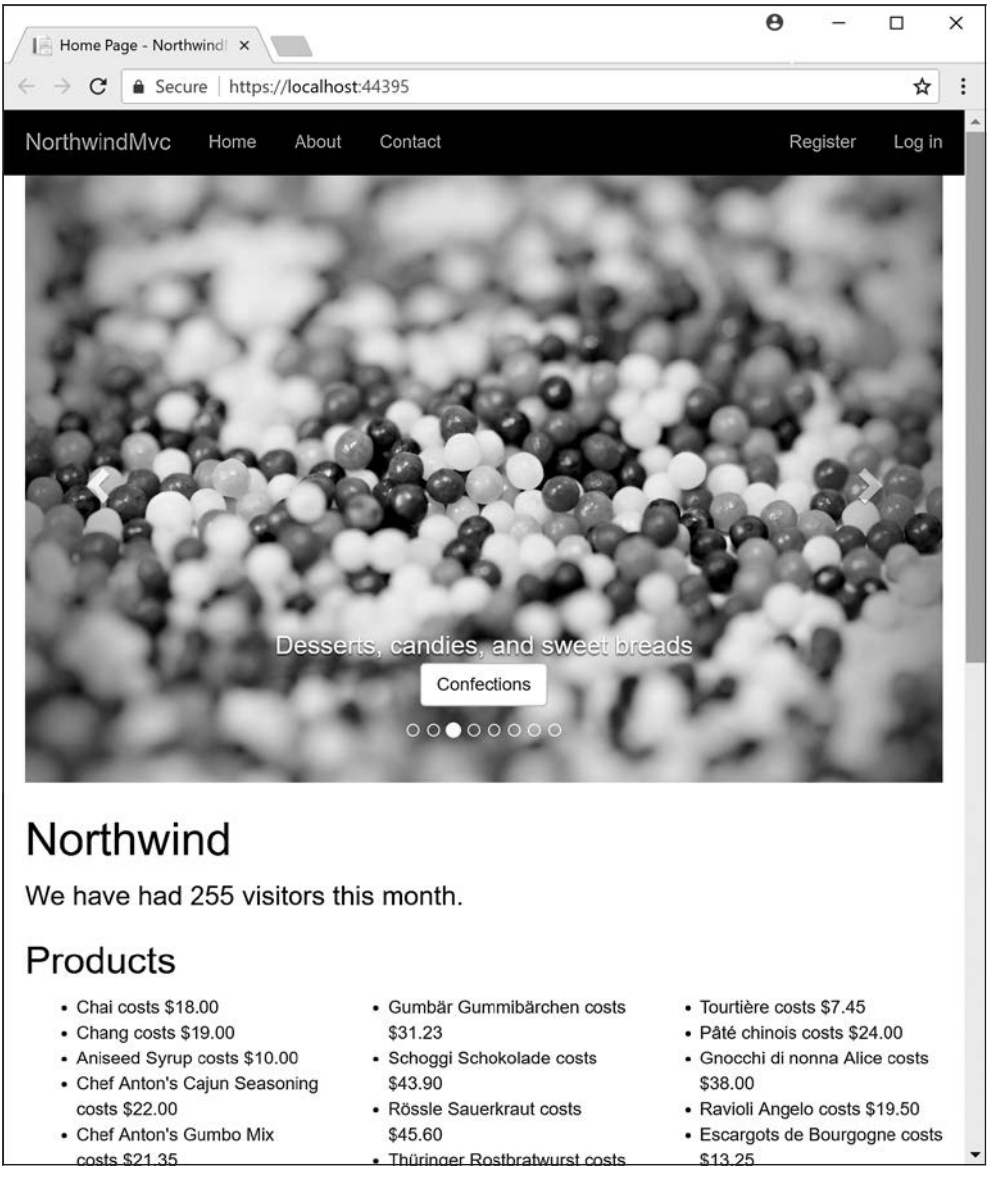


Рис. 15.13

Если вы в данный момент щелкнете на любой ссылке на товар, то увидите страницу с ошибкой `404` (рис. 15.14).

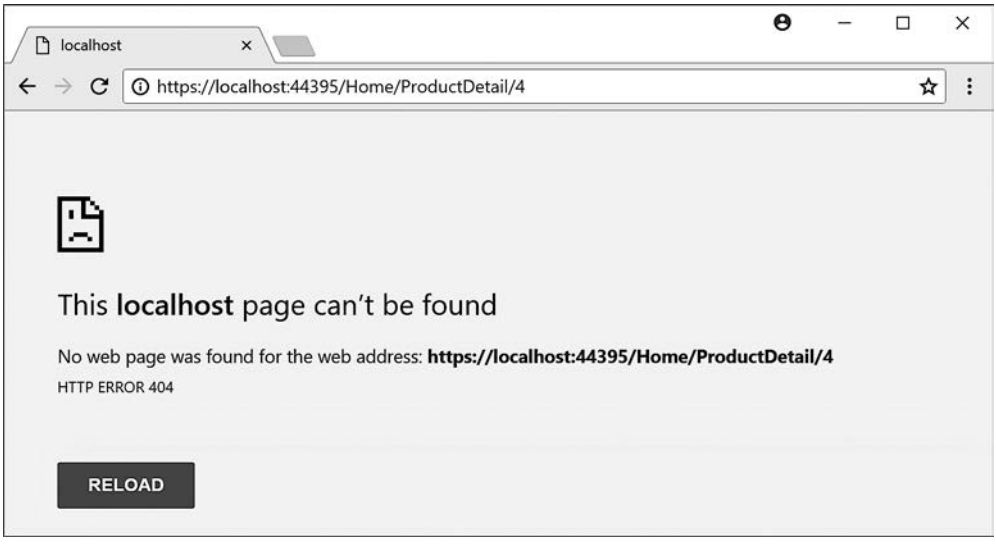


Рис. 15.14

Теперь, когда вы познакомились с принципами совместной работы моделей, представлений и контроллеров для реализации веб-приложения, рассмотрим некоторые распространенные сценарии, такие как передача параметров, призванные отобразить подробную информацию о товаре.

Передача параметров с помощью значения маршрута

Вернемся к классу `HomeController` и добавим показанный ниже метод действия `ProductDetail`.

```
public IActionResult ProductDetail(int? id)
{
  if (!id.HasValue)
  {
    return NotFound("You must pass a product ID in the route, for example, /Home/
ProductDetail/21");
  }
  var model = db.Products.SingleOrDefault(p => p.ProductID == id);
  if (model == null)
  {
    return NotFound($"A product with the ID of {id} was not found.");
  }
  return View(model); // передача модели представлению
}
```

Обратите внимание на следующие аспекты.

- В данном методе используется функция ASP.NET Core под названием *«привязка модели»* для автоматического соотнесения значения переменной `id`, переданной в пути, с параметром `id` в методе.
- Внутри метода мы проверяем, не присвоено ли переменной `id` значение `null`, и если да, то метод возвращает код статуса `404` и соответствующее сообщение.
- В противном случае подключаемся к БД и пытаемся получить товар, используя переменную `id`.
- Если искомый товар найден, то передаем его в представление, в противном случае возвращаем код статуса `404` и соответствующее сообщение.

Привязки моделей — очень мощные инструменты, и уже привязка, реализованная по умолчанию, предоставляет очень широкий функционал. Для более сложных сценариев можно создать собственную привязку, реализовав интерфейс IModelBinder, однако данная тема выходит за рамки книги.

Теперь нужно создать представление для этого запроса.

В Visual Studio 2017 откройте каталог `Views`, щелкните правой кнопкой мыши на каталоге Home и выполните команду Add ▸ New Item (Добавить ▸ Новый элемент). В открывшемся окне выберите пункт MVC View Page (Страница представления MVC) и задайте имя `ProductDetail.cshtml`.

В Visual Studio Code в каталоге `Views/Home` создайте файл с именем `ProductDetail.cshtml`.

Измените содержимое созданного файла, как показано в коде ниже:

```
@model Packt.CS7.Product
@{
    ViewData["Title"] = "Product Detail - " + Model.ProductName;
}
<h2>Product Detail</h2>
<hr />
<div>
  <dl class="dl-horizontal">
    <dt>Product ID</dt>
    <dd>@Model.ProductID</dd>
    <dt>Product Name</dt>
    <dd>@Model.ProductName</dd>
    <dt>Category ID</dt>
    <dd>@Model.CategoryID</dd>
    <dt>Unit Price</dt>
    <dd>@Model.UnitPrice.Value.ToString("C")</dd>
    <dt>Units In Stock</dt>
    <dd>@Model.UnitsInStock</dd>
  </dl>
</div>
```

Запустите веб-приложение и, когда появится главная страница со списком товаров, щелкните на одном из них, к примеру на товаре под номером 2, *Chang*. Результат должен выглядеть примерно так (рис. 15.15).

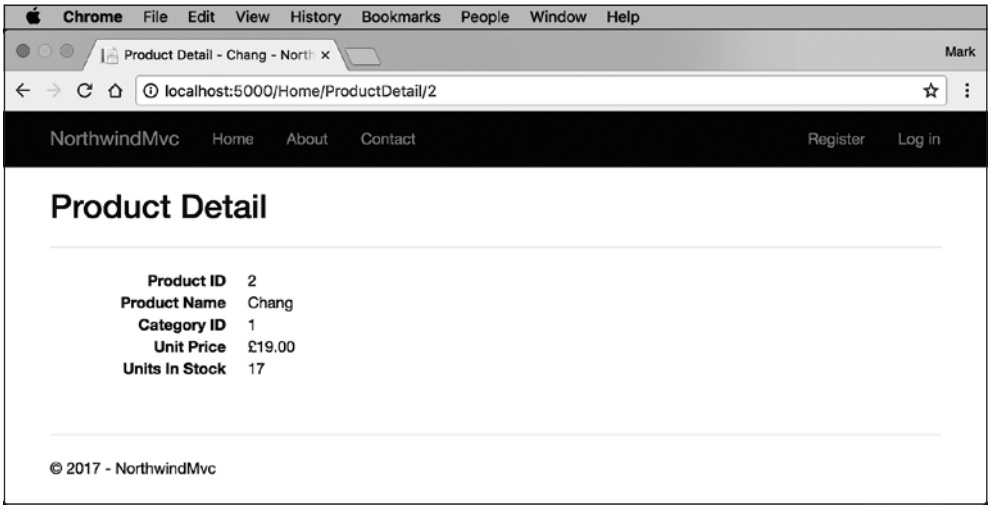


Рис. 15.15

Передача параметров с помощью строки запроса

В классе `HomeController` импортируйте пространство имен `Microsoft.EntityFrameworkCore`. Это необходимо для добавления метода расширения `Include`, позволяющего включать связанные сущности.

Добавьте метод действия, как показано в следующем листинге:

```
public IActionResult ProductsThatCostMoreThan(decimal? price)
{
  if (!price.HasValue)
  {
    return NotFound("You must pass a product price in the query string, for
example, /Home/ProductsThatCostMoreThan?price=50");
  }
  var model = db.Products.Include(p => p.Category).Include(
  p => p.Supplier).Where(p => p.UnitPrice > price).ToArray();
  if (model.Count() == 0)
  {
    return NotFound($"No products cost more than {price:C}.");
  }
  ViewData["MaxPrice"] = price.Value.ToString("C");
  return View(model); // передача модели представлению
}
```

В каталоге `Views/Home` создайте файл с именем `ProductsThatCostMoreThan.cshtml`. Измените его содержимое, как показано в коде, приведенном ниже:

```
@model IEnumerable<Packt.CS7.Product>
@{
    ViewData["Title"] = "Products That Cost More Than " + ViewData["MaxPrice"];
}
<h2>Products That Cost More Than @ViewData["MaxPrice"]</h2>
<table class="table">
  <tr>
    <th>Category Name</th>
    <th>Supplier's Company Name</th>
    <th>Product Name</th>
    <th>Unit Price</th>
    <th>Units In Stock</th>
  </tr>
  @foreach (var item in Model)
  {
     <tr>
       <td>
         @Html.DisplayFor(modelItem => item.Category.CategoryName)
       </td>
       <td>
         @Html.DisplayFor(modelItem => item.Supplier.CompanyName)
       </td>
       <td>
         @Html.DisplayFor(modelItem => item.ProductName)
       </td>
       <td>
         @Html.DisplayFor(modelItem => item.UnitPrice)
       </td>
       <td>
         @Html.DisplayFor(modelItem => item.UnitsInStock)
     </td>
  </tr>
}
</table>
```

В каталоге `Views/Home` найдите и откройте файл `Index.cshtml` и добавьте в его конец (но выше заголовка Products и списка товаров) указанный ниже элемент `form`. Благодаря ему пользователь сможет указать цену. Затем он может нажать кнопку отправки данных, чтобы вызвать метод действия, который выведет только те товары, чья цена превышает введенное значение:

```
<form asp-action="ProductsThatCostMoreThan" method="get">
  <input name="price" placeholder="Enter a product price" />
  <input type="submit" />
</form>
```

Запустите веб-приложение и на главной странице, прокрутив ее вниз, введите в форму цену товара, например `50`, а затем нажмите кнопку Submit (Отправить) (рис. 15.16).

Вы увидите таблицу товаров, цена которых превышает введенное значение (рис. 15.17).

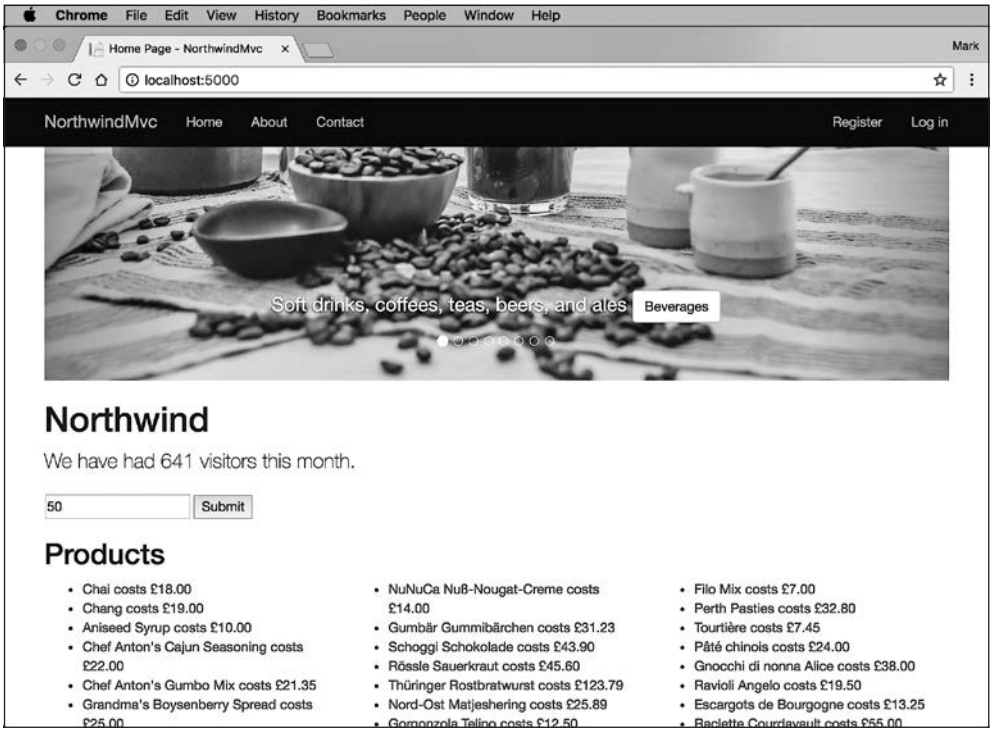


Рис. 15.16

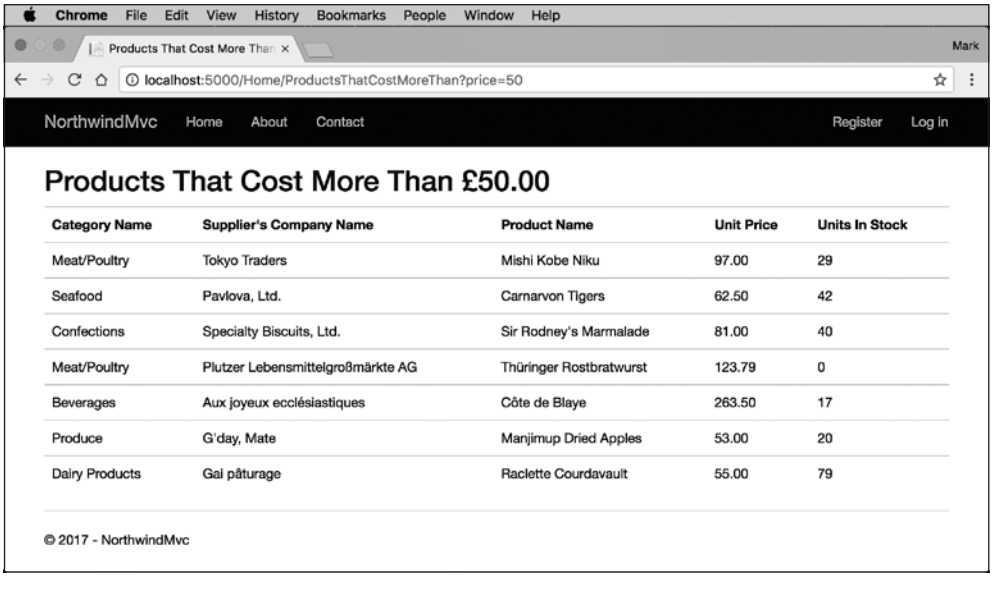


Рис. 15.17

Практические задания

Проверьте полученные знания. Для этого выполните приведенное упражнение и посетите указанные ресурсы, чтобы найти дополнительную информацию по темам данной главы.

Упражнение 15.1. Практика улучшения масштабируемости за счет понимания и реализации асинхронных методов

Несколько лет назад Стивен Клири (Stephen Cleary) написал отличную статью для журнала MSDN Magazine, в которой объяснил преимущества масштабируемости реализации асинхронных методов для ASP.NET. Те же самые принципы применимы для ASP.NET Core, причем даже в большей степени, поскольку в отличие от старой платформы ASP.NET, ASP.NET Core поддерживает асинхронные фильтры и другие компоненты.

Прочитайте статью, перейдя по ссылке https://msdn.microsoft.com/en-us/magazine/dn802603.aspx. Измените приложение, созданное вами в этой главе, таким образом, чтобы в новой версии в контроллере использовались асинхронные методы.

Дополнительные ресурсы

- Общие сведения о ASP.NET Core MVC: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview.
- Начало работы с ASP.NET Core MVC и Entity Framework Core в Visual Studio: https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro.
- Обработка запросов с контроллеров в ASP.NET Core MVC: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/actions.
- Привязка модели: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding.
- Представления в ASP.NET Core MVC: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview.

Резюме

В этой главе вы научились создавать сложные сайты с помощью ASP.NET Core MVC.

В следующей главе вы узнаете, как разрабатывать веб-приложения с помощью серверных веб-сервисов, созданных с применением ASP.NET Core и таких технологий, как Angular и React.

16 Создание веб-сервисов и приложений с помощью ASP.NET Core

Эта глава посвящена созданию веб-приложений на базе сочетания серверных сервисов, для разработки которых применяются веб-API ASP.NET Core, и клиентских *одностраничных приложений* (single page applications, SPA), проектируемых с помощью Angular или React.

В данной главе:

- создание сервисов с помощью веб-API ASP.NET Core и Visual Studio Code;
- документирование и тестирование служб с применением Swagger;
- создание SPA с помощью Angular;
- использование других шаблонов проектов.

Создание сервисов с помощью веб-API ASP.NET Core и Visual Studio Code

Хотя изначально протокол HTTP проектировался для передачи запросов и ответов в виде HTML-документов и других наглядных ресурсов, с помощью него также можно создавать сервисы. Рой Филдинг (Roy Fielding), один из главных авторов спецификации данного протокола, в своей диссертации «Архитектурные стили и дизайн сетевых программных архитектур» описал основные моменты «передачи состояния представления» (representational state transfer, REST) через протокол HTTP:

- URL для уникальной идентификации ресурсов;
- методы для выполнения общих задач, таких как `GET`, `POST`, `PUT` и `DELETE`;
- возможность согласовывать медиаформаты, например XML и JSON.

Веб-сервисы — сервисы, использующие протокол обмена сообщениями HTTP, поэтому иногда их еще называют сервисами HTTP или RESTful.

Обзор контроллеров ASP.NET Core

Для того чтобы упростить процедуру создания сервисов, разработчики платформы ASP.NET Core объединили то, что раньше было представлено двумя типами контроллеров.

В ранних версиях ASP.NET вы должны были выполнить наследование от `ApiController` для создания веб-API-сервиса и затем зарегистрировать маршруты API в той же таблице маршрутов, что и MVC.

С помощью платформы ASP.NET Core вы используете тот же базовый класс `Controller`, что и в MVC, за исключением того, что маршруты настраиваются самим контроллером на основе атрибутов, а не таблицы маршрутов.

Создание проекта веб-API ASP.NET Core

Мы создадим веб-сервис, позволяющий работать с базой данных `Northwind` с использованием ASP.NET Core, таким образом, что данные, хранящиеся в этой базе, можно применить на любой платформе, способной создавать HTTP-запросы и получать HTTP-ответы.

Visual Studio 2017

В Visual Studio 2017 откройте решение `Part3` и нажмите сочетание клавиш Ctrl+Shift+N или воспользуйтесь командой меню File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне Add New Project (Добавить новый проект) в списке Installed ▸ Templates (Установленные ▸ Шаблоны) раскройте раздел Visual C# и выберите пункт .NET Core. В центре диалогового окна выберите пункт ASP.NET Core Web Application (Веб-приложение ASP.NET Core), введите имя `NorthwindService` и нажмите кнопку OK.

В диалоговом окне New ASP.NET Core Web Application — NorthwindService (Создать веб-приложения ASP.NET Core — NorthwindService) выберите пункты раскрывающихся меню .NET Core и ASP.NET Core 2.0, после чего нажмите кнопку OK.

Visual Studio Code

В каталоге `Part3` создайте папку `NorthwindService` и откройте ее в Visual Studio.

На панели Integrated Terminal (Интегрированный терминал) введите такую команду для создания нового проекта веб-API ASP.NET Core:

```
dotnet new webapi
```

Использование Visual Studio 2017 и Visual Studio Code

В папке `Controllers` найдите и откройте файл `ValuesController.cs` и обратите внимание на следующие аспекты.

- Атрибут `[Route]` регистрирует относительный URL `/api/values` для того, чтобы клиенты могли создавать HTTP-запросы, обрабатываемые данным контроллером. Базовый маршрут `/api/`, после которого указывается имя контроллера, — это соглашение, позволяющее дифференцировать технологии MVC и Web-API. Впрочем, вам не обязательно придерживаться его. Если вы используете нотацию `[controller]`, как показано, то система задействует символы перед словом `Controller` в имени класса, кроме того, вы также можете просто ввести другое имя между квадратными скобками.
- Атрибут `[HttpGet]` регистрирует метод `Get`, отвечающий на HTTP-запросы `Get`, и возвращает массив значений типа `string`.
- Атрибут `[HttpGet]` с параметром регистрирует метод `Get` с параметром `id` для ответа на HTTP-запросы `Get`, содержащие в маршруте параметрическое значение.
- Атрибуты `[HttpPost]`, `[HttpPut]` и `[HttpDelete]` регистрируют три других метода для ответа на соответствующие HTTP-запросы, но на данный момент эти методы не выполняют никаких действий:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

namespace NorthwindService.Controllers
{
  [Route("api/[controller]")]
  public class ValuesController : Controller
  {
    // GET api/values
     [HttpGet]
    public IEnumerable<string> Get()
    {
      return new string[] { "value1", "value2" };
    }

    // GET api/values/5
     [HttpGet("{id}")]
    public string Get(int id)
    {
      return "value";
    }

    // POST api/values
     [HttpPost]
    public void Post([FromBody]string value)
```

```
        {
        }

        // PUT api/values/5
         [HttpPut("{id}")]
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE api/values/5
         [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
      }
  }
```

Если вы ранее использовали предыдущие версии веб-API ASP.NET Core для .NET Framework, то знаете, что можете создавать любые методы C#, начинающиеся с любого метода HTTP (GET, POST, PUT и т. д.), а контроллер автоматически выполнит нужный. В ASP.NET Core это уже не работает, так как мы больше не наследуем от ApiController. Таким образом, необходимо применить атрибуты наподобие [HttpGet] для явного соотнесения методов HTTP с методами C#. Несмотря на то что такой подход требует написания большего количества кода, он позволяет задавать любые имена методам контроллера.

Измените метод `GET`, чтобы он возвращал клиенту сообщение о том, какое значение параметра `id` отправил на сервер, как показано в листинге ниже:

```
// GET api/values/5
[HttpGet("{id}")]
public string Get(int id)
{
  return $"You sent me the id: {id}";
}
```

Запустите сайт.

В Visual Studio 2017 нажмите сочетание клавиш Ctrl+F5.

В Visual Studio Code введите команду `dotnet run`, а затем запустите Chrome и перейдите по адресу http://localhost:5000/api/values.

В браузере Chrome откройте Developer tools (Инструменты разработчика) и нажмите клавишу F5, чтобы обновить страницу.

Сервис веб-API должен вернуть документ JSON (рис. 16.1).

В Visual Studio 2017 не нужно вводить относительный URL /api/values, так как свойства проекта, указанные в категории Debug (Отладка), уже были настроены на выполнение этой работы за вас (рис. 16.2).

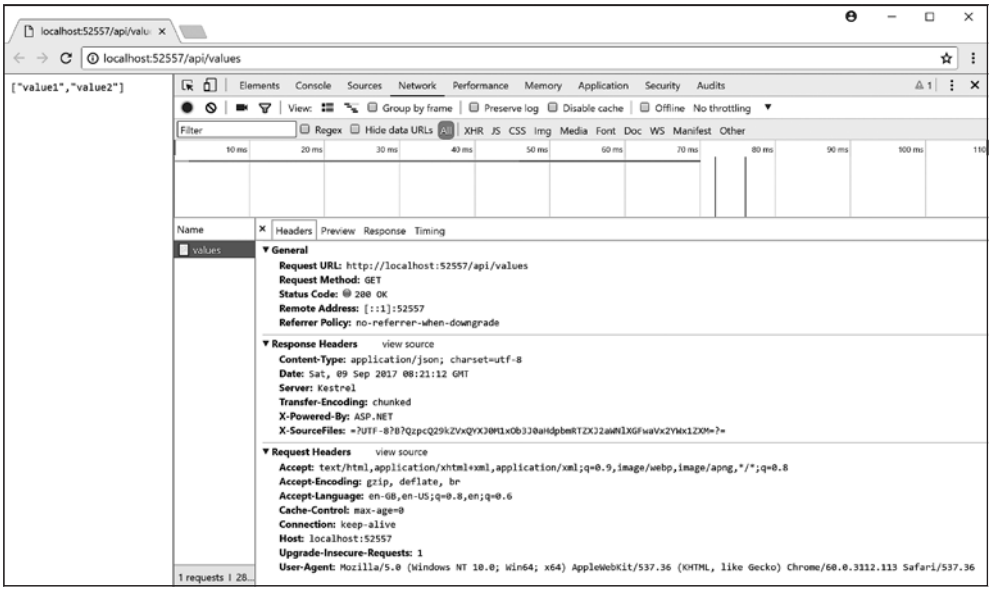


Рис. 16.1

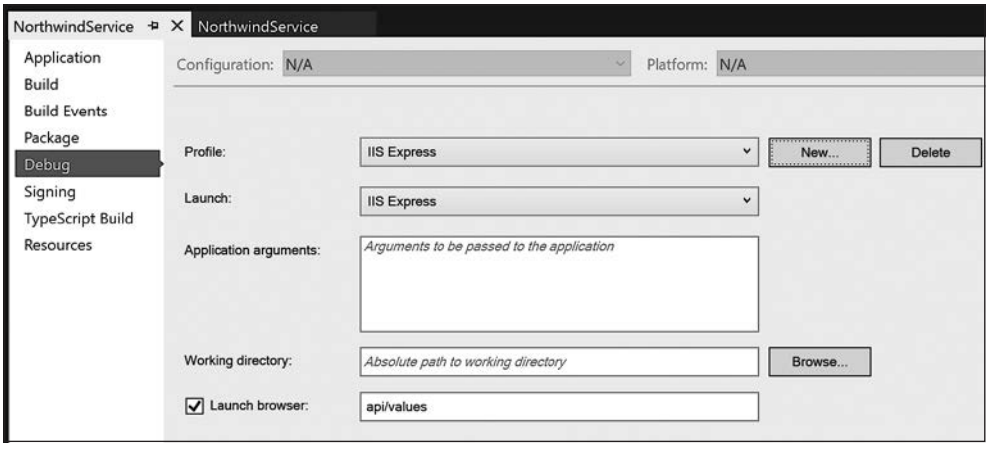


Рис. 16.2

Перейдите по адресу http://localhost:5000/api/values/42 и обратите внимание на ответ (рис. 16.3).

Закройте Google Chrome.

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) нажмите сочетание клавиш Ctrl+C, чтобы закрыть консольное приложение и отключить сервер Kestrel, на котором расположен ваш сервис ASP.NET Core.

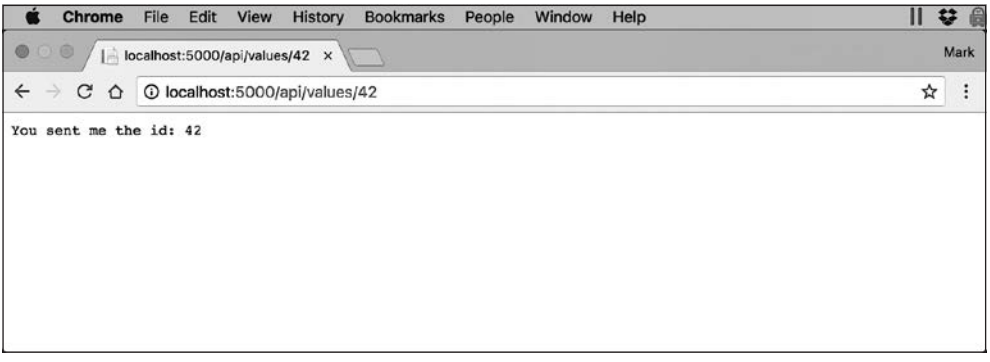


Рис. 16.3

Создание веб-сервиса для базы данных Northwind

В отличие от контроллеров для MVC контроллеры для веб-API не вызывают представления Razor с целью возвратить пользователям HTML-ответы в их браузеры. Вместо этого указанные контроллеры согласуют тип контента с клиентским приложением, которое выполнило HTTP-запрос, чтобы в HTTP-ответе корректно вернуть данные в одном из форматов: XML, JSON или X-WWW-FORMURLENCODED.

Клиентское приложение должно десериализовать данные из согласованного формата. В современных сервисах наиболее широко используется формат *JavaScript Object Notation* (JSON), поскольку он достаточно компактный и JavaScript отлично поддерживает его в браузере.

Мы будем ссылаться на сущностную модель Entity Framework Core для БД `Northwind`, созданную вами при прочтении главы 14.

Visual Studio 2017

В Visual Studio 2017 в проекте NorthwindService щелкните правой кнопкой мыши на категории Dependencies (Зависимости) и в контекстном меню выберите пункт Add Reference (Добавить ссылку). Выберите опцию NorthwindContextLib и нажмите кнопку OK.

Visual Studio Code

В Visual Studio Code в проекте NorthwindService откройте файл `NorthwindService.csproj` и добавьте ссылку из проекта на NorthwindContextLib, как показано в следующем листинге:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  <ProjectReference Include="..\NorthwindContextLib\NorthwindContextLib.csproj" />
</ItemGroup>
```

Visual Studio 2017 и Visual Studio Code

В Visual Studio 2017 и Visual Studio Code измените файл `Startup.cs`, импортируйте пространства имен `EntityFrameworkCore` и `Packt.CS7`, добавьте следующие инструкции в метод `ConfigureServices` перед вызовом `AddMvc`.

Для SQL Server LocalDB:

```
services.AddDbContext<Northwind>(options => options.UseSqlServer(
"Server=(localdb)\\mssqllocaldb;Database=Northwind;Trusted_Connection=True;
MultipleActiveResultSets=true"));
```

Для SQLite:

```
services.AddDbContext<Northwind>(options =>
options.UseSqlite("Data Source=../Northwind.db"));
```

Создание репозиториев данных для сущностей

Определение и реализация репозитория данных, позволяющего выполнять операции CRUD, — это рекомендуемое действие в программировании:

- C — Create (Создать);
- R — Retrieve/Read (Получить/Считать);
- U — Update (Обновить);
- D — Delete (Удалить).

Мы создадим репозиторий данных для таблицы Customers БД `Northwind`. При этом последуем современным рекомендациям программирования и сделаем наш репозиторий асинхронным относительно API.

В проекте NorthwindService создайте каталог `Repositories`.

Добавьте в этот каталог два файла класса с именами `ICustomerRepository.cs` и `CustomerRepository.cs`.

