EXPERT INSIGHT

C# 10 и .NET 6

Современная кросс-платформенная разработка

Шестое издание

Марк Прайс

Packt>

C# 10 and .NET 6 – Modern Cross-Platform Development

Sixth Edition

Build apps, websites, and services with ASP.NET Core 6, Blazor, and EF Core 6 using Visual Studio 2022 and Visual Studio Code

Mark J. Price

Packt>

BIRMINGHAM - MUMBAI

C# 10 и .NET 6

Шестое издание

Современная кросс-платформенная разработка

Марк Прайс

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.1
УДК 004.43
П68

Прайс Марк

П68 C# 10 и .NET 6. Современная кросс-платформенная разработка. — СПб.: Питер, 2023. — 848 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-2249-3

Шестое издание книги серьезно переработано, добавлены все новые функции, реализованные в версиях C# 10 и .NET 6.

Вы изучите принципы объектно-ориентированного программирования, научитесь писать, тестировать и отлаживать функции, реализовывать интерфейсы и наследовать классы. В издании рассматриваются API .NET, призванные решать такие задачи, как управление данными и их запросами, мониторинг и повышение производительности, а также работа с файловой системой, асинхронными потоками, сериализацией и шифрованием. В книге приведены примеры кода кросс-платформенных приложений, веб-сайтов и служб, которые вы можете создавать и развертывать на основе ASP.NET Core.

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

ББК 32.973.2-018.1
УДК 004.43

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1801077361 англ.

© Packt Publishing 2021. First published in the English language under the title ‘C# 10 and .NET 6 – Modern Cross-Platform Development - Sixth Edition.

ISBN 978-5-4461-2249-3

© Перевод на русский язык ООО «Прогресс книга», 2022

© Издание на русском языке, оформление ООО «Прогресс книга», 2022

© Серия «Для профессионалов», 2022

Краткое содержание

Об авторе 29

О научных редакторах 30

Предисловие 31

От издательства 37

Глава 1. Привет, C#! Здравствуй, .NET! 38

Глава 2. Говорим на языке C# 87

Глава 3. Управление потоком исполнения, преобразование типов и обработка исключений 140

Глава 4. Разработка, отладка и тестирование функций 179

Глава 5. Создание пользовательских типов с помощью объектно-ориентированного программирования 227

Глава 6. Реализация интерфейсов и наследование классов 270

Глава 7. Упаковка и распространение типов .NET 326

Глава 8. Работа с распространенными типами .NET 375

Глава 9. Работа с файлами, потоками и сериализация 429

Глава 10. Работа с данными с помощью Entity Framework Core 468

Глава 11. Создание запросов и управление данными с помощью LINQ 529

Глава 12. Улучшение производительности и масштабируемости с помощью многозадачности 568

Глава 13. Реальные приложения на C# и .NET 607

Глава 14. Разработка сайтов с помощью ASP.NET Core Razor Pages 637

Глава 15. Разработка сайтов с помощью паттерна MVC 686

Глава 16. Разработка и использование веб-сервисов 739

Глава 17. Создание пользовательских интерфейсов с помощью Blazor 791

Послесловие 843

Дополнительные материалы 845

Оглавление

Об авторе 29

О научных редакторах 30

Предисловие 31
- Примеры исходного кода 32
- Структура книги 32
- Необходимое программное обеспечение 35
- Скачивание цветных изображений для книги 35
- Условные обозначения 36

От издательства 37

Глава 1. Привет, C#! Здравствуй, .NET! 38
- Настройка среды разработки 40
  - Выбор подходящего инструмента и типа приложения для обучения 40
  - Кросс-платформенное развертывание 43
  - Скачивание и установка среды Visual Studio 2022 для Windows 44
  - Скачивание и установка среды Visual Studio Code 45
- Знакомство с .NET 48
  - Обзор .NET Framework 48
  - Проекты Mono, Xamarin и Unity 49
  - Обзор .NET Core 49
  - Обзор последующих версий .NET 50
  - Поддержка .NET Core 51
  - Особенность современной .NET 53

Темы современной .NET ... 54
Обзор .NET Standard ... 54
Платформы .NET и инструменты, используемые в изданиях этой книги ... 55
Знакомство с промежуточным языком ... 56
Сравнение технологий .NET ... 57
Разработка консольных приложений с использованием Visual Studio 2022 ... 57
Управление несколькими проектами с помощью Visual Studio 2022 ... 57
Написание кода с помощью Visual Studio 2022 ... 57
Компиляция и запуск кода с использованием Visual Studio ... 59
Написание программ верхнего уровня ... 61
Добавление второго проекта с помощью Visual Studio 2022 ... 62
Создание консольных приложений с помощью Visual Studio Code ... 64
Управление несколькими проектами с помощью Visual Studio Code ... 64
Написание кода с помощью Visual Studio Code ... 65
Компиляция и запуск кода с помощью инструмента dotnet ... 67
Добавление второго проекта в программе Visual Studio Code ... 68
Управление несколькими файлами с помощью Visual Studio Code ... 70
Изучение кода с помощью .NET Interactive Notebooks ... 70
Создание блокнота ... 70
Написание и запуск кода в блокноте ... 72
Сохранение кода в блокноте ... 73
Добавление в блокнот Markdown и специальных команд ... 73
Выполнение кода в нескольких ячейках ... 74
Использование .NET Interactive Notebooks для кода в этой книге ... 75
Просмотр папок и файлов для проектов ... 75
Общие папки и файлы ... 76
Код решений на GitHub ... 77
Использование репозитория GitHub для этой книги ... 77
Уточнение вопросов ... 77
Обратная связь ... 78
Скачивание кода из репозитория GitHub ... 78
Использование системы Git в Visual Studio Code и командной строки ... 79
Поиск справочной информации ... 80
Знакомство с Microsoft Docs ... 80
Получение справки для инструмента dotnet ... 80
Получение определений типов и их элементов ... 81
Поиск ответов на Stack Overflow ... 83

Поисковая система Google ... 84
Подписка на официальный блог .NET ... 85
Видеоблог Скотта Хансельмана ... 85
Практические задания ... 85
Упражнение 1.1. Проверочные вопросы ... 85
Упражнение 1.2. Практическое задание ... 86
Упражнение 1.3. Дополнительные ресурсы ... 86
Резюме ... 86

Глава 2. Говорим на языке C# ... 87
Введение в C# ... 87
Обзор версий языка и их функций ... 88
Стандарты C# ... 92
Версии компилятора C# ... 93
Основы языка C#: грамматика и терминология ... 95
Вывод версии компилятора ... 95
Грамматика языка C# ... 97
Терминология языка C# ... 99
Сравнение языков программирования с естественными языками ... 99
Изменение цветовой схемы синтаксиса ... 100
Помощь в написании правильного кода ... 100
Импорт пространств имен ... 101
Глаголы = методы ... 105
Существительные = типы данных, поля, переменные и свойства ... 105
Определение объема словаря C# ... 106
Работа с переменными ... 108
Присвоение имен и значений ... 109
Литеральные значения ... 109
Хранение текста ... 110
Хранение чисел ... 111
Целые числа ... 113
Хранение логических значений ... 117
Хранение объектов любого типа ... 118
Хранение данных динамического типа ... 119
Объявление локальных переменных ... 120
Получение и определение значений по умолчанию для типов ... 123
Хранение нескольких значений в массиве ... 124

Дальнейшее изучение консольных приложений 125
Отображение вывода пользователю 125
Получение пользовательского ввода 129
Упрощение работы с командной строкой 129
Получение клавиатурного ввода от пользователя 130
Передача аргументов в консольное приложение 131
Настройка параметров с помощью аргументов 133
Работа с платформами, не поддерживающими некоторые API 135
Практические задания 137
Упражнение 2.1. Проверочные вопросы 137
Упражнение 2.2. Проверочные вопросы о числовых типах 138
Упражнение 2.3. Практическое задание — числовые размеры и диапазоны 138
Упражнение 2.4. Дополнительные ресурсы 139
Резюме 139

Глава 3. Управление потоком исполнения, преобразование типов и обработка исключений 140
Работа с переменными 140
Унарные операции 141
Арифметические бинарные операции 142
Операция присваивания 144
Логические операции 144
Условные логические операции 145
Побитовые операции и операции побитового сдвига 146
Прочие операции 148
Операторы выбора 149
Ветвление с помощью оператора if 149
Сопоставление с образцом с помощью операторов if 151
Ветвление с помощью оператора switch 152
Сопоставление с образцом с помощью оператора switch 153
Упрощение операторов switch с помощью выражений switch 155
Операторы цикла 156
Оператор while 156
Оператор do 157
Оператор for 158
Оператор foreach 158

Приведение и преобразование типов 159
Явное и неявное приведение типов 160
Преобразование с помощью типа System.Convert 162
Округление чисел 162
Контроль правил округления 163
Преобразование значения любого типа в строку 164
Преобразование двоичного (бинарного) объекта в строку 165
Разбор строк для преобразования в числа или значения даты и времени 166
Обработка исключений 168
Оборачивание потенциально ошибочного кода в оператор try 169
Проверка переполнения 173
Выброс исключений переполнения с помощью оператора checked 173
Отключение проверки переполнения с помощью оператора unchecked 174
Практические задания 176
Упражнение 3.1. Проверочные вопросы 176
Упражнение 3.2. Циклы и переполнение 176
Упражнение 3.3. Циклы и операции 177
Упражнение 3.4. Обработка исключений 177
Упражнение 3.5. Проверка знания операций 178
Упражнение 3.6. Дополнительные ресурсы 178
Резюме 178

Глава 4. Разработка, отладка и тестирование функций 179
Написание функций 179
Пример таблицы умножения 180
Функции, возвращающие значение 182
Преобразование чисел из кардинального в порядковое 184
Вычисление факториалов с помощью рекурсии 185
Документирование функций с помощью XML-комментариев 188
Использование лямбда-выражений в реализациях функций 189
Отладка в процессе разработки 192
Преднамеренное добавление ошибок в код 192
Установка точек останова и начало отладки 193
Навигация с помощью панели средств отладки 197
Панели отладки 197
Пошаговое выполнение кода 198
Настройка точек останова 200

Ведение журнала событий во время разработки и выполнения проекта 202
Ведение журнала событий 202
Работа с типами Debug и Trace 203
Настройка прослушивателей трассировки 205
Переключение уровней трассировки 206
Модульное тестирование 212
Виды тестирования 212
Создание библиотеки классов, требующей тестирования 213
Разработка модульных тестов 214
Генерация и перехват исключений в функциях 217
Ошибки использования и ошибки выполнения 217
Часто выбрасываемые исключения в функциях 217
Стек вызовов 218
Где перехватывать исключения 221
Повторное создание исключений 221
Реализация шаблона tester-doer 223
Практические задания 224
Упражнение 4.1. Проверочные вопросы 224
Упражнение 4.2. Функции, отладка и модульное тестирование 225
Упражнение 4.3. Дополнительные ресурсы 225
Резюме 226

Глава 5. Создание пользовательских типов с помощью объектно-ориентированного программирования 227
Коротко об объектно-ориентированном программировании 227
Разработка библиотек классов 229
Создание библиотек классов 229
Определение классов в пространстве имен 230
Члены 231
Создание экземпляров классов 232
Импорт пространства имен для использования типа 233
Работа с объектами 234
Хранение данных в полях 235
Определение полей 235
Модификаторы доступа 236

Установка и вывод значений полей 236
Хранение значения с помощью типа enum 238
Хранение группы значений с помощью типа enum 239
Хранение нескольких значений с помощью коллекций 240
Коллекции дженериков 241
Создание статического поля 242
Создание константного поля 243
Создание поля только для чтения 244
Инициализация полей с помощью конструкторов 245
Запись и вызов методов 247
Возвращение значений из методов 247
Возвращение нескольких значений с помощью кортежей 248
Определение и передача параметров в методы 251
Перегрузка методов 252
Передача необязательных и именованных параметров 253
Управление передачей параметров 255
Ключевое слово ref 256
Разделение классов с помощью ключевого слова partial 256
Управление доступом с помощью свойств и индексаторов 257
Определение свойств только для чтения 257
Определение изменяемых свойств 259
Использование модификатора required при определении свойств во время создания экземпляра 260
Определение индексаторов 261
Сопоставление с образцом с помощью объектов 262
Создание и работа с библиотеками классов .NET 6 262
Определение пассажиров рейса 262
Изменения сопоставления с образцом в C# 9 или более поздних версиях 264
Работа с записями 265
Свойства только для инициализации 265
Записи 266
Позиционные элементы данных в записях 267
Практические задания 268
Упражнение 5.1. Проверочные вопросы 268
Упражнение 5.2. Дополнительные ресурсы 269
Резюме 269

Глава 6. Реализация интерфейсов и наследование классов 270
Настройка библиотеки классов и консольного приложения 271
Дополнительные сведения о методах 272
Реализация функциональности с помощью методов 272
Реализация функциональности с помощью операций 274
Реализация функциональности с помощью локальных функций 275
Подъем и обработка событий 276
Вызов методов с помощью делегатов 277
Определение и обработка делегатов 278
Определение и обработка событий 280
Обеспечение безопасности многократно используемых типов с помощью дженериков 281
Работа с типами, не являющимися дженериками 281
Работа с типами-дженериками 282
Реализация интерфейсов 284
Универсальные интерфейсы 284
Сравнение объектов при сортировке 285
Сравнение объектов с помощью отдельных классов 287
Неявные и явные реализации интерфейса 288
Определение интерфейсов с реализациями по умолчанию 289
Управление памятью с помощью ссылочных типов и типов значений 291
Определение ссылочных типов и типов значений 291
Хранение в памяти ссылочных типов и типов значений 292
Равенство типов 293
Определение типов struct 294
Ключевое слово record и тип struct 295
Освобождение неуправляемых ресурсов 296
Обеспечение вызова метода Dispose 298
Работа со значениями null 298
Создание типа, допускающего значение null 298
Ссылочные типы, допускающие значение null 299
Включение ссылочных типов, допускающих и не допускающих значение null 300
Объявление переменных и параметров, не допускающих значение null 301
Проверка на null 302
Наследование классов 304
Расширение классов 305

Сокрытие членов класса 305
Переопределение членов 306
Наследование от абстрактных классов 307
Предотвращение наследования и переопределения 309
Полиморфизм 310
Приведение в иерархиях наследования 311
Неявное приведение 311
Явное приведение 311
Обработка исключений приведения 312
Наследование и расширение типов .NET 314
Наследование исключений 314
Расширение типов при невозможности наследования 315
Использование анализатора для написания улучшенного кода 318
Подавление предупреждений 320
Практические задания 323
Упражнение 6.1. Проверочные вопросы 324
Упражнение 6.2. Создание иерархии наследования 324
Упражнение 6.3. Дополнительные ресурсы 325
Резюме 325

Глава 7. Упаковка и распространение типов .NET 326
Введение в .NET 6 326
.NET Core 1.0 327
.NET Core 1.1 328
.NET Core 2.0 328
.NET Core 2.1 328
.NET Core 2.2 329
.NET Core 3.0 329
.NET Core 3.1 329
.NET 5.0 330
.NET 6.0 330
Повышение производительности с .NET Core 2.0 до .NET 5 331
Проверка пакетов SDK для .NET на наличие обновлений 331
Компоненты .NET 331
Сборки, пакеты NuGet и пространства имен 332
Платформа Microsoft .NET и пакет SDK 333
Пространства имен и типы в сборках 334

Пакеты NuGet 334
Фреймворки....335
Импорт пространства имен в целях использования типа 335
Связь ключевых слов языка C# с типами .NET....336
Использование кода с устаревшими платформами, используя .NET Standard 339
Общие сведения о значениях по умолчанию для библиотек классов с различными пакетами SDK....340
Создание библиотеки классов .NET Standard 2.0 341
Управление пакетом SDK для .NET....341
Публикация и развертывание ваших приложений 343
Разработка консольного приложения для публикации 344
Команды dotnet 345
Получение информации о платформе .NET и ее окружении....346
Управление проектами....347
Публикация автономного приложения 347
Публикация однофайлового приложения 350
Уменьшение размера приложений с помощью обрезки 351
Декомпиляция сборок .NET 352
Декомпиляция с помощью расширения ILSpy для Visual Studio 2022 352
Декомпиляция с помощью расширения ILSpy для Visual Studio Code 353
Нет, вы не можете технически предотвратить декомпиляцию....357
Упаковка библиотек для распространения через NuGet....359
Ссылка на пакет NuGet....359
Упаковка библиотеки для NuGet 360
Изучение пакетов NuGet с помощью инструмента 364
Тестирование пакета библиотеки классов 365
Портирование приложений с .NET Framework на современной .NET....366
Можете ли вы портировать 366
Стоит ли портировать 367
Сравнение .NET Framework и современной .NET 368
.NET Portability Analyzer 368
.NET Upgrade Assistant 368
Использование библиотек, не скомпилированных для .NET Standard....369
Функции предварительного просмотра 371
Требование к функциям предварительного просмотра....371

Включение функций предварительного просмотра 372
Математические операции с дженериками 373
Практические задания 373
Упражнение 7.1. Проверочные вопросы 373
Упражнение 7.2. Дополнительные ресурсы 374
Упражнение 7.3. PowerShell 374
Резюме 374

Глава 8. Работа с распространенными типами .NET 375
Работа с числами 375
Большие целые числа 376
Работа с комплексными числами 377
Кватернионы 378
Работа с текстом 378
Извлечение длины строки 378
Извлечение символов строки 379
Разделение строк 379
Извлечение фрагмента строки 380
Проверка содержимого в строках 381
Конкатенация строк, форматирование и прочие члены типа string 382
Эффективное создание строк 383
Работа с датами и временем 383
Указание значений даты и времени 384
Глобализация с учетом дат и времени 386
Обработка дат/времени по отдельности 388
Сопоставление с образцом при помощи регулярных выражений 389
Проверка цифр, введенных в виде текста 389
Рост производительности регулярных выражений 391
Синтаксис регулярных выражений 391
Примеры регулярных выражений 392
Разбивка сложных строк, разделенных запятыми 393
Хранение нескольких объектов в коллекциях 394
Общие свойства коллекций 395
Повышение производительности за счет обеспечения пропускной способности коллекции 396
Выбор коллекции 396

Работа со списками ... 400
Работа со словарями ... 402
Работа с очередями ... 404
Сортировка коллекций ... 406
Более специализированные коллекции ... 407
Использование неизменяемых коллекций ... 407
Эффективные приемы работы с коллекциями ... 408
Работа с интервалами, индексами и диапазонами ... 409
Управление памятью с помощью интервалов ... 409
Идентификация позиций с помощью типа Index ... 410
Идентификация диапазонов с помощью типа Range ... 410
Использование индексов, диапазонов и интервалов ... 411
Работа с сетевыми ресурсами ... 412
Работа с URI, DNS и IP-адресами ... 412
Проверка соединения с сервером ... 414
Работа с отражением и атрибутами ... 415
Версии сборок ... 416
Чтение метаданных сборки ... 416
Создание пользовательских атрибутов ... 419
Возможности отражения ... 421
Работа с изображениями ... 421
Интернационализация кода ... 423
Обнаружение и изменение региональных настроек ... 424
Практические задания ... 426
Упражнение 8.1. Проверочные вопросы ... 426
Упражнение 8.2. Регулярные выражения ... 427
Упражнение 8.3. Методы расширения ... 427
Упражнение 8.4. Дополнительные ресурсы ... 428
Резюме ... 428

Глава 9. Работа с файлами, потоками и сериализация ... 429
Управление файловой системой ... 429
Работа с кросс-платформенными средами и файловыми системами ... 429
Управление дисками ... 431
Управление каталогами ... 432
Управление файлами ... 434
Управление путями ... 435

Извлечение информации о файле ... 436
Контроль работы с файлами ... 437
Чтение и запись с помощью потоков ... 438
Абстрактные и конкретные потоки ... 438
Запись в текстовые потоки ... 440
Запись в XML-потоки ... 442
Освобождение файловых ресурсов ... 443
Сжатие потоков ... 446
Сжатие с помощью алгоритма Brotli ... 448
Кодирование и декодирование текста ... 450
Кодировка строк в последовательности байтов ... 451
Кодирование и декодирование текста в файлах ... 454
Сериализация графов объектов ... 454
XML-сериализация ... 455
Генерация компактного XML ... 458
XML-десериализация ... 458
JSON-сериализация ... 459
Высокопроизводительная обработка JSON ... 460
Управление обработкой JSON ... 462
Новые методы расширения JSON для работы с ответами HTTP ... 465
Переход с Newtonsoft на новый JSON ... 465
Практические задания ... 465
Упражнение 9.1. Проверочные вопросы ... 465
Упражнение 9.2. XML-сериализация ... 466
Упражнение 9.3. Дополнительные ресурсы ... 467
Резюме ... 467

Глава 10. Работа с данными с помощью Entity Framework Core ... 468
Современные базы данных ... 468
Устаревшая Entity Framework ... 468
Entity Framework Core ... 469
Создание консольного приложения для работы с EF Core ... 470
Использование образца реляционной базы данных ... 470
Использование Microsoft SQL Server в Windows ... 472
Создание образца базы данных Northwind для SQL Server ... 473
Управление образцом базы данных Northwind с помощью Server Explorer ... 474
Использование SQLite ... 475

Создание образца базы данных Northwind для SQLite 476
Управление образцом базы данных Northwind в SQLiteStudio 477
Настройка EF Core 479
Выбор поставщика данных Entity Framework Core 479
Подключение к базе данных 480
Определение контекста базы данных Northwind 481
Определение моделей EF Core 483
Соглашения Entity Framework Core для определения модели 483
Использование атрибутов аннотаций Entity Framework Core для определения модели 484
Использование Entity Framework Core Fluent API для определения модели 485
Создание модели Entity Framework Core для таблиц Northwind 486
Добавление таблиц в контекстный класс базы данных Northwind 489
Настройка инструмента dotnet-ef 490
Создание шаблонов с использованием существующей базы данных 491
Настройка предварительных моделей 495
Запрос данных из моделей EF Core 496
Фильтрация включенных сущностей 498
Фильтрация и сортировка товаров 500
Получение сгенерированного SQL-кода 501
Логирование EF Core с помощью провайдера логов для пользователя 502
Сопоставление с образцом с помощью оператора Like 507
Определение глобальных фильтров 508
Схемы загрузки шаблонов с помощью EF Core 509
Жадная загрузка сущностей 509
Включение ленивой загрузки 510
Явная загрузка сущностей 511
Управление данными с помощью EF Core 514
Добавление сущностей 514
Обновление сущностей 515
Удаление сущностей 516
Объединение контекстов базы данных 517
Работа с транзакциями 518
Управление транзакциями с помощью уровней изоляции 518
Определение явной транзакции 519

Модели EF Core под названием Code First 520
Миграции 526
Практические задания 527
Упражнение 10.1. Проверочные вопросы 527
Упражнение 10.2. Экспорт данных с помощью различных форматов сериализации 527
Упражнение 10.3. Дополнительные ресурсы 528
Упражнение 10.4. Изучение базы данных NoSQL 528
Резюме 528

Глава 11. Создание запросов и управление данными с помощью LINQ 529
Написание выражений LINQ 529
Как работает LINQ 529
Создание выражений LINQ с помощью класса Enumerable 530
Фильтрация элементов с помощью метода Where 534
Ссылка на именованные методы 536
Упрощение кода за счет удаления явного создания экземпляра делегата 537
Использование лямбда-выражений 537
Сортировка элементов 538
Объявление запроса с помощью ключевого слова var или заданного типа 539
Фильтрация по типу 540
Работа с множествами с помощью LINQ 541
Использование LINQ с EF Core 543
Создание модели EF Core 544
Фильтрация и сортировка последовательностей 547
Проецирование последовательностей в новые типы 548
Объединение и группировка последовательностей 550
Агрегирование последовательностей 554
Подслащение синтаксиса LINQ с помощью синтаксического сахара 555
Использование нескольких потоков и параллельного LINQ 556
Разработка приложения с помощью нескольких потоков 556
Создание собственных методов расширения LINQ 559
Использование цепного метода расширения 562
Использование методов Mode и Median 562
Работа с LINQ to XML 563
Генерация XML с помощью LINQ to XML 563
Чтение XML с помощью LINQ to XML 564

Практические задания 565
Упражнение 11.1. Проверочные вопросы 566
Упражнение 11.2. Создание запросов LINQ 566
Упражнение 11.3. Дополнительные ресурсы 567
Резюме 567

Глава 12. Улучшение производительности и масштабируемости с помощью многозадачности 568
Процессы, потоки и задачи 568
Мониторинг производительности и использования ресурсов 569
Оценка эффективности типов 570
Мониторинг производительности и памяти с помощью диагностики 571
Измерение эффективности обработки строк 574
Мониторинг производительности и памяти с помощью Benchmark.NET 576
Асинхронное выполнение задач 580
Синхронное выполнение нескольких действий 580
Асинхронное выполнение нескольких действий с помощью задач 582
Ожидание выполнения задач 583
Задачи продолжения 584
Вложенные и дочерние задачи 586
Обертывание задач вокруг других объектов 587
Синхронизация доступа к общим ресурсам 589
Доступ к ресурсу из нескольких потоков 590
Применение взаимоисключающей блокировки к «раковине» 591
Синхронизация событий 594
Выполнение атомарных операций CPU 595
Использование других типов синхронизации 596
Ключевые слова async и await 597
Увеличение скорости отклика консольных приложений 597
Увеличение скорости отклика GUI-приложений 598
Улучшение масштабируемости клиент-серверных приложений 603
Общие типы, поддерживающие многозадачность 603
Ключевое слово await в блоках catch 604
Работа с асинхронными потоками 604

Практические задания 605
Упражнение 12.1. Проверочные вопросы 605
Упражнение 12.2. Дополнительные ресурсы 606
Резюме 606

Глава 13. Реальные приложения на C# и .NET 607
Модели приложений для C# и .NET 607
Разработка сайтов с помощью ASP.NET Core 608
Создание веб- и других сервисов 610
Создание мобильных и настольных приложений 611
Альтернативы .NET MAUI 612
Нововведения ASP.NET Core 613
ASP.NET Core 1.0 613
ASP.NET Core 1.1 613
ASP.NET Core 2.0 614
ASP.NET Core 2.1 614
ASP.NET Core 2.2 615
ASP.NET Core 3.0 615
ASP.NET Core 3.1 616
Blazor WebAssembly 3.2 616
ASP.NET Core 5.0 616
ASP.NET Core 6.0 616
Создание настольных приложений только для Windows 617
Устаревшие платформы приложений Windows 617
Современная поддержка .NET для устаревших платформ Windows 619
Структурирование проектов 619
Структурирование проектов в решении или рабочей области 619
Использование других шаблонов проектов 621
Установка дополнительных пакетов шаблонов 622
Разработка сущностной модели данных для базы данных Northwind 622
Разработка библиотеки классов для сущностных моделей с помощью SQLite 623
Создание библиотеки классов для сущностных моделей с помощью SQL Server 632
Практические задания 635
Упражнение 13.1. Проверочные вопросы 635
Упражнение 13.2. Дополнительные ресурсы 635
Резюме 635

Глава 14. Разработка сайтов с помощью ASP.NET Core Razor Pages 637
Веб-разработка 637
Протокол передачи гипертекста 637
Использование браузера Google Chrome для выполнения HTTP-запросов 639
Технологии клиентской веб-разработки 642
Обзор ASP.NET Core 643
Классический ASP.NET против современного ASP.NET Core 644
Создание пустого проекта ASP.NET Core 645
Тестирование и защита сайта 647
Управление средой хостинга 651
Разделение конфигурации для сервисов и конвейера 653
Как позволить сайту обрабатывать статический контент 655
Функция Razor Pages от ASP.NET Core 658
Добавление Razor Pages 658
Добавление кода к Razor Pages 658
Использование общих макетов в Razor Pages 660
Использование файлов с выделенным кодом в Razor Pages 662
Использование Entity Framework Core совместно с ASP.NET Core 665
Настройка Entity Framework Core как сервиса 665
Управление данными с помощью страниц Razor 667
Внедрение сервиса зависимостей в страницу Razor 669
Использование библиотек классов Razor 670
Создание библиотеки классов Razor 670
Отключение компактного режима просмотра папок в Visual Studio Code 671
Реализация функции сотрудников с помощью EF Core 672
Реализация частичного представления для отображения одного сотрудника 674
Использование и тестирование библиотеки классов Razor 675
Настройка сервисов и конвейера HTTP-запросов 676
Маршрутизация конечных точек 676
Проверка конфигурации маршрутизации конечных точек 677
Обобщение ключевых методов расширения промежуточного программного обеспечения 681
Визуализация HTTP-конвейера 682
Реализация анонимного встроенного делегата в качестве промежуточного программного обеспечения 682

Практические задания 684
Упражнение 14.1. Проверочные вопросы 684
Упражнение 14.2. Веб-страница, управляемая данными 685
Упражнение 14.3. Создание веб-страниц для консольных приложений 685
Упражнение 14.4. Дополнительные ресурсы 685
Резюме 685

Глава 15. Разработка сайтов с помощью паттерна MVC 686
Настройка сайта ASP.NET Core MVC 686
Создание сайтов ASP.NET Core MVC 687
Создание базы данных аутентификации для SQL Server LocalDB 688
Сайт ASP.NET Core MVC по умолчанию 689
Структура проекта сайта MVC 691
Обзор базы данных ASP.NET Core Identity 693
Работа сайта ASP.NET Core MVC 694
Инициализация ASP.NET Core MVC 694
Маршрутизация MVC по умолчанию 697
Контроллеры и действия 698
Соглашение о пути поиска представлений 701
Ведение журнала 702
Фильтры 703
Сущности и модели представлений 710
Представления 713
Добавление собственного функционала на сайт ASP.NET Core MVC 715
Определение пользовательских стилей 715
Настройка категории изображений 716
Синтаксис Razor 716
Определение типизированного представления 717
Проверка измененной главной страницы 720
Передача параметров с помощью значения маршрута 721
Тонкости привязки моделей 723
Проверка модели 727
Методы вспомогательного класса для представления 730
Отправка запросов в базу данных и использование шаблонов отображения 732
Улучшение масштабируемости с помощью асинхронных задач 735
Превращение методов действия контроллера в асинхронные 735

Практические задания 736
Упражнение 15.1. Проверочные вопросы 736
Упражнение 15.2. Реализация MVC для страницы, содержащей сведения о категориях 737
Упражнение 15.3. Улучшение масштабируемости за счет понимания и реализации асинхронных методов действий 737
Упражнение 15.4. Практика модульного тестирования контроллеров MVC 738
Упражнение 15.5. Дополнительные ресурсы 738
Резюме 738

Глава 16. Разработка и использование веб-сервисов 739
Разработка веб-сервисов с помощью Web API в ASP.NET Core 739
Аббревиатуры, типичные для веб-сервисов 739
HTTP-запросы и ответы для Web API 741
Разработка проекта Web API в ASP.NET Core 743
Функциональность веб-сервисов 746
Создание веб-сервиса для базы данных Northwind 748
Создание репозиториев данных для сущностей 750
Реализация контроллера Web API 753
Настройка репозитория клиента и контроллера Web API 755
Указание сведений о проблеме 759
Управление сериализацией XML 760
Документирование и тестирование веб-сервисов 761
Тестирование GET-запросов в браузерах 761
Тестирование HTTP-запросов с помощью расширения REST Client 763
Swagger 766
Тестирование запросов с помощью Swagger UI 768
Протоколирование HTTP 773
Обращение к веб-сервисам с помощью HTTP-клиентов 774
Класс HttpClient 774
Настройка HTTP-клиентов с помощью HttpClientFactory 775
Получение контроллером списка клиентов в формате JSON 776
Совместное использование ресурсов между разными источниками 778
Реализация расширенных функций веб-сервисов 780
Реализация проверки работоспособности API 780
Реализация анализаторов и соглашений Open API 781

Обработка проходных отказов 782
Добавление HTTP-заголовков в целях безопасности 782
Создание веб-сервисов с помощью минимальных API 784
Создание сервиса погоды с помощью минимальных API 785
Тестирование минимального сервиса погоды 786
Добавление прогнозов погоды на главную страницу сайта Northwind 787
Практические задания 789
Упражнение 16.1. Проверочные вопросы 789
Упражнение 16.2. Создание и удаление клиентов с помощью класса HttpClient 790
Упражнение 16.3. Дополнительные ресурсы 790
Резюме 790

Глава 17. Создание пользовательских интерфейсов с помощью Blazor 791
Знакомство с Blazor 791
Недостатки JavaScript 792
Silverlight: C# и .NET на основе плагина 792
WebAssembly: цель для Blazor 792
Модели хостинга Blazor 793
Компоненты Blazor 794
Различия между Blazor и Razor 795
Сравнение шаблонов проектов Blazor 795
Обзор шаблона проекта Blazor Server 795
Маршрутизация Blazor по компонентам страницы 802
Запуск шаблона проекта Blazor Server 805
Обзор шаблона проекта Blazor WebAssembly 807
Сборка компонентов с помощью Blazor Server 811
Определение и тестирование простого компонента 811
Создание маршрутизируемого компонента страницы 812
Получение сущностей в компоненте 813
Абстрагирование сервиса для компонента Blazor 816
Определение форм с помощью компонента EditForm 818
Создание и использование компонента формы клиента 819
Тестирование компонента формы клиента 823
Создание компонентов с помощью Blazor WebAssembly 824
Настройки сервера для Blazor WebAssembly 824
Настройка клиента для Blazor WebAssembly 827
Тестирование компонентов и сервиса Blazor WebAssembly 830

Улучшение приложений Blazor WebAssembly 832
Включение Blazor WebAssembly AOT 832
Поддержка прогрессивных веб-приложений 834
Анализатор совместимости браузеров для Blazor WebAssembly 835
Совместное использование компонентов Blazor в библиотеке классов 836
Взаимодействие с JavaScript 838
Библиотеки компонентов Blazor 841
Практические задания 841
Упражнение 17.1. Проверочные вопросы 841
Упражнение 17.2. Упражнения по созданию компонента таблицы умножения 842
Упражнение 17.3. Упражнения по созданию элемента навигации по стране 842
Упражнение 17.4. Дополнительные ресурсы 842
Резюме 842

Послесловие 843
Дальнейшее изучение C# и .NET 843
Следующее издание 844

Дополнительные материалы 845

Об авторе

Марк Дж. Прайс — обладатель сертификатов Microsoft Specialist: Programming in C# и Microsoft Specialist: Architecting Microsoft Azure Solutions; опыт работы — более 20 лет.

С 1993 года Марк сдал более 80 экзаменов компании Microsoft по программированию и специализируется на подготовке других людей к успешному прохождению тестирования. В период с 2001 по 2003 год посвящал все свое время разработке официального обучающего программного обеспечения в штаб-квартире Microsoft (Редмонд, США). В составе команды написал первый обучающий курс по C#, когда существовала лишь ранняя альфа-версия языка. Во время сотрудничества с Microsoft работал инструктором по повышению квалификации сертифицированных компанией специалистов на тренингах, посвященных C# и .NET. В настоящее время разрабатывает и поддерживает обучающие курсы для системы Digital Experience Platform (DXP). Кроме того, получил степень бакалавра компьютерных наук с отличием в Бристольском университете (Великобритания).

Microsoft CERTIFIED
Solutions Developer
App Builder

Microsoft Specialist
Programming in C#

О научных редакторах

Дамир Арх — профессионал с многолетним опытом разработки и сопровождения программного обеспечения (ПО): от сложных корпоративных программных проектов до современных потребительских мобильных приложений. Работал со многими языками, однако любимым остается C#. Стремясь к совершенствованию процессов разработки, Дамир является сторонником разработки через тестирование, непрерывной интеграции и непрерывного развертывания. Он делится своими знаниями, выступая в группах пользователей и на конференциях, ведет блоги и пишет статьи. Десять раз подряд получил престижную премию Microsoft MVP в области технологий разработки. В свободное время всегда в движении: любит пеший туризм, геокэшинг, бег и скалолазание.

Джованни Альсате Сандовал — системный инженер из города Медельин (Колумбия). Ему нравится все, что связано с разработкой программного обеспечения, новыми технологиями, паттернами проектирования и архитектурой ПО. Более 14 лет трудится разработчиком, техническим руководителем и архитектором ПО, в основном с технологиями компании Microsoft. Ему нравится разрабатывать системы OSS, также он внес свой вклад в Asp.NET Core SignalR, Polly, Apollo Server и другие системы. Принял участие в создании OSS-библиотеки Simmy, библиотеки OSS (chaos engineering) для .NET, основанной на Polly. Кроме того, является приверженцем DDD и горячим поклонником облачных технологий. Джованни — участник .NET Foundation, соорганизатор сообществ MDE.NET и .NET developers в Медельине, Колумбия. В последние годы специализируется на разработке надежных распределенных систем на основе облачных технологий. И последнее, но не менее важное: он твердо верит в командную работу. Его слова: «Меня бы здесь не было, если бы я не научился многому у всех моих талантливых коллег».

Сейчас Джованни трудится техническим директором в калифорнийском стартапе Curbit.

Предисловие

В книжном магазине вы увидите объемные книги по программированию, цель которых — предоставить полный материал по языку C#, библиотекам .NET, моделям приложений, таким как сайты, веб-сервисы, настольные и мобильные приложения.

Эта книга другая. Материал излагается кратко и увлекательно и содержит практические руководства по каждой теме. Широта повествования достигается за счет несколько меньшей глубины, но при желании вы найдете здесь множество ссылок, которые позволяют продолжить погружение в тему.

Издание одновременно представляет собой пошаговое руководство по изучению современных проверенных практик на языке C# с использованием кроссплатформенной .NET и краткое введение в основные типы приложений, которые можно создавать с их помощью. Книга лучше всего подходит новичкам в C# и .NET или программистам, которые работали с C# раньше и хотят усовершенствовать свои навыки.

Если у вас уже есть опыт работы со старыми версиями языка C#, то в первом разделе главы 2, касающейся C#, вы можете ознакомиться со сравнительными таблицами новых языковых функций и перейти к ним.

Если вы уже имеете опыт работы со старыми версиями .NET, то в первом разделе главы 7 можете ознакомиться с таблицами новых функций и сразу перейти к ним.

Я расскажу об интересных возможностях и подводных камнях C# и .NET, чтобы вы могли впечатлить коллег и потенциальных работодателей и быстро повысить свою продуктивность. Я мог бы нудно разъяснять каждую деталь и тем самым утомлять некоторых читателей. Но все же я надеюсь, что вы умеете пользоваться поисковыми системами типа Google и найдете в Интернете подробную информацию о темах, которые я посчитал необязательным включать в гайд для пользователей начального и среднего уровня ввиду ограничений, свойственных печатной книге.

Примеры исходного кода

Файлы примеров для выполнения упражнений из данной книги вы можете бесплатно скачать со страницы репозитория GitHub (https://github.com/markjprice/cs10dotnet6). Инструкции о том, как это сделать, я предоставлю в конце главы 1.

Структура книги

Глава 1 «Привет, C#! Здравствуй, .NET!» посвящена тому, как настроить среду разработки и с помощью Visual Studio или Visual Studio Code создавать простейшее приложение, используя язык C# и .NET. Разработка упрощенных консольных приложений предполагает знание того, как использовать программную функцию верхнего уровня, написанную на C# 9. Простые языковые конструкции и возможности библиотеки вы можете изучить в .NET Interactive Notebooks. Кроме того, вы сможете узнать, где найти справочную информацию и как связаться со мной через хранилище GitHub, чтобы получить помощь в решении проблемы или дать обратную связь по книге, которая позволит усовершенствовать ее будущие переиздания.

Глава 2 «Говорим на языке C#» знакомит нас с версиями языка C# и приводит таблицы, помогающие понять, в какой версии были представлены те или иные функциональные особенности языка. Вдобавок в этой главе вы познакомитесь с грамматикой и лексикой C#, которыми будете пользоваться каждый день, создавая исходный код своих приложений. В частности, вы узнаете, как объявлять переменные разных типов и работать с ними.

Глава 3 «Управление потоком исполнения, преобразование типов и обработка исключений» описывает использование операций для выполнения простых действий с переменными, включая сравнения, написание кода, который принимает решения, сопоставление с образцом в C# 7 — C# 10, повторение блока операторов и выполнение преобразования между типами. Кроме того, в главе рассказывается о том, как писать код для обработки исключений при их неизбежном возникновении.

Глава 4 «Разработка, отладка и тестирование функций» посвящена соблюдению принципа программирования *DRY* (Do not Repeat Yourself — «Не повторяйся») при создании многократно используемых функций. Вы узнаете, как с помощью инструментов отладки отслеживать и устранять ошибки, мониторить код в процессе его выполнения для диагностики проблем. Кроме того, я расскажу, как тщательно тестировать код, чтобы устранять ошибки и обеспечивать его стабильность и надежность до развертывания его в производственной среде.

Глава 5 «Создание пользовательских типов с помощью объектно-ориентированного программирования» знакомит с различными категориями элементов, которые может иметь тип, в том числе с полями для хранения данных и методами для выпол-

нения действий. Вы будете использовать концепции *объектно-ориентированного программирования (ООП)*, такие как агрегирование и инкапсуляция. Вы изучите языковые функции, такие как поддержка синтаксиса кортежей и переменные `out`, литералы для значений по умолчанию и автоматически определяемые имена элементов кортежей. Кроме того, вы научитесь определять и работать с неизменяемыми типами с помощью ключевого слова `record`, свойств только для инициализации и выражений, представленных в C# 9.

Глава 6 «Реализация интерфейсов и наследование классов» описывает создание новых типов из существующих с использованием объектно-ориентированного программирования. Вы узнаете, как определять операции и локальные функции, делегаты и события, как реализовывать интерфейсы с базовыми и производными классами, переопределять элементы типа. Кроме того, вы изучите концепции полиморфизма, научитесь создавать методы расширения и поймете, как выполнять приведение классов в иерархии наследования. Вы также узнаете о значительных изменениях в C# 8, вызванных введением ссылочных типов, допускающих значение `null` (nullable).

Глава 7 «Упаковка и распространение типов .NET» представляет версии .NET и содержит таблицы, в которых приведены новые функции, а также типы .NET, соответствующие стандарту .NET, и их отношение к языку C#. Вы научитесь создавать и компилировать код на любой из поддерживаемых операционных систем: Windows, macOS и Linux. Вы также узнаете, как разворачивать и упаковывать собственные приложения и библиотеки.

Глава 8 «Работа с распространенными типами .NET» описывает типы, позволяющие вашему коду выполнять типовые практические задачи, такие как управление числами и текстом, датой и временем, хранение элементов в коллекциях, работа с сетью и изображениями и реализация интернационализации.

Глава 9 «Работа с файлами, потоками и сериализация» касается взаимодействия с файловой системой, чтения и записи в файлы и потоки, кодирования текста и форматов сериализации, таких как JSON и XML, включая улучшенную функциональность и производительность классов `System.Text.Json`.

Глава 10 «Работа с данными с помощью Entity Framework Core» научит читать данные и записывать их в базы данных, такие как Microsoft SQL Server и SQLite, с помощью технологии *объектно-реляционного отображения данных* Entity Framework Core. Вы узнаете, как определить модели сущностей, которые можно сопоставлять с существующими таблицами в базе данных. Мы также расскажем, как определить модели Code First, которые могут создавать таблицы и базы данных во время выполнения программы.

Глава 11 «Создание запросов и управление данными с помощью LINQ» рассказывает о языковых расширениях *Language INtegrated Queries (LINQ)*, которые позволяют

работать с последовательностями элементов с их последующей фильтрацией, сортировкой и проецированием на различные программные выходы. Вы также узнаете о специальных возможностях Parallel LINQ (PLINQ) и LINQ для XML.

Глава 12 «Улучшение производительности и масштабируемости с помощью многозадачности» рассказывает, как одновременно выполнять несколько действий, чтобы повысить производительность, масштабируемость и продуктивность пользователей. Вы узнаете о функции `async Main` и о том, как с помощью типов в пространстве имен `System.Diagnostics` мониторить код с целью измерения производительности и эффективности.

Глава 13 «Реальные приложения на C# и .NET» знакомит с типами кроссплатформенных приложений, которые можно создавать с помощью C# и .NET. Вы также создадите модель EF Core для представления базы данных Northwind, с которой будете работать в последующих главах этой книги.

Глава 14 «Разработка сайтов с помощью ASP.NET Core Razor Pages» посвящена тому, как создавать сайты с современной HTTP-архитектурой на стороне сервера с помощью ASP.NET Core. Вы научитесь проектировать простые сайты, используя функцию ASP.NET Core, известную как Razor Pages, которая упрощает создание динамических веб-страниц для небольших сайтов. Вы также научитесь создавать запросы и получать ответы HTTP.

Глава 15 «Разработка сайтов с помощью паттерна MVC» посвящена созданию сложных сайтов таким образом, чтобы можно было легко их тестировать и управлять ими с помощью команд, использующих ASP.NET Core MVC. Кроме того, здесь описаны такие темы, как конфигурации запуска, аутентификация, маршруты, модели, представления и контроллеры.

Глава 16 «Разработка и использование веб-сервисов» описывает, как создавать поддерживающие REST веб-сервисы с помощью ASP.NET Core Web API и как правильно потреблять их, задействуя созданные фабрикой HTTP-клиенты.

Глава 17 «Разработка пользовательских веб-интерфейсов с помощью Blazor» посвящена созданию компонентов пользовательского веб-интерфейса с помощью Blazor, которые могут выполняться либо на стороне сервера, либо внутри браузера на стороне клиента. Вы проанализируете различия между Blazor Server и Blazor WebAssembly и узнаете, как создавать компоненты, упрощающие переключение между двумя моделями хостинга.

Эту книгу дополняют три главы, которые вы можете скачать по ссылке https://clck.ru/32EVw7.

Глава 18 «Создание и использование специализированных сервисов» рассказывает о создании сервисов с помощью gRPC, а также о том, как реализовывать взаимодей-

ствие между сервером и клиентом в режиме реального времени, используя SignalR. Кроме того, вы научитесь предоставлять данные в качестве веб-сервиса с помощью OData и размещать в облаке функции, реагирующие на триггеры, используя платформу Azure Functions.

Глава 19 «Разработка мобильных и настольных приложений с помощью .NET MAUI» знакомит вас с созданием кросс-платформенных мобильных и настольных приложений для Android, iOS, macOS и Windows. Вы изучите основы XAML, с помощью которых можно определять пользовательский интерфейс для графического приложения.

Глава 20 «Защита данных и приложений» посвящена тому, как с помощью шифрования защищать данные от просмотра злоумышленниками, а также тому, как применять хеширование и цифровые подписи для защиты данных от вмешательства или повреждения. Кроме того, вы узнаете, как с помощью аутентификации и авторизации защищать приложения от неавторизованных пользователей.

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

Необходимое программное обеспечение

Вы можете разрабатывать и разворачивать приложения C# и .NET с помощью Visual Studio Code на многих платформах, включая Windows, macOS и большинство модификаций Linux.

Вам необходимы операционная система, поддерживающая Visual Studio Code, и подключение к Интернету (для всех глав, кроме одной).

Вы также можете использовать Visual Studio для Windows или macOS либо сторонний инструмент, например JetBrains Rider.

Понадобится операционная система macOS для разработки приложений, описанных в главе 19, поскольку для компиляции приложений iOS вам потребуются macOS и Xcode.

Скачивание цветных изображений для книги

Вы также можете просмотреть и скачать сверстанный PDF-файл с цветными версиями оригинальных иллюстраций. Полноцветные изображения помогут быстрее разобраться в примерах. Файл доступен по адресу https://static.packt-cdn.com/downloads/9781801077361_ColorImages.pdf.

Условные обозначения

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

Фрагменты кода в тексте, имена таблиц баз данных, файлов, расширения файлов, пути, ввод пользователя, а также учетные записи в Twitter оформляются следующим образом: «Папки Controllers, Models и Views содержат классы ASP.NET Core и файлы `.cshtml` для выполнения на сервере».

Блок кода выглядит так:

```
// хранение элементов в индексированных позициях
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";
```

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

```
// хранение элементов в индексированных позициях
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";
```

Весь ввод/вывод в командной строке дается так:

```
dotnet new console
```

URL и имена папок оформляются шрифтом без засечек. Слова, которые вы видите на экране, например в меню или диалоговых окнах, отображаются в тексте так: «Нажмите кнопку Next (Далее), чтобы перейти к следующему экрану».

Новые термины и важные слова выделены *курсивом*.

Этот рисунок указывает на важные примечания и ссылки на внешние дополнительные источники информации.

Этот рисунок указывает на советы и рекомендации экспертов по разработке.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Рекомендуем просмотреть комментарии автора по адресу https://github.com/markjprice/cs10dotnet6/blob/main/errata.md: он поясняет некоторые примеры.

1 Привет, C#! Здравствуй, .NET!

Эта глава посвящена настройке среды разработки; сходствам и различиям современных версий .NET, .NET Core, .NET Framework, Mono, Xamarin и .NET Standard; а также тому, как с помощью различных инструментов создавать простейшие приложения, используя C# 10, .NET 6 и различные редакторы кода. Здесь же вы найдете сведения о том, где лучше всего искать справочную информацию.

Для этой книги в репозитории GitHub содержатся полные проекты приложений для всех редакторов кода: https://github.com/markjprice/cs10dotnet6.

Просто щелкните кнопкой мыши на точке или в приведенной выше ссылке измените .com на .dev, чтобы превратить репозиторий GitHub в редактор Visual Studio Code для Интернета, как показано на рис. 1.1.

Рис. 1.1. Visual Studio Code для Интернета в режиме реального времени, в котором открыт репозиторий книги на GitHub

Очень удобно работать в привычном редакторе кода. Вы можете сравнивать свой код с моим примером и при необходимости легко копировать и вставлять фрагменты.

В этой книге я использую термин *«современная .NET»* для обозначения платформы .NET 6 и ее предшественников, таких как .NET 5, происходящих из .NET Core. Я использую термин *«устаревшая .NET»* для обозначения платформ .NET Framework, Mono, Xamarin и .NET Standard. Современная .NET представляет собой объединение этих устаревших платформ и стандартов.

После главы 1 книгу можно разделить на три части: первая знакомит с грамматикой и терминологией языка C#; вторая содержит описание типов, доступных в .NET и предназначенных для создания функций приложения; третья включает примеры распространенных кросс-платформенных приложений, которые вы можете создавать с помощью C# и .NET.

Большинству людей проще изучать сложные темы, имитируя и повторяя действия, а не читая подробные теоретические объяснения. Поэтому я не буду перегружать книгу детальным объяснением каждого шага. Идея в том, чтобы дать вам задание написать некий код и посмотреть, что происходит при запуске.

Вам не нужно будет разбираться, как все работает. Вы поймете это в процессе создания собственных приложений и выйдя за рамки того, чему может научить книга.

Выражаясь словами Самюэля Джонсона, составившего в 1755 году толковый словарь английского языка, я, вероятно, допустил «несколько диких промахов и забавных несуразиц, от которых не может быть свободна ни одна из работ подобной сложности». Я принимаю на себя полную ответственность за них и надеюсь, что вы оцените мою попытку «поймать ветер» и написать книгу о таких быстро развивающихся технологиях, как C# и .NET, и приложениях, которые можно разработать с их помощью.

В этой главе:

- настройка среды разработки;
- знакомство с .NET;
- разработка консольных приложений с использованием Visual Studio 2022;
- создание консольных приложений с помощью Visual Studio Code;
- изучение кода с помощью .NET Interactive Notebooks;
- просмотр папок и файлов для проектов;
- управление исходным кодом с помощью GitHub;
- поиск справочной информации.

Настройка среды разработки

Прежде чем приступать к программированию, вам нужно настроить редактор кода для C#. Microsoft представляет целое семейство редакторов и *интегрированных сред разработки* (Integrated Development Environment, IDE):

- Visual Studio 2022 для Windows;
- Visual Studio 2022 для Mac;
- Visual Studio Code для Windows, Mac или Linux;
- GitHub Codespaces.

Сторонние производители разработали собственные редакторы кода C#, например JetBrains Rider.

Выбор подходящего инструмента и типа приложения для обучения

Как вы думаете, какой инструмент и тип приложения лучше всего подходят для изучения C# и .NET?

При обучении лучший инструмент — тот, который помогает писать и редактировать код. IDE предоставляют удобные в применении графические пользовательские интерфейсы, но чем они могут вам помочь? Более простой редактор кода, помогающий писать код, лучше всего подходит для обучения.

Теперь вы с уверенностью можете сказать, что лучшим является тот инструмент, с которым вы уже знакомы или который вы или ваша команда будете использовать в качестве ежедневного инструмента разработки. Поэтому я хочу, чтобы для решения своих задач вы выбрали любой редактор кода C# или IDE из числа описанных в книге, включая Visual Studio Code, Visual Studio для Windows, Visual Studio для Mac или даже JetBrains Rider.

В третьем издании книги я приводил подробные пошаговые инструкции как для Visual Studio для Windows, так и для Visual Studio Code, предназначенные для решения всех задач кодирования. К сожалению, это быстро всех запутало. Теперь в шестом издании подробные пошаговые инструкции по созданию нескольких проектов как в Visual Studio 2022 для Windows, так и в Visual Studio Code я даю только в главе 1. Затем я привожу названия проектов и общие инструкции, которые подходят для всех программ, и вы можете использовать любую подходящую именно вам.

Лучший тип приложения для изучения языковых конструкций C# и многих библиотек .NET — тот, который не отвлекает от ненужного кода приложения. Например, нет необходимости создавать настольное приложение Windows или сайт только для того, чтобы изучить оператор `switch`.

Поэтому я считаю, что лучший способ изучить возможности C# и .NET, описанные в главах 1–12, — создавать консольные приложения. Затем, в главах 13–19 и далее, вы будете учиться создавать сайты, сервисы, а также графические настольные и мобильные приложения.

Преимущества и недостатки расширения .NET Interactive Notebooks

Еще одно преимущество Visual Studio Code — расширение .NET Interactive Notebooks. Оно позволяет с легкостью писать простые фрагменты кода. С его помощью можно создать единый текстовый файл, в котором смешаны «ячейки» Markdown (форматированный текст) и код на C# и прочих подобных языках программирования, таких как PowerShell, F# и SQL (для баз данных).

Однако .NET Interactive Notebooks имеет и некоторые ограничения:

- не читаются введенные пользователем данные, например, вы не можете использовать `ReadLine` или `ReadKey`;
- невозможно передавать аргументы;
- отсутствует возможность определять собственные пространства имен;
- нет инструментов отладки (но они появятся в будущем).

Использование Visual Studio Code для разработки кросс-платформенных приложений

Visual Studio Code — самая современная и упрощенная кросс-платформенная среда разработки, разработанная компанией Microsoft. Среду можно запустить во всех распространенных операционных системах, включая Windows, macOS и множество разновидностей Linux, таких как Red Hat Enterprise Linux (RHEL) и Ubuntu.

Visual Studio Code хорошо подходит для современной кросс-платформенной разработки, поскольку имеет богатый пополняемый набор расширений для поддержки многих языков, помимо C#.

Поскольку эта среда кросс-платформенная и простая, ее можно установить на всех платформах, на которых будут развернуты ваши приложения, чтобы быстро исправлять ошибки и т. п. Выбор Visual Studio Code означает, что разработчик может использовать кросс-платформенный редактор кода для разработки кросс-платформенных приложений.

Visual Studio Code содержит широкий спектр инструментов для веб-разработки, но все еще слабо поддерживает разработку мобильных и настольных приложений.

Visual Studio Code поддерживается процессорами ARM, поэтому вы можете разрабатывать свои приложения на компьютерах Apple Silicon и Raspberry Pi.

На сегодняшний день Visual Studio Code — самая популярная интегрированная среда разработки: по результатам опроса Stack Overflow 2021 более 70 % профессиональных разработчиков выбрали именно ее.

Использование GitHub Codespaces для разработки в облаке

GitHub Codespaces — полностью сконфигурированная среда разработки, основанная на Visual Studio Code; может быть развернута в среде, размещенной в облаке, и доступна через любой браузер. GitHub Codespaces поддерживает репозитории Git, расширения и встроенный интерфейс командной строки, поэтому вы можете редактировать, запускать и тестировать с любого устройства.

Использование Visual Studio для разработки на Mac

В Microsoft Visual Studio 2022 для Mac можно разрабатывать приложения различных типов, включая консольные приложения, сайты, веб-сервисы, настольные и мобильные приложения.

Компиляция приложений для операционных систем Apple, таких как iOS, для работы на устройствах типа iPhone и iPad, требует наличия Xcode, который работает только на macOS.

Использование Visual Studio для разработки на Windows

В Microsoft Visual Studio 2022 для Windows можно разрабатывать множество типов приложений, включая консольные, настольные и мобильные, а также сайты и веб-сервисы. Вы можете с помощью Visual Studio 2022 для Windows и расширений Xamarin создавать кросс-платформенное мобильное приложение, однако для его компиляции вам все равно потребуется macOS и Xcode.

Программа работает в операционной системе Windows версии 7 SP1 или выше. Кроме того, вам понадобится операционная система Windows 10 или 11, чтобы разрабатывать приложения универсальной платформы Windows (UWP), которые устанавливаются из магазина Microsoft Store и запускаются в «песочнице» в рамках защиты вашего компьютера.

Что использовал я

Работая над этой книгой, я использовал следующие технические средства:

- ноутбук HP Spectre (Intel);
- настольный компьютер Apple Silicon Mac mini (M1);
- настольный компьютер Raspberry Pi 400 (ARM v8).

Кроме того, я использовал следующее программное обеспечение:

- Visual Studio Code в:
  - macOS на настольном компьютере Apple Silicon Mac mini (M1);
  - Windows 10 на ноутбуке HP Spectre (Intel);
  - Ubuntu 64 на Raspberry Pi 400;
- Visual Studio 2022 для Windows в:
  - Windows 10 на ноутбуке HP Spectre (Intel);
- Visual Studio 2022 для Mac на:
  - macOS на настольном компьютере Apple Silicon Mac mini (M1).

Надеюсь, что у вас тоже есть доступ к разнообразному аппаратному и программному обеспечению, так как видение различий в платформах позволяет лучше понимать проблемы разработки. Однако любой из вышеперечисленных комбинаций достаточно, чтобы изучить основы C# и .NET и научиться разрабатывать реальные приложения и сайты.

Дополнительную информацию о том, как писать код на C# и .NET, используя Raspberry Pi 400 с 64-битной версией Ubuntu Desktop, вы можете прочитать на сайте https://github.com/markjprice/cs9dotnet5-extras/blob/main/raspberry-pi-ubuntu64/README.md.