Файл `ICustomerRepository.cs` должен выглядеть следующим образом:

```
using Packt.CS7;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace NorthwindService.Repositories
{
  public interface ICustomerRepository
  {
    Task<Customer> CreateAsync(Customer c);

    Task<IEnumerable<Customer>> RetrieveAllAsync();

    Task<Customer> RetrieveAsync(string id);

    Task<Customer> UpdateAsync(string id, Customer c);

    Task<bool> DeleteAsync(string id);
  }
}
```

Файл `CustomerRepository.cs` должен выглядеть так:

```
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Packt.CS7;
using System.Collections.Generic;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading.Tasks;

namespace NorthwindService.Repositories
{
  public class CustomerRepository : ICustomerRepository
  {
    // кэшируем клиентов в потокобезопасном словаре для повышения
    // производительности
    private static ConcurrentDictionary<string, Customer> customersCache;

    private Northwind db;

    public CustomerRepository(Northwind db)
    {
      this.db = db;
      // предварительно загружаем клиентов из обычного словаря с ключом
      // CustomerID,
      // а затем конвертируем в потокобезопасный ConcurrentDictionary
      if (customersCache == null)
      {
        customersCache = new ConcurrentDictionary<string, Customer>(
        db.Customers.ToDictionary(c => c.CustomerID));
      }
    }

    public async Task<Customer> CreateAsync(Customer c)
    {
      // приводим CustomerID к верхнему регистру
      c.CustomerID = c.CustomerID.ToUpper();

      // добавляем в базу данных с помощью EF Core
      EntityEntry<Customer> added = await db.Customers.AddAsync(c);

      int affected = await db.SaveChangesAsync();

      if (affected == 1)
      {
        // если клиент новый, то добавляем в кэш, иначе вызываем
        // метод UpdateCache
        return customersCache.AddOrUpdate(c.CustomerID, c, UpdateCache);
      }
      else
      {
        return null;
      }
    }

    public async Task<IEnumerable<Customer>> RetrieveAllAsync()
```

```
{
  // для повышения производительности извлекаем из кэша
  return await Task.Run<IEnumerable<Customer>>(
  () => customersCache.Values);
}

public async Task<Customer> RetrieveAsync(string id)
{
  return await Task.Run(() =>
  {
    // для повышения производительности извлекаем из кэша
    id = id.ToUpper();
    Customer c;
    customersCache.TryGetValue(id, out c);
    return c;
  });
}

private Customer UpdateCache(string id, Customer c)
{
  Customer old;
  if (customersCache.TryGetValue(id, out old))
  {
    if (customersCache.TryUpdate(id, c, old))
    {
      return c;
    }
  }
  return null;
}

public async Task<Customer> UpdateAsync(string id, Customer c)
{
  return await Task.Run(() =>
  {
    // нормализуем идентификатор клиента
    id = id.ToUpper();
    c.CustomerID = c.CustomerID.ToUpper();

    // обновляем в базе данных
    db.Customers.Update(c);
    int affected = db.SaveChanges();
    if (affected == 1)
    {
      // обновляем в кэше
      return Task.Run(() => UpdateCache(id, c));
    }
    return null;
  });
}

public async Task<bool> DeleteAsync(string id)
{
  return await Task.Run(() =>
  {
```

```
        id = id.ToUpper();

        // удаляем из базы данных
        Customer c = db.Customers.Find(id);
        db.Customers.Remove(c);
        int affected = db.SaveChanges();
        if (affected == 1)
        {
          // удаляем из кэша
          return Task.Run(() => customersCache.TryRemove(id, out c));
        }
        else
        {
          return null;
        }
      });
    }
  }
}
```

Настройка и регистрация репозитория клиентов

Откройте файл `Startup.cs` и импортируйте данное пространство имен:

```
using NorthwindService.Repositories;
```

Добавьте следующую инструкцию в нижнюю часть метода `ConfigureServices`. Эта инструкция зарегистрирует репозиторий `CustomerRepository` для использования во время выполнения платформой ASP.NET Core, как показано в данном коде:

```
services.AddScoped<ICustomerRepository, CustomerRepository>();
```

Создание контроллера веб-API

Добавьте в каталог `Controllers` новый класс `CustomersController.cs`.

Мы могли бы удалить файл ValuesController.cs, однако иметь простой контроллер веб-API с минимальным количеством зависимостей для целей тестирований — это всегда хорошо.

Добавьте следующий код в класс `CustomersController` и обратите внимание на такие аспекты.

- Класс контроллера регистрирует путь, начинающийся с `api` и включающий имя контроллера, например `api/customers`.
- Для получения зарегистрированного репозитория клиентов в конструкторе используется внедрение зависимостей.
- Реализовано пять методов выполнения операций `CRUD` над клиентами: два метода `GET` (все клиенты или один клиент), `POST` (создать), `PUT` (обновить) и `DELETE`.
- Методу `GetCustomers` можно передать строковый параметр с названием государства. При отсутствии этого параметра будут возвращены все клиенты, а при его наличии клиенты отфильтруются по стране.

- В методе `GetCustomer` путь явно назван `GetCustomer`, таким образом, этот путь может быть использован для генерации URL после ввода данных нового клиента:

```
using Microsoft.AspNetCore.Mvc;
using Packt.CS7;
using NorthwindService.Repositories;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace NorthwindService.Controllers
{
  // базовый адрес: api/customers
   [Route("api/[controller]")]
  public class CustomersController : Controller
  {
    private ICustomerRepository repo;

    // конструктор вводит зарегистрированный репозиторий
    public CustomersController(ICustomerRepository repo)
    {
      this.repo = repo;
    }

    // GET: api/customers
    // GET: api/customers/?country=[country]
    [HttpGet]
    public async Task<IEnumerable<Customer>> GetCustomers(string country)
    {
      if (string.IsNullOrWhiteSpace(country))
      {
        return await repo.RetrieveAllAsync();
      }
      else
      {
        return (await repo.RetrieveAllAsync())
        .Where(customer => customer.Country == country);
      }
    }

    // GET: api/customers/[id]
    [HttpGet("{id}", Name = "GetCustomer")]
    public async Task<IActionResult> GetCustomer(string id)
    {
      Customer c = await repo.RetrieveAsync(id);
      if (c == null)
      {
        return NotFound(); // 404 Resource not found
      }
      return new ObjectResult(c); // 200 OK
    }

    // POST: api/customers
    // BODY: Customer (JSON, XML)
    [HttpPost]
    public async Task<IActionResult> Create([FromBody] Customer c)
    {
```

```
        if (c == null)
        {
          return BadRequest(); // 400 Bad request
        }
        Customer added = await repo.CreateAsync(c);
        return CreatedAtRoute("GetCustomer", // use named route
        new { id = added.CustomerID.ToLower() }, c); // 201 Created
      }

      // PUT: api/customers/[id]
      // BODY: Customer (JSON, XML)
      [HttpPut("{id}")]
      public async Task<IActionResult> Update(string id, [FromBody]
      Customer c)
      {
        id = id.ToUpper();
        c.CustomerID = c.CustomerID.ToUpper();

        if (c == null || c.CustomerID != id)
        {
          return BadRequest(); // 400 Bad request
        }

        var existing = await repo.RetrieveAsync(id);
        if (existing == null)
        {
          return NotFound(); // 404 Resource not found
        }

        await repo.UpdateAsync(id, c);
        return new NoContentResult(); // 204 No content
      }

      // DELETE: api/customers/[id]
      [HttpDelete("{id}")]
      public async Task<IActionResult> Delete(string id)
      {
        var existing = await repo.RetrieveAsync(id);
        if (existing == null)
        {
          return NotFound(); // 404 Resource not found
        }

        bool deleted = await repo.DeleteAsync(id);

        if (deleted)
        {
          return new NoContentResult(); // 204 No content
        }
        else
        {
          return BadRequest();
        }
      }
    }
  }
```

Документирование и тестирование сервисов с применением Swagger

Веб-сервис легко протестировать, создав GET-запросы через браузер.

Тестирование запросов GET в любом браузере

Запустите веб-сервис.

В адресной строке браузера Chrome введите следующий URL: http://localhost:5000/api/customers.

На экране вы должны увидеть возвращенный документ JSON, содержащий 91 клиента из БД `Northwind` (рис. 16.4).

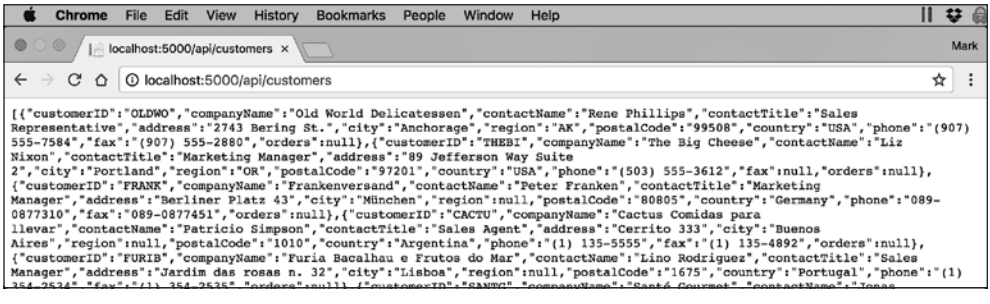


Рис. 16.4

В адресной строке браузера введите такой URL: http://localhost:5000/api/customers/alfki.

На экране вы должны увидеть возвращенный документ JSON, содержащий информацию только об одном клиенте — Alfreds Futterkiste (рис. 16.5).

Рис. 16.5

В адресной строке браузера введите следующий URL: http://localhost:5000/api/customers/?country=Germany.

На экране вы должны увидеть возвращенный документ JSON, содержащий информацию только из Германии (рис. 16.6).

Рис. 16.6

Но каким образом можно протестировать остальные методы, например `POST`, `PUT` и `DELETE`? И как задокументировать сервис, чтобы любой человек мог легко понять, как с ним взаимодействовать?

Тестирование запросов POST, PUT и DELETE с помощью Swagger

Swagger — самая популярная в мире технология для документирования и тестирования HTTP API.

Самая важная часть Swagger — стандарт *OpenAPI Specification*, определяющий соглашения, касающиеся ваших API в стиле REST, что означает *указание детальной информации об используемых ресурсах и операциях в форматах, удобочитаемых для человека и машины, с целью облегчить разработку, обнаружение возможностей и интеграцию*.

Есть еще одна полезная функция: интерфейс Swagger UI, автоматически генерирующий документацию для API и обладающий встроенными визуальными возможностями тестирования.

Дополнительную информацию о Swagger можно получить на сайте https://swagger.io.

Установка пакета Swagger

Чтобы использовать Swagger, нужно установить соответствующий пакет. Наиболее популярная реализация Swagger для ASP.NET Core называется Swashbuckle.

Visual Studio 2017

На панели Solution Explorer (Обозреватель решений) в проекте NorthwindService щелкните правой кнопкой мыши на пункте Dependencies (Зависимости) и выберите пункт Manage NuGet Packages (Управление пакетами NuGet).

На открывшейся панели перейдите на вкладку Browse (Обзор), выполните поиск пакета `Swashbuckle.AspNetCore` в строке поиска, выберите его и нажмите кнопку Install (Установить) (рис. 16.7).

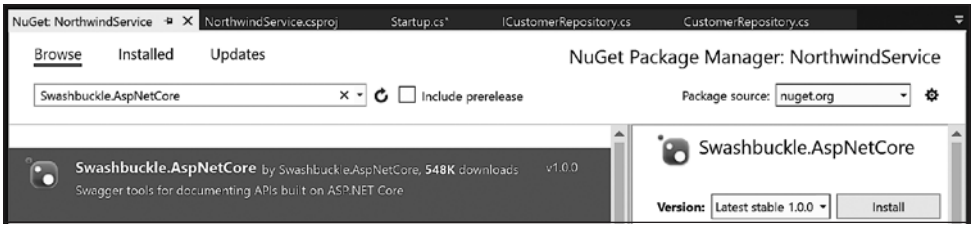


Рис. 16.7

Visual Studio Code

Откройте для редактирования файл `NorthwindService.csproj` и добавьте ссылку на пакет `Swashbuckle.AspNetCore`, как показано в следующем листинге (выделено полужирным):

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  <ProjectReference Include="..\NorthwindContextLib\NorthwindContextLib.csproj" />
  <PackageReference Include="Swashbuckle.AspNetCore" Version="1.0.0" />
</ItemGroup>
```

Visual Studio 2017 и Visual Studio Code

Откройте файл `Startup.cs` и импортируйте пространство имен `Swashbuckle Swagger`, как показано далее:

```
using Swashbuckle.AspNetCore.Swagger;
```

В методе `ConfigureServices` укажите инструкцию, добавляющую поддержку пакета Swagger с документацией для сервисов Northwind, как показано в листинге ниже:

```
// Зарегистрировать генератор Swagger и определить документ Swagger
// для сервиса Northwind
services.AddSwaggerGen(c =>
{
  c.SwaggerDoc("v1", new Info { Title = "Northwind Service API",
  Version = "v1" });
});
```

В методе `Configure` укажите инструкцию для использования Swagger и Swagger UI, а также для определения конечной точки документа JSON спецификации OpenAPI, как показано в следующем листинге:

```
app.UseSwagger();

app.UseSwaggerUI(c =>
{
  c.SwaggerEndpoint("/swagger/v1/swagger.json",
  "Northwind Service API V1");
});
```

Тестирование запросов GET с помощью Swagger UI

Запустите сайт и перейдите по адресу `/swagger/;`. Обратите внимание, что система автоматически обнаружила контроллеры `Customers` и `Values` и задокументировала их (рис. 16.8).

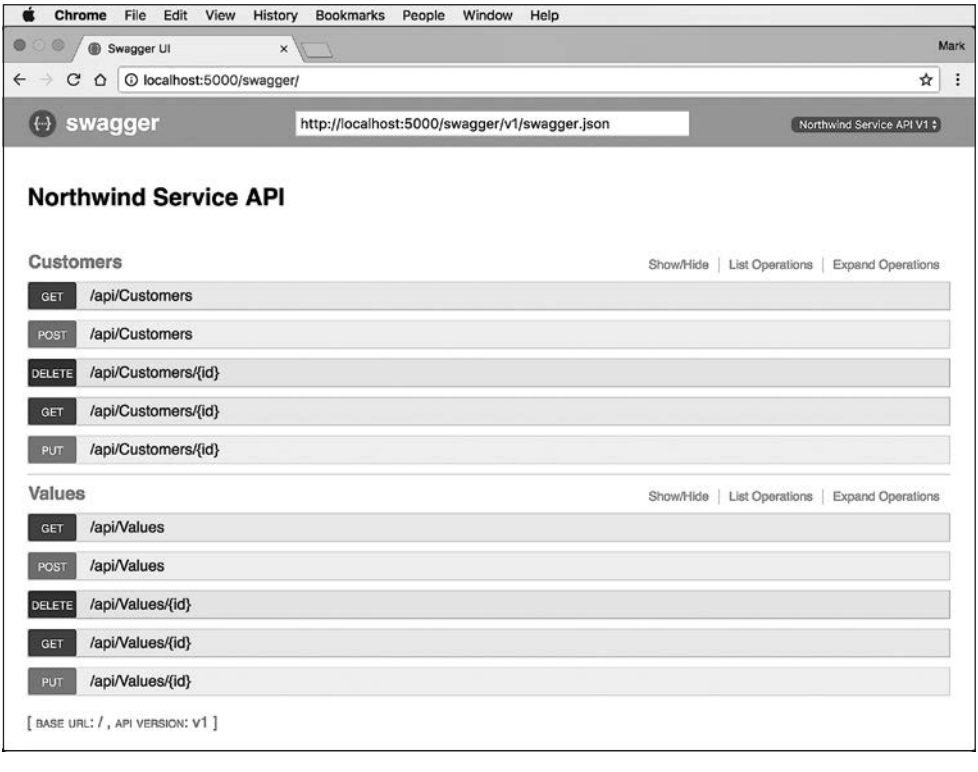


Рис. 16.8

Щелкните на методе GET /api/Customers/{id} и обратите внимание на обязательный параметр id для клиента (рис. 16.9).

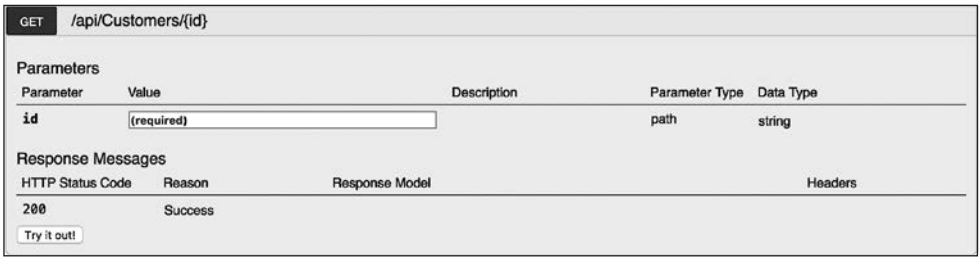


Рис. 16.9

Введите идентификатор `ALFKI`, нажмите кнопку Try it out! (Попробовать) и обратите внимание на значения полей Request URL (URL запроса), Response Body (Тело ответа), Response Code (Код ответа) и Response Headers (Заголовки ответа) (рис. 16.10).

Рис. 16.10

Тестирование запросов POST с помощью Swagger UI

Щелкните на методе POST /api/Customers.

Щелкните на теле элемента Example Value, чтобы скопировать его в поле для ввода значения параметра c, и измените разметку JSON для определения нового клиента, как показано в листинге ниже и на рис. 16.11.

```
{
  "customerID": "SUPER",
  "companyName": "Super Company",
  "contactName": "Rasmus Ibensen",
  "contactTitle": "Sales Leader",
  "address": "Rotterslef 23",
  "city": "Billund",
  "region": null,
  "postalCode": "4371",
  "country": "Denmark",
  "phone": "31 21 43 21",
  "fax": "31 21 43 22",
  "orders": null
}
```

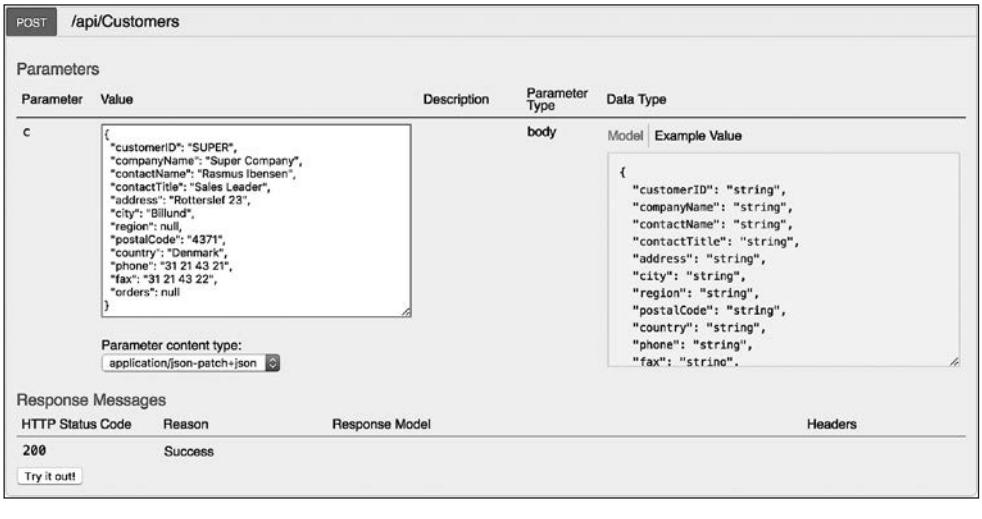


Рис. 16.11

Нажмите кнопку Try it out! (Попробовать) и обратите внимание на значения полей Request URL (URL запроса), Response Body (Тело ответа), Response Code (Код ответа) и Response Headers (Заголовки ответа) (рис. 16.12).

Код ответа 201 обозначает успешное создание клиента.

Рис. 16.12

Щелкните на методе GET /api/Customers, введите в поле параметра country (страна) значение Denmark и нажмите кнопку Try it out! (Попробовать), чтобы убедиться в добавлении нового клиента в базу данных (рис. 16.13).

Щелкните на методе DELETE /api/Customers/{id}, введите значение super для параметра id, нажмите кнопку Try it out! (Попробовать) и обратите внимание на Response Code (Код ответа) 204, обозначающий успешное удаление (рис. 16.14).

Нажмите кнопку Try it out! (Попробовать) еще раз и обратите внимание на Response Code (Код ответа) 404, обозначающий, что запрашиваемый клиент отсутствует в базе данных.

Воспользуйтесь методами GET, чтобы убедиться в удалении нового клиента из базы данных.

Тестирование обновлений с помощью метода PUT мы оставим читателям.

Закройте Chrome.

В Visual Studio Code на панели Integrated Terminal (Интегрированный терминал) нажмите сочетание клавиш Ctrl+C, чтобы закрыть консольное приложение и отключить сервер Kestrel, на котором расположен ваш сервис ASP.NET Core.

Теперь вы готовы приступить к созданию веб-приложения, вызывающего веб-сервис.

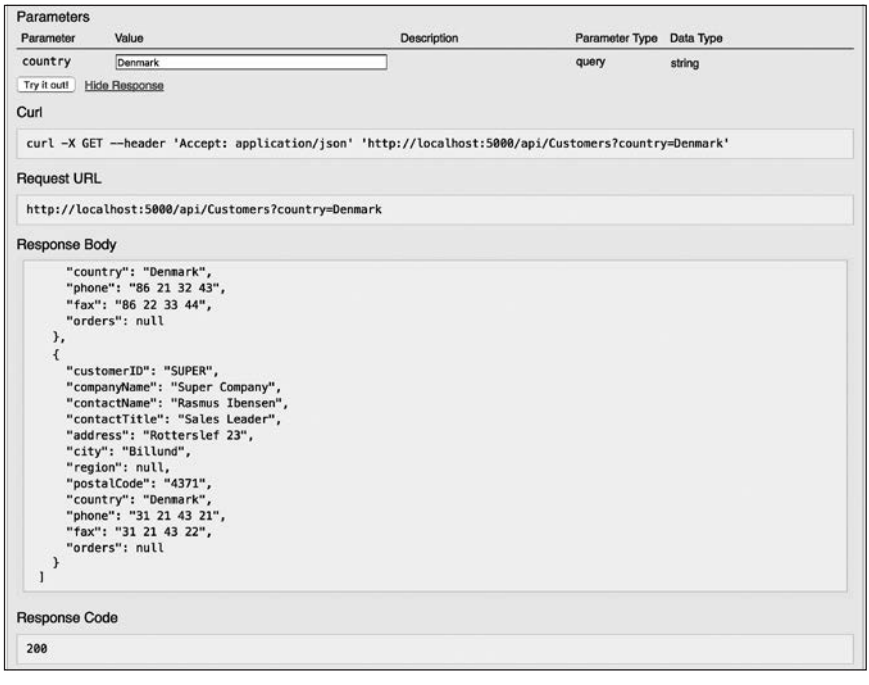


Рис. 16.13

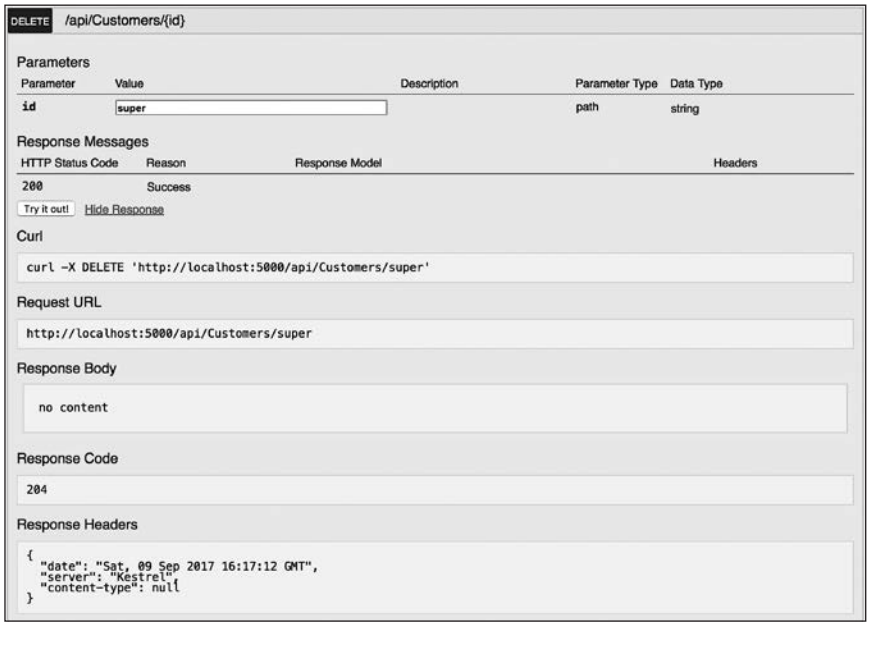


Рис. 16.14

Создание SPA с помощью Angular

Angular — популярный каркас разработки пользовательского интерфейса для мобильных и веб-приложений. В нем применяется TypeScript — созданный корпорацией Microsoft язык со строгим контролем типов, компилирующийся в JavaScript.

Шаблон проекта Angular

На платформе ASP.NET Core реализован специальный шаблон проекта для Angular. Рассмотрим его содержимое.

Visual Studio 2017

В Visual Studio 2017 откройте решение `Part3` и нажмите сочетание клавиш Ctrl+Shift+N или выполните команду меню File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне Add New Project (Добавить новый проект) в списке Installed ▸ Templates (Установленные ▸ Шаблоны) раскройте раздел Visual C# и выберите пункт .NET Core. В центре диалогового окна выберите пункт ASP.NET Core Web Application (Веб-приложение ASP.NET Core), введите имя `ExploreAngular` и нажмите кнопку OK.

В диалоговом окне New ASP.NET Core Web Application — ExploreAngular (Создать веб-приложения ASP.NET Core — ExploreAngular) выберите пункты раскрывающихся меню .NET Core и ASP.NET Core 2.0, после чего нажмите кнопку OK.

Visual Studio Code

В каталоге `Part3` создайте папку `ExploreAngular` и откройте ее в Visual Studio Code.

На панели Integrated Terminal (Интегрированный терминал) введите следующую команду для создания нового проекта веб-API ASP.NET Core, а затем воспользуйтесь менеджером пакетов Node Package Manager для установки зависимых пакетов:

```
dotnet new angular
npm install
```

При первой сборке менеджеры пакетов NuGet и NPM произведут загрузку всех зависимых пакетов, это может занять некоторое время. К сожалению, сообщений о происходящем с программой не будет, вследствие чего может даже показаться, что она не работает!

Visual Studio 2017 и Visual Studio Code

Запустите сайт (рис. 16.15).

Щелкните на команде Fetch data (Получить данные) (рис. 16.16).

Закройте Chrome, и посмотрим, как это работает.

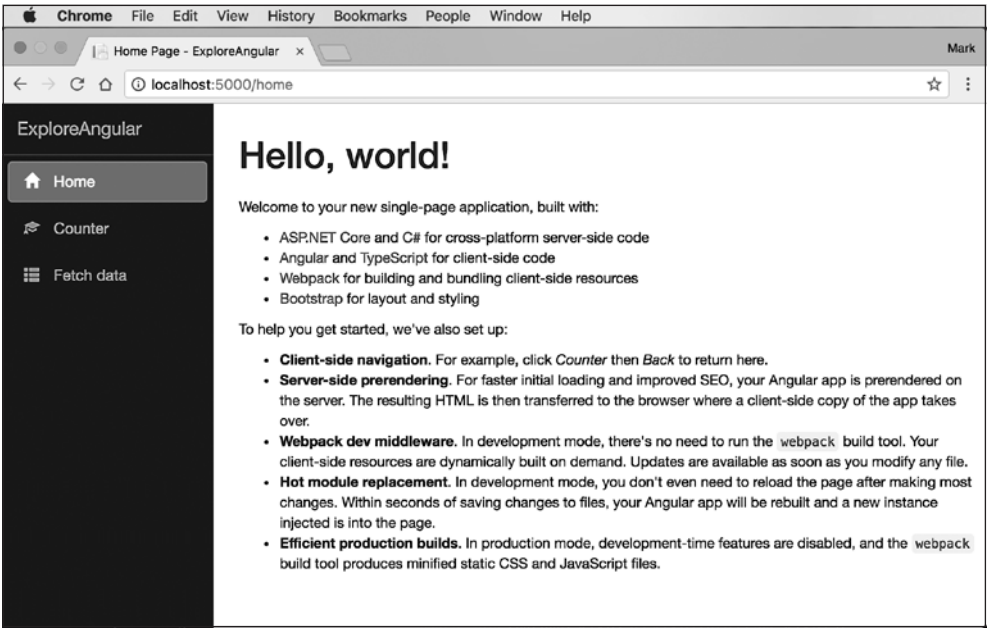


Рис. 16.15

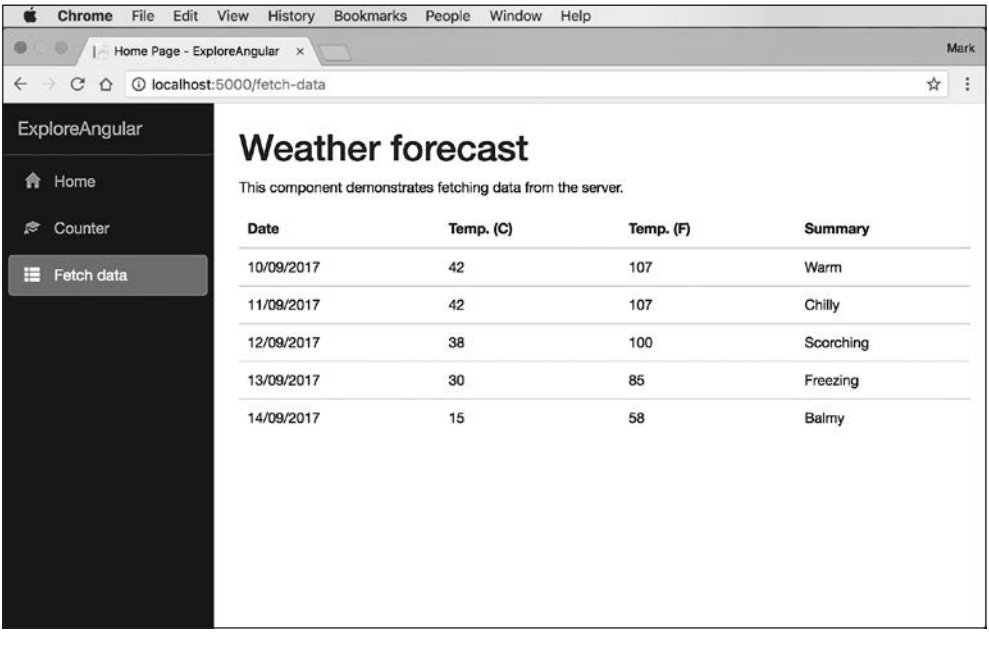


Рис. 16.16

Откройте файл `Startup.cs` и обратите внимание на следующий аспект: он практически такой же, как и в случае с шаблоном проекта MVC, за исключением того, что в нем реализована функциональность, описанная ниже.

- Для поддержки горячей замены модулей применяется промежуточное ПО Webpack. Это значит, что если разработчик вносит изменения в некий модуль Angular в тот момент, когда сайт запущен у пользователя, то данное изменение будет сразу же передано и задействовано на стороне клиента.
- Добавлен резервный путь на случай, если SPA не сможет отобразить представление `Index` контроллера `Home`.

В каталоге `Controllers` откройте файл `HomeController.cs` и обратите внимание на то, что метод `Index` не выполняет никаких особенных действий.

В каталоге `Views`, расположенном в папке `Home`, откройте файл `Index.cshtml`. Обратите внимание: элемент `<app>` ссылается на каталог `ClientApp/dist/main-server`, содержащий код стороны сервера, а элемент `<script>` ссылается на файл `main-client.js`, расположенный в папке `dist`, как показано в следующем листинге:

```
@{
  ViewData["Title"] = "Home Page";
}

<app asp-prerender-module="ClientApp/dist/main-server">Loading...</app>

<script src="~/dist/vendor.js" asp-append-version="true"></script>
@section scripts {
  <script src="~/dist/main-client.js"
  asp-append-version="true"></script>
}
```

Код на языке JavaScript серверной и клиентской сторон генерируется программой Webpack из кода на языке TypeScript, который вы пишете в папке `ClientApp`.

В каталоге `Controllers` откройте файл `SampleDataController.cs` и обратите внимание на то, что это сервис веб-API с единственным методом `GET`, возвращающим случайный прогноз погоды с помощью класса модели `WeatherForecast`.

В каталоге `ClientApp` обратите внимание на файл `boot.browser.ts`, содержащий инструкции на языке TypeScript, которые импортирует модуль AppModule из `app/app.module.browser`, как показано в коде ниже:

```
import { AppModule } from './app/app.module.browser';
```

В каталоге `ClientApp` раскройте папку `app`, откройте файл `app.module.browser.ts` и обратите внимание на то, что в нем содержится инструкция импорта модуля `AppModuleShared`, как показано в следующем коде:

```
import { AppModuleShared } from './app.module.shared';
```

Откройте файл `app.module.shared.ts` и обратите внимание на то, что в нем выполняется импорт пяти компонентов, как показано в листинге ниже:

```
import { AppComponent } from './components/app/app.component';
import { NavMenuComponent } from './components/navmenu/navmenu.component';
import { HomeComponent } from './components/home/home.component';
import { FetchDataComponent } from './components/fetchdata/fetchdata.component';
import { CounterComponent } from './components/counter/counter.component';
```

В каталоге `ClientApp` раскройте подкаталоги `app` и `components`, чтобы увидеть пять компонентов Angular для данного одностраничного приложения (рис. 16.17), и обратите внимание на следующие аспекты:

- `app` — основной компонент приложения, у него есть меню слева и маршрутизатор стороны клиента справа;
- `counter` — это компонент с простой кнопкой, совершающей операцию инкремента;
- `fetchdata` — компонент, запрашивающий данные о погоде у сервера;
- `home` — компонент по умолчанию, отображающий некий статический контент HTML;
- `navmenu` — компонент, делающий возможной маршрутизацию между другими компонентами на стороне клиента.

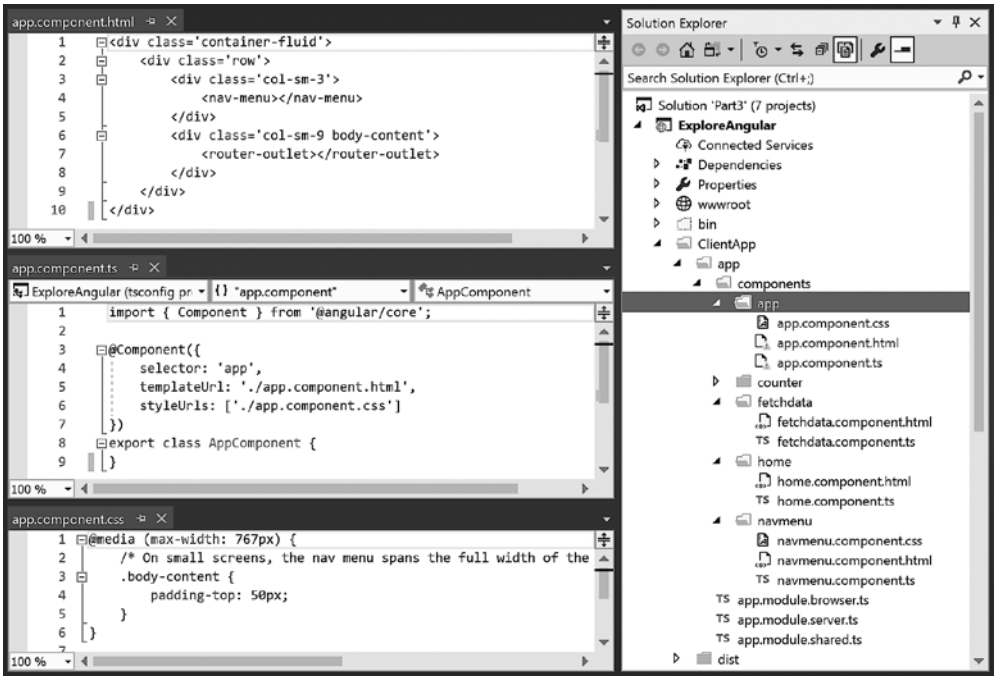


Рис. 16.17

Откройте файлы `home.component.ts` и `fetchdata.component.ts`, сравните их, как показано в листинге ниже, и обратите внимание вот на что:

- в компоненте `home` нет программного кода, только статический контент HTML;
- компонент `fetchdata` импортирует пространство имен `Http`, использует его в конструкторе, чтобы совершить HTTP-запрос `GET` к службе погоды, сохраняет ответ в публичном массиве объектов `WeatherForecast`, определенном в интерфейсе.