Кросс-платформенное развертывание

Выбранные вами редактор кода и используемая операционная система не влияют на то, где могут быть развернуты ваши программы.

Сейчас .NET 6 поддерживает следующие платформы для развертывания:

- *Windows* — Windows 7 SP1 или версии выше. Windows 10 версия 1607 или выше, включая Windows 11, Windows Server 2012 R2 SP1 или версии выше. Nano Server версия 1809 или выше;
- *Mac* — macOS Mojave (версия 10.14) или версии выше;
- *Linux* — Alpine Linux 3.13 или версии выше. CentOS 7 или версии выше. Debian 10 или версии выше. Fedora 32 или выше. openSUSE 15 или версии выше. Red Hat Enterprise Linux (RHEL) 7 или версии выше. SUSE Enterprise Linux 12 SP2 или версии выше. Ubuntu 16.04, 18.04, 20.04 или версии выше.
- *Android* — API 21 или версии выше.
- *iOS* — 10 или версии выше.

Поддержка Windows ARM64 в .NET 5 и более поздних версиях означает, что вы можете разрабатывать и развертывать на устройствах Windows ARM, например, Microsoft Surface Pro X. Но, как оказалось, эффективность работы на Mac с Apple M1 при запуске виртуальной машины Windows 10 в Parallels вырастает вдвое!

Скачивание и установка среды Visual Studio 2022 для Windows

Многие профессиональные разработчики Microsoft в своей повседневной работе используют Visual Studio 2022 для Windows. Даже если вы решите писать представленный в этой книге код с помощью Visual Studio Code, вы можете ознакомиться и с Visual Studio 2022 для Windows.

Если у вас нет компьютера с Windows, то можете пропустить этот подраздел и перейти к следующему, где научитесь скачивать и устанавливать Visual Studio Code на macOS или Linux.

С октября 2014 года компания Microsoft бесплатно предоставляет учащимся, участникам программ с открытым исходным кодом и другим частным лицам профессиональный выпуск Visual Studio для Windows. Он называется Community Edition. Любой из выпусков подходит для данной книги. Если вы еще не установили Visual Studio 2022, то сделайте это сейчас.

1. Скачайте Microsoft Visual Studio 2022 версии 17.0 или выше для Windows по следующей ссылке: https://visualstudio.microsoft.com/downloads/.
2. Начните установку.
3. На вкладке Workloads (Рабочие нагрузки) выберите следующие компоненты:
   - ASP.NET and web development (ASP.NET и веб-разработка);
   - Azure development (Разработка Azure);
   - .NET desktop development (Разработка рабочего стола .NET);
   - Desktop development with C++ (Настольная разработка на C++);
   - Universal Windows Platform development (Разработка универсальной платформы Windows);
   - Mobile development with .NET (Мобильная разработка с .NET).
4. На вкладке Individual components (Отдельные компоненты) в разделе Code tools (Инструменты кода) выберите следующие компоненты:
   - Class Designer (Разработчик классов);
   - Git for Windows;
   - PreEmptive Protection — Dotfuscator.

5. Нажмите кнопку Install (Установить) и подождите, пока программа установит выбранное программное обеспечение.
6. По завершении установки нажмите кнопку Launch (Запустить).
7. При первом запуске Visual Studio вам будет предложено войти в систему. Если у вас уже есть учетная запись Microsoft, то можете использовать ее. Если нет, то зарегистрируйтесь по ссылке https://signup.live.com/.
8. При первом запуске Visual Studio вам будет предложено настроить среду. В Development Settings (Настройки разработки) выберите вариант Visual C#. В качестве цветовой темы я выбрал синий, но вы можете выбрать любой другой.
9. Если вам необходимо настроить сочетания клавиш, то выберите команду меню Tools ▸ Options (Инструменты ▸ Параметры), а затем пункт Keyboard (Клавиатура).

Сочетания клавиш Microsoft Visual Studio для Windows

В этой книге я не буду описывать сочетания клавиш, поскольку их, как правило, настраивает сам пользователь. В других ситуациях, если сочетания популярны и поддерживаются разными редакторами кода, я постараюсь их описать. О том, как определить и настроить сочетания клавиш, вы можете прочитать на сайте https://docs.microsoft.com/en-us/visualstudio/ide/identifying-and-customizing-keyboard-shortcuts-in-visual-studio.

Скачивание и установка среды Visual Studio Code

За последние несколько лет Visual Studio Code быстро развилась и приятно удивила Microsoft своей популярностью. При большом желании вы можете изучить издание Insiders, которое представляет собой ежедневную сборку следующей версии.

Даже если вы планируете использовать в разработке только Visual Studio 2022 для Windows, я рекомендую вам скачать и установить программу Visual Studio Code и попытаться с ее помощью выполнить приведенные в этой главе задачи, а затем решить, хотите ли вы придерживаться далее только Visual Studio 2022.

Теперь вы готовы скачать и установить среду Visual Studio Code, среду .NET SDK и пакет C# и .NET Interactive Notebooks.

1. Скачайте и установите либо стабильную версию Stable, либо версию Insiders среды Visual Studio Code, пройдя по ссылке https://code.visualstudio.com/.

Более подробную информацию об установке среды разработки Visual Studio Code можно получить в официальном гайде по установке на сайте https://code.visualstudio.com/docs/setup/setup-overview.

2. Скачайте и установите среду .NET SDK версий 3.1, 5.0 и 6.0, перейдя по ссылке https://www.microsoft.com/net/download.

Чтобы научиться полноценно управлять пакетами SDK для .NET, необходимо установить несколько версий. В настоящее время поддерживаются три версии: .NET Core 3.1, .NET 5.0 и .NET 6.0. Вы также можете безопасно установить несколько близких к ним. В книге вы узнаете, как выбрать подходящие вам.

3. Для установки расширения C# вам необходимо сначала запустить приложение Visual Studio Code.
4. В Visual Studio Code нажмите кнопку Extensions (Расширения) или выберите команду меню View ▸ Extensions (Вид ▸ Расширения).
5. C# — одно из самых популярных доступных расширений, поэтому вы должны увидеть его в начале списка или можете ввести `C#` в поле поиска.
6. Нажмите кнопку Install (Установить) и дождитесь скачивания и установки пакета.
7. Чтобы найти расширение .NET Interactive Notebooks, введите `.NET Interactive` в поле поиска.
8. Нажмите кнопку Install (Установить) и дождитесь завершения установки.

Установка расширений

В последующих главах этой книги вы будете использовать дополнительные расширения. Если вы хотите установить их сейчас, то можете ознакомиться с их перечнем в табл. 1.1.

Таблица 1.1. Перечень дополнительных расширений

Расширение	**Описание**
C# для Visual Studio Code (работает на OmniSharp) `ms-dotnettools.csharp`	Поддержка редактирования C#, включая выделение синтаксиса, IntelliSense, переход к определению, поиск ссылок, поддержка отладки для .NET и проектов `csproj` для операционных систем Windows, macOS и Linux
.NET Interactive Notebooks `ms-dotnettools.dotnet-interactive-vscode`	Поддержка .NET Interactive в блокноте Visual Studio Code. Зависит от расширения Jupyter (`ms-toolsai.jupyter`)
Инструменты проекта MSBuild `tintoy.msbuild-project-tools`	Предоставляет файлы проектов IntelliSense для MSBuild, включая автозаполнение элементов `<PackageReference>`
REST-клиент `humao.rest-client`	Отправка HTTP-запроса и просмотр ответа непосредственно в Visual Studio Code

Расширение	Описание
Декомпилятор ILSpy .NET `icsharpcode.ilspy-vscode`	Декомпиляция сборки MSIL — поддержка сред современной .NET, .NET Framework, .NET Core и .NET Standard
Платформа Azure Functions для Visual Studio Code `ms-azuretools.vscode-azurefunctions`	Возможность создания, отладки, управления и распаковки внесерверных приложений из VS Code. Зависит от расширений учетной записи Azure (`ms-vscode.azure-account`) и ресурсов Azure (`ms-azuretools.vscode-azureresourcegroups`)
Репозитории GitHub `github.remotehub`	Возможность просмотра, поиска, редактирования и подтверждения в любом удаленном репозитории GitHub непосредственно из Visual Studio Code
SQL Server (mssql) для Visual Studio Code `ms-mssql.mssql`	Предназначено для повсеместной разработки Microsoft SQL Server, базы данных SQL Azure и хранилища данных SQL с большим набором функций
Поддержка Protobuf 3 для Visual Studio Code `zxh404.vscode-proto3`	Добавляет выделение и проверку синтаксиса, деление кода на фрагменты, автозавершение кода и форматирование кода, сопоставление фигурных скобок, а также строчные и блочные комментарии

Знакомство с версиями Microsoft Visual Studio Code

Компания Microsoft почти каждый месяц выпускает релиз и обновляет программу Visual Studio Code. Например:

- версия 1.59, август 2021 года, функциональная версия;
- версия 1.59.1, август 2021 года, исправленная версия.

В книге используется версия 1.59. Однако версия Microsoft Visual Studio Code менее важна, чем версия расширения C# для Visual Studio Code, которое вы установите позже.

Расширение C# не требуется, однако оно предоставляет технологию IntelliSense при вводе, навигации по коду и отладке, поэтому его весьма желательно установить и поддерживать C# более поздних версий.

Сочетания клавиш Microsoft Visual Studio Code

В этой книге я не буду знакомить вас с сочетаниями клавиш, используемыми для таких действий, как создание нового файла, поскольку они часто различаются в разных операционных системах. Ситуации, в которых я покажу сочетания клавиш, — это когда вам нужно многократно нажимать клавишу, например, во время отладки. Кроме того, с большой вероятностью эти сочетания одинаковы в различных операционных системах.

О том, как настроить сочетания клавиш для Visual Studio Code, вы можете узнать на сайте https://code.visualstudio.com/docs/getstarted/keybindings.

Я рекомендую вам скачать PDF-файл, в котором описаны сочетания клавиш для вашей операционной системы:

- *Windows* — https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf;
- *macOS* — https://code.visualstudio.com/shortcuts/keyboard-shortcuts-macos.pdf;
- *Linux* — https://code.visualstudio.com/shortcuts/keyboard-shortcuts-linux.pdf.

Знакомство с .NET

.NET 6, .NET Core, .NET Framework и Xamarin — связанные и зависимые друг от друга платформы для разработки приложений и сервисов. В данном разделе я познакомлю вас с каждой из этих платформ .NET.

Обзор .NET Framework

.NET Framework — платформа для разработки, включающая *общеязыковую исполняющую среду* (Common Language Runtime, CLR), которая управляет выполнением кода, и обширную *библиотеку базовых классов* (Base Class Library, BCL) для создания приложений.

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

Начиная с версии .NET Framework 4.5.2, платформа — официальный компонент операционной системы Windows. Компоненты имеют ту же поддержку, что и основные продукты, поэтому версия 4.5.2 и более поздние следуют политике жизненного цикла ОС Windows, на которой они установлены. Платформа .NET Framework установлена на более чем одном миллиарде компьютеров, поэтому должна обновляться как можно реже. Даже исправление ошибок может вызвать проблемы, так что данная платформа обновляется нечасто.

Для .NET Framework 4.0 или версии выше все приложения на компьютере, разработанные для .NET Framework, используют одну и ту же версию CLR и библиотек, хранящихся в *глобальном кэше сборок* (Global Assembly Cache, GAC). Это может привести к неполадкам, если некоторым приложениям потребуется определенная версия для совместимости.

В сущности, платформа .NET Framework работает только в среде Windows и считается устаревшей. Не рекомендуется использовать ее для создания новых приложений.

Проекты Mono, Xamarin и Unity

Сторонние разработчики создали .NET Framework под названием *Mono*. Хоть он и был кросс-платформенным, но значительно отставал от официальной реализации .NET Framework.

Проект Mono занял нишу в качестве основы мобильной платформы *Xamarin*, а также кросс-платформенных сред для разработки игр, таких как *Unity*.

В 2016 году Microsoft приобрела компанию Xamarin и теперь бесплатно предоставляет пользователям дорогостоящее решение Xamarin в качестве расширения для среды разработки Visual Studio. Кроме того, корпорация переименовала инструмент разработки Xamarin Studio, который мог создавать только мобильные приложения, в Visual Studio для Mac и внедрила возможность создавать другие типы проектов, такие как консольные приложения и веб-сервисы. Чтобы обеспечить более точное соответствие между удобством и производительностью, в Visual Studio 2022 для Mac компания Microsoft заменила компоненты редактора Xamarin Studio компонентами из Visual Studio 2022 для Windows. В числе прочего программа Visual Studio 2022 для Mac была переработана, чтобы сделать ее действительно нативным приложением пользовательского интерфейса macOS и повысить надежность и эффективность работы со встроенными вспомогательными технологиями macOS.

Обзор .NET Core

Сегодня мы живем в реально 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. Переписывая код .NET Framework с целью обеспечить истинную кросс-платформенность, сотрудники компании воспользовались возможностью реорганизовать компоненты .NET и удалить те из них, которые не считаются ядром.

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

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

Платформа .NET Core быстро развивается и ввиду возможности ее развертывания рядом с приложением может часто меняться, и эти изменения не повлияют на другие приложения .NET Core на той же машине. Большинство обновлений, которые Microsoft может внести в .NET Core и современную .NET, нельзя легко добавить в .NET Framework.

Обзор последующих версий .NET

На конференции разработчиков Microsoft Build в мае 2020 года команда .NET объявила, что их планы по унификации .NET отложены. Было объявлено, что версия .NET 5 будет выпущена 10 ноября 2020 года и объединит все различные платформы .NET, кроме мобильных. Единая платформа .NET будет поддерживать мобильные устройства только в .NET 6 в ноябре 2021 года.

Платформа .NET Core будет переименована в .NET, а в номере основной версии будет пропущен номер четыре, чтобы избежать путаницы с NET Framework 4.x. Microsoft планирует выпускать ежегодные основные версии каждый ноябрь, подобно тому как Apple выпускает основные версии iOS каждый сентябрь.

В табл. 1.2 показано, когда были выпущены основные версии .NET, когда запланированы будущие выпуски и какая версия используется в этой книге.

Таблица 1.2. Основные версии .NET

Версия	Дата выпуска	Издание	Выпуск
.NET Core RC1	Ноябрь 2015 года	Первое	Март 2016 года
.NET Core 1.0	Июнь 2016 года	—	—
.NET Core 1.1	Ноябрь 2016 года	—	—
.NET Core 1.0.4 и .NET Core 1.1.1	Март 2017 года	Второе	Март 2017 года
.NET Core 2.0	Август 2017 года	—	—
Обновление .NET Core для UWP в Windows 10 Fall Creators	Октябрь 2017 года	Третье	Ноябрь 2017 года
.NET Core 2.1 (LTS)	Май 2018 года	—	—
.NET Core 2.2 (текущая)	Декабрь 2018 года	—	—
.NET Core 3.0 (текущая)	Сентябрь 2019 года	Четвертое	Октябрь 2019 года
.NET Core 3.1 (LTS)	Декабрь 2019 года	—	—
Blazor WebAssembly 3.2 (текущая)	Май 2020 года	—	—
.NET 5.0 (текущая)	Ноябрь 2020 года	Пятое	Ноябрь 2020 года
.NET 6.0 (LTS)	Ноябрь 2021 года	Шестое	Ноябрь 2021 года
.NET 7.0 (текущая)	Ноябрь 2022 года	Седьмое	Ноябрь 2022 года
.NET 8.0 (LTS)	Ноябрь 2023 года	Восьмое	Ноябрь 2023 года

.NET Core 3.1 включает Blazor Server для создания веб-компонентов. Кроме того, в данный выпуск компания Microsoft планировала добавить Blazor WebAssembly, но отложила это включение. Позже Blazor WebAssembly был выпущен как необязательная надстройка для .NET Core 3.1. Поскольку он имеет версию 3.2, я внес его в табл. 1.2, чтобы исключить его из LTS .NET Core 3.1.

Поддержка .NET Core

Версии .NET имеют либо долгосрочную поддержку (long term support, LTS), либо текущую (current), как описано ниже.

- Релизы *LTS* стабильны и требуют меньше обновлений в течение срока их службы. Хорошо подходят для приложений, не требующих частых обновлений. Релизы LTS будут поддерживаться в течение трех лет после их выпуска или в течение года после выпуска следующей версии LTS.
- *Текущие* релизы включают функции, которые могут меняться в зависимости от обратной связи. Хорошо подходят для активно разрабатываемых приложений, поскольку предоставляют доступ к последним обновлениям. По истечении шести месяцев сопровождения или спустя 18 месяцев после появления в общем доступе предыдущая вспомогательная версия больше не будет поддерживаться.

В целях обеспечения безопасности и надежности критические исправления поставляются для релизов независимо от режима поддержки. Чтобы получить поддержку, вы всегда должны быть в курсе последних обновлений. Например, если система работает под версией 1.0, а версия 1.0.1 уже выпущена, значит, необходимо установить версию 1.0.1.

Чтобы лучше разобраться в текущих и LTS версиях, полезно проанализировать следующий рисунок: черный цвет обозначает версии с трехлетней поддержкой для LTS; серый — текущие версии; штриховка — поддержку версий, выпущенных в течение шести месяцев после нового основного или минорного релиза (рис. 1.2).

Рис. 1.2. Поддержка различных версий

Если вам нужна долгосрочная поддержка со стороны Microsoft, то выберите .NET 6.0 и используйте, пока не выйдет .NET 8.0, и даже после того, как Microsoft выпустит .NET 7.0. Это связано с тем, что .NET 7.0 будет текущим выпуском и, следовательно, лишится поддержки раньше, чем .NET 6.0. Просто помните: даже выпуски LTS вы должны обновлять до выпусков с исправленными ошибками, таких как 6.0.1.

Истекли сроки службы всех версий .NET Core и современной .NET, кроме следующих:

- срок службы .NET Core 3.1 истекает 3 декабря 2022 года;
- срок службы .NET 6.0 истекает в ноябре 2024 года.

Версии .NET Runtime и .NET SDK

Управление версиями .NET Runtime следует за семантическим управлением версиями, то есть большое приращение указывает на критические изменения, незначительное — на новые функции, а приращение исправлений — на исправление ошибок.

Управление версиями .NET SDK не следует за семантическим управлением версиями. Старший и дополнительный номера версий привязаны к версии среды выполнения, с которой они совпадают. Номер патча соответствует соглашению, указывающему основную и вспомогательную версии SDK.

Пример приведен в табл. 1.3.

Таблица 1.3. Версии .NET Runtime и .NET SDK

Изменения	Runtime	SDK
Первоначальная версия	6.0.0	6.0.100
SDK, исправленная версия	6.0.0	6.0.101
Runtime и SDK, исправленная версия	6.0.1	6.0.102
SDK, обновленная версия	6.0.1	6.0.200

Удаление старых версий .NET

Обновления среды выполнения .NET совместимы с основной версией, такой как 6.x, и обновленные выпуски .NET SDK поддерживают возможность создания приложений, ориентированных на предыдущие версии среды выполнения, что позволяет безопасно удалять старые версии.

Вы можете увидеть, какие SDK и среды выполнения установлены в настоящее время, используя следующие команды:

- `dotnet --list-sdks`;
- `dotnet --list-runtimes`.

Для удаления пакетов SDK для .NET в Windows используйте раздел *App & features*. В macOS или Windows используйте инструмент `dotnet-core-uninstall`. Он устанавливается по умолчанию.

Например, при написании четвертого издания я каждый месяц использовал следующую команду:

```
dotnet-core-uninstall remove --all-previews-but-latest --sdk
```

Особенность современной .NET

Современная версия .NET имеет модульную структуру по сравнению с устаревшей монолитной .NET Framework. Это открытый исходный код, и Microsoft открыто принимает решения об улучшениях и изменениях. Компания приложила особые усилия к повышению производительности современной .NET.

Дистрибутив меньше, чем последняя версия .NET Framework, благодаря удалению устаревших и не кросс-платформенных технологий. Например, Windows Forms и *Windows Presentation Foundation (WPF)* можно использовать для создания приложений с *графическим пользовательским интерфейсом* (graphical user interface, GUI), однако они тесно связаны с экосистемой операционной системы Windows, поэтому были удалены из .NET Core в операционных системах macOS и Linux.

Разработка для Windows

Одна из новых функций современной .NET — поддержка запуска старых приложений Windows Forms и WPF с помощью пакета Windows Desktop Pack, входящего в состав версии .NET Core 3.1 или версий выше для операционной системы Windows. Поэтому он считается более полным, чем пакет SDK для операционных систем macOS и Linux. При необходимости вы можете внести небольшие изменения в устаревшее приложение Windows, а затем перестроить его для .NET Core, чтобы воспользоваться новыми функциями и улучшениями производительности.

Веб-разработка

Компоненты ASP.NET Web Forms и Windows Communication Foundation (WCF) представляют собой устаревшие технологии для создания веб-приложений и сервисов. Сегодня они используются лишь некоторыми разработчиками, поэтому также были удалены из современной .NET. Вместо этого разработчики предпочитают задействовать компоненты ASP.NET MVC и ASP.NET Web-API, SignalR и gRPC. Эти технологии были реорганизованы и объединены в новый продукт, ASP.NET Core, работающий на платформе .NET. Вы узнаете о них в главах 14, 15, 16 и 18.

Некоторые разработчики .NET Framework озабочены тем, что в современной .NET отсутствуют веб-формы ASP.NET, WCF и Windows Workflow (WF), и надеются, что компания Microsoft пересмотрит данный вопрос. Существуют проекты с открытым исходным кодом, позволяющие WCF и WF перейти на современную .NET. Более подробную информацию можно получить на сайте https://devblogs.microsoft.com/dotnet/supporting-the-community-with-wf-and-wcf-oss-projects/. По следующей ссылке вы можете найти проект с открытым исходным кодом для компонентов Blazor Web Forms: https://github.com/FritzAndFriends/BlazorWebFormsComponents.

Разработка баз данных

Entity Framework (EF) 6 — технология объектно-реляционного отображения, предназначенная для работы с информацией, хранящейся в реляционных базах данных, таких как Oracle и Microsoft SQL Server. За годы развития данная технология погрязла в различных доработках, поэтому ее новый кросс-платформенный API подвергся оптимизации, будет поддерживать нереляционные базы данных, такие как Microsoft Azure Cosmos DB, и теперь называется Entity Framework Core. Об этом вы узнаете в главе 10.

Если у вас есть приложения, использующие устаревший EF, то его версия 6.3 поддерживается в .NET Core 3.0 или более поздней.

Темы современной .NET

С помощью Blazor компания Microsoft разработала сайт, представляющий основные темы современной .NET: https://themesof.NET/.

Обзор .NET Standard

В 2019 году с платформой .NET сложилась следующая ситуация: существует три ветви .NET, и все они разрабатываются компанией Microsoft:

- *.NET Core* — для кросс-платформенных и новых приложений;
- *.NET Framework* — для устаревших приложений;
- *Xamarin* — для мобильных приложений.

Все три ветви имеют свои достоинства и недостатки, поскольку предназначены для разных ситуаций. Это привело к тому, что разработчик должен изучить три платформы, каждая из которых раздражает своими странностями и ограничениями.

По этой причине Microsoft работает над .NET Standard: спецификацией для набора API, которая может быть реализована на всех платформах .NET. Например, указанием на наличие базовой поддержки является совместимость платформы с .NET Standard 1.4.

В рамках .NET Standard 2.0 и более поздних версий компания Microsoft привела все три платформы в соответствие с современным минимальным стандартом, что значительно упростило разработчикам совместное использование кода с любой разновидностью .NET.

Это позволило добавить в .NET Core 2.0 и более поздние версии большинство недостающих API, необходимых разработчикам для переноса старого кода, написанного для .NET Framework, в кросс-платформенную .NET Core. Однако некоторые API уже реализованы, но при работе выдают исключение о том, что фактически не должны использоваться! Обычно это происходит из-за различий в операционной системе, в которой вы запускаете .NET. Как обрабатывать эти исключения, вы узнаете в главе 2.

Важно понимать, что .NET Standard — просто стандарт. Вы не можете установить .NET Standard так же, как не можете установить HTML5. Чтобы использовать HTML5, вам необходимо установить браузер, который реализует стандарт HTML5.

Чтобы использовать .NET Standard, необходимо установить платформу .NET, которая реализует спецификацию .NET Standard. Последняя версия .NET Standard 2.1 реализована только в .NET Core 3.0, Mono и Xamarin. Для некоторых функций C# 8.0 требуется .NET Standard 2.1. Она не реализована в .NET Framework 4.8, поэтому мы рассматриваем .NET Framework как устаревшую.

После выпуска .NET 6 в ноябре 2021 года потребность в .NET Standard значительно уменьшилась, поскольку теперь существует единая среда разработки .NET. для всех платформ, включая мобильные. .NET 6 имеет один BCL и два CLR: CoreCLR оптимизирована для серверных или настольных приложений, таких как сайты и настольные приложения Windows, а среда выполнения Mono оптимизирована для мобильных приложений и приложений браузера с ограниченными ресурсами.

Даже в этом случае приложения и сайты, созданные для .NET Framework, должны будут поддерживаться, поэтому важно понимать, что вы можете создавать библиотеки классов .NET Standard 2.0, обратно совместимые с устаревшими платформами .NET.

Платформы .NET и инструменты, используемые в изданиях этой книги

В первом издании, написанном в марте 2016 года, я сосредоточился на функциональности .NET Core, но использовал .NET Framework, когда важные или полезные функции еще не были реализованы в .NET Core, поскольку это было еще до окончательного выпуска .NET Core 1.0. Visual Studio 2015 использовалась для большинства примеров, Visual Studio Code описывалась кратко.

Во втором издании все примеры кода .NET Framework были (почти) полностью убраны, чтобы читатели могли сосредоточиться на примерах .NET Core, которые действительно работают кросс-платформенно.

Третье издание завершило переход. Оно было переписано так, чтобы весь код был чистым .NET Core. Но предоставление пошаговых инструкций как для Visual Studio Code, так и для Visual Studio 2017 привело к усложнению.

Четвертое издание продолжило тенденцию, показывая только примеры кодирования с использованием Visual Studio Code для всех глав, кроме двух последних. В главе 20 использовалась Visual Studio, работающая в операционной системе Windows 10, а в главе 21 — Visual Studio для Mac.

В пятом издании глава 20 была перенесена в приложение Б, чтобы освободить место для новой главы 20. Проекты Blazor можно создавать с помощью Visual Studio Code.

В этом шестом издании глава 19 полностью обновлена, чтобы показать, как можно создавать мобильные и настольные кросс-платформенные приложения с помощью Visual Studio и *.NET MAUI* (Multi-platform App UI — многоплатформенный пользовательский интерфейс приложения).

К седьмому изданию и выпуску .NET 7 Visual Studio Code будет иметь расширение для поддержки .NET MAUI. После этого разработчики смогут использовать Visual Studio Code для всех примеров в книге.

Знакомство с промежуточным языком

Компилятор C# (под названием *Roslyn*), используемый инструментом командной строки `dotnet`, конвертирует ваш исходный код на языке C# в код на *промежуточном языке* (Itermediate Language, IL) и сохраняет его в *сборке* (DLL- или EXE-файле). Операторы кода на промежуточном языке (IL) похожи на код ассемблера, только выполняются с помощью виртуальной машины CoreCLR в .NET.

В процессе работы код IL загружается CoreCLR из сборки, *динамически* (just-in-time, JIT) компилируется компилятором в собственные инструкции CPU, а затем исполняется с помощью CPU на вашем компьютере.

Преимущество такого трехэтапного процесса компиляции заключается в том, что Microsoft может создавать CLR не только для Windows, но и для Linux и macOS. Один и тот же код IL запускается в любой среде благодаря второму процессу компиляции, который генерирует код для конкретной операционной системы и набора команд CPU.

Независимо от того, на каком языке написан исходный код, например C#, Visual Basic или F#, все приложения .NET используют код IL для своих инструкций, хранящихся в сборке. Microsoft и другие компании предоставляют инструменты дизассемблера, которые могут открывать сборку и раскрывать данный код IL, например расширение декомпилятора ILSpy .NET.

Сравнение технологий .NET

В табл. 1.4 перечисляются и сравниваются актуальные технологии .NET.

Таблица 1.4. Сравнение технологий .NET

Технология	Возможности	Хостовая ОС
Современная .NET	Современный набор функций, полная поддержка C# 8, 9 и 10, портирование существующих и создание новых приложений для Windows, мобильных и веб-приложений/сервисов	Windows, macOS, Linux, Android, iOS
.NET Framework	Устаревший набор функций, ограниченная поддержка C# 8, отсутствие поддержки C# 9 или 10, поддержка существующих приложений	Только Windows
Xamarin	Только для мобильных и настольных приложений	Android, iOS, macOS

Разработка консольных приложений с использованием Visual Studio 2022

Цель данного раздела — показать, как создать консольное приложение с помощью среды разработки Visual Studio 2022 для Windows.

Если у вас нет компьютера с установленной операционной системой Windows или вы хотите использовать Visual Studio Code, то можете пропустить этот раздел, поскольку код будет таким же, различаться будут лишь инструменты.

Управление несколькими проектами с помощью Visual Studio 2022

В Visual Studio 2022 используются так называемые *решения*, позволяющие работать с несколькими проектами одновременно. Мы будем применять решение для управления двумя проектами, которые вы создадите в этой главе.

Написание кода с помощью Visual Studio 2022

Начнем писать код!

1. Запустите среду разработки Visual Studio 2022.
2. В открывшемся диалоговом окне нажмите кнопку Create a new project (Создать новый проект).

3. В диалоговом окне Create a new project (Создать новый проект) в поле Search for templates (Поиск шаблонов) введите `console` и выберите вариант Console Application (Консольное приложение). Убедитесь, что выбрали шаблон проекта C#, а не другой язык, например F# или Visual Basic (рис. 1.3).

Рис. 1.3. Выбор шаблона проекта Console Application (Консольное приложение)

4. Нажмите кнопку Next (Далее).
5. В диалоговом окне Configure your new project (Настройка нового проекта) введите имя проекта `HelloCS`, укажите его расположение `C:\Code` и имя решения `Chapter01` (рис. 1.4).

Рис. 1.4. Настройка имени и расположения для вашего нового проекта

6. Нажмите кнопку Next (Далее).

Мы намеренно используем старый шаблон проекта .NET 5.0, чтобы взглянуть на полноценное консольное приложение. В следующем разделе вы создадите консольное приложение .NET 6.0 и проанализируете изменения.

7. В диалоговом окне Additional information (Дополнительная информация) в раскрывающемся списке Target Framework (Целевая платформа) обратите внимание на варианты Current (Текущая) и long-term support versions of .NET (длительная поддержка версий .NET). Выберите вариант .NET 5.0 (Current) (.NET 5.0 (Текущая)) и нажмите кнопку Create (Создать).
8. На панели Solution Explorer (Проводник решений) дважды щелкните на файле `Program.cs`, и обратите внимание, что на панели Solution Explorer (Проводник решений) отобразился проект HelloCS (рис. 1.5).

Рис. 1.5. Редактирование Program.cs в Visual Studio 2022

9. В файле `Program.cs` измените строку 9 так, чтобы текст, который выводится в консоль, сообщал: `Hello, C#!`.

Компиляция и запуск кода с использованием Visual Studio

Следующая задача — это компиляция и запуск кода.

1. В Visual Studio выберите команду меню Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки).
2. В окне консоли отобразится результат запуска вашего приложения (рис. 1.6).

Рис. 1.6. Запуск консольного приложения в Windows

3. Нажмите любую клавишу, чтобы закрыть окно консоли и вернуться в Visual Studio.
4. Выберите проект HelloCS, а затем на панели инструментов Solution Explorer (Проводник решений) нажмите кнопку Show All Files (Показать все файлы) и обратите внимание, что отображаются папки bin и obj, созданные компилятором (рис. 1.7)

Рис. 1.7. Отображение папок и файлов, созданных компилятором

Папки и файлы, созданные компилятором

Итак, компилятором сгенерированы две папки с именами obj и bin. Вам пока не нужно разбираться в их содержимом. Просто имейте в виду, что для работы компилятору необходимо создавать временные папки и файлы. Вы можете удалить эти папки с содержимым, после чего при необходимости они будут воссозданы программой. Разработчики часто так делают, чтобы «очистить» проект. Для этой цели

в Visual Studio в меню Build (Сборка) есть команда Clean Solution (Очистить решение), удаляющая ненужные временные файлы. Эквивалентная команда в Visual Studio Code — `dotnet clean`.

- Папка obj содержит один скомпилированный *объектный* файл для каждого файла исходного кода. Эти объекты еще не были объединены в окончательный исполняемый файл.
- Папка bin содержит *двоичный* исполняемый файл приложения или библиотеки классов. Более подробно эта тема рассматривается в главе 7.

Написание программ верхнего уровня

Вы можете подумать, что для вывода одного лишь сообщения `Hello, C#!` объем кода слишком велик.

Этот код написан с помощью шаблона. Но существует ли более простой способ?

Да, в C# 9 и более поздних версиях он есть и известен как *программы верхнего уровня*.

Сравним консольное приложение, созданное с помощью шаблона проекта, как показано ниже:

```
using System;

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

Код программы верхнего уровня нового простого консольного приложения выглядит так:

```
using System;

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

Намного проще, не так ли? Если вам пришлось начать с пустого файла и писать все операторы самостоятельно, то это лучше. Но как это работает?

Во время компиляции весь шаблонный код для определения класса `Program` и его метода `Main` генерируется и выполняется вместе с написанными вами операторами.

Основные моменты, которые следует помнить о программах верхнего уровня:

- любые операторы `using` по-прежнему должны располагаться в начале файла;
- в проекте может быть только один такой файл.

Оператор `using System;`, расположенный в начале файла, импортирует пространство имен `System`. Благодаря этому выполняется оператор `Console.WriteLine`. Более подробно о пространствах имен мы поговорим в следующей главе.

Добавление второго проекта с помощью Visual Studio 2022

Чтобы изучить программы верхнего уровня, добавим в наше решение второй проект.

1. В Visual Studio выберите команду меню File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Новый проект).
2. В диалоговом окне Add a new project (Добавить новый проект) в разделе Recent project templates (Недавние шаблоны проектов) выберите вариант Console Application [C#] (Консольное приложение [C#]) и нажмите кнопку Next (Далее).
3. В диалоговом окне Configure your new project (Настройка нового проекта) в поле Project name (Имя проекта) введите `TopLevelProgram`, оставьте путь `C:\Code\Chapter01` и нажмите кнопку Next (Далее).
4. В диалоговом окне Additional information (Дополнительная информация) выберите вариант .NET 6.0 (Long-term support) (.NET 6.0 (Длительная поддержка)) и нажмите кнопку Create (Создать).
5. На панели Solution Explorer (Проводник решений) в проекте `TopLevelProgram` дважды щелкните на файле `Program.cs`, чтобы открыть его.
6. Обратите внимание, что в файле `Program.cs` код состоит только из комментария и одного оператора, поскольку в приложении используется функционал программ верхнего уровня, представленный в C# 9:

```
// для дополнительной информации см. https://aka.ms/new-console-template
Console.WriteLine("Hello, World!");
```

Но, когда я ранее представлял концепцию программ верхнего уровня, нам нужен был оператор `using System;`, а здесь — нет. Почему?

Неявно импортированные пространства имен

Хитрость в том, что нам по-прежнему необходимо импортировать пространство имен `System`, но теперь это за нас делает сам функционал C# 10. Посмотрим, как это сделать.

1. На панели Solution Explorer (Проводник решений) выберите проект TopLevelProgram и нажмите кнопку Show All Files (Показать все файлы). Обратите внимание, что отображаются папки bin и obj, созданные компилятором.
2. Разверните папку obj/Debug/net6.0 и откройте файл `TopLevelProgram.GlobalUsings.g.cs`.
3. Обратите внимание, что этот файл автоматически создается компилятором для проектов, предназначенных для .NET 6, и что в нем применяется функция *глобального импорта* (нововведение в C# 10). Она импортирует некоторые часто используемые пространства имен, такие как `System`, чтобы задействовать во всех файлах кода:

```
// <autogenerated />
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.NET.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

Более подробная информация, касающаяся этой функции, приведена в следующей главе. Пока просто отмечу, что существенная разница между .NET 5 и .NET 6 заключается в том, что многие шаблоны проектов, например консольных приложений, используют новые языковые функции, скрывающие от разработчика, что происходит на самом деле.

4. В проекте TopLevelProgram в файле `Program.cs` измените оператор, чтобы вывести другое сообщение и версию операционной системы, как показано в коде ниже:

```
Console.WriteLine("Hello from a Top Level Program!");
Console.WriteLine(Environment.OSVersion.VersionString);
```

5. На панели Solution Explorer (Проводник решений) щелкните правой кнопкой мыши на решении Chapter01, выберите Set Startup Projects (Установить запускаемые проекты), а затем Current selection (Текущий выбор) и нажмите кнопку OK.
6. На панели Solution Explorer (Проводник решений) выберите проект TopLevelProgram (или любой файл или папку в нем) и обратите внимание, что теперь этот проект является запускаемым, поэтому его имя выделено жирным шрифтом.
7. Выберите команду меню Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки), чтобы запустить проект TopLevelProgram, и обратите внимание на результат (рис. 1.8).

Рис. 1.8. Запуск программы верхнего уровня в решении Visual Studio с двумя проектами в Windows

Создание консольных приложений с помощью Visual Studio Code

Цель этого раздела — продемонстрировать, как создавать консольные приложения в программе Visual Studio Code.

Если вы не планируете использовать Visual Studio Code или интерактивные блокноты .NET, то пропустите этот и следующий разделы, а затем перейдите к разделу «Просмотр папок и файлов для проектов» далее в этой главе.

Инструкции и иллюстрации в данном разделе предназначены для пользователей Windows, но при этом аналогичны и для Visual Studio Code в операционных системах macOS и Linux.

Основными отличиями будут собственные действия командной строки, такие как удаление файла: команда и путь, вероятно, будут разными для операционных систем Windows или macOS и Linux. К счастью, инструмент командной строки `dotnet` будет одинаковым на всех платформах.

Управление несколькими проектами с помощью Visual Studio Code

В Visual Studio Code используются так называемые *рабочие области*, позволяющие открывать несколько проектов одновременно и управлять ими. Мы будем использовать рабочую область для управления двумя проектами, которые вы создадите в этой главе.

Написание кода с помощью Visual Studio Code

Приступим к написанию кода!

1. Запустите Visual Studio Code.
2. Убедитесь, что у вас нет открытых файлов, папок или рабочих областей.
3. Выберите команду меню File ▸ Save Workspace As (Файл ▸ Сохранить рабочую область как).
4. В открывшемся диалоговом окне перейдите к своей пользовательской папке в macOS (моя называется markjprice), к папке Documents в операционной системе Windows или любому каталогу или диску, на котором хотите сохранить свои проекты.
5. Нажмите кнопку New Folder (Новая папка) и назовите папку Code. (Если вы выполняли шаги, которые указаны в разделе, посвященном программе Visual Studio 2022, то эта папка уже создана.)
6. В папке Code создайте папку и назовите ее Chapter01-vscode.
7. В папке Chapter01-vscode сохраните рабочую область под именем `Chapter01.code-workspace`.
8. Выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область) или нажмите кнопку Add Folder (Добавить папку).
9. В папке Chapter01-vscode создайте подпапку HelloCS.
10. Выберите папку HelloCS и нажмите кнопку Add (Добавить).
11. Выберите команду меню View ▸ Terminal (Вид ▸ Терминал).

Мы намеренно используем старый шаблон проекта .NET 5.0, чтобы рассмотреть полноценное консольное приложение. В следующем разделе вы создадите консольное приложение .NET 6.0 и проанализируете, что изменилось.

12. Взглянув на панель TERMINAL (Терминал), убедитесь, что вы находитесь в папке HelloCS, а затем с помощью инструмента `dotnet` создайте консольное приложение .NET 5.0, как показано ниже:

```
dotnet new console -f net5.0
```

13. Вы увидите, что инструмент командной строки `dotnet` создает в текущей папке проект Console Application, а на панели EXPLORER (Проводник) отображаются два созданных файла, `HelloCS.cs` и `Program.cs`, а также папка obj (рис. 1.9).

Рис. 1.9. Панель EXPLORER (Проводник) покажет, что были созданы два файла и папка

14. На панели EXPLORER (Проводник) выберите файл `Program.cs`, чтобы открыть его в окне редактора. При первом запуске Visual Studio Code, возможно, придется загружать и устанавливать зависимые объекты C#, такие как OmniSharp, .NET Core Debugger и Razor Language Server, если вы не сделали это при установке расширения C# или же были выпущены обновления этих компонентов. Visual Studio Code отразит ход выполнения в окне Output (Вывод) и в конечном итоге выведет сообщение `Finished`:

```
Installing C# dependencies...
Platform: win32, x86_64

Downloading package 'OmniSharp for Windows (.NET 4.6 / x64)' (36150
KB)..................... Done!
Validating download...
Integrity Check succeeded.
Installing package 'OmniSharp for Windows (.NET 4.6 / x64)'

Downloading package '.NET Core Debugger (Windows / x64)' (45048
KB)..................... Done!
Validating download...
Integrity Check succeeded.
Installing package '.NET Core Debugger (Windows / x64)'

Downloading package 'Razor Language Server (Windows / x64)' (52344
KB)..................... Done!
Installing package 'Razor Language Server (Windows / x64)'
Finished
```

Предыдущий вывод относится к программе Visual Studio для Windows. При работе в macOS или Linux вывод будет выглядеть немного иначе. В любом случае будут загружены и установлены эквивалентные компоненты для вашей операционной системы.

15. Папки с именами obj и bin будут созданы, и, когда вы увидите уведомление о том, что необходимые ресурсы отсутствуют, нажмите кнопку Yes (Да) (рис. 1.10).

Рис. 1.10. Сообщение, предупреждающее о добавлении необходимых ресурсов для сборки и отладки

16. Если уведомление исчезнет прежде, чем вы нажмете кнопку, можете отобразить его снова, щелкнув на значке колокольчика, расположенном в правой части строки состояния.
17. Через несколько секунд появится папка .vscode с файлами, которые Visual Studio Code использует во время отладки для предоставления таких функций, как IntelliSense. Более подробную информацию вы найдете в главе 4.
18. В файле `Program.cs` измените строку 9 так, чтобы текст, который выводится в консоль, сообщал: `Hello, C#!`.

Выберите команду меню File ▸ Auto Save (Файл ▸ Автосохранение). Она избавит вас от необходимости каждый раз вспоминать о сохранении перед повторным обновлением приложения.

Компиляция и запуск кода с помощью инструмента dotnet

Следующая задача — это компиляция и запуск кода.

1. Выберите меню View ▸ Terminal (Вид ▸ Терминал) и введите следующую команду:

```
dotnet run
```

2. На панели TERMINAL (Терминал) отобразится результат запуска вашего приложения (рис. 1.11).

Рис. 1.11. Результат запуска вашего первого приложения

Добавление второго проекта в программе Visual Studio Code

Добавим в нашу рабочую область второй проект для изучения программ верхнего уровня.

1. В программе Visual Studio Code выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область) или нажмите кнопку Add Folder (Добавить папку).
2. Перейдя в папку Chapter01-vscode, нажмите кнопку New Folder (Новая папка), чтобы создать подпапку TopLevelProgram. Затем выберите ее и нажмите кнопку Add (Добавить).
3. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и в появившемся списке выберите пункт TopLevelProgram. Как вариант, вы можете щелкнуть правой кнопкой мыши на папке TopLevelProgram на панели EXPLORER и в контекстном меню выбрать команду Open in Integrated Terminal (Открыть во встроенном терминале).
4. Взглянув на панель TERMINAL (Терминал), убедитесь, что вы находитесь в папке TopLevelProgram, а затем для создания нового консольного приложения введите следующую команду:

```
dotnet new console
```

Используя рабочие области, будьте осторожны при вводе команд на панели TERMINAL (Терминал). Прежде чем вводить потенциально опасные команды, убедитесь, что находитесь в правильной папке! Вот почему, прежде чем вводить команду для создания нового консольного приложения, я попросил вас создать новый терминал для TopLevelProgram.

5. Выберите команду меню View ▸ Command Palette (Вид ▸ Набор команд).
6. Введите `omni`, а затем в появившемся списке выберите пункт OmniSharp: Select Project (OmniSharp: Выбрать проект).
7. В списке из двух проектов выберите проект TopLevelProgram и при появлении запроса нажмите кнопку Yes (Да), чтобы добавить ресурсы, необходимые для отладки.

Чтобы включить отладку и другие полезные функции, такие как форматирование кода и переход к определению, необходимо указать OmniSharp, над каким проектом вы работаете в Visual Studio Code. Вы можете быстро переключаться между активными проектами, выбирая проект/папку, расположенные справа от значка пламени в левой части строки состояния.

8. На панели EXPLORER (Проводник) в папке TopLevelProgram выберите файл `Program.cs`, а затем измените существующий оператор, чтобы вывести другое сообщение, а также версию операционной системы, как показано ниже:

```
Console.WriteLine("Hello from a Top Level Program!");
Console.WriteLine(Environment.OSVersion.VersionString);
```

9. На панели TERMINAL (Терминал) введите команду для запуска программы:

```
dotnet run
```

10. Проанализируйте вывод на панели TERMINAL (Терминал) (рис. 1.12).

Рис. 1.12. Запуск программы верхнего уровня в рабочей области Visual Studio Code с двумя проектами в Windows

Если вы запускаете программу в среде macOS Big Sur, то операционная система среды будет другой, как показано в выводе ниже:

```
Hello from a Top Level Program!
Unix 11.2.3
```

Управление несколькими файлами с помощью Visual Studio Code

Если требуется одновременная работа с несколькими файлами, то можете размещать их рядом друг с другом по мере их редактирования.

1. На панели EXPLORER (Проводник) разверните два проекта.
2. Откройте оба файла `Program.cs` из двух проектов.
3. Нажав и удерживая кнопку мыши, перетащите вкладку окна редактирования для одного из ваших открытых файлов и расположите их так, чтобы вы могли одновременно видеть оба файла.

Изучение кода с помощью .NET Interactive Notebooks

Инструмент .NET Interactive Notebooks упрощает написание кода еще больше, чем программы верхнего уровня. Ему нужна программа Visual Studio Code. Так что если вы еще не установили среду разработки, то сейчас самое время сделать это.

Создание блокнота

В первую очередь нам необходимо создать блокнот.

1. В Visual Studio Code закройте все открытые рабочие области или папки.
2. Выберите команду меню View ▸ Command Palette (Вид ▸ Набор команд).
3. Введите `.NET inter`, а затем выберите пункт .NET Interactive: Create new blank notebook (.NET Interactive: Создать новый пустой блокнот) (рис. 1.13).
4. Когда будет предложено выбрать расширение файла, выберите Create as '.dib' (Создать как .dib).

Рис. 1.13. Создание нового пустого блокнота .NET

.dib — экспериментальный формат файлов, созданный компанией Microsoft во избежание путаницы и проблем совместимости с форматом .ipynb, используемым в интерактивных блокнотах Python. Ранее это расширение файла предназначалось только для блокнотов Jupyter, которые могут содержать интерактивную (I) смесь данных, кода на языке Python (PY) и вывода в файле блокнота (NB). В .NET Interactive Notebooks эта концепция была расширена и теперь позволяет использовать C#, F#, SQL, HTML, JavaScript, Markdown и другие языки. Формат .dib поддерживает смешение языков программирования. Кроме того, поддерживается преобразование между форматами файлов .dib и .ipynb.

5. В качестве языка программирования по умолчанию выберите C#.
6. Если доступна обновленная версия .NET Interactive, то вам придется подождать, пока не будет установлено обновление. Выберите команду меню View ▸ Output (Вид ▸ Вывод) и в раскрывающемся списке выберите пункт NET Interactive: diagnostics (NET Interactive: диагностики). Вам придется немного подождать. Создание блокнота может занять пару минут, так как потребуется запустить среду размещения для .NET. Если через несколько минут ничего не произойдет, закройте программу Visual Studio Code и перезапустите ее.
7. После того как расширение .NET Interactive Notebooks установлено, в окне диагностик OUTPUT (Вывод) будет указано, что процесс запущен (номер вашего процесса и порта будет отличаться от приведенного ниже в выводе):

```
Extension started for VS Code Stable.
...
Kernel process 12516 Port 59565 is using tunnel uri http://
localhost:59565/
```

Написание и запуск кода в блокноте

Теперь мы можем писать код в блокноте.

1. Первая ячейка уже должна быть настроена на C# (.NET Interactive), но если это не так, то в правом нижнем углу ячейки кода выберите язык C# (.NET Interactive). Обратите также внимание на другие языки программирования для ячейки кода (рис. 1.14).
2. Внутри ячейки кода C# (.NET Interactive) введите оператор для вывода сообщения в консоль и обратите внимание, что вам не нужно заканчивать оператор точкой с запятой, как это обычно делается в полном приложении:

```
Console.WriteLine("Hello, .NET Interactive!")
```

Рис. 1.14. Выбор языка программирования для блокнота .NET Interactive

3. Нажмите кнопку Execute Cell (Выполнить ячейку), расположенную слева от ячейки кода, и обратите внимание на вывод, который появляется в сером поле под ячейкой (рис. 1.15).

Рис. 1.15. Запуск кода в блокноте и просмотр вывода

Сохранение кода в блокноте

Как и любой другой файл, мы должны сохранить блокнот.

1. Выберите команду меню File ▸ Save As (Файл ▸ Сохранить как).
2. Откройте папку Chapter01-vscode и сохраните файл под именем `Chapter01.dib`.
3. Закройте вкладку редактора `Chapter01.dib`.

Добавление в блокнот Markdown и специальных команд

Мы можем смешивать и сопоставлять ячейки, содержащие разметку Markdown и код, с помощью специальных команд.

1. Выберите команду меню File ▸ Open File (Файл ▸ Открыть файл) и выберите файл `Chapter01.dib`.
2. Если появится вопрос Do you trust the authors of these files? (Доверяете ли вы этому файлу?), нажмите кнопку Open (Открыть).
3. Чтобы добавить ячейку Markdown, наведите указатель мыши на блок кода и выберите пункт +Markdown.
4. Введите уровень заголовка 1, как показано в следующем Markdown:

```
# Глава 1
Mixing *rich* **text** and code is cool!
```

5. Чтобы завершить редактирование ячейки и просмотреть разметку Markdown, нажмите галочку, расположенную в правом верхнем углу ячейки.

Если ваши ячейки расположены в неправильном порядке, то вы можете переместить их и расположить по-другому.

6. Наведите указатель мыши между ячейками с разметкой Markdown и кодом и выберите пункт +Code.
7. Введите команду для вывода информации о версии .NET Interactive, как показано ниже:

```
#!about
```

8. Нажмите кнопку Execute Cell (Выполнить ячейку) и проанализируйте результат (рис. 1.16).

Рис. 1.16. Сочетание Markdown, кода и специальных команд в блокноте .NET Interactive

Выполнение кода в нескольких ячейках

Если у вас в блокноте несколько ячеек кода, то вы должны выполнить код из предыдущих ячеек, прежде чем их контекст станет доступным в последующих.

1. В нижней части блокнота добавьте новую ячейку кода, а затем введите оператор для объявления переменной и назначения ей целочисленного значения, как показано ниже:

   ```
   int number = 8;
   ```

2. В нижней части блокнота добавьте новую ячейку кода, а затем введите оператор для вывода числовой переменной `number`:

   ```
   Console.WriteLine(number);
   ```

3. Обратите внимание, что вторая ячейка кода не подозревает о переменной `number`, поскольку та была определена и назначена в другой ячейке кода, также известной как контекст (рис. 1.17).
4. В первой ячейке нажмите кнопку Execute Cell (Выполнить ячейку), чтобы объявить и присвоить значение переменной, а затем во второй ячейке нажмите

кнопку Execute Cell (Выполнить ячейку), чтобы вывести переменную `number`. Проанализируйте результат. (Кроме того, в первой ячейке можно нажать кнопку Execute Cell and Below (Выполнить ячейку и опуститься вниз.)

Рис. 1.17. Переменная number не существует в текущей ячейке или контексте

Если связанный код разделен между двумя ячейками, то не забудьте выполнить код из предыдущей ячейки, прежде чем выполнять код из последующей. В начале блокнота расположены кнопки Clear Outputs (Очистить выходные данные) и Run All (Запустить все). Это очень удобно, так как вы можете нажать их последовательно, чтобы проверить, что все ячейки кода выполняются корректно, если расположены в правильном порядке.

Использование .NET Interactive Notebooks для кода в этой книге

В остальных главах я не буду приводить четкие инструкции по использованию блокнотов, но в репозитории GitHub для этой книги вы при необходимости найдете блокноты решений (https://github.com/markjprice/cs10dotnet6/tree/main/notebooks). Я предполагаю, что многие читатели захотят запустить мои блокноты для языковых и библиотечных функций, описанных в главах 2–12, чтобы увидеть их в действии и изучить, не прибегая к необходимости писать законченное приложение, даже если это просто консольное приложение.

Просмотр папок и файлов для проектов

В этой главе вы создали два проекта — `HelloCS` и `TopLevelProgram`.

Программа Visual Studio Code использует файл рабочей области для управления несколькими проектами. В Visual Studio 2022 для этой же цели используется файл решения. Вы также создали интерактивный блокнот .NET Interactive.

В результате получилась структура папок и файлов, которые будут использоваться в последующих главах при работе с несколькими проектами (рис. 1.18).

Рис. 1.18. Структура папок и файлов для двух проектов из этой главы

Общие папки и файлы

Хотя файлы `.code-workspace` и `.sln` различны, проектные папки и файлы, такие как `HelloCS` и `TopLevelProgram`, идентичны для Visual Studio 2022 и Visual Studio Code. Это означает, что вы можете использовать в работе одновременно оба редактора кода.

- В Visual Studio 2022, открыв решение, выберите команду меню File ▸ Add Existing Project (Файл ▸ Добавить существующий проект), чтобы добавить файл проекта, созданный другим инструментом.
- В Visual Studio Code, открыв рабочую область, выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область), чтобы добавить папку проекта, созданную другим инструментом.

Исходный код, как и файлы .csproj и .cs, идентичен, однако папки bin и obj, которые автоматически создаются компилятором, могут иметь различные файлы, что может привести к ошибкам. Если вы хотите открыть проект из программы Visual Studio 2022 в Visual Studio Code или наоборот, то предварительно удалите временные папки bin и obj. Вот почему в этой главе я попросил вас создать другую папку для решений Visual Studio Code.

Код решений на GitHub

Код решений для этой книги в репозитории GitHub содержит отдельные папки для файлов Visual Studio Code, Visual Studio 2022 и блокнота .NET Interactive, как показано в следующем списке:

- решения Visual Studio 2022: https://github.com/markjprice/cs10dotnet6/tree/main/vs4win;
- решения Visual Studio Code: https://github.com/markjprice/cs10dotnet6/tree/main/vscode;
- решения .NET Interactive Notebook: https://github.com/markjprice/cs10dotnet6/tree/main/notebooks.

Если потребуется вспомнить, как создавать несколько проектов и управлять ими в редакторе кода, то вернитесь к этой главе. В репозитории GitHub приведены пошаговые инструкции для четырех редакторов кода (Visual Studio 2022 для Windows, Visual Studio Code, Visual Studio 2022 для Mac и JetBrains Rider), а также дополнительные иллюстрации: https://github.com/markjprice/cs10dotnet6/blob/main/docs/code-editors/.

Использование репозитория GitHub для этой книги

Git — широкоиспользуемая система управления исходным кодом. GitHub — компания, сайт и настольное приложение, которое облегчает работу с Git. Компания Microsoft приобрела GitHub в 2018 году, поэтому интеграция GitHub с инструментами Microsoft будет усиливаться.

Для этой книги я создал репозиторий на GitHub и использую его, чтобы:

- сохранить исходный код примеров, который можно поддерживать после выхода книги из печати;
- предоставлять дополнительные материалы, такие как исправления опечаток, небольшие улучшения, списки полезных ссылок и более длинные публикации, которые не поместились в эту книгу;
- дать читателям возможность при необходимости связаться со мной.

Уточнение вопросов

Если вы оказались в затруднении, следуя какой-либо из инструкций в книге, или заметили ошибку в тексте либо коде, то сообщите о проблеме в репозитории GitHub.

1. Откройте браузер и перейдите по ссылке https://github.com/markjprice/cs10dotnet6/issues.

2. Нажмите кнопку New Issue (Новый выпуск).
3. Введите как можно больше информации, которая поможет мне проанализировать проблему. Например:
   1) ваша операционная система, например Windows 11 64-bit или macOS Big Sur версии 11.2.3;
   2) ваше оборудование, например Intel, Apple Silicon или ARM CPU;
   3) ваш редактор кода, например Visual Studio 2022, Visual Studio Code и т. п., включая номер версии;
   4) фрагмент кода, которого, по вашему мнению, будет достаточно для выявления проблемы;
   5) описание ожидаемого и наблюдаемого поведения;
   6) снимки экрана (при возможности).

Я не всегда могу быстро ответить на такое сообщение. Но я хочу, чтобы моя книга была полезна читателям, так что если могу помочь, то с радостью сделаю это.

Обратная связь

Если вы хотите написать отзыв о книге, то на странице `README.md` репозитория GitHub доступны ссылки на некоторые опросы. Вы можете оставить отзыв анонимно или, если ожидаете ответа, указать адрес электронной почты. На указанный адрес электронной почты я отправлю только ответ на ваш отзыв.

Я люблю читать отзывы моих читателей о том, что им нравится, а что — нет, как они программируют на C# и .NET, а также предложения по улучшению. Пожалуйста, пишите мне!

Заранее благодарю вас за полезный и искренний отзыв.

Скачивание кода из репозитория GitHub

Я использовал веб-сервис GitHub для хранения ответов ко всем упражнениям, приведенным в конце каждой главы этой книги. Получить к ним доступ можно по ссылке https://github.com/markjprice/cs10dotnet6.

Если вы хотите скачать все файлы решений без работы с Git, нажмите зеленую кнопку Code (Код) и выберите пункт Download ZIP (Скачать ZIP) (рис. 1.19).

Рекомендую вам добавить предыдущую ссылку в ваши любимые закладки, так как я использую репозиторий GitHub для этой книги, чтобы публиковать исправления и другие полезные ссылки.

Рис. 1.19. Скачивание ZIP-файла из репозитория

Использование системы Git в Visual Studio Code и командной строки

Среда разработки Visual Studio Code имеет встроенную поддержку системы Git, но требует установить Git в используемой операционной системе, поэтому вам необходимо установить Git 2.0 или более позднюю версию, прежде чем получить доступ к этим функциям.

Вы можете скачать дистрибутив Git по ссылке https://git-scm.com/download.

Если хотите использовать графический интерфейс, то можете скачать GitHub Desktop по ссылке https://desktop.github.com.

Клонирование репозитория с примерами из книги

Клонируем репозиторий с примерами из книги. Далее вы воспользуетесь терминалом в программе Visual Studio Code. Но вы можете вводить команды в любой другой оболочке командной строки или окне терминала.

1. Создайте папку Repos-vscode в пользовательской папке, или в папке Documents, или там, где вы хотите хранить свои репозитории Git.
2. В программе Visual Studio Code откройте папку Repos-vscode.
3. Выберите команду меню View ▸ Terminal (Вид ▸ Терминал) и введите следующую команду:

```
git clone https://github.com/markjprice/cs10dotnet6.git
```

4. Клонирование всех решений для всех глав займет примерно минуту (рис. 1.20).

Рис. 1.20. Клонирование репозитория с примерами из книги с помощью Visual Studio Code

Поиск справочной информации

Этот раздел главы посвящен тому, как найти достоверную информацию о программировании в Интернете.

Знакомство с Microsoft Docs

Главный ресурс, позволяющий получить справочные сведения об инструментах и платформах Microsoft для разработчиков, — Microsoft Docs. Вы можете найти его по адресу docs.microsoft.com.

Получение справки для инструмента dotnet

В командной строке вы можете запросить справочную информацию об инструменте `dotnet`.

1. Чтобы открыть в окне браузера официальную документацию для команды `dotnet new`, введите в командной строке или в терминале программы Visual Studio Code следующую команду:

```
dotnet help new
```

2. Чтобы вывести справочную информацию в командной строке, используйте ключ `-h` или `--help`:

```
dotnet new console -h
```

3. Вы увидите следующий вывод (фрагмент):

```
Console Application (C#)
Author: Microsoft
Description: A project for creating a command-line application that can
run on .NET Core on Windows, Linux and macOS
Options:
  -f|--framework. The target framework for the project.
                      net6.0           - Target net6.0
                      net5.0           - Target net5.0
                      netcoreapp3.1.   - Target netcoreapp3.1
                      netcoreapp3.0.   - Target netcoreapp3.0
                  Default: net6.0

--langVersion    Sets langVersion in the created project file text –
Optional
```

Получение определений типов и их элементов

Одна из наиболее полезных функций в редакторе кода — Go To Definition (Перейти к определению). Данная функция также доступна в Visual Studio Code и Visual Studio 2022. Она позволяет увидеть, как выглядит общедоступное определение типа или элемента, полученное путем чтения метаданных в скомпилированной сборке.

Некоторые инструменты, такие как ILSpy .NET Decompiler, могут даже выполнить реверс-инжиниринг метаданных и кода IL в C#.

Рассмотрим пример использования функции Go To Definition (Перейти к определению).

1. В программе Visual Studio 2022 или Visual Studio Code откройте решение/рабочую область `Chapter01`.
2. В проекте `HelloCS` в файле `Program.cs` в методе `Main` введите следующий код, чтобы объявить целочисленную переменную `z`:

   ```
   int z;
   ```

3. Щелкните на слове `int`, а затем щелкните правой кнопкой мыши и в контекстном меню выберите пункт Go To Definition (Перейти к определению).

4. В появившемся окне вы можете увидеть определение типа данных `int` (рис. 1.21). Видно, что `int`:
   - определяется с помощью ключевого слова `struct`;
   - находится в сборке `System.Runtime`;
   - находится в пространстве имен `System`;
   - называется `Int32`;
   - является, таким образом, псевдонимом для типа `System.Int32`;
   - реализует интерфейсы, такие как `IComparable`;
   - имеет постоянные значения для своих максимальных и минимальных значений;
   - имеет такие методы, как `Parse`.

Рис. 1.21. Тип данных int

При попытке использовать Go To Definition (Перейти к определению) в Visual Studio Code иногда появляется сообщение об ошибке No definition found (Определение не найдено). Это связано с тем, что расширение C# «не знает» о текущем проекте. Чтобы решить эту проблему, выберите команду меню View ▸ Command Palette (Вид ▸ Палитра команд), введите omni, найдите пункт OmniSharp: Select Project (OmniSharp: Выбрать проект) и выберите его, а затем и правильный проект, с которым хотите работать.

В настоящее время функция Go To Definition (Перейти к определению) не очень полезна для вас, поскольку вы еще не знаете, что означают эти термины.

Прочитав главы 2–6, в которых рассказывается о C#, вы будете знать достаточно, чтобы эта функция стала полезной.

5. Прокрутите окно редактора кода вниз и найдите метод `Parse` с параметром `string` в строке 106 и комментариями, объясняющими его работу в строках 86–105 (рис. 1.22).

Рис. 1.22. Комментарии к методу Parse с параметром string

В комментарии вы увидите, что сотрудники компании Microsoft задокументировали:

- описание метода;
- параметры, такие как `string`, которые можно передать методу;
- возвращаемое значение метода, включая его тип данных;
- три исключения, возникшие при вызове этого метода исключения, включая `ArgumentNullException`, `FormatException` и `OverflowException`. Теперь мы знаем, что нам необходимо обернуть вызов данного метода в оператор `try` и то, какие исключения необходимо перехватить.

Я надеюсь, вам уже не терпится узнать, что все это значит!

Вы почти дошли до конца главы и уже в следующей главе погрузитесь в детали языка C#. Но сначала посмотрим, куда еще вы можете обратиться за получением справочной информации.

Поиск ответов на Stack Overflow

Stack Overflow — 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 на этом сайте.

1. Откройте браузер.
2. Перейдите на сайт DuckDuckGo.com и введите запрос, результаты которого отображены на рис. 1.23:

```
!so securestring
```

Рис. 1.23. Результаты поиска на Stack Overflow для securestring

Поисковая система Google

Вы можете выполнять поиск на сайте Google, задавая дополнительные настройки, чтобы увеличить вероятность нахождения нужной информации.

1. Перейдите в Google.
2. Например, если вы ищете в Google информацию о `garbage collection` с помощью обычного запроса, то обнаружите много рекламы услуг по сбору мусора в вашем районе, прежде чем увидите ссылку в «Википедии» о том, что такое сбор мусора в области компьютерных наук.
3. Повысить эффективность поиска можно, ограничив его полезным сайтом, например Stack Overflow, и удалив языки, которые могут быть неактуальны в момент поиска, такие как C++, Rust и Python, или добавив C# и .NET, как показано в следующем поисковом запросе:

```
garbage collection site:stackoverflow.com +C# -Java
```

Подписка на официальный блог .NET

Есть один отличный блог, на который рекомендую подписаться, чтобы быть в курсе новостей .NET. Его ведут группы разработчиков .NET. Доступен по адресу https://devblogs.microsoft.com/dotnet/.

Видеоблог Скотта Хансельмана

У Скотта Хансельмана из Microsoft есть отличный канал на YouTube, посвященный различным компьютерным фишкам: http://computerstufftheydidnteachyou.com/.

Рекомендую всем, кто так или иначе работает с компьютерами.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 1.1. Проверочные вопросы

Постарайтесь ответить на следующие вопросы. Обратите внимание: хотя большинство ответов можно найти в этой главе, для получения ответов на некоторые вопросы потребуется провести онлайн-исследования или написать код.

1. Среда разработки Visual Studio 2022 превосходит Visual Studio Code?
2. Платформа .NET 6 лучше .NET Framework?
3. Что такое .NET Standard и почему он все еще важен?
4. Почему на платформе .NET для разработки приложений программисты могут использовать разные языки, например C# и F#?
5. Как называется метод точки входа консольного приложения .NET и как его объявить?
6. Что такое программа верхнего уровня и как получить доступ к аргументам командной строки?
7. Что вы вводите в командной строке, чтобы создать и выполнить исходный код C#?
8. Каковы преимущества использования расширения .NET Interactive Notebooks для написания кода на языке C#?

9. Где найти справочную информацию по ключевому слову C#?
10. Где найти решения общих проблем программирования?

Приложение «Ответы на проверочные вопросы» вы найдете в конце книги.

Упражнение 1.2. Практическое задание

Вам не нужно устанавливать программу Visual Studio Code, Visual Studio 2022 для Windows или Mac для разработки на C#. Посетите сайт .NET Fiddle — dotnetfiddle.NET — и программируйте в онлайн-режиме.

Упражнение 1.3. Дополнительные ресурсы

На мой взгляд, эта книга содержит все фундаментальные знания и описание навыков, которыми должен владеть разработчик C# и .NET. Некоторые более сложные примеры включены в виде ссылок на документацию Microsoft или статьи сторонних авторов. Дополнительные материалы можно найти в репозитории GitHub для этой книги.

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-1---hello-c-welcome-net, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе мы с вами:

- настроили среду разработки;
- обсудили сходства и различия между современной .NET, .NET Core, .NET Framework, Xamarin и .NET Standard;
- научились пользоваться средами разработки Visual Studio Code с .NET SDK и Visual Studio 2022 для Windows и создали несколько простых консольных приложений;
- использовали .NET Interactive Notebooks, чтобы выполнить фрагменты кода в рамках обучения;
- узнали, как скачать код решения для данной книги из репозитория GitHub;
- и, самое главное, научились находить справочную информацию.

В следующей главе вы научитесь изъясняться на языке C#.

2 Говорим на языке C#

Данная глава посвящена основам языка программирования C#. Вы научитесь писать операторы, используя грамматику C#, а также познакомитесь с некоторыми общеупотребительными словарями, которыми будете пользоваться ежедневно. Уже к концу главы вы будете чувствовать себя уверенно, зная, как временно хранить информацию в памяти вашего компьютера и работать с ней.

В этой главе:

- введение в C#;
- основы языка C#, грамматика и терминология;
- работа с переменными;
- более детальное изучение консольных приложений.

Введение в C#

Этот раздел посвящен языку C# — грамматике и терминологии, которыми вы будете пользоваться каждый день при написании исходного кода своих приложений.

Языки программирования имеют много общего с человеческими, за исключением того, что в языках программирования вы можете создавать собственные слова, как доктор Сьюз!

В книге, написанной доктором Сьюзом в 1950 году, «Если бы у меня был свой зоопарк», есть такие строки:

Что дальше? Хочу без затей
Поймать просто невероятных зверей,
Каких вы представить себе не могли,
В стране под названием Гдетовдали!
Смотрите: Махлышка, Павлюн и Свердец,
Носульщик, Унылыщик, Цветец-Размышлец![1]

[1] Перевод с англ. Викентия Борисова. Источник: https://stihi.ru/. — *Здесь и далее примеч. пер.*

Обзор версий языка и их функций

Данный раздел посвящен языку программирования C# и написан скорее для начинающих, поэтому в нем рассматриваются основные темы, которые должны знать все разработчики: от объявления переменных до хранения данных и определения собственных пользовательских типов данных.

В этой книге представлены особенности языка C# от версии 1.0 до последней версии 10.0.

Если вы уже знакомы с устаревшими версиями C# и хотите узнать о новых функциях в самых последних версиях языка, то ниже я привел список версий и их важных новых функций, указав номер главы и название темы.

C# 1.0

Версия C# 1.0 была выпущена в 2002 году и включала в себя все важные функции статически типизированного объектно-ориентированного современного языка, как вы увидите в главах 2–6.

C# 2.0

Версия C# 2.0 вышла в 2005 году и была нацелена на обеспечение строгой типизации данных с использованием дженериков в целях повышения производительности кода и уменьшения количества ошибок типов, включая темы, перечисленные в табл. 2.1.

Таблица 2.1. Темы книги, относящиеся к C# 2.0

Особенность	Глава	Тема
Типы, допускающие значение `null`	6	Создание значимого типа, допускающего значение `null`
Дженерики (обобщения)	6	Создание типов, более пригодных для повторного использования с помощью дженериков

C# 3.0

Версия C# 3.0 вышла в 2007 году и была направлена на включение декларативного программирования с помощью *Language INtegrated Queries (LINQ)* и связанных с ним функций, таких как анонимные типы и лямбда-выражения, включая темы, перечисленные в табл. 2.2.

Таблица 2.2. Темы книги, относящиеся к C# 3.0

Особенность	Глава	Тема
Неявно типизированные локальные переменные	2	Определение типа локальной переменной
LINQ	11	Все темы, рассматривающиеся в главе 11

C# 4.0

Версия C# 4.0 была выпущена в 2010 году и фокусировалась на улучшении взаимодействия с динамическими языками, такими как F# и Python, включая темы, перечисленные в табл. 2.3.

Таблица 2.3. Темы книги, относящиеся к C# 4.0

Особенность	Глава	Тема
Динамические типы	2	Типы `dynamic`, их хранение
Именованные/необязательные аргументы	5	Необязательные параметры и именованные аргументы

C# 5.0

Версия C# 5.0 вышла в 2012 году и была направлена на упрощение поддержки асинхронных операций за счет автоматической реализации сложных конечных машин при написании того, что выглядит как синхронные операторы, включая тему из табл. 2.4.

Таблица 2.4. Тема книги, относящаяся к C# 5.0

Особенность	Глава	Тема
Упрощенные асинхронные методы	12	Знакомство с методами `async` и `await`

C# 6.0

Версия C# 6.0 вышла в 2015 году и была ориентирована на незначительные детали языка, включая темы, перечисленные в табл. 2.5.

C# 7.0

Версия C# 7.0 вышла в марте 2017 года и была направлена на добавление функциональных языковых возможностей, таких как кортежи и сопоставление с образцом, а также незначительные уточнения языка, включая темы, перечисленные в табл. 2.6.

Таблица 2.5. Темы книги, относящиеся к C# 6.0

Особенность	Глава	Тема
`static` (статический импорт)	2	Упрощение использования консоли
Интерполированные строки	2	Отображение вывода для пользователя
Члены класса с телами в виде выражений	5	Определение свойств только для чтения

Таблица 2.6. Темы книги, относящиеся к C# 7.0

Особенность	Глава	Тема
Двоичные литералы и разделители цифр	2	Хранение целых чисел
Сопоставление с образцом	3	Сопоставление с образцом с помощью оператора `if`
Переменные `out`	5	Управление передачей параметров
Кортежи	5	Объединение нескольких значений в кортежи
Локальные функции	6	Определение локальных функций

C# 7.1

Версия C# 7.1 вышла в августе 2017 года и была нацелена на незначительные изменения в языке, включая темы, перечисленные в табл. 2.7.

Таблица 2.7. Темы книги, относящиеся к C# 7.1

Особенность	Глава	Тема
Литеральные выражения по умолчанию	5	Установка значений полей с использованием литералов по умолчанию
Автоматически определяемые имена элементов кортежа	5	Вывод имен кортежей
Функция `async` Main	12	Сокращение времени реагирования для консольных приложений

C# 7.2

Версия C# 7.2 вышла в ноябре 2017 года и была нацелена на незначительные уточнения в языке, включая темы, перечисленные в табл. 2.8.

Таблица 2.8. Темы книги, относящиеся к C# 7.2

Особенность	Глава	Тема
Начальное подчеркивание в числовых литералах	2	Хранение целых чисел
Незавершающие именованные аргументы	5	Необязательные параметры и именованные аргументы
Модификатор доступа `private protected`	5	Описание модификаторов доступа
Операторы `==` и `!=` с кортежами	5	Сравнение кортежей

C# 7.3

Версия C# 7.3 была выпущена в мае 2018 года и фокусировалась на ориентированном на производительность безопасном коде, который совершенствует переменные `ref`, указатели и `stackalloc`. Они редко нужны большинству разработчиков, поэтому не рассматриваются в книге.

C# 8

Версия C# 8 вышла в сентябре 2019 года и была посвящена серьезному изменению языка, связанного с обработкой типов `null`, включая темы, перечисленные в табл. 2.9.

Таблица 2.9. Темы книги, относящиеся к C# 8

Особенность	Глава	Тема
Ссылочные типы, допускающие значение `null`	6	Создание ссылочного типа, допускающего значение `null`
Выражение `switch`	3	Упрощение операторов `switch` с помощью выражений `switch`
Методы интерфейса по умолчанию	6	Описание методов интерфейса по умолчанию

C# 9

Версия C# 9 вышла в ноябре 2020 года и была ориентирована на типы записей, уточнения сопоставления с образцом и консольные приложения с минимальным кодом, включая темы, перечисленные в табл. 2.10.

Таблица 2.10. Темы книги, относящиеся к C# 9

Особенность	Глава	Тема
Консольные приложения с минимальным кодом	1	Программы верхнего уровня
Целевой тип `new`	2	Использование нового целевого типа для создания экземпляров объектов
Улучшенное сопоставление с образцом	5	Сопоставление с образцом с помощью объектов
Записи	5	Работа с записями

C# 10

Версия C# 10 вышла в ноябре 2021 года и была ориентирована на уменьшение объема кода, необходимого для решения распространенных задач, включая темы, перечисленные в табл. 2.11.

Таблица 2.11. Темы книги, относящиеся к C# 10

Особенность	Глава	Тема
Импорт глобального пространства имен	2	Импорт пространств имен
Константные строковые литералы	2	Форматирование с использованием интерполированных строк
Пространства имен в файловой области	5	Упрощение объявлений пространств имен
Необходимые свойства	5	Требование установки свойств во время создания экземпляра
Работа со структурами	6	Работа с типами `struct`
Проверка на `null`	6	Проверка на `null` в параметрах метода

Стандарты C#

За прошедшие годы компания Microsoft представила для стандартизации несколько версий C# (табл. 2.12).

Таблица 2.12. Версии C#, представленные Microsoft для стандартизации

Версии C#	Стандарт ECMA	Стандарт ИСО/МЭК
1.0	ECMA-334:2003	ИСО/МЭК 23270:2003
2.0	ECMA-334:2006	ИСО/МЭК 23270:2006
5.0	ECMA-334:2017	ИСО/МЭК 23270:2018

Стандарт C# 6 находится в стадии разработки. Продолжается и работа над добавлением функций в C# 7. В 2014 году Microsoft сделала C# языком с открытым исходным кодом.

В настоящее время существует три общедоступных репозитория GitHub для максимально открытой работы над C# и связанными технологиями, как показано в табл. 2.13.

Таблица 2.13. Репозитории GitHub для работы над C# и связанными технологиями

Описание	Ссылка
Синтаксис языка C#	https://github.com/dotnet/csharplang
Реализация компилятора	https://github.com/dotnet/roslyn
Стандарт описания языка	https://github.com/dotnet/csharpstandard

Версии компилятора C#

Компиляторы языка .NET для языков C#, Visual Basic и F#, также известные как Roslyn, входят в состав пакета .NET SDK. Чтобы применить определенную версию C#, вы должны иметь установленную версию .NET SDK, одну из перечисленных в табл. 2.14.

Таблица 2.14. Версии компилятора C#

.NET SDK	Компилятор Roslyn	C#
1.0.4	2.0–2.2	7.0
1.1.4	2.3–2.4	7.1
2.1.2	2.6–2.7	7.2
2.1.200	2.8–2.10	7.3
3.0	3.0–3.4	8.0
5.0	3.8	9.0
6.0	3.9–3.10	10.0

При создании библиотек классов вы можете выбрать целевую платформу .NET Standard, а также версии современной .NET. Они содержат версии языка C# по умолчанию (табл. 2.15).

Таблица 2.15. Версии платформы .NET Standard и языка C# по умолчанию

.NET Standard	C#
2.0	7.3
2.1	8.0

Вывод версии SDK

Посмотрим на имеющиеся версии .NET SDK и компилятора языка C#.

1. В macOS выберите команду меню View ▸ Terminal (Вид ▸ Терминал). В Windows запустите оболочку командной строки.
2. Для определения имеющейся версии .NET SDK введите следующую команду:

```
dotnet --version
```

3. Обратите внимание, что версия на момент написания — 6.0.100, что указывает на то, что это начальная версия SDK без каких-либо исправлений ошибок или введения новых функций:

```
6.0.100
```

Включение версии компилятора на определенном языке

Инструменты разработчика, такие как Visual Studio и интерфейс командной строки `dotnet`, предполагают, что вы хотите применять последнюю основную версию компилятора языка C# по умолчанию. Поэтому до выпуска C# 8.0 версия C# 7.0 была последней основной и использовалась по умолчанию. Чтобы задействовать обновления доработанных версий C#, таких как 7.1, 7.2 или 7.3, вам необходимо добавить элемент конфигурации `<LangVersion>` в файл проекта следующим образом:

```
<LangVersion>7.3</LangVersion>
```

После выпуска C# 10.0 с .NET 6.0, если Microsoft выпустит компилятор версии C# 10.1 и вы захотите использовать его новые возможности, вам придется добавить элемент конфигурации в файл проекта следующим образом:

```
<LangVersion>10.1</LangVersion>
```

Потенциальные значения для `<LangVersion>` показаны в табл. 2.16.

Таблица 2.16. Потенциальные значения для <LangVersion>

LangVersion	Описание
7, 7.1, 7.2, 7.3, 8, 9, 10	При вводе определенного номера версии данный компилятор будет использоваться, если был установлен
`latestmajor`	Использует версию с наибольшим мажорным номером, например 7.0 в августе 2019 года, 8.0 в октябре 2019 года, 9.0 в ноябре 2020 года, 10.0 в ноябре 2021 года
`latest`	Использует версию с наибольшими мажорным и минорным номерами, например 7.2 в 2017 году, 7.3 в 2018 году, 8 в 2019 году, 10 в 2021 году
`preview`	Использует самую новую из доступных предварительных версий, например 10.0 в июле 2021 года с установленным .NET 6.0 Preview 6

Когда проект будет создан, вы можете отредактировать файл `.csproj` и добавить элемент `<LangVersion>`:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>

</Project>
```

Ваши проекты должны быть написаны с ориентиром на NET 6.0, чтобы использовать все возможности C# 10.

Если вы используете Visual Studio Code, но еще не установили расширение MSBuild project tools, то сделайте это сейчас. Вы получите IntelliSense при редактировании файлов .csproj, в том числе упростится добавление элемента <LangVersion> с соответствующими значениями.

Основы языка C#: грамматика и терминология

Чтобы изучить основы C#, вы можете использовать инструмент .NET Interactive Notebook, который избавляет от необходимости создавать приложения.

В процессе изучения возможностей языка вам нужно будет создать приложение. Простейший тип приложения — консольное.

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

Вывод версии компилятора

Напишем код, который выводит версию компилятора.

1. Если вы изучили главу 1, то у вас уже есть папка Code. Если нет, то создайте ее.
2. Откройте редактор кода и создайте консольное приложение с настройками, описанными ниже:
   1) шаблон проекта: Console Application [C#]/`console`;
   2) файл и папка рабочей области/решения: `Chapter02`;
   3) файл и папка проекта: `Vocabulary`.

Если вы забыли, как это делать, или не прочли главу 1, то вернитесь к ней, так как там приведены пошаговые инструкции по созданию рабочей области/решения с несколькими проектами.

3. Откройте файл `Program.cs` и в его начале под оператором `using` добавьте оператор, отображающий текущую версию C# как ошибку:

```
#error version
```

4. Запустите консольное приложение:
   1) в программе Visual Studio Code на панели TERMINAL (Терминал) введите команду `dotnet run`;
   2) в программе Visual Studio выберите команду меню Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки). Когда будет предложено продолжить и запустить последнюю успешную сборку, нажмите кнопку No (Нет).
5. Обратите внимание, что версия компилятора и языковая версия отображаются как номер сообщения компилятора об ошибке `CS8304` (рис. 2.1).

Рис. 2.1. Версия компилятора и языковая версия отображаются как номер сообщения компилятора об ошибке

6. В сообщении об ошибке в окне PROBLEMS (Проблемы) программы Visual Studio Code или в окне Error List (Перечень ошибок) программы Visual Studio будет указано `Compiler version: '4.0.0...'` с языковой версией `10.0`.
7. Закомментируйте оператор, вызывающий ошибку:

```
// #error version
```

8. Обратите внимание, что сообщения компилятора об ошибках исчезают.

Грамматика языка C#

К грамматике языка C# относятся *операторы* и *блоки*. Вы также можете оставлять пояснения к своему коду в виде комментариев.

Написание комментариев никогда не должно выступать в качестве единственного способа документирования вашего кода. Можно использовать и другие способы: выбирать понятные имена для переменных и функций, писать модульные тесты и создавать настоящие документы.

Операторы

В русском языке мы обозначаем конец повествовательного предложения точкой. Предложение может состоять из разного количества слов и фраз. Порядок слов в предложении тоже относится к правилам грамматики. К примеру, по-русски мы обычно говорим «черный кот», но в опреденных случаях допустимо сказать и «кот черный».

В английском языке прилагательное *«черный»* всегда предшествует существительному *«кот»*: the black cat. Во французском языке порядок иной, прилагательное указывается после существительного: le chat noir. Порядок имеет значение.

Язык C# обозначает конец *оператора* точкой с запятой. При этом оператор может состоять из нескольких *переменных* и *выражений*. Например, в указанном ниже операторе `totalPrice` — это переменная, а `subtotal + salesTax` — выражение:

```
var totalPrice = subtotal + salesTax;
```

Выражение состоит из операнда `subtotal`, операции `+` и второго операнда `salesTax`. Порядок операндов и операций имеет значение.

Комментарии

Чтобы описать предназначение кода, вы можете добавлять комментарии, предваряя их двумя символами косой черты: `//`. Компилятор игнорирует любой текст после этих символов и до конца строки, например:

```
// налог с продаж должен быть добавлен к промежуточной сумме
var totalPrice = subtotal + salesTax;
```

Для оформления многострочного комментария используйте символы `/*` в начале комментария и `*/` в его конце:

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

Хорошо спроектированный код, включая сигнатуры функций с правильно именованными параметрами и инкапсуляцией классов, в какой-то степени является самодокументируемым. Если в коде слишком много комментариев и пояснений, то спросите себя: можно ли его переписать, то есть провести рефакторинг, чтобы сделать более понятным, не используя длинные комментарии?

В редакторе кода есть инструменты, позволяющие добавлять и удалять символы комментариев:

- в программе Visual Studio 2022 для Windows выберите команды меню Edit ▸ Advanced ▸ Comment Selection (Правка ▸ Дополнительно ▸ Закомментировать выделенное) или Uncomment Selection (Раскомментировать выделенное);
- в программе Visual Studio Code выберите команды меню Edit ▸ Toggle Line Comment (Правка ▸ Переключить комментарий строки) или Toggle Block Comment (Переключить блок комментариев).

Код комментируют, добавляя описательный текст над оператором или после него. Вы можете закомментировать текст, добавляя символы комментария перед операторами или вокруг них, чтобы сделать их неактивными.

Раскомментирование означает удаление символов комментария.

Блоки

В русском языке мы обозначаем абзацы, начиная текст с новой строки. В языке C# *блоки* кода заключаются в фигурные скобки: `{}`.

Каждый блок начинается с объявления, описывающего то, что мы определяем. К примеру, блок может определять *пространство имен*, *класс*, *метод* или *оператор*, такой как `foreach`.

Более подробно о пространствах имен, классах и методах вы узнаете позднее в этой и последующих главах, а сейчас кратко познакомимся с некоторыми из следующих концепций:

- *пространство имен* содержит такие типы, как классы, для их группировки;
- *класс* содержит члены объекта, включая методы;
- *метод* содержит операторы, реализующие действие, которое может выполнять объект.

Примеры операторов и блоков

В шаблоне консольного приложения .NET 5.0 примеры операторов C# создаются программой. Я добавил несколько комментариев к операторам и блокам кода, как показано ниже:

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

namespace Basics
{ // открывающая фигурная скобка указывает на начало блока
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!"); // оператор
    }
  }
} // закрывающая фигурная скобка указывает на конец блока
```

Терминология языка C#

Терминологический словарь языка C# состоит из *ключевых слов*, *символов* и *типов*.

Среди предопределенных, зарезервированных ключевых слов можно выделить `using`, `namespace`, `class`, `static`, `int`, `string`, `double`, `bool`, `if`, `switch`, `break`, `while`, `do`, `for`, `foreach`, `and`, `or`, `not`, `record` и `init`.

А вот некоторые из символов: `"`, `'`, `+`, `-`, `*`, `/`, `%`, `@` и `$`.

Существуют и другие ключевые слова, имеющие особое значение только в определенном контексте.

В языке C# около 100 ключевых слов.

Сравнение языков программирования с естественными языками

В английском языке более 250 тысяч различных слов. Каким же образом языку C# удается обходиться лишь сотней ключевых слов? Более того, почему C# так трудно выучить, если он содержит всего 0,0416 % слов по сравнению с количеством в английском языке?

Одно из ключевых различий между человеческим языком и языком программирования заключается в том, что разработчики должны иметь возможность определять

новые «слова» с новыми значениями. Помимо примерно 100 ключевых слов на языке C#, эта книга научит вас некоторым из сотен тысяч «слов», определенных другими разработчиками, но вы также узнаете, как определять собственные.

Программисты во всем мире вынуждены изучать английский язык, поскольку большинство языков программирования основаны на латинице и используют английские слова, такие как namespace и class. Существуют языки программирования, использующие другие естественные языки, такие как арабский, но они считаются редкими. Если вам интересно узнать больше, то можете посмотреть видеоуроки на YouTube, демонстрирующие процесс программирования на арабском языке: https://youtu.be/dkO8cdwf6v8.

Изменение цветовой схемы синтаксиса

По умолчанию Visual Studio Code и Visual Studio выделяют ключевые слова C# синим цветом, чтобы их было легче отличить от другого кода. Вы можете изменить цветовую схему.

1. В программе Visual Studio Code выберите команду меню Code ▸ Preferences ▸ Color Theme (Код ▸ Параметры ▸ Цветовая схема) (находится в меню File (Файл) в операционной системе Windows).
2. Выберите цветовую схему. Для справки: я буду использовать цветовую схему Light+ (default light), чтобы снимки экрана, напечатанные в книге, выглядели четко.
3. В программе Visual Studio выберите команду меню Tools ▸ Options (Инструменты ▸ Параметры).
4. В диалоговом окне Options (Параметры) выберите пункт Fonts and Colors (Шрифты и цвета), а затем настройте параметры цвета элементов кода.

Помощь в написании правильного кода

Простые текстовые редакторы, например приложение Notepad (Блокнот), не помогут правильно писать по-английски. Точно так же это приложение не поможет вам написать правильный код на языке C#.

Программа Microsoft Word помогает писать без ошибок, подчеркивая ошибки волнистой линией: орфографические — красной (например, icecream следует писать ice-cream или ice cream), а грамматические — зеленой (например, если первое слово в предложении должно начинаться с прописной буквы).

Расширение C# для Visual Studio Code и Visual Studio тоже постоянно отслеживает, как вы набираете код, выделяя орфографические и грамматические ошибки

с помощью цветных волнистых линий. Например, имя метода `WriteLine` должно писаться с заглавной буквой `L`, а операторы — заканчиваться точкой с запятой.

Рассмотрим вышесказанное на примере.

1. В файле `Program.cs` замените в методе `WriteLine` прописную букву `L` на строчную.
2. Удалите точку с запятой в конце оператора.
3. В программе Visual Studio Code выберите команду меню View ▸ Problems (Вид ▸ Проблемы) или в программе Visual Studio выберите команду меню View ▸ Error List (Вид ▸ Перечень ошибок) и обратите внимание, что ошибки кода подчеркиваются красной волнистой линией, а также показывается подробная информация (рис. 2.2).

Рис. 2.2. Окно Error List (Перечень ошибок), отображающее две ошибки компиляции

4. Исправьте две ошибки в коде.

Импорт пространств имен

`System` — это пространство имен, как бы «адрес» для типа. Чтобы обратиться к кому-то лично, понадобится использовать код типа `Oxford.HighStreet.BobSmith`, сообщающий, что надо искать человека по имени Боб Смит на улице Хай-стрит в городе Оксфорд.

Строка `System.Console.WriteLine` сообщает компилятору, что следует искать метод `WriteLine` в типе `Console` в пространстве имен `System`. Чтобы упростить наш код, в шаблон проекта Console Application для каждой версии .NET до 6.0 в начале файла кода добавлен оператор, который дает компилятору указание всегда искать в пространстве имен `System` типы, не имеющие префикса их пространства имен:

```
using System; // импорт пространства имен System
```

Так выполняется *импорт пространства имен*. В результате все доступные типы в этом пространстве доступны для вашей программы без необходимости вводить префикс пространства и будут отображаться в IntelliSense во время написания кода.

В блокноты .NET Interactive большинство пространств имен импортируются автоматически.

Неявный и глобальный импорт пространств имен

Традиционно каждый файл `.cs`, которому необходимо импортировать пространства имен, должен начинаться с операторов `using` для импорта этих пространств. Пространства имен, такие как `System` и `System.Linq`, необходимы почти во всех файлах `.cs`, поэтому первые несколько строк каждого файла `.cs`, как правило, содержат несколько операторов `using`:

```
using System;
using System.Linq;
using System.Collections.Generic;
```

При создании сайтов и сервисов с помощью ASP.NET Core возникает необходимость импортировать в каждом файле десятки пространств имен.

В C# 10 представлены некоторые новые функции, упрощающие импорт пространств имен.

Во-первых, оператор `global using` позволяет импортировать пространство имен только в одном файле `.cs`, и оно будет доступно во всех файлах `.cs`. Вы можете поместить операторы `global using` в файл `Program.cs`, но я рекомендую создать для них отдельный файл, например, `GlobalUsings.cs` или `GlobalNamespaces.cs`.

```
global using System;
global using System.Linq;
global using System.Collections.Generic;
```

По мере того как разработчики привыкают к новой функции C#, я надеюсь, что одно соглашение об именах для этого файла станет стандартом.

Во-вторых, любые проекты .NET 6.0, использующие компилятор C# 10, для неявного глобального импорта некоторых общих пространств имен, таких как `System`, в папке obj создают файл `.cs`. Конкретный список неявно импортированных пространств имен зависит от того, какой SDK вы выбрали (табл. 2.17).

Таблица 2.17. SDK и неявно импортированные пространства имен

SDK	Неявный импорт пространства имен

Microsoft.NET.Sdk	`System`
	`System.Collections.Generic`
	`System.IO`
	`System.Linq`
	`System.NET.Http`
	`System.Threading`
	`System.Threading.Tasks`

Microsoft.NET.Sdk.Web	То же, что Microsoft.NET.Sdk, а также:
	`System.NET.Http.Json`
	`Microsoft.AspNetCore.Builder`
	`Microsoft.AspNetCore.Hosting`
	`Microsoft.AspNetCore.Http`
	`Microsoft.AspNetCore.Routing`
	`Microsoft.Extensions.Configuration`
	`Microsoft.Extensions.DependencyInjection`
	`Microsoft.Extensions.Hosting`
	`Microsoft.Extensions.Logging`

Microsoft.NET.Sdk.Worker	То же, что Microsoft.NET.Sdk, а также:
	`Microsoft.Extensions.Configuration`
	`Microsoft.Extensions.DependencyInjection`
	`Microsoft.Extensions.Hosting`
	`Microsoft.Extensions.Logging`

Рассмотрим автоматически сгенерированный файл с примерами неявного импорта.

1. На панели Solution Explorer (Обозреватель решений) выберите проект Vocabulary, нажмите кнопку Show All Files (Показать все файлы) и обратите внимание на созданные компилятором папки bin и obj.
2. Разверните папки obj, Debug и net6.0. Затем найдите и откройте файл `Vocabulary.GlobalUsings.g.cs`.
3. Обратите внимание, что этот файл автоматически создается компилятором для проектов .NET 6.0 и что в нем импортируются некоторые часто используемые пространства имен, включая `System.Threading`:

```
// <autogenerated />
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
```

```
global using global::System.NET.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

4. Закройте файл `Vocabulary.GlobalUsings.g.cs`.
5. На панели Solution Explorer (Обозреватель решений) выберите проект, а затем добавьте в его файл дополнительные пространства имен, которые импортируются неявным образом:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <Using Remove="System.Threading" />
    <Using Include="System.Numerics" />
  </ItemGroup>

</Project>
```

6. Сохраните изменения в файле проекта.
7. Разверните папки obj, Debug и net6.0. Затем найдите и откройте файл `Vocabulary.GlobalUsings.g.cs`.
8. Обратите внимание, что в данном файле теперь импортируется пространство `System.Numerics` вместо `System.Threading`, как показано ниже (выделено жирным шрифтом):

```
// <autogenerated />
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.NET.Http;
global using global::System.Threading.Tasks;
global using global::System.Numerics;
```

9. Закройте файл `Vocabulary.GlobalUsings.g.cs`.

Для всех пакетов SDK вы можете отключить функцию неявно импортированных пространств имен, удалив элемент `<ImplicitUsings>` из файла проекта или изменив его значение на `disable`, как показано в следующей разметке::

```
<ImplicitUsings>enable</ImplicitUsings>
```

Глаголы = методы

В русском языке глаголы используются для описания действия. В языке C# действия называются *методами*, и их в нем доступны буквально сотни тысяч. В русском языке глаголы меняют форму в зависимости от времени: действие происходило, происходит или будет происходить. К примеру, Амир *попрыгал* в прошлом, Бет *прыгает* в настоящем, они *прыгали* в прошлом и Чарли *будет прыгать* в будущем.

В языке C# вызов или выполнение методов, таких как `WriteLine`, различаются в зависимости от специфики действия. Это так называемая *перегрузка*, которую мы более подробно изучим в главе 5. Рассмотрим пример:

```
// выводит текущий символ конца строки
// по умолчанию это возврат каретки и перевод строки
Console.WriteLine();

// выводит приветствие и возврат каретки
Console.WriteLine("Hello Ahmed");

// выводит отформатированное число и дату и возврат каретки
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, переменные и свойства

В русском языке существительные — это названия предметов или живых существ. К примеру, Шарик — имя (кличка) собаки. Словом «собака» называется тип живого существа с именем Шарик. Скомандовав Шарику 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*, *переменные* и *свойства*. Например:

- `Animal` и `Car` — типы, то есть существительные для категоризации предметов;
- `Head` и `Engine` — поля или свойства, то есть существительные, которые принадлежат `Animal` и `Car`;
- `Fido` и `Bob` — переменные, то есть существительные, относящиеся к конкретному предмету.

Для языка C# доступны десятки тысяч типов. Обратите внимание: я сказал не «В языке C# доступны десятки тысяч типов». Разница едва уловима, но очень

важна. Язык C# содержит лишь несколько ключевых слов для типов данных, таких как `string` и `int`. Строго говоря, C# не определяет какие-либо типы. Ключевые слова наподобие `string` определяют типы как *псевдонимы*, представляющие типы на платформе, на которой запускается C#.

C# не может 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, предоставляющая десятки тысяч типов для C#. Они включают `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 такая игра, *«20 вопросов»*, в которой первым делом спрашивают о категории 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). Что они означают, вы узнаете в главе 6. Например, в языке C# ключевое слово `string` — это `class` (класс), но `int` — это `struct` (структура). Поэтому лучше использовать термин *«тип»* для обозначения их обоих.

Определение объема словаря C#

Мы знаем, что в языке C# более 100 ключевых слов, но сколько в нем типов? В нашем простом консольном приложении мы напишем код, позволяющий подсчитать количество типов и методов, доступных в языке C#.

Не волнуйтесь, если не понимаете, как работает этот код. В нем используется техника под названием *«отражение»*.

1. Начнем с импорта пространства имен `System.Reflection` в начале файла `Program.cs`:

   ```
   using System.Reflection;
   ```

2. Удалите оператор, выводящий текст `Hello World!`, и замените его кодом, показанным ниже:

   ```
   Assembly? assembly = Assembly.GetEntryAssembly();
   if (assembly == null) return;

   // перебор сборок, на которые ссылается приложение
   foreach (AssemblyName name in assembly.GetReferencedAssemblies())
   {
     // загрузка сборки для чтения данных
     Assembly a = Assembly.Load(name);
   ```

```
  // объявление переменной для подсчета количества методов
  int methodCount = 0;

  // перебор всех типов в сборке
  foreach (TypeInfo t in a.DefinedTypes)
  {
    // добавление количества методов
    methodCount += t.GetMethods().Count();
  }

  // вывод количества типов и их методов
  Console.WriteLine(
    "{0:N0} types with {1:N0} methods in {2} assembly.",
    arg0: a.DefinedTypes.Count(),
    arg1: methodCount, arg2: name.Name);
}
```

3. Вы увидите следующий вывод, в котором отображается фактическое количество типов и методов, доступных вам в простейшем приложении при работе в операционной системе macOS. Количество отображаемых типов и методов может различаться в зависимости от используемой операционной системы, как показано ниже:

```
// Вывод в Windows
0 types with 0 methods in System.Runtime assembly.
106 types with 1,126 methods in System.Linq assembly.
44 types with 645 methods in System.Console assembly.

// Вывод в macOS
0 types with 0 methods in System.Runtime assembly.
103 types with 1,094 methods in System.Linq assembly.
57 types with 701 methods in System.Console assembly.
```

Почему сборка System.Runtime содержит нулевые типы? Эта сборка особенная, поскольку содержит только средства переадресации типов, а не фактические типы. Переадресатор типов представляет собой тип, реализованный вне .NET или другим способом.

4. Добавьте операторы в начало файла после импорта пространства имен, чтобы объявить некоторые переменные:

```
using System.Reflection;

// объявление некоторых неиспользуемых переменных
// с помощью типов в дополнительных сборках
System.Data.DataSet ds;
HttpClient client;
```

Из-за объявления переменных, использующих типы в других сборках, эти сборки загружаются с приложением, что позволяет коду видеть все типы и методы в них. Компилятор предупредит вас о наличии неиспользуемых переменных, но это не остановит выполнение вашего кода.

5. Снова запустите консольное приложение и проанализируйте результаты. Они должны выглядеть примерно так:

```
// Вывод в Windows
0 types with 0 methods in System.Runtime assembly.
383 types with 6,854 methods in System.Data.Common assembly.
456 types with 4,590 methods in System.NET.Http assembly.
106 types with 1,126 methods in System.Linq assembly.
44 types with 645 methods in System.Console assembly.

// Вывод в macOS
0 types with 0 methods in System.Runtime assembly.
376 types with 6,763 methods in System.Data.Common assembly.
522 types with 5,141 methods in System.NET.Http assembly.
103 types with 1,094 methods in System.Linq assembly.
57 types with 701 methods in System.Console assembly.
```

Надеюсь, теперь вы понимаете, почему изучение языка C# — достаточно сложная задача. Типов и методов множество, причем методы — только одна категория членов, которую может иметь тип, а вы и другие программисты постоянно определяют новые типы и члены!

Работа с переменными

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

Обычно данные поступают в программы из файлов, баз данных или через пользовательский ввод. Данные могут быть на время помещены в переменные, которые хранятся в памяти работающей программы. Когда она завершается, данные стираются из памяти. Данные обычно выводятся в файлы и базы, 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.

Вы можете управлять этими характеристиками, выбрав определенный тип. Простые распространенные типы, например `int` и `double`, можно представить как хранилища разного размера, при этом менее объемное займет меньше памяти, но может быть обработано не очень быстро. Например, добавление 16-битных чисел

может обрабатываться не так быстро, как добавление 64-битных чисел в 64-битной операционной системе. Одни хранилища можно разместить поближе, чтобы иметь к ним быстрый доступ, а другие — убрать подальше в большое хранилище.

Присвоение имен и значений

Теперь обсудим соглашения об именовании, которым рекомендуется следовать. Взгляните на табл. 2.18.

Таблица 2.18. Соглашения об именовании

Правило	Примеры	Использование
Верблюжий регистр	`cost`, `orderDetail`, `dateOfBirth`	Локальные переменные и закрытые поля
Прописной стиль (он же стиль Pascal)	`String`, `Int32`, `Cost`, `DateOfBirth`, `Run`	Имена типов, открытые поля и другие члены, такие как методы

Следование набору соглашений об именовании позволит другим разработчикам (и вам самим в будущем!) легко понять ваш код.

Блок кода, показанный ниже, отражает пример объявления и инициализации локальной переменной путем присвоения ей значения с символом `=`. Обратите внимание: вы можете вывести имя переменной, используя ключевое слово `nameof`, появившееся в версии C# 6.0:

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

Сообщение в двойных кавычках в этом фрагменте кода было перенесено на вторую строку, поскольку страница книги слишком узкая. При вводе подобных операторов в окне редактора кода набирайте их в одну строку.

Литеральные значения

При присвоении значения переменной часто используются *литеральные* значения. Что это такое? Литералами обозначаются фиксированные значения. Типы данных используют разные обозначения для их литеральных значений, и в следующих нескольких разделах вы увидите примеры того, как с помощью литеральных нотаций присваивать значения переменным.

Хранение текста

При работе с текстом отдельная буква, например `A`, сохраняется как тип `char`.

В действительности все сложнее. Для представления египетского иероглифического письма A002 (U+13001) требуются два значения System.Char (известные как суррогатные пары): \uD80C и \uDC01. Не всегда верно, что один символ равен одной букве. В противном случае вам не избежать проблем с работой ваших приложений.

Тип `char` задается с использованием *одинарных* кавычек, оборачивающих литеральное значение, или путем присвоения возвращаемого значения при фиктивном вызове функции:

```
char letter = 'A'; // присваивание литеральных символов
char digit = '1';
char symbol = '$';
char userChoice = GetSomeKeystroke(); // присваивание из фиктивной функции
```

Если же используется последовательность символов, например слово `Bob`, то такое значение сохраняется как тип `string` и задается с использованием *двойных* кавычек, оборачивающих литеральное значение, или путем присвоения возвращаемого значения при фиктивном вызове функции:

```
string firstName = "Bob"; // присваивание литеральных строк
string lastName = "Smith";
string phoneNumber = "(215) 555-4256";

// присваивание строки, возвращаемой фиктивной функцией
string address = GetAddressFromDatabase(id: 563);
```

Дословные литеральные строки

При сохранении текста в переменной `string` вы можете включить escape-последовательности, которые представляют собой специальные символы, такие как табуляции и новые строки. Это можно сделать с помощью обратного слеша:

```
string fullNameWithTabSeparator = "Bob\tSmith";
```

Но что будет, если вы сохраняете путь к файлу в операционной системе Windows и одно из имен папок начинается с буквы `T`?

```
string filePath = "C:\televisions\sony\bravia.txt";
```

Компилятор преобразует `\t` в символ табуляции и вы получите ошибку!

Вам необходимо использовать префикс `@`, чтобы использовать *дословную (verbatim)* литеральную строку, как показано ниже:

```
string filePath = @"C:\televisions\sony\bravia.txt";
```

Подытожим:

- *литеральная строка* — символы, заключенные в двойные кавычки. Они могут использовать escape-символы, такие как `\t` для табуляции. Чтобы вывести обратный слеш, используйте два символа `\\`;
- *дословная литеральная строка* — литеральная строка с префиксом `@` для отключения управляющих символов. Вдобавок позволяет строковому значению занимать несколько строк, поскольку пробелы рассматриваются как отдельные символы, а не как инструкции для компилятора;
- *интерполированная строка* — литеральная строка с префиксом `$` для включения встроенных форматированных переменных. Вы узнаете больше об этом позже в текущей главе.

Хранение чисел

Числа — данные, которые можно использовать для выполнения арифметических операций, к примеру умножения. Телефонный номер — это не число. Чтобы определиться, какое значение присваивать переменной: числовое или нет, — спросите себя, нужно ли вам умножать два телефонных номера или использовать в значении специальные символы, например, так: (414) 555-1234. В этих случаях число представляет собой последовательность символов, поэтому должно храниться как строка.

Числа могут быть *натуральными* (например, 42) и использоваться для подсчета, а также отрицательными (например, –42). Это *целые числа*. Кроме того, числа могут быть *вещественными*, иначе называемыми *действительными* (например, 3,9 (с дробной частью)). В программировании они представлены числами *одинарной* и *двойной точности с плавающей запятой*.

Рассмотрим на примере чисел.

1. Откройте редактор кода и создайте консольное приложение `Numbers` в рабочей области/решении `Chapter02`:
   1) в программе Visual Studio Code выберите `Numbers` в качестве активного проекта OmniSharp. Если появится предупреждение о том, что необходимые ресурсы отсутствуют, то нажмите кнопку `Yes` (Да), чтобы их добавить;
   2) в программе Visual Studio настройте стартовый проект в соответствии с текущим выбором.

2. В файле `Program.cs` удалите существующий код, а затем введите операторы для объявления нескольких числовых переменных, используя различные типы данных, как показано ниже:

```
// целое число без знака означает положительное целое число или 0
uint naturalNumber = 23;

// целое число означает отрицательное или положительное целое число или 0
int integerNumber = -23;

// float означает число одинарной точности с плавающей запятой
// суффикс F указывает, что это литерал типа float
float realNumber = 2.3F;

// double означает число двойной точности с плавающей запятой
double anotherRealNumber = 2.3; // литерал типа double
```

Хранение целых чисел

Возможно, вы знаете, что компьютеры хранят все в виде битов. Значение бита равно 0 или 1. Это называется *двоичной системой счисления*. Люди используют *десятичную систему счисления*.

Система десятичных чисел, также известная как Base 10, имеет 10 в качестве *основы*, то есть десять цифр от 0 до 9. Эта система чисел чаще всего используется человеком, однако в науке, инженерии и вычислительной технике популярны и другие системы счисления. Система двоичных чисел, также известная как Base 2, имеет 2 в качестве основы, что означает наличие двух цифр: 0 и 1.

В табл. 2.19 показано, как компьютеры хранят число `10` в двоичной системе счисления. Обратите внимание на бит со значением `1` в столбцах 8 и 2; 8 + 2 = 10.

Таблица 2.19. Хранение целых чисел

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`.

Улучшение читабельности с помощью разделителей разрядов чисел

Из двух новых возможностей версии C# 7.0 и более поздних версий одна касается использования символа подчеркивания (_) в качестве разделителя групп разрядов чисел, а вторая внедряет поддержку двоичных литералов.

Вы можете использовать символ подчеркивания в любых числовых литералах, включая десятичную, двоичную или шестнадцатеричную систему счисления.

Например, вы можете записать значение для миллиона в десятичной системе (Base 10) в виде `1_000_000`.

Вы даже можете использовать распространенный в Индии принцип 2/3: 10_00_000.

Записи в двоичной системе

Чтобы использовать запись в двоичной системе (Base 2), задействуя только 1 и 0, начните числовой литерал с `0b`. Чтобы применить запись в шестнадцатеричной системе (Base 16), используя от 0 до 9 и от A до F, начните числовой литерал с `0x`.

Целые числа

Рассмотрим несколько примеров.

1. В файле `Program.cs` введите следующий код, чтобы объявить некоторые числовые переменные, используя знак подчеркивания в качестве разделителя:

```
// три переменные, которые хранят число 2 миллиона
int decimalNotation = 2_000_000;
int binaryNotation = 0b_0001_1110_1000_0100_1000_0000;
int hexadecimalNotation = 0x_001E_8480;

// убедитесь, что три переменные имеют одинаковое значение
// оба оператора выводят true
Console.WriteLine($"{decimalNotation == binaryNotation}");
Console.WriteLine(
  $"{decimalNotation == hexadecimalNotation}");
```

2. Запустите код и обратите внимание, что в результате все три числа совпадают:

```
True
True
```

Компьютеры всегда могут точно представлять целые числа, используя тип `int` или один из его родственных типов, например `long` и `short`.

Хранение вещественных чисел

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

Большинство языков программирования реализуют стандарт IEEE для арифметики с плавающей запятой. IEEE 754 — это технический стандарт для арифметики с плавающей запятой, установленный в 1985 году Институтом инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers, IEEE).

В табл. 2.20 показано, как компьютер хранит число `12.75` в двоичной системе счисления. Обратите внимание на бит со значением `1` в столбцах 8, 4, 1/2 и 1/4.

8 + 4 + 1/2 + 1/4 = 12 3/4 = 12,75.

Таблица 2.20. Хранение вещественных чисел

128	64	32	16	8	4	2	1	.	1/2	1/4	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 видите, число `12.75` может быть точно представлено в двоичной системе. Однако такое возможно не для всех чисел. И скоро вы в этом убедитесь.

Пример работы с числами

В языке C# есть *операция* `sizeof()`, возвращающая количество байтов, используемых в памяти данным типом. Некоторые типы имеют члены с именами `MinValue` и `MaxValue`, возвращающие минимальные и максимальные значения, которые могут храниться в переменной этого типа. Теперь мы собираемся с помощью этих функций создать консольное приложение для изучения типов чисел.

1. В файле `Program.cs` введите операторы для отображения размера трех числовых типов данных:

   ```
   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}.");
   ```

   Ширина печатных страниц в этой книге заставляет переносить строковые значения (в двойных кавычках) на несколько строк. Вам необходимо ввести их в одну строку, иначе вы получите ошибки компиляции.

2. Запустите код и проанализируйте результат (рис. 2.3).

Рис. 2.3. Информация о числовых типах данных

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

Почему переменная `double` может хранить бóльшие значения, чем переменная `decimal`, но при этом задействует вполовину меньше памяти? Разберемся!

Сравнение типов double и decimal

Теперь вы напишете код, чтобы сравнить типы `double` и `decimal`. Не беспокойтесь, если пока не понимаете синтаксис, хотя он и не слишком сложный.

1. Введите операторы для объявления двух переменных `double`, суммируйте их и сравните с ожидаемым результатом, а затем запишите результат в консоль, как показано ниже:

```
Console.WriteLine("Using doubles:");
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}");
}
```

2. Запустите код и проанализируйте результат:

```
Using doubles:
0.1 + 0.2 does NOT equal 0.3
```

В тех регионах, где для десятичного разделителя используется запятая, результат будет выглядеть немного иначе:

```
0,1 + 0,2 does NOT equal 0,3
```

Тип `double` не обеспечивает точность, поскольку некоторые числа, например `0.1`, не могут быть представлены как значения с плавающей запятой.

Используйте тип `double` только если точность неважна, особенно при сравнении двух чисел; к примеру, при измерении роста человека. Необходимо также сравнивать только значения, используя операторы «больше чем» и «меньше чем», но не «равно».

Проблема в предыдущем коде заключается в том, как компьютер хранит число `0.1` или кратное ему. Чтобы представить `0.1` в двоичном формате, компьютер хранит `1` в столбце 1/16, `1` в столбце 1/32, `1` в столбце 1/256, `1` в столбце 1/512 и т. д.

Число `0.1` в десятичной системе представлено как `0.00011001100110011`... с бесконечным повторением (табл. 2.21).

Таблица 2.21. Сравнение типов double и decimal

4	2	1	.	1/2	1/4	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 Р-17 и та попала в американские казармы в городе Дхарам, в результате чего погибли 28 американских солдат; более подробно можно прочитать на сайте https://www.ima.umn.edu/~arnold/disasters/patriot.html.

1. Скопируйте и операторы, которые вы написали ранее (использовавшие переменные `double`).
2. Затем измените операторы, чтобы они использовали числа типа `decimal`, и переименуйте переменные в `c` и `d`:

```
Console.WriteLine("Using decimals:");
decimal c = 0.1M; // суффикс M обозначает десятичное литеральное значение
decimal d = 0.2M;

if (c + d == 0.3M)
{
  Console.WriteLine($"{c} + {d} equals {0.3M}");
}
else
{
  Console.WriteLine($"{c} + {d} does NOT equal {0.3M}");
}
```

3. Запустите код и проанализируйте результат:

```
Using decimals:
0.1 + 0.2 equals 0.3
```

Тип `decimal` точен, поскольку хранит значение как большое целое число и смещает десятичную запятую. К примеру, 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.

Тип int используйте для натуральных чисел, а double — для вещественных. Тип decimal применяйте для денежных расчетов, измерений в чертежах и машиностроительных схемах и повсюду, где важна точность вещественных чисел.

Типу `double` присущи некоторые полезные специальные значения. Так, `double.NaN` представляет значение, не являющееся числом (например, деление на ноль), `double.Epsilon` — наименьшее положительное число, которое может быть сохранено как значение `double`, а `double.PositiveInfinity` и `double.NegativeInfinity` — бесконечно большое положительное и отрицательное значения.

Хранение логических значений

Логическое значение может содержать только одно из двух литеральных значений: или `true` (истина), или `false` (ложь), как показано в этом коде:

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

Логические значения чаще всего используются при ветвлении и зацикливании, как вы увидите в главе 3.

Хранение объектов любого типа

Специальный тип `object` позволяет хранить данные любого типа, но такая гибкость требует жертв: код получается более сложным и менее производительным. Поэтому по возможности вы должны избегать использования данного типа. Ниже показано, как применять эти типы, если они вам нужны.

1. Откройте редактор кода и создайте консольное приложение `Variables` в рабочей области/решении `Chapter02`.
2. В программе Visual Studio Code выберите `Variables` в качестве активного проекта OmniSharp. Когда вы увидите всплывающее предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку Yes (Да), чтобы добавить их.
3. В файле `Program.cs` введите операторы для объявления и использования некоторых переменных, задействуя тип `object`:

```
object height = 1.88; // хранение double в объекте
object name = "Amir"; // хранение string в объекте
Console.WriteLine($"{name} is {height} metres tall.");

int length1 = name.Length; // Выдаст ошибку компиляции!
int length2 = ((string)name).Length; // сообщаем компилятору, что это
строка
Console.WriteLine($"{name} has {length2} characters.");
```

4. Запустите код и обратите внимание, что четвертый оператор не может скомпилироваться, поскольку тип данных переменной `name` неизвестен компилятору (рис. 2.4).

Рис. 2.4. Тип object не имеет свойства Length

5. Добавьте двойную косую черту комментария в начало оператора, который не может скомпилироваться, чтобы «закомментировать» его и сделать неактивным.
6. Снова запустите код с помощью команды `dotnet run` и обратите внимание, что компилятор может получить доступ к длине строки, если программист явно

сообщает компилятору, что переменная `object` содержит строку, добавляя префикс с выражением приведения наподобие `(string)`:

```
Amir is 1.88 metres tall.
Amir has 4 characters.
```

Тип `object` доступен с самой первой версии языка C#, но в версии C# 2.0 и более поздних в качестве альтернативы используются более эффективные *дженерики* (мы рассмотрим их в главе 6), которые обеспечивают желаемую гибкость, не снижая производительность.