```
// home.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'home',
  templateUrl: './home.component.html'
})
export class HomeComponent {
}

// fetchdata.component.ts
import { Component, Inject } from '@angular/core';
import { Http } from '@angular/http';

@Component({
  selector: 'fetchdata',
  templateUrl: './fetchdata.component.html'
})
export class FetchDataComponent {
public forecasts: WeatherForecast[];

constructor(http: Http, @Inject('BASE_URL') baseUrl: string) {
  http.get(baseUrl + 'api/SampleData/WeatherForecasts')
  .subscribe(result => {
    this.forecasts = result.json() as WeatherForecast[];
    }, error => console.error(error));
  }
}

interface WeatherForecast {
  dateFormatted: string;
  temperatureC: number;
  temperatureF: number;
  summary: string;
}
```

Вызов NorthwindService

Теперь, когда вы получили некоторое представление о том, каким образом сочетаются компоненты технологии Angular и могут получать данные от сервисов, мы изменим компонент `home` для загрузки списка клиентов из `NorthwindService`.

Но прежде будет полезно явно указать номера портов для сайтов NorthwindService и ExploreAngular, а также включить возможность *обмена ресурсами с запросом происхождения* (cross-origin resource sharing, CORS), чтобы сайт ExploreAngular мог вызывать сервис `NorthwindService`.

Политика единого происхождения, используемая в браузерах по умолчанию, в целях повышения безопасности не позволяет коду, загруженному из одного источника, получать доступ к ресурсам, загруженным из других источников. Политика CORS подходит для разрешения запросов от указанных доменов. Изучите более подробную информацию о CORS и ASP.NET Core, перейдя на сайт https://docs.microsoft.com/en-us/aspnet/core/security/cors.

Visual Studio 2017

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте NorthwindService и выберите пункт Properties (Свойства).

Перейдите на вкладку Debug (Отладка) и в категории Web Server Settings (Параметры веб-сервера) измените номер порта в поле App URL (URL приложения) на `5001`, как показано ниже:

```
http://localhost:5001/
```

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте ExploreAngular и выберите пункт Properties (Свойства).

Перейдите на вкладку Debug (Отладка) и в категории Web Server Settings (Параметры веб-сервера) измените номер порта в поле App URL (URL приложения) на `5002`.

Visual Studio Code

В проекте NorthwindService откройте файл `Program.cs` и добавьте в метод `BuildWebHost` вызов метода расширения `UseUrls`, чтобы указать номер порта `5001`, как показано полужирным шрифтом в листинге ниже:

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
  .UseStartup<Startup>()
  .UseUrls("http://localhost:5001")
  .Build();
```

В проекте ExploreAngular откройте файл `Program.cs` и добавьте в метод `BuildWebHost` вызов метода расширения `UseUrls`, чтобы указать номер порта `5002`, как показано полужирным шрифтом в следующем листинге:

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
  .UseStartup<Startup>()
  .UseUrls("http://localhost:5002")
  .Build();
```

Visual Studio 2017 и Visual Studio Code

В проекте NorthwindService откройте файл `Startup.cs` и добавьте инструкцию в начало метода `ConfigureServices`, чтобы включить поддержку CORS, как показано полужирным шрифтом в листинге ниже:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddCors();
```

Добавьте инструкцию в метод `Configure` перед вызовом `UseMvc`, чтобы задействовать CORS и разрешить получение запросов с сайта ExploreAngular, как показано полужирным шрифтом в следующем листинге:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }

  app.UseCors(c => c.WithOrigins("http://localhost:5002"));

  app.UseMvc();
```

CORS необходимо использовать перед MVC, иначе ничего не сработает!

Изменение компонента home для вызова NorthwindService

Откройте файл `home.component.ts` и измените его, как показано в листинге ниже:

```
import { Component, Inject } from '@angular/core';
import { Http } from '@angular/http';

@Component({
  selector: 'home',
  templateUrl: './home.component.html'
})
export class HomeComponent {
  public customers: Customer[];

constructor(http: Http, @Inject('BASE_URL') baseUrl: string) {
  http.get('http://localhost:5001/api/customers').subscribe(result => {
    this.customers = result.json() as Customer[];
    }, error => console.error(error));
  }
}

interface Customer {
  customerID: string;
```

```
  companyName: string;
  contactName: string;
  contactTitle: string;
  address: string;
  city: string;
  region: string;
  postalCode: string;
  country: string;
  phone: string;
  fax: string;
}
```

Откройте файл `home.component.html` и измените его, как показано в следующем листинге:

```
<h1>Customers</h1>
<p>These customers have been loaded from the NorthwindService.</p>
<p *ngIf="!customers"><em>Loading customers... please wait.</em></p>
<table class='table' *ngIf="customers">
  <thead>
    <tr>
      <th>ID</th>
      <th>Company Name</th>
      <th>Contact Name</th>
      <th>City</th>
      <th>Country</th>
    </tr>
  </thead>
  <tbody>
  <tr *ngFor="let customer of customers">
    <td>{{ customer.customerID }}</td>
    <td>{{ customer.companyName }}</td>
    <td>{{ customer.contactName }}</td>
    <td>{{ customer.city }}</td>
    <td>{{ customer.country }}</td>
  </tr>
</tbody>
</table>
```

Тестирование вызова сервиса компонентом Angular

Чтобы проверить внесенные изменения, нужно сначала запустить сайт NorthwindService, а затем ExploreAngular.

Visual Studio 2017

На панели Solution Explorer (Обозреватель решений) выберите проект NorthwindService и выполните команду Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5.

Запустится браузер Chrome и отобразит документ JSON со списком всех клиентов из БД `Northwind`.

Не закрывайте Chrome.

На панели Solution Explorer (Обозреватель решений) выберите проект ExploreAngular и выполните команду Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5.

Visual Studio Code

В Visual Studio Code откройте каталог `NorthwindService` и на панели Integrated Terminal (Интегрированный терминал) введите следующую команду, позволяющую задать среду внешнего размещения и запуска сайта:

```
ASPNETCORE_ENVIRONMENT=Development dotnet run
```

В Visual Studio Code выполните команду File ▸ New Window (Файл ▸ Новое окно), откройте каталог `ExploreAngular` и на панели Integrated Terminal (Интегрированный терминал) введите следующую команду, чтобы задать среду внешнего размещения и запуска сайта:

```
ASPNETCORE_ENVIRONMENT=Development dotnet run
```

Запустите браузер Chrome и введите адрес: http://localhost:5002.

Visual Studio 2017 и Visual Studio Code

Компонент Angular сначала выполнится на сервере и отправит предварительно подготовленную HTML-страницу со списком всех клиентов, после этого произойдет синхронный вызов сервиса `NorthwindService`. На короткое время ожидания будет отображено сообщение Loading (Загрузка), а затем таблица обновится согласно ответу JSON (рис. 16.18).

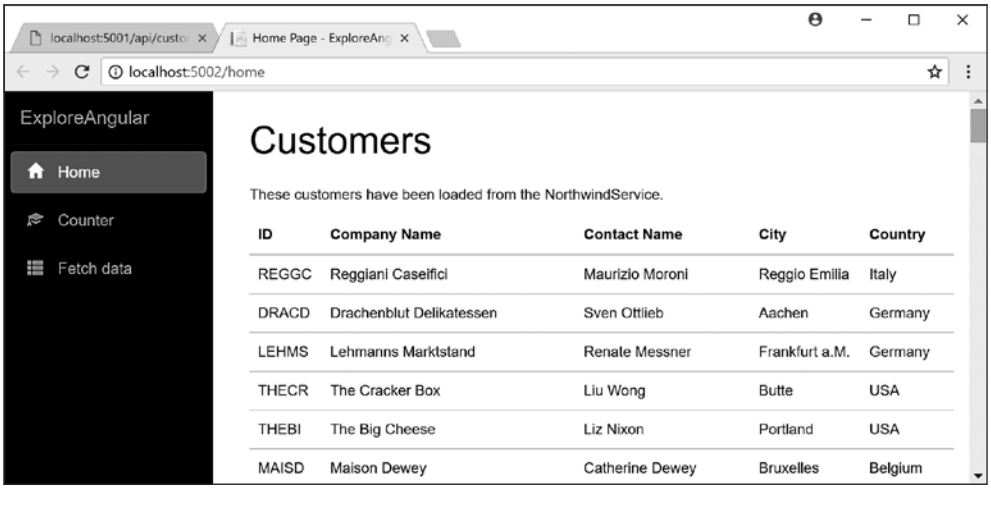


Рис. 16.18

Использование других шаблонов проектов

Установка .NET Core SDK 2.0 включает большое количество других шаблонов.

В окне интерпретатора командной строки или терминала введите следующую команду:

```
dotnet new --help
```

В результате вы увидите список установленных в настоящее время шаблонов (рис. 16.19).

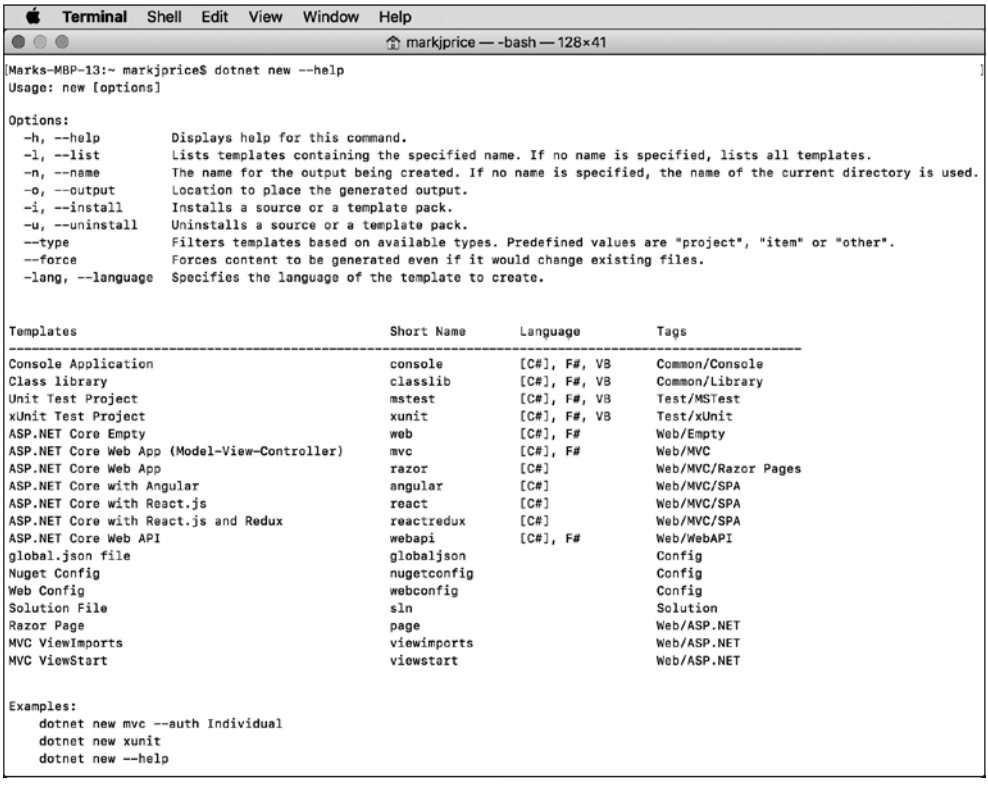


Рис. 16.19

Установка дополнительных пакетов шаблонов. Запустите браузер и перейдите на сайт http://dotnetnew.azurewebsites.net, чтобы просмотреть перечень доступных шаблонов (рис. 16.20).

Щелкните на ссылке ASP.NET Core with Aurelia и обратите внимание на инструкции по установке и использованию этого шаблона (рис. 16.21).

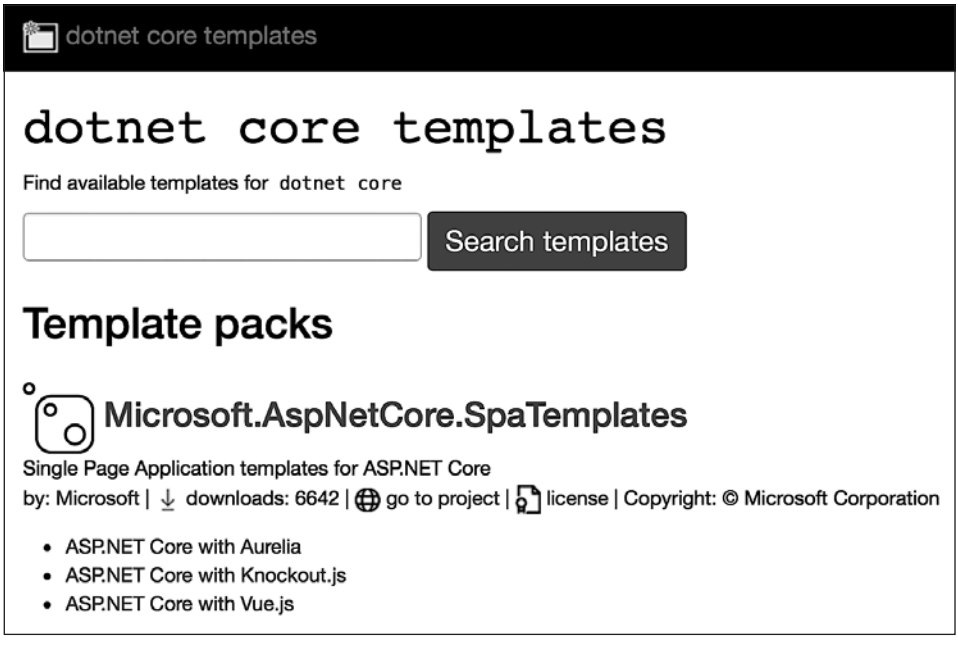


Рис. 16.20

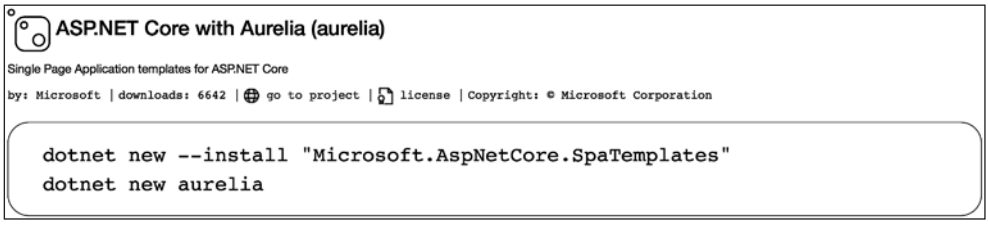


Рис. 16.21

Практические задания

Попрактикуйтесь и более детально исследуйте темы этой главы.

Упражнение 16.1. React и Redux

Используя `React.js` и шаблон проекта `Redux`, создайте новый проект, а затем попробуйте изменить его код, чтобы из него осуществлялся вызов сервиса `NorthwindService`, как мы делали для Angular:

```
dotnet new reactredux
```

Дополнительные ресурсы

- Создание веб-API: https://docs.microsoft.com/en-us/aspnet/core/mvc/web-api/.
- Инструменты Swagger: https://swagger.io/tools/.
- Swashbuckle для ASP.NET Core: https://github.com/domaindrivendev/Swashbuckle.AspNetCore.
- Angular: https://angular.io.
- TypeScript: https://www.typescriptlang.org.
- React: https://facebook.github.io/react/.
- Redux: https://redux.js.org.

Резюме

Из этой главы вы узнали, как создать сервис веб-API ASP.NET Core, который можно разместить кросс-платформенно. Кроме того, вы узнали о том, как протестировать и задокументировать API веб-сервисов, используя Swagger, а также о способах создания одностраничных приложений с помощью технологии Angular, вызывающей веб-сервис, даже если он размещен на другом домене.

В следующей главе рассказывается, как создавать кросс-платформенные приложения для Universal Windows Platform, применяя XAML.

17 Разработка Windows-приложений с помощью языка XAML и системы проектирования Fluent

В данной главе я расскажу о том, чего можно достичь, прибегнув к XAML при разработке графических пользовательских интерфейсов приложений, предназначенных, в частности, для универсальной платформы Windows (UWP). Вы познакомитесь с некоторыми новыми функциями UI, такими как система проектирования Fluent, доступная в версии Windows 10 Fall Creators Update.

В этой небольшой главе возможности языка XAML и платформы UWP будут затронуты лишь поверхностно. Однако я надеюсь, что у вас возникнет интерес к этой классной технологии и платформе и желание узнать о них больше.

Считайте главу 17 небольшой агитационной кампанией, рекламирующей самые лакомые возможности UWP, XAML и системы проектирования Fluent, в том числе *шаблоны элементов управления*, привязку данных и анимацию!

НЕКОТОРЫЕ ВАЖНЫЕ МОМЕНТЫ ОБ ЭТОЙ ГЛАВЕ

UWP-приложения хотя и не являются кросс-платформенными, но задействуют принцип «кросс-платформенной» разработки для устройств, которые работают под управлением современных версий операционной системы Windows. Для выполнения упражнений из данной главы понадобятся операционная система Windows 10 с установленным обновлением Fall Creators Update и среда разработки Visual Studio 2017 версии не ниже 15.4. Универсальная платформа Windows поддерживает .NET Native, а это значит, что код компилируется в нативные процессорные инструкции для уменьшения затрат памяти и более быстрого выполнения.

В данной главе:

- обзор современной платформы Windows;
- разработка современных Windows-приложений;
- использование ресурсов и шаблонов;
- привязка данных;
- создание приложений с помощью Windows Template Studio.

Общие сведения о современной платформе Windows

Корпорация Microsoft продолжает совершенствовать свою платформу Windows, которая включает в себя множество технологий для разработки современных приложений:

- универсальную платформу Windows 6.0;
- систему проектирования Fluent;
- XAML Standard 1.0.

Общие сведения об универсальной платформе Windows

Универсальная платформа Windows (UWP) — это новейшее решение Microsoft для разработки приложений, предназначенных к выполнению в операционных системах семейства Windows.

UWP обеспечивает гарантированный API для различных типов устройств. Вы создаете один пакет приложения; он загружается в один магазин и распространяется на все типы устройств, для которых разработано ваше приложение. Эти устройства функционируют под управлением операционных систем Windows 10, Windows 10 Mobile, Xbox One и Microsoft HoloLens.

Обновление Windows 10 Fall Creators Update, выпущенное 17 октября 2017 года, включает платформу UWP 6.0, созданную на специально разработанной реализации .NET Core 2.

Язык XAML и универсальная платформа Windows предоставляют панели макетов, адаптирующие отображение дочерних элементов управления, чтобы можно было наиболее эффективно использовать возможности устройства, на котором выполняется приложение. Своего рода это технология адаптивного отображения веб-страниц, только в отношении Windows-приложений.

Язык XAML и UWP предоставляют триггеры визуального состояния для коррекции макета согласно динамическим изменениям, таким как смена ориентации планшета.

Универсальная платформа Windows содержит механизмы, позволяющие определить возможности используемого устройства, а затем задействовать дополнительные функции вашего приложения, чтобы применить его максимально эффективно.

Обзор системы проектирования Fluent

Система проектирования Fluent от корпорации Microsoft будет выпускаться в несколько этапов, а не сразу вся, что создало бы эффект «Большого взрыва». Таким образом разработчики смогут постепенно перейти от традиционных стилей пользовательского интерфейса к более современным.

Этап 1, доступный в Windows 10 Fall Creators Update, включает в себя следующие функции:

- акриловый материал;
- подключенные анимации;
- параллакс;
- эффект отображения.

Заливка элементов пользовательского интерфейса акриловым материалом

Акриловый материал — это полупрозрачная кисть с эффектом размытия, которую можно применять для заливки элементов UI, чтобы добавить глубину и перспективу в приложения. Акриловый материал позволяет видеть, что находится позади приложения, а также элементы, находящиеся позади открытой панели. Акриловый материал можно настроить в соответствии с вашими предпочтениями по цветам и степеням прозрачности.

Более подробную информацию о том, как и когда задействовать акриловый материал, можно получить, перейдя по ссылке https://docs.microsoft.com/en-gb/windows/uwp/style/acrylic.

Соединение элементов пользовательского интерфейса с помощью анимаций

При переходах по UI анимированные элементы, отрисовывающие связи между экранами, позволяют пользователям понять, где именно они находятся в данный момент и как взаимодействовать с приложением.

Более подробные сведения о том, как и когда использовать подключенные анимации, находятся на сайте https://docs.microsoft.com/en-gb/windows/uwp/style/connected-animation.

Параллакс и эффект отображения

Параллакс придает приложениям более современный облик, а эффект отображения позволяет понять, какой элемент является интерактивным, *подсвечивая* UI и тем самым привлекая внимание пользователя по мере его перехода между элементами.

О том, как и когда применять эффект отображения для привлечения внимания пользователей к элементам интерфейса, можно узнать, перейдя по ссылке https://docs.microsoft.com/en-gb/windows/uwp/style/reveal.

Набор стандартов XAML Standard 1.0

В 2006 году корпорация Microsoft выпустила систему Windows Presentation Foundation (WPF), ставшую пионером в области использования языка XAML. Она применяется и сегодня для проектирования настольных приложений (к примеру, среда разработки Microsoft Visual Studio 2017 представляет собой WPF-приложение).

Язык XAML можно задействовать для создания:

- *приложений для магазина Windows Store* для операционных систем Windows 10, Windows 10 Mobile, Xbox One и Microsoft HoloLens;
- *настольных WPF-приложений* для операционной системы Windows, включая Windows 7 и более поздние версии.

По аналогии с .NET существуют версии XAML с небольшими функциональными различиями для разных платформ. Как .NET Standard 2.0 есть воплощение идеи объединения различных платформ .NET, так и XAML Standard 1.0 — попытка сделать то же самое с XAML.

Упрощение кода с помощью языка XAML

Язык XAML упрощает код C#, особенно при работе над пользовательским интерфейсом.

Допустим, вам нужно разместить горизонтально две или даже больше кнопок, чтобы создать панель инструментов. На языке C# вы должны написать следующий код:

```
var toolbar = new StackPanel();
toolbar.Orientation = Orientation.Horizontal;
var newButton = new Button();
newButton.Content = "New";
newButton.Background = new SolidColorBrush(Colors.Pink);
toolbar.Children.Add(newButton);
var openButton = new Button();
openButton.Content = "Open";
openButton.Background = new SolidColorBrush(Colors.Pink);
toolbar.Children.Add(openButton);
```

На языке XAML код можно упростить. При обработке XAML-кода задаются эквивалентные свойства и вызываются аналогичные методы для достижения той же цели, которая подразумевается в ранее указанном коде C#:

```
<StackPanel Name="toolbar" Orientation="Horizontal">
  <Button Name="newButton" Background="Pink">New</Button>
  <Button Name="OpenButton" Background="Pink">Open</Button>
</StackPanel>
```

Язык XAML — альтернативный (лучший) способ объявления и создания экземпляров .NET-типов.

Использование общих элементов управления

Существует множество предопределенных элементов управления, которые можно применять для разработки стандартных UI. Практически все версии языка XAML поддерживают эти элементы управления (табл. 17.1).

Таблица 17.1

Элемент (-ы) управления	Описание
Button, Menu, Toolbar	Выполнение действий
CheckBox, RadioButton	Выбор вариантов
Calendar, DatePicker	Выбор дат
ComboBox, ListBox, ListView, TreeView	Выбор элементов из списков и иерархических деревьев
Canvas, DockPanel, Grid, StackPanel, WrapPanel	Контейнеры макета с различным влиянием на потомков
Label, TextBlock	Отображение текста только для чтения
RichTextBox, TextBox	Редактируемый текст
Image, MediaElement	Встраивание изображений, видео- и аудиофайлов
DataGrid	Просмотр и редактирование связанных данных
Scrollbar, Slider, StatusBar	Различные элементы пользовательского интерфейса

Разработка современных Windows-приложений

Мы начнем с создания простого приложения Windows с некоторыми стандартными элементами управления и современными функциями системы Fluent, такими как акриловый материал.

Прежде чем приступить к созданию приложений для универсальной платформы Windows, необходимо активизировать режим разработчика в операционной системе Windows 10.

Активизация режима разработчика

Откройте меню Start (Пуск), щелкните на значке шестеренки Settings (Параметры) и выберите пункт Update & Security (Обновление и безопасность). В разделе For developers (Для разработчиков) установите переключатель в положение Developer mode (Режим разработчика) (рис. 17.1).

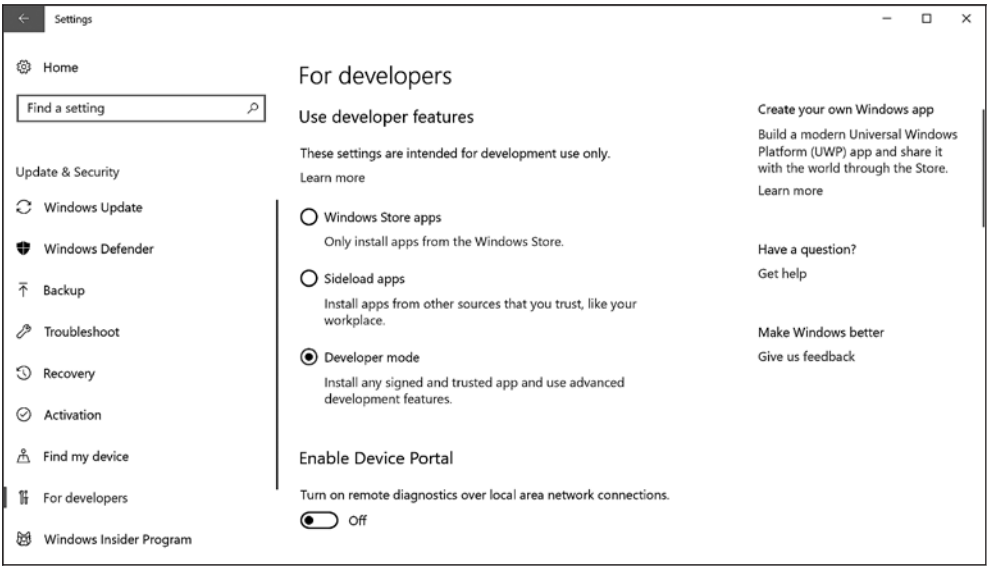


Рис. 17.1

Согласитесь с предупреждением о том, что режим «может подвергнуть риску безопасность устройства и личные данные или нанести вред устройству», а затем закройте окно Settings (Параметры). Возможно, вам придется перезагрузить компьютер.

Создание проекта UWP-приложения

В Visual Studio 2017 откройте решение `Part3` и выполните команду File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне Add New Project (Добавить новый проект) в списке Installed (Установленные) выполните команду Visual C# ▸ Windows Universal (Visual C# ▸ Универсальные приложения Windows). В центре диалогового окна выберите пункт Blank App (Universal Windows) (Пустое приложение (Универсальное приложение для Windows)), присвойте ему имя `FluentUwpApp`, а затем нажмите кнопку OK.

В диалоговом окне New Universal Windows Project (Новый универсальный проект для Windows) выберите последнюю версию Windows 10 в раскрывающихся списках

Target Version (Целевая версия) и Minimum Version (Минимальная версия) и нажмите кнопку OK (рис. 17.2).

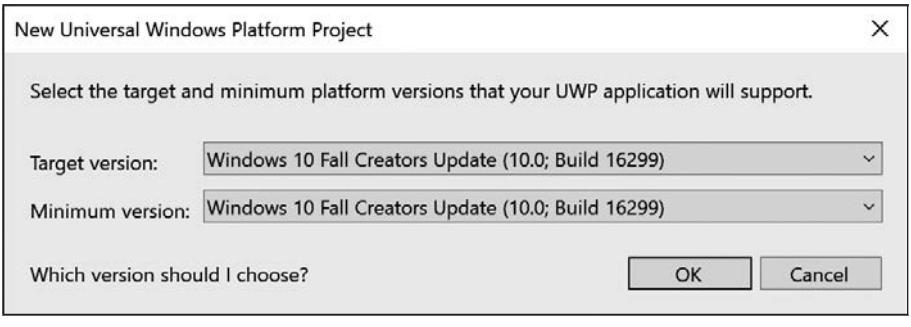


Рис. 17.2

Разработчики, создающие UWP-приложения для широкой аудитории, должны выбирать последнюю сборку Windows 10 в раскрывающемся списке Minimum Version (Минимальная версия). Разработчикам, создающим корпоративные приложения, следует выбрать наиболее старую минимальную версию. Сборка 10240 была выпущена в июле 2015 года и лучше всего подходит для максимальной совместимости, но не обеспечивает доступ к современным функциям, таким как система проектирования Fluent.

На панели Solution Explorer (Обозреватель решений) дважды щелкните на файле `MainPage.xaml`, чтобы открыть его для редактирования. Вы увидите панель Design (Конструктор), отображающую графический вид приложения, и панель XAML с содержимым файла `MainWindow.xaml`. Этими панелями можно управлять следующим образом:

- панели расположены по горизонтали, но вы можете переключиться на режим отображения по вертикали или свернуть одну из панелей, нажав соответствующую кнопку в правой части интерфейса между панелями;
- можно поменять панели местами, нажав кнопку ↑↓ между панелями;
- можно прокручивать и изменять масштаб обеих панелей (рис. 17.3).

Установите масштаб, равный 100 %. В меню View (Вид) выберите пункт Toolbox (Панель инструментов) или нажмите сочетание клавиш Ctrl+W, X. Обратите внимание: панель содержит категории Common XAML Controls (Типовые элементы управления XAML), All XAML Controls (Все элементы управления XAML) и General (Общие). В верхней части панели находится поле поиска.

Введите в поле поиска буквы bu. Обратите внимание, как отфильтруется список элементов управления.

Перетащите элемент управления Button с панели Toolbox (Панель инструментов) на панель Design (Конструктор). Измените размер элемента управления Button, нажав

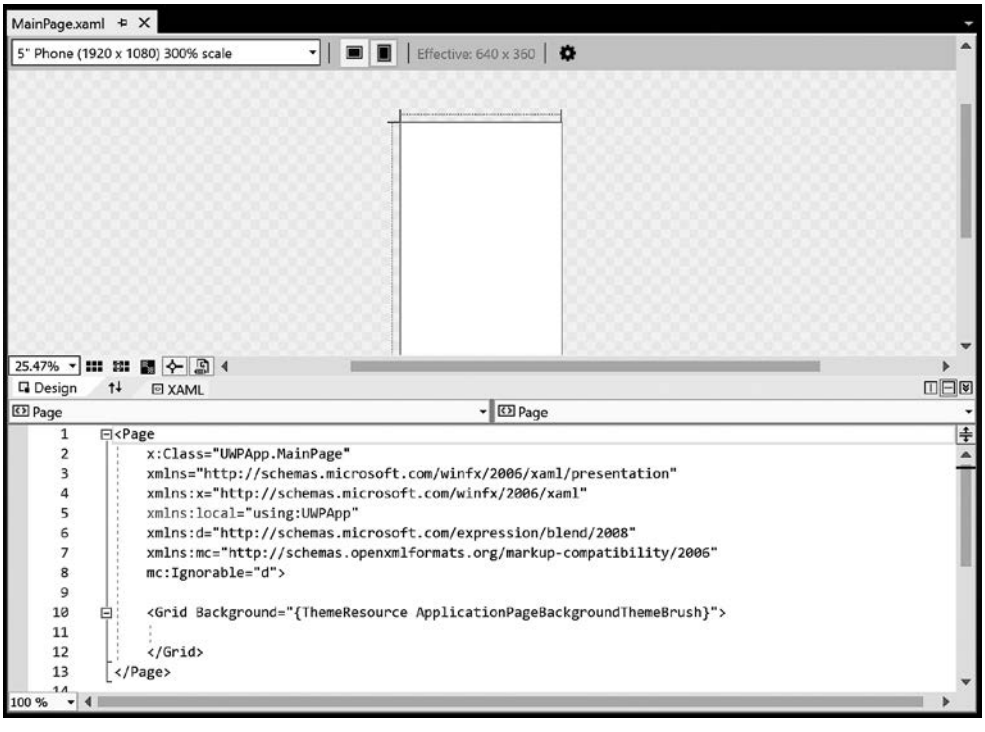


Рис. 17.3

и удерживая кнопку мыши на любом из восьми квадратных маркеров и перетаскивая мышь. Обратите внимание: для кнопки определены фиксированные ширина и высота, а также поля сверху и слева (40 и 30 единиц соответственно) для придания ей верного размера и положения на сетке (рис. 17.4).

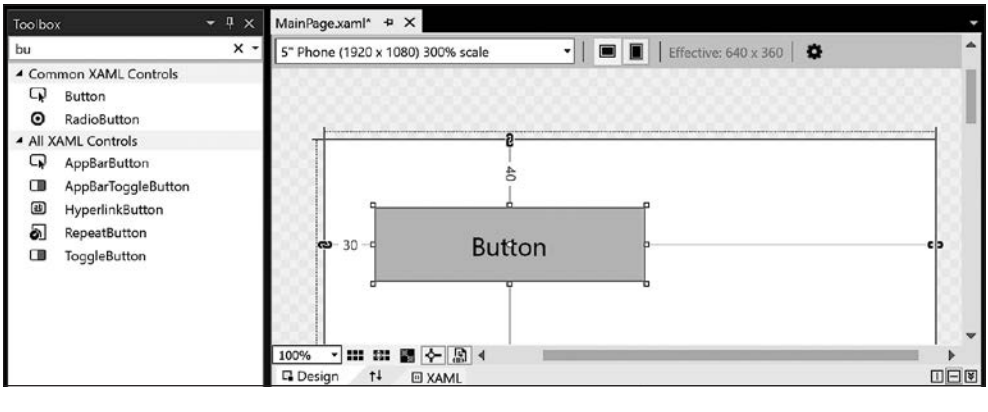


Рис. 17.4

Хотя вы можете перетаскивать элементы управления вручную, для компоновки макета лучше использовать панель XAML. Так удобнее позиционировать элементы относительно друг друга и реализовывать адаптивный дизайн.

На панели XAML найдите элемент `Button` и удалите его.

На панели XAML добавьте в элементе `Grid` следующую разметку:

```
<Button Margin="6" Padding="6" Name="clickMeButton">
  Click Me
</Button>
```

Измените масштаб, установив его равным 50 %, и обратите внимание, что кнопка автоматически меняет размер согласно своему содержимому, то есть тексту `Click Me`, выравнивается по вертикали по центру и по горизонтали влево (рис. 17.5).

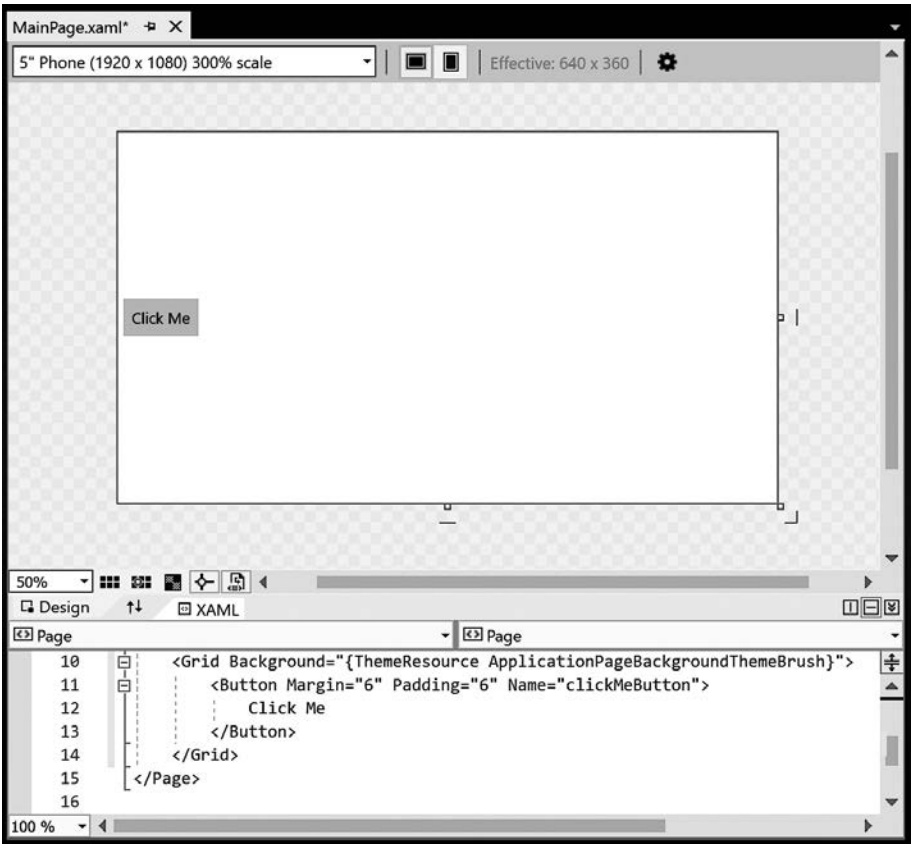


Рис. 17.5

Измените код на панели XAML, чтобы обернуть элемент `Button` элементом `StackPanel`, ориентированным горизонтально и имеющим светло-серую заливку,

который находится в элементе StackPanel, ориентированном вертикально (по умолчанию). Обратите внимание на изменение компоновки макета:

```
<StackPanel>
  <StackPanel Orientation="Horizontal" Padding="4" Background="LightGray"
Name="toolbar">
    <Button Margin="6" Padding="6" Name="clickMeButton">
      Click Me
    </Button>
  </StackPanel>
</StackPanel>
```

Измените код элемента Button, чтобы задать новый обработчик для события Click. Когда меню IntelliSense отобразит пункт <New Event Handler> (<Новый обработчик событий>) (рис. 17.6), нажмите клавишу Enter.

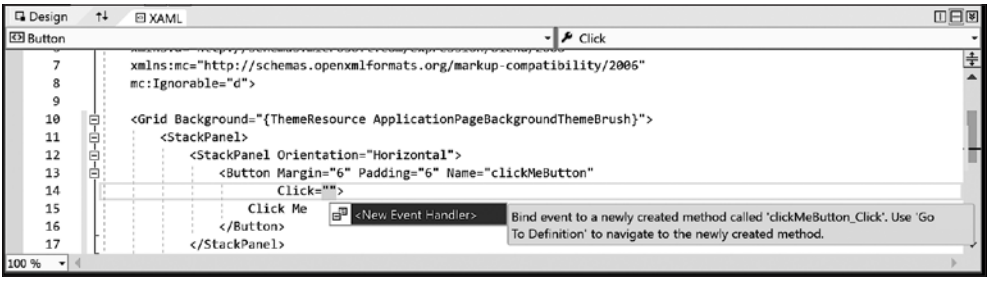


Рис. 17.6

Выполните команду View ▸ Code (Вид ▸ Код) или нажмите клавишу F7 для просмотра кода.

Добавьте в код файла MainPage.xaml.cs к обработчику событий следующую инструкцию, отображающую на кнопке текущее время, как показано в листинге ниже (выделено полужирным шрифтом):

```
private void clickMeButton_Click(object sender, RoutedEventArgs e)
{
  clickMeButton.Content = DateTime.Now.ToString("hh:mm:ss");
}
```

Выполните команду Build ▸ Configuration Manager (Сборка ▸ Диспетчер конфигураций), в строке проекта FluentUwpApp установите флажки Build (Сборка) и Deploy (Развертывание), выберите в раскрывающемся списке Platform (Платформа) пункт x64 и нажмите кнопку Close (Закрыть) (рис. 17.7).

Запустите приложение, выбрав в меню Debug (Отладка) пункт Start Without Debugging (Запуск без отладки) или нажав сочетание клавиш Ctrl+F5.

Нажмите кнопку Click Me (Нажми меня).

Каждый раз, когда вы нажимаете кнопку, ее текст меняется, показывая текущее время.

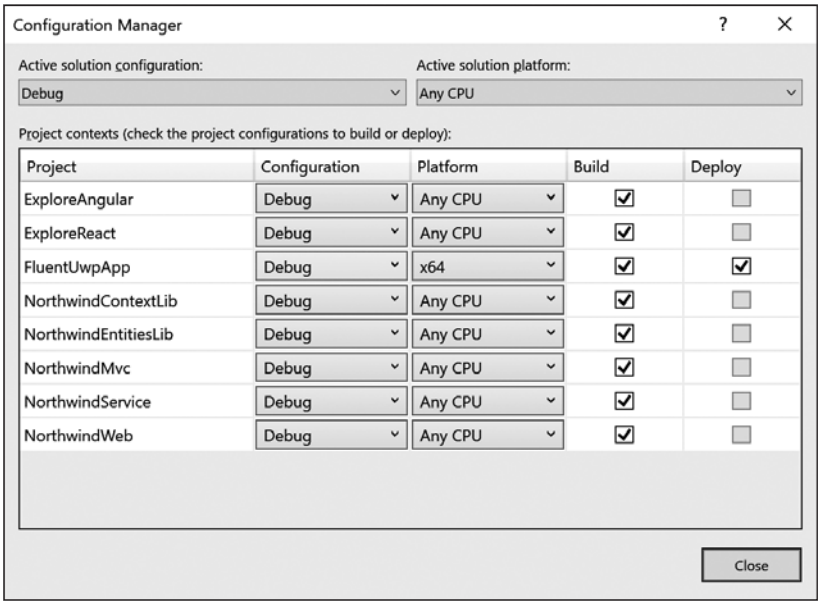


Рис. 17.7

Обзор основных элементов управления и акрилового материала

Откройте файл `MainPage.xaml`, настройте фон сетки на применение акрилового материала для окон и добавьте несколько элементов на панель стека сразу после кнопки, чтобы пользователь мог ввести свое имя и включить акриловый материал, как показано в следующем листинге:

```
<Grid Background="{ThemeResource SystemControlAcrylicWindowBrush}">
  <StackPanel>
    <StackPanel Orientation="Horizontal" Padding="4"
                Background="LightGray" Name="toolbar">
      <Button Margin="6" Padding="6" Name="clickMeButton"
              Click="clickMeButton_Click">
        Click Me
      </Button>
      <TextBlock Text="First name:"
                 VerticalAlignment="Center" Margin="4" />
      <TextBox PlaceholderText="Enter your name"
                 VerticalAlignment="Center" Width="200" />
      <CheckBox IsChecked="True" Content="Enable acrylic background"
                 Margin="20,0,0,0" Name="enableAcrylic" />
    </StackPanel>
  </StackPanel>
</Grid>
```

Запустите приложение, выбрав в меню Debug (Отладка) команду Start Without Debugging (Запуск без отладки) или нажав сочетание клавиш Ctrl+F5, и обратите внимание на то, что через пользовательский интерфейс, в том числе подкрашенный акриловый материал, просвечивают оранжевые скалы и голубое небо, изображенные на фоновом рисунке (рис. 17.8).

Рис. 17.8

Акриловый материал потребляет большое количество системных ресурсов, поэтому, если приложение теряет фокус или уровень заряда аккумуляторной батареи устройства низок, данный эффект отключается автоматически.

Обзор эффекта отображения

Эффект отображения встроен в ряд элементов управления, например `ListView` и `NavigationView`, с которыми вы познакомитесь позже. Для других элементов управления вы можете включить подсветку, применив стиль темы.

Откройте файл `MainPage.xaml` и добавьте новую горизонтальную панель стека под панелью, используемой в качестве панели инструментов, а также добавьте сетку с кнопками для разметки калькулятора, как показано в листинге ниже:

```
<StackPanel Orientation="Horizontal">
  <Grid Background="DarkGray" Margin="10"
        Padding="5" Name="gridCalculator">
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
```

```
      <ColumnDefinition/>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Row="0" Grid.Column="0" Content="X" />
    <Button Grid.Row="0" Grid.Column="1" Content="/" />
    <Button Grid.Row="0" Grid.Column="2" Content="+" />
    <Button Grid.Row="0" Grid.Column="3" Content="-" />
    <Button Grid.Row="1" Grid.Column="0" Content="7" />
    <Button Grid.Row="1" Grid.Column="1" Content="8" />
    <Button Grid.Row="1" Grid.Column="2" Content="9" />
    <Button Grid.Row="1" Grid.Column="3" Content="0" />
    <Button Grid.Row="2" Grid.Column="0" Content="4" />
    <Button Grid.Row="2" Grid.Column="1" Content="5" />
    <Button Grid.Row="2" Grid.Column="2" Content="6" />
    <Button Grid.Row="2" Grid.Column="3" Content="." />
    <Button Grid.Row="3" Grid.Column="0" Content="1" />
    <Button Grid.Row="3" Grid.Column="1" Content="2" />
    <Button Grid.Row="3" Grid.Column="2" Content="3" />
    <Button Grid.Row="3" Grid.Column="3" Content="=" />
  </Grid>
</StackPanel>
```

Добавьте в элемент `Page` обработчик событий для `Loaded`, как показано в следующем листинге (выделено полужирным шрифтом):

```
<Page
  x:Class="FluentUwpApp.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:FluentUwpApp"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  Loaded="Page_Loaded">
```

Добавьте в файл `MainPage.xaml` инструкции в метод `Page_Loaded` для циклического прохода по всем кнопкам калькулятора, устанавливая для них одинаковый размер и применяя стиль отображения, как показано в листинге ниже:

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
  foreach (Button b in gridCalculator.Children.OfType<Button>())
  {
    b.FontSize = 24;
    b.Width = 54;
    b.Height = 54;
    b.Style = Resources.ThemeDictionaries["ButtonRevealStyle"] as Style;
  }
}
```

Запустите приложение, выполнив команду Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или воспользовавшись сочетанием клавиш Ctrl+F5, и обратите внимание, что калькулятор запускается с простым серым интерфейсом (рис. 17.9).

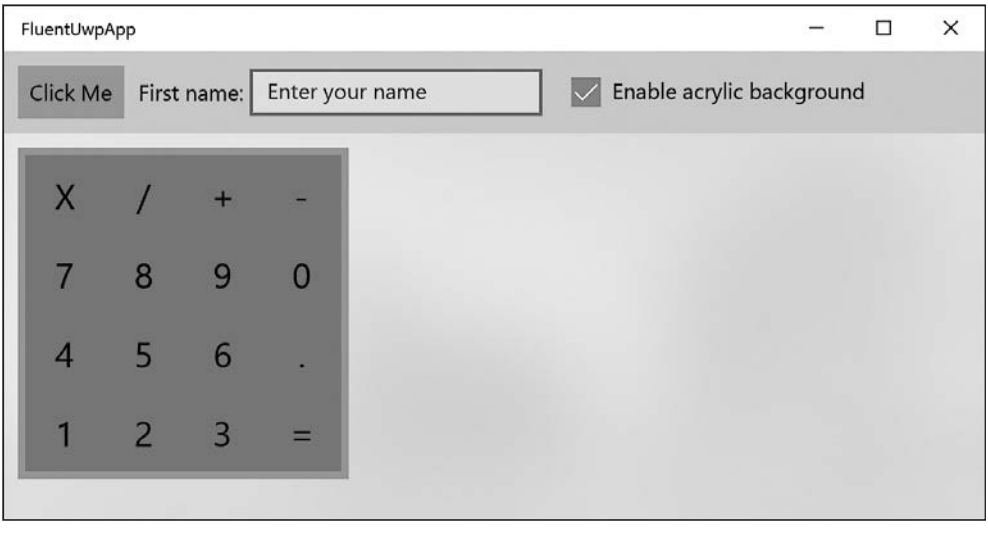


Рис. 17.9

Когда пользователь устанавливает указатель мыши на калькулятор, включается эффект отображения (рис. 17.10).

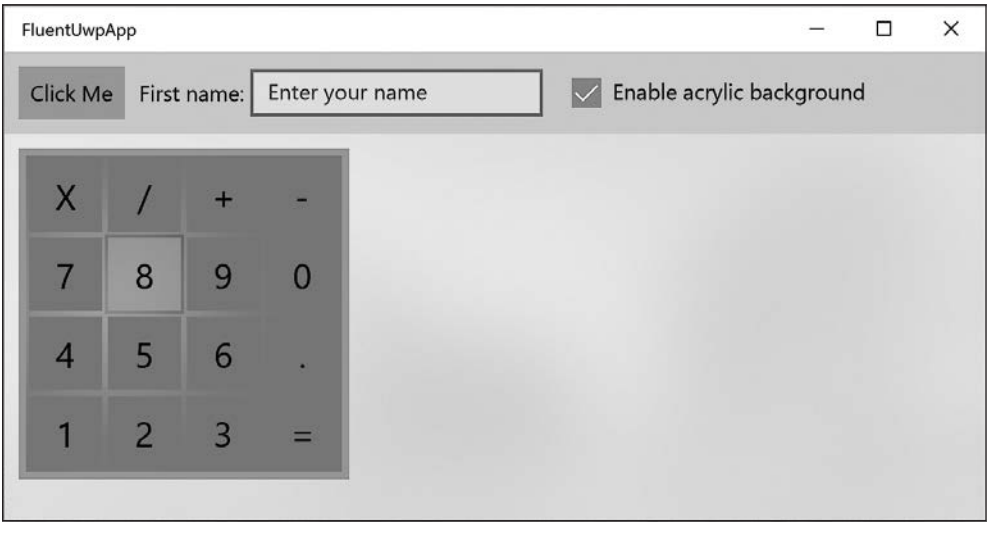


Рис. 17.10

Установка дополнительных элементов управления

Вдобавок к десяткам встроенных элементов управления, можно установить дополнительные в формате пакетов NuGet. Один из лучших пакетов — UWP Community Toolkit, более подробную информацию о котором можно получить, перейдя по ссылке http://www.uwpcommunitytoolkit.com.

В проекте FluentUwpApp щелкните правой кнопкой мыши на пункте References (Ссылки) и выберите пункт Manage NuGet Packages (Управление пакетами NuGet). Перейдите на вкладку Browse (Обзор) и выполните поиск пакета Microsoft.Toolkit.Uwp.UI.Controls, после чего нажмите кнопку Install (Установить) (рис. 17.11).

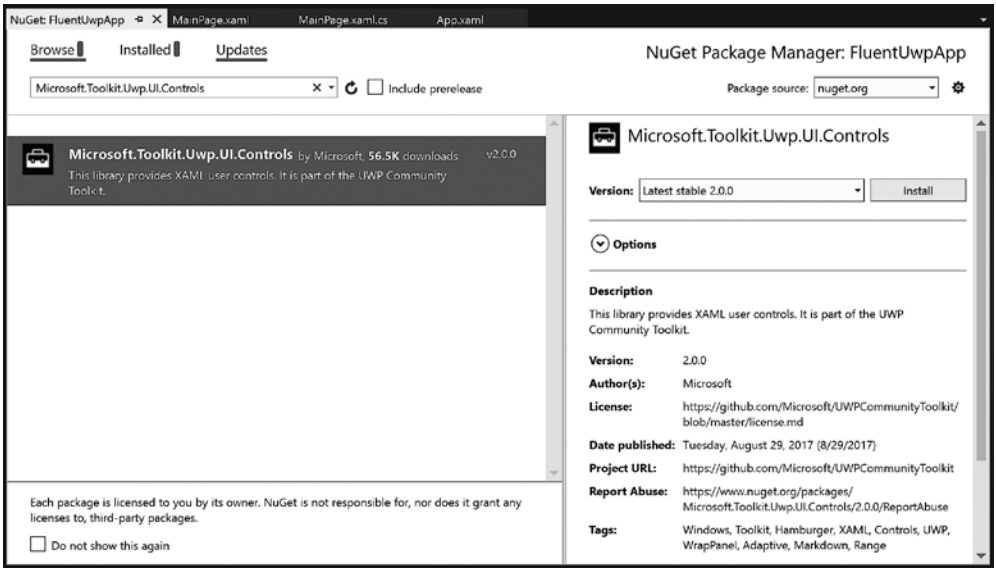


Рис. 17.11

Изучите изменения и примите лицензионное соглашение.

Откройте файл `MainPage.xaml` и в элемент `Page` импортируйте пространство имен набора инструментов в качестве префикса `kit`, как показано в следующем листинге (выделено полужирным шрифтом):

```
<Page
  x:Class="FluentUwpApp.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:FluentUwpApp"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:kit="using:Microsoft.Toolkit.Uwp.UI.Controls"
  mc:Ignorable="d"
  Loaded="Page_Loaded">
```

После сетки калькулятора добавьте поле ввода текста и блок текста с разметкой (`markdown text block`), как показано в листинге ниже:

```
<TextBox Name="markdownSource" Text="# Welcome"
         Header="Enter some Markdown text:"
         VerticalAlignment="Stretch" Margin="5"
         AcceptsReturn="True" />
<kit:MarkdownTextBlock
         Text="{Binding ElementName=markdownSource, Path=Text}"
         VerticalAlignment="Stretch"
         HorizontalAlignment="Stretch" Margin="5"/>
```

Запустите приложение, выполнив команду Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или воспользовавшись сочетанием клавиш Ctrl+F5, и обратите внимание, что пользователь может ввести синтаксис разметки в поле ввода текста — и этот синтаксис будет обработан в блоке текста с разметкой (рис. 17.12).

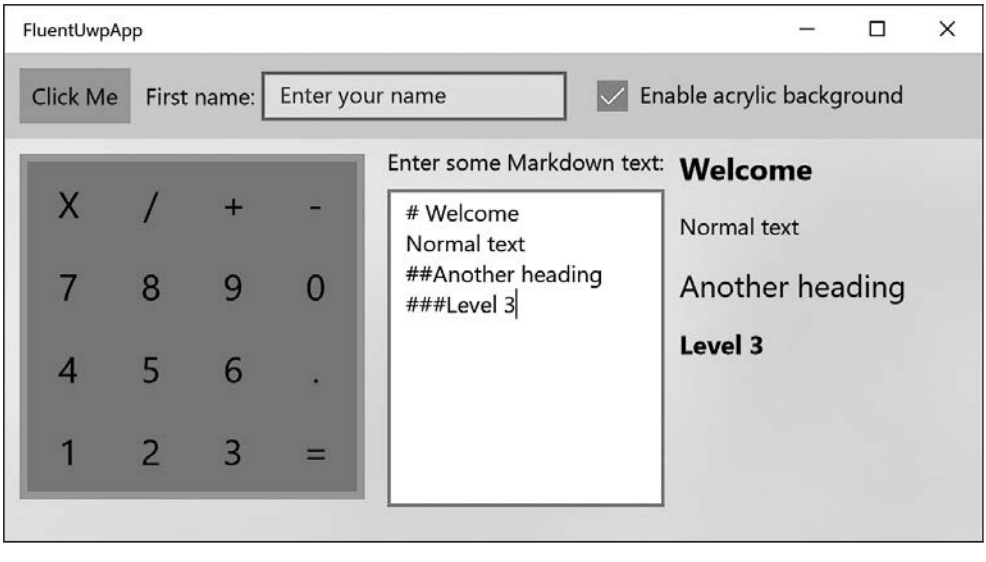


Рис. 17.12

Пакет UWP Community Toolkit включает в себя десятки элементов управления, анимаций, расширений и сервисов.

Использование ресурсов и шаблонов

При разработке GUI понадобится часто задействовать ресурсы, например кисти, для окрашивания фона элементов управления. Эти ресурсы можно определить в одном месте и сделать доступными для всего приложения.

Общий доступ к ресурсам

На панели Solution Explorer (Обозреватель решений) дважды щелкните на файле `App.xaml` (рис. 17.13).

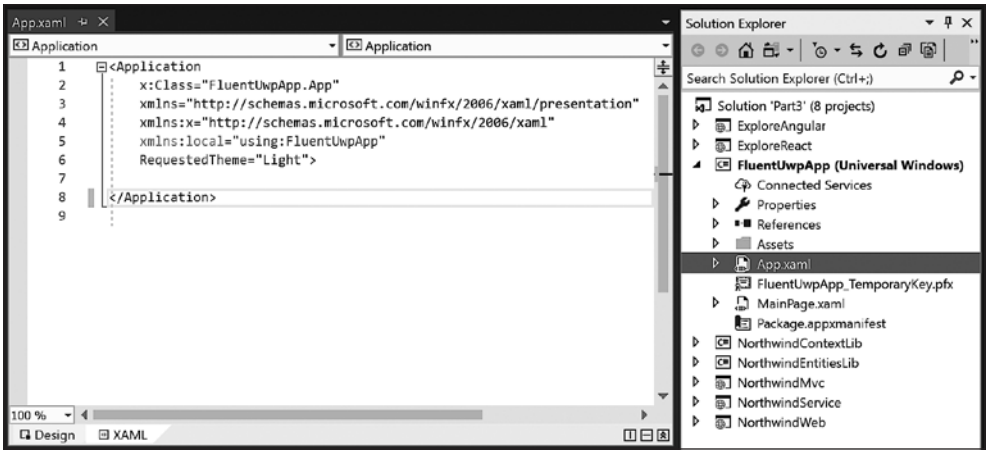


Рис. 17.13

Добавьте такую разметку к существующей в элементе `<Application>`:

```
<Application.Resources>
  <LinearGradientBrush x:Key="rainbow">
    <GradientStop Color="Red" Offset="0" />
    <GradientStop Color="Orange" Offset="0.1" />
    <GradientStop Color="Yellow" Offset="0.3" />
    <GradientStop Color="Green" Offset="0.5" />
    <GradientStop Color="Blue" Offset="0.7" />
    <GradientStop Color="Indigo" Offset="0.9" />
    <GradientStop Color="Violet" Offset="1" />
  </LinearGradientBrush>
</Application.Resources>
```

В файле `MainPage.xaml` измените код элемента `StackPanel` панели инструментов, чтобы окрасить фон кистью `rainbow`, как показано в следующем листинге:

```
<StackPanel Orientation="Horizontal" Padding="4"
Background="{StaticResource rainbow}" Name="toolbar">
```

На панели Design (Конструктор) вы увидите ресурс `rainbow` и доступные элементы в меню IntelliSense (рис. 17.14).

Ресурс может быть экземпляром любого объекта. Чтобы предоставить к нему общий доступ в приложении, определите его в файле App.xaml и присвойте ему уникальный ключ (Key). Чтобы применить ресурс к свойству элемента, используйте ключевые слова StaticResource и Key.

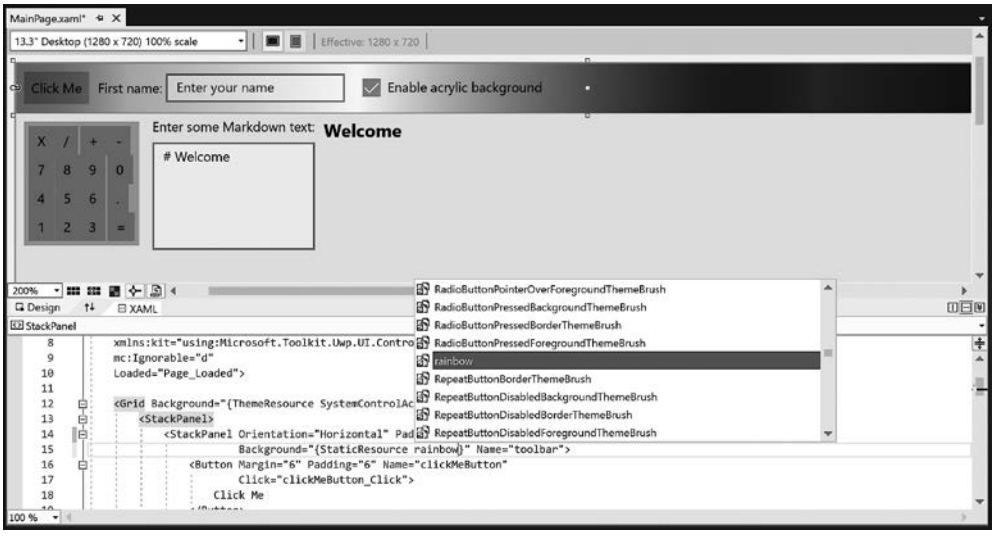


Рис. 17.14

Ресурсы могут быть определены и сохранены внутри любого XAML-элемента, а не только на уровне приложения. Так, например, если ресурс нужен только для MainPage, его можно определить там. Вы также можете динамически загружать XAML-файлы во время выполнения.

Замена шаблона элемента управления

Вы можете изменить внешний вид элемента управления, заменив его шаблон по умолчанию. Кнопка по умолчанию отображается плоской и прозрачной.

Один из наиболее популярных общих ресурсов — `Style`. Он позволяет определять сразу несколько свойств. Если стиль имеет уникальный ключ, то его нужно указывать явно, как и в предыдущем примере с линейным градиентом. Если ключа нет, то он будет применен автоматически на основании значения свойства `TargetType`.

Добавьте в файл `App.xaml` в элемент `<Application.Resources>` показанную ниже разметку и обратите внимание на то, что элемент `<Style>` автоматически установит свойство `Template` для всех элементов управления `TargetType` (то есть кнопок), чтобы применить к ним определенный шаблон:

```
<ControlTemplate x:Key="DarkGlassButton" TargetType="Button">
  <Border BorderBrush="#FFFFFFFF"
          BorderThickness="1,1,1,1" CornerRadius="4,4,4,4">
    <Border x:Name="border" Background="#7F000000"
            BorderBrush="#FF000000" BorderThickness="1,1,1,1"
            CornerRadius="4,4,4,4">
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="*"/>
```

```
          <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Border Opacity="0" HorizontalAlignment="Stretch"
                x:Name="glow" Width="Auto" Grid.RowSpan="2"
                CornerRadius="4,4,4,4">
        </Border>
        <ContentPresenter HorizontalAlignment="Center"
                          VerticalAlignment="Center" Width="Auto"
                          Grid.RowSpan="2" Padding="4"/>
        <Border HorizontalAlignment="Stretch" Margin="0,0,0,0"
                x:Name="shine" Width="Auto"
                CornerRadius="4,4,0,0">
          <Border.Background>
            <LinearGradientBrush EndPoint="0.5,0.9"
                                 StartPoint="0.5,0.03">
              <GradientStop Color="#99FFFFFF" Offset="0"/>
              <GradientStop Color="#33FFFFFF" Offset="1"/>
            </LinearGradientBrush>
          </Border.Background>
        </Border>
      </Grid>
    </Border>
  </Border>
</ControlTemplate>
<Style TargetType="Button">
  <Setter Property="Template"
          Value="{StaticResource DarkGlassButton}" />
  <Setter Property="Foreground" Value="White" />
</Style>
```

Перезапустите приложение и проанализируйте результат вывода. Обратите внимание на эффект *«затененного стекла»* кнопки (рис. 17.15).

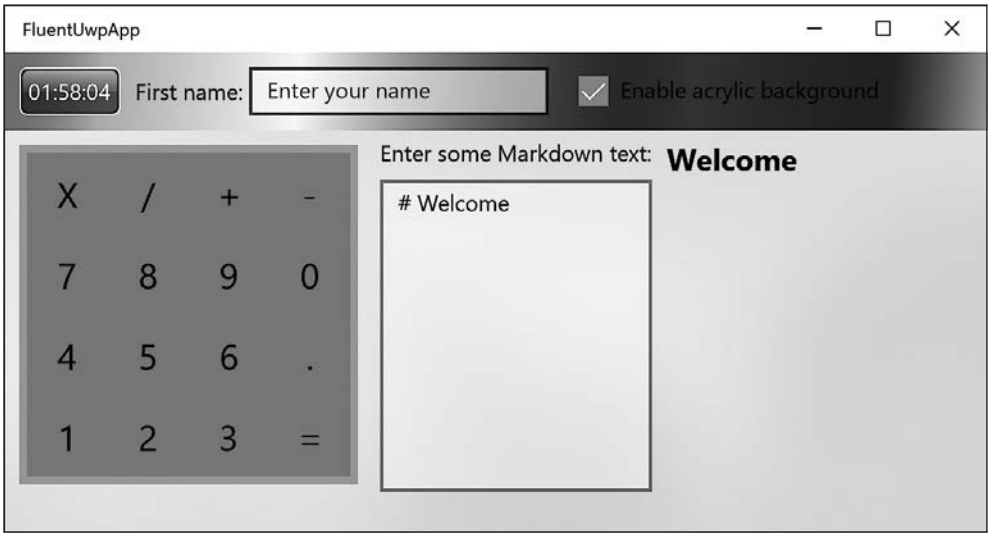


Рис. 17.15

Кнопки калькулятора с эффектом «затененного стекла» не влияют на работу приложения, поскольку их стили заменяются с помощью кода после загрузки страницы.

Привязка данных

При разработке графических пользовательских интерфейсов часто требуется привязывать свойство одного элемента управления к другому или к определенным данным.