Хранение данных динамического типа

Существует еще один специальный тип, `dynamic`, который тоже позволяет хранить данные любого типа и, подобно типу `object`, делает это за счет производительности. Ключевое слово `dynamic` было введено в версии C# 4.0. В отличие от типа `object`, для значения, хранящегося в такой переменной, можно вызывать его члены без явного приведения. Воспользуемся типом `dynamic`.

1. Добавьте операторы для объявления переменной `dynamic` и присвойте литеральное значение `string`. Затем присвойте целочисленное значение и массив целочисленных значений:

```
// хранение строки в объекте dynamic
// строка имеет свойство Length
dynamic something = "Ahmed";

// int не имеет свойства Length
// something = 12;

// массив любого типа имеет свойство Length
// something = new[] { 3, 5, 7 };
```

2. Добавьте оператор, позволяющий получить длину переменной `dynamic`:

```
// компилируется, но может вызвать исключение во время
// выполнения, если вы позже сохраните тип данных,
// у которого нет свойства Length
Console.WriteLine($"Length is {something.Length}");
```

3. Запустите код и обратите внимание, что он работает, так как значение `string` имеет свойство `Length`:

```
Length is 5
```

4. Раскомментируйте оператор, присваивающий значение `int`.

5. Запустите код и обратите внимание на ошибку времени выполнения, поскольку значение типа `int` не имеет свойства `Length`:

```
Unhandled exception. Microsoft.CSharp.RuntimeBinder.
RuntimeBinderException: 'int' does not contain a definition for 'Length'
```

6. Раскомментируйте оператор, присваивающий массив.
7. Запустите код и обратите внимание на вывод, поскольку массив из трех значений `int` имеет свойство `Length`:

```
Length is 3
```

Ограничения типа `dynamic` заключаются в том, что редакторы кода не отображают меню IntelliSense, призванное помогать при наборе кода. Это связано с тем, что компилятор не может проверить тип во время сборки. Вместо этого общеязыковая исполняющая среда проверяет член во время выполнения и выдает исключение, если тот отсутствует.

Отследить нарушения, возникшие во время выполнения, помогают исключения. Более подробную информацию о них и о том, как с ними обращаться, вы получите в главе 3.

Объявление локальных переменных

Локальные переменные объявляются внутри методов и существуют лишь во время выполнения последних. Как только метод возвращается, память, выделенная для хранения любых локальных переменных, освобождается.

Строго говоря, типы значений освобождаются, а ссылочные типы должны ожидать сборки мусора. В чем разница между типами значений и ссылочными типами, вы узнаете в главе 6.

Определение типа локальной переменной

Рассмотрим локальные переменные, объявленные с определенными типами и с использованием определения типов.

1. Введите операторы для объявления и присвоения значений некоторым локальным переменным, используя определенные типы, как показано ниже:

```
int population = 66_000_000; // 66 миллионов человек в Великобритании
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
```

Зеленой волнистой линией (в зависимости от выбранного редактора кода и цветовой схемы) будут подчеркнуты имена переменных, значения которым присвоены, но нигде не используются.

Вывод типа локальной переменной

Вы можете использовать ключевое слово `var` для объявления локальных переменных. Компилятор определит тип данных по значению, введенному вами после операции присваивания, `=`.

Числовой литерал без десятичной запятой определяется как переменная `int`, если не добавлен следующий суффикс:

- `L` означает `long`;
- `UL` означает `ulong`;
- `M` означает `decimal`;
- `D` означает `double`;
- `F` означает `float`.

Числовой литерал с десятичной запятой определяется как `double`. Если добавить суффикс `M`, то определяется как переменная `decimal`; если `F` — то как переменная `float`.

Двойные кавычки обозначают переменную `string`, а одинарные — переменную `char`. Значения `true` и `false` определяют тип `bool`.

1. Измените операторы так, чтобы использовать ключевое слово `var`:

```
var population = 66_000_000; // 66 миллионов человек в Великобритании
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
```

2. Наведите указатель мыши на каждое из ключевых слов `var` и обратите внимание, что в редакторе кода отображается всплывающая подсказка с информацией о выведенном типе.
3. В начале файла класса импортируйте пространство имен для работы с XML, чтобы можно было объявлять некоторые переменные, используя типы в данном пространстве:

```
using System.Xml;
```

Если вы используете инструмент .NET Interactive Notebooks, добавьте операторы using в отдельную ячейку кода выше той, в которой вы пишете основной код. Затем нажмите кнопку Execute Cell (Выполнить ячейку), чтобы убедиться, что пространства имен импортированы. После этого они будут доступны в последующих ячейках кода.

4. Под предыдущими операторами добавьте операторы для создания новых объектов:

```
// удачное применение var, поскольку он избегает повторного типа,
// как показано во втором более подробном операторе
var xml1 = new XmlDocument();
XmlDocument xml2 = new XmlDocument();

// неудачное применение var, поскольку мы не можем определить тип,
// поэтому должны использовать конкретное объявление типа,
// как показано во втором операторе
var file1 = File.CreateText("something1.txt");
StreamWriter file2 = File.CreateText("something2.txt");
```

Несмотря на несомненное удобство ключевого слова var, умные программисты стараются избегать его, чтобы обеспечить читабельность кода и определение типов на глаз. Что касается меня, то я использую это ключевое слово, только если тип и без того ясен. Например, в коде, показанном выше, первый оператор так же понятен и ясен, как и второй, в котором указывается тип переменной xml, но первый короче. Тем не менее третий оператор не совсем ясно показывает тип переменной file, поэтому лучше использовать четвертый вариант, поскольку он показывает, что тип — StreamWriter. Если сомневаетесь, то уточните!

Создание экземпляров объектов с помощью целевого типа выражения new

В C# 9 Microsoft для создания экземпляров объектов представила другой синтаксис, известный как *целевой тип выражения new*. При создании экземпляра объекта вы можете сначала указать тип, а затем использовать `new`, не повторяя тип:

```
XmlDocument xml3 = new(); // целевой тип выражения new (версия C# 9)
```

Если у вас есть тип с полем или свойством, которое необходимо установить, то тип можно вывести:

```
class Person
{
  public DateTime BirthDate;
}

Person kim = new();
kim.BirthDate = new(1967, 12, 26); // вместо new DateTime(1967, 12, 26)
```

Создание экземпляров объектов с помощью целевого типа выражения new не поддерживается компиляторами C# версии ниже 9. Далее в этой книге я использовал целевой тип выражения new. Пожалуйста, дайте мне знать, если я что-то пропустил!

Получение и определение значений по умолчанию для типов

Большинство примитивных типов, кроме `string`, представляют собой *типы значений*. Это значит, им должны присваиваться значения. Вы можете определить значение по умолчанию типа, используя операцию `default()` и передав тип в качестве параметра. Вы можете установить значение типа по умолчанию, используя ключевое слово `default`.

Тип `string` является *ссылочным*. Это значит, что переменные `string` содержат адрес памяти значения, а не само значение. Переменная ссылочного типа может иметь значение `null`, которое считается литералом, указывающим, что переменная не ссылается ни на что (пока). Значение `null` является значением по умолчанию для всех ссылочных типов.

Более подробную информацию о типах значений и ссылочных типах вы найдете в главе 6.

Рассмотрим значения по умолчанию.

1. Добавьте операторы для отображения значений по умолчанию `int`, `bool`, `DateTime` и `string`:

```
Console.WriteLine($"default(int) = {default(int)}");
Console.WriteLine($"default(bool) = {default(bool)}");
Console.WriteLine($"default(DateTime) = {default(DateTime)}");
Console.WriteLine($"default(string) = {default(string)}");
```

2. Запустите код и проанализируйте результат. Обратите внимание, что ваш вывод для даты и времени может быть отформатирован по-другому, если вы не запускаете его в Великобритании, и что значения `null` не выводятся как пустая строка:

```
default(int) = 0
default(bool) = False
default(DateTime) = 01/01/0001 00:00:00
default(string) =
```

3. Добавьте операторы для объявления числа. Присвойте значение, а затем сбросьте его до значения по умолчанию:

```
int number = 13;
Console.WriteLine($"number has been set to: {number}");
number = default;
Console.WriteLine($"number has been reset to its default: {number}");
```

4. Запустите код и проанализируйте результат:

```
number has been set to: 13
number has been reset to its default: 0
```

Хранение нескольких значений в массиве

Если вам нужно сохранить несколько значений одного и того же типа, можете объявить *массив*. Например, вы можете сделать это, когда вам нужно сохранить четыре имени в массиве `string`.

Код, показанный ниже, объявляет массив для хранения четырех значений `string`. Затем в нем сохраняются значения `string` с индексами позиций от 0 до 3 (индексация массивов ведется с нуля, поэтому последний элемент всегда на единицу меньше, чем длина массива).

Вы заблуждаетесь, если думаете, что элементы всех массивов индексируются с нуля. В .NET наиболее распространен szArray, одномерный массив с индексацией с 0 и привычным синтаксисом []. Но в .NET существует и многомерный массив mdArray, элементы которого не обязательно должны индексироваться с 0. Они редко используются, но вам следует знать, что они существуют.

И наконец, выполняется перебор каждого элемента в массиве с помощью оператора `for`, о чем мы более подробно поговорим в главе 3.

Рассмотрим пример использования массива.

1. Добавьте операторы для объявления и использования массива значений `string`, как показано ниже:

```
string[] names; // может ссылаться на любой по размеру массив строк

// выделение памяти для четырех строк в массиве
names = new string[4];

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

// перебор имен
for (int i = 0; i < names.Length; i++)
{
  // вывести элемент в позиции индекса i
  Console.WriteLine(names[i]);
}
```

2. Запустите код и проанализируйте результат:

```
Kate
Jack
Rebecca
Tom
```

Массивы всегда имеют фиксированный размер, поэтому вам нужно предварительно решить, сколько элементов вы хотите сохранить в массиве, прежде чем создавать его.

Альтернативой показанному выше определению массива может служить использование синтаксиса инициализатора массива:

```
string[] names2 = new[] { "Kate", "Jack", "Rebecca", "Tom" };
```

Когда вы используете синтаксис `new[]` для выделения памяти под массив, вы должны поместить хотя бы один элемент в фигурные скобки, чтобы компилятор мог определить тип данных.

Массивы удобны для временного хранения нескольких элементов, а *коллекции* предпочтительны при динамическом добавлении и удалении элементов. О коллекциях мы поговорим в главе 8.

Дальнейшее изучение консольных приложений

Мы уже создали и использовали базовые консольные приложения, но сейчас находимся на этапе, когда должны еще больше углубиться в них.

Консольные приложения основаны на тексте и запускаются в командной строке. Обычно они выполняют простые задачи, развивающиеся по заданному сценарию, такие как компиляция файла или шифрование раздела файла конфигурации.

Точно так же им можно передавать аргументы для управления поведением.

Примером этого может быть создание консольного приложения на языке F# с указанным именем вместо имени текущей папки, как показано в следующей командной строке:

```
dotnet new console -lang "F#" --name "ExploringConsole"
```

Отображение вывода пользователю

Две основные задачи любого консольного приложения заключаются в записи и чтении данных. Мы уже использовали метод `WriteLine` для вывода. Если бы не требовался возврат каретки в конце каждой строки, то мы могли бы применить метод `Write`.

Форматирование с помощью пронумерованных позиционных аргументов

Использование пронумерованных позиционных аргументов — один из способов создания форматированных строк.

Эта функция поддерживается такими методами, как `Write` и `WriteLine`, а для методов, которые ее не поддерживают, параметр `string` можно отформатировать с помощью метода `Format` типа `string`.

Первые несколько примеров кода в этом разделе будут работать из блокнотов .NET Interactive, поскольку они предназначены для вывода на консоль. Далее в этом разделе вы узнаете о вводе через консоль, что, к сожалению, в блокноте сделать невозможно.

Рассмотрим пример.

1. Откройте редактор кода и создайте консольное приложение `Formatting` в рабочей области/решении `Chapter02`.
2. В программе Visual Studio Code выберите `Formatting` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` добавьте операторы для объявления некоторых числовых переменных и записи их в консоль:

```
int numberOfApples = 12;
decimal pricePerApple = 0.35M;

Console.WriteLine(
  format: "{0} apples costs {1:C}",
  arg0: numberOfApples,
  arg1: pricePerApple * numberOfApples);

string formatted = string.Format(
  format: "{0} apples costs {1:C}",
  arg0: numberOfApples,
  arg1: pricePerApple * numberOfApples);

//WriteToFile(formatted); // записывает строку в файл
```

Метод `WriteToFile` — несуществующий, используется для демонстрации примера.

Изучив форматирование строк, вы перестанете именовать параметры в духе format:, arg0: и arg1:. В коде выше используется неканонический стиль, чтобы показать вам, пока вы учитесь, откуда взялись 0 и 1.

Форматирование с помощью интерполированных строк

Версия C# 6.0 и выше содержит удобную функцию *интерполяции строк*. Строка с префиксом `$` должна содержать фигурные скобки вокруг имени переменной или выражения для вывода текущего значения этой переменной или выражения в строке.

1. В конце файла `Program.cs` введите оператор:

```
Console.WriteLine($"{numberOfApples} apples costs {pricePerApple *
numberOfApples:C}");
```

2. Запустите код и проанализируйте результат:

```
12 apples costs £4.20
```

Для коротких отформатированных значений `string` интерполированная строка может читаться проще. Но для примеров в книге, где код должен переноситься на несколько строк, процесс чтения может быть сложным. Во многих примерах кода в этой книге я буду использовать пронумерованные позиционные аргументы.

Другая причина, по которой следует избегать интерполированных строк, заключается в том, что их нельзя прочитать из файлов локализации.

До версии C# 10 строковые константы можно было объединять только с помощью конкатенации:

```
private const string firstname = "Omar";
private const string lastname = "Rudberg";
private const string fullname = firstname + " " + lastname;
```

В C# 10 теперь можно использовать интерполированные строки:

```
private const string fullname = $"{firstname} {lastname}";
```

Это допустимо только для объединения значений строковых констант и не поддерживается другими типами, такими как числа, которые требуют преобразования типов данных во время выполнения.

Форматирующие строки

Переменная или выражение могут быть отформатированы с использованием форматирующей строки после запятой или двоеточия.

Код `N0` форматирует число с запятыми в качестве разделителей тысяч и без дробной части. Код `C` форматирует число в значение валюты. Формат последней

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.

Полный синтаксис форматирующего элемента выглядит так:

```
{ index [, alignment ] [ : formatString ] }
```

Каждый форматирующий элемент можно выравнивать, что полезно при выводе таблиц значений. Некоторые значения необходимо будет выровнять по левому или правому краю в пределах ширины символов. Значения выравнивания — целые числа. При выравнивании по правому краю значения будут положительными целыми числами, а при выравнивании по левому — отрицательными.

Например, чтобы вывести на печать таблицу фруктов и их количество, мы могли бы выровнять имена в столбце из десяти символов по левому краю и выровнять по правому краю количество, отформатированное как числа с нулевыми десятичными знаками в столбце из шести символов.

1. В конце файла `Program.cs` введите операторы:

```
string applesText = "Apples";
int applesCount = 1234;

string bananasText = "Bananas";
int bananasCount = 56789;

Console.WriteLine(
  format: "{0,-10} {1,6}",
  arg0: "Name",
  arg1: "Count");

Console.WriteLine(
  format: "{0,-10} {1,6:N0}",
  arg0: applesText,
  arg1: applesCount);

Console.WriteLine(
  format: "{0,-10} {1,6:N0}",
  arg0: bananasText,
  arg1: bananasCount);
```

2. Запустите код и обратите внимание на эффект выравнивания и числового формата, как показано в этом выводе:

```
Name     Count
Apples   1,234
Bananas 56,789
```

Получение пользовательского ввода

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

Если для этого раздела вы используете блокнот .NET Interactive, то обратите внимание, что он не поддерживает чтение ввода с консоли с помощью функции Console.ReadLine(). Вместо этого вы должны установить литеральные значения, как показано в следующем коде: string? firstName = "Gary";. В ряде случаев это удобнее, поскольку вы можете быстро изменить строковое значение и нажать кнопку Execute Cell (Выполнить ячейку) вместо того, чтобы перезапускать консольное приложение каждый раз, когда вы хотите ввести другое значение string.

Получим ввод от пользователя.

1. Введите следующие операторы, чтобы запросить у пользователя имя и возраст, а затем вывести эту информацию:

```
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}.");
```

2. Запустите код. Затем введите имя и возраст, как показано в этом выводе:

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

Вопросительные знаки в конце string? указывает, что при вызове ReadLine может быть возвращено значение null (пустое). Подробнее об этом вы узнаете в главе 6.

Упрощение работы с командной строкой

В языке C# 6.0 и более поздних версиях оператор `using` можно использовать не только для импорта пространства имен, но и для дальнейшего упрощения кода за счет импорта статического класса. Теперь в нашем коде не нужно указывать тип

`Console`. Вы можете найти все вхождения и удалить их с помощью функции замены в любом редакторе кода.

1. В начале файла `Program.cs` добавьте оператор для *статического импорта* класса `System.Console`, как показано ниже:

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

2. Выделите первое слово `Console.` в коде, убедившись, что также выбрали точку после слова `Console`.
3. В программе Visual Studio выберите команду меню Edit ▸ Find and Replace ▸ Quick Replace (Правка ▸ Найти и заменить ▸ Быстро заменить) или в программе Visual Studio Code выберите команду меню Edit ▸ Replace (Правка ▸ Заменить) и обратите внимание, что отображается диалоговое окно с наложением, готовое для ввода значения, которым вы хотите заменить вариант Console. (рис. 2.5).

Рис. 2.5. Использование диалогового окна Replace (Заменить) в программе Visual Studio для упрощения кода

4. Оставьте поле замены пустым и нажмите кнопку Replace All (Заменить все) (вторая из двух кнопок справа от поля замены), затем закройте панель поиска, щелкнув на значке × в правом верхнем углу.

Получение клавиатурного ввода от пользователя

Мы можем получить клавиатурный ввод от пользователя с помощью метода `ReadKey`. Он ожидает, пока пользователь нажмет клавишу или комбинацию клавиш, которая затем возвращается как значение `ConsoleKeyInfo`.

Вы не сможете вызвать метод `ReadKey`, используя блокнот .NET Interactive. Но если вы создали консольное приложение, то мы можем рассмотреть считывание нажатий клавиш.

1. Введите код, предлагающий пользователю нажать одну из клавиш (или сочетаний), а затем вывести информацию о ней:

```
Write("Press any key combination: ");
ConsoleKeyInfo key = ReadKey();
WriteLine();
WriteLine("Key: {0}, Char: {1}, Modifiers: {2}",
  arg0: key.Key,
  arg1: key.KeyChar,
  arg2: key.Modifiers);
```

2. Запустите код, нажмите клавишу K и проанализируйте результат:

```
Press any key combination: k
Key: K, Char: k, Modifiers: 0
```

3. Запустите код; нажав и удерживая клавишу Shift, нажмите клавишу K и проанализируйте результат:

```
Press any key combination: K
Key: K, Char: K, Modifiers: Shift
```

4. Запустите код, нажмите клавишу F12 и проанализируйте результат:

```
Press any key combination:
Key: F12, Char: , Modifiers: 0
```

При запуске консольного приложения на панели TERMINAL (Терминал) программы Visual Studio Code некоторые комбинации клавиш будут захвачены редактором кода или операционной системой, прежде чем ваше приложение сможет их обработать.

Передача аргументов в консольное приложение

Вероятно, вам интересно узнать, как получить любые аргументы, которые могут быть переданы консольному приложению.

В .NET версии ниже 6.0 в шаблонах консольного приложения это было очевидно:

```
using System;

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

Аргументы `string[] args` объявляются и передаются в методе `Main` класса `Program`. Это массив, используемый для передачи аргументов в консольное приложение. Но в программах верхнего уровня, используемых в версии .NET 6.0 и более поздних, класс `Program` и его метод `Main` скрыты вместе с объявлением строкового массива `args`. Хитрость в том, что он все еще существует.

Аргументы командной строки разделяются пробелами. Другие символы, например дефисы и двоеточия, рассматриваются как часть значения аргумента.

Чтобы включить пробелы в значение аргумента, заключите значение аргумента в одинарные или двойные кавычки.

Представьте, что требуется возможность вводить имена цветов переднего плана и фона, а также размеры окна терминала в командной строке. Мы могли бы получать цвета и числа, считывая их из массива `args`, который всегда передается в метод `Main`, также известный как точка входа консольного приложения.

1. Откройте редактор кода и создайте консольное приложение `Arguments` в рабочей области/решении `Chapter02`. Вы не сможете использовать блокнот .NET Interactive, так как вы не можете передавать в него аргументы.
2. В программе Visual Studio Code выберите `Arguments` в качестве активного проекта OmniSharp.
3. Добавьте оператор для статического импорта типа `System.Console` и оператор для вывода количества аргументов, переданных приложению, как показано ниже:

```
using static System.Console;

WriteLine($"There are {args.Length} arguments.");
```

Не забывайте статически импортировать тип System.Console во всех будущих проектах, чтобы упростить код. Эти инструкции я не стану повторять.

4. Запустите код и проанализируйте результат:

```
There are 0 arguments.
```

5. В программе Visual Studio выберите команду меню Project ▸ Arguments Properties (Проект ▸ Свойства аргументов), выберите вкладку Debug (Отладка) и в поле Application arguments (Аргументы приложения) введите несколько аргументов, сохраните изменения, а затем запустите консольное приложение (рис. 2.6).

Рис. 2.6. Ввод аргументов приложения в программе Visual Studio

6. В программе Visual Studio Code на панели `TERMINAL` (Терминал) введите несколько аргументов после команды `dotnet run`, как показано в следующей командной строке:

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

7. Обратите внимание, что результат указывает на четыре аргумента:

```
There are 4 arguments.
```

8. Чтобы перечислить или перебрать (то есть выполнить цикл) значения этих четырех аргументов, добавьте следующие операторы после вывода длины массива:

```
foreach (string arg in args)
{
  WriteLine(arg);
}
```

9. Запустите код и проанализируйте результат, который дает подробную информацию о четырех аргументах:

```
There are 4 arguments.
firstarg
second-arg
third:arg
fourth arg
```

Настройка параметров с помощью аргументов

Теперь мы будем применять эти аргументы, чтобы пользователь мог выбрать цвет фона, переднего плана, ширины и высоты окна вывода, а также размер курсора. Последнее значение может быть представлено целым числом от 1, что означает линию внизу ячейки курсора, до 100, что означает процент от высоты ячейки курсора.

Пространство имен `System` уже импортировано, поэтому компилятор знает о типах `ConsoleColor` и `Enum`.

1. Добавьте операторы, запрашивающие у пользователя три аргумента, а затем проанализируйте эти аргументы и примените для настройки цвета и размера окна консоли:

```
if (args.Length < 3)
{
  WriteLine("You must specify two colors and cursor size, e.g.");
  WriteLine("dotnet run red yellow 50");
  return; // прекращение запуска
}

ForegroundColor = (ConsoleColor)Enum.Parse(
  enumType: typeof(ConsoleColor),
  value: args[0],
  ignoreCase: true);

BackgroundColor = (ConsoleColor)Enum.Parse(
  enumType: typeof(ConsoleColor),
  value: args[1],
  ignoreCase: true);

CursorSize = int.Parse(args[2]);
```

Настройка CursorSize поддерживается только в Windows.

2. В программе Visual Studio выберите команду меню Project ▸ Arguments Properties (Проект ▸ Свойства аргументов) и измените аргументы на `red yellow 50`. Запустите консольное приложение и обратите внимание, что размер курсора уменьшился в два раза, а цвета в окне изменились (рис. 2.7).

Рис. 2.7. Настройка цветов и размера курсора в Windows

3. В программе Visual Studio Code запустите код с аргументами, чтобы установить красный цвет переднего плана, желтый цвет фона и размер курсора 50 %:

```
dotnet run red yellow 50
```

В операционной системе macOS вы увидите необработанное исключение (рис. 2.8).

Рис. 2.8. Необработанное исключение в неподдерживаемой операционной системе macOS

Хотя компилятор не выдал ошибку или предупреждение, во время выполнения некоторые API-вызовы могут вызывать ошибку на отдельных платформах. Консольное приложение, работающее в Windows, может изменять размер курсора, однако в macOS это невозможно, и при попытке сделать это вы получите ошибку.

Работа с платформами, не поддерживающими некоторые API

Как же мы можем решить эту проблему? С помощью обработчика исключений. Более подробную информацию об операторе `try-catch` вы узнаете в главе 3, поэтому сейчас просто введите код.

1. Измените код, чтобы обернуть строки, которые изменяют размер курсора, в оператор `try`:

```
try
{
  CursorSize = int.Parse(args[2]);
}
catch (PlatformNotSupportedException)
```

```
{
  WriteLine("The current platform does not support changing the size of
the cursor.");
}
```

2. Если код запущен на macOS, обратите внимание, что исключение перехвачено и пользователю отображено понятное сообщение.

Еще один способ справиться с различиями в операционных системах — использовать класс `OperatingSystem` в пространстве имен `System`:

```
if (OperatingSystem.IsWindows())
{
  // выполнить код, работающий только в Windows
}
else if (OperatingSystem.IsWindowsVersionAtLeast(major: 10))
{
  // выполнить код, работающий только в Windows 10 или более поздней версии
}
else if (OperatingSystem.IsIOSVersionAtLeast(major: 14, minor: 5))
{
  // выполнить код, работающий только в iOS 14.5 или более поздней версии
}
else if (OperatingSystem.IsBrowser())
{
  // выполнить код, работающий только в браузере с Blazor
}
```

Класс `OperatingSystem` содержит эквивалентные методы для других распространенных ОС, таких как Android, iOS, Linux, macOS и даже для браузера, что полезно для веб-компонентов Blazor.

Третий способ работы с разными платформами — использование операторов условной компиляции.

Существует четыре директивы препроцессора, управляющие условной компиляцией: `#if`, `#elif`, `#else` и `#endif`.

Вы определяете символы с помощью команды `#define`:

```
#define MYSYMBOL
```

Многие символы определяются автоматически, как показано в табл. 2.22.

Таблица 2.22. Символы, определяемые автоматически

Framework	Символы
.NET Standard	`NETSTANDARD2_0`, `NETSTANDARD2_1` и т. д.
Modern .NET	`NET6_0`, `NET6_0_ANDROID`, `NET6_0_IOS`, `NET6_0_WINDOWS` и т. д.

Затем вы можете вводить операторы, которые будут компилироваться только для указанных платформ:

```
#if NET6_0_ANDROID
// компилировать операторы, работающие только в Android
#elif NET6_0_IOS
// компилировать операторы, работающие только в iOS
#else
// компилировать операторы, работающие в любой ОС
#endif
```

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 2.1. Проверочные вопросы

Получить лучший ответ на некоторые из этих вопросов можно, проведя собственное исследование. Я хочу, чтобы вы «мыслили вне книги», поэтому сознательно не предоставил все ответы.

Я хочу научить вас пользоваться дополнительной информацией из других источников.

1. Какой оператор можно ввести в файл C#, чтобы узнать версию компилятора и языка?
2. Каковы два типа комментариев в C#?
3. В чем разница между дословной и интерполированной строкой?
4. Почему следует быть осторожными при использовании значений `float` и `double`?
5. Как определить, сколько байтов памяти использует такой тип, как `double`?
6. Когда следует использовать ключевое слово `var`?
7. Каков новейший способ создания экземпляра класса, такого как `XmlDocument`?
8. Почему следует быть осторожными при использовании типа `dynamic`?
9. Как выровнять строку формата по правому краю?
10. Какой символ разделяет аргументы в консольном приложении?

Приложение доступно для скачивания в репозитории GitHub: https://github.com/markjprice/cs10dotnet6.

Упражнение 2.2. Проверочные вопросы о числовых типах

Какой тип следует выбрать для каждого указанного ниже числа?

1. Телефонный номер.
2. Рост.
3. Возраст.
4. Размер оклада.
5. Международный стандартный книжный номер.
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 50 000 сотрудников на предприятие).

Упражнение 2.3. Практическое задание — числовые размеры и диапазоны

В решении/рабочей области `Chapter02` создайте проект консольного приложения `Exercise03`, которое 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`.

Результат работы вашего приложения должен выглядеть примерно так (рис. 2.9).

```
Microsoft Visual Studio Debug Console
--------------------------------------------------------------------------
Type    Byte(s) of memory              Min                            Max
--------------------------------------------------------------------------
sbyte   1                             -128                            127
byte    1                                0                            255
short   2                            -32768                          32767
ushort  2                                0                          65535
int     4                       -2147483648                     2147483647
uint    4                                0                     4294967295
long    8              -9223372036854775808            9223372036854775807
ulong   8                                0           18446744073709551615
float   4                    -3.4028235E+38                  3.4028235E+38
double  8           -1.7976931348623157E+308        1.7976931348623157E+308
decimal 16   -79228162514264337593543950335  79228162514264337593543950335
--------------------------------------------------------------------------
```

Рис. 2.9. Результат вывода размеров числового типа

Решения для всех упражнений доступны для скачивания или клонирования из репозитория GitHub по ссылке: https://github.com/markjprice/cs10dotnet6.

Упражнение 2.4. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-2---speaking-c, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали, как:

- объявлять переменные с помощью явно указанного или автоматически определенного типа;
- использовать некоторые встроенные числовые, текстовые и логические типы;
- выбирать между различными числовыми типами;
- управлять форматированием вывода в консольных приложениях.

В следующей главе вы узнаете об операциях, ветвлении, переборе, преобразовании типов и о том, как обрабатывать исключения.

3 Управление потоком исполнения, преобразование типов и обработка исключений

Данная глава посвящена написанию кода, который выполняет простые операции с переменными, принимает решения, выполняет сопоставление с образцом, повторяет операторы или блоки, преобразует значения переменных или выражений из одного типа в другой, обрабатывает исключения и проверяет переполнение числовых переменных.

В этой главе:

- работа с переменными;
- операторы выбора;
- операторы цикла;
- приведение и преобразование типов;
- обработка исключений;
- проверка переполнения.

Работа с переменными

Операции[1] (operators) применяют простые действия, такие как сложение и умножение, к *операндам*, например к переменным и литеральным значениям. Обычно они возвращают новое значение, являющееся результатом операции, которую можно присвоить переменной.

---

[1] Английское слово operator, соответствующее термину «операция», иногда ошибочно переводят как «оператор». На самом деле (по историческим причинам) русский термин «оператор» соответствует английскому statement. Разговаривая с коллегами, скорее всего, вы будете использовать термин «оператор» как аналог англоязычного operator.

Большинство операций — бинарные, то есть работают с двумя операндами, как показано в следующем псевдокоде:

```
var resultOfOperation = firstOperand operator secondOperand;
```

Примеры бинарных операций, в том числе сложения и умножения:

```
int x = 5;
int y = 3;
int resultOfAdding = x + y;
int resultOfMultiplying = x * y;
```

Некоторые операции — унарные, то есть работают с одним операндом и могут применяться до или после него, как показано в следующем псевдокоде:

```
var resultOfOperation = onlyOperand operator;
var resultOfOperation2 = operator onlyOperand;
```

Примеры унарных операций включают инкременты и извлечение типа или его размера в байтах:

```
int x = 5;
int postfixIncrement = x++;
int prefixIncrement = ++x;
Type theTypeOfAnInteger = typeof(int);
int howManyBytesInAnInteger = sizeof(int);
```

Тернарная операция работает с тремя операндами, как показано в следующем псевдокоде:

```
var resultOfOperation = firstOperand firstOperator
  secondOperand secondOperator thirdOperand;
```

Унарные операции

К двум самым распространенным унарным операциям относятся операции инкремента (увеличения), `++`, и декремента (уменьшения), `--`, числа. Напишем пример кода, чтобы посмотреть, как они работают.

1. Если вы прочли предыдущие главы, то в вашей пользовательской папке уже есть папка Code. Если нет, то создайте ее.
2. Откройте редактор кода и создайте консольное приложение с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter03`;
   3) файл и папка проекта: `Operators`.

3. В начале файла `Program.cs` статически импортируйте тип `System.Console`.
4. В файле `Program.cs` объявите две целочисленные переменные с именами `a` и `b`, задайте `a` равной `3`, увеличьте `a`, присваивая результат `b`, а затем выведите значения, как показано ниже:

```
int a = 3;
int b = a++;
WriteLine($"a is {a}, b is {b}");
```

5. Перед запуском консольного приложения задайте себе вопрос: как вы думаете, каким будет значение переменной `b` при выводе? Далее запустите код и проанализируйте результат:

```
a is 4, b is 3
```

Переменная `b` имеет значение `3`, поскольку операция `++` выполняется *после* присваивания; это известно как *постфиксная операция*. Если вам нужно увеличить значение *перед* присваиванием, то используйте *префиксную операцию*.

6. Скопируйте и вставьте операторы, а затем измените их, чтобы переименовать переменные и использовать префиксную операцию, как показано ниже:

```
int c = 3;
int d = ++c; // увеличение c перед присваиванием
WriteLine($"c is {c}, d is {d}");
```

7. Перезапустите код и проанализируйте результат:

```
a is 4, b is 3
c is 4, d is 4
```

Из-за путаницы с префиксами и постфиксами операций инкремента и декремента при присваивании разработчики языка программирования Swift планируют отказаться от поддержки этой операции в версии 3. Я рекомендую программистам на языке C# никогда не сочетать операции ++ и –– с операцией присваивания =. Лучше выполнять эти действия как отдельные операторы.

Арифметические бинарные операции

Инкремент и декремент — унарные арифметические операции. Другие арифметические операции обычно бинарные и позволяют выполнять арифметические действия с двумя числами.

1. Добавьте операторы, которые объявляют и присваивают значения двум целочисленным переменным с именами `e` и `f`, а затем выполните пять обычных бинарных арифметических операций для двух чисел, как показано ниже:

```
int e = 11;
int f = 3;
WriteLine($"e is {e}, f is {f}");
WriteLine($"e + f = {e + f}");
WriteLine($"e - f = {e - f}");
WriteLine($"e * f = {e * f}");
WriteLine($"e / f = {e / f}");
WriteLine($"e % f = {e % f}");
```

2. Перезапустите код и проанализируйте результат:

```
e is 11, f is 3
e + f = 14
e - f = 8
e * f = 33
e / f = 3
e % f = 2
```

Чтобы понять, как работают операции деления (`/`) и остатка от деления (деления по модулю) (`%`) при применении к целым числам (натуральным числам), вам нужно вспомнить уроки средней школы. Представьте, что у вас есть 11 леденцов и три друга.

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

3. Добавьте операторы для объявления переменной и присвойте значение переменной типа `double` с именем `g`, чтобы показать разницу между делением на целые и действительные числа:

```
double g = 11.0;
WriteLine($"g is {g:N1}, f is {f}");
WriteLine($"g / f = {g / f}");
```

4. Перезапустите код и проанализируйте результат:

```
g is 11.0, f is 3
g / f = 3.6666666666666665
```

Если брать вещественные числа в качестве первого операнда, например значение переменной `g`, равное `11.0`, то операция деления возвращает значение с плавающей запятой, например `3.6666666666665`, а не целое число.

Операция присваивания

Вы уже использовали самую распространенную операцию присваивания, `=`.

Чтобы сократить ваш код, вы можете объединить операцию присваивания с другими, такими как арифметические операции:

```
int p = 6;
p += 3; // эквивалентно p = p + 3;
p -= 3; // эквивалентно p = p – 3;
p *= 3; // эквивалентно p = p * 3;
p /= 3; // эквивалентно p = p / 3;
```

Логические операции

Логические операции работают с логическими значениями и возвращают в результате значение `true` (истина) или `false` (ложь). Рассмотрим бинарные логические операции, работающие с двумя логическими значениями.

1. Откройте редактор кода и создайте консольное приложение `BooleanOperators` в рабочей области/решении `Chapter03`:
   1) в программе Visual Studio Code выберите `BooleanOperators` в качестве активного проекта OmniSharp. Когда появится предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку Yes (Да), чтобы добавить их;
   2) в программе Visual Studio в разделе Startup Project (Стартовый проект) установите переключатель в положение Current selection (Текущий выбор).

Не забудьте статически импортировать тип System.Console, чтобы упростить операторы в консольном приложении.

2. В файле `Program.cs` добавьте операторы для объявления двух логических переменных со значениями `true` и `false`, а затем выведите таблицы истинности, отображающие результаты применения логических операций `AND`, `OR` и `XOR` (исключающее ИЛИ), как показано ниже:

```
bool a = true;
bool b = false;

WriteLine($"AND | a     | b    ");
WriteLine($"a   | {a & a,-5} | {a & b,-5} ");
WriteLine($"b   | {b & a,-5} | {b & b,-5} ");
WriteLine();
WriteLine($"OR  | a     | b    ");
WriteLine($"a   | {a | a,-5} | {a | b,-5} ");
WriteLine($"b   | {b | a,-5} | {b | b,-5} ");
```

```
WriteLine();
WriteLine($"XOR | a     | b    ");
WriteLine($"a   | {a ^ a,-5} | {a ^ b,-5} ");
WriteLine($"b   | {b ^ a,-5} | {b ^ b,-5} ");
```

3. Запустите код и проанализируйте результат:

```
AND | a     | b
a   | True  | False
b   | False | False

OR  | a     | b
a   | True  | True
b   | True  | False

XOR | a     | b
a   | False | True
b   | True  | False
```

Для логической операции `AND` `&` (логическое И) оба операнда должны быть равными `true`, чтобы в результате вернулось значение `true`. Для логической операции `OR` `|` (логическое ИЛИ), любой операнд может быть равным `true`, чтобы в результате вернулось значение `true`. Для логической операции `XOR` `^` (исключающее ИЛИ) один из операндов может быть равным `true` (но не оба!), чтобы результат был равным `true`.

Условные логические операции

Условные логические операции аналогичны логическим операциям, но вместо одного символа вы используете два, например, `&&` вместо `&` или `||` вместо `|`.

В главе 4 вы узнаете о функциях более подробно, а сейчас мне нужно представить функции, чтобы объяснить условные операции, также известные как короткозамкнутый аналог логических операций.

Функция выполняет определенный список операторов, а затем возвращает значение. Оно может быть логическим значением (например, `true`), которое используется в логической операции. Рассмотрим пример использования логических операторов.

1. В конце файла `Program.cs` введите операторы для объявления функции, которая выводит сообщение в консоль и возвращает значение `true`:

```
static bool DoStuff()
{
  WriteLine("I am doing some stuff.");
  return true;
}
```

Если вы используете .NET Interactive Notebook, то напишите функцию DoStuff в отдельной ячейке кода, а затем выполните ее, чтобы сделать ее контекст доступным для других ячеек кода.

2. После оператора `WriteLine` выполните операцию `AND &` с переменными `a` и `b` и проанализируйте результат вызова функции:

```
WriteLine();
WriteLine($"a & DoStuff() = {a & DoStuff()}");
WriteLine($"b & DoStuff() = {b & DoStuff()}");
```

3. Запустите код, проанализируйте результат и обратите внимание, что функция была вызвана дважды — для `a` и для `b`, как показано в выводе ниже:

```
I am doing some stuff.
a & DoStuff() = True
I am doing some stuff.
b & DoStuff() = False
```

4. Замените операторы `&` операторами `&&`:

```
WriteLine($"a && DoStuff() = {a && DoStuff()}");
WriteLine($"b && DoStuff() = {b && DoStuff()}");
```

5. Запустите код, проанализируйте результат и обратите внимание, что функция работает, когда объединяется с переменной `a`, но не запускается в момент объединения с переменной `b`. Это связано с тем, что переменная `b` имеет значение `false`, поэтому в любом случае результат будет ложным, ввиду чего функция не выполняется:

```
I am doing some stuff.
a && DoStuff() = True
b && DoStuff() = False // Функция DoStuff не выполнена!
```

Теперь вы можете понять, почему условные логические операции описаны как короткозамкнутые. Они могут сделать ваши приложения более эффективными, но могут и вносить незначительные ошибки в тех случаях, когда вы предполагаете, что функция будет вызываться всегда. Лучше всего избегать их при использовании в сочетании с функциями, вызывающими побочные эффекты.

Побитовые операции и операции побитового сдвига

Побитовые операции влияют на биты в числе. Операции побитового сдвига могут выполнять некоторые распространенные арифметические вычисления намного быстрее, чем классические операции. Например, умножение на 2.

Рассмотрим побитовые операции и операции побитового сдвига.

1. Откройте редактор кода и создайте консольное приложение `BitwiseAndShiftOperators` в рабочей области/решении `Chapter03`.
2. В программе Visual Studio Code выберите `BitwiseAndShiftOperators` в качестве активного проекта OmniSharp. Когда появится предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку `Yes` (Да), чтобы добавить их.
3. В файле `Program.cs` добавьте операторы, объявляющие две целочисленные переменные со значениями `10` и `6`, а затем выведите результаты применения побитовых операций `AND`, `OR` и `XOR` (исключающее ИЛИ):

```
int a = 10; // 00001010
int b = 6; // 00000110

WriteLine($"a = {a}");
WriteLine($"b = {b}");
WriteLine($"a & b = {a & b}"); // только столбец 2-го бита
WriteLine($"a | b = {a | b}"); // столбцы 8, 4 и 2-го битов
WriteLine($"a ^ b = {a ^ b}"); // столбцы 8 и 4-го битов
```

4. Запустите код и проанализируйте результат:

```
a = 10
b = 6
a & b = 2
a | b = 14
a ^ b = 12
```

5. В файле `Program.cs` добавьте следующие операторы, чтобы вывести результаты применения операции сдвига влево для перемещения битов переменной `a` на три столбца, умножения `a` на `8` и сдвига вправо битов переменной `b` на один столбец:

```
// 01010000 — сдвиг влево переменной a на три битовых столбца
WriteLine($"a << 3 = {a << 3}");

// умножение переменной a на 8
WriteLine($"a * 8 = {a * 8}");

// 00000011 сдвиг вправо переменной b на один битовый столбец
WriteLine($"b >> 1 = {b >> 1}");
```

6. Запустите код и проанализируйте результат:

```
a << 3 = 80
a * 8 = 80
b >> 1 = 3
```

Результат `80` получился из-за того, что биты в нем были смещены на три столбца влево, поэтому биты-единицы переместились в столбцы 64-го и 16-го битов,

а 64 + 16 = 80. Это эквивалентно умножению на 8, однако процессоры могут выполнить битовый сдвиг быстрее. Результат 3 получился из-за того, что биты-единицы в b были сдвинуты на один столбец и оказались во втором и первом столбцах.

Обратите внимание, что при работе с целочисленными значениями операторы & и | являются побитовыми, а при работе с булевыми значениями, такими как true и false, — логическими.

Мы можем продемонстрировать эти операции, преобразовав целочисленные значения в двоичные строки нулей и единиц.

1. В конце файла Program.cs добавьте функцию для преобразования целочисленного значения в двоичную (Base 2) строку, содержащую до восьми нулей и единиц:

```
static string ToBinaryString(int value)
{
  return Convert.ToString(value, toBase: 2).PadLeft(8, '0');
}
```

2. Выше функции добавьте операторы для вывода значений переменных a, b и результатов побитовых операций:

```
WriteLine();
WriteLine("Outputting integers as binary:");
WriteLine($"a =     {ToBinaryString(a)}");
WriteLine($"b =     {ToBinaryString(b)}");
WriteLine($"a & b = {ToBinaryString(a & b)}");
WriteLine($"a | b = {ToBinaryString(a | b)}");
WriteLine($"a ^ b = {ToBinaryString(a ^ b)}");
```

3. Запустите код и проанализируйте результат:

```
Outputting integers as binary:
a =     00001010
b =     00000110
a & b = 00000010
a | b = 00001110
a ^ b = 00001100
```

Прочие операции

Операции nameof и sizeof весьма удобны при работе с типами:

- операция nameof возвращает короткое имя (без пространства имен) переменной, типа или члена в виде значения string, что полезно при выводе сообщений об исключениях;

- операция `sizeof` возвращает размер в байтах простых типов, что полезно для определения эффективности хранения данных.

Существует большое разнообразие других операций. Например, точка между переменной и ее членами называется *операцией доступа к элементу*, а круглые скобки в конце имени функции или метода называются *операцией вызова*:

```
int age = 47;

// Сколько операций в следующем операторе?
char firstDigit = age.ToString()[0];

// Здесь четыре операции:
// = — операция присваивания
// . — операция доступа
// () — операция вызова
// [] — операция доступа к индексатору
```

Операторы выбора

Код каждого приложения должен обеспечивать возможность выбора одного из нескольких вариантов и ветвиться в соответствии с ним. Два оператора выбора в языке C# носят имена `if` и `switch`. Первый применим для любых сценариев, а второй позволяет упростить код в некоторых распространенных случаях. Например, когда есть одна переменная, которая может иметь несколько значений, каждое из которых требует особой обработки.

Ветвление с помощью оператора if

Оператор `if` определяет, по какой ветви кода необходимо следовать после вычисления логического выражения. Если выражение имеет значение `true`, то блок выполняется. Блок `else` необязателен и выполняется, если выражение в `if` принимает значение `false`. Оператор `if` может быть вложенным.

Оператор `if` комбинируется с другими операторами `if`, как в случае с `else if`:

```
if (выражение1)
{
  // работает, если выражение1 возвращает значение true
}
else if (выражение2)
{
  // работает, если выражение1 возвращает значение false,
  // а выражение2 — true
}
else if (выражение3)
{
```

```
  // работает, если выражение1 и выражение2 возвращают значение false,
  // а выражение3 — true
}
else
{
  // работает, если все выражения возвращают значение false
}
```

Каждое логическое значение выражения оператора `if` может быть независимым от других и, в отличие от операторов `switch`, не обязательно должно ссылаться на одно значение.

Создадим консольное приложение для изучения операторов выбора, таких как `if`.

1. Откройте редактор кода и создайте консольное приложение `SelectionStatements` в рабочей области/решении `Chapter03`.
2. В программе Visual Studio Code выберите `SelectionStatements` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` добавьте следующие операторы для проверки того, состоит ли пароль по крайней мере из восьми символов:

```
string password = "ninja";

if (password.Length < 8)
{
  WriteLine("Your password is too short. Use at least 8 characters.");
}
else
{
  WriteLine("Your password is strong.");
}
```

4. Запустите код и проанализируйте результат:

```
Your password is too short. Use at least 8 characters.
```

Почему в операторах if необходимы фигурные скобки

Поскольку в каждом блоке указывается только один оператор, представленный выше код можно переписать без фигурных скобок:

```
if (password.Length < 8)
  WriteLine("Your password is too short. Use at least 8 characters.");
else
  WriteLine("Your password is strong.");
```

Такой формат оператора `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 сентябре 2012 года в ней присутствовала ошибка в коде протокола *Secure Sockets Layer (SSL)*. Из-за этого любой пользователь, который подключался через браузер Safari к защищенным сайтам, например к сервису интернет-банкинга, не был защищен должным образом, поскольку важная проверка была случайно пропущена.

Только из возможности опустить фигурные скобки не следует так делать. Ваш код не становится «более эффективным» без них, вместо этого он менее удобен в сопровождении и потенциально более уязвим.

Сопоставление с образцом с помощью операторов if

Новая возможность версии C# 7.0 — сопоставление с образцом. В операторе `if` можно использовать ключевое слово `is` в сочетании с объявлением локальной переменной, чтобы сделать ваш код более безопасным.

1. Добавьте приведенные ниже операторы. Если значение, хранящееся в переменной `o`, представляет собой тип `int`, то значение присваивается локальной переменной `i`, которая затем может применяться в операторе `if`. Это безопаснее, чем использование переменной `o`, поскольку мы точно знаем, что `i` — это `int`, а не что-то другое:

```
// добавьте или удалите "", чтобы изменить поведение
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!");
}
```

2. Запустите код и проанализируйте результат:

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

3. Удалите двойные кавычки вокруг значения `3`, чтобы значение, хранящееся в переменной `o`, стало типом `int` вместо `string`.
4. Перезапустите код и проанализируйте результат:

```
3 x 4 = 12
```

Ветвление с помощью оператора switch

Оператор `switch` отличается от `if` тем, что проверяет одно выражение на соответствие трем или больше условиям (`case`). Каждый оператор `case` относится к одному выражению. Каждый раздел `case` должен заканчиваться:

- ключевыми словами `break` (например, в коде ниже `case 1`);
- или ключевыми словами `goto case` (например, в коде ниже `case 2`);
- либо не иметь никаких операторов (например, в коде ниже `case 3`);
- или ключевым словом `goto`, которое ссылается на именованную метку (например, в коде ниже `case` 5);
- или ключевым словом `return` для выхода из текущей функции (не показано в коде).

Напишем пример кода для изучения операторов `switch`.

1. Введите код для оператора `switch`. Обратите внимание: предпоследний оператор — метка, к которой можно перейти, а первый оператор генерирует случайное число в диапазоне 1–6 (но число 7 в коде — это верхняя граница). Оператор `switch` выбирает ветвь, основываясь на значении этого числа:

```
int number = (new Random()).Next(1, 7);
WriteLine($"My random number is {number}");

switch (number)
{
  case 1:
    WriteLine("One");
    break; // переход в конец оператора switch
  case 2:
    WriteLine("Two");
    goto case 1;
  case 3: // блок, содержащий несколько случаев
  case 4:
    WriteLine("Three or four");
    goto case 1;
  case 5:
    goto A_label;
  default:
    WriteLine("Default");
    break;
} // конец оператора switch

WriteLine("After end of switch");
A_label:
WriteLine($"After A_label");
```

Вы можете перейти к другому условию или метке с помощью ключевого слова goto. Его применение не одобряется большинством программистов, но может стать хорошим решением при кодировании логики в некоторых сценариях. Не стоит использовать его слишком часто.

2. Запустите код несколько раз, чтобы посмотреть на происходящее при различных условиях случайных чисел, как показано в примере:

```
// первый рандомный запуск
My random number is 4
Three or four
One
After end of switch
After A_label

// второй рандомный запуск
My random number is 2
Two
One
After end of switch
After A_label

// третий рандомный запуск
My random number is 6
Default
After end of switch
After A_label

// четвертый рандомный запуск
My random number is 1
One
After end of switch
After A_label

// пятый рандомный запуск
My random number is 5
After A_label
```

Сопоставление с образцом с помощью оператора switch

Как и `if`, оператор `switch` поддерживает сопоставление с образцом в версии C# 7.0 и более поздних. Значения условий `case` больше не должны быть литеральными. Они могут быть образцами.

Рассмотрим пример того, как сопоставлять с образцом с помощью оператора `switch`, используя путь к папке. Если вы работаете в операционной системе macOS, то

поменяйте местами закомментированный оператор, который устанавливает переменную пути, и замените мое имя пользователя именем вашей пользовательской папки.

1. Добавьте следующие операторы, чтобы объявить путь `string` к файлу, открыть его в режиме только для чтения как поток и затем показать сообщение в зависимости от типа и возможностей потока:

```
// string path = "/Users/markjprice/Code/Chapter03";
string path = @"C:\Code\Chapter03";

Write("Press R for read-only or W for writeable: ");
ConsoleKeyInfo key = ReadKey();
WriteLine();

Stream? s;

if (key.Key == ConsoleKey.R)
{
  s = File.Open(
    Path.Combine(path, "file.txt"),
    FileMode.OpenOrCreate,
    FileAccess.Read);
}
else
{
  s = File.Open(
    Path.Combine(path, "file.txt"),
    FileMode.OpenOrCreate,
    FileAccess.Write);
}

string message;

switch (s)
{
  case FileStream writeableFile when s.CanWrite:
    message = "The stream is a file that I can write to.";
    break;
  case FileStream readOnlyFile:
    message = "The stream is a read-only file.";
    break;
  case MemoryStream ms:
    message = "The stream is a memory address.";
    break;
  default: // всегда выполняется последним, несмотря на текущее положение
    message = "The stream is some other type.";
    break;
  case null:
    message = "The stream is null.";
```

```
    break;
}

WriteLine(message);
```

2. Запустите код и обратите внимание, что переменная `s` объявлена как тип `Stream`, вследствие чего это может быть любой подтип потока, например поток памяти или файлов. В данном коде поток создается с помощью метода `File.Open`, который возвращает файловый поток, и в зависимости от нажатия вами клавиши будет доступен для записи или только для чтения, поэтому результатом станет сообщение, описывающее ситуацию, как показано ниже:

```
The stream is a file that I can write to.
```

На платформе .NET доступно несколько подтипов `Stream`, включая `FileStream` и `MemoryStream`. В версии C# 7.0 ваш код может быть более лаконичным благодаря ветвлению на основе подтипа потока и объявлению/назначению локальной переменной для безопасного использования. Более подробно пространство имен `System.IO` и тип `Stream` мы разберем в главе 9.

Кроме того, операторы `case` могут содержать ключевое слово `when` для выполнения более точного сопоставления с образцом. В первом операторе `case` в предыдущем коде совпадение было бы установлено только в том случае, если поток — это `FileStream`, а его свойство `CanWrite` истинно.

Упрощение операторов switch с помощью выражений switch

В версии C# 8.0 или более поздних вы можете упростить операторы `switch`, используя *выражения switch*.

Операторы `switch` очень просты, но требуют большого количества кода. Выражения `switch` предназначены для упрощения кода, который нужно набирать, при этом выражая то же самое намерение в сценариях, где все варианты возвращают значение для установки одной переменной. В выражениях `switch` для обозначения возвращаемого значения используется лямбда `=>`.

Для сравнения выполним предыдущий код, в котором использовался оператор `switch`, с помощью выражения `switch`.

1. Добавьте следующие операторы, чтобы установить сообщение в зависимости от типа и возможностей потока, использующего выражение `switch`:

```
message = s switch
{
```

```
    FileStream writeableFile when s.CanWrite
      => "The stream is a file that I can write to.",
    FileStream readOnlyFile
      => "The stream is a read-only file.",
    MemoryStream ms
      => "The stream is a memory address.",
    null
      => "The stream is null.",
    _
      => "The stream is some other type."
  };

  WriteLine(message);
```

Основное отличиое — удаление ключевых слов `case` и `break`. Символ подчеркивания используется для представления возвращаемого значения по умолчанию.

2. Запустите код и обратите внимание, что результат такой же, как и раньше.

Операторы цикла

Операторы цикла повторяют блок операторов либо пока условие истинно (операторы `while` и `for`), либо для каждого элемента в коллекции (оператор `foreach`). Выбор того, какой оператор следует использовать, основывается на сочетании простоты понимания решения логической задачи и личных предпочтений.

Оператор while

Оператор `while` оценивает логическое выражение и продолжает цикл, пока оно остается истинным. Рассмотрим операторы цикла на примере.

1. Откройте редактор кода и создайте консольное приложение `IterationStatements` в рабочей области/решении `Chapter03`.
2. В программе Visual Studio Code выберите `IterationStatements` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` введите операторы, определив цикл `while`, выполняющийся, пока значение целочисленной переменной меньше `10`:

```
int x = 0;

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

4. Запустите код и проанализируйте результат (должны быть представлены числа от 0 до 9):

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

Оператор do

Оператор `do` похож на `while`, за исключением того, что логическое выражение проверяется в конце блока кода, а не в начале, что означает обязательное выполнение кода хотя бы один раз.

1. Введите код, показанный ниже:

```
string? password;

do
{
  Write("Enter your password: ");
  password = ReadLine();
}
while (password != "Pa$$w0rd");

WriteLine("Correct!");
```

2. Запустите код и обратите внимание, что вам будет предложено ввести пароль несколько раз, пока вы не введете его правильно:

```
Enter your password: password
Enter your password: 12345678
Enter your password: ninja
Enter your password: correct horse battery staple
Enter your password: Pa$$w0rd
Correct!
```

3. В качестве дополнительной задачи добавьте операторы, ограничивающие количество попыток ввода пароля десятью, после чего, если правильный пароль так и не был введен, выводится сообщение об ошибке.

Оператор for

Оператор `for` аналогичен `while`, за исключением более лаконичного синтаксиса. Он комбинирует:

- *выражение инициализатора*, которое выполняется однократно при запуске цикла;
- *условное выражение*, выполняющееся на каждой итерации в начале цикла, чтобы проверить, следует ли продолжать цикл;
- *выражение-итератор*, которое выполняется в каждом цикле в конце выполнения оператора.

Оператор `for` обычно используется с целочисленным счетчиком. Рассмотрим пример.

1. Введите оператор `for` для вывода чисел от 1 до 10, как показано ниже:

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

2. Запустите код и проанализируйте результат, который должен быть представлен числами в диапазоне от 1 до 10.

Оператор foreach

Оператор `foreach` несколько отличается от предыдущих трех операторов цикла.

С его помощью можно выполнить блок операторов для каждого элемента в последовательности, к примеру в массиве или коллекции. Каждый элемент обычно доступен только для чтения, и если во время итерации последовательность изменяется, скажем, путем добавления или удаления элемента, то будет вызвано исключение.

Рассмотрим пример.

1. Введите операторы для создания массива строковых переменных, а затем выведите длину каждой из них:

```
string[] names = { "Adam", "Barry", "Charlie" };

foreach (string name in names)
{
  WriteLine($"{name} has {name.Length} characters.");
}
```

2. Запустите код и проанализируйте результат:

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

Принципы работы оператора foreach

При создании любого типа с несколькими элементами, такого как массив или коллекция, необходимо убедиться, что программист может использовать оператор `foreach` для перечисления элементов типа.

Технически оператор `foreach` будет работать с любым типом, соблюдающим правила, изложенные ниже.

1. Тип должен иметь метод `GetEnumerator`, который возвращает объект.
2. Возвращаемый объект должен иметь свойство `Current` и метод `MoveNext`.
3. Метод `MoveNext` должен изменять значение `Current` и возвращать значение `true`, если есть другие элементы для перечисления, или значение `false`, если элементов больше нет.

Существуют интерфейсы с именами `IEnumerable` и `IEnumerable <T>`, которые формально определяют эти правила, но технически компилятору не требуется тип для реализации этих интерфейсов.

Компилятор превращает код оператора `foreach` из предыдущего примера в нечто похожее на псевдокод, показанный ниже:

```
IEnumerator e = names.GetEnumerator();

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

Из-за применения итератора переменная, объявленная в операторе `foreach`, не может использоваться для изменения значения текущего элемента.

Приведение и преобразование типов

Вам часто понадобится преобразовывать значения переменных между разными типами. Например, данные часто вводятся в виде текста в консоли, поэтому изначально сохраняются в переменной типа `string`, а затем их необходимо преобразовать

в дату/время, число или какой-либо другой тип данных, в зависимости от того, как они должны храниться и обрабатываться.

Иногда вам нужно будет выполнить преобразование числовых типов, например целого числа и числа с плавающей запятой, прежде чем выполнять вычисления.

Преобразование также называется *приведением*; оно имеет две разновидности: *неявное* и *явное*. Неявное происходит автоматически, и это безопасно, то есть вы не потеряете никакую информацию.

Явное приведение должно быть выполнено вручную, поскольку вы можете потерять информацию, например точность числа. При явном приведении вы сообщаете компилятору C#, что понимаете риск и принимаете его на себя.

Явное и неявное приведение типов

Вы можете неявно привести переменную `int` к типу `double`, что безопасно, поскольку никакая информация не потеряется.

1. Откройте редактор кода и создайте консольное приложение `CastingConverting` в рабочей области/решении `Chapter03`.
2. В программе Visual Studio Code выберите `CastingConverting` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` введите операторы для объявления и назначения переменных `int` и `double`, а затем неявно приведите значение целого числа при назначении его переменной `double`, как показано ниже:

```
int a = 10;
double b = a; // тип int можно безопасно привести к double
WriteLine(b);
```

4. Объявите переменные типов `double` и `int`, а затем неявно преобразуйте значение типа `double` при присвоении его целочисленной переменной:

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

5. Запустите код и проанализируйте результат:

```
Error: (6,9): error CS0266: Cannot implicitly convert type 'double' to
'int'. An explicit conversion exists (are you missing a cast?)
```

Это сообщение об ошибке также появится в списке ошибок Visual Studio или в окне PROBLEMS (Проблемы) программы Visual Studio Code.

Вы не можете неявно преобразовать переменную `double` в переменную `int`, поскольку это потенциально небезопасно и может привести к потере данных. Необходимо явно привести переменную `double` к переменной `int`, заключив в круглые скобки тип, к которому вы хотите привести тип `double`. Парой круглых скобок обозначается *операция приведения типа*. И даже в этом случае вы должны остерегаться, что часть после десятичной запятой будет отброшена без предупреждения, поскольку вы решили выполнить явное приведение типов и, следовательно, понимаете последствия.

6. Измените оператор присваивания переменной `d`, как показано в коде, приведенном ниже:

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

7. Запустите код и проанализируйте результат:

```
10
9
```

Следует выполнять аналогичную операцию при преобразовании значений между бóльшими и меньшими целыми числами. Снова напомню: будьте осторожны, поскольку вы можете потерять данные, ведь слишком большое значение будет скопировано, а затем интерпретировано так, как вы этого, возможно, не ожидаете!

8. Введите операторы для объявления и присвоения 64-битной переменной типа `long` 32-битной переменной типа `int`, с помощью небольшого значения, которое будет работать, и слишком большого значения, которое не будет работать:

```
long e = 10;
int f = (int)e;
WriteLine($"e is {e:N0} and f is {f:N0}");
e = long.MaxValue;
f = (int)e;
WriteLine($"e is {e:N0} and f is {f:N0}");
```

9. Запустите код и проанализируйте результат:

```
e is 10 and f is 10
e is 9,223,372,036,854,775,807 and f is -1
```

10. Измените значение переменной `e` на пять миллиардов:

```
e = 5_000_000_000;
```

11. Запустите код и проанализируйте результат:

```
e is 5,000,000,000 and f is 705,032,704
```

Преобразование с помощью типа System.Convert

Вместо операции приведения типа можно использовать тип `System.Convert`. Он может преобразовывать во все числовые типы чисел C# и из них, а также в логические значения, строки и значения даты и времени.

Рассмотрим пример.

1. В начале файла `Program.cs` статически импортируйте класс `System.Convert`, как показано ниже:

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

2. В конце файла `Program.cs` добавьте следующие операторы, чтобы объявить и присвоить значение переменной типа `double`, преобразовать его в целое число, а затем записать оба значения в консоль:

```
double g = 9.8;
int h = ToInt32(g); // метод System.Convert
WriteLine($"g is {g} and h is {h}");
```

3. Запустите код и проанализируйте результат:

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

Одно из различий между приведением и преобразованием заключается в том, что при преобразовании значение `double 9,8` округляется до `10` вместо отбрасывания части после десятичной запятой.

Округление чисел

Вы убедились, что операция приведения типа отбрасывает десятичную часть вещественного числа, а при преобразовании с помощью метода `System.Convert` число округляется в бóльшую или меньшую сторону. Но каковы правила округления?

Правила округления

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

Посмотрим, следует ли язык C# тому же правилу начальной школы.

1. Добавьте операторы для объявления массива значений типа `double` и наполнения его значениями, преобразуйте каждый элемент массива в целое число, а затем запишите результат в консоль, как показано ниже:

```
double[] doubles = new[]
  { 9.49, 9.5, 9.51, 10.49, 10.5, 10.51 };

foreach (double n in doubles)
{
  WriteLine($"ToInt32({n}) is {ToInt32(n)}");
}
```

2. Запустите код и проанализируйте результат:

```
ToInt32(9.49) is 9
ToInt32(9.5) is 10
ToInt32(9.51) is 10
ToInt32(10.49) is 10
ToInt32(10.5) is 10
ToInt32(10.51) is 11
```

Обратите внимание, что правила округления в языке C# несколько отличаются от правила, принятого в начальной школе:

- число всегда округляется в *меньшую сторону*, если десятичная часть меньше 0,5;
- число всегда округляется в *бóльшую сторону*, если десятичная часть больше 0,5;
- число округляется в *бóльшую сторону*, если десятичная часть равна 0,5, а целая часть — нечетная, и округляется в *меньшую сторону*, если целая часть — четная.

Эта схема известна как *банковское округление* и является более предпочтительной, поскольку уменьшает вероятность ошибки. К сожалению, другие языки, например JavaScript, используют правила округления, принятые в начальной школе.

Контроль правил округления

Вы можете контролировать правила округления, используя метод `Round` класса `Math`.

1. Добавьте операторы для округления каждого из значений `double`, используя правило округления «от нуля», также известное как округление «в бóльшую сторону», а затем запишите результат в консоль, как показано в следующем примере:

```
foreach (double n in doubles)
{
  WriteLine(format:
    "Math.Round({0}, 0, MidpointRounding.AwayFromZero) is {1}",
    arg0: n,
    arg1: Math.Round(value: n, digits: 0,
            mode: MidpointRounding.AwayFromZero));
}
```

2. Запустите код и проанализируйте результат:

```
Math.Round(9.49, 0, MidpointRounding.AwayFromZero) is 9
Math.Round(9.5, 0, MidpointRounding.AwayFromZero) is 10
Math.Round(9.51, 0, MidpointRounding.AwayFromZero) is 10
Math.Round(10.49, 0, MidpointRounding.AwayFromZero) is 10
Math.Round(10.5, 0, MidpointRounding.AwayFromZero) is 11
Math.Round(10.51, 0, MidpointRounding.AwayFromZero) is 11
```

Проверяйте правила округления в каждом языке программирования, с которым работаете. Они могут работать не так, как вы ожидаете!

Преобразование значения 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 осмысленно представлены в виде текста, поэтому возвращают свое пространство имен и название типа.

Рассмотрим пример преобразования в тип `string`.

1. Введите операторы для объявления некоторых переменных, преобразуйте их в представление `string` и запишите их в консоль:

```
int number = 12;
WriteLine(number.ToString());

bool boolean = true;
WriteLine(boolean.ToString());

DateTime now = DateTime.Now;
WriteLine(now.ToString());

object me = new();
WriteLine(me.ToString());
```

2. Запустите код и проанализируйте результат:

```
12
True
02/28/2021 17:33:54
System.Object
```

Преобразование двоичного (бинарного) объекта в строку

Допустим, у вас есть двоичный (бинарный) объект, такой как изображение или видео, и вы хотите сохранить его или передать. В некоторых случаях вы не хотите отправлять необработанные биты, поскольку не знаете, как они могут быть истолкованы, например, сетевым протоколом, передающим этот объект, или другой операционной системой, которая считывает двоичный объект хранилища.

Самое безопасное — преобразовать двоичный объект в строку безопасных символов. Программисты называют это кодировкой *Base64*.

Методы `ToBase64String` и `FromBase64String` типа `Convert` выполняют такое преобразование.

Рассмотрим пример.

1. Добавьте операторы для создания массива байтов, случайно заполненного байтовыми значениями. Запишите в консоль каждый байт в красиво отформатированном виде, а затем те же байты, преобразованные в Base64, как показано ниже:

```
// выделение массива из 128 байт
byte[] binaryObject = new byte[128];

// заполнение массива случайными байтами
(new Random()).NextBytes(binaryObject);

WriteLine("Binary Object as bytes:");

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

// преобразование в строку Base64 и вывод в виде текста
string encoded = ToBase64String(binaryObject);

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

   По умолчанию значение `int` будет выводиться в десятичной записи, то есть base10. Вы можете использовать коды форматирования, такие как `:X`, чтобы отформатировать значение, используя шестнадцатеричную систему счисления.

2. Запустите код и проанализируйте результат:

```
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 +eUgYX5PeoHsSfBJHY7U99tU
r6CBBbi+zvg2kHrUNkIEdYEbq1HOBWOsInLedC9Xf8vnR7diw/QtYZOFGOoGFxKuRKgNuEyJha
k81eJG4FnJ3xCv7e+KobGN7kq+SO x5pQpfLzCHSsd/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`. Только несколько типов имеют метод `Parse`, включая все числовые типы и тип `DateTime`.

Рассмотрим пример.

1. Добавьте следующие операторы, чтобы разобрать целое число и значение даты и времени из строк, а затем запишите результат в консоль:

```
int age = int.Parse("27");
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}.");
```

2. Запустите код и проанализируйте результат:

```
I was born 27 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).

Существует множество других форматов даты и времени. О них можно прочитать на сайте: https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-date-and-time-format-strings#table-of-format-specifiers.

Ошибки при использовании метода Parse

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

1. Добавьте следующий оператор, чтобы попытаться превратить строку, содержащую буквы, в целочисленную переменную:

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

2. Запустите код и проанализируйте результат:

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

Как и в предыдущем сообщении об ошибке, вы увидите трассировку стека. Я не включил тему трассировок в эту книгу, поскольку она слишком большая.

Как избежать исключений с помощью метода TryParse

Чтобы избежать ошибок, вместо `Parse` вы можете использовать метод `TryParse`. Он пробует преобразовать исходную строку и, если это возможно, возвращает `true`. В противном случае возвращает `false`.

Ключевое слово `out` требуется для того, чтобы разрешить методу `TryParse` устанавливать переменную `count` при проведении преобразования.

Рассмотрим пример.

1. Замените объявление `int count` операторами, чтобы применить метод `TryParse`, и попросите пользователя ввести число для количества яиц, как показано ниже:

```
Write("How many eggs are there? ");
string? input = ReadLine(); // или используйте "12" в блокноте

if (int.TryParse(input, out int count))
{
  WriteLine($"There are {count} eggs.");
}
```

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

2. Запустите код, введите число `12` и проанализируйте результат:

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

3. Снова запустите код, введите `twelve` (или в блокноте измените значение `string` на «двенадцать») и проанализируйте результат:

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

Помимо этого, вы можете использовать методы типа `System.Convert` для преобразования значений `string` в другие типы. Однако, как и метод `Parse`, он выдает ошибку, если преобразование невозможно.

Обработка исключений

Вы ознакомились с несколькими сценариями, при выполнении которых возникали ошибки. В некоторых языках программирования при появлении ошибок возвращаются их коды. В .NET используются более разнообразные исключения, предназначенные только для отчетов об ошибках, в отличие от возвращаемых значений, которые имеют несколько вариантов применения. Когда это происходит, мы говорим, что *возникло исключение времени выполнения*.

При возникновении исключения поток приостанавливается, и если вызывающий код определил оператор `try-catch`, то ему предоставляется возможность обработать это исключение. Если текущий метод не обрабатывает его, то вызывающий метод перемещается далее по стеку вызовов.

Как вы видели, в случае ошибки консольное приложение или блокнот .NET Interactive по умолчанию выводит информацию об исключении, включая трассировку стека, и прекращает работу. Лучше так, чем позволить коду продолжать выполняться, когда он потенциально поврежден. Ваш код должен перехватывать и обрабатывать только те исключения, которые он понимает и может правильно исправить.

Старайтесь избегать написания кода, который будет выдавать исключение, когда это возможно; например, путем выполнения проверок оператора if. Но иногда это невозможно. В данном случае лучшим вариантом будет разрешить перехват исключения компонентом более высокого уровня, вызывающим ваш код. Более подробную информацию можно прочитать в главе 4.

Оборачивание потенциально ошибочного кода в оператор try

Предполагая, что оператор может вызвать ошибку, вы должны заключить его в блок `try`. Например, анализ строки для преобразования в число может вызвать ошибку. Любые операторы в блоке `catch` будут выполняться только в том случае, если в блоке `try` выбрасывается исключение.

В блоке `catch` можно не производить никаких операций. Рассмотрим пример.

1. Откройте редактор кода и создайте консольное приложение `HandlingExceptions` в рабочей области/решении `Chapter03`.
2. В программе Visual Studio Code выберите `HandlingExceptions` в качестве активного проекта OmniSharp.
3. Добавьте следующие операторы, чтобы предложить пользователю ввести свой возраст, а затем записать значение возраста в консоль:

```
WriteLine("Before parsing");
Write("What is your age? ");
string? input = ReadLine(); // или используйте значение "49" в блокноте

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

Вы увидите следующее сообщение компилятора: Warning CS8604 Possible null reference argument for parameter 's' in 'int int.Parse(string s)'. По умолчанию в новых проектах .NET 6 используются ссылочные типы, допускающие значение null, поэтому вы будете получать гораздо больше подобных предупреждений компилятора. В рабочих проектах следует добавить код для проверки значений null и их соответствующей обработки. Я не буду включать в книгу данный код, поскольку примеры не предназначены для использования в реальных приложениях, а проверки на null загромождают листинги и занимают много места. В нашем случае невозможно, чтобы ввод был null, поскольку пользователь должен нажать клавишу Enter, чтобы функция ReadLine вернула значение, а это вернет пустую строку. В примерах кода в данной книге вы увидите еще сотни примеров переменных, которые могут иметь значение null. Эти предупреждения можно игнорировать, если речь о примерах кода в этой книге. Вам нужны подобные предупреждения только тогда, когда вы пишете рабочий код для реальных приложений. Подробнее об обработке значений null вы узнаете в главе 6.

Данный код включает в себя два сообщения, которые нужно указывать *перед* синтаксическим анализом и *после* него, чтобы сделать процесс прохождения кода более понятным. Это будет особенно полезно, поскольку пример кода становится все более сложным.

4. Запустите код, введите `49` и проанализируйте результат:

```
Before parsing
What is your age? 49
You are 49 years old.
After parsing
```

5. Запустите код еще раз, введите `Kermit` и проанализируйте результат:

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

В процессе выполнения кода исключение ошибки было перехвачено, и сообщение по умолчанию и трассировка стека не выводились, а консольное приложение продолжало работать. Исключение было перехвачено, но хорошо бы увидеть тип ошибки, которая произошла.

В реальных приложениях никогда не следует использовать подобный пустой оператор catch, так как он «проглатывает» исключения и скрывает потенциальные проблемы. Вам необходимо хотя бы вести учет исключений, если вы не можете или не хотите обрабатывать его должным образом, или повторно выбросить его, чтобы код более высокого уровня мог принять решение. Более подробно о ведении журнала событий вы узнаете в главе 4.

Перехват всех исключений

Чтобы получить информацию о возможном исключении любого типа, вы можете объявить переменную типа `System.Exception` в блоке `catch`.

1. Добавьте объявление переменной исключения в блок `catch` и используйте его для записи информации об исключении в консоль:

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

2. Запустите код, снова введите `Kermit` и проанализируйте результат:

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

Перехват определенных исключений

Теперь, зная тип возникшей ошибки, вы можете улучшить код, перехватывая ошибку только данного типа и изменив сообщение, которое выводится пользователю.

1. Не трогая существующий блок `catch`, выше него добавьте новый блок `catch`, как показано ниже:

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

2. Запустите код, снова введите `Kermit` и проанализируйте результат:

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

Причина, по которой мы пишем универсальный блок `catch` в самом низу, состоит в том, что могут возникнуть исключения другого типа.

3. Запустите код, введите `9876543210` и проанализируйте результат:

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

Добавим еще один блок `catch` для этого типа исключения.

1. Не трогая существующий блок `catch`, добавьте новый блок `catch`:

```
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.");
}
```

2. Перезапустите код, введите `9876543210` и проанализируйте результат:

```
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 наследования исключений. С наследованием вы познакомитесь в главе 5, а пока не беспокойтесь об этом — компилятор отобразит ошибку сборки, если вы будете обрабатывать исключения в неправильном порядке.

По возможности старайтесь избегать перехвата исключений. Как правило, необходимо распространять стек вызовов для обработки на уровне, где содержится больше информации о работе кода, который может изменить логику обработки исключений.

Перехват с помощью фильтров

В оператор `catch` вы можете добавить фильтры, используя ключевое слово `when`:

```
Write("Enter an amount: ");
string? amount = ReadLine();
try
{
  decimal amountValue = decimal.Parse(amount);
}
catch (FormatException) when (amount.Contains("$"))
{
  WriteLine("Amounts cannot use the dollar sign!");
}
catch (FormatException)
{
  WriteLine("Amounts must only contain digits!");
}
```

Проверка переполнения

Ранее мы убедились, что при приведении между числовыми типами можно потерять данные, к примеру при приведении из переменной `long` к переменной `int`. Если значение, хранящееся в типе, слишком велико, то возникнет переполнение.

Выброс исключений переполнения с помощью оператора checked

Оператор `checked` дает .NET команду при возникновении переполнения вызывать исключение, вместо того чтобы позволять ему происходить автоматически, что делается по умолчанию из соображений производительности.

Определим начальное значение переменной `int` как максимальное значение минус один. Затем увеличим его несколько раз, выводя каждый раз значение переменной. Как только переменная станет больше своего максимального значения, код снова начнет работать от минимального значения, увеличивая каждый раз значение переменной.

Рассмотрим пример.

1. Откройте редактор кода и создайте консольное приложение `CheckingForOverflow` в рабочей области/решении `Chapter03`.
2. В программе Visual Studio Code выберите `CheckingForOverflow` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` добавьте операторы для объявления и назначения целого числа, которое на единицу меньше его максимально возможного значения, а затем увеличьте его и запишите его значение в консоль три раза, как показано ниже:

```
int x = int.MaxValue - 1;
WriteLine($"Initial value: {x}");
x++;
WriteLine($"After incrementing: {x}");
x++;
WriteLine($"After incrementing: {x}");
x++;
WriteLine($"After incrementing: {x}");
```

4. Запустите код и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
After incrementing: -2147483648
After incrementing: -2147483647
```

5. Теперь с помощью блока оператора `checked` научим компилятор предупреждать нас о переполнении, обернув операторы:

```
checked
{
  int x = int.MaxValue - 1;
  WriteLine($"Initial value: {x}");
  x++;
  WriteLine($"After incrementing: {x}");
  x++;
  WriteLine($"After incrementing: {x}");
  x++;
  WriteLine($"After incrementing: {x}");
}
```

6. Запустите код и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
Unhandled Exception: System.OverflowException: Arithmetic operation
resulted in an overflow.
```

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

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

8. Запустите код и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
The code overflowed but I caught the exception.
```

Отключение проверки переполнения с помощью оператора unchecked

В предыдущем подразделе мы говорили о дефолтном переполнении *во время выполнения* и о том, как использовать оператор `checked` для изменения этого поведения. Здесь вы узнаете о переполнении *во время компиляции* и о том, как с помощью оператора `unchecked` изменять это поведение.

Чтобы запретить проверку переполнения, можно использовать ключевое слово `unchecked`. Оно отключает проверки переполнения, выполняемые компилятором в блоке кода.

Рассмотрим пример.

1. Введите оператор, приведенный ниже, в конце предыдущих операторов. Компилятор не выполнит его, поскольку знает, что это приведет к переполнению:

```
int y = int.MaxValue + 1;
```

2. Наведите указатель мыши на ошибку и обратите внимание, что проверка *во время компиляции* отображается как сообщение об ошибке (рис. 3.1).

Рис. 3.1. Проверка в окне PROBLEMS (Проблемы) во время компиляции

3. Чтобы отключить проверки во время компиляции, поместите оператор в блок `unchecked`, запишите значение `y` в консоль, уменьшите его и повторите, как показано ниже:

```
unchecked
{
  int y = int.MaxValue + 1;
  WriteLine($"Initial value: {y}");
  y--;
  WriteLine($"After decrementing: {y}");
  y--;
  WriteLine($"After decrementing: {y}");
}
```

4. Запустите код и проанализируйте результат:

```
Initial value: -2147483648
After decrementing: 2147483647
After decrementing: 2147483646
```

Разумеется, вряд ли возникнет ситуация, когда вы захотите явно отключить проверку, поскольку будет допускаться переполнение. Но, возможно, в каком-то из ваших сценариев вам понадобится реализовать подобное поведение компилятора.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 3.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Что произойдет, если разделить переменную `int` на `0`?
2. Что произойдет, если разделить переменную `double` на `0`?
3. Что происходит при переполнении переменной `int`, то есть когда вы присваиваете ей значение, выходящее за пределы допустимого диапазона?
4. В чем разница между операторами `x = y++;` и `x = ++y;`?
5. В чем разница между операторами `break`, `continue` и `return` при использовании в операторах цикла?
6. Из каких трех частей состоит оператор `for` и какие из них обязательны?
7. В чем разница между операциями `=` и `==`?
8. Будет ли скомпилирован следующий оператор?

   ```
   for ( ; true; ) ;
   ```

9. Что представляет символ подчеркивания `_` в выражении `switch`?
10. Какой интерфейс должен реализовать объект, чтобы его можно было перечислить с помощью оператора `foreach`?

Упражнение 3.2. Циклы и переполнение

Что произойдет при выполнении кода, приведенного ниже?

```
int max = 500;
for (byte i = 0; i < max; i++)
{
  WriteLine(i);
}
```

Создайте в рабочей области/решении `Chapter03` консольное приложение `Exercise02` и введите этот код. Запустите консольное приложение и проанализируйте результат. Что произошло?

Какой код вы могли бы добавить (не изменяя строки предыдущего кода), чтобы получать предупреждение о проблеме?

Упражнение 3.3. Циклы и операции

FizzBuzz — командная детская игра, обучающая делению. Игроки по очереди называют натуральные числа, заменяя те, которые делятся на 3, словом *fizz*, а те, которые делятся на 5, — *buzz*. Если встречается число, делимое и на 3, и на 5, то говорят *fizzbuzz*.

Создайте в рабочей области/решении `Chapter03` консольное приложение `Exercise03`, вывод которого имитирует игру FizzBuzz, выполняющую счет до 100. Результат должен выглядеть примерно так, как показано на рис. 3.2.

Рис. 3.2. Смоделированный вывод игры FizzBuzz

Упражнение 3.4. Обработка исключений

Создайте в рабочей области/решении `Chapter03` консольное приложение `Exercise04`, которое запрашивает у пользователя два числа в диапазоне от 0 до 255, а затем делит первое на второе:

```
Enter a number between 0 and 255: 100
Enter another number between 0 and 255: 8
100 divided by 8 is 12
```

Напишите код обработчиков исключений, перехватывающих любые вызванные ошибки, как показано ниже:

```
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.
```

Упражнение 3.5. Проверка знания операций

Каковы будут значения переменных `x` и `y` после выполнения следующих операторов?

1. Операции инкремента и сложения:

```
x = 3;
y = 2 + ++x;
```

2. Операции бинарного сдвига:

```
x = 3 << 2;
y = 10 >> 1;
```

3. Побитовые операции:

```
x = 10 & 8;
y = 10 | 7;
```

Упражнение 3.6. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-3---controlling-flow-and-converting-types, чтобы получить дополнительную информацию по темам, приведенным в главе.

Резюме

В этой главе вы поэкспериментировали с операциями, узнали, как разветвлять и зацикливать код, выполнять преобразование типов и перехватывать исключения.

Теперь вы готовы узнать, как повторно использовать блоки кода, определяя функции, как передавать в них значения и возвращать их, а также как отслеживать ошибки в коде и удалять их!

4 Разработка, отладка и тестирование функций

В главе рассказывается, как писать функции для повторного использования кода, отлаживать логические ошибки в процессе разработки, регистрировать исключения во время выполнения. Кроме того, вы узнаете, как с помощью модульного тестирования кода устранять ошибки и обеспечивать стабильность и надежность кода.

В этой главе:

- написание функций в языке C#;
- отладка в процессе разработки;
- ведение журнала событий во время выполнения;
- модульное тестирование;
- генерация и перехват исключений в функциях.

Написание функций

Фундаментальный принцип программирования — *DRY* (Do not Repeat Yourself — «Не повторяйся»).

Если во время программирования вы многократно пишете одни и те же операторы, то их следует превращать в функцию. Функции похожи на крошечные программы, каждая из которых выполняет одну маленькую задачу. Например, вы можете написать функцию для расчета НДС, а затем повторно многократно использовать ее в финансовом приложении.

Как и программы, функции обычно имеют ввод и вывод. Их иногда называют черными ящиками, в которые вы загружаете исходные данные с одной стороны, а результат появляется с другой. После того как функции созданы, вам не нужно думать о том, как они работают.

Пример таблицы умножения

Допустим, вы хотите помочь своему ребенку выучить таблицу умножения. Для этого вы хотите упростить создание таблицы умножения для конкретного числа, скажем, для 12[1]:

```
1 x 12 = 12
2 x 12 = 24
...
12 x 12 = 144
```

Ранее вы уже ознакомились с оператором `for`, поэтому знаете, что его можно использовать для генерации повторяющихся строк вывода, когда имеется регулярный шаблон, например таблица умножения на 12, как показано ниже:

```
for (int row = 1; row <= 12; row++)
{
  Console.WriteLine($"{row} x 12 = {row * 12}");
}
```

Однако вместо вывода таблицы умножения на 12 мы хотим сделать ее более гибкой, чтобы она могла выводить результаты перемножения для любого числа. Мы можем сделать это, создав функцию.

Функции для таблицы умножения

Изучим функции, создав одну для вывода любой таблицы умножения для чисел от 0 до 255, умноженных на числа от 1 до 12:

1. Откройте редактор кода и создайте консольное приложение с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter04`;
   3) файл и папка проекта: `WritingFunctions`.
2. Статически импортируйте пространство имен `System.Console`.
3. В файле `Program.cs` добавьте операторы для определения функции `TimesTable`:

```
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}");
  }
```

[1] В британских школах традиционно изучаются таблицы умножения до 12, а не до 10, как в России и других русскоязычных странах.

```
  WriteLine();
}
```

В этом коде обратите внимание на ряд моментов, касающихся функции `TimesTable`:

- ей должно быть передано байтовое (`byte`) значение `number`;
- эта функция статическая, поскольку вызывается статическим методом `Main`;
- она не возвращает значение вызывающей стороне, поэтому оно объявляется с ключевым словом `void` перед его именем;
- эта функция использует оператор `for`, чтобы вывести таблицу умножения для переданного ей числа.

4. После оператора, который статически импортирует класс `Console`, и перед `TimesTable` вызовите функцию и для параметра `number` передайте значение типа `byte`, например, `6`:

```
using static System.Console;

TimesTable(6);
```

Если функция имеет один или несколько параметров, не имеющих особого смысла, то можете дополнительно указать имя параметра и его значение, например: TimesTable(number: 6).

5. Запустите код и проанализируйте результат:

```
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
```

6. Измените число, переданное в функцию `TimesTable`, на другие значения от `0` до `255` и убедитесь, что вывод таблицы корректен.
7. Обратите внимание, что если передать число другого типа, например `int`, `double` или `string`, то будет возвращена следующая ошибка:

```
Error: (1,12): error CS1503: Argument 1: cannot convert from 'int' to
'byte'
```

Функции, возвращающие значение

Предыдущая функция выполняла действия (цикл и запись в консоль), но не возвращала значение. Допустим, вам нужно рассчитать налог с продаж или налог на добавленную стоимость (НДС). В Европе ставки НДС могут варьироваться от 8 % в Швейцарии до 27 % в Венгрии. В США государственные налоги с продаж могут варьироваться от 0 % в Орегоне до 8,25 % в Калифорнии.

Налоговые ставки постоянно варьируются в зависимости от многих факторов. Нет необходимости сообщать мне, что налоговая ставка, например, в Вирджинии составляет 6 %. Благодарю.

Реализуем функцию для расчета налогов в различных регионах мира.

1. Добавьте функцию `CalculateTax`.

```
static decimal CalculateTax(
  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": // Мэн
```

```
    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;
}
```

Обратите внимание, что функция `CalculateTax`:

- имеет два входных параметра: `amount`, который будет представлен суммой потраченных денег, и `twoLetterRegionCode`, представленный регионом, в котором потрачена сумма;
- выполнит вычисление с помощью оператора `switch`, а затем вернет налог с продаж или НДС на сумму в виде десятичного значения (`decimal`); итак, перед именем функции мы объявили тип данных возвращаемого значения `decimal`.

2. Закомментируйте вызов метода `TimesTable` и вызовите метод `CalculateTax`, введите сумму, например `149`, и код региона, например `FR`:

```
// TimesTable(6);

decimal taxToPay = CalculateTax(amount: 149, twoLetterRegionCode: "FR");
WriteLine($"You must pay {taxToPay} in tax.");
```

3. Запустите код и проанализируйте результат.

```
You must pay 29.8 in tax.
```

Можно форматировать переменную taxToPay в валюту, используя {taxToPay:C}, но в данном случае формат валюты будет соответствовать вашему региону. Например, для Великобритании результат будет следующим: 29,80 GBP.

Можете ли вы представить какие-либо проблемы с функцией `CalculateTax`? Что произойдет, если пользователь введет код, например, `fr` или `UK`? Как вы могли бы переписать функцию, чтобы улучшить ее? Будет ли применение выражения `switch` вместо оператора `switch` более понятным?

Преобразование чисел из кардинального в порядковое

Числа, которые используются для подсчета, называются *кардинальными*, например 1, 2 и 3. В то же время числа, применяемые для обозначения порядка, называются *порядковыми*, например 1-е, 2-е и 3-е.

Создадим функцию для преобразования кардинальных чисел в порядковые.

1. Напишите функцию `CardinalToOrdinal`, которая преобразует кардинальное значение типа `int` в порядковое значение `string`; например, преобразует `1` в `1st`, `2` в `2nd` и т. д., как показано ниже:

```
static string CardinalToOrdinal(int number)
{
  int lastTwoDigits = number % 100;
  switch (lastTwoDigits)
  {
    case 11:// особые случаи с 11-го по 13-й
    case 12:
    case 13:
      return $"{number}th";
    default:
      int lastDigit = number % 10;

      string suffix = lastDigit switch
      {
        1 => "st",
        2 => "nd",
        3 => "rd",
        _ => "th"
      };
      return $"{number}{suffix}";
  }
}
```

   В этом коде обратите внимание на следующие моменты:

   - функция `CardinalToOrdinal` имеет один вход — параметр типа `int` с именем `number` и один выход — возвращаемое значение типа `string`;
   - *оператор* `switch` используется для обработки особых случаев — `11`, `12` и `13`;
   - затем *выражение* `switch` обрабатывает все остальные случаи: если последней цифрой является `1`, то используется суффикс `st`, если `2` — то `nd`, если `3` — то `rd`, если любая другая цифра, то используется суффикс `th`.

2. Создайте функцию `RunCardinalToOrdinal`, которая использует оператор `for` для перебора значений от 1 до 40, вызывая функцию `CardinalToOrdinal` для каждого числа и записывая возвращаемую строку в консоль, разделенную пробелом:

```
static void RunCardinalToOrdinal()
{
  for (int number = 1; number <= 40; number++)
  {
    Write($"{CardinalToOrdinal(number)} ");
  }
  WriteLine();
}
```

3. Закомментируйте вызов метода `CalculateTax` и вызовите метод `RunCardinalToOrdinal`:

```
// TimesTable(6);

// decimal taxToPay = CalculateTax(amount: 149, twoLetterRegionCode: "FR");
// WriteLine($"You must pay {taxToPay} in tax.");

RunCardinalToOrdinal();
```

4. Запустите код и проанализируйте результат:

```
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 равен 120. Факториал натурального числа — это число, умноженное на «себя минус один», затем на «себя минус два» и т. д. до 1. Пример: $5 \times 4 \times 3 \times 2 \times 1 = 120$.

Факториалы записываются следующим образом: 5!, где *факториал* обозначается восклицательным знаком. Пример: 5! = 120, то есть *факториал пяти равен ста двадцати*. Восклицательный знак — хорошее обозначение для факториалов, поскольку они очень быстро увеличиваются в размерах.

Мы напишем функцию `Factorial`, которая вычислит факториал для `int`, переданного ей в качестве параметра. Мы будем использовать технику, называемую *рекурсией*, то есть функцию, вызывающую саму себя в своей реализации, прямо или косвенно.

1. Добавьте функцию `Factorial` и функцию для ее вызова:

```
static int Factorial(int number)
{
  if (number < 1)
```

```
  {
    return 0;
  }
  else if (number == 1)
  {
    return 1;
  }
  else
  {
    return number * Factorial(number - 1);
  }
}
```

Как и прежде, в приведенном выше коде есть несколько важных моментов:

- если введенное число равно нулю или отрицательное, то функция `Factorial` возвращает `0`;
- если введенное число равно `1`, то функция `Factorial` возвращает `1` и, следовательно, перестает вызывать сама себя;
- если входной параметр больше единицы, то функция `Factorial` умножает число на результат вызова самой себя с переданным в качестве параметра числом, на единицу меньшим. Это делает функцию рекурсивной.

Рекурсия может показаться разумным решением, однако в результате использования она может вызвать переполнение стека из-за слишком большого количества вызовов функций, поскольку при каждом ее вызове для хранения данных используется слишком много памяти. Итерация — более практичное, хотя и менее лаконичное решение в таких языках, как C#. Более подробную информацию вы можете найти на сайте https://en.wikipedia.org/wiki/Recursion_(computer_science)#Recursion_versus_iteration.

2. Добавьте функцию `RunFactorial`, которая использует оператор `for` для вывода факториалов чисел от 1 до 14, вызывает функцию `Factorial`, а затем выводит результат, отформатированный с помощью кода `N0`, который означает форматирование числа и применение разделителей тысяч с нулевыми десятичными разрядами и повторяется в цикле, пока не будет введено другое число:

```
static void RunFactorial()
{
  for (int i = 1; i < 15; i++)
  {
    WriteLine($"{i}! = {Factorial(i):N0}");
  }
}
```

3. Закомментируйте вызов метода `RunCardinalToOrdinal` и вызовите метод `RunFactorial`.

4. Запустите код и проанализируйте результат:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5,040
8! = 40,320
9! = 362,880
10! = 3,628,800
11! = 39,916,800
12! = 479,001,600
13! = 1,932,053,504
14! = 1,278,945,280
```

В предыдущем выводе это не сразу очевидно, но факториалы от 13 и выше будут перекрывать тип `int`, так как они слишком большие. 12! = 479 001 600, что составляет около полумиллиарда. Максимальное положительное значение, которое может быть сохранено в переменной типа `int`, составляет около двух миллиардов. 13! = 6 227 020 800, что составляет около шести миллиардов, и при сохранении в виде 32-битного целого числа оно переполняется без каких-либо проблем.

Что делать, чтобы получать уведомление о переполнении?

Конечно, мы смогли решить проблему, касающуюся 13! и 14!, используя 64-битное целое число `long` вместо 32-битного целого числа `int`, но мы снова быстро достигнем предела переполнения.

Цель этого подраздела — помочь вам понять, что числа могут выходить за границы, и показать, как с этим справиться, а не как вычислять факториалы!

1. Измените функцию `Factorial` для проверки переполнения:

```
checked // для переполнения
{
  return number * Factorial(number - 1);
}
```

2. Измените функцию `RunFactorial` для обработки исключения переполнения при вызове функции `Factorial`:

```
try
{
  WriteLine($"{i}! = {Factorial(i):N0}");
}
catch (System.OverflowException)
{
  WriteLine($"{i}! is too big for a 32-bit integer.");
}
```

3. Запустите код и проанализируйте результат:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5,040
8! = 40,320
9! = 362,880
10! = 3,628,800
11! = 39,916,800
12! = 479,001,600
13! is too big for a 32-bit integer.
14! is too big for a 32-bit integer.
```

Документирование функций с помощью XML-комментариев

По умолчанию при вызове такой функции, как `CardinalToOrdinal`, редактор кода отображает всплывающую подсказку с основной информацией (рис. 4.1).

Рис. 4.1. Всплывающая подсказка, по умолчанию отображающая запись простого метода

Усовершенствуем всплывающую подсказку, добавив дополнительную информацию.

1. Если вы используете Visual Studio Code с расширением `C#`, то выберите команду меню View ▸ Command Palette ▸ Preferences: Open Settings (UI) (Вид ▸ Палитра команд ▸ Настройки: Открыть настройки (UI)). Затем найдите команду `formatOnType` и убедитесь, что она добавлена. Комментарии к XML-документации в языке C# — это встроенная функция Visual Studio 2022.
2. В строке над функцией `CardinalToOrdinal` введите три косые черты и обратите внимание, что расширение превратит это в комментарий XML и распознает наличие у метода одного параметра `number`.

3. К XML-документу введите подходящий комментарий для описания входного параметра и возвращаемого значения для функции `CardinalToOrdinal`, как показано ниже:

```
/// <summary>
/// Передайте 32-битное целое число, и оно будет преобразовано
/// в его порядковый эквивалент.
/// </summary>
/// <param name="number">Number is a cardinal value e.g. 1, 2, 3 and so on.
</param>
/// <returns>Number as an ordinal value e.g. 1st, 2nd, 3rd and so on.
</returns>
```

4. Теперь при вызове функции вы увидите более подробную информацию (рис. 4.2).

Рис. 4.2. Всплывающая подсказка, отображающая более подробную сигнатуру метода

На момент написания шестого издания комментарии XML-документации языка C# не работали в блокнотах .NET Interactive.

Добавляйте XML-комментарии в код для всех ваших функций.

Использование лямбда-выражений в реализациях функций

F# — первый строго типизированный функциональный язык программирования Microsoft, который, как и C#, для выполнения .NET компилируется в IL. Функциональные языки произошли от лямбда-исчисления; вычислительная система основана только на функциях. Код больше похож на математические функции, чем на пошаговый рецепт.

Ниже представлены некоторые из важных особенностей функциональных языков.

- *Модульность*. Это же преимущество определения функций в C# применимо и к функциональным языкам. Разделите большой сложный код на более мелкие части.

- *Неизменяемость*. Переменных в контексте C# не существует. Любое значение данных внутри функции не может измениться. Вместо этого новое значение данных может быть создано из существующего. Это уменьшает количество ошибок.
- *Сопровождаемость*. Код чище и понятнее (для склонных к математике программистов!).

Начиная с версии C# 6, компания Microsoft работает над добавлением функций в язык для поддержки более функционального подхода. Например, добавление *кортежей* и *сопоставление с образцом* в версии C# 7, *ненулевые ссылочные типы* в C# 8, улучшение сопоставления с образцом и добавление записей, то есть *неизменяемых объектов* в C# 9.

В C# 6 Microsoft добавила поддержку *членов функций, содержащих выражения*. Рассмотрим это на примере.

Последовательность чисел Фибоначчи всегда начинается с 0 и 1. Затем остальная часть последовательности генерируется с использованием правила сложения двух предыдущих чисел, как показано ниже:

```
0 1 1 2 3 5 8 13 21 34 55 ...
```

Очередным членом в последовательности будет 34 + 55, то есть 89.

Воспользуемся последовательностью Фибоначчи, чтобы продемонстрировать разницу между императивной и декларативной реализацией функции.

1. Добавьте императивную функцию `FibImperative`, как показано ниже:

```
static int FibImperative(int term)
{
  if (term == 1)
  {
    return 0;
  }
  else if (term == 2)
  {
    return 1;
  }
  else
  {
    return FibImperative(term - 1) + FibImperative(term - 2);
  }
}
```

2. Добавьте функцию `RunFibImperative`, вызывающую функцию `FibImperative` внутри оператора `for`, выполняющего цикл от 1 до 30:

```
static void RunFibImperative()
{
  for (int i = 1; i <= 30; i++)
```

```
  {
    WriteLine("The {0} term of the Fibonacci sequence is {1:N0}.",
      arg0: CardinalToOrdinal(i),
      arg1: FibImperative(term: i));
  }
}
```

3. Закомментируйте другие вызовы метода и вызовите метод `RunFibImperative`.
4. Запустите код и проанализируйте результат:

```
The 1st term of the Fibonacci sequence is 0.
The 2nd term of the Fibonacci sequence is 1.
The 3rd term of the Fibonacci sequence is 1.
The 4th term of the Fibonacci sequence is 2.
The 5th term of the Fibonacci sequence is 3.
The 6th term of the Fibonacci sequence is 5.
The 7th term of the Fibonacci sequence is 8.
The 8th term of the Fibonacci sequence is 13.
The 9th term of the Fibonacci sequence is 21.
The 10th term of the Fibonacci sequence is 34.
The 11th term of the Fibonacci sequence is 55.
The 12th term of the Fibonacci sequence is 89.
The 13th term of the Fibonacci sequence is 144.
The 14th term of the Fibonacci sequence is 233.
The 15th term of the Fibonacci sequence is 377.
The 16th term of the Fibonacci sequence is 610.
The 17th term of the Fibonacci sequence is 987.
The 18th term of the Fibonacci sequence is 1,597.
The 19th term of the Fibonacci sequence is 2,584.
The 20th term of the Fibonacci sequence is 4,181.
The 21st term of the Fibonacci sequence is 6,765.
The 22nd term of the Fibonacci sequence is 10,946.
The 23rd term of the Fibonacci sequence is 17,711.
The 24th term of the Fibonacci sequence is 28,657.
The 25th term of the Fibonacci sequence is 46,368.
The 26th term of the Fibonacci sequence is 75,025.
The 27th term of the Fibonacci sequence is 121,393.
The 28th term of the Fibonacci sequence is 196,418.
The 29th term of the Fibonacci sequence is 317,811.
The 30th term of the Fibonacci sequence is 514,229.
```

5. Добавьте декларативную функцию `FibFunctional`:

```
static int FibFunctional(int term) =>
  term switch
  {
    1 => 0,
    2 => 1,
    _ => FibFunctional(term - 1) + FibFunctional(term - 2)
  };
```

6. Добавьте функцию и вызовите ее внутри оператора `for`, выполняющего цикл от 1 до 30:

```
static void RunFibFunctional()
{
  for (int i = 1; i <= 30; i++)
  {
    WriteLine("The {0} term of the Fibonacci sequence is {1:N0}.",
      arg0: CardinalToOrdinal(i),
      arg1: FibFunctional(term: i));
  }
}
```

7. Закомментируйте вызов метода `RunFibImperative` и вызовите его.
8. Запустите код и проанализируйте результат (который будет таким же, как раньше).

Отладка в процессе разработки

В данном разделе вы научитесь отлаживать ошибки во время разработки. Необходимо использовать редактор кода с инструментами отладки, например Visual Studio или Visual Studio Code. На момент написания книги вы не можете использовать .NET Interactive Notebooks для отладки кода, но в будущем эта возможность будет доступна.

Настройка отладчика OmniSharp для Visual Studio Code может оказаться сложной задачей. Я включил инструкции для наиболее распространенных проблем, но если вы все еще в затруднении, то можете изучить документацию на сайте https://github.com/OmniSharp/omnisharp-vscode/blob/master/debugger.md.

Преднамеренное добавление ошибок в код

Рассмотрим отладку, создав консольное приложение с преднамеренной ошибкой, для отслеживания и исправления которой мы затем будем использовать специальный инструментарий.

1. Откройте редактор кода и создайте консольное приложение `Debugging` в рабочей области/решении `Chapter04`.
2. В Visual Studio Code выберите `Debugging` в качестве активного проекта OmniSharp. Когда вы увидите всплывающее предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку Yes (Да), чтобы добавить их.

3. В Visual Studio настройте стартовый проект для решения в соответствии с текущим выбором.
4. В файле `Program.cs` добавьте функцию с преднамеренной ошибкой:

```
static double Add(double a, double b)
{
  return a * b; // Преднамеренная ошибка!
}
```

5. Под функцией `Add` добавьте операторы для объявления некоторых переменных, а затем сложите их вместе с помощью ошибочной функции:

```
double a = 4.5;
double b = 2.5;
double answer = Add(a, b);
WriteLine($"{a} + {b} = {answer}");

WriteLine("Press ENTER to end the app.");
ReadLine(); // ожидание нажатия клавиши ENTER пользователем
```

6. Запустите консольное приложение и проанализируйте результат:

```
4.5 + 2.5 = 11.25
```

В этом коде обнаружена следующая ошибка: `4.5` плюс `2.5` должно в результате давать `7`, но никак не `11.25`!

Мы воспользуемся инструментами отладки, чтобы расправиться с этой ошибкой.

Установка точек останова и начало отладки

Точки останова позволяют помечать строки кода, на которых следует приостановить выполнение программы, чтобы найти ошибки.

Использование Visual Studio 2022

1. Установим точку останова, а затем начнем отладку с помощью Visual Studio 2022.
2. Щелкните правой кнопкой мыши на операторе, который определяет переменную `a`.
3. Выберите команду меню Debug ▸ Toggle Breakpoint (Отладка ▸ Переключить точку останова) или нажмите клавишу F9. Слева на полях появится красный кружок, указывающий на то, что точка останова была установлена (рис. 4.3).

Рис. 4.3. Переключение точек останова в Visual Studio 2022

Точки останова можно устанавливать и убирать с помощью клавиши F9, а также щелкнув на поле левой кнопкой мыши. А если на точке останова щелкнуть правой кнопкой мыши, то вы увидите дополнительные команды, например выключение или изменение параметров точки останова.

4. Выберите команду меню Debug ▸ Start Debugging (Отладка ▸ Начать отладку) или нажмите клавишу F5. Visual Studio запустит консольное приложение, а затем приостановит выполнение при обнаружении точки останова. Это называется режимом приостановки выполнения. Затем появятся следующие дополнительные окна: Locals (Локальные переменные) (отображающие текущие значения локальных переменных), Watch 1 (Отслеживаемые выражения 1) (отображающие любые определеные вами отслеживаемые выражения), Call Stack (Вызов стека), Exception Settings (Настройки исключений) и Immediate Window (Окно интерпретации). Появится также панель инструментов Debugging (Отладка). Строка кода, которая будет выполнена следующей, подсвечивается желтым, и на нее указывает стрелка такого же цвета, размещенная на поле (рис. 4.4).

Рис. 4.4. Режим приостановки в Visual Studio 2022

Если вы не хотите знать, как с помощью Visual Studio Code начать отладку, то можете пропустить следующий пункт и перейти сразу к подразделу «Навигация с помощью панели средств отладки» далее в этой главе.

Использование Visual Studio Code

Установим точку останова и начнем процесс отладки с помощью Visual Studio Code.

1. Щелкните правой кнопкой мыши на операторе, определяющем переменную a.
2. Выберите команду меню Run ▸ Toggle Breakpoint (Выполнить ▸ Переключить точку останова) или нажмите клавишу F9. Слева на полях появится красный кружок, указывающий на то, что точка останова была установлена (рис. 4.5).

Рис. 4.5. Переключение точек останова в Visual Studio Code

Точки останова можно устанавливать и убирать с помощью клавиши F9 или щелкнув на поле левой кнопкой мыши. А если на точке останова щелкнуть правой кнопкой мыши, то вы увидите дополнительные команды, например выключение или изменение параметров точки останова.

Logpoints, также известные как точки трассировки, указывают на то, что вы хотите записать некую информацию, не прибегая к фактической остановке выполнения кода в данной точке.

3. Выберите команду меню View ▸ Run (Вид ▸ Выполнить) или можете нажать кнопку Run and Debug (Выполнение и отладка) (треугольная кнопка воспроизведения и значок ошибки), расположенную на левой панели навигации (см. рис. 4.5).
4. В верхней части панели DEBUG (Отладка) откройте раскрывающийся список справа от кнопки Start Debugging (Начать отладку) (зеленая треугольная кнопка воспроизведения) и выберите пункт .NET Core Launch (console) (Debugging) (рис. 4.6).

Рис. 4.6. Выбор проекта для отладки в Visual Studio Code

Если в раскрывающемся списке отсутствуют данные для проекта Debugging, это означает, что в данном проекте отсутствуют необходимые для отладки ресурсы. Эти данные хранятся в папке .vscode. Чтобы создать эту папку для проекта, выберите команду меню View ▸ Command Palette (Вид ▸ Палитра команд), выберите OmniSharp: Select Project, а затем выберите проект Debugging. Через несколько секунд появится запрос Required assets to build and debug are missing from 'Debugging'. Add them? (Необходимые ресурсы для сборки и отладки в разделе Debugging (Отладка) отсутствуют. Добавить их?). Нажмите Yes (Да), чтобы добавить их.

5. В верхней части панели DEBUG (Отладка) нажмите кнопку Start Debugging (Начать отладку) (зеленая треугольная кнопка) и выберите команду меню Run ▸ Start Debugging (Выполнить ▸ Начать отладку) или нажмите клавишу F5. Программа Visual Studio Code запускает консольное приложение, а затем при достижении точки останова приостанавливает его работу. Это называется режимом приостановки выполнения. Строка кода, которая будет выполнена следующей, подсвечивается желтым цветом, и на нее указывает стрелка такого же цвета, размещенная на поле (рис. 4.7).

Рис. 4.7. Режим приостановки в Visual Studio Code

Навигация с помощью панели средств отладки

Программа Visual Studio Code содержит специальную панель с дополнительными элементами управления, упрощающими доступ к средствам отладки. Программа Visual Studio 2022 содержит одну кнопку на стандартной панели инструментов, позволяющую запускать или продолжать отладку, и отдельную панель инструментов отладки для остальных средств.

На рис. 4.8 показаны некоторые из элементов управления.

Рис. 4.8. Панели инструментов отладки в Visual Studio 2022 и Visual Studio Code

- Continue (Продолжить)/F5 — запускает код с текущей позиции до достижения следующей точки останова.
- Step Over (Перешагнуть)/F10, Step Into (Шагнуть внутрь)/F11 и Step Out (Шагнуть наружу)/Shift+F11 (синие стрелки над точками) — различными способами пошагово выполняют код (будет рассмотрено далее).
- Restart (Перезапустить)/Ctrl+Shift+F5 или Cmd+Shift+F5 (круглая стрелка) — завершает выполнение и сразу вновь запускает программу.
- Stop (Остановить)/Shift+F5 (красный квадрат) — прекращает выполнение сеанса отладки.

Панели отладки

Во время отладки в Visual Studio Code и в Visual Studio отображаются дополнительные окна, позволяющие по мере выполнения кода отслеживать полезную информацию, например переменные.

Ниже перечислены наиболее полезные окна.

- VARIABLES (Переменные), включая Locals (Локальные), отображает имена, значение и типы всех используемых локальных переменных. Отслеживайте содержимое этого окна, когда отлаживаете свой код.
- WATCH или WATCH 1 (Наблюдение) отображает значения переменных и выражений, которые вы вводите вручную.
- CALL STACK (Стек вызовов) отображает стек вызовов функций.
- BREAKPOINTS (Точки останова) отображает все ваши точки останова, что позволяет лучше контролировать их.

В режиме приостановки в нижней части области редактирования также имеется полезная панель.

DEBUG CONSOLE (Консоль отладки) или Immediate Window (Окно непосредственной отладки) обеспечивает интерактивное взаимодействие с вашим кодом. Вы можете запросить состояние программы, например введя имя переменной. Здесь вы можете задать вопрос и получить на него ответ. К примеру, чтобы спросить «Сколько будет 1 + 2?», надо набрать 1+2 и нажать клавишу Enter (рис. 4.9).

Рис. 4.9. Запрос состояния программы

Пошаговое выполнение кода

Рассмотрим несколько способов пошагового выполнения кода.

1. Выберите команду меню Run/Debug ▸ Step Into (Выполнить/Отладка ▸ Шагнуть внутрь), или нажмите кнопку Step Into (Шагнуть внутрь) на панели инструментов, или нажмите клавишу F11. Желтая подсветка переместится на одну строку кода.
2. Выберите команду меню Run/Debug ▸ Step Over (Выполнить/Отладка ▸ Перешагнуть), или нажмите кнопку Step Over (Перешагнуть) на панели инструментов, или нажмите клавишу F10. Желтая подсветка переместится на одну строку кода. На данный момент вы можете видеть, что нет никакой разницы между использованием команды Step Into (Шагнуть внутрь) или Step Over (Перешагнуть).
3. Теперь вы окажетесь на строке вызова метода Add (рис. 4.10).

Рис. 4.10. Пошаговое выполнение кода

Разницу между командами Step Into (Шагнуть внутрь) и Step Over (Перешагнуть) можно увидеть, когда следующим оператором является вызов метода.

- Если вы нажмете кнопку Step Into (Шагнуть внутрь), то отладчик войдет *в* метод, так что вы сможете пройти по каждой его строке.
- Если вы нажмете кнопку Step Over (Перешагнуть), то отладчик выполнит метод *целиком*. Отладчик не пропустит метод, не выполнив его.

4. Нажмите кнопку Step Into (Шагнуть внутрь).
5. Наведите указатель мыши на параметры a или b в окне редактирования кода и обратите внимание на всплывающую подсказку, показывающую их текущее значение.
6. Выберите выражение a * b, щелкните на нем правой кнопкой мыши и выберите Add to Watch (Добавить в контрольное значение) или Add Watch (Добавить контрольное значение). Выражение добавляется в окно WATCH (Контрольное значение), показывая, что эта операция умножает a на b и выдает результат 11.25.
7. В окне WATCH (Контрольное значение) или Watch 1 (Контрольное значение 1) щелкните правой кнопкой мыши на выражении и выберите Remove Expression (Удалить выражение) или Delete Watch (Удалить контрольное значение).
8. Исправьте ошибку, сменив операцию * на + в функции Add.
9. Остановите, перекомпилируйте и перезапустите отладку. Для этого нажмите кнопку с круговой стрелкой Restart (Перезагрузить) либо сочетание клавиш Ctrl+Shift+F5 или Cmd+Shift+F5.
10. Перешагните функцию, при этом заметив, что она теперь правильно рассчитывается, затем нажмите кнопку Continue (Продолжить) или клавишу F5.

11. В программе Visual Studio Code обратите внимание, что при записи в консоль во время отладки вывод отображается на панели DEBUG CONSOLE (Консоль отладки) вместо панели TERMINAL (Терминал), как показано на рис. 4.11.

Рис. 4.11. Запись на панели DEBUG CONSOLE (Консоль отладки) во время отладки

Настройка точек останова

Вы можете легко настроить более сложные точки останова.

1. Если вы все еще выполняете отладку кода, то нажмите кнопку Stop (Стоп), или выберите команду меню Run/Debug ▸ Stop Debugging (Выполнить/Отладка ▸ Остановить отладку), или нажмите комбинацию клавиш Shift+F5.
2. Выберите команду меню Run ▸ Remove All Breakpoints (Выполнить ▸ Удалить все точки останова) или Debug ▸ Delete All Breakpoints (Отладка ▸ Удалить все точки останова).
3. Щелкните кнопкой мыши на операторе `WriteLine`, который выводит ответ.
4. Установите точку останова, нажав клавишу F9 или выбрав команду меню Run/Debug ▸ Toggle Breakpoint (Выполнить/Отладка ▸ Переключить точку останова).
5. В программе Visual Studio Code щелкните правой кнопкой мыши на точке останова и выберите Edit Breakpoint (Изменить точку останова). Введите выражение, например, «переменная `answer` должна быть больше 9», нажмите `Enter` и обратите внимание, что для активации точки останова выражение должно иметь значение `true` (рис. 4.12).
6. В программе Visual Studio щелкните правой кнопкой мыши на точке останова и выберите Conditions (Условия). Введите выражение, например, «переменная `answer` должна быть больше 9», нажмите `Enter` и обратите внимание, что для активации точки останова выражение должно иметь значение `true`.
7. Начните отладку и обратите внимание, что точка останова не достигнута.
8. Остановите отладку.

Рис. 4.12. Настройка точек останова в Visual Studio Code

9. Измените точку останова или ее условия и установите ее выражение в значение меньше 9.
10. Начните отладку и обратите внимание, что точка останова достигнута.
11. Остановите отладку.
12. Измените точку останова или ее условия (в Visual Studio выберите Add condition (Добавить условие)) и выберите пункт Hit Count (Количество обращений), затем введите число, например, 3. Это значит, что вам нужно будет достичь точки останова три раза, прежде чем она активируется (рис. 4.13).

Рис. 4.13. Настройка точки останова с помощью выражения и горячего счетчика с помощью Visual Studio 2022

13. Чтобы увидеть результат, наведите указатель мыши на красный кружок точки останова (рис. 4.14).

Теперь вы исправили ошибку, используя отдельные средства отладки, и увидели некоторые расширенные возможности для установки точек останова.

Рис. 4.14. Настроенные точки останова в Visual Studio Code

Ведение журнала событий во время разработки и выполнения проекта

Если вы считаете, что все ошибки были удалены из вашего кода, то должны скомпилировать релизную версию и развернуть приложение, чтобы люди могли его использовать. Однако не существует кода без ошибок, и во время выполнения они могут возникать непредвиденно.

Как известно, конечные пользователи плохо запоминают и неохотно и неточно описывают свои действия в момент, когда произошла ошибка. Поэтому вам не следует полагаться на то, что они предоставят полезную информацию, которая позволит воспроизвести проблему, чтобы понять ее причину, а затем исправить ее. Вместо этого вы можете *инструментировать свой код* или вести журнал необходимых событий[1].

Добавляйте в свое приложение код для логирования событий, особенно при возникновении исключений, чтобы вы могли просматривать события и с их помощью отслеживать и устранять проблему. В главах 10 и 15 мы еще встретимся с логированием, но в целом это обширная тема, так что в книге мы сможем изучить только ее основы.