Привязка к элементам

Добавьте в файл `MainPage.xaml` текстовый блок для ввода команд, ползунковый регулятор для установки угла поворота; сетку, содержащую панель со стеком элементов и текстовые блоки, отображающие угол поворота в градусах; радиальный датчик из набора инструментов UWP Community Toolkit и красный квадрат, который будет вращаться, как показано в листинге ниже:

```
<TextBlock Grid.ColumnSpan="2" Margin="10">
  Use the slider to rotate the square:</TextBlock>
<Slider Value="180" Minimum="0" Maximum="360"
        Name="sliderRotation" Margin="10,0" />
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <StackPanel Orientation="Horizontal"
              VerticalAlignment="Center"
              HorizontalAlignment="Center">
    <TextBlock
    Text="{Binding ElementName=sliderRotation, Path=Value}"
               FontSize="30" />
    <TextBlock Text="degrees" FontSize="30" Margin="10,0" />
  </StackPanel>
  <kit:RadialGauge Grid.Column="1" Minimum="0"
                   Maximum="360"
                   Value="{Binding ElementName=sliderRotation,
                   Path=Value}"
                   Height="200" Width="200" />
  <Rectangle Grid.Column="2" Height="100" Width="100" Fill="Red">
    <Rectangle.RenderTransform>
      <RotateTransform
        Angle="{Binding ElementName=sliderRotation, Path=Value}" />
      </Rectangle.RenderTransform>
  </Rectangle>
</Grid>
```

Обратите внимание: текст, значение радиального датчика и угол поворота — все эти данные привязаны к значению ползункового регулятора.

Запустите приложение и перетащите ползунок, чтобы повернуть красный квадрат (рис. 17.16).

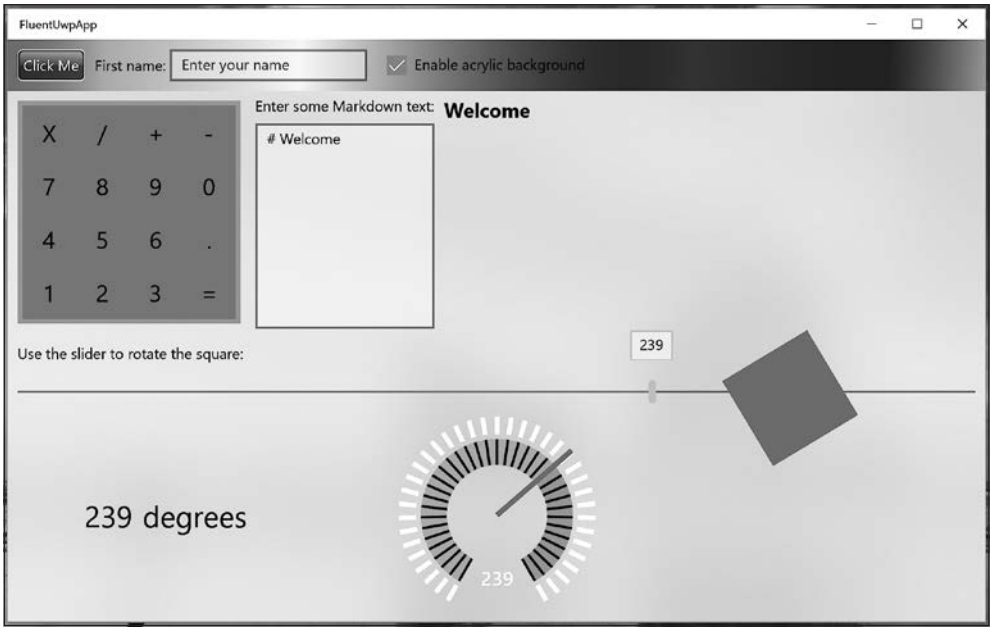


Рис. 17.16

Привязка к источникам данных

Чтобы продемонстрировать привязку к источникам данных, мы создадим приложение для базы данных `Northwind`, отображающее категории и товары.

Изменение NorthwindService

Откройте решение `Part3` и разверните проект NorthwindService.

Щелкните правой кнопкой мыши на каталоге `Controllers` и выполните команду Add ▸ New Item (Добавить ▸ Новый элемент), в открывшемся диалоговом окне выберите элемент Web API Controller Class (Класс контроллера веб-API), присвойте ему имя `CategoriesControllers.cs` и измените код так, чтобы в этом классе было два метода `HttpGet`, использующих контекст базы данных `Northwind` для получения списка всех категорий или единственной категории по ее ID, как показано в следующем листинге:

```
using Microsoft.AspNetCore.Mvc;
using Packt.CS7;
using System.Collections.Generic;
using System.Linq;

namespace NorthwindService.Controllers
```

```
{
  [Route("api/[controller]")]
  public class CategoriesController : Controller
  {
    private readonly Northwind db;

    public CategoriesController(Northwind db)
    {
      this.db = db;
    }

    // GET: api/categories
    [HttpGet]
    public IEnumerable<Category> Get()
    {
      var categories = db.Categories.ToArray();
      return categories;
    }

    // GET api/categories/5
    [HttpGet("{id}")]
    public Category Get(int id)
    {
      var category = db.Categories.Find(id);
      return category;
    }
  }
}
```

Щелкните правой кнопкой мыши на каталоге `Controllers` и выполните команду Add ▸ New Item (Добавить ▸ Новый элемент), в открывшемся диалоговом окне выберите элемент Web API Controller Class (Класс контроллера веб-API), присвойте ему имя `ProductsControllers.cs` и измените код так, чтобы в этом классе было два метода `HttpGet`, использующих контекст базы данных `Northwind` для получения списка всех продуктов или продуктов в категории по ее ID, как показано в листинге ниже:

```
using Microsoft.AspNetCore.Mvc;
using Packt.CS7;
using System.Collections.Generic;
using System.Linq;

namespace NorthwindService.Controllers
{
  [Route("api/[controller]")]
  public class ProductsController : Controller
  {
    private readonly Northwind db;

    public ProductsController(Northwind db)
    {
      this.db = db;
    }

    // GET: api/products
    [HttpGet]
```

```
    public IEnumerable<Product> Get()
    {
      var products = db.Products.ToArray();
      return products;
    }

    // GET api/products/5
    [HttpGet("{id}")]
    public IEnumerable<Product> GetByCategory(int id)
    {
      var products = db.Products.Where(
        p => p.CategoryID == id).ToArray();
        return products;
    }
  }
}
```

Протестируйте работу сервиса `NorthwindService`, введя относительный URL `/api/products/1` для подтверждения того, что сервис вернет только перечень напитков.

Создание приложения Northwind

Откройте решение `Part3` и выполните команду File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) выполните команду Visual C# ▸ Windows Universal (Универсальные приложения Windows). В центре диалогового окна выберите пункт Blank App (Universal Windows) (Пустое приложение (универсальные приложения для Windows)), введите расположение `C:\Code\Part3`, задайте имя `NorthwindFluent` и нажмите кнопку OK.

Выберите последнюю версию Windows 10 в раскрывающихся списках Target Version (Целевая версия) и Minimum Version (Минимальная версия) и нажмите кнопку OK.

В проекте NorthwindFluent щелкните на элементе Blank Page (Пустая страница)[1] с именем `NotImplementedPage` и внутри существующего элемента `Grid` добавьте блок текста с надписью `"Not yet implemented"`, центрируйте данный элемент относительно страницы, как показано в следующем листинге:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <TextBlock Text="Not yet implemented." VerticalAlignment="Center"
HorizontalAlignment="Center" FontSize="20" />
</Grid>
```

Добавьте в проект NorthwindFluent ссылку на проект NorthwindEntities.

Добавьте в проект NorthwindFluent несколько изображений в каталог `Assets` и присвойте им такие имена:

- `categories.jpeg`;
- `category1-small.jpeg`;
- `category2-small.jpeg` и т. д.

[1] Обратите внимание: в версии VS2017 15.5.4 он не был создан автоматически. — *Примеч. пер.*

Добавьте в проект NorthwindFluent класс `CategoriesViewModel` и заполните его свойство `Categories`, используя контекст базы данных `Northwind`, как показано в листинге ниже:

```
using Packt.CS7;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Net.Http;
using System.Runtime.Serialization.Json;

namespace NorthwindFluent
{
  public class CategoriesViewModel
  {
    public class CategoryJson
    {
      public int categoryID;
      public string categoryName;
      public string description;
    }

    public ObservableCollection<Category> Categories { get; set; }

    public CategoriesViewModel()
    {
      using (var http = new HttpClient())
      {
        http.BaseAddress = new Uri("http://localhost:5001/");

        var serializer = new
        DataContractJsonSerializer(typeof(List<CategoryJson>));

        var stream = http.GetStreamAsync("api/categories").Result;

        var cats = serializer.ReadObject(stream) as List<CategoryJson>;

        var categories = cats.Select(c => new Category
                    { CategoryID = c.categoryID,
                      CategoryName = c.categoryName,
                      Description = c.description });
        Categories = new ObservableCollection<Category>(categories);
      }
    }
  }
}
```

Нам требовалось определить класс CategoryJson, поскольку класс DataContractJsonSerializer недостаточно умен для распознавания верблюжьего регистра, используемого в JSON, и автоматической конвертации в регистр заголовков, применяющийся в C#. Поэтому самым простым решением будет выполнить преобразование вручную с помощью проекции LINQ.

Добавьте в проект NorthwindFluent класс `CategoryIDToImageConverter`, реализуйте интерфейс `IValueConverter` и конвертируйте целое число, соответствующее идентификатору категории, в действительный путь к соответствующему файлу изображения, как показано в следующем листинге:

```
using System;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Media.Imaging;

namespace NorthwindFluent
{
  public class CategoryIDToImageConverter : IValueConverter
  {
    public object Convert(object value, Type targetType, object parameter, string
language)
    {
      int n = (int)value;
      string path =
      $"{Environment.CurrentDirectory}/Assets/category{n}-small.jpeg";
      var image = new BitmapImage(new Uri(path));
      return image;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
string language)
    {
      throw new NotImplementedException();
    }
  }
}
```

Добавьте в проект NorthwindFluent элемент Blank Page (Пустая страница) с именем `CategoriesPage`, как показано в листинге ниже:

```
<Page
  x:Class="NorthwindFluent.CategoriesPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:NorthwindFluent"
  xmlns:nw="using:Packt.CS7"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
  <Page.Resources>
    <local:CategoryIDToImageConverter x:Key="id2image" />
  </Page.Resources>

  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <ParallaxView Source="{x:Bind ForegroundElement}" VerticalShift="50">
      <Image x:Name="BackgroundImage" Source="Assets/categories.jpeg"
             Stretch="UniformToFill"/>
    </ParallaxView>
    <ListView x:Name="ForegroundElement"
              ItemsSource="{x:Bind ViewModel.Categories}">
```

```
      <ListView.Header>
        <Grid Padding="20"
              Background="{ThemeResource SystemControlAcrylicElementBrush}">
          <TextBlock Style="{StaticResource TitleTextBlockStyle}"
                     FontSize="24"
                     VerticalAlignment="Center"
                     Margin="12,0"
                     Text="Categories"/>
        </Grid>
      </ListView.Header>
      <ListView.ItemTemplate>
        <DataTemplate x:DataType="nw:Category">
          <Grid Margin="4">
            <Grid.ColumnDefinitions>
              <ColumnDefinition />
              <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <Image Source="{x:Bind CategoryID,
                   Converter={StaticResource id2image}}"
                   Stretch="UniformToFill" Height="200"
                   Width="300" />
            <StackPanel
                   Background=
                   "{ThemeResource SystemControlAcrylicElementMediumHighBrush}"
                   Padding="10" Grid.Column="1">
              <TextBlock Text="{x:Bind CategoryName}" FontSize="20" />
              <TextBlock Text="{x:Bind Description}" FontSize="16" />
            </StackPanel>
          </Grid>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
  </Grid>
</Page>
```

Обратите внимание, что данный код выполняет следующие задачи:

- импортирует пространство имен `Packt.CS7`, используя префикс элемента `nw`;
- определяет ресурсную страницу, инстанцирующую конвертер идентификаторов категорий в изображения;
- применяет параллакс для установки большого изображения в качестве прокручивающегося фона элемента переднего плана — списочного представления категорий — таким образом, что при прокрутке списка происходит небольшое смещение фонового изображения;
- привязывает списочное представление к коллекции категорий модели;
- задает акриловый заголовок для списочного представления при выводе в приложение;
- устанавливает для списочного представления шаблон элемента для обработки каждой категории с помощью имени, описания и изображения, основываясь

на результате преобразования идентификатора категории в изображение, загружаемое из файловой системы.

Откройте файл `CategoriesPage.xaml.cs` и добавьте инструкции для определения свойства `ViewModel`, а затем установите это свойство в конструкторе, как показано в следующем листинге:

```
public sealed partial class CategoriesPage : Page
{
  public CategoriesViewModel ViewModel { get; set; }

  public CategoriesPage()
  {
    this.InitializeComponent();
    ViewModel = new CategoriesViewModel();
  }
}
```

Откройте файл `MainPage.xaml` и добавьте элементы для определения представления в виде дерева, на панели которого автоматически используется акриловый материал и эффект отображения, как показано в листинге ниже:

```
<Page
  x:Class="NorthwindFluent.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:NorthwindFluent"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <NavigationView x:Name="NavView"
                  ItemInvoked="NavView_ItemInvoked"
                  Loaded="NavView_Loaded">

  <NavigationView.AutoSuggestBox>
    <AutoSuggestBox x:Name="ASB" QueryIcon="Find"/>
  </NavigationView.AutoSuggestBox>

  <NavigationView.HeaderTemplate>
    <DataTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto"/>
          <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <TextBlock Style="{StaticResource TitleTextBlockStyle}"
                   FontSize="28"
                   VerticalAlignment="Center"
                   Margin="12,0"
                   Text="Northwind Fluent"/>
        <CommandBar Grid.Column="1"
                       HorizontalAlignment="Right"
```

```
                    DefaultLabelPosition="Right"
                    Background=
                "{ThemeResource SystemControlBackgroundAltHighBrush}">
            <AppBarButton Label="Refresh" Icon="Refresh"
                   Name="RefreshButton" Click="RefreshButton_Click"/>
          </CommandBar>
        </Grid>
      </DataTemplate>
    </NavigationView.HeaderTemplate>

    <Frame x:Name="ContentFrame">
      <Frame.ContentTransitions>
        <TransitionCollection>
          <NavigationThemeTransition/>
        </TransitionCollection>
      </Frame.ContentTransitions>
    </Frame>

  </NavigationView>
</Page>
```

Откройте файл `MainPage.xaml` и измените его содержимое, как показано в следующем листинге:

```
using System;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace NorthwindFluent
{
  public sealed partial class MainPage : Page
  {
    public MainPage()
    {
      this.InitializeComponent();
    }

    private void NavView_Loaded(object sender, RoutedEventArgs e)
    {
      NavView.MenuItems.Add(new NavigationViewItem
      { Content = "Categories",
        Icon = new SymbolIcon(Symbol.BrowsePhotos),
        Tag = "categories" });
      NavView.MenuItems.Add(new NavigationViewItem
      { Content = "Products",
        Icon = new SymbolIcon(Symbol.AllApps),
        Tag = "products" });
      NavView.MenuItems.Add(new NavigationViewItem
      { Content = "Suppliers",
        Icon = new SymbolIcon(Symbol.Contact2),
        Tag = "suppliers" });
      NavView.MenuItems.Add(new NavigationViewItemSeparator());
      NavView.MenuItems.Add(new NavigationViewItem
```

```
      { Content = "Customers",
        Icon = new SymbolIcon(Symbol.People),
        Tag = "customers" });
      NavView.MenuItems.Add(new NavigationViewItem
      { Content = "Orders",
        Icon = new SymbolIcon(Symbol.PhoneBook),
        Tag = "orders" });
      NavView.MenuItems.Add(new NavigationViewItem
      { Content = "Shippers",
        Icon = new SymbolIcon(Symbol.PostUpdate),
        Tag = "shippers" });
    }

    private void NavView_ItemInvoked(
      NavigationView sender, NavigationViewItemInvokedEventArgs args)
    {
      switch (args.InvokedItem.ToString())
      {
        case "Categories":
              ContentFrame.Navigate(typeof(CategoriesPage));
              break;
        default:
              ContentFrame.Navigate(typeof(NotImplementedPage));
              break;
      }
    }

    private async void RefreshButton_Click(
            object sender, RoutedEventArgs e)
    {
      var notImplementedDialog = new ContentDialog
      {
        Title = "Not implemented",
        Content =
          "The Refresh functionality has not yet been implemented.",
          CloseButtonText = "OK"
      };
      ContentDialogResult result = await notImplementedDialog.ShowAsync();
    }
  }
}
```

Кнопка Refresh (Обновить) еще не реализована, поэтому можно отобразить диалоговое окно. Как и большинство API в UWP, метод для отображения диалогового окна асинхронный.

Ключевое слово `await` можно использовать для любого задания `Task`. Это, в свою очередь, означает, что основной поток не будет заблокирован на время ожидания, но запомнит свое местоположение среди инструкций. Таким образом по завершении задания `Task` основной поток продолжает выполнение с того же момента, на котором оно было прервано. Это позволяет писать код, который выглядит так же просто, как синхронный, но на заднем плане является более сложным.

Внутри ключевое слово await создает машину состояний для управления сложностью передачи состояний между рабочими потоками и потоком пользовательского интерфейса.

Обратите внимание: чтобы использовать ключевое слово `await`, нужно отметить содержащий его метод ключевым словом `async`. Эти ключевые слова всегда работают в паре.

Убедитесь в том, что веб-сервис работает и вы настроили приложение NorthwindFluent для развертывания, а затем запустите приложение, выполнив команду Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или воспользовавшись сочетанием клавиш Ctrl+F5.

Измените размер окна, чтобы продемонстрировать отзывчивый дизайн, таким образом:

- при узком окне панель навигации скрывается, и пользователь должен щелкнуть на значке гамбургера с тремя горизонтальными линиями, чтобы отобразить панель навигации;
- при среднем окне панель навигации узкая и отображает только значки, акриловый фон отключен;
- при широком окне панель навигации отображается полностью, акриловый фон включен.

Наводя указатель мыши на панель навигации, обратите внимание на эффект отображения, а при щелчке на пунктах меню (например, Orders) — на то, что панель подсветки расширяется и включается анимация, сопровождающая ее перемещение. Кроме того, отображается сообщение Not yet implemented (Еще не реализовано) (рис. 17.17).

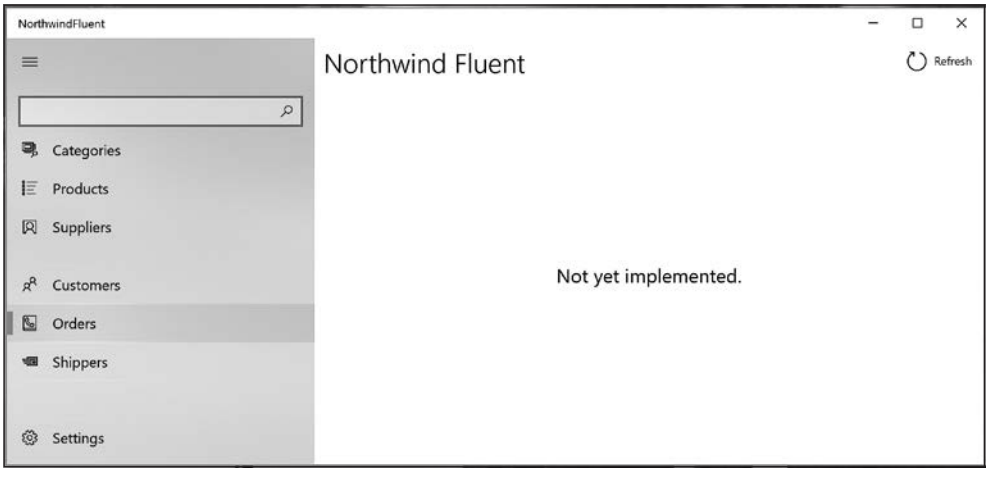


Рис. 17.17

Щелкните на пункте Categories и обратите внимание на акриловый заголовок Categories в приложении и эффект параллакса на фоновом изображении при прокрутке вверх и вниз по списку категорий (рис. 17.18).

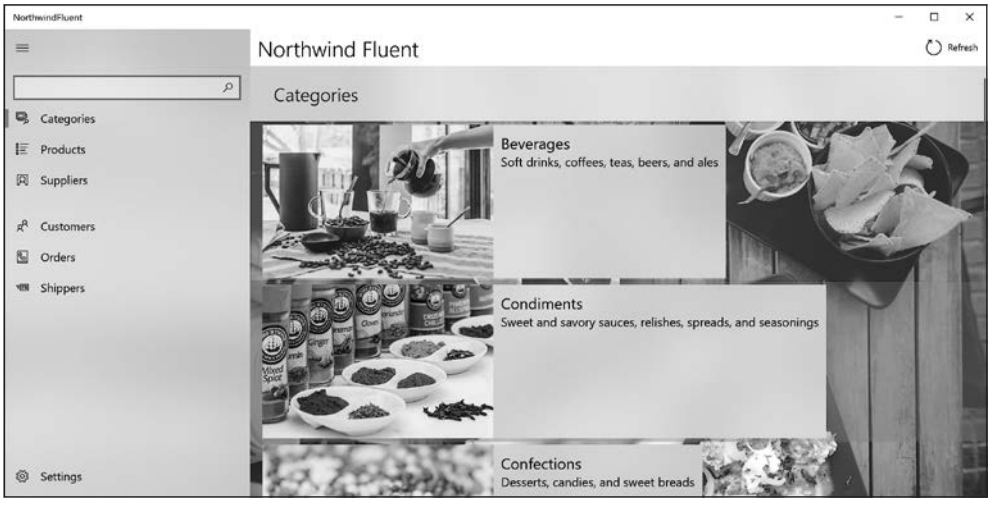


Рис. 17.18

Щелкните на разделе с гамбургерами, чтобы свернуть навигационное представление и предоставить больше места для списка категорий (рис. 17.19).

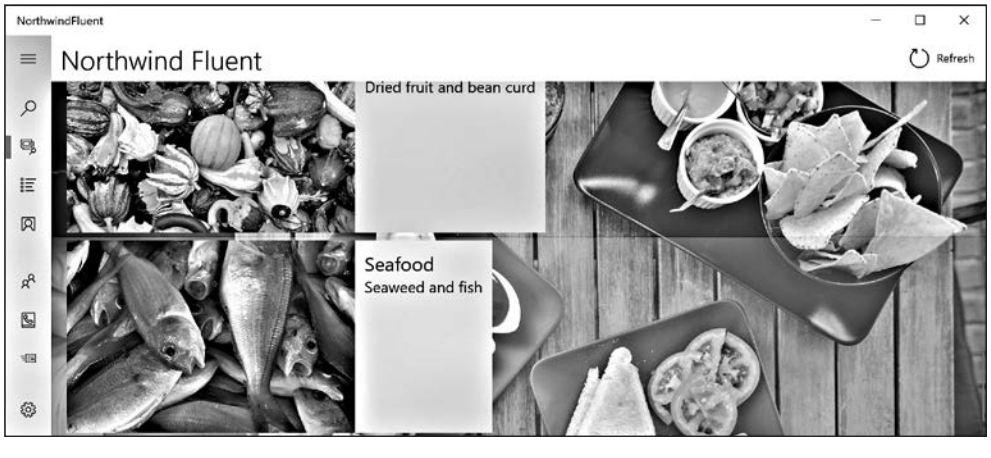


Рис. 17.19

Реализацию отображения страницы продуктов в момент, когда пользователь щелкает на выбранной категории, оставим в качестве упражнения для читателя.

Создание приложений с помощью Windows Template Studio

Корпорация Microsoft создала расширение Windows Template Studio для Visual Studio, чтобы ускорить создание приложений UWP. Мы воспользуемся этим расширением для разработки нового приложения, демонстрирующего функционал и практические советы по применению Windows Template Studio.

Установка Windows Template Studio

Выполните команду Tools ▸ Extensions and Updates (Средства ▸ Расширения и обновления), выберите категорию Online (В сети) и в окне поиска введите `Windows Template Studio`, после чего нажмите кнопки Download (Скачать) и Close (Закрыть) (рис. 17.20).

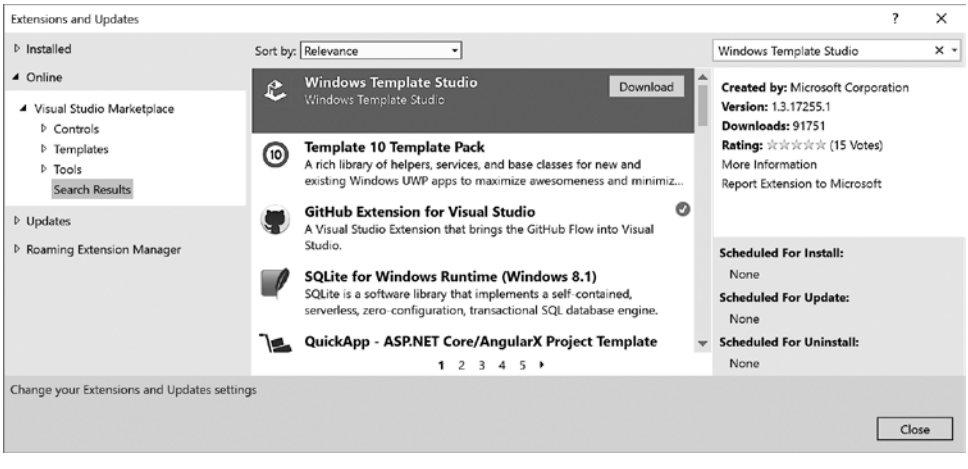


Рис. 17.20

Выйдите из Visual Studio 2017 и дождитесь установки расширения с помощью VSIX Installer (Установщик VSIX).

В настоящее время программа Windows Template Studio может сбоить, если в решении реализовано сразу несколько проектов, поэтому я порекомендовал бы всегда создавать проекты в новом решении с помощью данной программы. После создания проекта вы сможете вручную добавить его в другое решение.

Выбор типов проектов, сред, страниц и функциональности

Запустите Visual Studio 2017, откройте решение `Part3` и выполните команду File ▸ New ▸ Project (Файл ▸ Новый ▸ Проект).

В диалоговом окне New Project (Новый проект) в списке Installed (Установленные) выполните команду Visual C# ▸ Windows Universal (Универсальные прило-

жения Windows). В центре диалогового окна выберите пункт Template Studio (Universal Windows) (Пустое приложение (универсальные приложения для Windows)), введите расположение `C:\Code\Part3`, задайте имя `NorthwindUwp` и нажмите кнопку OK.

В диалоговом окне Select project type and framework (Выберите тип и фреймворк проекта) установите флажки Navigation Pane и MVVM Basic, затем нажмите кнопку Next (Далее) (рис. 17.21).

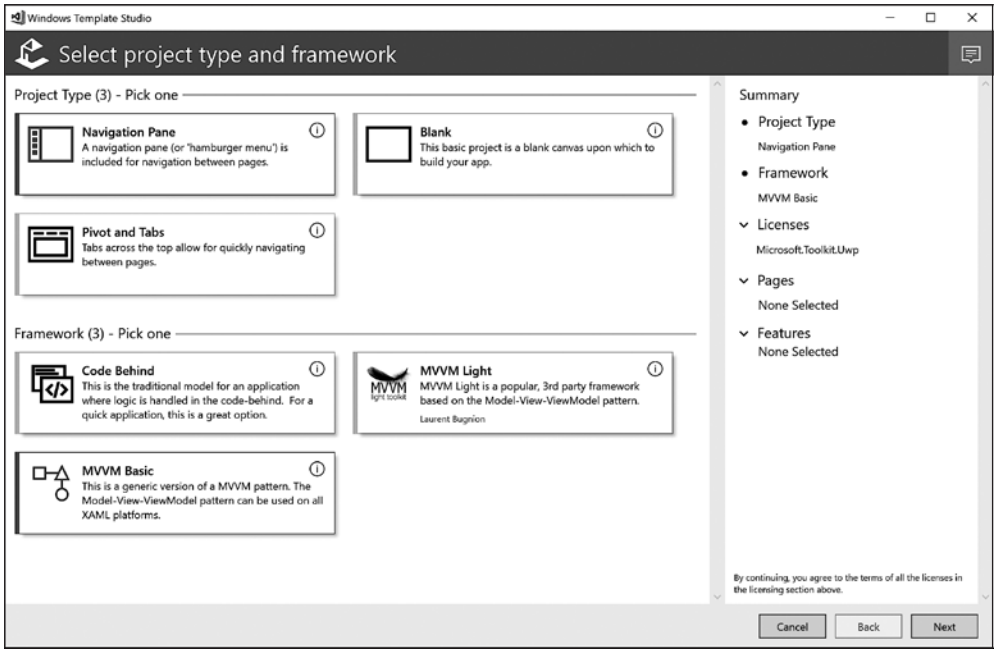


Рис. 17.21

В диалоговом окне Select pages and features (Выбрать страницы и функции) нажмите плюс на панели Settings (Настройки), оставьте имя `Settings` и нажмите кнопку с галочкой (рис. 17.22).

Повторите вышеописанные действия для следующих панелей:

- добавьте страницу Blank с именем `AboutNorthwindApp`;
- добавьте страницу Web View с именем `AngularWebView`;
- добавьте страницу Master/Detail с именем `CategoryMasterDetail`;
- добавьте страницу Grid с именем `CustomersGrid`;
- добавьте страницу Map с именем `CustomerMap`;
- добавьте элемент First Run Prompt.

В разделе Summary (Сводка) подтвердите создание вышеперечисленных страниц, нажав кнопку Create (Создать) (рис. 17.23).

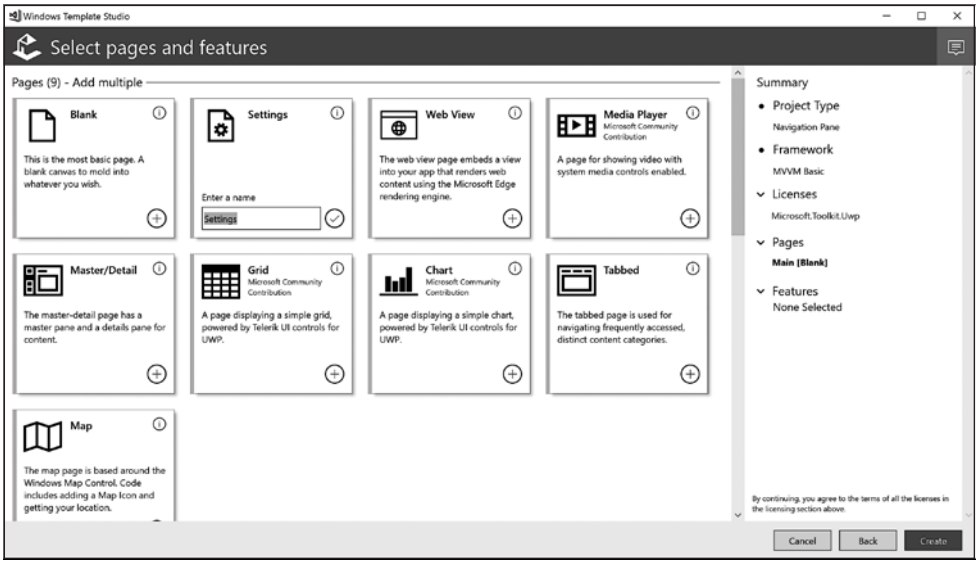


Рис. 17.22

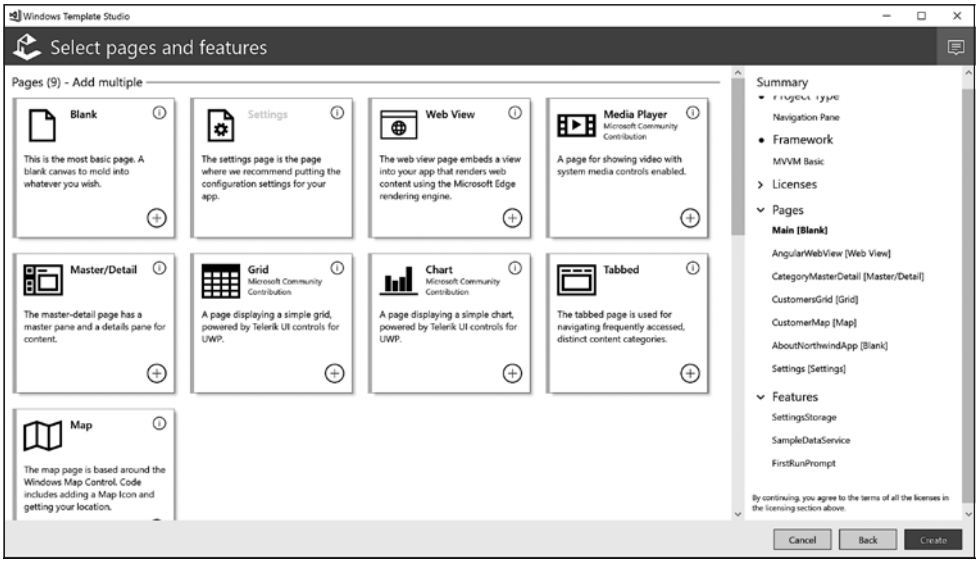


Рис. 17.23

Если создание нового проекта завершится успешно, то закройте решение и откройте решение `Part3`.

Выполните команду File ▸ Add ▸ Existing Project (Файл ▸ Добавить ▸ Существующий проект) и выберите проект NorthwindUwp.csproj.

Ретаргетинг проекта

На панели Solution Explorer (Обозреватель решений) в проекте NorthwindUwp откройте пункт Properties (Свойства), выберите последнюю версию Windows 10 в раскрывающихся списках Target Version (Целевая версия) и Minimum Version (Минимальная версия) и нажмите кнопку OK.

На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на решении Part3 и выберите пункт Properties (Свойства) или воспользуйтесь сочетанием клавиш Alt+Enter.

Установите переключатель в положение Multiple startup projects (Запуск нескольких проектов), выберите, чтобы проект ExploreAngular запускался первым, за ним следовал проект NorthwindService и, наконец, проект NorthwindUwp (рис. 17.24).

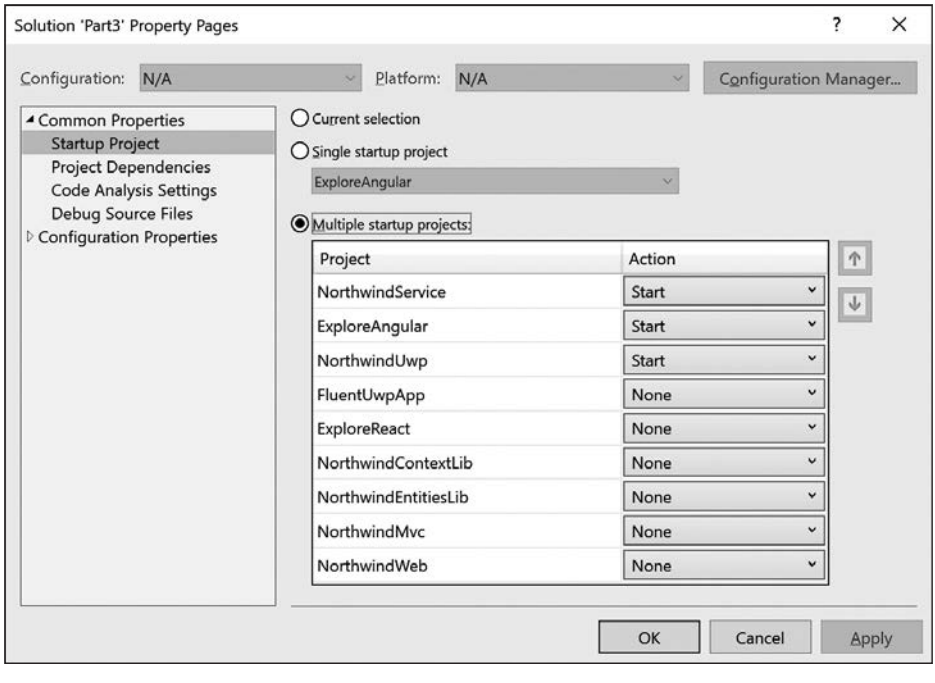


Рис. 17.24

Настройка нескольких видов

На панели Solution Explorer (Обозреватель решений) раскройте проект NorthwindUwp и каталог `ViewModels`, откройте файл `AngularWebViewViewModel.cs`, и измените строковую константу `DefaultUrl` ранее созданного вами веб-приложения Angular, прослушивающего порт `5002`, как показано в коде ниже:

```
private const string DefaultUrl = "http://localhost:5002";
```

На панели Solution Explorer (Обозреватель решений) раскройте проект NorthwindUwp и каталог `ViewModels`, откройте файл `ShellViewModel.cs` и измените метод `PopulateNavItems`, чтобы в нем использовались несколько другие символы, как показано в следующем листинге (выделено полужирным шрифтом):

```
_primaryItems.Add(ShellNavigationItem.FromType<MainPage>("Shell_Main".GetLocalized(),
Symbol.Home));
_primaryItems.Add(ShellNavigationItem.FromType<AngularWebViewPage>("Shell_
AngularWebView".GetLocalized(), Symbol.Globe));
_primaryItems.Add(ShellNavigationItem.FromType<CategoryMasterDetailPage>("Shell_
CategoryMasterDetail".GetLocalized(), Symbol.ContactInfo));
_primaryItems.Add(ShellNavigationItem.FromType<CustomersGridPage>("Shell_
CustomersGrid".GetLocalized(), Symbol.PhoneBook));
_primaryItems.Add(ShellNavigationItem.FromType<CustomerMapPage>("Shell_
CustomerMap".GetLocalized(), Symbol.Map));
_primaryItems.Add(ShellNavigationItem.FromType<AboutNorthwindAppPage>("Shell_
AboutNorthwindApp".GetLocalized(), Symbol.Help));
```

Раскройте каталоги `Strings` и `en-us` и откройте файл `Resources.resw`.

В нижней части таблицы измените свойство Value (Значение) ресурсов, начинающихся со слова `Shell_`, чтобы эти ресурсы лучше отвечали нашим задачам (рис. 17.25).

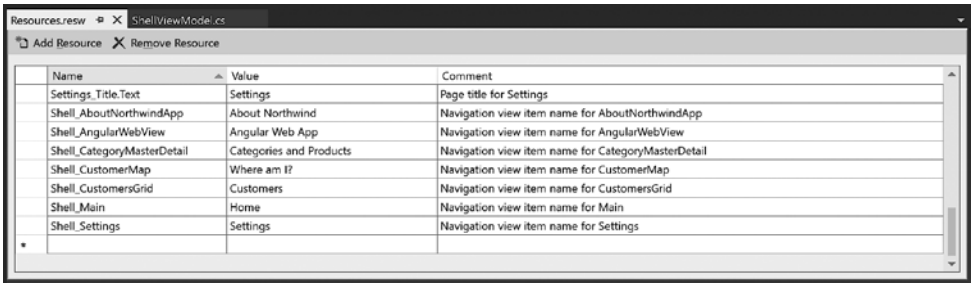


Рис. 17.25

Выполните команду Build ▸ Configuration Manager (Сборка ▸ Диспетчер конфигураций), в строке проекта NorthwindUwp установите флажки Build (Сборка) и Deploy (Развертывание) и нажмите кнопку Close (Закрыть).

Тестирование функциональности приложения

Запустите приложение, выполнив команду Debug ▸ Start (Отладка ▸ Начать отладку) или нажав клавишу F5.

После запуска приложения щелкните на меню «гамбургер» и выберите приложение Angular Web App; обратите внимание, что произойдет загрузка приложения (рис. 17.26).

Щелкните на элементе меню Where am I? (Где я?) и нажмите кнопку Yes (Да), чтобы разрешить приложению получать доступ к вашему точному местоположению (рис. 17.27).

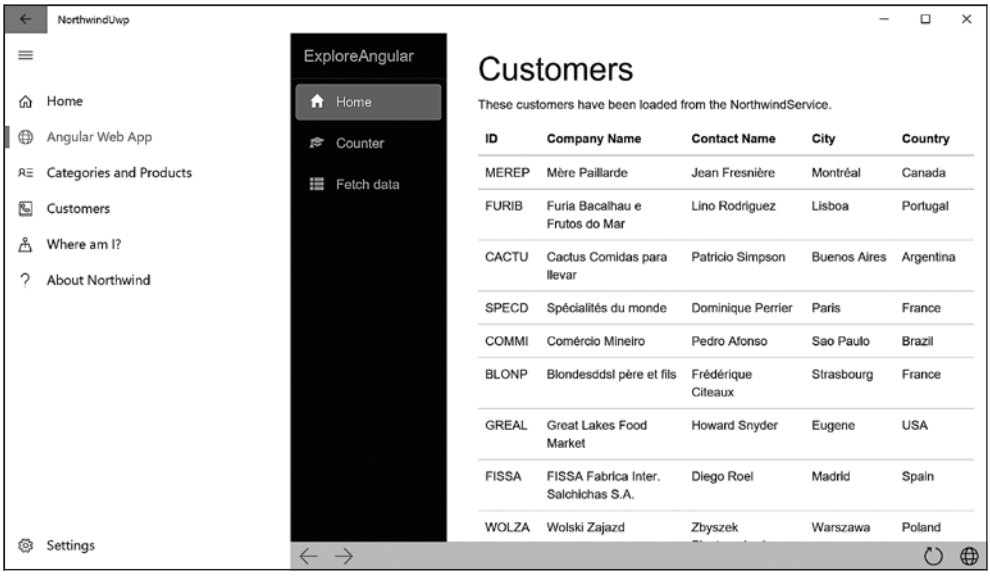


Рис. 17.26

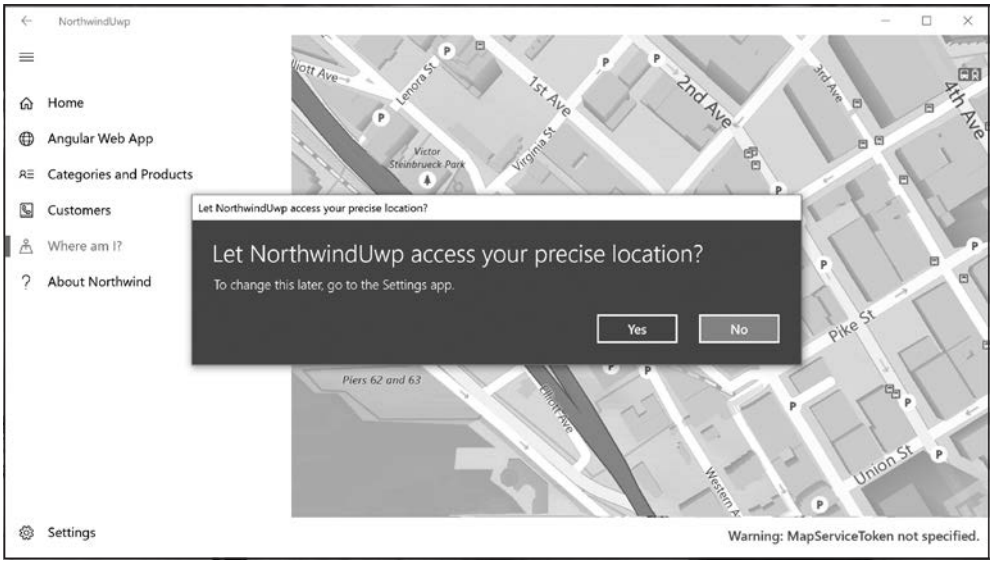


Рис. 17.27

Обратите внимание на кнопку с обратной стрелкой, возвращающую пользователя на предыдущие экраны.

Щелкните на пункте Settings (Настройки) и выберите тему Dark.

Выберите пункт Customers (Заказчики) и обратите внимание на сетку с образцами данных.

Щелкните на пункте Categories and Products (Категории и продукты) и обратите внимание на образцы данных.

Закройте приложение.

Практические задания

Посетите указанные ресурсы, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Дополнительные ресурсы

- Подготовка устройства к разработке: https://docs.microsoft.com/en-us/windows/uwp/get-started/enable-your-device-for-development.
- Введение в работу с универсальной платформой Windows: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide.
- Начало работы с UWP Community Toolkit: https://docs.microsoft.com/en-gb/windows/uwpcommunitytoolkit/Getting-Started.
- Проектирование и пользовательский интерфейс приложений UWP: https://developer.microsoft.com/en-us/windows/apps/design.
- Статьи с инструкциями для UWP-приложений в Windows 10: https://developer.microsoft.com/en-us/windows/apps/develop.

Резюме

Из этой главы вы узнали, как с помощью языка XAML и новой системы проектирования Fluent создавать графические пользовательские интерфейсы, включая такие свойства, как акриловый материал, свет и перспективный параллакс. Кроме того, узнали, как обмениваться ресурсами в приложении, заменять шаблоны элементов управления, осуществлять привязку к данным и элементам управления, а также предотвращать блокировку потоков, применяя многозадачность и ключевые слова `async` и `await` языка C#.

В следующей главе вы научитесь создавать мобильные приложения с помощью Xamarin.Forms.

18 Разработка мобильных приложений с помощью XAML и Xamarin.Forms

Эта глава посвящена проектированию на языке C# кросс-платформенных мобильных приложений для операционных систем iOS, Android и других мобильных платформ. Приложение, которое мы создадим далее, позволит просматривать список клиентов и управлять сведениями о них из базы данных Northwind.

Мобильное приложение будет обращаться к сервису Northwind, созданному с помощью веб-API ASP.NET Core в главе 16. Если данный сервис еще не готов, то вернитесь к главе 16 и разработайте его.

Клиентское мобильное приложение Xamarin.Forms мы напишем в среде разработки Visual Studio для Mac.

Для выполнения упражнений из этой главы вам понадобятся компьютер, работающий под управлением операционной системы macOS, а также среды разработки Xcode и Visual Studio для Mac.

В данной главе:

- знакомство с Xamarin и Xamarin.Forms;
- разработка мобильных приложений с помощью Xamarin.Forms.

Знакомство с Xamarin и Xamarin.Forms

Платформа *Xamarin* позволяет разработчикам создавать на языке C# мобильные приложения для операционных систем Apple iOS (iPhone и iPad), Google Android и Windows Mobile. Она основана на сторонней открытой реализации платформы .NET под названием Mono. Уровень бизнес-логики может быть написан один раз

и разделен между всеми мобильными платформами. Взаимодействие с UI и API различается на разных мобильных платформах, поэтому уровень пользовательского опыта обычно для каждой платформы свой.

Xamarin.Forms в качестве расширения Xamarin

Инструментарий *Xamarin.Forms* расширяет возможности Xamarin, упрощая кроссплатформенную мобильную разработку благодаря написанию общего кода для обеспечения требуемого пользовательского опыта, а также для реализации бизнес-логики.

Как и в приложениях для универсальной платформы Windows, в Xamarin.Forms применяется язык разметки XAML для однократного определения пользовательского интерфейса для всех платформ сразу с помощью абстрагирования компонентов данного интерфейса, специфичных для той или иной платформы. Приложения, созданные на основе Xamarin.Forms, формируют UI из нативных виджетов платформы, поэтому являются удобными и привлекательными, а также идеально подходят под целевую мобильную платформу.

Слой взаимодействия с пользователем, созданный с помощью Xamarin.Forms, никогда не будет идеально сочетаться с каждой конкретной платформой, будучи клиентской специализированной сборкой с применением Xamarin.Forms, но для корпоративных мобильных приложений этот пакет более чем хорош.

Говорим «мобильность», подразумеваем «облака»

Мобильные приложения часто взаимодействуют с облачными сервисами. Сатья Наделла (Satya Nadella), генеральный исполнительный директор Microsoft, как-то отлично выразился:

> *«Для меня стратегия mobile first означает не мобильность устройств, а мобильность индивидуального опыта. [...] Единственный способ, с помощью которого вы сможете организовать мобильность этих приложений и данных, — это облако».*

Как вы могли заметить, читая данную книгу, создать сервис веб-API ASP.NET Core для поддержки мобильного приложения можно благодаря любой среде разработки Visual Studio.

Для проектирования приложений Xamarin.Forms разработчики могут пользоваться Visual Studio 2017 или Visual Studio для Mac.

Для компиляции программ под платформу Windows Mobile потребуется операционная система Windows и среда разработки Visual Studio 2017. Для компиляции

приложений для iOS понадобится операционная система macOS, Visual Studio для Mac и Xcode. Итак, почему в этой главе я выбрал платформу Mac для разработки мобильных приложений?

Устройств под управлением операционной системы Android в разы больше по сравнению с iOS, однако учтем следующее:

- на каждый доллар, который пользователь Android тратит на приложения, пользователь iOS тратит десять долларов;
- на каждый час, который пользователь Android проводит в Интернете, у пользователя iOS приходится два часа.

Таким образом, нужно иметь в виду, что пользователи iOS гораздо больше взаимодействуют со своими устройствами; это важно при монетизации мобильных приложений, будь то их продажа, внутренние покупки или реклама.

Итоговые данные о том, какие среды разработки могут использоваться для создания и компиляции тех или иных приложений, приведены в табл. 18.1.

Таблица 18.1

	iOS	Android	Windows Mobile	Веб-API ASP.NET Core
Рынок устройств	14 %	86 %	< 0,1 %	нет данных
Visual Studio Code	Нет	Нет	Нет	Да
Visual Studio для Mac	Да	Да	Нет	Да
Visual Studio 2017	Нет	Да	Да	Да

Если вы хотите получить больше информации о Xamarin, то я рекомендую приобрести книгу Xamarin: Cross-Platform Mobile Application Development Learning Path Джонатана Пепперса (Jonathan Peppers), Джорджа Таскоса (George Taskos) и Кена Билгина (Can Bilgin). Подробнее о ней можно узнать, перейдя по ссылке https://www.packtpub.com/application-development/xamarin-crossplatform-mobile-application-development.

Разработка мобильных приложений с помощью Xamarin.Forms

Мы создадим мобильное приложение для управления списком клиентов в базе данных Northwind, которое может быть запущено на iOS либо Android.

Если вы никогда ранее не запускали Xcode, то сделайте это сейчас для подтверждения того, что все необходимые компоненты установлены и зарегистрированы.

Установка Android SDK

Для создания приложений под Android вы должны установить хотя бы один комплект средств разработки Android SDK. Установка Visual Studio для Mac по умолчанию уже включает один комплект Android SDK, однако зачастую это старая версия для поддержки наибольшего количества устройств Android. Для использования новейших возможностей Xamarin.Forms нужно установить более новую версию Android SDK.

Запустите Visual Studio для Mac и выполните команду Visual Studio Community ▸ Preferences (Visual Studio Community ▸ Настройки).

В диалоговом окне Preferences (Настройки) перейдите к разделу Projects ▸ SDK Locations (Проекты ▸ Расположение SDK) и выберите нужные платформы, например, Android 8.0 — Oreo (рис. 18.1).

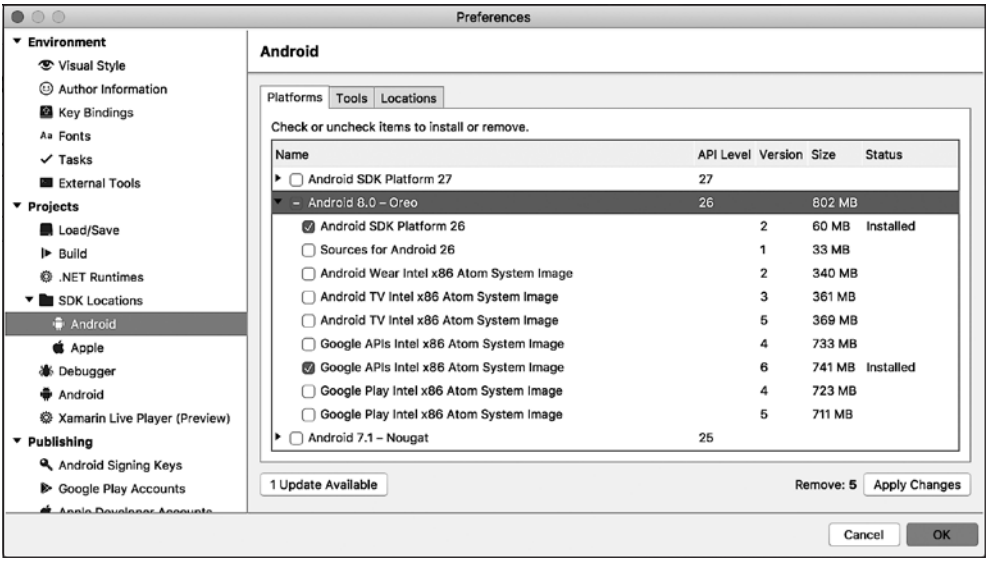


Рис. 18.1

При установке Android SDK вы должны выбрать хотя бы один образ системы (System Image), который будет использоваться как эмулятор виртуальной машины для тестирования.

Создание решения Xamarin.Forms

Выполните команду File ▸ New Solution (Файл ▸ Новое решение).

В открывшемся диалоговом окне выберите пункт App (Приложение) в категории Multiplatform (Кросс-платформенные проекты).

В разделе Xamarin.Forms выберите пункт Blank Forms App (Пустое приложение Forms) (рис. 18.2).

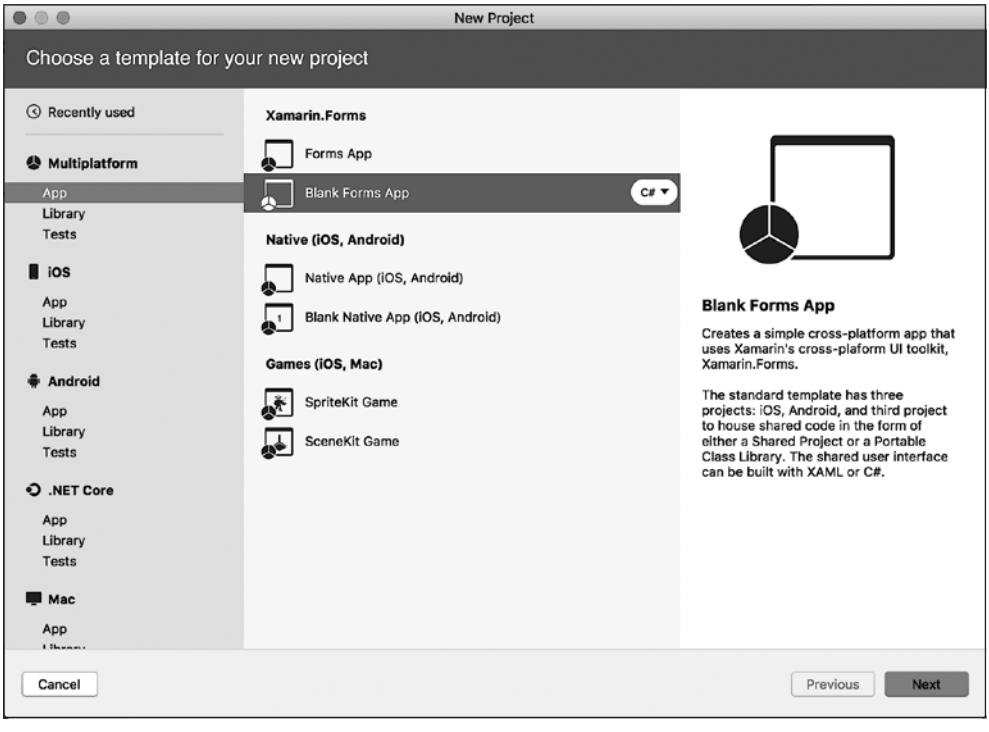


Рис. 18.2

Нажмите кнопку Next (Далее).

В поле App Name (Имя приложения) введите текст `NorthwindMobile`, а в поле Organization Identifier (Идентификатор организации) — значение `com.packt`. Установите переключатель Shared Code (Общий код) в положение Use Shared Library (Использовать общую библиотеку) и активизируйте флажок Use XAML for the user interface files (Использовать XAML для файлов пользовательского интерфейса) (рис. 18.3).

Нажмите кнопку Next (Далее).

В поле Solution name (Имя решения) укажите значение `Part3Mobile`, а в поле Location (Расположение) — значение `/Users/ваше_имя/Code` (рис. 18.4).

Нажмите кнопку Create (Создать).

Через несколько мгновений будут созданы решение и три проекта. В Visual Studio для Mac выполните команду Build ▸ Build All (Сборка ▸ Собрать все) и дождитесь, когда программа загрузит все обновленные пакеты и соберет проекты (рис. 18.5).

Щелкните правой кнопкой мыши на решении Part3Mobile и выберите пункт Update NuGet Packages (Обновить пакеты NuGet).

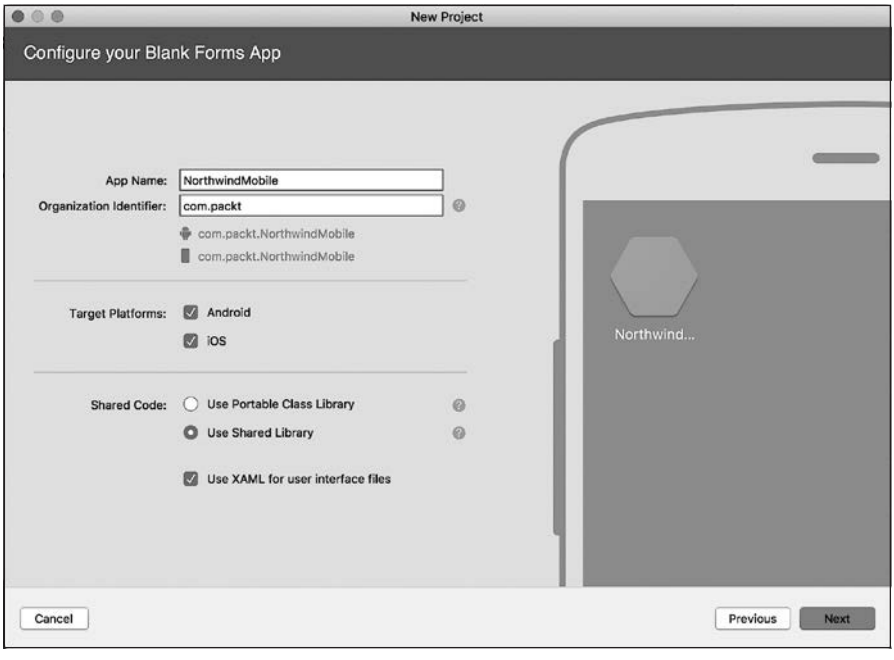


Рис. 18.3

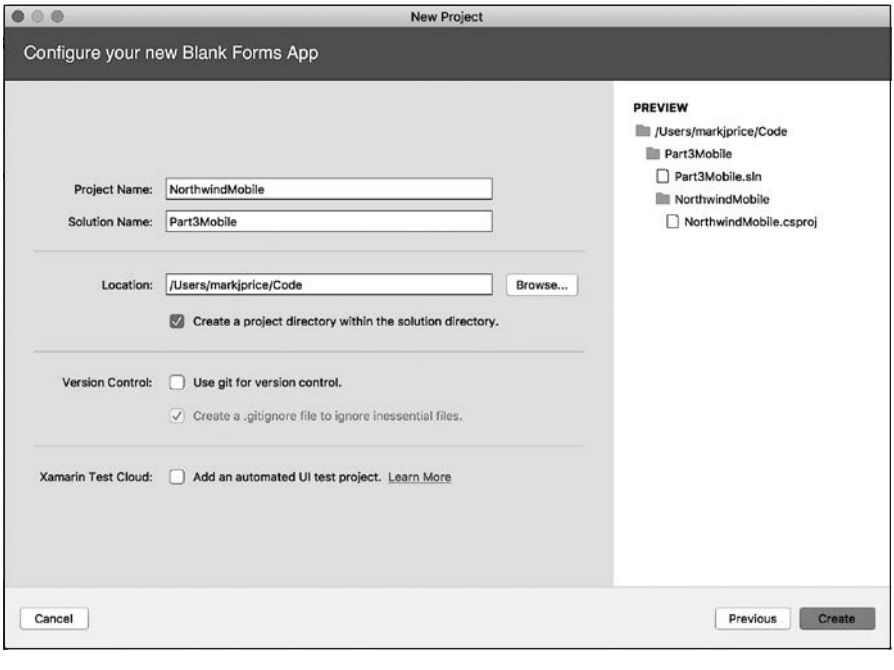


Рис. 18.4

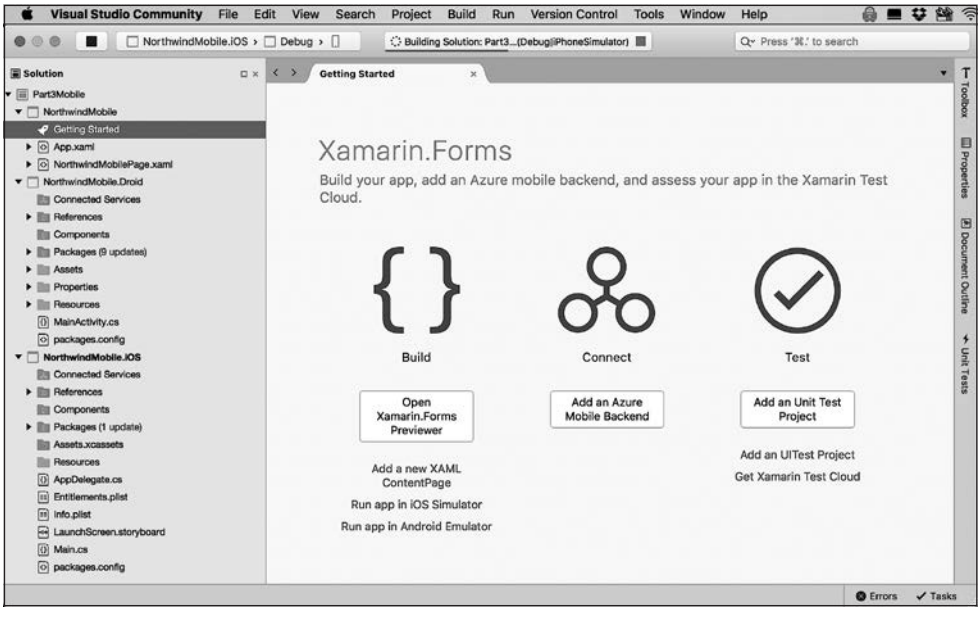


Рис. 18.5

Создание модели

Мы могли бы воспользоваться созданной ранее библиотекой моделей данных с сущностями .NET Standard 2.0, но нам нужно реализовать двустороннюю привязку данных, поэтому мы создадим новый класс для представления сущностей клиентов в мобильном приложении.

Щелкните правой кнопкой мыши на проекте NorthwindMobile, в контекстном меню выполните команду Add ▸ New Folder (Добавить ▸ Новая папка) и присвойте созданному каталогу имя `Models`.

Щелкните правой кнопкой мыши на каталоге `Models` и в контекстном меню выполните команду Add ▸ New File (Добавить ▸ Новый файл).

В диалоговом окне New File (Новый файл) выполните команду General ▸ Empty Class (General ▸ Пустой класс), присвойте классу имя `Customer` (рис. 18.6) и нажмите кнопку New (Новый).

Измените инструкции, как показано в листинге ниже:

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;

namespace NorthwindMobile.Models
{
  public class Customer : INotifyPropertyChanged
  {
```

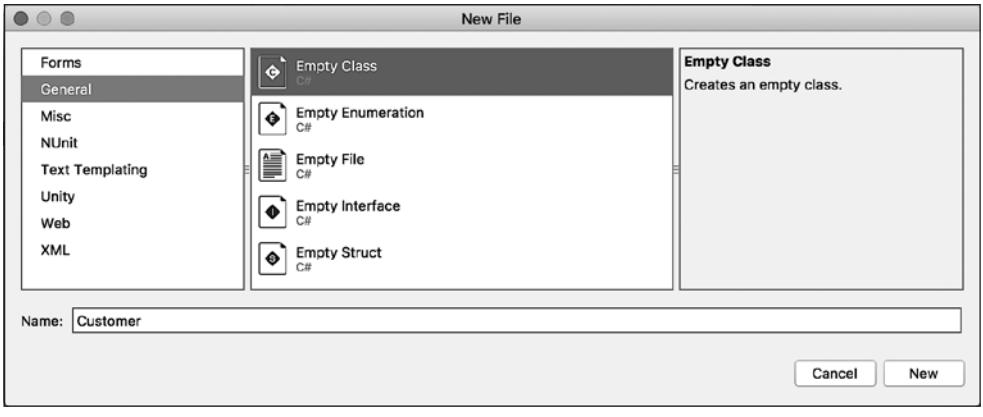


Рис. 18.6

```
public static IList<Customer> Customers;

 static Customer()
 {
   Customers = new ObservableCollection<Customer>();
 }

 public event PropertyChangedEventHandler PropertyChanged;

 private string customerID;
 private string companyName;
 private string contactName;
 private string city;
 private string country;
 private string phone;

 public string CustomerID
 {
   get { return customerID; }
   set
   {
     customerID = value;
     PropertyChanged?.Invoke(this,
     new PropertyChangedEventArgs("CustomerID"));
   }
 }

 public string CompanyName
 {
   get { return companyName; }
   set
   {
     companyName = value;
     PropertyChanged?.Invoke(this,
```

```
    new PropertyChangedEventArgs("CompanyName"));
  }
}

public string ContactName
{
  get { return contactName; }
  set
  {
    contactName = value;
    PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs("ContactName"));
  }
}
public string City
{
  get { return city; }
  set
  {
    city = value;
    PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs("City"));
  }
}

public string Country
{
  get { return country; }
  set
  {
    country = value;
    PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs("Country"));
  }
}

public string Phone
{
  get { return phone; }
  set
  {
    phone = value;
    PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs("Phone"));
  }
}

public string Location
{
  get
  {
    return string.Format("{0}, {1}", City, Country);
```

```
      }
    }

    // для тестирования перед вызовом веб-сервиса
    public static void SampleData()
    {

      Customers.Clear();

      Customers.Add(new Customer
      {
        CustomerID = "ALFKI",
        CompanyName = "Alfreds Futterkiste",
        ContactName = "Maria Anders",
        City = "Berlin",
        Country = "Germany",
        Phone = "030-0074321"
      });

      Customers.Add(new Customer
      {
        CustomerID = "FRANK",
        CompanyName = "Frankenversand",
        ContactName = "Peter Franken",
        City = "München",
        Country = "Germany",
        Phone = "089-0877310"
      });

      Customers.Add(new Customer
      {
        CustomerID = "SEVES",
        CompanyName = "Seven Seas Imports",
        ContactName = "Hari Kumar",
        City = "London",
        Country = "UK",
        Phone = "(171) 555-1717"
      });
    }
  }
}
```

Обратите внимание на следующие моменты.

- Класс реализует `INotifyPropertyChanged`, поэтому компоненты пользовательского интерфейса с двусторонним связыванием, такие как `Editor`, будут обновлять свойство и наоборот. Все держится на событии `PropertyChanged`, которое возникает при изменении одного из свойств.
- После загрузки из сервиса данные о клиентах будут локально кэшироваться в мобильном приложении с помощью `ObservableCollection`. Это обеспечивает поддержку уведомлений для любых связанных компонентов пользовательского интерфейса, таких как `ListView`.

- Помимо свойств для сохранения значений, полученных из REST-сервиса, класс определяет свойство `Location`, установленное только для чтения. Оно будет служить для привязки в сводном списке клиентов.
- В целях тестирования, когда REST-сервис недоступен, применяется метод для использования данных трех демоклиентов.

Создание интерфейса для набора телефонных номеров

Щелкните правой кнопкой мыши на каталоге `NorthwindMobile` и в контекстном меню выполните команду Add ▸ New File (Добавить ▸ Новый файл).

В диалоговом окне New File (Новый файл) выполните команду General ▸ Empty Interface (General ▸ Пустой интерфейс), присвойте интерфейсу имя `IDialer` и нажмите кнопку New (Новый).