Ведение журнала событий

Платформа .NET включает в себя несколько встроенных способов инструментирования вашего кода путем добавления функций ведения журнала. В этой книге мы рассмотрим только основные из них. Однако логирование — это область, в которой создана богатая система мощных решений, расширяющих возможности Microsoft.

[1] Применяется также термин «логирование» (от англ. logging) или «журналирование».

Я не могу дать конкретные рекомендации, так как структура ведения журнала зависит от ваших потребностей. В книге я приведу лишь общие рекомендации:

- Apache log4net;
- NLog;
- Serilog.

Работа с типами Debug и Trace

Существует два типа, которые можно использовать для добавления простой регистрации в ваш код: `Debug` и `Trace`.

Прежде чем мы углубимся в их изучение, коротко рассмотрим каждый из них:

- класс `Debug` используется для добавления события, которое записывается только во время разработки;
- класс `Trace` служит для добавления события, которое записывается как во время разработки, так и во время выполнения.

Вы видели, как используется тип `Console` и его метод `WriteLine` для предоставления вывода в окно консоли. Но существуют и типы `Debug` и `Trace`, более гибкие в отношении того, куда пишется вывод.

Классы `Debug` и `Trace` могут вести записи в любой *прослушиватель трассировки* (trace listener). Это тип, который можно настроить для записи результатов в любом необходимом для вас месте, когда вызывается метод `WriteLine`. Платформа .NET предоставляет несколько прослушивателей трассировки, и вы даже можете создать собственный, унаследовав тип `TraceListener`.

Запись в прослушиватель трассировки по умолчанию

Один из прослушивателей трассировки, класс `DefaultTraceListener`, настраивается автоматически и записывает на панель `DEBUG CONSOLE` (Консоль отладки) в среде разработки Visual Studio Code или в окне `Debug` Visual Studio. В коде вы можете вручную добавить другие прослушиватели трассировки.

Рассмотрим пример.

1. Откройте редактор кода и создайте консольное приложение `Instrumenting` в рабочей области/решении `Chapter04`.
2. В Visual Studio Code выберите `Instrumenting` в качестве активного проекта OmniSharp. Когда вы увидите всплывающее предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку `Yes` (Да), чтобы добавить их.

3. В файл `Program.cs` импортируйте пространство имен `System.Diagnostics`.
4. Создайте сообщение с помощью классов `Debug` и `Trace`.

```
Debug.WriteLine("Debug says, I am watching!");
Trace.WriteLine("Trace says, I am watching!");
```

5. В Visual Studio выберите команду меню View ▸ Output (Вид ▸ Отладка) и убедитесь, что выбран параметр Show output from: Debug (Показать вывод из: Отладка).
6. Начните отладку консольного приложения `Instrumenting` и обратите внимание, что на панели DEBUG CONSOLE (Консоль отладки) в Visual Studio Code или окне Output (Вывод) в Visual Studio 2022 отображаются два сообщения, смешанные с другой информацией отладки наподобие загруженных библиотек DLL (рис. 4.15 и 4.16).

Рис. 4.15. На панели DEBUG CONSOLE (Консоль отладки) в Visual Studio Code отображаются два сообщения синего цвета

Рис. 4.16. В окне Output (Вывод) в Visual Studio 2022 отображаются результаты отладки, включая два сообщения

Настройка прослушивателей трассировки

Теперь мы настроим другой прослушиватель трассировки, который будет производить запись в текстовый файл.

1. Перед вызовами `Debug.WriteLine` и `Trace.WriteLine` добавьте оператор для создания нового текстового файла на рабочем столе. Передайте его в новый прослушиватель трассировки, который знает, как производить запись в текстовый файл. Затем включите автоматическую очистку буфера, как показано ниже:

```
// запись в текстовый файл, расположенный в папке проекта
Trace.Listeners.Add(new TextWriterTraceListener(
  File.CreateText(Path.Combine(Environment.GetFolderPath(
    Environment.SpecialFolder.DesktopDirectory), "log.txt"))));

// модуль записи текста буферизируется, поэтому данная опция
// вызывает функцию Flush() для всех прослушивателей после записи
Trace.AutoFlush = true;

Debug.WriteLine("Debug says, I am watching!");
Trace.WriteLine("Trace says, I am watching!");
```

Любой тип, представляющий файл, обычно реализует буфер для повышения производительности. Вместо немедленной записи в файл данные записываются в буфер и только после заполнения буфера записываются в один фрагмент в файл. Такое поведение может сбить с толку при отладке, поскольку мы не сразу видим результаты! Включение AutoFlush означает, что метод Flush вызывается автоматически после каждой записи.

2. В Visual Studio Code запустите конфигурацию релиза консольного приложения, введя следующую команду на панели TERMINAL (Терминал) для проекта `Instrumenting`, и обратите внимание, что ничего не произошло:

```
dotnet run --configuration Release
```

3. В Visual Studio 2022 на стандартной панели инструментов в раскрывающемся списке Solution Configurations (Конфигурации решения) выберите Release (Релиз) (рис. 4.17).

Рис. 4.17. Выбор конфигурации Release в Visual Studio

4. В Visual Studio 2022 запустите конфигурацию релиза консольного приложения, выбрав команду меню Debug ▸ Start Without Debugging (Отладка ▸ Начать без отладки).

5. На рабочем столе откройте файл `log.txt` и обратите внимание, что он содержит сообщение `"Trace says, I am watching!"`.
6. В Visual Studio Code запустите конфигурацию отладки консольного приложения, введя следующую команду на панели TERMINAL (Терминал) для проекта `Instrumenting`:

```
dotnet run --configuration Debug
```

7. В Visual Studio на стандартной панели инструментов в раскрывающемся списке Solution Configurations (Конфигурации решения) выберите Debug (Отладка), а затем запустите консольное приложение, выбрав команду меню Debug ▸ Start Debugging (Отладка ▸ Начать отладку).
8. На рабочем столе откройте файл `log.txt` и обратите внимание, что он содержит оба сообщения: `"Debug says, I am watching!"` и `"Trace says, I am watching!"`.

При работе с режимом конфигурации Debug активны классы и Debug, и Trace, и их результаты будут отображаться на панели DEBUG CONSOLE (Консоль отладки). При работе с режимом конфигурации Release отображается только вывод Trace. Поэтому вы можете свободно использовать вызовы Debug.WriteLine по всему коду, зная, что они будут автоматически удалены при создании релизной версии вашего приложения и, следовательно, не повлияют на производительность.

Переключение уровней трассировки

Вызовы `Trace.WriteLine` остаются в вашем коде даже после выпуска. Таким образом, было бы хорошо контролировать их вывод. Мы можем это сделать с помощью *переключателя трассировки*.

Значение переключателя трассировки можно установить с помощью числа или слова. Например, число `3` можно заменить словом `Info`, как показано в табл. 4.1.

Таблица 4.1. Переключение уровней трассировки

Число	Слово	Описание
0	`Off`	Ничего не выводит
1	`Error`	Выводит только ошибки
2	`Warning`	Выводит ошибки и предупреждения
3	`Info`	Выводит ошибки, предупреждения и информацию
4	`Verbose`	Выводит все уровни

Рассмотрим использование переключателей трассировки. Для начала нам потребуется добавить несколько пакетов NuGet, чтобы разрешить загрузку настроек конфигурации из JSON-файла настроек `appsettings`.

Добавление пакетов в проект в Visual Studio Code

Visual Studio Code не имеет механизма для добавления пакетов NuGet в проект, поэтому мы будем использовать инструменты командной строки.

1. Перейдите к панели TERMINAL (Терминал) для проекта `Instrumenting`.
2. Введите команду:

```
dotnet add package Microsoft.Extensions.Configuration
```

3. Введите команду:

```
dotnet add package Microsoft.Extensions.Configuration.Binder
```

4. Введите команду:

```
dotnet add package Microsoft.Extensions.Configuration.Json
```

5. Введите команду:

```
dotnet add package Microsoft.Extensions.Configuration.FileExtensions
```

Команда dotnet add package добавляет в файл проекта ссылку на пакет NuGet. Он будет скачан в процессе сборки. Команда dotnet add reference добавляет в файл проекта ссылку между проектами. Ссылочный проект будет скомпилирован при необходимости в процессе сборки.

Добавление пакетов в проект в Visual Studio 2022

Visual Studio содержит графический пользовательский интерфейс для добавления пакетов.

1. На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте Instrumenting и выберите Manage NuGet Packages (Управление пакетами NuGet).
2. Выберите вкладку Browse (Обзор).
3. В поле поиска введите `Microsoft.Extensions.Configuration`.

4. Выберите каждый из этих пакетов NuGet и нажмите кнопку Install (Установить) (рис. 4.18):
   1) `Microsoft.Extensions.Configuration`;
   2) `Microsoft.Extensions.Configuration.Binder`;
   3) `Microsoft.Extensions.Configuration.Json`;
   4) `Microsoft.Extensions.Configuration.FileExtensions`.

Рис. 4.18. Установка пакетов NuGet с помощью Visual Studio 2022

Существуют также пакеты для загрузки конфигурации из XML-, INI-файлов, переменных среды и командной строки. Используйте в ваших проектах наиболее подходящий метод настройки конфигурации.

Проверка пакетов проектов

1. Откройте файл `Instrumenting.csproj` (дважды щелкните на проекте Instrumenting на панели Solution Explorer (Обозреватель решений) в Visual Studio) и создайте раздел `<ItemGroup>` с дополнительными пакетами NuGet:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.Extensions.Configuration"
```

```
      Version="6.0.0" />
    <PackageReference
      Include="Microsoft.Extensions.Configuration.Binder"
      Version="6.0.0" />
    <PackageReference
      Include="Microsoft.Extensions.Configuration.FileExtensions"
      Version="6.0.0" />
    <PackageReference
      Include="Microsoft.Extensions.Configuration.Json"
      Version="6.0.0" />
  </ItemGroup>

</Project>
```

2. Добавьте файл `appsettings.json` в папку Instrumenting.
3. Измените содержимое файла `appsettings.json`, чтобы определить параметр `PacktSwitch` со значением `Level`:

```
{
  "PacktSwitch": {
    "Level": "Info"
  }
}
```

4. В Visual Studio 2022 на панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на файле `appsettings.json`, выберите Properties (Свойства), а затем в окне Properties (Свойства) замените Copy to Output Directory (Копировать в выходной каталог) на Copy if newer (Копировать, если новее). Это необходимо выполнить, так как в отличие от редактора Visual Studio Code, запускающего консольное приложение в папке проекта, Visual Studio запускает консольное приложение в `Instrumenting\bin\Debug\net6.0` или `Instrumenting\bin\Release\net6.0`.
5. В начале файла `Program.cs` импортируйте пространство имен `Microsoft.Extensions.Configuration`.
6. Добавьте несколько операторов в конец файла `Program.cs`, чтобы создать строитель конфигурации, который ищет в текущей папке файл `appsettings.json`. Скомпонуйте конфигурацию, создайте переключатель трассировки, установите уровень, привязав его к конфигурации, и затем выведите четыре уровня переключения трассировки:

```
ConfigurationBuilder builder = new();

builder.SetBasePath(Directory.GetCurrentDirectory())
  .AddJsonFile("appsettings.json",
    optional: true, reloadOnChange: true);

IConfigurationRoot configuration = builder.Build();
```

```
TraceSwitch ts = new(
  displayName: "PacktSwitch",
  description: "This switch is set via a JSON config.");

configuration.GetSection("PacktSwitch").Bind(ts);

Trace.WriteLineIf(ts.TraceError, "Trace error");
Trace.WriteLineIf(ts.TraceWarning, "Trace warning");
Trace.WriteLineIf(ts.TraceInfo, "Trace information");
Trace.WriteLineIf(ts.TraceVerbose, "Trace verbose");
```

7. Установите точку останова на операторе `Bind`.
8. Начните отладку консольного приложения `Instrumenting`. На панели VARIABLES (Переменные) или Locals (Локальные переменные) разверните наблюдение за переменной `ts` и обратите внимание, что ее `Level` (Уровень) установлен в значение `Off` (Выкл.), а значения `TraceError`, `TraceWarning` и т. д. равны `false` (рис. 4.19).

Рис. 4.19. Просмотр свойств переменной переключателя трассировки в Visual Studio 2022

9. Перейдите к вызову метода `Bind`, нажав кнопку Step Into (Шагнуть внутрь) либо Step Over (Перешагнуть) или клавиши F11 либо F10, и обратите внимание, что переменная `ts` следит за обновлениями вплоть до уровня `Info`.
10. Перейдите к четырем вызовам метода `Trace.WriteLineIf` и обратите внимание, что все уровни вплоть до `Info` записываются на панели DEBUG CONSOLE (Консоль отладки) или в окне Output — Debug (Вывод — отладка), но не уровень `Verbose` (рис. 4.20).

Рис. 4.20. Различные уровни трассировки, отображаемые на панели DEBUG CONSOLE (Консоль отладки) в Visual Studio Code

11. Остановите отладку.
12. Измените содержимое файла `appsettings.json`, чтобы установить уровень `2` (то есть предупреждение), как показано в следующем файле JSON:

```
{
  "PacktSwitch": {
    "Level": "2"
  }
}
```

13. Сохраните изменения.
14. В Visual Studio Code запустите консольное приложение, введя следующую команду на панели TERMINAL (Терминал) для проекта `Instrumenting`:

```
dotnet run --configuration Release
```

15. В Visual Studio на стандартной панели инструментов в раскрывающемся списке Solution Configurations (Конфигурации решения) выберите Release (Релиз), а затем запустите консольное приложение, выбрав команду меню Debug ▸ Start Without Debugging (Отладка ▸ Начать без отладки).
16. Откройте файл `log.txt` и обратите внимание, что на сей раз из четырех потенциальных уровней трассировки выводится только трассировка с уровнями «ошибка» и «предупреждение», как показано в следующем текстовом файле:

```
Trace says, I am watching!
Trace error
Trace warning
```

Если аргумент не передан, то уровень переключателя трассировки по умолчанию считается `Off` (`0`), поэтому не выводится ни один из уровней переключателя.

Модульное тестирование

Исправление ошибок в коде стоит дорого. Чем раньше будет обнаружена ошибка в процессе разработки, тем дешевле будет ее исправить.

Модульное тестирование прекрасно подходит для поиска ошибок на ранних стадиях разработки. Некоторые разработчики даже следуют принципу, согласно которому программисты должны создавать модульные тесты, прежде чем писать код, и это называется *разработкой на основе тестирования* (Test-Driven Development, TDD).

Компания Microsoft имеет собственную систему модульного тестирования, известную как *MS Test*. Существует также фреймворк *Nunit*. Однако мы будем использовать сторонний фреймворк *xUnit.NET*. Он создан той же командой, которая написала Nunit, но в нем исправлены ошибки, которые, по мнению команды, были допущены ранее. Фреймворк xUnit более расширяемый и имеет улучшенную поддержку.

Виды тестирования

Модульное тестирование — лишь один из многих типов тестирования, описанных в табл. 4.2.

Таблица 4.2. Виды тестирования

Виды тестирования	Описание
Unit (модульное)	Проверяется наименьшая единица кода, обычно метод или функция. С помощью модульного тестирования выполняется тестирование отдельного изолированного модуля или части кода. В каждом модуле необходимо выполнить несколько тестов: одни с типичными входными и ожидаемыми выходными данными, другие с экстремальными входными значениями для проверки границ, а третьи с заведомо неправильными входными данными для проверки обработки исключений
Integration (интеграционное)	Проверяется, работают ли мелкие и более крупные блоки как единое целое. Иногда может быть включена интеграция с внешними компонентами, для которых нет исходного кода
System (системное)	Проверяется вся система, в которой будет работать ваше программное обеспечение
Performance (тестирование производительности)	Проверяется производительность вашего ПО; например, код должен вернуть пользователю веб-страницу с данными менее чем за 20 миллисекунд
Load (нагрузочное)	Проверяется количество запросов, которое ваше программное обеспечение может обрабатывать одновременно, сохраняя требуемую производительность, например одновременное нахождение 10 000 пользователей на сайте
User Acceptance (приемочное пользовательское)	Проверяется, могут ли пользователи успешно выполнять свои задачи, используя ваше ПО

Создание библиотеки классов, требующей тестирования

Сначала мы создадим функцию, которую необходимо тестировать. Мы создадим ее в проекте библиотеки классов. Библиотека классов — это пакет кода, который может распространяться и использоваться другими приложениями .NET.

1. Откройте редактор кода и создайте в рабочей области/решении `Chapter04` проект Class Library под названием `CalculatorLib`.
2. Переименуйте файл `Class1.cs` в `Calculator.cs`.
3. Измените файл, чтобы определить класс `Calculator` (с преднамеренной ошибкой!), как показано ниже:

```
namespace Packt
{
  public class Calculator
  {
    public double Add(double a, double b)
    {
      return a * b;
    }
  }
}
```

4. Скомпилируйте проект библиотеки классов:
   1) в Visual Studio 2022 выберите команду меню Build ▸ Build CalculatorLib (Сборка ▸ Сборка CalculatorLib);
   2) в Visual Studio Code на панели TERMINAL (Терминал) введите команду `dotnet build`.
5. Откройте редактор кода и в рабочей области/решении `Chapter04` создайте проект xUnit Test Project [C#] под названием `CalculatorLibUnitTests`. Шаблон `dotnet new` называется `xunit`.
6. Если вы используете Visual Studio, то на панели Solution Explorer (Обозреватель решений) выберите проект CalculatorLibUnitTests, выберите команду меню Project ▸ Add Project Reference... (Проект ▸ Добавить ссылку на проект...), установите флажок, чтобы выбрать проект CalculatorLib, и нажмите OK.
7. Если вы работаете в Visual Studio Code, используйте команду `dotnet add reference` или выберите файл `CalculatorLibUnitTests.csproj` и измените конфигурацию, добавив группу элементов со ссылкой на проект `CalculatorLib`:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
```

```
    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.10.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.3">
      <IncludeAssets>runtime; build; native; contentfiles;
        analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="coverlet.collector" Version="3.0.2">
      <IncludeAssets>runtime; build; native; contentfiles;
        analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>

  <ItemGroup>
    <ProjectReference
      Include="..\CalculatorLib\CalculatorLib.csproj" />
  </ItemGroup>
</Project>
```

8. Соберите проект `CalculatorLibUnitTests`.

Разработка модульных тестов

Правильно разработанный модульный тест будет состоять из трех частей (трех A):

- *размещение* (arrange) — объявляет и создает экземпляры переменных для ввода и вывода;
- *действие* (act) — выполняет тестируемый модуль. В нашем случае это означает вызов метода, который необходимо протестировать;
- *утверждение* (assert) — создает одно или несколько утверждений о выводе. Если оно не соответствует действительности, то тест считается неудачным. Например, при сложении 2 и 2 мы ожидаем, что результат будет 4[1].

Теперь мы напишем модульные тесты для класса `Calculator`.

1. Переименуйте класс `UnitTest1.cs` в `CalculatorUnitTests.cs` и откройте его.
2. В Visual Studio Code переименуйте класс в `CalculatorUnitTests`. (Visual Studio предлагает переименовать класс при переименовании файла.)
3. Импортируйте пространство имен `Packt`.

[1] Этот принцип называется AAA от англ. Arrange-Act-Assert.

4. Измените класс `CalculatorUnitTests`, чтобы создать два метода тестирования для сложения 2 и 2 и 2 и 3:

```
using Packt;
using Xunit;

namespace CalculatorLibUnitTests
{
  public class CalculatorUnitTests
  {
    [Fact]
    public void TestAdding2And2()
    {
      // размещение
      double a = 2;
      double b = 2;
      double expected = 4;
      Calculator calc = new();

      // действие
      double actual = calc.Add(a, b);

      // утверждение
      Assert.Equal(expected, actual);
    }

    [Fact]
    public void TestAdding2And3()
    {
      // размещение
      double a = 2;
      double b = 3;
      double expected = 5;
      Calculator calc = new();

      // действие
      double actual = calc.Add(a, b);

      // утверждение
      Assert.Equal(expected, actual);
    }
  }
}
```

Выполнение модульных тестов в Visual Studio Code

Теперь мы готовы запустить модульные тесты и посмотреть результаты.

1. На панели `TERMINAL` (Терминал) проекта `CalculatorLibUnitTest` введите следующую команду:

```
dotnet test
```

2. Обратите внимание: результаты показывают, что два теста выполнены, один пройден и один не пройден (рис. 4.21).

Рис. 4.21. Результаты модульного теста на панели TERMINAL (Терминал) в Visual Studio Code

Выполнение модульных тестов в Visual Studio

Теперь мы готовы запустить модульные тесты и проанализировать результаты.

1. Выберите команду меню Test ▸ Run All Tests (Тест ▸ Запустить все тесты).
2. Обратите внимание, что в окне Test Explorer (Обозреватель тестов) результаты показывают, что два теста выполнены, один пройден и один не пройден (рис. 4.22).

Рис. 4.22. Результаты модульного тестирования, показанные в окне Test Explorer (Обозреватель тестов) Visual Studio 2022

Исправление ошибок

Теперь мы можем исправить ошибки.

1. Исправьте ошибку в методе `Add`.
2. Запустите модульные тесты еще раз, чтобы убедиться, что ошибка исправлена и оба теста пройдены.

Генерация и перехват исключений в функциях

В главе 3 вы изучили исключения и то, как использовать оператор `try-catch` для их обработки. Но вам необходимо перехватывать и обрабатывать исключение только в том случае, если у вас достаточно информации для решения проблемы. Если вы этого не сделаете, то вам следует позволить исключению пройти через стек вызовов на более высокий уровень.

Ошибки использования и ошибки выполнения

Ошибки использования возникают, когда разработчик неправильно использует функцию, обычно передавая в качестве параметров недопустимые значения. Их можно избежать, если использовать допустимые значения. Когда разработчики только начинают изучать C# и .NET, они предполагают, что исключения всегда можно избежать, поскольку все ошибки — это ошибки использования. Все ошибки использования должны быть исправлены до выпуска в рабочую среду.

Ошибки выполнения — это когда во время выполнения происходят ситуации, которые нельзя исправить, написав «лучший» код. Ошибки выполнения можно разделить на *программные* и *системные*. Если вы пытаетесь получить доступ к сетевому ресурсу, но сеть не работает, то вам необходимо обработать эту системную ошибку, зарегистрировав исключение и, возможно, через некоторое время повторив попытку. Но некоторые системные ошибки, такие как нехватка памяти, обработать невозможно. Если вы попытаетесь открыть несуществующий файл, то, вероятно, сможете перехватить эту ошибку и обработать ее программно, создав новый файл. Программные ошибки можно исправить программно, написав умный код. Системные ошибки, как правило, не могут быть исправлены программно.

Часто выбрасываемые исключения в функциях

Очень редко возникает необходимость определять новые типы исключений для определения ошибок использования. Платформа .NET уже определяет многие из них.

При определении собственных функций с параметрами ваш код должен проверять значения параметров и генерировать исключения, если они имеют значения, препятствующие корректной работе вашей функции.

Например, если параметр не должен иметь значение `null`, то сгенерируйте исключение `ArgumentNullException`. В случае других проблем сгенерируйте исключение `ArgumentException`, `NotSupportedException` или `InvalidOperationException`. Для любого исключения отобразите сообщение, описывающее проблему, адресованное тем, кто должен будет его увидеть (обычно это разработчики библиотек классов и функций или конечные пользователи, если они находятся на самом высоком уровне приложения с графическим интерфейсом):

```
static void Withdraw(string accountName, decimal amount)
{
  if (accountName is null)
  {
    throw new ArgumentNullException(paramName: nameof(accountName));
  }

  if (amount < 0)
  {
    throw new ArgumentException(
      message: $"{nameof(amount)} cannot be less than zero.");
  }

  // параметры процесса
}
```

Если функция не может успешно выполнить свою задачу, вы должны считать, что произошел сбой функции. Вам следует сообщить об этом, создав исключение.

Вам никогда не придется добавлять оператор `try-catch`, чтобы перехватывать эти ошибки использования. В первую очередь необходимо, чтобы приложение корректно работало. Эти исключения должны заставить разработчика, вызывающего функцию, исправить свой код, чтобы предотвратить проблему. Они должны быть исправлены до развертывания в производственной среде. Это не означает, что ваш код не должен генерировать исключения типа ошибки использования. Вам необходимо заставить других разработчиков правильно вызывать ваши функции!

Стек вызовов

Точкой входа для консольного приложения .NET является метод `Main` класса `Program`, независимо от того, определили ли вы этот класс и метод явно, или они были программно созданы функцией верхнего уровня.

Метод `Main` будет вызывать другие методы, которые, в свою очередь, вызывают другие методы и т. д., и они могут находиться в текущем проекте или в проектах, на которые ссылаются, и пакетах NuGet (рис. 4.23).

Рис. 4.23. Цепочка вызовов методов, создающих стек вызовов

Создадим аналогичную цепочку методов и исследуем, где мы могли бы перехватывать и обрабатывать исключения.

1. Откройте редактор кода и создайте библиотеку классов `CallStackException-HandlingLib` в рабочей области/решении `Chapter04`.
2. Переименуйте файл `Class1.cs` в `Calculator.cs`.
3. Откройте файл `Calculator.cs` и измените его содержимое:

```
using static System.Console;

namespace Packt;

public class Calculator
{
  public static void Gamma() // public – для вызова вне фукнции
  {
    WriteLine("In Gamma");
    Delta();
  }

  private static void Delta() // private – для вызова только внутри функции
  {
    WriteLine("In Delta");
    File.OpenText("bad file path");
  }
}
```

4. Откройте редактор кода и создайте консольное приложение `CallStackExceptionHandling` в рабочей области/решении `Chapter04`.

5. В Visual Studio Code выберите `CallStackExceptionHandling` в качестве активного проекта OmniSharp. Когда вы увидите всплывающее предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку Yes (Да), чтобы добавить их.
6. В проекте `CallStackExceptionHandling` добавьте ссылку на проект `CallStackExceptionHandlingLib`.
7. В файле `Program.cs` добавьте операторы для определения двух методов и их цепочек вызовов, а также методы в библиотеке классов:

```
using Packt;

using static System.Console;

WriteLine("In Main");
Alpha();

static void Alpha()
{
  WriteLine("In Alpha");
  Beta();
}

static void Beta()
{
  WriteLine("In Beta");
  Calculator.Gamma();
}
```

8. Запустите консольное приложение и проанализируйте результат:

```
In Main
In Alpha
In Beta
In Gamma
In Delta
Unhandled exception. System.IO.FileNotFoundException: Could not find file
'C:\Code\Chapter04\CallStackExceptionHandling\bin\Debug\net6.0\bad file
path'.
   at Microsoft.Win32.SafeHandles.SafeFileHandle.CreateFile(...
   at Microsoft.Win32.SafeHandles.SafeFileHandle.Open(...
   at System.IO.Strategies.OSFileStreamStrategy..ctor(...
   at System.IO.Strategies.FileStreamHelpers.ChooseStrategyCore(...
   at System.IO.Strategies.FileStreamHelpers.ChooseStrategy(...
   at System.IO.StreamReader.ValidateArgsAndOpenPath(...
   at System.IO.File.OpenText(String path) in ...
   at Packt.Calculator.Delta() in C:\Code\Chapter04\
CallStackExceptionHandlingLib\Calculator.cs:line 16
   at Packt.Calculator.Gamma() in C:\Code\Chapter04\
```

```
CallStackExceptionHandlingLib\Calculator.cs:line 10
   at <Program>$.<<Main>$>g__Beta|0_1() in C:\Code\Chapter04\
CallStackExceptionHandling\Program.cs:line 16
   at <Program>$.<<Main>$>g__Alpha|0_0() in C:\Code\Chapter04\
CallStackExceptionHandling\Program.cs:line 10
   at <Program>$.<Main>$(String[] args) in C:\Code\Chapter04\
CallStackExceptionHandling\Program.cs:line 5
```

Обратите внимание на моменты, указанные ниже.

- Стек вызовов перевернут. Начиная с конца стека, вы обнаружите, что:
  - первый вызов — это функция точки входа `Main` в автоматически сгенерированном классе `Program`. Здесь аргументы передаются в виде массива строк;
  - второй вызов — функция `Alpha`;
  - третий вызов — функция `Beta`;
  - четвертый вызов — функция `Gamma`;
  - пятый вызов — функция `Delta`. Она пытается открыть файл, передавая неверный путь к нему, что приводит к возникновению исключения. Любая функция с оператором `try-catch` может перехватить это исключение. Если это не так, то оператор автоматически передается вверх по стеку вызовов, пока не достигнет вершины, где .NET выведет исключение (и сведения об этом стеке вызовов).

Где перехватывать исключения

Разработчики могут решить, хотят ли они перехватывать исключение рядом с точкой сбоя или централизованно выше по стеку вызовов. Это позволяет упростить и стандартизировать ваш код. Возможно, вы знаете, что вызов исключения может вызвать один или несколько типов исключений, но вам не нужно обрабатывать ни одно из них в текущей точке стека вызовов.

Повторное создание исключений

Иногда вам необходимо перехватить исключение, зарегистрировать его, а затем повторно сгенерировать. Существует три способа повторного создания исключения внутри блока `catch`.

1. Чтобы сгенерировать уже перехваченное исключение с исходным стеком вызовов, вызовите `throw`.
2. Чтобы сгенерировать уже перехваченное исключение, как если бы оно было сгенерировано на текущем уровне стека вызовов, вызовите `throw` с перехваченным

исключением, например `throw ex`. Как правило, это плохая практика, поскольку теряется некоторая потенциально полезная информацию для отладки.

3. Перехваченное исключение можно обернуть в другое, включающее дополнительную информацию, которая может помочь вызывающей стороне понять проблему. Для этого сгенерируйте новое исключение и передайте перехваченное исключение в качестве параметра `innerException`.

Если бы при вызове функции `Gamma` могла возникнуть ошибка, то мы могли бы перехватить исключение, а затем выполнить один из трех методов повторной генерации исключения, как показано ниже:

```
try
{
  Gamma();
}
catch (IOException ex)
{
  LogException(ex);

  // сгенерировать перехваченное исключение, как если бы оно произошло здесь
  // это приведет к потере исходного стека вызовов
  throw ex;

  // повторно создать перехваченное исключение и сохранить исходный стек вызовов
  throw;

  // сгенерировать новое исключение с вложенным в него перехваченным исключением
  throw new InvalidOperationException(
    message: "Calculation had invalid values. See inner exception for why.",
    innerException: ex);
}
```

Рассмотрим пример.

1. В проекте `CallStackExceptionHandling` в файле `Program.cs` в функции `Beta` вокруг вызова функции `Gamma` добавьте оператор `try-catch`:

```
static void Beta()
{
  WriteLine("In Beta");

  try
  {
    Calculator.Gamma();
  }
  catch (Exception ex)
  {
    WriteLine($"Caught this: {ex.Message}");
```

```
    throw ex;
  }
}
```

2. Обратите внимание на зеленую волнистую линию под оператором `ex`, означающую, что вы можете потерять информацию о стеке вызовов.
3. Запустите код и обратите внимание, что в результате отсутствуют некоторые сведения о стеке вызовов.

```
Caught this: Could not find file 'C:\Code\Chapter04\
CallStackExceptionHandling\bin\Debug\net6.0\bad file path'.
Unhandled exception. System.IO.FileNotFoundException: Could not find file
'C:\Code\Chapter04\CallStackExceptionHandling\bin\Debug\net6.0\bad file
path'.
File name: 'C:\Code\Chapter04\CallStackExceptionHandling\bin\Debug\net6.0\
bad file path'
   at <Program>$.<<Main>$>g__Beta|0_1() in C:\Code\Chapter04\
CallStackExceptionHandling\Program.cs:line 25
   at <Program>$.<<Main>$>g__Alpha|0_0() in C:\Code\Chapter04\
CallStackExceptionHandling\Program.cs:line 11
   at <Program>$.<Main>$(String[] args) in C:\Code\Chapter04\
CallStackExceptionHandling\Program.cs:line 6
```

4. Удалите `ex`, заменив оператор `throw ex;` на `throw;` при переброске данных.
5. Запустите код и обратите внимание, что выходные данные включают все сведения о стеке вызовов.

Реализация шаблона tester-doer

Использование *шаблона tester-doer* поможет избежать возникновения некоторых исключений (но не устранить их полностью). В нем используются пары функций: одна — для выполнения теста, вторая — для выполнения действия, которое завершится ошибкой, если тест не будет пройден.

Платформа .NET сама реализует этот шаблон. Например, прежде, чем добавлять элемент в коллекцию путем вызова метода `Add`, вы можете проверить, доступен ли он только для чтения, что приведет к сбою `Add` при добавлении и, следовательно, к возникновению исключения.

Например, прежде, чем снимать деньги с банковского счета, вы можете проверить, не перерасходован ли счет.

```
if (!bankAccount.IsOverdrawn())
{
  bankAccount.Withdraw(amount);
}
```

Недостатки при использовании шаблона tester-doer

Использование шаблона tester-doer может увеличить нагрузку на производительность, поэтому вы также можете реализовать *шаблон try*, который, объединяет тестирование и выполнение в одну функцию, как это видно на примере с `TryParse`.

Еще одна проблема возникает, когда вы используете несколько потоков. В данном случае один поток может вызвать тестовую функцию, и она вернет успешный результат. Затем выполняется другой поток, который меняет состояние. Затем исходный поток продолжает выполнение, предполагая, что все выполнено успешно, но это не так. Это называется состоянием гонки, о котором вы узнаете в главе 12.

Если вы реализуете собственную функцию шаблона `try` и она не работает, то не забудьте установить для параметра `out` значение по умолчанию соответствующего типа, а затем вернуть значение `false`:

```
static bool TryParse(string? input, out Person value)
{
  if (someFailure)
  {
    value = default(Person);
    return false;
  }

  // успешно проанализировал строку в Person
  value = new Person() { ... };
  return true;
}
```

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 4.1. Проверочные вопросы

Ответьте на следующие вопросы. Если вы не знаете, как ответить, попробуйте поискать ответы в Google, но всегда помните, что все ответы находятся в приложении.

1. Что в языке C# означает ключевое слово `void`?
2. В чем разница между императивным и функциональным стилями программирования?
3. В программе Visual Studio Code или Visual Studio какова разница между нажатием клавиш F5, Ctrl или Cmd+F5, Shift+F5 и Ctrl или Cmd+Shift+F5?

4. Куда записывает выходные данные метод `Trace.WriteLine`?
5. Каковы пять уровней трассировки?
6. Чем различаются классы `Debug` и `Trace`?
7. Как называются три A модульного теста?
8. При написании модульного теста с помощью xUnit каким атрибутом вы должны дополнять методы тестирования?
9. Какая команда `dotnet` выполняет тесты xUnit?
10. Какой оператор следует использовать для повторного создания перехваченного исключения `ex` без потери трассировки стека?

Упражнение 4.2. Функции, отладка и модульное тестирование

Простые множители — это комбинация наименьших простых чисел, которые при умножении в результате дают исходное число. Рассмотрим пример:

- простой множитель 4: 2×2;
- простой множитель 7: 7;
- простой множитель 30: $5 \times 3 \times 2$;
- простой множитель 40: $5 \times 2 \times 2 \times 2$;
- простой множитель 50: $5 \times 5 \times 2$.

Создайте рабочую область/решение `PrimeFactors`, содержащую три проекта; библиотеку классов с методом `PrimeFactors`, которая при передаче переменной `int` в качестве параметра возвращает строку, показывающую ее простые множители; проект модульных тестов; и консольное приложение для его использования.

Для простоты можно предположить, что наибольшее введенное число будет равно `1000`.

Используйте инструменты отладки и напишите модульные тесты с целью убедиться, что ваша функция работает правильно с несколькими входными параметрами и возвращает правильные выходные параметры.

Упражнение 4.3. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-4---writing-debugging-and-testing-functions, чтобы получить дополнительную информацию по темам, приведенным в этой главе.

Резюме

В этой главе вы изучили, как в среде разработки Visual Studio Code писать многоразовые функции с входными параметрами и возвращаемыми значениями как в императивном, так и в функциональном стиле. Кроме того, вы узнали, как использовать средства отладки и диагностики кода в Visual Studio и Visual Studio Code для исправления любых ошибок в нем. Наконец, вы научились генерировать и перехватывать исключения в функциях и разбираться в стеке вызовов.

В следующей главе вы узнаете, как создавать пользовательские типы с помощью технологий объектно-ориентированного программирования.

5 Создание пользовательских типов с помощью объектно-ориентированного программирования

Данная глава посвящена созданию пользовательских типов с помощью принципов *объектно-ориентированного программирования (ООП)*. Вы узнаете о различных категориях элементов, которые может иметь тип, в том числе о полях для хранения данных и методах для выполнения действий. Вы будете применять концепции ООП, такие как агрегирование и инкапсуляция. Вы изучите языковые функции, такие как поддержка синтаксиса кортежей и переменные `out`, литералы для значений по умолчанию и автоматически определяемые имена кортежей.

В этой главе:

- коротко об ООП;
- сборка библиотек классов;
- хранение данных с помощью полей;
- запись и вызов методов;
- управление доступом с помощью свойств и индексаторов;
- сопоставление с образцом с помощью объектов;
- работа с записями.

Коротко об объектно-ориентированном программировании

Объект в реальном мире — это предмет, например автомобиль или человек. Объект в программировании часто представляет нечто в реальном мире, например товар или банковский счет, но может быть и чем-то более абстрактным.

В языке C# используются классы `class` (обычно) или структуры `struct` (редко) для определения каждого типа объекта. О разнице между классами и структурами вы узнаете в главе 6. Можно представить тип как шаблон объекта.

Ниже кратко описаны концепции объектно-ориентированного программирования.

- *Инкапсуляция* — комбинация данных и действий, связанных с объектом. К примеру, тип `BankAccount` может иметь такие данные, как `Balance` и `AccountName`, а также действия, такие как `Deposit` и `Withdraw`. При инкапсуляции часто возникает необходимость управлять тем, кто и что может получить доступ к этим действиям и данным, например ограничивать доступ к внутреннему состоянию объекта или его изменению извне.
- *Композиция* — то, из чего состоит объект. К примеру, объект `Car` (Автомобиль) состоит из разных частей, таких как четыре объекта `Wheel` (Колесо), несколько объектов `Seat` (Сиденье) и один объект `Engine` (Двигатель).
- *Агрегирование* касается всего, что может быть объединено с объектом. Например, объект `Person` не является частью объекта `Car`, но человек может сесть на сиденье водителя (`Seat`), а затем стать им (`Driver`). Два отдельных объекта объединены, чтобы сформировать новый компонент.
- *Наследование* — многократное использование кода с помощью *подкласса*, производного от *базового класса* или *суперкласса*. Все функциональные возможности базового класса наследуются и становятся доступными в *производном* классе. Например, базовый или суперкласс `Exception` имеет несколько членов, которые имеют одинаковую реализацию во всех исключениях. А подкласс, он же производный класс `SqlException`, наследует эти члены и имеет дополнительные, имеющие отношение только к тем случаям, когда возникает исключение базы данных SQL — например, свойство, содержащее информацию о подключении к базе данных.
- *Абстракция* — передача основной идеи объекта и игнорирование его деталей или особенностей. Язык C# имеет ключевое слово `abstract`, которое формализует концепцию. Если класс не явно абстрактный, то его можно описать как конкретный. Базовые классы часто абстрактны, например, суперкласс `Stream` — абстрактный, а его подклассы, такие как `FileStream` и `MemoryStream`, — конкретные. Создавать объекты можно только с помощью конкретных классов; абстрактные же могут использоваться лишь в качестве основы для других классов, поскольку им не хватает некоторой реализации. Абстракцию сложно сбалансировать. Если вы сделаете класс слишком абстрактным, то больше классов смогут наследовать его, но при этом уменьшится количество возможностей для совместного использования.
- *Полиморфизм* заключается в переопределении производным классом унаследованных методов для реализации собственного поведения.

Разработка библиотек классов

Сборки библиотек классов группируют типы в легко развертываемые модули (DLL-файлы). Не считая раздела, где вы изучали модульное тестирование, до сих пор вы создавали только консольные приложения или блокноты .NET Interactive, содержащие ваш код. Но, чтобы он стал доступен для нескольких проектов, его следует помещать в сборки библиотек классов, как это делает Microsoft.

Создание библиотек классов

Первая задача — создать повторно используемую библиотеку классов .NET.

1. Откройте редактор кода и создайте библиотеку классов с такими настройками:
   1) шаблон проекта: Class Library/`classlib`;
   2) файл и папка рабочей области/решения: `Chapter05`;
   3) файл и папка проекта: `PacktLibrary`.
2. Откройте файл `PacktLibrary.csproj` и обратите внимание, что по умолчанию библиотеки классов нацелены на .NET 6 и, следовательно, могут работать только с другими сборками, совместимыми с .NET 6, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

3. Измените целевую платформу для поддержки .NET Standard 2.0 и удалите записи, допускающие неявное использование и значение NULL:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

4. Сохраните и закройте файл.
5. Удалите файл `Class1.cs`.

6. Скомпилируйте проект, чтобы другие проекты позже могли на него ссылаться:
   1) в Visual Studio Code введите следующую команду: `dotnet build`;
   2) в Visual Studio выберите команду меню Build ▸ Build PacktLibrary (Сборка ▸ Создать PacktLibrary).

Чтобы использовать новейшие функции языка C# и платформы .NET, поместите типы в библиотеку классов .NET 6. Для поддержки устаревших платформ .NET, таких как .NET Core, .NET Framework и Xamarin, поместите типы, которые повторно можно использовать, в библиотеку классов .NET Standard 2.0.

Определение классов в пространстве имен

Следующая задача — определить класс, который будет представлять человека.

1. Добавьте новый файл класса `Person.cs`.
2. Статически импортируйте `System.Console`.
3. Установите пространство имен `Packt.Shared`.

Мы делаем это, поскольку важно поместить ваши классы в логически названное пространство имен. Лучшее название пространства имен будет специфичным для домена, например System.Numerics для типов, связанных с расширенными числами. В нашем случае мы создадим типы Person, BankAccount и WondersOfTheWorld, и у них нет общего домена, поэтому мы будем использовать более общий Packt.Shared.

Ваш файл класса теперь должен выглядеть следующим образом:

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

namespace Packt.Shared
{
  public class Person
  {
  }
}
```

Обратите внимание, что ключевое слово `public` языка C# указывается перед классом. Это ключевое слово называется *модификатором доступа*, управляющим тем, как осуществляется доступ к данному классу.

Если вы явно не определили доступ к классу с помощью ключевого слова `public`, то он будет доступен только в определяющей его сборке. Это следствие того, что неявный модификатор доступа для класса считается `internal` (внутренним). Нам же нужно, чтобы класс был доступен за пределами сборки, поэтому необходимо ключевое слово `public`.

Упрощение объявлений пространств имен

Если вы работаете в .NET 6.0 и, следовательно, используете C# 10 или более позднюю версию, то можете завершить объявление пространства имен точкой с запятой и удалить фигурные скобки, чтобы упростить код:

```
using System;

namespace Packt.Shared; // класс в этом файле находится
                        // в данном пространстве имен

public class Person
{
}
```

Описанное выше также известно как объявление пространства имен в файловой области. У вас может быть только одно пространство имен в файловой области для каждого файла. Далее в главе мы будем использовать это в библиотеке классов, предназначенной для .NET 6.0.

Поместите каждый новый тип в отдельный файл, чтобы можно было использовать объявления пространств имен в файловой области.

Члены

У этого типа еще нет членов, инкапсулированных в него. Скоро мы их создадим. Членами могут быть поля, методы или специализированные версии их обоих. Их описание представлено ниже.

- *Поля* используются для хранения данных. Существует три специализированные категории полей:
  - *константы* — данные в них никогда не меняются. Компилятор буквально копирует данные в любой код, который их читает;
  - *поля, доступные только для чтения,* — данные в них не могут измениться после создания экземпляра класса, но могут быть рассчитаны или загружены из внешнего источника во время создания экземпляра;

- *события* — данные ссылаются на один или несколько методов, вызываемых автоматически при возникновении определенной ситуации, например при нажатии кнопки. Тема событий будет рассмотрена в главе 6.

- *Методы* используются для выполнения операторов. Вы уже ознакомились с некоторыми примерами в главе 4. Существует четыре специализированные категории методов:
  - *конструкторы* выполняются, когда вы используете ключевое слово `new` для выделения памяти и создания экземпляра класса;
  - *свойства* выполняются, когда необходимо получить доступ к данным. Они обычно хранятся в поле, но могут храниться извне или рассчитываться во время выполнения. Использование свойств — предпочтительный способ инкапсуляции полей, если только не требуется выдать наружу адрес памяти поля;
  - *индексаторы* выполняются, когда необходимо получить доступ к данным с помощью синтаксиса «массива» `[]`;
  - *операции* выполняются, когда необходимо применить операции типа `+` и `/` для операндов вашего типа.

Создание экземпляров классов

В этом подразделе мы создадим *экземпляр* класса `Person`.

Ссылка на сборку

Прежде чем мы сможем создать экземпляр класса, нам нужно сослаться на сборку, которая его содержит. Мы будем использовать класс в консольном приложении.

1. Откройте редактор кода и создайте консольное приложение `PeopleApp` в рабочей области/решении `Chapter05`.
2. Если вы используете Visual Studio Code:
   1) выберите `PeopleApp` в качестве активного проекта OmniSharp. Когда вы увидите всплывающее предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку Yes (Да), чтобы добавить их;
   2) измените файл `PeopleApp.csproj`, чтобы добавить ссылку на проект в `PacktLibrary`, как показано ниже (выделено жирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
```

```
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="../PacktLibrary/PacktLibrary.csproj" />
  </ItemGroup>

</Project>
```

3) на панели TERMINAL (Терминал) введите команду для компиляции проекта PeopleApp и его зависимого проекта PacktLibrary:

```
dotnet build
```

3. Если вы используете Visual Studio:
   1) настройте стартовый проект для решения в соответствии с текущим выбором;
   2) на панели Solution Explorer (Обозреватель решений) выберите проект PeopleApp, выберите команду меню Project ▸ Add Project Reference (Проект ▸ Добавить ссылку на проект), установите флажок, чтобы выбрать проект PacktLibrary, и нажмите кнопку OK;
   3) выберите команду меню Build ▸ Build PeopleApp (Сборка ▸ Создать PeopleApp).

Импорт пространства имен для использования типа

Теперь мы готовы написать операторы для работы с классом Person.

1. В проекте/папке PeopleApp откройте файл Program.cs.
2. В начале файла Program.cs удалите комментарий и введите следующие операторы, чтобы импортировать пространство имен для нашего класса Person и статически импортировать класс Console:

```
using Packt.Shared;
using static System.Console;
```

3. В файле Program.cs введите операторы:
   - для создания экземпляра типа Person;
   - для вывода экземпляра с помощью его текстового описания.

   Ключевое слово new выделяет память для объекта и инициализирует любые внутренние данные. Мы могли бы использовать var вместо имени класса Person, но тогда нам нужно было бы указать Person после ключевого слова new:

```
// Person bob = new Person(); // C# 1.0 или более поздние версии
// var bob = new Person(); // C# 3.0 или более поздние версии
Person bob = new(); // C# 9.0 или более поздние версии
WriteLine(bob.ToString());
```

Вы можете спросить: «Почему у переменной bob имеется метод ToString? Класс Person пуст!» Не беспокойтесь, скоро вы все узнаете!

4. Запустите код и проанализируйте результат:

```
Packt.Shared.Person
```

Работа с объектами

Хотя наш класс Person явно не наследуется ни от какого типа, все типы в конечном счете прямо или косвенно наследуются от специального типа System.Object.

Реализация метода ToString в типе System.Object выдает полные имена пространства имен и типа.

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 — базовый класс (или суперкласс), а Б — производный (подкласс). В нашем случае System.Object — базовый класс (суперкласс), а Person — производный (подкласс).

Мы также можем использовать псевдоним C# — ключевое слово object:

```
public class Person : object
```

Наследование System.Object

Сделаем наш класс явно наследуемым от object, а затем рассмотрим, какие члены имеют все объекты.

1. Измените свой класс Person, чтобы он мог явно наследовать от object.
2. Затем установите указатель мыши внутри ключевого слова object и нажмите клавишу F12 или щелкните правой кнопкой мыши на ключевом слове object и выберите Go to Definition (Перейти к определению).

Вы увидите определенные Microsoft тип System.Object и его члены. Вам пока не нужно детально разбираться во всем этом, но обратите внимание на метод ToString, показанный на рис. 5.1.

Исходите из предположения, что другие программисты знают о том, что если наследование не указано, то класс наследуется от System.Object.

Рис. 5.1. Определение класса System.Object

Хранение данных в полях

Теперь определим в классе несколько полей для хранения информации о человеке.

Определение полей

Допустим, мы решили, что информация о человеке включает имя и дату рождения. Мы инкапсулируем оба значения в классе `Person` и сделаем поля общедоступными.

В классе `Person` напишите операторы, чтобы объявить два общедоступных поля для хранения имени и даты рождения человека:

```
public class Person : object
{
  // поля
  public string Name;
  public DateTime DateOfBirth;
}
```

Вы можете использовать любые типы для полей, включая массивы и коллекции, такие как списки и словари. С их помощью вы сможете хранить несколько значений в одном именованном поле. В данном примере информация о человеке содержит только одно имя и одну дату рождения.

Модификаторы доступа

При реализации инкапсуляции важно выбрать степень видимости членов.

Обратите внимание: работая с классом, мы использовали ключевое слово `public` по отношению к созданным полям. Если бы мы этого не сделали, то поля были бы закрытыми, то есть доступными только в пределах класса.

Существует четыре ключевых слова для модификаторов доступа и две их комбинации. Вы можете применить их к члену класса, например к полю или методу, как показано в табл. 5.1.

Таблица 5.1. Модификаторы доступа

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 типом и любым другим типом, который наследуется от него
`public`	Неограниченный доступ
`internal protected`	Доступ ограничен содержащим типом и любым другим типом в текущей сборке, а также любым другим типом, который наследуется от содержащего. Аналогичен вымышленному модификатору доступа `internal_or_protected`
`private protected`	Доступ ограничен содержащим типом и любым другим типом, который наследуется от него и находится в той же сборке. Аналогичен вымышленному модификатору доступа `internal_and_protected`. Эта комбинация доступна только для версии C# 7.2 или более поздних

Следует явно применять один из модификаторов доступа ко всем членам типа, даже если будет использоваться модификатор по умолчанию, именуемый private. Кроме того, поля, как правило, должны быть private или protected, и вам следует создать общедоступные свойства public, чтобы получить или установить значения полей, поскольку так можно контролировать доступ. Вы сделаете это позже в данной главе.

Установка и вывод значений полей

Теперь мы будем использовать эти поля в вашем коде.

1. Убедитесь, что в начале файла `Program.cs` импортировано пространство имен `System`. Нам необходимо сделать это, чтобы использовать тип `DateTime`.

2. После того как экземпляр bob будет создан, добавьте следующие операторы, чтобы задать имя и дату рождения человека, а затем выведите эти поля в надлежащем формате:

```
bob.Name = "Bob Smith";
bob.DateOfBirth = new DateTime(1965, 12, 22); // C# 1.0 или более
                                              // поздние версии

WriteLine(format: "{0} was born on {1:dddd, d MMMM yyyy}",
  arg0: bob.Name,
  arg1: bob.DateOfBirth);
```

Мы могли бы также использовать интерполяцию строк, но длинные строки будут переноситься на несколько строк, что может усложнить чтение кода в данной книге. Обратите внимание: в примерах кода, описанных в книге, {0} — это заполнитель для arg0 и т. д.

3. Запустите код и проанализируйте результат:

```
Bob Smith was born on Wednesday, 22 December 1965
```

Помните, что вывод может выглядеть по-разному в зависимости от вашего региона, то есть языка и культуры.

Код формата для arg1 состоит из нескольких частей: dddd означает день недели, d — день месяца, MMMM — название месяца. Строчные буквы m используются для значений времени в минутах, yyyy означает полное число года, yy — год, обозначенный двумя цифрами.

Вы также можете инициализировать поля, используя сокращенный синтаксис *инициализатора объекта* с фигурными скобками. Рассмотрим пример.

4. Добавьте операторы ниже существующего кода, чтобы создать запись еще об одном человеке, женщине по имени Alice. Обратите внимание на иной формат кода даты рождения при записи в консоль:

```
Person alice = new()
{
  Name = "Alice Jones",
  DateOfBirth = new(1998, 3, 7) // C# 9.0 и более поздние версии
};

WriteLine(format: "{0} was born on {1:dd MMM yy}",
  arg0: alice.Name,
  arg1: alice.DateOfBirth);
```

5. Запустите код и проанализируйте результат:

```
Alice Jones was born on 07 Mar 98
```

Хранение значения с помощью типа 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 ограниченного списка. Например, человек может хотеть посетить несколько определенных чудес света из числа тех семи. Мы можем сохранять такие данные, определив тип `enum`.

Тип `enum` весьма эффективен по части хранения одного или нескольких вариантов, поскольку внутри использует целочисленные значения в сочетании с таблицей поиска описаний строк.

1. Добавьте новый файл `WondersOfTheAncientWorld.cs` в проект `PacktLibrary`.
2. Измените код в файле `WondersOfTheAncientWorld.cs`:

```
namespace Packt.Shared
{
  public enum WondersOfTheAncientWorld
  {
    GreatPyramidOfGiza,
    HangingGardensOfBabylon,
    StatueOfZeusAtOlympia,
    TempleOfArtemisAtEphesus,
    MausoleumAtHalicarnassus,
    ColossusOfRhodes,
    LighthouseOfAlexandria
  }
}
```

Если вы используете блокнот .NET Interactive, то ячейка кода, содержащая тип enum, должна находиться над ячейкой кода, определяющей класс `Person`.

3. В классе `Person` добавьте в список полей следующий оператор:

```
public WondersOfTheAncientWorld FavoriteAncientWonder;
```

4. В файле `Program.cs` добавьте следующие операторы:

```
bob.FavoriteAncientWonder = WondersOfTheAncientWorld.
StatueOfZeusAtOlympia;

WriteLine(
format: "{0}'s favorite wonder is {1}. Its integer is {2}.",
  arg0: bob.Name,
  arg1: bob.FavoriteAncientWonder,
  arg2: (int)bob.FavoriteAncientWonder);
```

5. Запустите код и проанализируйте результат:

```
Bob Smith's favorite wonder is StatueOfZeusAtOlympia. Its integer is 2.
```

В целях эффективности значение `enum` хранится как `int`. Значения `int` присваиваются автоматически, начиная с `0`, поэтому третье чудо света в нашем перечислении имеет значение `2`. Вы можете присваивать значения `int`, которые не перечислены в списке. Они будут выводиться как значение `int` вместо имени, поскольку совпадение не будет найдено.

Хранение группы значений с помощью типа enum

Для списка перечислений мы могли бы создать коллекцию экземпляров `enum` (о коллекциях поговорим чуть позже), но есть способ получше. Мы можем скомбинировать несколько вариантов в одно значение с помощью *флагов*.

1. Измените перечисление, дополнив его атрибутом `[System.Flags]`. Явно установите значение `byte` для каждого чуда света, представляющего разные битовые столбцы:

```
namespace Packt.Shared
{
  [System.Flags]
  public enum WondersOfTheAncientWorld : byte
  {
    None                      = 0b_0000_0000, // то есть 0
    GreatPyramidOfGiza        = 0b_0000_0001, // то есть 1
    HangingGardensOfBabylon   = 0b_0000_0010, // то есть 2
    StatueOfZeusAtOlympia     = 0b_0000_0100, // то есть 4
    TempleOfArtemisAtEphesus  = 0b_0000_1000, // то есть 8
    MausoleumAtHalicarnassus  = 0b_0001_0000, // то есть 16
    ColossusOfRhodes          = 0b_0010_0000, // то есть 32
    LighthouseOfAlexandria    = 0b_0100_0000  // то есть 64
  }
}
```

Мы явно присваиваем значения каждому варианту, которые не будут перекрывать друг друга при просмотре битов, хранящихся в памяти. Мы также должны дополнить тип `enum` атрибутом `System.Flags`, поэтому при возврате значения оно может автоматически сопоставляться с несколькими строками, разделенными запятыми, вместо того чтобы возвращать значение `int`.

Обычно тип `enum` использует переменные `int`, но поскольку нам не нужны настолько большие значения, мы можем уменьшить требования к памяти на 75 % (то есть 1 байт на значение вместо 4 байт), дав ему указание использовать переменные типа `byte`.

Если требуется указать, что наш список перечислений 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. Значения битов

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

2. В классе `Person` добавьте к списку полей следующий оператор:

```
public WondersOfTheAncientWorld BucketList;
```

3. В файл `Program.cs` добавьте следующие операторы, чтобы установить список перечислений с помощью операции `|` (логическое ИЛИ) для комбинирования значений `enum`. Мы также могли бы установить значение, приведя число `18` к типу `enum`, как показано в комментарии, но не должны этого делать, поскольку это затруднит понимание кода:

```
bob.BucketList =
  WondersOfTheAncientWorld.HangingGardensOfBabylon
  | WondersOfTheAncientWorld.MausoleumAtHalicarnassus;

// bob.BucketList = (WondersOfTheAncientWorld)18;

WriteLine($"{bob.Name}'s bucket list is {bob.BucketList}");
```

4. Запустите код и проанализируйте результат:

```
Bob Smith's bucket list is HangingGardensOfBabylon,
MausoleumAtHalicarnassus
```

Используйте значения enum для сохранения комбинаций отдельных вариантов. Наследуйте тип enum от byte, если вариантов не более 8, от ushort — если их не более 16, от uint — если не более 32 и от ulong — если не более 64.

Хранение нескольких значений с помощью коллекций

Добавим поле для хранения данных о детях человека. Перед вами пример агрегирования, поскольку дети — экземпляр класса, связанного с этим человеком, но не часть самого человека. Мы будем использовать дженерик коллекции `List<T>`, который может хранить упорядоченную коллекцию любого типа. О коллекциях мы поговорим в главе 8, а пока просто следуйте инструкциям, изложенным ниже.

1. В начало файла класса `Person.cs` импортируйте пространство имен `System.Collections.Generic`:

```
using System.Collections.Generic; // List<T>
```

2. Объявите новое поле в классе `Person`:

```
public List<Person> Children = new List<Person>();
```

Код `List<Person>` читается как «список `Person`», например «тип свойства `Children` — это список экземпляров `Person`». Мы явно изменили библиотеку классов, чтобы она поддерживалась платформой .NET Standard 2.0 (использующей компилятор C# 7), поэтому не можем использовать оператор `new` с целевым типом для инициализации поля `Children`. Если бы библиотека поддерживалась платформой .NET 6.0, то могли бы использовать оператор `new` с целевым типом следующим образом:

```
public List<Person> Children = new();
```

Мы должны убедиться, что коллекция инициализируется новым экземпляром списка `Person`, прежде чем сможем добавить в нее элементы, иначе поле будет `null` и выдаст исключения времени выполнения, когда мы пытаемся использовать любой из его членов, например `Add`.

Коллекции дженериков

Угловые скобки в названии типа `List<T>` — это функциональность языка C#, которая называется *«дженерики»* и была добавлена в 2005 году в версии языка C# 2.0. Это просто красивый способ сделать коллекцию *сильно типизированной*, а именно такой, чтобы компилятор максимально точно знал, какие типы объектов могут храниться в коллекции. Дженерики улучшают производительность и правильность кода.

Сильная типизированность отличается от *статической*. Старые типы `System.Collection` статически типизированы и содержат слабо типизированные элементы `System.Object`. Новые типы `System.Collection.Generic` статически типизированы и содержат сильно типизированные экземпляры `<T>`.

По иронии судьбы термин *«дженерики»* (от англ. generic — «обобщенный») означает, что мы можем использовать более конкретный статический тип!

1. В файле `Program.cs` введите следующие операторы, чтобы добавить двух детей Боба, а затем показать, сколько у него детей и как их зовут:

```
bob.Children.Add(new Person { Name = "Alfred" }); // C# 3.0 и более
                                                  // поздние версии
bob.Children.Add(new() { Name = "Zoe" }); // C# 9.0 и более поздние версии
```

```
WriteLine(
  $"{bob.Name} has {bob.Children.Count} children:");
for (int childIndex = 0; childIndex < bob.Children.Count; childIndex++)
{
  WriteLine($" {bob.Children[childIndex].Name}");
}
```

Мы также можем использовать оператор `foreach` для перечисления коллекции. В качестве дополнительной задачи измените оператор `for` так, чтобы он выводил ту же информацию с помощью `foreach`.

2. Запустите код и проанализируйте результат:

```
Bob Smith has 2 children:
  Alfred
  Zoe
```

Создание статического поля

Поля, которые мы создавали до сих пор, были членами *экземпляра*; это значит, для каждого созданного экземпляра класса существует разное значение каждого поля. Переменные `bob` и `alice` имеют разные значения поля `Name`.

Иногда требуется определить поле, в котором есть только одно значение, общее для всех экземпляров. Это так называемые *статические* члены, поскольку поля — не единственные члены, которые могут быть статическими.

Далее рассмотрим, чего можем достичь с помощью полей `static`.

1. В проект `PacktLibrary` добавьте файл класса `BankAccount.cs`.
2. Измените класс, предоставив ему три поля, два поля экземпляра и одно статическое поле:

```
namespace Packt.Shared
{
  public class BankAccount
  {
    public string AccountName; // член экземпляра
    public decimal Balance; // член экземпляра
    public static decimal InterestRate; // общий член
  }
}
```

Каждый экземпляр класса `BankAccount` будет иметь собственные значения `AccountName` и `Balance`, но все экземпляры будут использовать общее значение `InterestRate`.

3. В файл `Program.cs` добавьте операторы для установления общей процентной ставки, а затем создайте два экземпляра типа `BankAccount`:

```
BankAccount.InterestRate = 0.012M; // хранит общее значение

BankAccount jonesAccount = new(); // C# 9.0 и более поздние версии
jonesAccount.AccountName = "Mrs. Jones";
jonesAccount.Balance = 2400;

WriteLine(format: "{0} earned {1:C} interest.",
  arg0: jonesAccount.AccountName,
  arg1: jonesAccount.Balance * BankAccount.InterestRate);

BankAccount gerrierAccount = new();
gerrierAccount.AccountName = "Ms. Gerrier";
gerrierAccount.Balance = 98;

WriteLine(format: "{0} earned {1:C} interest.",
  arg0: gerrierAccount.AccountName,
  arg1: gerrierAccount.Balance * BankAccount.InterestRate);
```

`:C` — это код, благодаря которому .NET понимает, какой формат представления валюты для чисел нужно использовать. В главе 8 вы узнаете, как управлять региональными настройками, определяющими символ валюты. На данный момент будет использоваться значение по умолчанию для вашей операционной системы. Я живу в Лондоне, поэтому в моем результате будут отображаться британские фунты стерлингов (`£`).

4. Запустите код и проанализируйте результат:

```
Mrs. Jones earned £28.80 interest.
Ms. Gerrier earned £1.18 interest.
```

Поля — не единственные элементы, которые могут быть статическими. Конструкторы, методы, свойства и другие элементы также могут быть статическими.

Создание константного поля

Если значение поля *никогда не* должно меняться, то вы можете использовать ключевое слово `const` и присваивать значение во время компиляции.

1. В файле `Program.cs` добавьте код, приведенный ниже:

```
// константы
public const string Species = "Homo Sapiens";
```

2. Чтобы получить значение константного поля, вы должны написать имя класса, а не имя экземпляра класса. В файл `Program.cs` добавьте оператор для вывода имени и вида Боба в консоль:

```
WriteLine($"{bob.Name} is a {Person.Species}");
```

3. Запустите приложение и проанализируйте результат:

```
Bob Smith is a Homo Sapiens
```

Примеры полей `const` в типах Microsoft включают `System.Int32.MaxValue` и `System.Math.PI`, поскольку ни одно из этих значений никогда не изменится, как вы можете видеть на рис. 5.2.

Рис. 5.2. Примеры констант

Констант следует избегать по двум важным причинам: значение должно быть известно во время компиляции и должно выражаться в виде литеральной строки, логического или числового значения. Каждая ссылка в поле const заменяется литеральным значением во время компиляции, что, следовательно, не будет изменено, если значение изменится в будущей версии, а вы не перекомпилируете сборки, которые ссылаются на него, чтобы получить новое значение.

Создание поля только для чтения

Как правило, лучший способ создать поля, значения которых не должны меняться, — пометить их как «только для чтения».

1. В файле `Program.cs` добавьте следующий оператор, чтобы объявить экземпляр поля, доступного только для чтения, для хранения информации о родной планете человека:

```
// поля только для чтения
public readonly string HomePlanet = "Earth";
```

2. В файле `Program.cs` добавьте оператор для записи имени Боба и родной планеты в консоль:

```
WriteLine($"{bob.Name} was born on {bob.HomePlanet}");
```

3. Запустите код и проанализируйте результат:

```
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 любого исполняемого оператора. Таким образом, поле только для чтения может быть установлено с применением конструктора. Каждая ссылка в поле активна, поэтому любые будущие изменения будут правильно отражены при вызове кода.

Вы также можете объявить поля `static readonly` (только для чтения), значения которых будут общими для всех экземпляров типа.

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 любые поля будут установлены кодом, использующим этот тип.

1. В файле `Person.cs` добавьте операторы после существующего поля `HomePlanet`, доступного только для чтения, чтобы определить второе поле только для чтения, а затем установить поля `Name` и `Instantiated` в конструктор:

```
// поля только для чтения
public readonly string HomePlanet = "Earth";
public readonly DateTime Instantiated;

// конструкторы
public Person()
{
  // установка значений по умолчанию для полей,
  // включая поля только для чтения
  Name = "Unknown";
  Instantiated = DateTime.Now;
}
```

2. В файле `Program.cs` добавьте операторы для создания экземпляра нового человека, а затем выведите начальные значения его полей:

```
Person blankPerson = new();

WriteLine(format:
  "{0} of {1} was created at {2:hh:mm:ss} on a {2:dddd}.",
  arg0: blankPerson.Name,
  arg1: blankPerson.HomePlanet,
  arg2: blankPerson.Instantiated);
```

3. Запустите код и проанализируйте результат:

```
Unknown of Earth was created at 11:58:12 on a Sunday
```

Определение нескольких конструкторов

Вы можете иметь несколько конструкторов в типе. Это позволяет побудить разработчиков устанавливать начальные значения для полей.

1. В файле `Person.cs` добавьте следующие операторы, чтобы определить второй конструктор, который позволяет разработчику устанавливать начальные значения для имени человека и родной планеты:

```
public Person(string initialName, string homePlanet)
{
  Name = initialName;
  HomePlanet = homePlanet;
  Instantiated = DateTime.Now;
}
```

2. В файле `Program.cs` добавьте операторы для создания еще одного человека, используя конструктор с двумя параметрами.

```
Person gunny = new(initialName: "Gunny", homePlanet: "Mars");

WriteLine(format:
  "{0} of {1} was created at {2:hh:mm:ss} on a {2:dddd}.",
  arg0: gunny.Name,
  arg1: gunny.HomePlanet,
  arg2: gunny.Instantiated);
```

3. Запустите код и проанализируйте результат:

```
Gunny of Mars was created at 11:59:25 on a Sunday
```

Конструкторы представляют собой особую категорию методов. Рассмотрим их более подробно.

Запись и вызов методов

Методы — это члены типа, которые выполняют блок операторов. Это функции, принадлежащие типу.

Возвращение значений из методов

Методы могут возвращать одно значение или ничего не возвращать.

- Если указать тип возврата `void` перед именем метода, то метод выполнит определенные действия, но не вернет значение.
- Если перед именем метода указать тип возвращаемого значения, то метод выполнит определенные действия и вернет значение.

В качестве примера вам предстоит создать два метода:

- `WriteToConsole` — выполнит действие (запись строки), но ничего не возвратит из метода, определенного со словом `void`;
- `GetOrigin` — вернет строковое значение, поскольку перед именем метода указано ключевое слово `string`.

Теперь напишем код.

1. В файле `Person.cs` добавьте операторы для определения двух методов:

```
// методы
public void WriteToConsole()
{
  WriteLine($"{Name} was born on a {DateOfBirth:dddd}.");
}

public string GetOrigin()
{
  return $"{Name} was born on {HomePlanet}.";
}
```

2. В файле `Program.cs` добавьте операторы для вызова двух методов:

```
bob.WriteToConsole();
WriteLine(bob.GetOrigin());
```

3. Запустите код и проанализируйте результат:

```
Bob Smith was born on a Wednesday.
Bob Smith was born on Earth.
```

Возвращение нескольких значений с помощью кортежей

Методы могут возвращать только одно значение одного типа. Он может быть простым, таким как `string` в предыдущем примере, сложным, например `Person`, или коллекцией, например `List<Person>`.

Представьте, что нужно определить метод, который возвращает оба значения: и `string`, и `int`. Мы могли бы определить новый класс `TextAndNumber` с полями `string` и `int` и вернуть экземпляр этого сложного типа, как показано ниже:

```
public class TextAndNumber
{
  public string Text;
  public int Number;
}

public class LifeTheUniverseAndEverything
{
  public TextAndNumber GetTheData()
  {
    return new TextAndNumber
    {
      Text = "What's the meaning of life?",
      Number = 42
    };
  }
}
```

Однако определять класс просто для объединения двух значений не нужно, поскольку в современных версиях C# мы можем использовать *кортежи*. Это эффективный способ объединить два и более значения в одно целое.

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 в 2010 году (был добавлен тип `System.Tuple`).

Языковая поддержка кортежей

Только в C# 7.0 в 2017 году была добавлена поддержка синтаксиса языка для кортежей, использующих круглые скобки `()`. Наряду с началом поддержки кортежей в языке C# 7 на платформе .NET стал доступен новый тип `System.ValueTuple`, который более эффективен в некоторых распространенных ситуациях, чем старый тип `System.Tuple` из версии .NET 4.0.

Рассмотрим кортежи на примере.

1. В файле `Person.cs` добавьте операторы для определения метода, который возвращает кортеж со значениями типа `string` и `int`:

```
public (string, int) GetFruit()
{
  return ("Apples", 5);
}
```

2. В файле `Program.cs` добавьте операторы для вызова метода `GetFruit`, а затем выведите поля кортежа с автоматическими именами `Item1` и `Item2`:

```
(string, int) fruit = bob.GetFruit();

WriteLine($"{fruit.Item1}, {fruit.Item2} there are.");
```

3. Запустите код и проанализируйте результат:

```
Apples, 5 there are.
```

Присвоение имен полям кортежа

Для доступа к полям кортежа используются имена по умолчанию: `Item1`, `Item2` и т. д.

Вы можете явно указать имена полей.

1. В файле `Person.cs` добавьте операторы для определения метода, который возвращает кортеж с именованными полями, как показано ниже:

```
public (string Name, int Number) GetNamedFruit()
{
  return (Name: "Apples", Number: 5);
}
```

2. В файле `Program.cs` добавьте операторы для вызова метода и выведите именованные поля кортежа:

```
var fruitNamed = bob.GetNamedFruit();

WriteLine($"There are {fruitNamed.Number} {fruitNamed.Name}.");
```

3. Запустите код и проанализируйте результат:

```
There are 5 Apples.
```

Автоматическое определение имен элементов кортежей

Если вы создаете кортеж из другого объекта, то можете использовать функцию, представленную в C# 7.1, которая называется *автоматическим определением имени элемента кортежа*.

В файле Program.cs создайте два кортежа, каждый из которых состоит из значений string и int:

```
var thing1 = ("Neville", 4);
WriteLine($"{thing1.Item1} has {thing1.Item2} children.");

var thing2 = (bob.Name, bob.Children.Count);
WriteLine($"{thing2.Name} has {thing2.Count} children.");
```

В версии C# 7.0 оба кортежа будут использовать схемы именования Item1 и Item2. В версии C# 7.1 и более поздних версиях во втором примере можно использовать имена Name и Count.

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:

```
// сохранение возвращаемого значения в переменной кортежа с двумя полями
(string TheName, int TheNumber) tupleWithNamedFields = bob.GetNamedFruit();
// tupleWithNamedFields.TheName
// tupleWithNamedFields.TheNumber

// деконструкция возвращаемого значения на две отдельные переменные
(string name, int number) = GetNamedFruit();
// переменая name
// переменая number
```

Это приводит к разделению кортежа на части и назначению указанных частей новым переменным.

1. В файле Program.cs добавьте операторы для деконструкции кортежа, возвращаемого из метода GetFruit:

   ```
   (string fruitName, int fruitNumber) = bob.GetFruit();

   WriteLine($"Deconstructed: {fruitName}, {fruitNumber}");
   ```

2. Запустите код и проанализируйте результат:

   ```
   Deconstructed: Apples, 5
   ```

Деконструкция типов

Кортежи — не единственный тип, который можно деконструировать. Любой тип может иметь специальные методы Deconstruct, которые разбивают объект на части. Реализуем некоторые из этих методов для класса Person.

1. В файл `Person.cs` добавьте два метода `Deconstruct` с параметрами `out`, определенными для частей, которые необходимо деконструировать:

```
// деконструкторы
public void Deconstruct(out string name, out DateTime dob)
{
  name = Name;
  dob = DateOfBirth;
}

public void Deconstruct(out string name,
  out DateTime dob, out WondersOfTheAncientWorld fav)
{
  name = Name;
  dob = DateOfBirth;
  fav = FavoriteAncientWonder;
}
```

2. В файл `Program.cs` добавьте операторы для деконструкции объекта `bob`:

```
// деконструкция Person

var (name1, dob1) = bob;
WriteLine($"Deconstructed: {name1}, {dob1}");

var (name2, dob2, fav2) = bob;
WriteLine($"Deconstructed: {name2}, {dob2}, {fav2}");
```

3. Запустите код и проанализируйте результат:

```
Deconstructed: Bob Smith, 22/12/1965 00:00:00
Deconstructed: Bob Smith, 22/12/1965 00:00:00, StatueOfZeusAtOlympia
B
```

Определение и передача параметров в методы

Методы могут иметь параметры, переданные им для изменения их поведения. Параметры определяются по аналогии с объявлением переменных, только внутри круглых скобок кода метода, как вы видели чуть выше в главе, когда мы говорили о конструкторах. Рассмотрим пример.

1. В файле `Person.cs` добавьте операторы для определения двух методов: первого без параметров и второго с одним параметром:

```
public string SayHello()
{
  return $"{Name} says 'Hello!'";
}
```

```
public string SayHelloTo(string name)
{
  return $"{Name} says 'Hello {name}!'";
}
```

2. В файле `Program.cs` добавьте операторы для вызова двух методов и запишите возвращаемое значение в консоль:

```
WriteLine(bob.SayHello());
WriteLine(bob.SayHelloTo("Emily"));
```

3. Запустите код и проанализируйте результат:

```
Bob Smith says 'Hello!'
Bob Smith says 'Hello Emily!'
```

При вводе оператора, который вызывает метод, технология IntelliSense отображает всплывающую подсказку, содержащую имя и тип любых параметров, а также возвращаемый тип метода (рис. 5.3).

Рис. 5.3. Всплывающая подсказка IntelliSense для метода без перегрузок

Перегрузка методов

Вместо двух методов с разными именами мы могли бы присвоить обоим методам одно и то же имя. Это допустимо, поскольку каждому методу присуща собственная сигнатура.

Сигнатура метода — список типов параметров, которые могут быть переданы при вызове метода. Перегруженные методы не могут отличаться только типом возвращаемого значения.

1. В файле `Person.cs` измените имя метода `SayHelloTo` на `SayHello`.
2. В файле `Program.cs` измените вызов метода, чтобы использовать метод `SayHello`, и обратите внимание, что в краткой информации о методе содержатся сведения о наличии у него одной дополнительной перегрузки, 1/2, а также 2/2 (рис. 5.4).

Используйте перегруженные методы для упрощения классов так, чтобы казалось, что у них меньше методов.

Рис. 5.4. Всплывающая подсказка IntelliSense для перегруженного метода

Передача необязательных и именованных параметров

Другой способ упростить методы заключается в использовании необязательных параметров. Вы определяете параметр как необязательный, устанавливая значение по умолчанию в списке параметров метода. Необязательные параметры всегда должны указываться последними в списке параметров.

Сейчас вы создадите метод с тремя необязательными параметрами.

1. В файле `Person.cs` добавьте операторы для определения метода:

```
public string OptionalParameters(
  string command = "Run!",
  double number = 0.0,
  bool active = true)
{
  return string.Format(
    format: "command is {0}, number is {1}, active is {2}",
    arg0: command,
    arg1: number,
    arg2: active);
}
```

2. В файле `Program.cs` добавьте оператор для вызова метода и запишите его возвращаемое значение в консоль:

```
WriteLine(bob.OptionalParameters());
```

3. При вводе кода появится меню IntelliSense с краткими сведениями, в которых указаны три необязательных параметра со значениями по умолчанию (рис. 5.5).

Рис. 5.5. IntelliSense отображает необязательные параметры при вводе кода

4. Запустите код и проанализируйте результат:

```
command is Run!, number is 0, active is True
```

5. В файле `Program.cs` добавьте следующий оператор, чтобы передать значение `string` для параметра `command` и значение `double` для параметра `number`:

```
WriteLine(bob.OptionalParameters("Jump!", 98.5));
```

6. Запустите код и проанализируйте результат:

```
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`.

Именование значений параметров при вызове методов

Необязательные параметры при вызове метода часто сочетаются с именованными, поскольку именование параметра позволяет передавать значения в порядке, отличном от того, в котором они были объявлены.

1. В файле `Program.cs` добавьте следующий оператор, чтобы передать значение `string` для параметра `command` и значение `double` для параметра `number`, но уже с использованием именованных параметров, чтобы порядок их передачи можно было изменить:

```
WriteLine(bob.OptionalParameters(
  number: 52.7, command: "Hide!"));
```

2. Запустите код и проанализируйте результат:

```
command is Hide!, number is 52.7, active is True
```

Вы даже можете использовать именованные параметры, чтобы пропустить необязательные параметры.

3. В файле `Program.cs` добавьте следующий оператор, чтобы передать значение `string` для параметра `command` с помощью позиционного порядка, пропустите параметр `number` и используйте именованный параметр `active`:

```
WriteLine(bob.OptionalParameters("Poke!", active: false));
```

4. Запустите приложение и проанализируйте результат:

```
command is Poke!, number is 0, active is False
```

Управление передачей параметров

Передать параметр в метод можно 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»*.

Рассмотрим несколько примеров передачи параметров.

1. В файле `Person.cs` добавьте операторы для определения метода с тремя параметрами: `in`, `ref` и `out`:

```
public void PassingParameters(int x, ref int y, out int z)
{
  // параметры out не могут иметь значения по умолчанию
  // и должны быть инициализированы внутри метода
  z = 99;

  // увеличиваем каждый параметр
  x++;
  y++;
  z++;
}
```

2. В файле `Program.cs` добавьте операторы для объявления нескольких переменных `int` и передачи их в метод:

```
int a = 10;
int b = 20;
int c = 30;

WriteLine($"Before: a = {a}, b = {b}, c = {c}");
bob.PassingParameters(a, ref b, out c);
WriteLine($"After: a = {a}, b = {b}, c = {c}");
```

3. Запустите код и проанализируйте результат:

```
Before: a = 10, b = 20, c = 30
After: a = 10, b = 21, c = 100
```

   - При передаче переменной как параметра по умолчанию передается текущее значение, а не сама переменная. Следовательно, у `x` есть копия переменной `a`. Данная переменная сохраняет свое первоначальное значение `10`.
   - При передаче переменной в качестве параметра `ref` в метод передается ссылка на переменную. Следовательно, `y` — ссылка на переменную `b`. Данная переменная инкрементируется при инкрементировании параметра `y`.

- При передаче переменной в качестве параметра `out` в метод передается ссылка на переменную. Следовательно, `z` — ссылка на переменную `c`. Значение переменной `c` заменяется любым кодом, выполняемым в методе. Мы могли бы упростить код в методе `Main`, не присваивая значение `30` переменной `c`, поскольку оно всегда будет заменяться.

Упрощение параметров

В языке C# 7.0 можно упростить код, использующий переменные `out`.

В файле `Program.cs` добавьте операторы для объявления еще нескольких переменных, включая параметр `out` с именем `f`, объявленный прямо в операторе вызова метода (`inline`), как показано ниже:

```
int d = 10;
int e = 20;

WriteLine($"Before: d = {d}, e = {e}, f doesn't exist yet!");

// упрощенный синтаксис параметра out (C# 7.0 или более поздней версии)
bob.PassingParameters(d, ref e, out int f);
WriteLine($"After: d = {d}, e = {e}, f = {f}");
```

Ключевое слово ref

В версии C# 7.0 или более поздних ключевое слово `ref` используется не только для передачи параметров в метод, но и для значения `return`. Это позволяет внешней переменной ссылаться на внутреннюю и изменять ее значение после вызова метода. Это может быть полезно в сложных сценариях, например, для передачи пустых контейнеров, требующих последующего заполнения, в больших структурах данных, но выходит за рамки данной книги.

Разделение классов с помощью ключевого слова partial

При командной работе над большими проектами или при работе с особенно большими и сложными реализациями классов удобно разделять определения сложных классов на несколько файлов. Сделать это можно с помощью ключевого слова `partial`.

Допустим, нам необходимо добавить в класс `Person` операторы, которые автоматически создаются инструментом, подобным ORM (object-relational mapper — «объектно-реляционный отображатель»), который считывает информацию о схеме из базы данных. Если класс определен как `partial`, то мы можем разбить его на файлы кода, которые сгенерированы автоматически и отредактированы вручную.

Рассмотрим пример.

1. В файле `Person.cs` добавьте ключевое слово `partial`, как показано ниже (выделено жирным шрифтом):

```
namespace Packt.Shared
{
  public partial class Person
  {
```

2. В проекте/папке `PacktLibrary` добавьте новый файл класса `PersonAutoGen.cs`.
3. Добавьте операторы в новый файл:

```
namespace Packt.Shared
{
  public partial class Person
  {
  }
}
```

Остальная часть кода, который мы напишем в этой главе, будет находиться в файле `PersonAutoGen.cs`.

Управление доступом с помощью свойств и индексаторов

Ранее вы создали метод `GetOrigin`, который возвращал значение `string`, содержащее информацию об имени и происхождении человека. В таких языках, как Java, этот способ используется часто. В C# есть более хороший подход: *свойства*.

Свойство — 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

Свойство только для чтения имеет лишь реализацию `get`.

1. В файле `PersonAutoGen.cs` в классе `Person` добавьте операторы для определения трех свойств:
   1) первое играет ту же роль, что и метод `GetOrigin`, используя синтаксис свойств, который работает со всеми версиями языка C# (хотя задействует синтаксис интерполяции строк из C# 6 и более поздних версий);

2) второе возвращает приветственное сообщение, используя синтаксис лямбда-выражения (=>) из C# 6 и более поздних версий;

3) третье вычисляет возраст человека.

Код представлен ниже:

```
// свойство, определенное с помощью синтаксиса C# 1–5
public string Origin
{
  get
  {
    return $"{Name} was born on {HomePlanet}";
  }
}

// два свойства, определенные с помощью синтаксиса
// лямбда-выражений C# 6+
public string Greeting => $"{Name} says 'Hello!'";

public int Age => System.DateTime.Today.Year - DateOfBirth.Year;
```

Очевидно, это не лучший способ подсчета возраста, но мы не учимся рассчитывать возраст по датам рождения. О том, как сделать это правильно, можно узнать на сайте https://stackoverflow.com/questions/9/how-do-i-calculate-someones-age-in-c.

2. В файле `Program.cs` добавьте операторы для получения свойств:

```
Person sam = new()
{
  Name = "Sam",
  DateOfBirth = new(1972, 1, 27)
};

WriteLine(sam.Origin);
WriteLine(sam.Greeting);
WriteLine(sam.Age);
```

3. Запустите код и проанализируйте результат:

```
Sam was born on Earth
Sam says 'Hello!'
49
```

В выводе отображается `49`, поскольку я запустил консольное приложение 15 августа 2021 года, когда Сэму было 49 лет.

Определение изменяемых свойств

Чтобы создать изменяемое свойство, вы должны использовать старый синтаксис и предоставить пару методов: не только часть `get`, но и часть `set`.

1. В файле `PersonAutoGen.cs` добавьте оператор для определения свойства `string`, которое имеет методы `get` и `set` (также известные как геттер и сеттер):

   ```
   public string FavoriteIceCream { get; set; } // автосинтаксис
   ```

   Несмотря на то что вы не создали вручную поле для хранения названия любимого мороженого человека, оно создается автоматически компилятором.

   Иногда требуется дополнительный контроль над установкой свойств. В этом случае следует использовать более детализированный синтаксис и вручную создать поле `private` для хранения значения свойства.

2. В файле `PersonAutoGen.cs` добавьте операторы для определения поля `string` и свойства `string`, которое имеет и `get`, и `set`:

   ```
   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.");
       }
     }
   }
   ```

Не рекомендуется добавлять в геттеры и сеттеры слишком много кода. Это будет означать, что у вас проблема со структурой написания кода. Чтобы упростить реализации, рассмотрите возможность добавления частных методов, которые вы будете вызывать в сеттерах и геттерах.

3. В файле `Program.cs` добавьте следующие операторы, чтобы задать любимое мороженое и цвет Сэма, а затем запишите их в консоль:

```
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}.");
```

4. Запустите код и проанализируйте результат:

```
Sam's favorite ice-cream flavor is Chocolate Fudge.
Sam's favorite primary color is Red.
```

Если вы попытаетесь установить для цвета любое значение, кроме красного, зеленого или синего, то код вызовет исключение. Затем вызывающий код может воспользоваться оператором `try`, чтобы отобразить сообщение об ошибке.

Используйте свойства вместо полей, если хотите проверить, какое значение может быть сохранено, привязать данные в XAML (эту тему мы рассмотрим в главе 19), а также если хотите читать из поля и записывать в него, не используя методы, такие как GetAge и SetAge.

Использование модификатора required при определении свойств во время создания экземпляра

В C# 10 введен модификатор `required`. Если вы используете его для определения свойства, то компилятор гарантирует, что при его создании вы присвоите свойству значение:

```
public class Book
{
  public required string Isbn { get; set; }
  public string Title { get; set; }
}
```

Если вы попытаетесь создать экземпляр `Book` без определения свойства `Isbn`, вы получите ошибку компилятора:

```
Book novel = new();
```

Ключевое слово required может быть не включено в окончательную версию .NET 6, поэтому отнеситесь к данному подразделу как к теории.

Определение индексаторов

Индексаторы позволяют вызывающему коду использовать синтаксис массива, чтобы получить доступ к свойству. Например, тип `string` определяет *индексатор*, поэтому вызывающий код может обращаться к отдельным символам в `string` по отдельности.

Мы определим индексатор, чтобы упростить доступ к информации о детях человека.

1. В файле `PersonAutoGen.cs` добавьте следующие операторы, чтобы определить индексатор, получающий и устанавливающий дочерний элемент на основе индекса (позиции) ребенка:

```
// индексаторы
public Person this[int index]
{
  get
  {
    return Children[index]; // передача индексатору List<T>
  }
  set
  {
    Children[index] = value;
  }
}
```

   Вы можете перегружать индексаторы так, чтобы можно было использовать разные типы в качестве их параметров. Например, так же, как вы передаете значение типа `int`, вы можете передавать и значение типа `string`.

2. В файле `Program.cs` введите следующие операторы, чтобы добавить к `Sam` два дочерних элемента и получить к ним доступ, используя более длинное поле `Children` и укороченный синтаксис индексатора:

```
sam.Children.Add(new() { Name = "Charlie" });
sam.Children.Add(new() { 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}");
```

3. Запустите код и проанализируйте результат:

```
Sam's first child is Charlie
Sam's second child is Ella
Sam's first child is Charlie
Sam's second child is Ella
```

Сопоставление с образцом с помощью объектов

В главе 3 вы познакомились с основами сопоставления с образцом. В следующем разделе мы рассмотрим данную тему более подробно.

Создание и работа с библиотеками классов .NET 6

Расширенные функции сопоставления с образцом доступны только в библиотеках классов .NET, поддерживающих C# 9 или более поздние версии.

1. Откройте редактор кода и создайте библиотеку класса `PacktLibraryModern` в рабочей области/решении `Chapter05`.
2. В проекте `PeopleApp` добавьте ссылку на библиотеку класса `PacktLibraryModern`:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="../PacktLibrary/PacktLibrary.csproj" />
    <ProjectReference
      Include="../PacktLibraryModern/PacktLibraryModern.csproj" />
  </ItemGroup>
</Project>
```

3. Соберите проект `PeopleApp`.

Определение пассажиров рейса

В следующем примере мы определим некоторые классы, которые представляют различные типы пассажиров, а затем для определения стоимости полета будем использовать выражение переключения с сопоставлением с образцом.

1. В проекте/папке `PacktLibraryModern` переименуйте файл `Class1.cs` в `FlightPatterns.cs`.
2. В файле `FlightPatterns.cs` добавьте операторы для определения трех типов пассажиров с разными свойствами:

```
namespace Packt.Shared; // пространство имен C# 10 с файловой областью

public class BusinessClassPassenger
{
```

```
  public override string ToString()
  {
    return $"Business Class";
  }
}

public class FirstClassPassenger
{
  public int AirMiles { get; set; }

  public override string ToString()
  {
    return $"First Class with {AirMiles:N0} air miles";
  }
}

public class CoachClassPassenger
{
  public double CarryOnKG { get; set; }

  public override string ToString()
  {
    return $"Coach Class with {CarryOnKG:N2} KG carry on";
  }
}
```

3. В файле `Program.cs` добавьте операторы для определения массива объектов, содержащего пять пассажиров различных типов и значений свойств. Затем перечислите их, выводя стоимость их полета:

```
object[] passengers = {
  new FirstClassPassenger { AirMiles = 1_419 },
  new FirstClassPassenger { AirMiles = 16_562 },
  new BusinessClassPassenger(),
  new CoachClassPassenger { CarryOnKG = 25.7 },
  new CoachClassPassenger { CarryOnKG = 0 },
};

foreach (object passenger in passengers)
{
  decimal flightCost = passenger switch
  {
    FirstClassPassenger p when p.AirMiles > 35000 => 1500M,
    FirstClassPassenger p when p.AirMiles > 15000 => 1750M,
    FirstClassPassenger _                         => 2000M,
    BusinessClassPassenger _                      => 1000M,
    CoachClassPassenger p when p.CarryOnKG < 10.0 => 500M,
    CoachClassPassenger _                         => 650M,
    _                                             => 800M
  };

  WriteLine($"Flight costs {flightCost:C} for {passenger}");
}
```

При анализе этого кода обратите внимание на следующие моменты:

- для сопоставления с образцом свойств объекта вы должны назвать локальную переменную, которую затем можно будет использовать в выражении, таком как p;
- чтобы сопоставить образец только с типом, вы можете использовать _, чтобы исключить локальную переменную;
- выражение switch также использует _ для обозначения своей ветви по умолчанию.

4. Запустите код и проанализируйте результат.

```
Flight costs £2,000.00 for First Class with 1,419 air miles
Flight costs £1,750.00 for First Class with 16,562 air miles
Flight costs £1,000.00 for Business Class
Flight costs £650.00 for Coach Class with 25.70 KG carry on
Flight costs £500.00 for Coach Class with 0.00 KG carry on
```

Изменения сопоставления с образцом в C# 9 или более поздних версиях

Предыдущие примеры работали с версией C# 8. Теперь мы рассмотрим некоторые изменения в C# 9 и более поздних версиях. Для начала, вам больше не нужно использовать символ подчеркивания для исключения при сопоставлении типов.

1. В файле Program.cs закомментируйте синтаксис C# 8 и добавьте синтаксис C# 9 и более поздних версий, чтобы изменить ветви для пассажиров первого класса, что позволит использовать вложенное выражение переключения и новую поддержку условных выражений, таких как >:

```
decimal flightCost = passenger switch
{
  /* Синтаксис C# 8
  FirstClassPassenger p when p.AirMiles > 35000 => 1500M,
FirstClassPassenger p when p.AirMiles > 15000   => 1750M,
FirstClassPassenger                             => 2000M, */

  // Синтаксис C# 9 или более поздних версий
  FirstClassPassenger p => p.AirMiles switch
  {
    > 35000 => 1500M,
    > 15000 => 1750M,
    _       => 2000M
  },
```

```
    BusinessClassPassenger                        => 1000M,
    CoachClassPassenger p when p.CarryOnKG < 10.0 => 500M,
    CoachClassPassenger                           => 650M,
    _                                             => 800M
  };
```

2. Запустите код, проанализируйте результат и обратите внимание, что он такой же, как раньше.

Вы также можете использовать реляционный шаблон в сочетании с образцом свойства, чтобы избежать вложенного выражения переключения, как показано ниже:

```
FirstClassPassenger { AirMiles: > 35000 } => 1500,
FirstClassPassenger { AirMiles: > 15000 } => 1750M,
FirstClassPassenger => 2000M,
```

Работа с записями

Прежде чем начать подробно изучать новую функцию языка записей в C# 9 и более поздних версиях, мы рассмотрим некоторые другие связанные новые функции.

Свойства только для инициализации

В этой главе использован синтаксис инициализации объекта для создания экземпляров объектов и установки начальных свойств. Эти свойства также можно изменить после того, как экземпляр уже создан.

Иногда будет возникать необходимость рассматривать свойства как поля `readonly`, чтобы их можно было установить во время создания экземпляра, но не после него. Новое ключевое слово `init` позволяет это сделать. Его можно использовать вместо ключевого слова `set`.

1. В проекте/папке `PacktLibraryModern` добавьте новый файл `Records.cs`.
2. В файле `Records.cs` определите неизменяемый класс для пассажира:

```
namespace Packt.Shared; // пространство имен C# 10 с файловой областью

public class ImmutablePerson
{
  public string? FirstName { get; init; }
  public string? LastName { get; init; }
}
```

3. В файле `Program.cs` добавьте операторы для создания экземпляра нового неизменяемого пассажира, а затем попробуйте изменить одно из его свойств:

```
ImmutablePerson jeff = new()
{
  FirstName = "Jeff",
  LastName = "Winger"
};

jeff.FirstName = "Geoff";
```

4. Скомпилируйте консольное приложение и обратите внимание на ошибку компиляции:

```
Program.cs(254,7): error CS8852: Init-only property or indexer
'ImmutablePerson.FirstName' can only be assigned in an object initializer,
or on 'this' or 'base' in an instance constructor or an 'init' accessor.
[/Users/markjprice/Code/Chapter05/PeopleApp/PeopleApp.csproj]
```

5. Закомментируйте попытку установить свойство `FirstName` после создания экземпляра.

Записи

Свойства, доступные только для инициализации, обеспечивают некоторую неизменность C#. Вы можете развить эту концепцию с помощью *записей*. Они определяются с помощью ключевого слова `record` вместо ключевого слова `class`. Это делает весь объект неизменным, поэтому он работает как значение при сравнении. Более подробно сравнение классов, записей и типов значений мы обсудим в главе 6.

Записи не должны содержать состояние (свойства и поля), которое изменяется после того, как экземпляр создан. Вместо этого вы создаете новые записи из существующих с любым измененным состоянием. Это называется недеструктивной мутацией. Для этого в языке C# 9 введено ключевое слово `with`.

1. В файле `Records.cs` добавьте запись `ImmutableVehicle`:

```
public record ImmutableVehicle
{
  public int Wheels { get; init; }
  public string? Color { get; init; }
  public string? Brand { get; init; }
}
```

2. В файле `Program.cs` добавьте операторы для создания `car`, а затем его измененной копии:

```
ImmutableVehicle car = new()
{
  Brand = "Mazda MX-5 RF",
  Color = "Soul Red Crystal Metallic",
  Wheels = 4
};

ImmutableVehicle repaintedCar = car
  with { Color = "Polymetal Grey Metallic" };

WriteLine($"Original car color was {car.Color}.");
WriteLine($"New car color is {repaintedCar.Color}.");
```

3. Запустите код, проанализируйте результат и обратите внимание на изменение цвета автомобиля в измененной копии:

```
Original car color was Soul Red Crystal Metallic.
New car color is Polymetal Grey Metallic.
```

Позиционные элементы данных в записях

Используя позиционные элементы данных, можно значительно упростить синтаксис для определения записи.

Упрощение членов данных в записях

Вместо того чтобы использовать синтаксис инициализации объекта с фигурными скобками иногда вы можете предоставить конструктор с позиционными параметрами, как вы видели ранее в этой главе. Вы также можете комбинировать его с деконструктором для разделения объекта на отдельные части:

```
public record ImmutableAnimal
{
  public string Name { get; init; }
  public string Species { get; init; }
  public ImmutableAnimal(string name, string species)
  {
    Name = name;
    Species = species;
  }

  public void Deconstruct(out string name, out string species)
  {
    name = Name;
    species = Species;
  }
}
```

Для вас могут быть сгенерированы свойства, конструктор и деструктор:

1. В файле `Records.cs` добавьте операторы для определения другой записи с помощью синтаксиса позиционирования записей:

```
// более простой способ определить запись
// автоматически генерирует свойства, конструктор и деконструктор
public record ImmutableAnimal(string Name, string Species);
```

2. В файле `Program.cs` добавьте операторы для создания и деконструкции изменяемых животных:

```
ImmutableAnimal oscar = new("Oscar", "Labrador");
var (who, what) = oscar; // calls Deconstruct method
WriteLine($"{who} is a {what}.");
```

3. Запустите код и проанализируйте результат:

```
Oscar is a Labrador.
```

В главе 6 вы снова столкнетесь с записями, когда мы будем изучать поддержку C# 10 для создания записей struct.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 5.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какие шесть модификаторов доступа вы знаете и для чего они используются?
2. Чем различаются ключевые слова `static`, `const` и `readonly` при их применении к члену типа?
3. Для чего используется конструктор?
4. Зачем применять атрибут `[Flags]` к типу `enum`, когда нужно хранить комбинированные значения?
5. В чем польза ключевого слова `partial`?
6. Что вы знаете о кортежах?
7. Для чего служит ключевое слово `record`?

8. Что такое перегрузка?
9. В чем разница между полем и свойством?
10. Как сделать параметр метода необязательным?

Упражнение 5.2. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-5---building-your-own-types-with-object-oriented-programming, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали, как создавать пользовательские типы с помощью объектно-ориентированного программирования (ООП), а также о ряде категорий членов и типов, включая поля для хранения данных и методы для выполнения действий. Вы использовали концепции ООП, такие как агрегирование и инкапсуляция. Вы увидели примеры использования функций современной C#, таких как усовершенствованное сопоставление с образцом, свойства, доступные только для инициализации, и записи.

В следующей главе вы углубитесь в эти концепции, определяя делегаты и события, реализуя интерфейсы и наследуя существующие классы.

6 Реализация интерфейсов и наследование классов

Данная глава посвящена тому, как создавать новые типы на основе существующих, используя *объектно-ориентированное программирование (ООП)*. Вы узнаете о том, как определять операции и локальные функции для выполнения простых действий, определять делегаты и события для обмена сообщениями между типами, как реализовывать интерфейсы для общей функциональности, а также о дженериках. Кроме того, мы обсудим, чем различаются ссылочные типы и типы значений, как создавать новые классы, наследуя их от базовых для многократного применения функциональности, как переопределять унаследованный член типа, использовать полиморфизм. Наконец, мы поговорим о том, как создавать методы расширения и выполнять приведение классов в иерархии наследования.

В этой главе:

- настройка библиотеки классов и консольного приложения;
- подробнее о методах;
- подъем и обработка событий;
- оптимизация типов под многократное использование с помощью дженериков;
- реализация интерфейсов;
- управление памятью с помощью ссылочных типов и типов значений;
- работа со значениями `null`;
- наследование классов;
- приведение в иерархиях наследования;
- наследование и расширение типов .NET.
- использование анализатора для написания лучшего кода.

Настройка библиотеки классов и консольного приложения

Мы начнем с определения рабочей области/решения с двумя проектами, подобным созданному в главе 5. Даже если вы выполнили все упражнения в той главе, следуйте инструкциям ниже, так как в библиотеке классов мы будем использовать функции C# 10, поэтому необходимо ориентироваться на .NET 6.0, а не на .NET Standard 2.0.

1. Откройте редактор кода и создайте рабочую область/решение `Chapter06`.
2. Создайте проект библиотеки класса с такими настройками:
   1) шаблон проекта: Class Library/`classlib`;
   2) файл и папка рабочей области/решения: `Chapter06`;
   3) файл и папка проекта: `PacktLibrary`.
3. Создайте консольное приложение с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter06`;
   3) файл и папка проекта: `PeopleApp`.
4. В проекте `PacktLibrary` переименуйте файл `Class1.cs` в `Person.cs`.
5. Измените код файла `Person.cs`:

```
using static System.Console;

namespace Packt.Shared;

public class Person : object
{
  // поля
  public string? Name; // символ ? допускает нулевое значение
  public DateTime DateOfBirth;
  public List<Person> Children = new(); // C# 9 и более поздние версии

  // методы
  public void WriteToConsole()
  {
    WriteLine($"{Name} was born on a {DateOfBirth:dddd}.");
  }
}
```

6. В проекте PeopleApp добавьте ссылку на проект PacktLibrary:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference
      Include="..\PacktLibrary\PacktLibrary.csproj" />
  </ItemGroup>

</Project>
```

7. Соберите проект PeopleApp и обратите внимание на сообщение, что оба проекта успешно собраны.

Дополнительные сведения о методах

Сделаем так, чтобы два экземпляра класса Person могли «производить потомство». Мы можем реализовать это, написав методы. Методы экземпляра — действия, которые объект выполняет к себе; статические методы — действия, которые выполняет тип.

Ваш выбор зависит от того, что больше всего подходит для действия.

Имеет смысл использовать статические методы и методы экземпляров для выполнения аналогичных действий. Например, тип string имеет как статический метод Compare, так и метод экземпляра CompareTo. Это позволяет программистам, применяющим ваш тип, выбрать, как задействовать функциональность, тем самым увеличивая гибкость.

Реализация функциональности с помощью методов

Начнем с реализации некоторых функций, используя как статические методы, так и методы экземпляра.

1. Добавьте один метод экземпляра и один статический метод в класс Person, что позволит двум объектам Person производить потомство:

```
// статический метод для "размножения"
public static Person Procreate(Person p1, Person p2)
```

```
{
  Person baby = new()
  {
    Name = $"Baby of {p1.Name} and {p2.Name}"
  };

  p1.Children.Add(baby);
  p2.Children.Add(baby);

  return baby;
}

// метод экземпляра для "размножения"
public Person ProcreateWith(Person partner)
{
  return Procreate(this, partner);
}
```

Обратите внимание на следующие моменты:

- в статическом методе `static` с именем `Procreate` создаваемые объекты `Person` передаются как параметры с именами `p1` и `p2`;
- новый класс `Person` под названием `baby` создается с именем, составленным из комбинации двух человек, которые произвели потомство. Это можно изменить позже, установив для возвращаемой переменной `baby` свойство `Name`;
- объект `baby` добавляется в коллекцию `Children` обоих родителей, а затем возвращается. Классы — это ссылочные типы, то есть добавляется ссылка на объект `baby`, сохраненный в памяти, но не копия объекта (позже в этой главе вы узнаете разницу между ссылочными типами и типами значений);
- в методе экземпляра `ProcreateWith` объект `Person`, для которого производится потомство, передается в качестве параметра `partner` и вместе с `this` передается статическому методу `Procreate` для повторного использования реализации метода; `this` — это ключевое слово, которое ссылается на текущий экземпляр класса.

Метод, который создает новый объект или изменяет существующий, должен возвращать ссылку на него, чтобы вызывающая сторона могла получить доступ к результатам.

2. В проекте `PeopleApp` в начале файла `Program.cs` удалите комментарий и импортируйте пространство имен для нашего класса `Person` и статически импортируйте тип `Console`:

```
using Packt.Shared;
using static System.Console;
```

3. В файле `Program.cs` создайте три объекта, обозначающие людей, у которых появятся дети. Обратите внимание: чтобы добавить символ двойной кавычки в строку, необходимо поставить перед ним символ обратной косой черты с кавычкой, `\"`:

```
Person harry = new() { Name = "Harry" };
Person mary = new() { Name = "Mary" };
Person jill = new() { Name = "Jill" };

// вызов метода экземпляра
Person baby1 = mary.ProcreateWith(harry);
baby1.Name = "Gary";

// вызов статического метода
Person baby2 = Person.Procreate(harry, jill);

WriteLine($"{harry.Name} has {harry.Children.Count} children.");
WriteLine($"{mary.Name} has {mary.Children.Count} children.");
WriteLine($"{jill.Name} has {jill.Children.Count} children.");
WriteLine(
  format: "{0}'s first child is named \"{1}\".",
  arg0: harry.Name,
  arg1: harry.Children[0].Name);
```

4. Запустите код и проанализируйте результат:

```
Harry has 2 children.
Mary has 1 children.
Jill has 1 children.
Harry's first child is named "Gary".
```

Реализация функциональности с помощью операций

Класс `System.String` имеет статический метод `Concat`, который объединяет два строковых значения и возвращает результат:

```
string s1 = "Hello ";
string s2 = "World!";
string s3 = string.Concat(s1, s2);
WriteLine(s3); // Hello World!
```

Вызов такого метода, как `Concat`, работает, но программисту может быть привычнее использовать символ операции `+` для сложения двух строковых значений, как показано ниже:

```
string s3 = s1 + s2;
```

Хорошо известна библейская фраза «Плодитесь и размножайтесь», то есть «Производите потомство». Напишем код так, чтобы символ `*` (умножение) позволял «размножаться» двум объектам `Person`.

Мы делаем это путем определения операции `static` для символа *. Синтаксис скорее похож на метод, поскольку, по сути, операция — это метод, но вместо имени метода используется символ, что сокращает синтаксис.

1. В файле `Person.cs` создайте операцию `static` для символа *:

```
// операция "размножения"
public static Person operator *(Person p1, Person p2)
{
  return Person.Procreate(p1, p2);
}
```

В отличие от методов, операции не отображаются в списках IntelliSense для типа. Для каждой операции, которую вы определяете, также создайте метод, поскольку программисту может быть неочевидно, что операция доступна. Реализация операции может затем вызвать метод, повторно используя написанный вами код. Вторая причина предоставления метода заключается в том, что операции поддерживаются не каждым языковым компилятором. Например, хотя арифметические операции, такие как *, поддерживаются в Visual Basic и F#, нет требования, чтобы другие языки поддерживали все операции, поддерживаемые в языке C#.

2. В файле `Program.cs` после вызова метода `Procreate` и перед операторами, которые записывают в консоль, используйте операцию *, чтобы создать еще один объект `baby`:

```
// вызов статического метода
Person baby2 = Person.Procreate(harry, jill);

// вызов операции
Person baby3 = harry * mary;
```

3. Запустите код и проанализируйте результат:

```
Harry has 3 children.
Mary has 2 children.
Jill has 1 children.
Harry's first child is named "Gary".
```

Реализация функциональности с помощью локальных функций

К новым возможностям языка C# 7.0 относится определение *локальных функций*.

Локальные функции эквивалентны локальным переменным. Другими словами, это методы, доступные только внутри содержащего их метода, в котором они были определены. В других языках программирования такие функции иногда называются *вложенными* или *внутренними*.

Локальные функции могут быть определены где угодно внутри метода: в начале, в конце или даже где-то посередине!

Мы воспользуемся локальной функцией для реализации вычисления факториала.

1. В файле `Person.cs` добавьте операторы для определения функции `Factorial`, которая использует локальную функцию внутри себя, чтобы вычислять результат:

```
// 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)// локальная функция
  {
    if (localNumber < 1) return 1;
    return localNumber * localFactorial(localNumber - 1);
  }
}
```

2. В файле `Program.cs` добавьте оператор для вызова функции `Factorial` и запишите возвращаемое значение в консоль:

```
WriteLine($"5! is {Person.Factorial(5)}");
```

3. Запустите код и проанализируйте результат:

```
5! is 120
```

Подъем и обработка событий

Методы часто описываются как *действия, которые объект может выполнять либо над собой, либо над связанными объектами*. К примеру, класс `List<T>` может добавить в себя элемент или очистить сам себя, а `File` может создавать или удалять файл в файловой системе.

События часто описываются как *действия, которые происходят с объектом*. Например, в пользовательском интерфейсе `Button` есть событие `Click`, определяющее, что происходит с кнопкой при щелчке на ней. `FileSystemWatcher` прослушивает файловую систему на предмет уведомлений об изменениях и генерирует такие события, как `Created` и `Deleted`, которые запускаются при изменении папки или файла.

События можно представить еще и как способ обмена сообщениями между двумя объектами.

События построены на *делегатах*, поэтому начнем с рассмотрения того, как работают делегаты.

Вызов методов с помощью делегатов

Вы уже знакомы с наиболее распространенным способом вызова или выполнения метода: он заключается в использовании операции `.` для доступа к методу по его имени. Например, метод `Console.WriteLine` дает типу `Console` указание получить доступ к методу `WriteLine`.

Другой способ вызова или выполнения метода заключается в применении *делегата*. 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 делегат, чтобы его можно было безопасно вызвать с помощью правильных типов параметров.

Для примера представьте, что в классе `Person` есть метод, который должен иметь тип `string`, передаваемый как единственный параметр, и возвращать `int`:

```
public int MethodIWantToCall(string input)
{
  return input.Length; // не имеет значения, что делает этот метод
}
```

Я мог бы вызвать этот метод на экземпляре класса `Person` с именем `p1` следующим образом:

```
int answer = p1.MethodIWantToCall("Frog");
```

Как вариант, я мог бы определить делегат с совпадающей сигнатурой для косвенного вызова метода. Обратите внимание: имена параметров могут не совпадать. Должны совпадать только типы параметров и возвращаемые значения, как показано в коде ниже:

```
delegate int DelegateWithMatchingSignature(string s);
```

Теперь я могу создать экземпляр делегата, указать им на метод и, наконец, вызвать делегат (который вызывает метод!).

```
// создайте экземпляр делегата, который указывает на метод
DelegateWithMatchingSignature d = new(p1.MethodIWantToCall);

// вызов делегата, который вызывает метод
int answer2 = d("Frog");
```

Вероятно, вы задались вопросом: «А в чем смысл?» Отвечу лаконично: в гибкости.

Например, мы могли бы использовать делегаты для создания очереди методов, которые нужно вызвать по порядку. Действия в очереди, которые необходимо выполнить, распространены в сервисах, призванных обеспечивать улучшенную масштабируемость.

Другой пример — разрешить параллельное выполнение нескольких действий. Делегаты имеют встроенную поддержку асинхронных операций, которые выполняются в другом потоке, что может повысить скорость отклика. О том, как это сделать, вы узнаете в главе 12.

Самый важный пример: делегаты позволяют реализовывать события для отправки сообщений между различными объектами, которые не должны знать друг о друге. События — это пример слабой связи между компонентами, так как компонентам не обязательно знать друг о друге, им просто необходимо знать сигнатуру события.

Делегаты и события — одни из самых продвинутых функций языка C#, и на то, чтобы овладеть ими, может потребоваться некоторое время, поэтому не беспокойтесь, если сразу не поняли принцип их работы!

Определение и обработка делегатов

Microsoft предоставляет два предопределенных делегата для использования в качестве событий. Их сигнатуры просты, но обладают гибкостью:

```
public delegate void EventHandler(
  object? sender, EventArgs e);

public delegate void EventHandler<TEventArgs>(
  object? sender, TEventArgs e);
```

Если хотите определить событие в собственных типах, то вам следует использовать один из этих двух предопределенных делегатов.

Рассмотрим пример.

1. Добавьте операторы в класс `Person` и обратите внимание на следующие моменты:
   - код определяет поле делегата `EventHandler` с именем `Shout`;
   - код определяет поле `int` для хранения `AngerLevel`;
   - код определяет метод `Poke`;

- каждый раз, когда человека толкают (`Poke`), уровень его раздражения (`AngerLevel`) растет. Когда он 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`:

```
// поле делегата
public EventHandler? Shout;

// поле данных
public int AngerLevel;

// метод
public void Poke()
{
  AngerLevel++;

  if (AngerLevel >= 3)
  {
    // если что-то прослушивается...
    if (Shout != null)
    {
      // ...затем вызовите делегат
      Shout(this, EventArgs.Empty);
    }
  }
}
```

Проверка того, является ли объект `null` до вызова одного из его методов, выполняется очень часто. C# 6.0 и более поздние версии позволяют упростить проверки на `null`, используя символ `?` перед операцией `.` (точка):

```
Shout?.Invoke(this, EventArgs.Empty);
```

2. В конце файла `Program.cs` добавьте метод с соответствующей сигнатурой, который получает ссылку на объект `Person` из параметра `sender` и выводит некую информацию о них:

```
static void Harry_Shout(object? sender, EventArgs e)
{
  if (sender is null) return;
  Person p = (Person)sender;
  WriteLine($"{p.Name} is this angry: {p.AngerLevel}.");
}
```

Соглашение Microsoft для имен методов, которые обрабатывают события, называется `ObjectName_EventName`.

3. В файле `Program.cs` добавьте оператор для назначения метода полю делегата:

```
harry.Shout = Harry_Shout;
```

4. Добавьте операторы для четырехкратного вызова метода `Poke` после того, как будет назначен метод для события `Shout`:

```
harry.Shout = Harry_Shout;
harry.Poke();
harry.Poke();
harry.Poke();
harry.Poke();
```

5. Запустите код и проанализируйте результат. Обратите внимание, что, когда Гарри толкают первые два раза, он молчит, а раздражается и начинает кричать лишь после того, как его толкнули не менее трех раз:

```
Harry is this angry: 3.
Harry is this angry: 4.
```

Определение и обработка событий

Теперь вы увидели, как делегаты реализуют наиболее важную функциональность событий: возможность определить сигнатуру для метода, который может быть реализован совершенно другим фрагментом кода, а затем вызвать этот метод и любые другие, подключенные к полю делегатов.

Но как насчет событий? Их меньше, чем вы думаете.

При назначении метода для поля делегата не следует использовать простую операцию присваивания, как мы делали в предыдущем примере.

Делегаты — многоадресные; это значит, что вы можете назначить несколько делегатов одному полю делегата. Вместо оператора присваивания `=` мы могли бы использовать операцию `+=`, чтобы добавить больше методов к тому же полю делегата. При вызове делегата вызываются все назначенные методы, хотя вы не можете контролировать порядок их вызова.

Если поле делегата `Shout` уже ссылается на один или несколько методов, то назначение нового метода перекроет все остальные. С помощью делегатов, которые используются для событий, мы обычно хотим убедиться, что программист применяет либо операцию `+=`, либо операцию `-=` для назначения и удаления методов.

1. Чтобы осуществить это, в файле `Program.cs` добавьте ключевое слово `event` в объявление поля делегата:

```
public event EventHandler? Shout;
```

2. Соберите проект `PeopleApp` и обратите внимание на сообщение компилятора об ошибке:

```
Program.cs(41,13): error CS0079: The event 'Person.Shout' can only appear
on the left hand side of += or -=
```

   Это (почти) все, что делает ключевое слово `event`! Если у вас никогда не будет нескольких методов, назначенных для поля делегата, то вам не нужны «события». Однако я по-прежнему рекомендую указывать ваше значение и то, что вы ожидаете, что поле делегата будет использоваться как событие.

3. Измените присвоение метода, чтобы использовать операцию `+=`:

```
harry.Shout += Harry_Shout;
```

4. Запустите код и обратите внимание, что он работает, как раньше.

Обеспечение безопасности многократно используемых типов с помощью дженериков

В 2005 году в версии C# 2.0 и .NET Framework 2.0 компания Microsoft представила функцию под названием *«дженерики»* (или «обобщения»), которая позволяет повысить безопасность типов при их повторном использовании и увеличить их эффективность. Она дает программисту возможность передавать типы в качестве параметров, аналогично тому как вы можете передавать объекты.

Работа с типами, не являющимися дженериками

Сначала рассмотрим пример работы с типом, не являющимся дженериком, чтобы вы могли понять проблему, для решения которой были созданы дженерики, такие как слабо типизированные параметры и значения, а также проблемы с производительностью, вызванные использованием `System.Object`.

`System.Collections.Hashtable` можно использовать для хранения нескольких значений, каждое из которых имеет уникальный ключ, который впоследствии можно использовать для поиска его значения. Ключ и значение могут быть любыми объектами, так как они объявлены как `System.Object`. Хотя при этом упрощается сохранение типов значений, таких как целые цисла, сам процесс происходит медленно и легче сделать ошибки, ведь при добавлении элементов проверки типов не выполняются.

Рассмотрим пример.