Измените код интерфейса `IDialer`, как показано в следующем листинге:

```
namespace NorthwindMobile
{
  public interface IDialer
  {
    bool Dial(string number);
  }
}
```

Реализация интерфейса для набора телефонных номеров под iOS

Щелкните правой кнопкой мыши на каталоге `NorthwindMobile.iOS` и в контекстном меню выполните команду Add ▸ New File (Добавить ▸ Новый файл).

В диалоговом окне New File (Новый файл) выполните команду General ▸ Empty Class (General ▸ Пустой класс), присвойте классу имя `PhoneDialer` и нажмите кнопку New (Новый).

Измените его содержимое, как показано в листинге, приведенном ниже:

```
using Foundation;
using NorthwindMobile.iOS;
using UIKit;
using Xamarin.Forms;

[assembly: Dependency(typeof(PhoneDialer))]
namespace NorthwindMobile.iOS
{
  public class PhoneDialer : IDialer
  {
    public bool Dial(string number)
    {
      return UIApplication.SharedApplication.OpenUrl(
      new NSUrl("tel:" + number));
    }
  }
}
```

Реализация интерфейса для набора телефонных номеров под Android

Щелкните правой кнопкой мыши на каталоге `NorthwindMobile.Droid` в контекстном меню и выполните команду Add ▸ New File (Добавить ▸ Новый файл).

В диалоговом окне New File (Новый файл) выполните команду General ▸ Empty Class (General ▸ Пустой класс), присвойте классу имя `PhoneDialer` и нажмите кнопку New (Новый).

Измените его содержимое, как показано в следующем листинге:

```
using Android.Content;
using Android.Telephony;
using NorthwindMobile.Droid;
using System.Linq;
using Xamarin.Forms;
using Uri = Android.Net.Uri;

[assembly: Dependency(typeof(PhoneDialer))]
namespace NorthwindMobile.Droid
{
  public class PhoneDialer : IDialer
  {
    public bool Dial(string number)
    {
      var context = Forms.Context;
      if (context == null)
      return false;

      var intent = new Intent(Intent.ActionCall);
      intent.SetData(Uri.Parse("tel:" + number));

      if (IsIntentAvailable(context, intent))
      {
        context.StartActivity(intent);
        return true;
      }

      return false;
    }

    public static bool IsIntentAvailable(Context context, Intent intent)
    {
      var packageManager = context.PackageManager;

      var list = packageManager.QueryIntentServices(intent, 0)
      .Union(packageManager.QueryIntentActivities(intent, 0));

      if (list.Any())
      return true;
      var manager = TelephonyManager.FromContext(context);
      return manager.PhoneType != PhoneType.None;
    }
  }
}
```

В каталоге `NorthwindMobile.Droid` раскройте подкаталог `Properties` и откройте файл `AndroidManifest.xml`.

В области Required permissions (Требуемые разрешения) установите флажок CallPhone (рис. 18.7).

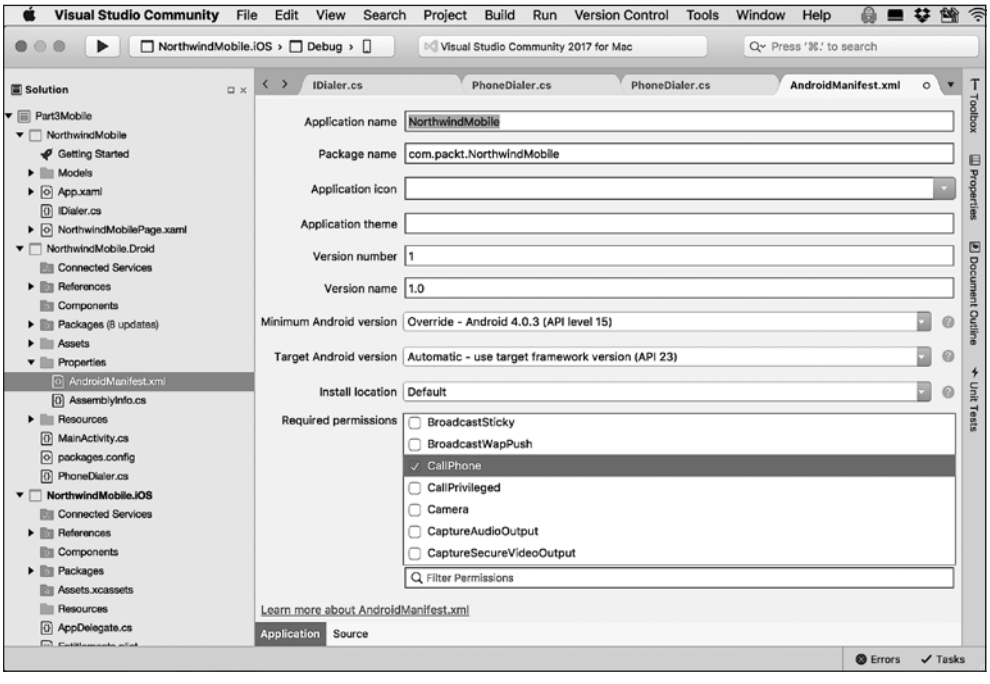


Рис. 18.7

Создание представлений для списка клиентов и подробной информации о клиенте

Щелкните правой кнопкой мыши на файле `NorthwindMobilePage.xaml`, нажмите кнопку Remove (Удалить), а затем нажмите кнопку Remove from Project (Удалить из проекта).

Щелкните правой кнопкой мыши на проекте NorthwindMobile, в контекстном меню выполните команду Add ▸ New Folder (Добавить ▸ Новая папка) и присвойте созданному каталогу имя `Views`.

Щелкните правой кнопкой мыши на каталоге `Views` и в контекстном меню выполните команду Add ▸ New File (Добавить ▸ Новый файл).

В диалоговом окне New File (Новый файл) выполните команду Forms ▸ Forms ContentPage Xaml (Формы ▸ Forms ContentPage Xaml), присвойте файлу имя `CustomersList` и нажмите кнопку New (Новый) (рис. 18.8).

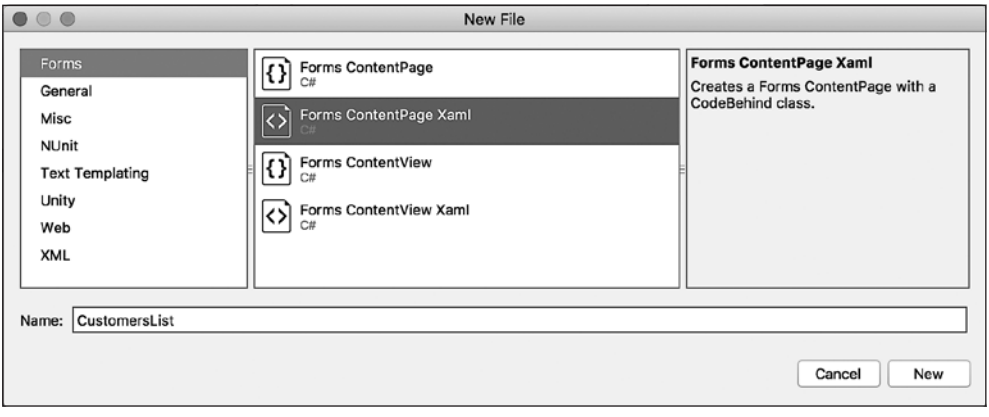


Рис. 18.8

Создание представления для списка клиентов

В каталоге `NorthwindMobile` найдите и откройте файл `CustomersList.xaml` и измените его содержимое, как показано в листинге ниже:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
 xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
 x:Class="NorthwindMobile.Views.CustomersList"
 Title="List">
  <ContentPage.Content>
    <ListView ItemsSource="{Binding .}"
     VerticalOptions="Center" HorizontalOptions="Center"
     IsPullToRefreshEnabled="True"
     ItemTapped="Customer_Tapped"
     Refreshing="Customers_Refreshing">
      <ListView.Header>
        <Label Text="Northwind Customers"
         BackgroundColor="Silver" />
     </ListView.Header>
     <ListView.ItemTemplate>
      <DataTemplate>
        <TextCell Text="{Binding CompanyName}"
         Detail="{Binding Location}">
           <TextCell.ContextActions>
             <MenuItem Clicked="Customer_Phoned" Text="Phone" />
             <MenuItem Clicked="Customer_Deleted"
               Text="Delete" IsDestructive="True" />
           </TextCell.ContextActions>
         </TextCell>
       </DataTemplate>
     </ListView.ItemTemplate>
    </ListView>
  </ContentPage.Content>
```

```
  <ContentPage.ToolbarItems>
    <ToolbarItem Text="Add" Activated="Add_Activated"
     Order="Primary" Priority="0" />
  </ContentPage.ToolbarItems>
</ContentPage>
```

Обратите внимание на следующие моменты.

- У элемента `ContentPage` есть атрибут `Title` со значением `List`.
- Были написаны обработчики таких событий: загрузка списка клиентов при появлении представления, выбор (касанием) клиента пользователем (для загрузки информации о клиенте), смахивание вниз (обновление списка), удаление сведений о клиенте смахиванием влево и касанием кнопки Delete (Удалить).
- Шаблон данных определяет способ отображения сведений о каждом клиенте: крупный шрифт для названия компании и мелкий — для адреса компании.
- Доступна кнопка Add (Добавить), позволяющая пользователям перейти к подробному представлению при добавлении нового клиента.

Измените содержимое файла `CustomersList.xaml.cs`, как показано в коде, приведенном ниже:

```
using System;
using System.Threading.Tasks;
using NorthwindMobile.Models;
using Xamarin.Forms;
namespace NorthwindMobileApp.Views
{
  public partial class CustomersList : ContentPage
  {
    public CustomersList()
    {
      InitializeComponent();
      Customer.SampleData();
      BindingContext = Customer.Customers;
    }

    async void Customer_Tapped(object sender, ItemTappedEventArgs e)
    {
      Customer c = e.Item as Customer;
      if (c == null) return;
      // переход к подробному представлению и вывод сведений
      // о выбранном клиенте
      await Navigation.PushAsync(new CustomerDetails(c));
    }

    async void Customers_Refreshing(object sender, EventArgs e)
    {
      ListView listView = sender as ListView;
      listView.IsRefreshing = true;
```

```
      // имитация обновления
      await Task.Delay(1500);
      listView.IsRefreshing = false;
    }

    void Customer_Deleted(object sender, EventArgs e)
    {
      MenuItem menuItem = sender as MenuItem;
      Customer c = menuItem.BindingContext as Customer;
      Customer.Customers.Remove(c);
    }

    async void Customer_Phoned(object sender, EventArgs e)
    {
      MenuItem menuItem = sender as MenuItem;
      Customer c = menuItem.BindingContext as Customer;
      if (await this.DisplayAlert("Dial a Number",
          "Would you like to call " + c.Phone + "?",
          "Yes", "No"))
      {
        var dialer = DependencyService.Get<IDialer>();
        if (dialer != null)
        dialer.Dial(c.Phone);
      }
    }

    async void Add_Activated(object sender, EventArgs e)
    {
      await Navigation.PushAsync(new CustomerDetails());
    }
  }
}
```

Обратите внимание на следующие аспекты.

- Элемент `BindingContext` установлен на список-образец `Customers` в конструкторе страницы.
- При касании названия клиента в списочном представлении пользователь попадает в подробное представление (которое вы создадите на следующем шаге).
- Оттягивание списочного представления вниз запускает симулированное обновление, занимающее полторы секунды.
- При удалении из списочного представления клиент также удаляется из привязанной коллекции клиентов.
- При горизонтальном проведении пальцем по информации о клиенте и касании кнопки Phone (Позвонить) программа спрашивает пользователя, следует ли набрать телефонный номер клиента. При положительном ответе она получит реализацию этой функции в зависимости от применяемой платформы, после чего будет задействован механизм распознания зависимостей для набора телефонного номера клиента.
- Нажимая кнопку Add (Добавить), пользователь попадает на страницу информации, где может заполнить необходимые данные о новом клиенте.

Создание представления для подробной информации о клиенте

Добавьте еще один файл XAML ContentPage Forms, присвоив ему имя `CustomerDetails`.

Откройте файл `CustomerDetails.xaml` и измените его содержимое, как показано в листинге ниже. Обратите внимание на следующие аспекты:

- значение `Title` элемента `ContentPage` установлено в режим Edit Customer;
- элемент `Grid` с двумя столбцами и шестью строками используется для разметки макета;
- элементы `Editor` имеют двустороннюю привязку к свойствам класса `Customer`.

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
 xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
 x:Class="NorthwindMobile.Views.CustomerDetails" Title="Edit Customer">
  <ContentPage.Content>
    <StackLayout VerticalOptions="Fill"
     HorizontalOptions="Fill">
    <Grid BackgroundColor="Silver">
      <Grid.ColumnDefinitions>
        <ColumnDefinition/> <ColumnDefinition/>
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition/> <RowDefinition/> <RowDefinition/>
        <RowDefinition/> <RowDefinition/> <RowDefinition/>
      </Grid.RowDefinitions>
      <Label Text="Customer ID"
       VerticalOptions="Center" Margin="6" />
      <Editor Text="{Binding CustomerID, Mode=TwoWay}"
       Grid.Column="1" />
      <Label Text="Company Name" Grid.Row="1"
       VerticalOptions="Center" Margin="6" />
      <Editor Text="{Binding CompanyName, Mode=TwoWay}"
       Grid.Column="1" Grid.Row="1" />
      <Label Text="Contact Name" Grid.Row="2"
       VerticalOptions="Center" Margin="6" />
      <Editor Text="{Binding ContactName, Mode=TwoWay}"
       Grid.Column="1" Grid.Row="2" />
      <Label Text="City" Grid.Row="3"
       VerticalOptions="Center" Margin="6" />
      <Editor Text="{Binding City, Mode=TwoWay}"
       Grid.Column="1" Grid.Row="3" />
      <Label Text="Country" Grid.Row="4"
       VerticalOptions="Center" Margin="6" />
      <Editor Text="{Binding Country, Mode=TwoWay}"
       Grid.Column="1" Grid.Row="4" />
      <Label Text="Phone" Grid.Row="5"
       VerticalOptions="Center" Margin="6" />
      <Editor Text="{Binding Phone, Mode=TwoWay}"
       Grid.Column="1" Grid.Row="5" />
    </Grid>
    <Button x:Name="InsertButton" Text="Insert Customer"
     Clicked="InsertButton_Clicked" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

Откройте файл CustomerDetail.xaml.cs и измените его содержимое, как показано в следующем листинге:

```
using System;
using NorthwindMobile.Models;
using Xamarin.Forms;

namespace NorthwindMobile.Views
{
  public partial class CustomerDetails : ContentPage
  {
    private bool newCustomer = false;

    public CustomerDetails()
    {
      InitializeComponent();
      BindingContext = new Customer();
      newCustomer = true;
      Title = "Add Customer";
    }

    public CustomerDetails(Customer customer)
    {
      InitializeComponent();
      BindingContext = customer;
      InsertButton.IsVisible = false;
    }

    async void InsertButton_Clicked(object sender, EventArgs e)
    {
      if (newCustomer)
      {
        Customer.Customers.Add((Customer)BindingContext);
      }
      await Navigation.PopAsync(animated: true);
    }
  }
}
```

Откройте файл App.xaml.cs.

Импортируйте пространство имен NorthwindMobile.Views.

Измените инструкцию, которая задает MainPage, чтобы создавался экземпляр CustomersList, обернутый в NavigationPage, как показано в этом коде:

```
MainPage = new NavigationPage(new CustomersList());
```

Тестирование мобильного приложения в среде iOS

Щелкните на значке смартфона на панели инструментов Debug (Отладка) и выберите пункт iPhone X iOS 11.0 (рис. 18.9).

Нажмите кнопку Start (Запуск) на панели управления или выполните команду Run ▸ Start Debugging (Запуск ▸ Начать отладку).

Через несколько секунд вы увидите окно, имитирующее работу вашего мобильного приложения (рис. 18.10).

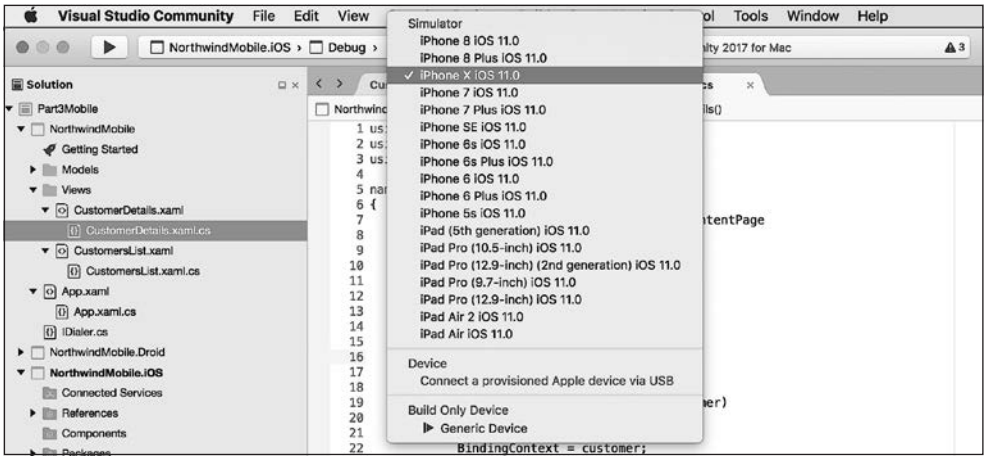


Рис. 18.9

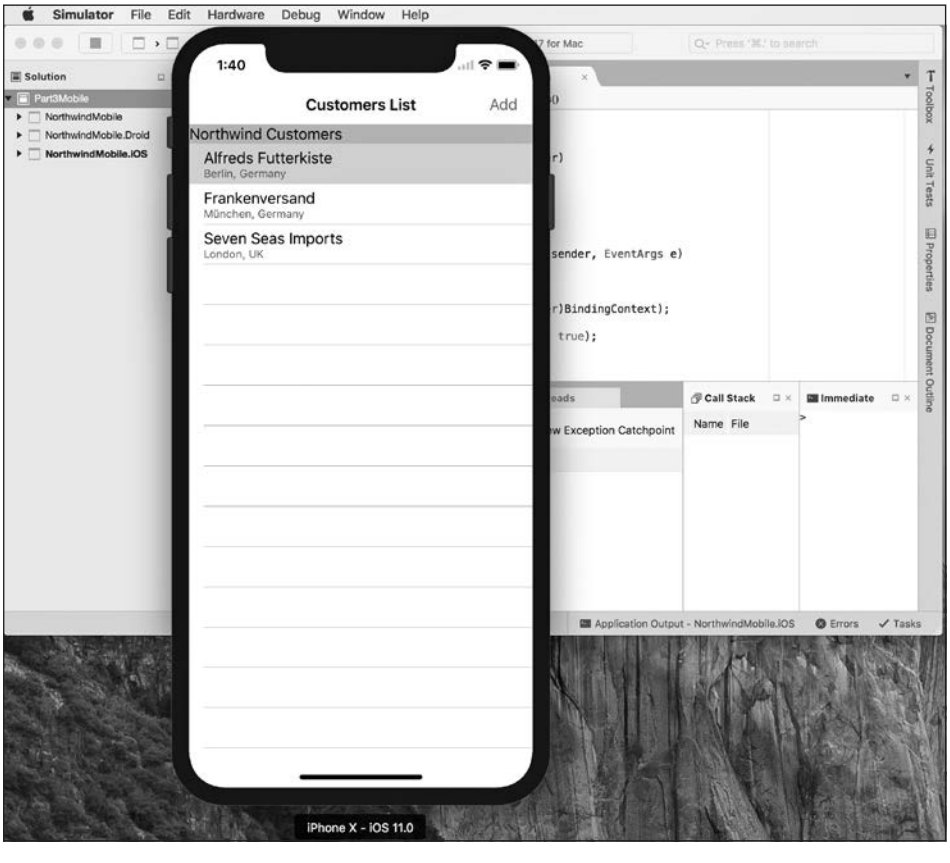


Рис. 18.10

Щелкните на строке любого клиента и измените название компании (рис. 18.11).

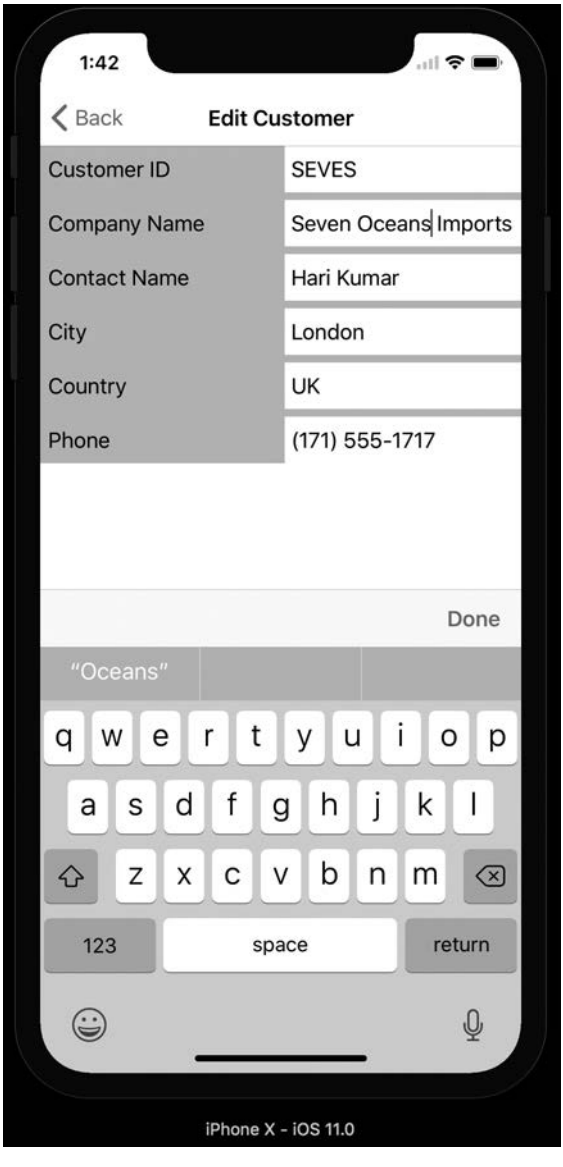


Рис. 18.11

Щелкните на ссылке Back (Назад), чтобы вернуться к списку клиентов. Обратите внимание на то, что название компании изменилось.

Щелкните на ссылке Add (Добавить).

Заполните поля ввода для добавления записи о новом клиенте (рис. 18.12).

Рис. 18.12

Щелкните на ссылке Insert Customer (Вставить клиента) и обратите внимание на то, что новый клиент был внесен в список.

Смахните строку с именем одного из клиентов влево, чтобы открыть кнопки двух действий, Phone и Delete (Удалить) (рис. 18.13).

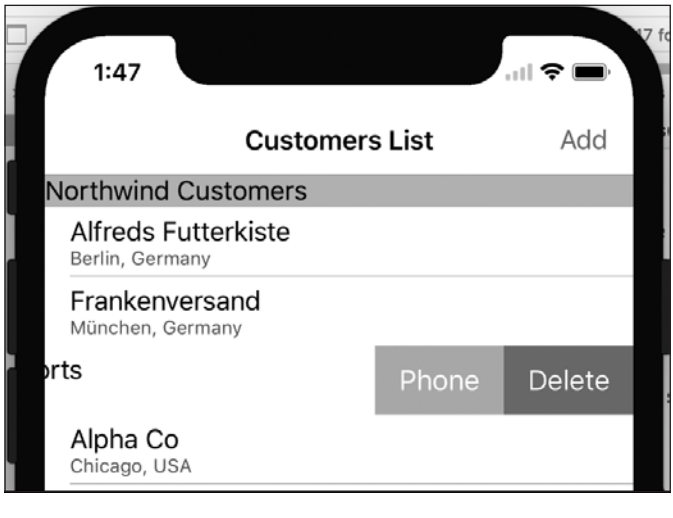


Рис. 18.13

Щелкните на ссылке Phone и обратите внимание на появившееся уведомление с подтверждением телефонного вызова клиента (рис. 18.14).

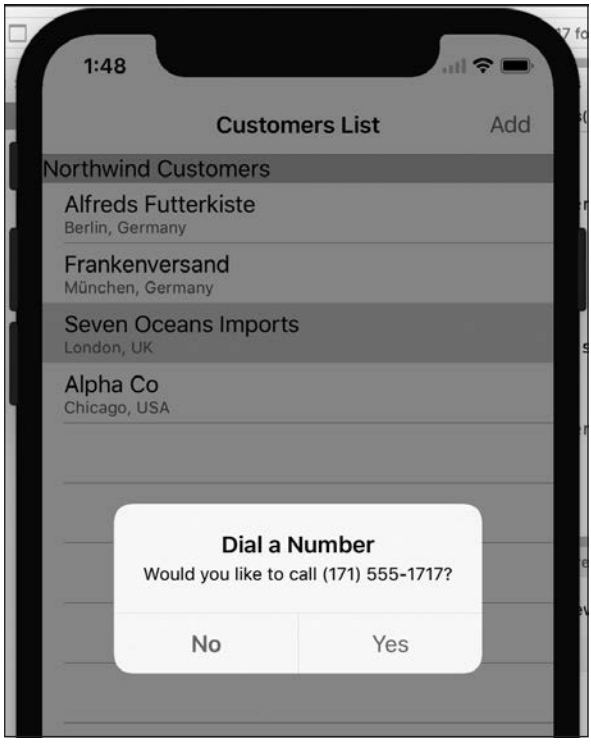


Рис. 18.14

Смахните строку с именем одного из клиентов влево, чтобы открыть кнопки двух действий, Phone и Delete (Удалить). Щелкните на ссылке Delete (Удалить) и обратите внимание на то, что сведения о клиенте были удалены.

Завершите работу симулятора.

Добавление NuGet-пакетов для вызова REST-сервиса

В проекте NorthwindMobile.iOS щелкните правой кнопкой мыши на каталоге `Packages` (Пакеты) и в контекстном меню выберите пункт Add Packages (Добавить пакеты).

В диалоговом окне Add Packages (Добавить пакеты) введите запрос `http` в поле поиска. Выберите пакет System.Net.Http и нажмите кнопку Add Package (Добавить пакет) (рис. 18.15).

В диалоговом окне с лицензионным соглашением нажмите кнопку Accept (Принимаю).

В проекте NorthwindMobile.iOS щелкните правой кнопкой мыши на каталоге `Packages` (Пакеты) и в контекстном меню выберите пункт Add Packages (Добавить пакеты).

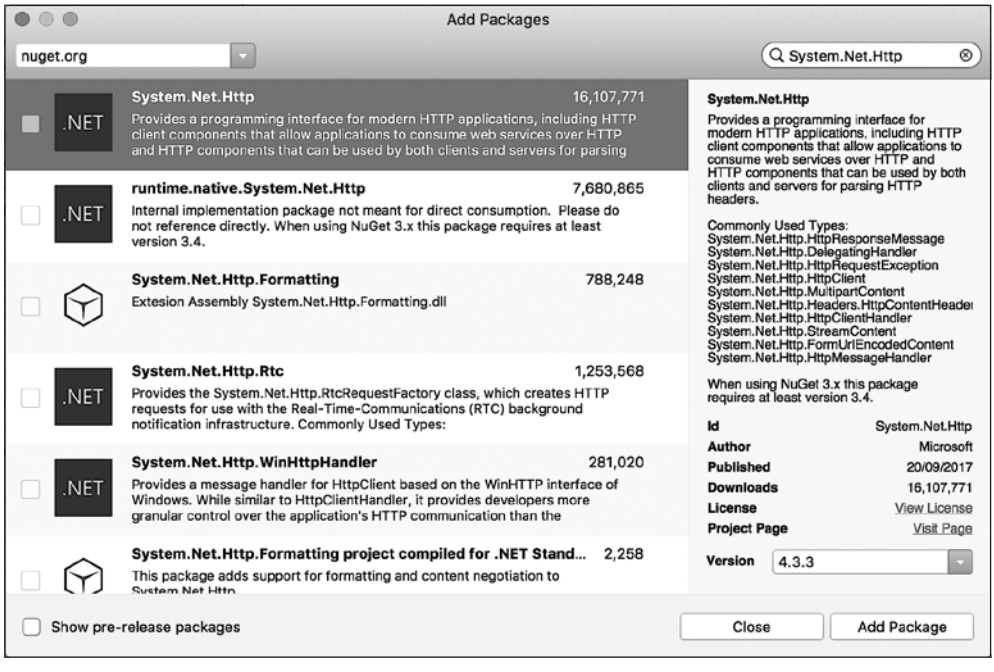


Рис. 18.15

В диалоговом окне Add Packages (Добавить пакеты) введите запрос `Json.NET` в поле поиска. Выберите пакет Json.NET и нажмите кнопку Add Package (Добавить пакет) (рис. 18.16).

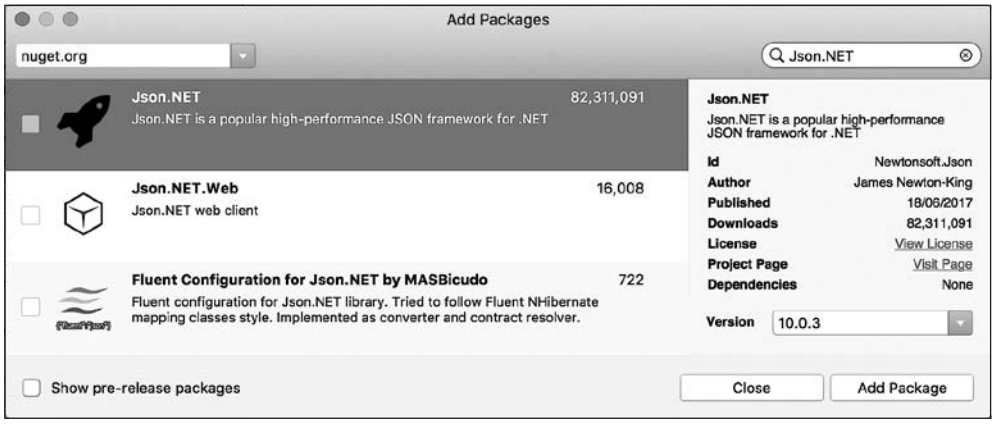


Рис. 18.16

Примите условия лицензионного соглашения.

Повторите указанные шаги, чтобы добавить эти же два пакета NuGet в проект NorthwindMobile.Android.

Получение данных о клиентах с помощью сервиса

Откройте файл `CustomersList.xaml.cs` и импортируйте следующие пространства имен (строки, которые выделены полужирным шрифтом):

```
using System.Threading.Tasks;
using NorthwindMobile.Models;
using Xamarin.Forms;
using System;
using System.Linq;
using System.Collections.Generic;
using System.Net.Http;
using System.Net.Http.Headers;
using Newtonsoft.Json;
```

Измените код конструктора `CustomersList` так, чтобы загружать список клиентов через сервис прокси-сервера вместо метода `SampleData`, как показано в следующем листинге:

```
public CustomersList()
{
  InitializeComponent();

  //Customer.SampleData();

  var client = new HttpClient();
  client.BaseAddress = new Uri(
    "http://localhost:5001/api/customers");

  client.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/json"));

  HttpResponseMessage response = client.GetAsync("").Result;

  response.EnsureSuccessStatusCode();

  string content =
    response.Content.ReadAsStringAsync().Result;
  var customersFromService = JsonConvert.DeserializeObject
  <IEnumerable<Customer>>(content);

  foreach (Customer c in customersFromService
    .OrderBy(customer => customer.CompanyName)
  {
    Customer.Customers.Add(c);
  }

  BindingContext = Customer.Customers;
}
```

В Visual Studio Code запустите проект NorthwindService.

В Visual Studio для Mac запустите проект NorthwindMobile и обратите внимание на то, что сведения о 91 клиенте загружены с помощью веб-сервиса (рис. 18.17).

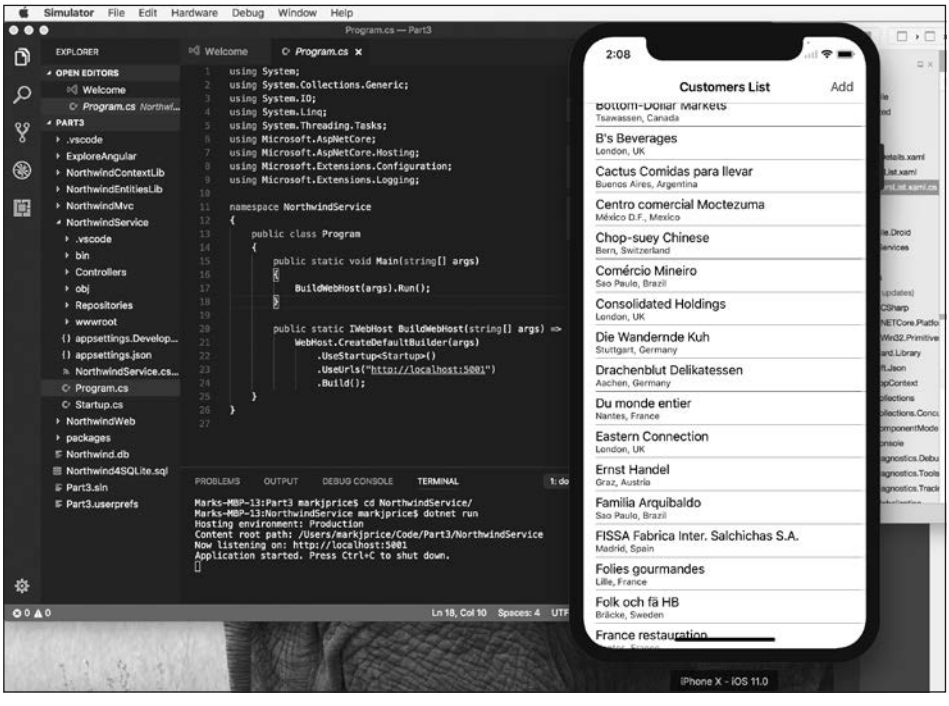


Рис. 18.17

Практические задания

Посетите указанные ресурсы, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Дополнительные ресурсы

- Visual Studio Code для разработчиков на Mac: https://channel9.msdn.com/Series/Visual-Studio-Code-for-Mac-Developers.
- Xamarin.Forms: https://www.xamarin.com/forms.
- Центр разработки Xamarin: https://developer.xamarin.com.

Резюме

В этой главе вы узнали, как разрабатывать кросс-платформенные мобильные приложения для мобильных устройств iOS и Android (и потенциально для других платформ) с помощью инструмента Xamarin.Forms и применять сервис REST/HTTP, задействуя пакеты `HttpClient` и `Newtonsoft.Json`.

Заключение

Хочется верить, что чтение этой книги о языке C# 7.1 и интерфейсах программирования .NET Core 2.0 вдохновило вас на размышления о том, как с помощью C# и .NET Core создавать собственные современные приложения с хорошей архитектурой и запускать их кросс-платформенно на Windows, macOS и Linux.

Используйте ссылки на источники, которые мы помещаем в конце каждой главы, чтобы углубить ваши знания по темам, изученным при чтении книги. Ниже приведен ряд дополнительных источников, способных помочь в процессе дальнейшего обучения.

- Переход на .NET Core в Red Hat Enterprise Linux: https://developers.redhat.com/promotions/dot-net-core/.
- Документация по функциям Azure: https://docs.microsoft.com/en-us/azure/.
- .NET на платформе Google Cloud: https://cloud.google.com/dotnet/.
- Университет Xamarin: https://www.xamarin.com/university.

Имея в арсенале инструменты и технологии C# и .NET Core, вы можете завоевать вселенную кросс-платформенной разработки и создавать приложения любого типа в соответствии с вашими потребностями.

Приложение. Ответы на проверочные вопросы

Глава 1. Привет, C#! Здравствуй, .NET Core!

1. Почему на платформе .NET Core для разработки приложений программисты могут использовать разные языки, например C# и F#?

   Ответ. Платформа .NET Core поддерживает разные языки, поскольку для каждого из них есть компилятор, преобразующий исходный код в промежуточный язык (IL). Затем этот IL-код преобразуется в машинные инструкции для конкретного центрального процессора во время выполнения с помощью общеязыковой исполняющей среды (CLR).

2. Какие команды нужно ввести в окне консоли для сборки и запуска исходного кода на языке C#?

   Ответ. Используя встроенные в платформу .NET Core инструменты интерфейса командной строки (CLI) и перейдя в каталог, содержащий файл `ИмяПроекта.csproj`, нужно выполнить команду `dotnet run`.

3. Какое сочетание клавиш разработчики применяют в режиме Visual C# для сохранения, компиляции и запуска приложения без отладки?

   Ответ. Ctrl+F5.

4. С помощью какого сочетания клавиш в Visual Studio Code открывается панель Integrated Terminal (Интегрированный терминал)?

   Ответ. Ctrl+` (обратная засечка).

5. Среда разработки Visual Studio 2017 лучше, чем Visual Studio Code?

   Ответ. Нет. Каждая из них оптимизирована под разные задачи. Visual Studio 2017 — мощная, многофункциональная и позволяет создавать программы с графическим пользовательским интерфейсом, к примеру приложения WPF,

UWP и Xamarin, но доступна только для операционной системы Windows. Visual Studio Code — компактна, оптимизирована, ориентирована на набор кода в командной строке и доступна для разных операционных систем.

6. Платформа .NET Core эффективнее .NET Framework?

   Ответ. Зависит от того, что вам нужно. .NET Core — это сокращенная кросс-платформенная версия полнофункционального «старичка» .NET Framework.

7. Чем .NET Native отличается от .NET Core?

   Ответ. .NET Native — это AoT-компилятор, создающий сборки машинного кода, обеспечивая лучшую производительность и затрачивая меньше ресурсов памяти, а также использующий статические сборки .NET, позволяя избавиться от зависимости от CoreCLR.

8. Что такое .NET Standard и почему эта технология так важна?

   Ответ. .NET Standard — это API, реализуемый платформой .NET. Текущие версии .NET Framework, .NET Core и Xamarin реализуют .NET Standard 2.0 с целью предоставить единый стандартный API, который разработчики смогут изучить и настроить.

9. В чем заключается разница между Git и GitHub?

   Ответ. Git — это распределенная система управления версиями исходного кода. GitHub — популярный веб-сервис, реализующий доступ к системе Git.

10. Как называется метод точки входа консольного приложения .NET и как его объявить?

    Ответ. Он называется `Main`, а ниже указан код его объявления. Рекомендуется использовать массив `string` для аргументов командной строки и тип возвращаемого значения `int`, хоть это и не обязательно.

```
public static void Main() // минимум
public static int Main(string[] args) // рекомендуется
```

Глава 2. Говорим на языке C#

Какой тип нужно выбрать для каждого указанного ниже числа?

1. Телефонный номер.

   Ответ. Тип `string`.

2. Рост.

   Ответ. Тип `float` или `double`.

3. Возраст.

   Ответ. Тип `int` для быстродействия или `byte` (0–255) для размера.

4. Размер оклада.

   Ответ. Тип `decimal`.

5. ISBN книги.

   Ответ. Тип `string`.

6. Цена книги.

   Ответ. Тип `decimal`.

7. Вес книги.

   Ответ. Тип `float` или `double`.

8. Размер населения страны.

   Ответ. Тип `uint` (от 0 до примерно 4 миллиардов).

9. Количество звезд во Вселенной.

   Ответ. Тип `ulong` (от 0 до примерно 18 квадриллионов) или `System.Numerics.BigInteger` (допускает произвольно большие целые числа).

10. Количество сотрудников на каждом из предприятий малого и среднего бизнеса (примерно до 50 000 сотрудников на каждом предприятии).

    Ответ. Поскольку существуют сотни тысяч малых и средних предприятий, необходимо использовать размер памяти в качестве определяющего фактора, поэтому выберите `ushort`, так как данный тип занимает всего 2 байта в отличие от `int`, занимающего 4 байта.

Глава 3. Управление потоком выполнения и преобразование типов

1. Где нужно искать справку по ключевому слову языка C#?

   Ответ. На сайте https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/.

2. Где следует искать решения распространенных задач программирования?

   Ответ. https://stackoverflow.com.

3. Что случится, если разделить переменную `int` на `0`?

   Ответ. Вызов исключения `DivideByZeroException` при делении целого или дробного числа.

4. Что произойдет в случае деления переменной `double` на `0`?

   Ответ. Тип `double` содержит особое значение `Infinity`. Экземпляры чисел с плавающей запятой могут иметь специальные значения — `NaN` (не число), `PositiveInfinity` и `NegativeInfinity`.

5. Что происходит при переполнении переменной `int`, то есть когда ей присваивается значение, выходящее за пределы допустимого диапазона?

   Ответ. Циклическое переполнение, если только не поместить инструкцию в блок `checked`. В последнем случае будет вызвано исключение `OverflowException`.

6. Чем отличаются инструкции `x = y++;` и `x = ++y;`?

   Ответ. В случае инструкции `x = y++;` будет присвоено значение `x`, а затем инкрементировано значение `y`. В случае инструкции `x = ++y;` сначала будет инкрементировано значение `y`, а затем результат присвоен переменной `x`.

7. В чем разница между ключевыми словами `break`, `continue` и `return` при использовании в инструкции цикла?

   Ответ. Инструкция `break` завершает весь цикл и продолжает выполнение кода после цикла, инструкция `continue` завершает текущую итерацию цикла и продолжает выполнение с начала цикла (следующая итерация), а инструкция `return` завершает текущий вызов метода и продолжает выполнение после вызова метода.

8. Из каких трех частей состоит инструкция `for` и какие из них обязательны?

   Ответ. Три части инструкции `for` — это инициализатор, условие и инкремент. Условие требуется обязательно и должно быть логическим выражением, которое возвращает значение `true` или `false`, а две другие части опциональны.

9. В чем разница между операторами `=` и `==`?

   Ответ. `=` — оператор присваивания значений переменным, а `==` — оператор равенства, возвращающий значение `true` или `false`.

10. Будет ли скомпилирована такая инструкция: `for ( ; true; ) ;`?

    Ответ. Да. Для инструкции `for` требуется только логическое выражение. Инструкции `initializer` и `incrementer` опциональны и могут быть опущены. Данная инструкция `for` будет выполняться постоянно. Это пример бесконечного цикла.

Упражнение 3.1

Что произойдет при выполнении кода, приведенного ниже?

```
int max = 500;
for (byte i = 0; i < max; i++)
{
  WriteLine(i);
}
```

- Код будет циклически работать без остановки, поскольку значение переменной `i` может быть только в диапазоне от 0 до 255. Поэтому, как только значение якобы превышает `255`, оно возвращается к `0` и, следовательно, всегда будет меньше чем `max (500)`.
- Для предотвращения непрерывного выполнения цикла можно обернуть код инструкцией проверки `checked`. Это приведет к тому, что исключение будет вызываться после достижения значения `255`, например:

```
254
255
System.OverflowException says Arithmetic operation resulted in an overflow.
```

Глава 4. Создание, отладка и тестирование функций

1. Для чего в языке C# используется ключевое слово `void`?

   Ответ. Оно указывает, что метод не имеет возвращаемого значения.

2. Сколько параметров может быть у метода в языке C#?

   Ответ. Метод с 16 383 параметрами может быть скомпилирован, запущен и вызван. Если параметров больше хотя бы на один, то во время выполнения будет выведено сообщение о возникшем исключении. В IL реализованы предопределенные коды операции для загрузки не более четырех параметров, а также специальный код операции для загрузки не более 16 бит (65 536) параметров. Наилучшая практика заключается в ограничении ваших методов тремя или четырьмя параметрами. Можно объединить несколько параметров в один класс для инкапсуляции их в один параметр. Более подробную информацию см. на сайте http://stackoverflow.com/questions/12658883/what-is-the-maximum-number-of-parameters-that-a-c-sharp-method-can-be-defined-as.

3. Чем отличается назначение клавиши F5 и сочетаний клавиш Ctrl+F5, Shift+F5 и Ctrl+Shift+F5 при использовании Visual Studio 2017?

   Ответ. Клавиша F5 сохраняет, компилирует, запускает и присоединяет отладчик, сочетание клавиш Ctrl+F5 сохраняет, компилирует и запускает его, сочетание Shift+F5 останавливает отладчик, а Ctrl+Shift+F5 перезапускает его.

4. Куда производится запись вывода метода `Trace.WriteLine`?

   Ответ. Метод `Trace.WriteLine` записывает вывод в любой настроенный должным образом прослушиватель трассировки. По умолчанию это в том числе и окно Output (Вывод) Visual Studio 2017, однако вы можете настроить запись вывода в консоль, текстовый файл или любой собственный прослушиватель.

5. Каковы пять уровней трассировки?

   Ответ. `0` = `None`, `1` = `Error`, `2` = `Warning`, `3` = `Info` и `4` = `Verbose`.

6. Чем отличаются классы `Debug` и `Trace`?

   Ответ. Метод `Debug` активен только в период разработки. Метод `Trace` активен и в период разработки и после выпуска релизной версии приложения.

7. Как расшифровывается принцип AAA в модульном тестировании?

   Ответ. Arrange, Act, Assert — подготовка, действие, проверка.

8. Каким атрибутом следует сопроводить методы тестирования при написании теста с помощью xUnit?

   Ответ. Атрибутом `[Fact]`.

9. Какая команда `dotnet` отвечает за запуск тестов xUnit?

   Ответ. Команда `dotnet test`.

10. Что такое TDD?

    Ответ. Test Driven Development — разработка на основе тестирования. Больше информации см. на сайте https://docs.microsoft.com/en-us/dotnet/core/testing/.

Глава 5. Создание пользовательских типов с помощью объектно-ориентированного программирования

1. Какие четыре модификатора доступа вы знаете и для чего они используются?

   Ответ. Четыре модификатора доступа и их предназначение перечислены ниже:

   - `private` — доступ ограничен содержащим типом;
   - `internal` — доступ ограничен содержащим типом и любым другим типом в текущей сборке;
   - `protected` — доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса;
   - `public` —неограниченный доступ.

2. В чем разница между ключевыми словами `static`, `const` и `readonly`?

   Ответ. Разница между ключевыми словами `static`, `const` и `readonly` показана ниже:

   - `static` — создает элемент, общий для всех экземпляров и с доступом из типа;
   - `const` — устанавливает поле как фиксированное литеральное значение, которое никогда не должно меняться;
   - `readonly` — создает поле, которое может быть назначено только во время выполнения с помощью конструктора.

3. Для чего служит конструктор?

   Ответ. Конструктор служит для выделения памяти и инициализации значений полей.

4. Зачем с ключевым словом `enum` используется атрибут `[Flags]`, если требуется хранить комбинированные значения?

   Ответ. Если не применить данный атрибут к типу `enum` при сохранении скомбинированных значений, то сохраненное значение `enum`, являющееся комбинацией, будет возвращаться в качестве сохраненного целочисленного значения вместо списка текстовых значений, разделенных запятыми.

5. В чем польза ключевого слова `partial`?

   Ответ. Его можно задействовать для разбивки определения типа на несколько файлов.

6. Что такое кортеж?

   Ответ. Это структура данных, состоящая из нескольких частей.

7. Для чего в C# используется ключевое слово `ref`?

   Ответ. Ключевое словов `ref` языка C# преобразует параметр, переданный по текущему значению в параметр, передаваемый по указателю, то есть ссылке.

8. Что такое перегрузка?

   Ответ. Перегрузка — это определение нескольких методов с одинаковыми именами, но разными входящими параметрами.

9. Чем поле отличается от свойства?

   Ответ. Поле — место хранения данных, на которое может быть создана ссылка. Свойство — один или два метода, получающие и/или устанавливающие значение.

10. Как сделать параметр метода необязательным?

    Ответ. Для этого нужно задать данному параметру значение по умолчанию.

Глава 6. Реализация интерфейсов и наследование классов

1. Что такое делегат?

   Ответ. Это типобезопасный указатель на метод. Его можно использовать для выполнения любого метода с соответствующей сигнатурой.

2. Что такое событие?

   Ответ. Это поле, являющееся делегатом, к которому применено ключевое слово `event`. Оно гарантирует, что используются только операторы `+=` и `-=` для безопасного объединения нескольких делегатов без замены каких-либо существующих обработчиков событий.

3. Как связаны базовый и производный классы?

   Ответ. Производный класс (подкласс) — класс, унаследованный от базового класса (суперкласса).

4. В чем разница между ключевыми словами `is` и `as`?

   Ответ. Оператор `is` возвращает значение `true`, если объект может быть приведен к типу, а оператор `as` в этом случае возвращает указатель. В противном случае возвращается значение `null`.

5. Какое ключевое слово используется для предотвращения наследования класса и переопределения метода?

   Ответ. Ключевое слово `sealed`.

   Дополнительную информацию о нем можно найти на сайте https://msdn.microsoft.com/en-us/library/88c54tsw.aspx.

6. Какое ключевое слово служит для предотвращения создания экземпляра класса с помощью нового ключевого слова `new`?

   Ответ. Ключевое слово `abstract`.

   Дополнительную информацию о нем можно найти на сайте https://msdn.microsoft.com/en-us/library/sf985hc5.aspx.

7. Какое ключевое слово применяется для переопределения члена класса?

   Ответ. Ключевое слово `virtual`.

   Дополнительную информацию о нем можно найти на сайте https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx.

8. Чем деструктор отличается от деконструктора?

   Ответ. Деструктор, также известный как финализатор, следует использовать для освобождения ресурсов, принадлежащих объекту. Деконструктор — новая функция C# 7, которая позволяет разбивать сложный объект на более мелкие части.

9. Как выглядят сигнатуры конструкторов, которые должны иметь все исключения?

   Ответ. Ниже приведены такие сигнатуры конструкторов:

   - конструктор без параметров;
   - конструктор со строковым параметром, обычно называемым `message`;
   - конструктор со строковым параметром, обычно называемым `message`, и параметром исключения, обычно называемым `innerException`.

10. Что такое метод расширения и как его определить?

    Ответ. Метод расширения — это способ действия компилятора, который делает статический метод статического класса одним из членов типа. Расширяемый тип определяется с помощью префикса `this`.

Глава 7. Обзор и упаковка типов .NET Standard

1. Чем пространства имен отличаются от сборок?

   Ответ. Пространство имен — логический контейнер типа. Сборка — физический контейнер типа.

2. Как сослаться на другой проект в файле `.csproj`?

   Ответ

   ```
   <ItemGroup>
     <ProjectReference Include="..\Calculator\Calculator.csproj" />
   </ItemGroup>
   ```

3. Чем пакет отличается от метапакета?

   Ответ. Метапакет — это группа пакетов.

4. Какой тип .NET представлен в C# псевдонимом `float`?

   Ответ. Тип `System.Single`.

5. Какова разница между пакетами `NETStandard.Library` и `Microsoft.NETCore.App`?

   Ответ. Пакет `Microsoft.NETCore.App` — супермножество для пакета `NETStandard.Library`. Это значит, что в первом пакете реализуются все интерфейсы API .NET Standard 2.0, а также дополнительные API, характерные для платформы .NET Core 2.0.

6. Чем отличаются платформозависимые и автономные приложения .NET Core?

   Ответ. Платформозависимым приложениям .NET Core для работы требуется наличие версии .NET Core для используемой операционной системы. Автономные приложения .NET Core содержат все необходимое для запуска без установки дополнительного программного обеспечения.

7. Что такое RID?

   Ответ. Идентификатор среды выполнения. Значения RID используются для идентификации целевых платформ, на которых планируется запуск приложения .NET Core.

8. В чем разница между командами `dotnet pack` и `dotnet publish`?

   Ответ. Команда `dotnet pack` создает пакет NuGet. Команда `dotnet publish` помещает приложение и его зависимости в каталог для развертки в размещающей системе.

9. Какие типы приложений, написанных для .NET Framework, можно портировать в .NET Core?

   Ответ. Консольные, ASP.NET MVC, веб-API ASP.NET.

10. Можно ли в проектах .NET Core использовать пакеты, написанные для .NET Framework?

    Ответ. Да, при условии, что эти пакеты вызывают только интерфейсы, поддерживаемые в .NET Standard 2.0.

Глава 8. Использование распространенных типов .NET Standard

1. Какое максимальное количество символов может быть сохранено в переменной типа `string`?

   Ответ. Максимальный размер переменной `string` может быть равен 2 Гбайт или примерно одному миллиарду символов, поскольку каждая переменная `char` задействует два байта из-за внутреннего применения для символов кодировки Юникода (UTF-16).

2. В каких случаях и почему нужно использовать класс `SecureString`?

   Ответ. Строковый тип хранит текстовые данные в памяти слишком долго, и при этом они не защищены. Тип `SecureString` шифрует текст и гарантирует немедленное освобождение памяти. Элемент `PasswordBox` системы WPF сохраняет пароль в виде переменной `SecureString`, а при запуске нового процесса параметр `Password` должен быть переменной `SecureString`. Для получения более подробной информации посетите сайт http://stackoverflow.com/questions/141203/when-would-i-need-a-securestring-in-net.

3. Когда целесообразно применять тип `StringBuilder`?

   Ответ. При конкатенации свыше трех переменных `string` с помощью типа `StringBuilder` можно снизить потребление ресурсов памяти и улучшить производительность в сравнении со способами, предусматривающими использование метода `string.Concat` и оператора `+`.

4. В каких ситуациях нужно задействовать `LinkedList`?

   Ответ. Каждый элемент в связанном списке содержит ссылку на предыдущие и следующие одноуровневые элементы, а также и на сам список, поэтому его нужно применять, когда элементы необходимо вставлять и удалять из позиций в списке, не перемещая элементы в памяти.

5. В каких случаях следует использовать класс `SortedDictionary` вместо класса `SortedList`?

   Ответ. Класс `SortedList` задействует меньше памяти, чем `SortedDictionary`, а `SortedDictionary` быстрее выполняет операции добавления и удаления несортированных данных. Если список заполняется с помощью уже отсортированных данных, то `SortedList` работает быстрее, чем `SortedDictionary`. Для получения более подробной информации посетите сайт http://stackoverflow.com/questions/935621/whats-the-difference-between-sortedlist-and-sorteddictionary.

6. Каков ISO-код языковых и региональных параметров ISO для валлийского языка?

   Ответ. `cy-GB`. Полный список кодов языковых и региональных параметров можно найти на сайте http://timtrott.co.uk/culture-codes/.

7. В чем разница между локализацией, глобализацией и интернационализацией?

   Ответ. *Локализация* оказывает влияние на пользовательский интерфейс приложения. Определяется нейтральными (только языковыми) или специфичными (языковыми и региональными) настройками. Вы предоставляете текст и другие значения на нескольких языковых версиях. К примеру, метка текстового поля с именем может быть отображена как `First name` на английском языке и `Prénom` на французском языке.

   Интернационализация влияет на данные приложения. Определяется языковыми и региональными настройками, к примеру `en-GB` для британской версии английского языка или `fr-CA` — для канадской версии французского языка. Эти настройки нужно определить для правильного форматирования десятичных значений денежных единиц, например канадских долларов вместо евро.

   Глобализация — это сочетание локализации и интернационализации.

8. Что означает символ `$` в регулярных выражениях?

   Ответ. Он представляет конец ввода.

9. Как в регулярных выражениях представить цифры?

   Ответ. `\d+` или `[0-9]+`.

10. Почему *нельзя* применять официальный стандарт для адресов электронной почты при создании регулярного выражения, цель которого — проверка адреса электронной почты пользователя?

    Ответ. Результат того не стоит. Проверка адреса электронной почты с помощью официальной спецификации не позволяет убедиться, действительно ли этот адрес существует или является ли человек, указавший адрес, его владельцем. Для получения более подробной информации посетите сайты http://davidcel.is/posts/stop-validating-email-addresses-with-regex/ и http://stackoverflow.com/questions/201323/using-a-regular-expression-to-validate-an-email-address.

Глава 9. Работа с файлами, потоками и сериализация

1. Чем отличается применение классов `File` и `FileInfo`?

   Ответ. Класс `File` содержит статические методы, поэтому его экземпляр не может быть создан. Он лучше всего подходит для одноразовых задач, таких как копирование файла. Класс `FileInfo` требует создания экземпляра объекта, представляющего файл. Его лучше всего использовать, когда нужно выполнить несколько операций с одним и тем же файлом.

2. В чем разница между методом `ReadByte` и методом `Read` потока?

   Ответ. Метод `ReadByte` при каждом вызове возвращает один байт, а метод `Read` заполняет временный массив байтами до указанной длины. Обычно метод `Read` рекомендуется применять для обработки сразу последовательности байтов.

3. В каких случаях используются классы `StringReader`, `TextReader` и `StreamReader`?

   Ответ. Класс `StringReader` служит для эффективного чтения из строки, хранящейся в памяти. `TextReader` — это абстрактный класс, который наследуют классы `StringReader` и `StreamReader` для совместной функциональности. Класс `StreamReader` используется для чтения строк из потока, который может быть текстовым файлом любого типа, включая форматы XML и JSON.

4. Для чего предназначен тип `DeflateStream`?

   Ответ. Данный тип реализует тот же алгоритм сжатия, что и GZIP, но без циклического избыточного кода, поэтому, хотя и создает меньшие по размеру сжатые файлы, не может выполнять проверку целостности данных при распаковке.

5. Сколько байтов затрачивается на символ при использовании кодировки UTF-8?

   Ответ. Зависит от символа. Большинство символов латинского алфавита хранятся с помощью одного байта. Другим символам может потребоваться два и более байта для хранения.

6. Что такое граф объектов?

   Ответ. Таковым является любой экземпляр классов в памяти, которые ссылаются друг на друга, тем самым формируя набор связанных объектов. Например, объект `Customer` может иметь свойство, ссылающееся на набор экземпляров `Order`.

7. Какой формат сериализации лучше всего подходит для наименьших затрат памяти?

   Ответ. Объектная нотация JavaScript (JSON).

8. Какой формат сериализации наиболее пригоден для кросс-платформенной совместимости?

   Ответ. Расширяемый язык разметки (XML), хотя сегодня JSON еще более эффективен.

9. Какая библиотека лучше всего подходит для работы с форматом сериализации JSON?

   Ответ. Библиотека `Newtonsoft.JSON`.

10. Сколько пакетов для сериализации доступно на сайте NuGet.org?

    Ответ. 778 пакетов.

Глава 10. Защита данных и приложений

1. Какой из алгоритмов шифрования, доступных на платформе .NET, лучше всего подойдет для симметричного шифрования?

   Ответ. Алгоритм AES.

2. Какой из алгоритмов шифрования, доступных на платформе .NET, наиболее пригоден для асимметричного шифрования?

   Ответ. Алгоритм RSA.

3. Что такое радужная атака?

   Ответ. В радужной атаке используется таблица предварительно вычисленных хешей паролей. При краже базы данных хешей паролей хакер может быстро произвести сопоставление с хешами радужной таблицы и определить изначальные пароли. Более подробную информацию можно найти на сайте http://learncryptography.com/hash-functions/rainbow-tables.

4. При использовании алгоритмов шифрования лучше применять большой или маленький размер блока?

   Ответ. Для алгоритмов шифрования лучше задействовать блоки небольшого размера.

5. Что такое хеш?

   Ответ. Это вывод фиксированного размера, являющийся результатом обработки ввода произвольного размера хеш-функцией. Хеш-функции — однонаправленные, то есть единственный способ восстановить изначальный ввод — вручную опробовать все возможные варианты ввода и сравнить результаты.

6. Что такое подпись?

   Ответ. Это значение, подставляемое в цифровой документ для удостоверения его подлинности. Действительная подпись сообщает получателю, что документ был создан известным отправителем.

7. Чем отличаются симметричное и асимметричное шифрование?

   Ответ. При симметричном шифровании используется общий ключ как для шифрования, так и для расшифровки. При ассиметричном применяется общедоступный ключ для шифрования и личный — для расшифровки.

8. Как расшифровывается RSA?

   Ответ. Rivest, Shamir, Adleman — фамилии трех человек, публично описавших алгоритм в 1978 году.

9. Почему пароли следует посолить перед сохранением?

   Ответ. Для снижения скорости атак по радужным словарям.

10. SHA1 — это протокол хеширования, созданный в Агентстве национальной безопасности Соединенных Штатов. Почему вы никогда не должны его использовать?

    Ответ. Он больше не считается безопасным. Ни один современный браузер уже не принимает сертификаты SHA-1 SSL.

Глава 11. Работа с базами данных с помощью Entity Framework Core

1. Какой тип вы бы использовали для свойства, представляющего таблицу, например для свойства `Products` контекста БД `Northwind`?

   Ответ. Тип `DbSet<T>`, где `T` — тип сущности, например, `Product`.

2. Какой тип вы бы применили для свойства, представляющего отношение «один-ко-многим», например для свойства `Products` сущности `Category`?

   Ответ. Тип `ICollection<T>`, где `T` — тип сущности, например, `Product`.

3. Какое соглашение, касающееся первичных ключей, действует в EF Core?

   Ответ. Свойство с именем `ID` или `ClassNameID` считается первичным ключом. Если тип этого свойства является одним из следующих, то свойство также обозначается как столбец `IDENTITY`: `tinyint`, `smallint`, `int`, `bigint`, `guid`.

4. Когда бы вы применили атрибут аннотации в классе элемента?

   Ответ. Данный атрибует следует применять в классе элемента, когда соглашения не могут определить правильное сопоставление между классами и таблицами. К примеру, если имя класса не соответствует имени таблицы или имя свойства не соответствует имени столбца.

5. Почему вы предпочли бы использовать Fluent API, а не атрибуты аннотации?

   Ответ. Вы можете выбрать Fluent API, а не атрибуты аннотации, если соглашения не позволяют определить правильное сопоставление между классами и таблицами или если требуется, чтобы классы элементов были чистыми и свободными от постороннего кода.

6. Что означает уровень изоляции транзакции `Serializable`?

   Ответ. Наибольшее количество блоков применяется для гарантии полной изоляции от других процессов, работающих с затрагиваемыми данными.

7. Что возвращает метод `DbContext.SaveChanges()`?

   Ответ. Целочисленное значение — количество затронутых сущностей.

8. В чем разница между «жадной» и явной загрузками?

   Ответ. «Жадная» загрузка означает, что связанные сущности включаются в исходный запрос к базе данных, таким образом, их уже не нужно будет загружать позднее. Явная загрузка означает, что связанные сущности не включаются в исходный запрос к базе данных, следовательно, их потребуется явно загружать перед использованием.

9. Как бы вы определили класс сущности EF Core для этой таблицы?

```
CREATE TABLE Employees(
  EmpID INT IDENTITY,
  FirstName NVARCHAR(40) NOT NULL,
  Salary MONEY
)
```

   Ответ. Используйте следующий класс:

```
public class Employee
{
  [Column("EmpID")]
  public int EmployeeID { get; set; }

  [Required]
  [StringLength(40)]
  public string FirstName { get; set; }

  [Column(TypeName = "money")]
  public decimal? Salary { get; set; }
}
```

10. Каковы преимущества объявления свойств сущности как `virtual`?

    Ответ. Ожидается, что следующая версия EF Core будет поддерживать «ленивую» загрузку при определении свойств сущности как `virtual`.

Глава 12. Создание запросов и управление данными с помощью LINQ

1. Каковы две обязательные составные части LINQ?

   Ответ. Поставщик данных LINQ и методы расширения LINQ. Для доступа к методам расширения LINQ следует импортировать пространство имен `System.Linq`, а затем обращаться к сборке поставщика данных того типа данных, с которыми нужно работать.

2. Какой метод расширения LINQ вы бы использовали для возврата из типа набора свойств?

   Ответ. Метод `Select`, позволяющий осуществлять проекцию (выбор) свойств.

3. Какой метод расширения LINQ вы бы применили для фильтрации последовательности?

   Ответ. Метод `Where` позволяет выполнить фильтрацию путем предоставления делегата (или лямбда-выражения), который возвращает логическое значение, показывающее, нужно ли включать значение в результаты.

4. Перечислите шесть методов расширения LINQ, выполняющих консолидацию (укрупнение) данных.

   Ответ. Методы `Max`, `Min`, `Count`, `Average`, `Sum` и `Aggregate`.

5. Чем отличаются методы расширения `Select` и `SelectMany`?

   Ответ. Метод `Select` возвращает именно то, что вы указали для возврата. Метод `SelectMany` проверяет, не являются ли выбранные вами элементы `IEnumerable<T>`, а затем разбивает их на более мелкие части. Например, если выбранный вами тип — строковое значение (то есть `IEnumerable<char>`), то метод `SelectMany` разобьет каждое возвращенное строковое значение на соответствующие отдельные значения `char`.

Глава 13. Улучшение производительности и масштабируемости с помощью многозадачности

1. Какую информацию о процессе вы можете выяснить?

   Ответ. У класса `Process` множество свойств, в числе которых `ExitCode`, `ExitTime`, `Id`, `MachineName`, `PagedMemorySize64`, `ProcessorAffinity`, `StandardInput`, `StandardOutput`, `StartTime`, `Threads` и `TotalProcessorTime`. О свойствах класса `Process` можно узнать на сайте https://msdn.microsoft.com/en-us/library/System.Diagnostics.Process_properties(v=vs.110).aspx.

2. Насколько точен тип `Stopwatch`?

   Ответ. Класс `Stopwatch` может быть точен до наносекунды (одна миллиардная секунды), но не стоит всецело полагаться н него. Можно увеличить точность, установив привязку задачи к процессору, как описано в статье, доступной по ссылке https://www.codeproject.com/Articles/61964/Performance-Tests-Precise-Run-Time-Measurements-wi.

3. Какой суффикс по соглашению должен быть применен к методу, возвращаемому `Task` или `Task<T>`?

   Ответ. Суффикс `Async`. Например, `OpenAsync` при использовании метода `Open`.

4. Какое ключевое слово следует применить к объявлению метода, чтобы в нем можно было использовать ключевое слово `await`?

   Ответ. К объявлению метода нужно применить ключевое слово `async`.

5. Как создать дочернюю задачу?

   Ответ. Для этого необходимо вызвать метод `Task.Factory.StartNew` с параметром `TaskCreationOptions.AttachToParent`.

6. Почему не стоит использовать ключевое слово `lock`?

   Ответ. Оно не позволяет указать время ожидания, что может привести к взаимоблокировке. Вместо него задействуйте метод `Monitor.Enter` со структурой `TimeSpan` и метод `Monitor.Exit`.

7. В каких случаях нужно применить класс `Interlocked`?

   Ответ. Когда в коде используются целые числа и числа с плавающей запятой, распределяемые между несколькими потоками.

8. Когда класс `Mutex` следует использовать вместо класса `Monitor`?

   Ответ. Используйте класс `Mutex`, когда необходимо совместно применять ресурсы через границы процесса.

9. Позволяет ли применение ключевых слов `async` и `await` улучшить производительность веб-приложений? Если нет, то зачем их использовать?

   Ответ. Нет. В случае с веб-приложением или веб-сервисом использование этих ключевых слов улучшает масштабируемость, но не производительность конкретных запросов, поскольку требуются дополнительные усилия по управлению потоками.

10. Можно ли отменить задачу? Как?

    Ответ. Да, отменить задание возможно. Описание действия представлено на сайте https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/cancel-an-async-task-or-a-list-of-tasks.

Марк Дж. Прайс

C# 7 и .NET Core. Кросс-платформенная разработка для профессионалов

3-е издание

Перевели с английского *М. Сагалович, С. Черников*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 18.05.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 51,600. Тираж 1200. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216;*
e-mail: books@piter.com

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217;*
e-mail: kuznetsov@piter.com

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**