1. В файле `Program.cs` создайте экземпляр не дженерик-коллекции `System.Collections.Hashtable`, а затем добавьте в нее четыре элемента:

```
// не дженерик-коллекция поиска
System.Collections.Hashtable lookupObject = new();

lookupObject.Add(key: 1, value: "Alpha");
lookupObject.Add(key: 2, value: "Beta");
lookupObject.Add(key: 3, value: "Gamma");
lookupObject.Add(key: harry, value: "Delta");
```

2. Добавьте операторы для определения ключа со значением `2` и используйте его для поиска значения в хеш-таблице:

```
int key = 2; // поиск значения, содержащего 2 в качестве ключа
WriteLine(format: "Key {0} has value: {1}",
  arg0: key,
  arg1: lookupObject[key]);
```

3. Добавьте следующие операторы, чтобы использовать объект `harry` для поиска его значения:

```
// поиск значения, содержащего harry в качестве ключа
WriteLine(format: "Key {0} has value: {1}",
  arg0: harry,
  arg1: lookupObject[harry]);
```

4. Запустите код и проанализируйте результат:

```
Key 2 has value: Beta
Key Packt.Shared.Person has value: Delta
```

Код работает, однако всегда существует вероятность ошибок, так как для ключа или значения можно использовать абсолютно любой тип. Если другой разработчик использовал ваш объект поиска и ожидал, что все элементы будут определенного типа, он может привести их к этому типу и получить исключения, поскольку некоторые значения могут быть другого типа. Объект поиска с большим количеством элементов также приведет к низкой производительности.

Избегайте использования типов в пространстве имен System.Collections.

Работа с типами-дженериками

Запись `System.Collections.Generic.Dictionary<TKey, Tvalue>` можно использовать для хранения нескольких значений, каждое из которых имеет уникальный ключ, который впоследствии можно использовать для быстрого поиска его значения.

И ключ, и значение могут быть любыми объектами. Но вы должны указать компилятору, какие типы ключа и значения будут использоваться при первом создании экземпляра коллекции. Для этого необходимо указать типы для *параметров-дженериков* в угловых скобках (`<>`), `Tkey` и `Tvalue`.

Если тип-дженерик имеет один определяемый тип, то он должен называться T, например List<T>, где T — тип, хранящийся в списке. Когда тип-дженерик имеет несколько определяемых типов, необходимо использовать T в качестве префикса в имени и присвоить понятное имя, например Dictionary<Tkey, Tvalue>.

Это обеспечивает гибкость, скорость и легкость в избежании ошибок, поскольку при добавлении элементов выполняются проверки типов.

Напишем код с помощью дженериков.

1. В файле `Program.cs` создайте экземпляр дженерик-коллекции поиска `Dictionary<Tkey, Tvalue>`, а затем добавьте в него четыре элемента:

```
// дженерик-коллекция поиска
Dictionary<int, string> lookupIntString = new();

lookupIntString.Add(key: 1, value: "Alpha");
lookupIntString.Add(key: 2, value: "Beta");
lookupIntString.Add(key: 3, value: "Gamma");
lookupIntString.Add(key: harry, value: "Delta");
```

2. Обратите внимание на ошибку компиляции при использовании объекта `harry` в качестве ключа:

```
/Users/markjprice/Code/Chapter06/PeopleApp/Program.cs(98,32):
error CS1503: Argument 1: cannot convert from 'Packt.Shared.Person'
to 'int' [/ Users/markjprice/Code/Chapter06/PeopleApp
/PeopleApp.csproj]
```

3. Замените `harry` на `4`.
4. Добавьте операторы, чтобы установить ключ в значение `3`, и используйте его для поиска значения в словаре:

```
key = 3;
WriteLine(format: "Key {0} has value: {1}",
  arg0: key,
  arg1: lookupIntString[key]);
```

5. Запустите код и обратите внимание, что он работает:

```
Key 3 has value: Gamma
```

Реализация интерфейсов

С помощью интерфейсов можно объединить разные типы, чтобы создать новые элементы. В качестве примеров интерфейсов можно привести выступы на деталях конструктора «Лего», которые позволяют им соединяться, или стандарты для электрических вилок и розеток.

Если тип реализует интерфейс, то гарантирует остальной части .NET, что поддерживает определенную функцию. Вот почему интерфейсы иногда называют контрактами.

Универсальные интерфейсы

В табл. 6.1 представлено несколько универсальных интерфейсов, которые могут реализовать ваши типы.

Таблица 6.1. Универсальные интерфейсы

Интерфейс	Метод (-ы)	Описание
`IComparable`	`CompareTo(other)`	Определяет метод сравнения, который тип реализует для упорядочения или сортировки экземпляров
`IComparer`	`Compare(first, second)`	Определяет метод сравнения, который вторичный тип реализует для упорядочения или сортировки экземпляров первичного типа
`IDisposable`	`Dispose()`	Определяет метод, позволяющий освобождать неуправляемые ресурсы более эффективно, чем ожидание финализатора (дополнительные сведения см. в подразделе «Освобождение неуправляемых ресурсов» далее в этой главе)
`IFormattable`	`ToString(format, culture)`	Определяет метод, поддерживающий региональные параметры, для форматирования значения объекта в строковое представление
`IFormatter`	`Serialize(stream, object)` и `Deserialize(stream)`	Определяет методы преобразования объекта в поток байтов и из него для хранения или передачи
`IFormatProvider`	`GetFormat(type)`	Определяет метод для форматирования ввода на основе настроек языка и региона

Сравнение объектов при сортировке

Один из наиболее распространенных интерфейсов, который вы можете реализовать, — это `IComparable`. Он содержит метод `CompareTo`. Существует два варианта работы метода: первый — с типом `object`, допускающим значение `NULL`, а другой — с типом-дженериком, допускающим значение `NULL`:

```
namespace System
{
  public interface IComparable
  {
    int CompareTo(object? obj);
  }

  public interface IComparable<in T>
  {
    int CompareTo(T? other);
  }
}
```

Например, тип `string` реализует `IComparable`, возвращая значение `-1`, если строка меньше сравниваемой, или значение `1`, если больше. Тип `int` реализует `IComparable`, возвращая значение `-1`, если число `int` меньше сравниваемого, или значение `1`, если больше.

Если тип реализует один из интерфейсов `IComparable`, то массивы и коллекции могут его сортировать.

Прежде чем реализовывать интерфейс `IComparable` и его метод `CompareTo` для класса `Person`, посмотрим, что происходит при сортировке массива экземпляров `Person`.

1. В файле `Program.cs` добавьте операторы, которые создают массив экземпляров `Person` и записывают элементы в консоль, затем пытаются его отсортировать и снова записать элементы в консоль:

```
Person[] people =
{
  new() { Name = "Simon" },
  new() { Name = "Jenny" },
  new() { Name = "Adam" },
  new() { Name = "Richard" }
};

WriteLine("Initial list of people:");
foreach (Person p in people)
{
  WriteLine($" {p.Name}");
}
```

```
WriteLine("Use Person's IComparable implementation to sort:");
Array.Sort(people);
foreach (Person p in people)
{
  WriteLine($" {p.Name}");
}
```

2. Запустите код, и будет выброшено исключение. Как следует из текста ошибки, чтобы устранить проблему, наш тип должен реализовать интерфейс `Icomparable`:

```
Unhandled Exception: System.InvalidOperationException: Failed to compare
two elements in the array. ---> System.ArgumentException: At least one
object must implement IComparable.
```

3. В файле `Person.cs` после наследования от объекта добавьте запятую и введите `Icomparable<Person>`, как показано ниже:

```
public class Person : object, IComparable<Person>
```

Ваш редактор кода подчеркнет красным новый код, предупреждая о том, что вы еще не реализовали обещанный метод. Программа может сгенерировать каркас реализации, если вы нажмете кнопку в виде лампочки и выберете в контекстном меню команду Implement interface (Реализовать интерфейс).

4. Прокрутите код вниз, чтобы найти сгенерированный метод, и удалите оператор, выбрасывающий ошибку `NotImplementedException`:

```
public int CompareTo(Person? other)
{
  throw new NotImplementedException();
}
```

5. Добавьте оператор для вызова метода `CompareTo` поля `Name`, в котором используется реализация `CompareTo` для типа `string`, как показано ниже (выделено жирным шрифтом):

```
public int CompareTo(Person? other)
{
  if (Name is null) return 0;
  return Name.CompareTo(other?.Name);
}
```

Мы решили сравнить два экземпляра `Person`, сравнив их поля `Name`. Это значит, `Person` будут сортироваться в алфавитном порядке по имени. Чтобы упростить, во всех этих примерах я не добавлял проверки на `null`.

6. Запустите код и обратите внимание, что на этот раз все работает:

```
Initial list of people:
  Simon
  Jenny
```

```
  Adam
  Richard
Use Person's IComparable implementation to sort:
  Adam
  Jenny
  Richard
  Simon
```

Если вы захотите отсортировать массив или коллекцию экземпляров вашего типа, то реализуйте интерфейс `Icomparable`.

Сравнение объектов с помощью отдельных классов

В некоторых случаях вы можете не иметь доступа к исходному коду типа, и он может не реализовывать интерфейс `Icomparable`. К счастью, есть еще один способ сортировать экземпляры типа: создать вторичный тип, реализующий несколько иной интерфейс под названием `Icomparer`.

1. В проекте `PacktLibrary` добавьте класс `PersonComparer`, содержащий класс, реализующий интерфейс `Icomparer`, который сравнивает двух людей, то есть два экземпляра `Person`. Он сравнивает длину их поля `Name` или, если имена имеют одинаковую длину, сравнивает их в алфавитном порядке:

```
namespace Packt.Shared;

public class PersonComparer : IComparer<Person>
{
  public int Compare(Person? x, Person? y)
  {
    if (x is null || y is null)
    {
      return 0;
    }
    // сравниваем длину имени...
    int result = x.Name.Length.CompareTo(y.Name.Length);

    // ...если равны...
    if (result == 0)
    {
      // ...затем сравнниваем по именам...
      return x.Name.CompareTo(y.Name);
    }
    else // в результате должно получиться -1 или 1
    {
      // ...в противном случае сравниваем по длине
      return result;
    }
  }
}
```

2. В файле `Program.cs` добавьте операторы для сортировки массива, используя эту альтернативную реализацию:

```
WriteLine("Use PersonComparer's IComparer implementation to sort:");
Array.Sort(people, new PersonComparer());
foreach (Person p in people)
{
  WriteLine($" {p.Name}");
}
```

3. Запустите код и проанализируйте результат:

```
Use PersonComparer's IComparer implementation to sort:
  Adam
  Jenny
  Simon
  Richard
```

На сей раз, сортируя массив `people`, мы явно просим алгоритм сортировки использовать тип `PersonComparer`, чтобы сортировка шла сначала по самым коротким именам, например Адам, а затем по самым длинным, например Ричард; а когда длины двух или более имен равны (Дженни и Саймон), они должны быть отсортированы в алфавитном порядке.

Неявные и явные реализации интерфейса

Интерфейсы могут быть реализованы неявно и явно. Неявные реализации более просты и используются чаще. Явные необходимы только в том случае, если тип имеет несколько методов с одинаковым именем и сигнатурой.

Например, как `IGamePlayer`, так и `IkeyHolder` могут иметь метод `Lose` с одинаковыми параметрами, поскольку и игра, и ключ могут быть потеряны. В типе, который должен реализовать оба интерфейса, неявным методом может быть только одна реализация `Lose`. Если оба интерфейса могут использовать одну и ту же реализацию, то это работает. В противном случае другой метод `Lose` должен быть реализован по-другому и вызываться явно:

```
public interface IGamePlayer
{
  void Lose();
}

public interface IKeyHolder
{
  void Lose();
}

public class Person : IGamePlayer, IKeyHolder
{
```

```
  public void Lose() // неявная реализация
  {
    // реализация потери ключа
  }

  void IGamePlayer.Lose() // явная реализация
  {
    // реализация потери игры
  }
}

// вызов неявных и явных реализаций метода Lose
Person p = new();
p.Lose(); // вызов неявной реализации потери ключа

((IGamePlayer)p).Lose(); // вызов явной реализации потери игры

IGamePlayer player = p as IGamePlayer;
player.Lose(); // вызов явной реализации потери игры
```

Определение интерфейсов с реализациями по умолчанию

Функция языка, представленная в C# 8.0, — *реализация интерфейса по умолчанию*.

Рассмотрим пример.

1. В проекте `PacktLibrary` создайте файл `IPlayable.cs`.
2. Измените оператор, чтобы определить общедоступный интерфейс `IPlayable` с двумя методами `Play` и `Pause`, как показано ниже:

```
namespace Packt.Shared;

public interface IPlayable
{
  void Play();
  void Pause();
}
```

3. В проекте `PacktLibrary` создайте файл `DvdPlayer.cs`.
4. Измените операторы в файле, чтобы реализовать интерфейс `IPlayable`:

```
using static System.Console;

namespace Packt.Shared;

public class DvdPlayer : IPlayable
{
  public void Pause()
  {
```

```
    WriteLine("DVD player is pausing.");
  }

  public void Play()
  {
    WriteLine("DVD player is playing.");
  }
}
```

Представленное выше полезно, но что, если мы решим добавить третий метод, `Stop`? До выхода версии C# 8.0 это было бы невозможно в случае, если хотя бы один тип реализует исходный интерфейс. Иными словами, интерфейс — это контракт о том, что какой-то определенный тип обязательно реализует свой функционал.

Версия C# 8.0 позволяет интерфейсу добавлять новые члены после выпуска, если они имеют реализацию по умолчанию. Сторонникам чистого C# не нравится эта идея, но по практическим причинам (например, чтобы избежать критических изменений или определить совершенно новый интерфейс) она полезна, и другие языки, такие как Java и Swift, поддерживают аналогичные приемы.

Поддержка реализаций интерфейса по умолчанию требует некоторых фундаментальных изменений в базовой платформе, поэтому они поддерживаются в C#, только если целевая платформа — .NET 5 или более поздняя версия, .NET Core 3.0 или более поздняя версия, .NET Standard 2.1. Вот почему данная функциональность не поддерживается в .NET Framework.

5. Измените интерфейс `IPlayable`, добавив метод `Stop` с реализацией по умолчанию, как показано ниже:

```
using static System.Console;

namespace Packt.Shared;

public interface IPlayable
{
  void Play();
  void Pause();

  void Stop() // реализация интерфейса по умолчанию
  {
    WriteLine("Default implementation of Stop.");
  }
}
```

6. Соберите проект `PeopleApp` и обратите внимание, что проекты компилируются успешно, несмотря на то что класс `DvdPlayer` не реализует функцию `Stop`. Позднее мы могли бы переопределить реализацию `Stop` по умолчанию, добавив ее в классе `DvdPlayer`.

Управление памятью с помощью ссылочных типов и типов значений

Несколько раз я упоминал ссылочные типы. Рассмотрим их более подробно.

Существует две категории памяти: *стек* и *куча*. В современных операционных системах оба этих объекта могут находиться где угодно в физической или виртуальной памяти.

Стек работает быстрее (поскольку управляется непосредственно процессором и использует механизм «первым вошел — первым вышел», а значит, его данные с большей вероятностью будут храниться в кэш-памяти `L1` или `L2`), но его размер ограничен, в то время как куча более медленная, но увеличивается динамически.

Например, на моем macOS на панели `TERMINAL` (Терминал) я могу ввести команду `ulimit -a` и увидеть, что размер стека ограничен 8192 Кбайт, а другая память «не ограничена». Вот почему так легко получить «переполнение стека».

Определение ссылочных типов и типов значений

Объекты можно создавать с помощью трех ключевых слов C#: `class`, `record` и `struct`. Все они могут иметь одинаковые члены (например, поля и методы). Разница между ними заключается в распределении памяти.

Когда вы определяете тип с помощью слов `record` или `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).

Если вы определяете тип с помощью слов `record struct` или `struct`, то определяете *тип значения*. Это значит, что память для самого объекта выделяется в стеке.

Если в `struct` для полей используются типы, которые сами не являются структурами, то эти поля будут храниться в куче, то есть данные для этого объекта хранятся как в стеке, так и в куче!

К наиболее распространенным структурным типам относятся:

- *типы чисел* `System` — `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`;
- *прочие типы* `System` — `char`, `DateTime` и `bool`;
- *типы* `System.Drawing` — `Color`, `Point` и `Rectangle`.

Почти все остальные типы являются классами, включая `string`.

Помимо разницы в том, где в памяти хранятся данные типа, другое важное отличие состоит в том, что вы не можете наследовать от `struct`.

Хранение в памяти ссылочных типов и типов значений

Представьте, что вы создали консольное приложение, в котором объявлены некоторые переменные:

```
int number1 = 49;
long number2 = 12;
System.Drawing.Point location = new(x: 4, y: 5);
Person kevin = new() { Name = "Kevin",
  DateOfBirth = new(year: 1988, month: 9, day: 23) };
Person sally;
```

Проанализируем, какая память выделяется в стеке и куче при выполнении этих операторов (рис. 6.1).

Рис. 6.1. Распределение значений и ссылочных типов в куче

- Переменная `number1` представляет собой тип значения (также известный как структура), поэтому размещается в стеке и использует 4 байта памяти, поскольку представляет собой 32-битное целое число. Его значение `49` хранится непосредственно в переменной.
- Переменная `number2` также представляет собой тип значения, поэтому также размещается в стеке и использует 8 байт, поскольку является 64-битным целым числом.
- Переменная `location` также представляет собой тип значения, поэтому размещается в стеке и использует 8 байт, поскольку состоит из двух 32-битных целых чисел, `x` и `y`.

- Переменная `kevin` имеет ссылочный тип (также известный как класс), поэтому 8 байт для 64-битного адреса памяти (при условии 64-битной операционной системы) выделяются в стеке, а в куче достаточно байтов для хранения экземпляра `Person`.
- Переменная `sally` имеет ссылочный тип, поэтому в стеке выделяется 8 байт для 64-битного адреса памяти. В данном случае переменная равна нулю, это означает, что для нее еще не выделена память в куче.

Вся выделенная память для ссылочного типа хранится в куче. Если тип значения, такой как `DateTime`, используется для поля ссылочного типа, например, `Person`, то значение `DateTime` будет храниться в куче.

Если тип значения имеет поле, которе является ссылочным типом, то эта часть типа значения будет храниться в куче. `Point` — это тип значения, состоящий из двух полей, оба из которых являются типом значения, поэтому весь объект может быть размещен в стеке. Если бы тип значения `Point` содержал поле ссылочного типа, например, `string`, то байты `string` хранились бы в куче.

Равенство типов

Как правило, две переменные сравнивают с помощью операций `==` и `!=`. Поведение этих двух операций отличается для ссылочных типов и типов значений.

Когда вы проверяете равенство двух переменных типа значения, .NET буквально сравнивает значения этих переменных в стеке и возвращает `true`, если они равны:

```
int a = 3;
int b = 3;
WriteLine($"a == b: {(a == b)}"); // true
```

Когда вы проверяете равенство двух переменных ссылочного типа, .NET сравнивает адреса памяти этих переменных и возвращает значение `true`, если они равны:

```
Person a = new() { Name = "Kevin" };
Person b = new() { Name = "Kevin" };
WriteLine($"a == b: {(a == b)}"); // false
```

Это происходит потому, что они не являются одним и тем же объектом. Если обе переменные четко указывают на один и тот же объект в куче, то они будут равны.

```
Person a = new() { Name = "Kevin" };
Person b = a;
WriteLine($"a == b: {(a == b)}"); // true
```

Единственное исключение — тип `string`. Это ссылочный тип, но операции равенства были переопределены, чтобы заставить их вести себя так, как если бы они были типами значений:

```
string a = "Kevin";
string b = "Kevin";
WriteLine($"a == b: {(a == b)}"); // true
```

Вы можете сделать нечто подобное со своими классами, чтобы операции равенства возвращали значение `true`, даже если они не являются одним и тем же объектом (имеют одинаковый адрес памяти в куче) и если вместо этого их поля содержат одинаковые значения (в данной книге эта тема не рассматривается). В качестве альтернативы используйте `record class`, так как одно из преимуществ заключается в том, что они реализуют это поведение для вас.

Определение типов struct

Рассмотрим пример того, как определять типы значений.

1. В проекте `PacktLibrary` создайте файл `DisplacementVector.cs`.
2. Измените файл, как показано в коде ниже, и обратите внимание на следующие моменты:
   - тип объявляется с помощью ключевого слова `struct` вместо `class`;
   - используются два поля `int`, названные `X` и `Y`;
   - применяется конструктор для установки начальных значений для `X` и `Y`;
   - используется операция для сложения двух экземпляров, которая возвращает новый экземпляр типа с `X`, сложенным с `X`, и `Y`, сложенным с `Y`.

```
namespace Packt.Shared;

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(
        vector1.X + vector2.X,
        vector1.Y + vector2.Y);
      }
    }
```

3. В файле `Program.cs` добавьте операторы для создания двух новых экземпляров `DisplacementVector`, сложите их и выведите результат:

```
DisplacementVector dv1 = new(3, 5);
DisplacementVector dv2 = new(-2, 7);
DisplacementVector dv3 = dv1 + dv2;

WriteLine($"({dv1.X}, {dv1.Y}) + ({dv2.X}, {dv2.Y}) = ({dv3.X},
{dv3.Y})");
```

4. Запустите код и проанализируйте результат:

```
(3, 5) + (-2, 7) = (1, 12)
```

Если все поля вашего типа используют не более 16 байт стековой памяти, в нем в качестве полей применяются только типы значений и вы не планируете наследовать от своего типа, то сотрудники компании Microsoft рекомендуют задействовать тип struct. Если ваш тип использует более 16 байт стековой памяти, или в нем в качестве полей применяются ссылочные типы, или вы планируете наследовать от него, то задействуйте тип class.

Ключевое слово record и тип struct

В C# 10 появилась возможность использовать ключевое слово `record` с типами `struct` и классами.

Мы могли бы определить тип `DisplacementVector` следующим образом:

```
public record struct DisplacementVector(int X, int Y);
```

Ввиду этого изменения Microsoft рекомендует явно указывать `class`, если вам необходимо определить `record class`, хотя ключевое слово `class` является необязательным:

```
public record class ImmutableAnimal(string Name);
```

Освобождение неуправляемых ресурсов

В предыдущей главе вы узнали, что конструкторы могут использоваться для инициализации полей и тип может иметь несколько конструкторов. Представьте, что конструктор выделяет неуправляемый ресурс, то есть нечто неподконтрольное .NET, например файл или мьютекс, находящиеся под управлением операционной системы. Неуправляемый ресурс нужно освободить вручную, поскольку платформа .NET не способна сделать это автоматически, используя функцию автоматической сборки мусора.

В рамках этой темы я продемонстрирую несколько примеров кода, которые не нужно создавать в вашем текущем проекте.

Каждый тип может иметь один *метод завершения (финализатор)*, который будет вызываться средой выполнения .NET при возникновении необходимости освободить ресурсы. Финализатор имеет то же имя, что и конструктор, то есть имя типа, но с префиксом в виде символа тильды (~)

Не путайте финализатор (метод завершения, также иногда называемый *деструктором*) с методом `Deconstruct`. Первый освобождает ресурсы, то есть уничтожает объект. Второй возвращает объект, разбитый на составные части, и использует синтаксис C# для деконструкции, например, для работы с кортежами:

```
public class Animal
{
  public Animal() // конструктор
  {
    // выделяем любые неуправляемые ресурсы
  }

  ~Animal() // финализатор, он же деструктор
  {
    // освобождаем любые неуправляемые ресурсы
  }
}
```

В этом примере кода представлены минимальные действия, которые вы должны совершать при работе с неуправляемыми ресурсами. Проблема с предоставлением только финализатора такова: сборщику мусора .NET потребуется дважды выполнить сборку мусора, чтобы полностью освободить выделенные ресурсы для данного типа.

Хоть это и не обязательно, рекомендуется также предоставить метод, позволяющий разработчику, использующему ваш тип, явно освобождать ресурсы, чтобы сборщик мусора мог немедленно и детерминированно освободить управляемые части неуправляемого ресурса, такой как файл, а затем освободить управляемую часть памяти объекта за одну сборку вместо двух.

Для этого существует стандартный механизм — реализация интерфейса `Idisposable`, как показано в следующем примере:

```
public class Animal : IDisposable
{
  public Animal()
  {
    // выделяем неуправляемый ресурс
  }

  ~Animal() // финализатор
  {
    Dispose(false);
  }

  bool disposed = false; // Ресурсы были освобождены?

  public void Dispose()
  {
    Dispose(true);

    // сообщаем сборщику мусора, что ему не нужно вызывать финализатор
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing)
  {
    if (disposed) return;

    // освобождаем *неуправляемый* ресурс
    // ...

    if (disposing)
    {
      // освобождаем любые другие *управляемые* ресурсы
      // ...
    }
    disposed = true;
  }
}
```

Здесь реализовано два метода `Dispose` — `public` и `protected`.

- Метод `public void Dispose` должен вызываться разработчиком, использующим ваш тип. При вызове необходимо освободить как неуправляемые, так и управляемые ресурсы.
- Метод `protected virtual void Dispose` (с параметром `bool`) используется для реализации удаления ресурсов. Необходимо проверить параметр `disposing` и поле `disposed`: если финализатор уже запущен и вызвал метод `~Animal`, то необходимо освободить только неуправляемые ресурсы.

С помощью вызова метода `GC.SuppressFinalize(this)` можно уведомить сборщик мусора о том, что больше не нужно запускать финализатор, и устранить необходимость во второй сборке.

Обеспечение вызова метода Dispose

Когда используется тип, реализующий интерфейс `IDisposable`, гарантировать вызов открытого метода `Dispose` можно, применив оператор `using`, как показано ниже:

```
using (Animal a = new())
{
  // код, использующий экземпляр Animal
}
```

Компилятор преобразует ваш код в нечто подобное показанному ниже, гарантируя, что, даже если будет вызвано исключение, метод `Dispose` все равно будет вызван:

```
Animal a = new();
try
{
  // код, использующий экземпляр Animal
}
finally
{
  if (a != null) a.Dispose();
}
```

Практические примеры освобождения неуправляемых ресурсов с помощью интерфейса `IDisposable`, операторов `using` и блоков `try ... finally` вы увидите в главе 9.

Работа со значениями null

Теперь вы знаете, как хранить примитивные значения, например числа в переменных `struct`. Но что, если переменная еще не имеет значения? Как мы можем указать это? В языке C# есть концепция значения `null`, которое можно использовать для обозначения того, что переменная не была установлена.

Создание типа, допускающего значение null

По умолчанию типы значений, такие как `int` и `DateTime`, должны всегда иметь значение; отсюда и их название. Иногда, например при считывании значений, хранящихся в базе данных, которая может иметь пустые, отсутствующие или нулевые значения, удобно разрешить типу значения допускать значение `null`. Мы называем

его *типом, допускающим значение null*. Такое разрешение можно дать, добавив к типу вопросительный знак в качестве суффикса при объявлении переменной.

Рассмотрим пример.

1. Откройте редактор кода и создайте консольное приложение `NullHandling` в рабочей области/решении `Chapter06`. Для данного раздела требуется полное приложение с файлом проекта, поэтому вы не сможете использовать блокнот .NET Interactive.
2. В Visual Studio Code выберите `NullHandling` в качестве активного проекта OmniSharp. В Visual Studio выберите `NullHandling` в качестве запускаемого проекта.
3. В файле `Program.cs` введите операторы для объявления и присвоения значений, включая `null`, переменным типа `int`:

```
int thisCannotBeNull = 4;
thisCannotBeNull = null; // ошибка компиляции!

int? thisCouldBeNull = null;
WriteLine(thisCouldBeNull);
WriteLine(thisCouldBeNull.GetValueOrDefault());

thisCouldBeNull = 7;
WriteLine(thisCouldBeNull);
WriteLine(thisCouldBeNull.GetValueOrDefault());
```

4. Закомментируйте оператор, который выдает ошибку компиляции.
5. Запустите код и проанализируйте результат:

```
0
7
7
```

Первая строка пуста, поскольку выводится значение `null`!

Ссылочные типы, допускающие значение null

Использование значения `null` настолько распространено во многих языках, что многие опытные программисты никогда не сомневаются в необходимости его существования. Однако существует множество сценариев, в которых мы могли бы написать лучший, более простой код, если переменная не допускает значение `null`.

Введение ссылочных типов, допускающих и не допускающих значение `null`, — самое существенное изменение в версии C# 8.0. Вы, вероятно, думаете: «Но подождите! Ссылочные типы ведь уже допускают `null`!»

И вы были бы правы, однако в C# 8.0 и более поздних версиях ссылочные типы можно настроить так, чтобы они больше не допускали значение `null`, установив параметр на уровне файла или проекта, чтобы включить эту новую полезную функцию. Поскольку это весьма существенное изменение для языка C#, компания Microsoft решила сделать необходимым явное включение данной функции.

Потребуется несколько лет, чтобы эта новая языковая функция C# оказала воздействие, поскольку тысячи библиотек и приложений все еще будут работать по-старому. Даже Microsoft не успела полностью реализовать эту функцию для всех основных пакетов современной .NET до .NET 5.

При переходе вы можете выбрать один из нескольких подходов для своих проектов:

- *по умолчанию* — никаких изменений не требуется. Ссылочные типы, не допускающие значение `null`, не поддерживаются;
- *включить для проекта, отключить в файлах* — включить функцию на уровне проекта и отключить для всех файлов, которые должны оставаться совместимыми со старыми версиями. К данному подходу Microsoft прибегает внутри собственной компании, когда обновляет пакеты для того, чтобы использовать эту новую функцию;
- *включать в файлах* — включить функцию только для отдельных файлов.

Включение ссылочных типов, допускающих и не допускающих значение null

Чтобы включить эту функцию в проект, добавьте в файл проекта следующий код:

```
<PropertyGroup>
  ...
  <Nullable>enable</Nullable>
</PropertyGroup>
```

Теперь это делается по умолчанию в шаблонах проектов, предназначенных для .NET 6.0.

Чтобы отключить эту функцию в файле, добавьте в начало кода следующую команду:

```
#nullable disable
```

Чтобы включить эту функцию в файле, добавьте в начало кода такую команду:

```
#nullable enable
```

Объявление переменных и параметров, не допускающих значение null

Если вы активируете ссылочные типы, допускающие значение `null`, и хотите, чтобы такому типу было присвоено значение `null`, то вам необходимо использовать тот же синтаксис, что и для типа значения, допускающего значение `null`, то есть добавить символ `?` после объявления типа.

Итак, как работают ссылочные типы, допускающие значение `null`? Рассмотрим пример, в котором при хранении информации об адресе может потребоваться ввести значения для улицы, города и региона, но номер здания можно оставить пустым, то есть `null`.

1. В проекте `NullHandling.csproj` в конце файла `Program.cs` добавьте операторы для объявления класса `Address` с четырьмя полями:

```
class Address
{
  public string? Building;
  public string Street;
  public string City;
  public string Region;
}
```

2. Обратите внимание, что через несколько секунд расширение C# предупреждает о проблемах с полями, не допускающими значение `null`, такими как `Street` (рис. 6.2).

Рис. 6.2. Сообщения, предупреждающие о полях, не допускающих значения null, в окне PROBLEMS (Проблемы)

3. Каждому из трех полей назначьте пустое значение `string`, не допускающее значение `null`:

```
public string Street = string.Empty;
public string City = string.Empty;
public string Region = string.Empty;
```

4. В начале файла `Program.cs` статически импортируйте `Console`, а затем добавьте операторы для создания экземпляра адреса и установки его свойств:

```
Address address = new();
address.Building = null;
address.Street = null;
address.City = "London";
address.Region = null;
```

5. Обратите внимание на предупреждающее сообщение (рис. 6.3).

Рис. 6.3. Сообщение, предупреждающее о присвоении значения null полю, не допускающему это значение

Вот почему новая функция языка называется ссылочными типами, допускающими значение `null`. Начиная с версии C# 8.0, ссылочные типы без дополнений могут стать типами, не допускающими это значение. Для того чтобы ссылочный тип допускал значение `null`, используется тот же синтаксис, который применяется для типов значений.

Проверка на null

Важно проверять, содержит ли значение `null` переменная ссылочного типа или типа, допускающего значение `null`. В противном случае может возникнуть исключение `NullReferenceException`, что приведет к ошибке при выполнении кода. Прежде чем использовать переменную, допускающую значение `null`, следует выполнить проверку на `null`:

```
// проверяем, что переменная не равна нулю, прежде чем использовать ее
if (thisCouldBeNull != null)
{
  // получаем доступ к члену thisCouldBeNull
  int length = thisCouldBeNull.Length; // может возникнуть исключение
  ...
}
```

Для сравнения: код C# 7 сочетается с операцией ! (НЕ) в качестве альтернативы !=.

```
if (!(thisCouldBeNull is null))
{
```

Для сравнения: код C# 9 включает is not как еще более понятную альтернативу:

```
if (thisCouldBeNull is not null)
{
```

Если вы пытаетесь использовать член переменной, которая может равна null, используйте операцию доступа к членам с проверкой на null (?.):

```
string authorName = null;

// следующий код генерирует исключение NullReferenceException
int x = authorName.Length;

// вместо того чтобы генерировать исключение, y присваивается null
int? y = authorName?.Length;
```

Иногда требуется либо назначить переменную в качестве результата, либо использовать альтернативное значение, например 3, если переменная равна null. Это можно сделать с помощью операции объединения с null (??):

```
// результатом будет значение 3, если authorName?.Length равно null
int result = authorName?.Length ?? 3;
Console.WriteLine(result);
```

Даже если вы добавите ссылочные типы, допускающие значение null, вам в любом случае необходимо проверять параметры, не допускающие значение null, на наличие null и вызывать исключение ArgumentNullException.

Проверка на null в параметрах метода

При определении методов с параметрами рекомендуется выполнять проверку на null.

В более ранних версиях C# вам приходилось добавлять операторы if для проверки значений параметров на null, а затем создавать исключение ArgumentNullException для любого параметра, имеющего значение null:

```
public void Hire(Person manager, Person employee)
{
  if (manager == null)
  {
    throw new ArgumentNullException(nameof(manager));
  }
```

```
  if (employee == null)
  {
    throw new ArgumentNullException(nameof(employee));
  }
  ...
}
```

В C# 11 может появиться новый суффикс `!!`, упрощающий работу:

```
public void Hire(Person manager!!, Person employee!!)
{
  ...
}
```

Итак, оператор `if` добавлен и генерация исключения выполнена.

Наследование классов

Тип `Person`, созданный нами ранее, неявно произведен (унаследован) от типа `object`, псевдонима `System.Object`. Теперь мы создадим подкласс, наследуемый от класса `Person`.

1. В проекте `PacktLibrary` создайте файл класса `Employee.cs`.
2. Измените его содержимое, чтобы определить класс `Employee`, производный от `Person`, как показано ниже:

```
using System;

namespace Packt.Shared;

public class Employee : Person
{
}
```

3. В проекте `PeopleApp` в файле `Program.cs` добавьте операторы для создания экземпляра класса `Employee`:

```
Employee john = new()
{
  Name = "John Jones",
  DateOfBirth = new(year: 1990, month: 7, day: 28)
};
john.WriteToConsole();
```

4. Запустите код и проанализируйте результат:

```
John Jones was born on a Saturday.
```

Обратите внимание, что класс `Employee` унаследовал все члены класса `Person`.

Расширение классов

Теперь мы добавим несколько членов, относящихся только к сотрудникам (`Employee`), тем самым расширив класс.

1. В файле `Employee.cs` добавьте следующие операторы, чтобы определить два свойства — для кода сотрудника и даты найма:

```
public string? EmployeeCode { get; set; }
public DateTime HireDate { get; set; }
```

2. В файле `Program.cs` добавьте операторы для определения кода сотрудника Джона и даты найма:

```
john.EmployeeCode = "JJ001";
john.HireDate = new(year: 2014, month: 11, day: 23);
WriteLine($"{john.Name} was hired on {john.HireDate:dd/MM/yy}");
```

3. Запустите код и проанализируйте результат:

```
John Jones was hired on 23/11/14
```

Сокрытие 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 в отношении сотрудника.

1. В файле `Employee.cs` добавьте операторы для переопределения метода `WriteToConsole`, как показано ниже (выделено жирным шрифтом):

```
using static System.Console;

namespace Packt.Shared;

public class Employee : Person
{
  public string? EmployeeCode { get; set; }
  public DateTime HireDate { get; set; }

  public void WriteToConsole()
  {
    WriteLine(format:
      "{0} was born on {1:dd/MM/yy} and hired on {2:dd/MM/yy}",
      arg0: Name,
      arg1: DateOfBirth,
      arg2: HireDate);
  }
}
```

2. Запустите код и проанализируйте результат:

```
John Jones was born on 28/07/90 and hired on 01/01/01
John Jones was hired on 23/11/14
```

Ваша среда разработки предупредит вас о том, что ваш метод теперь скрывает метод от класса `Person`, отобразив зеленую волнистую линию под именем вашего метода. Окно PROBLEMS (Проблемы)/Error List (Список ошибок) содержит более подробную информацию, а компилятор выдаст предупреждение при сборке и запуске консольного приложения (рис. 6.4).

Рис. 6.4. Сообщение о сокрытии метода

Избавиться от предупреждения можно, добавив в метод ключевое слово `new`, указывающее на то, что вы намеренно замещаете старый метод, как показано ниже:

```
public new void WriteToConsole()
```

Переопределение членов

Вместо того чтобы скрывать метод, обычно лучше *переопределить* его. Вы можете переопределять члены только в том случае, если базовый класс допускает это, применяя ключевое слово `virtual` к любым методам, которые должны разрешать переопределение.

Рассмотрим пример.

1. В файле `Program.cs` добавьте оператор для записи значения переменной `john` в консоль с помощью ее строкового представления, как показано ниже:

```
WriteLine(john.ToString());
```

2. Запустите код и обратите внимание, что метод `ToString` наследуется от типа `System.Object`, поэтому реализация выводит пространство имен и имя типа, как показано ниже:

```
Packt.Shared.Employee
```

3. В файле `Person.cs` переопределите это поведение, добавив метод `ToString` для вывода имени человека, а также имени типа:

```
// переопределенные методы
public override string ToString()
{
  return $"{Name} is a {base.ToString()}";
}
```

Ключевое слово `base` позволяет подклассу получать доступ к членам своего суперкласса, то есть *базового класса*, от которого он наследуется.

4. Запустите код и проанализируйте результат. Теперь, когда вызывается метод `ToString`, он выводит имя человека, а также возвращает реализацию базового класса `ToString`:

```
John Jones is a Packt.Shared.Employee
```

Многие существующие API, например Entity Framework Core от Microsoft, DynamicProxy от Castle и модели контента Episerver, требуют, чтобы свойства, определяемые программистами в своих классах, определялись как виртуальные, чтобы их можно было переопределить. Проанализируйте и решите, какие члены вашего метода и свойства должны быть указаны как virtual.

Наследование от абстрактных классов

Ранее в этой главе вы узнали об интерфейсах, которые могут определять набор членов, чтобы соответствовать базовому уровню функциональности. Они полезны, но их основное ограничение заключается в том, что до выхода версии C# 8 они не могли предоставить собственную реализацию.

Это проблема, особенно если вам по-прежнему необходимо создавать библиотеки классов, которые будут работать с .NET Framework и другими платформами, не поддерживающими .NET Standard 2.1.

На этих более ранних платформах вы могли использовать абстрактные классы как нечто среднее между чистым интерфейсом и полностью реализованным классом.

Когда класс помечен как `abstract`, это означает, что он не может быть создан, поскольку вы указываете, что класс не реализован. Прежде чем его можно будет создать, требуется дополнительная реализация.

Например, класс `System.IO.Stream` — абстрактный, поскольку реализует общие функции, необходимые всем потокам, но не является полным, поэтому вы не можете создать его экземпляр с помощью `new Stream()`.

Сравним два типа интерфейса и два типа класса:

```
public interface INoImplementation // C# 1.0 и более поздние версии
{
  void Alpha(); // должен быть реализован производным типом
}

public interface ISomeImplementation // C# 8.0 и более поздние версии
{
  void Alpha(); // должен быть реализован производным типом
  void Beta()
  {
    // реализация по умолчанию; может быть переопределен
  }
}

public abstract class PartiallyImplemented // C# 1.0 и более поздние версии
{
  public abstract void Gamma(); // должен быть реализован производным типом

  public virtual void Delta() // может быть переопределен
  {
    // реализация
  }
}

public class FullyImplemented : PartiallyImplemented, ISomeImplementation
{
  public void Alpha()
  {
    // реализация
  }

  public override void Gamma()
  {
    // реализация
  }
}

// вы можете создать экземпляр полностью реализованного класса
FullyImplemented a = new();
```

```
// все остальные типы выдают ошибки компиляции
PartiallyImplemented b = new(); // Ошибка компиляции!
ISomeImplementation c = new(); // Ошибка компиляции!
INoImplementation d = new(); // Ошибка компиляции!
```

Предотвращение наследования и переопределения

Вы можете предотвратить наследование своего класса, указав в его определении ключевое слово `sealed` (запечатанный). Никто не сможет наследовать от запечатанного класса `ScroogeMcDuck`:

```
public sealed class ScroogeMcDuck
{
}
```

Примером запечатанного класса может служить `string`. Компания Microsoft внедрила в него некоторые критические оптимизации, на которые наследование может повлиять негативным образом, и поэтому запечатала его.

Вы можете предотвратить переопределение метода `virtual` в своем классе, указав имя метода с ключевым словом `sealed`. Никто не сможет изменить голос Леди Гага:

```
using static System.Console;

namespace Packt.Shared;

public class Singer
{
  // метод virtual позволяет переопределить этот метод
  public virtual void Sing()
  {
    WriteLine("Singing...");
  }
}

public class LadyGaga : Singer
{
   // запечатанность предотвращает переопределение метода в подклассах
  public sealed override void Sing()
  {
    WriteLine("Singing with style...");
  }
}
```

Вы можете запечатать только переопределенный метод.

Полиморфизм

Теперь вы знаете два способа изменения поведения унаследованного метода. Мы можем скрыть его с помощью ключевого слова `new` *(неполиморфное наследование)* или переопределить *(полиморфное наследование)*.

Оба способа могут вызывать базовый класс или суперкласс с помощью ключевого слова `base`. Так в чем же разница?

Все зависит от типа переменной, содержащей ссылку на объект. Например, переменная типа `Person` может содержать ссылку на класс `Person` или любой тип, производный от класса `Person`.

Рассмотрим пример.

1. В файле `Employee.cs` добавьте операторы для переопределения метода `ToString`, чтобы он записывал имя и код сотрудника в консоль, как показано ниже:

```
public override string ToString()
{
  return $"{Name}'s code is {EmployeeCode}";
}
```

2. В файле `Program.cs` добавьте операторы для создания сотрудника `Alice`, сохраните их в переменной типа `Person` и вызовите методы `WriteToConsole` и `ToString` для обеих переменных:

```
Employee aliceInEmployee = new()
  { Name = "Alice", EmployeeCode = "AA123" };

Person aliceInPerson = aliceInEmployee;
aliceInEmployee.WriteToConsole();
aliceInPerson.WriteToConsole();
WriteLine(aliceInEmployee.ToString());
WriteLine(aliceInPerson.ToString());
```

3. Запустите код и проанализируйте результат:

```
Alice was born on 01/01/01 and hired on 01/01/01
Alice was born on a Monday
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 — сотрудник (`Employee`), поэтому вызывает метод `WriteToConsole` класса `Person`.

Если метод переопределяется с помощью ключевых слов `virtual` и `override`, то компилятор «понимает», что, хоть переменная и объявлена как класс `Person`, сам объект — это `Employee`, и потому вызывается реализация `Employee` метода `ToString`.

Модификаторы доступа и их влияние на работу кода показаны в табл. 6.2.

Таблица 6.2. Модификаторы доступа

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 с практической точки зрения кажется малоперспективной. Если вы освоите данную концепцию — прекрасно; а если нет — не волнуйтесь. Некоторым программистам нравится объявлять себя всезнайками, 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 подробностей работы системы впрыска.

Чтобы изменить реализацию унаследованного метода, по возможности используйте ключевые слова virtual и override, а не new.

Приведение в иерархиях наследования

Приведение типов несколько отличается от преобразования типов. Приведение выполняется между похожими типами, нпример между 16- и 32-битными целыми числами или между суперклассом и одним из его подклассов. Преобразование выполняется между разными типами, например между текстом и числом.

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

Вы можете пойти другим путем и применить явное приведение, добавив круглые скобки вокруг типа, который хотите преобразовать в качестве префикса.

1. В файле `Program.cs` добавьте оператор для назначения переменной `aliceInPerson` новой переменной `Employee`, как показано ниже:

```
Employee explicitAlice = aliceInPerson;
```

2. Программа Visual Studio Code отображает ошибку компиляции и подчеркивает ее красной волнистой линией (рис. 6.5).

Рис. 6.5. Явная ошибка компиляции при приведении

3. Измените оператор, добавив перед именем присваиваемой переменной явное приведение к типу `Employee`, как показано ниже:

```
Employee explicitAlice = (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. В реальном приложении с более сложным кодом текущее значение этой переменной могло быть установлено на экземпляр `Student`, и тогда данный оператор может вызвать ошибку `InvalidCastException`.

Справиться с ней можно, написав оператор `try`, но существует более удобный способ: проверить текущий тип объекта с помощью ключевого слова `is`.

1. Оберните оператор явного приведения оператором `if`, как показано ниже:

```
if (aliceInPerson is Employee)
{
  WriteLine($"{nameof(aliceInPerson)} IS an Employee");
  Employee explicitAlice = (Employee)aliceInPerson;
  // безопасно выполняем что-либо с explicitAlice
}
```

2. Запустите код и проанализируйте результат:

```
aliceInPerson IS an Employee
```

Вы можете еще больше упростить код, используя шаблон объявления, и это позволит избежать необходимости выполнять явное приведение типов, как показано в коде ниже:

```
if (aliceInPerson is Employee explicitAlice)
{
  WriteLine($"{nameof(aliceInPerson)} IS an Employee");
  // безопасно выполняем что-либо с explicitAlice
}
```

Кроме того, для приведения вы можете использовать ключевое слово `as`. Вместо того чтобы вызывать исключение, ключевое слово `as` возвращает `null`, если тип не может быть приведен.

3. В файл `Program.cs` добавьте операторы для приведения переменной `Alice` с помощью ключевого слова `as`, а затем проверьте, не равно ли возвращаемое значение `null`:

```
Employee? aliceAsEmployee = aliceInPerson as Employee; // может быть null

if (aliceAsEmployee != null)
{
  WriteLine($"{nameof(aliceInPerson)} AS an Employee");
  // безопасно выполняем что-либо с aliceAsEmployee
}
```

Поскольку доступ к переменной со значением `null` может вызвать ошибку `NullReferenceException`, вы должны всегда проверять значение на `null`, прежде чем использовать результат.

4. Запустите код и проанализируйте результат:

```
aliceInPerson AS an Employee
```

Что, если вы хотите выполнить блок операторов, когда Alice не является сотрудником?

Раньше вам приходилось использовать операцию `!`:

```
if (!(aliceInPerson is Employee))
```

В C# 9 и более поздних версиях вы можете использовать ключевое слово `not`:

```
if (aliceInPerson is not Employee)
```

Используйте ключевые слова is и as, чтобы избежать вызова исключений при приведении производных типов. Если вы этого не сделаете, то должны написать операторы try … catch для InvalidCastException.

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 компании Microsoft, чтобы унаследовать часть или все его поведение, а затем переопределить или расширить его.

Наследование исключений

В качестве примера наследования мы выведем новый тип исключения.

1. В проекте `PacktLibrary` создайте файл `PersonException.cs`.
2. Измените код, чтобы определить класс `PersonException` с тремя конструкторами:

```
namespace Packt.Shared;

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, вследствие чего мы должны явно объявить и явно вызвать реализации конструктора `base` в `System.Exception`, чтобы сделать их доступными для программистов, которые могут захотеть применить эти конструкторы с нашим пользовательским исключением.

3. В файле `Person.cs` добавьте следующие операторы, чтобы определить метод, который выдает исключение, если параметр даты/времени меньше даты рождения человека:

```
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}!");
  }
}
```

4. В файле `Program.cs` добавьте операторы для определения того, что произойдет, если Джон Джонс переместится в машине времени слишком далеко:

```
try
{
  john.TimeTravel(when: new(1999, 12, 31));
  john.TimeTravel(when: new(1950, 12, 25));
}
catch (PersonException ex)
{
  WriteLine(ex.Message);
}
```

5. Запустите код и проанализируйте результат:

```
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 передавайте им те же три конструктора, явно вызывающие встроенные конструкторы.

Расширение типов при невозможности наследования

Ранее вы узнали, как можно использовать модификатор `sealed` для предотвращения наследования.

Компания Microsoft применила 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# 3.0.

Применение статических методов для многократного использования функций

Начиная с первой версии языка C#, мы можем создавать статические методы для многократного использования функций, таких как возможность проверки того, содержит ли строка адрес электронной почты. Эта реализация будет применять регулярное выражение, более подробную информацию о котором вы получите в главе 8.

Рассмотрим пример.

1. В проекте `PacktLibrary` создайте класс `StringExtensions`, как показано ниже, и обратите внимание на следующие моменты:
   - класс импортирует пространство имен для обработки регулярных выражений;
   - статический метод `IsValidEmail` использует тип `Regex` для проверки совпадений с простым шаблоном электронной почты, который ищет допустимые символы до и после символа `@`.

```
using System.Text.RegularExpressions;

namespace Packt.Shared;

public class StringExtensions
{
  public static bool IsValidEmail(string input)
  {
    // используем простое регулярное выражение для проверки того,
    // что входная строка является действительным адресом
    // электронной почты
    return Regex.IsMatch(input,
      @"[a-zA-Z0-9\.-_]+@[a-zA-Z0-9\.-_]+");
  }
}
```

2. В файл `Program.cs` добавьте операторы для проверки двух примеров адресов электронной почты:

```
string email1 = "pamela@test.com";
string email2 = "ian&test.com";

WriteLine("{0} is a valid e-mail address: {1}",
  arg0: email1,
  arg1: StringExtensions.IsValidEmail(email1));

WriteLine("{0} is a valid e-mail address: {1}",
  arg0: email2,
  arg1: StringExtensions.IsValidEmail(email2));
```

3. Запустите код и проанализируйте результат:

```
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.

Применение методов расширения для многократного использования функций

Методы `static` легко превратить в методы расширения.

1. В файле `StringExtensions.cs` добавьте модификатор `static` перед классом, а модификатор `this` — перед типом `string`, как показано ниже (выделено жирным шрифтом):

   ```
   public static class StringExtensions
   {
     public static bool IsValidEmail(this string input)
     {
   ```

   Благодаря этим двум изменениям компилятор будет знать, что он должен рассматривать код как метод, расширяющий тип `string`.

2. В файле `Program.cs` добавьте операторы, чтобы использовать метод расширения для значений типа `string`, которые необходимо проверить на допустимые адреса электронной почты:

   ```
   WriteLine("{0} is a valid e-mail address: {1}",
     arg0: email1,
     arg1: email1.IsValidEmail());

   WriteLine("{0} is a valid e-mail address: {1}",
     arg0: email2,
     arg1: email2.IsValidEmail());
   ```

   Обратите внимание на небольшое изменение в синтаксисе вызова метода `IsValidEmail`. При этом устаревший и длинный синтаксис все еще работает.

3. Метод расширения `IsValidEmail` теперь выглядит как метод, аналогичный всем действительным методам экземпляра типа `string`, например `IsNormalized` и `Insert` (рис. 6.6).

Рис. 6.6. Методы расширения отображаются в IntelliSense наряду с методами экземпляра

4. Запустите код и проанализируйте результат, который будет таким же, как и раньше.

Методы расширения не могут заменить или переопределить используемые методы экземпляра. Так, вы не можете переопределить метод Insert. Метод расширения будет отображаться в IntelliSense как перегрузка, но вместо метода расширения с таким же именем и сигнатурой будет вызываться метод экземпляра.

Может показаться, что методы расширения не дают большого преимущества, однако в главе 11 вы увидите несколько чрезвычайно эффективных способов их использования.

Использование анализатора для написания улучшенного кода

.NET-анализаторы находят потенциальные проблемы и предлагают способы их устранения. StyleCop — это широко используемый анализатор, помогающий создавать более качественный код C#.

Рассмотрим это на практике, изучив, как улучшить код в шаблоне проекта для консольного приложения, рассчитанного на использование в .NET 5.0 и содержащего класс `Program` с методом `Main`.

1. Откройте редактор кода и создайте консольное приложение, задав такие настройки:
   1) шаблон проекта: Console Application/`console -f net5.0`;
   2) файл и папка рабочей области/решения: `Chapter06`;
   3) файл и папка проекта: `CodeAnalyzing`;
   4) целевая платформа: .NET 5.0 (Current).
2. В проекте `CodeAnalyzing` добавьте ссылку на пакет для `StyleCop.Analyzers`.
3. Для управления настройками StyleCop добавьте в проект файл JSON `stylecop.json`.
4. Измените его содержимое следующим образом:

```
{
  "$schema": "https://raw.githubusercontent.com/DotNetAnalyzers/
  StyleCopAnalyzers/master/StyleCop.Analyzers/StyleCop.Analyzers/Settings/
  stylecop.schema.json",
```

```
  "settings": {
  }
}
```

Запись $schema включает IntelliSense при редактировании файла stylecop.json в редакторе кода.

5. Отредактируйте файл проекта, измените целевую платформу на `net6.0`, добавьте записи для настройки файла `stylecop.json`, чтобы он не включался в опубликованные развертывания, и включите его в качестве дополнительного файла для обработки, как показано ниже (выделено жирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <None Remove="stylecop.json" />
  </ItemGroup>

  <ItemGroup>
    <AdditionalFiles Include="stylecop.json" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="StyleCop.Analyzers" Version="1.2.0-*">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers
      </IncludeAssets>
    </PackageReference>
  </ItemGroup>

</Project>
```

6. Соберите проект.
7. Вы увидите предупреждения об ошибках (рис. 6.7).
8. Например, необходимо, чтобы директивы использования были помещены в объявление пространства имен:

```
C:\Code\Chapter06\CodeAnalyzing\Program.cs(1,1): warning SA1200: Using
directive should appear within a namespace declaration [C:\Code\Chapter06\
CodeAnalyzing\CodeAnalyzing.csproj]
```

Рис. 6.7. Предупреждения анализатора кода StyleCop

Подавление предупреждений

Существует несколько способов подавить предупреждение, включая добавление кода и настройку конфигурации.

Подавить использование атрибута можно следующим образом:

```
[assembly:SuppressMessage("StyleCop.CSharp.OrderingRules",
"SA1200:UsingDirectivesMustBePlacedWithinNamespace", Justification =
"Reviewed.")]
```

А так можно подавить использование директивы:

```
#pragma warning disable SA1200 // UsingDirectivesMustBePlacedWithinNamespace
using System;
#pragma warning restore SA1200 // UsingDirectivesMustBePlacedWithinNamespace
```

Рассмотрим пример подавления предупреждения, изменив файл `stylecop.json`.

1. В файле `stylecop.json` добавьте параметр конфигурации, чтобы разрешить использование операторов `using` за пределами пространства имен:

```
{
  "$schema": "https://raw.githubusercontent.com/DotNetAnalyzers/
  StyleCopAnalyzers/master/StyleCop.Analyzers/StyleCop.Analyzers/Settings/
  stylecop.schema.json",
  "settings": {
    "orderingRules": {
```

```
      "usingDirectivesPlacement": "outsideNamespace"
    }
  }
}
```

2. Соберите проект и обратите внимание, что предупреждение SA1200 исчезло.
3. В файле `stylecop.json` задайте для директив `using` значение `preserve`, что позволяет использовать операторы как внутри, так и вне пространства имен:

```
"orderingRules": {
  "usingDirectivesPlacement": "preserve"
}
```

Исправление ошибок в коде

Теперь исправим все остальные ошибки.

1. В файле `CodeAnalyzing.csproj` добавьте элемент для автоматического создания XML-файла для документации:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <GenerateDocumentationFile>true</GenerateDocumentationFile>
  </PropertyGroup>
```

2. В файле `stylecop.json` добавьте параметр конфигурации, чтобы предоставить значения для документации для названия компании и копирайта:

```
{
  "$schema": "https://raw.githubusercontent.com/DotNetAnalyzers/
  StyleCopAnalyzers/master/StyleCop.Analyzers/StyleCop.Analyzers/Settings/
  stylecop.schema.json",
  "settings": {
     "orderingRules": {
       "usingDirectivesPlacement": "preserve"
     },
     "documentationRules": {
       "companyName": "Packt",
       "copyrightText": "Copyright (c) Packt. All rights reserved."
     }
  }
}
```

3. В файле `Program.cs` к заголовку файла добавьте комментарии с текстом компании и копирайтом, переместите объявление `using System;` внутрь пространства

имен и установите явные модификаторы доступа и комментарии XML для класса и метода:

```
// <copyright file="Program.cs" company="Packt">
// Copyright (c) Packt. All rights reserved.
// </copyright>

namespace CodeAnalyzing
{
  using System;

  /// <summary>
  /// основной класс для этого консольного приложения
  /// </summary>
  public class Program
  {
    /// <summary>
    /// основная точка входа для этого консольного приложения
    /// </summary>
    /// <param name="args">A string array of arguments passed
    /// to the console app.</param>
    public static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

4. Соберите проект.
5. Разверните папку bin/Debug/net6.0 и обратите внимание на автоматически сгенерированное имя файла `CodeAnalyzing.xml`:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CodeAnalyzing</name>
    </assembly>
    <members>
        <member name="T:CodeAnalyzing.Program">
            <summary>
            The main class for this console app.
            </summary>
        </member>
        <member name="M:CodeAnalyzing.Program.Main(System.String[])">
            <summary>
            The main entry point for this console app.
            </summary>
            <param name="args">A string array of arguments passed to the
            console app.</param>
        </member>
    </members>
</doc>
```

Общие рекомендации StyleCop

Внутри файла кода необходимо упорядочить содержимое в соответствии со следующим списком.

1. Директивы внешнего псевдонима.
2. Использование директив.
3. Пространства имен.
4. Делегаты.
5. Перечисления.
6. Интерфейсы.
7. Структуры.
8. Классы.

Внутри класса, записи, структуры или интерфейса необходимо упорядочить содержимое в соответствии со следующим списком.

1. Поля.
2. Конструкторы.
3. Деструкторы (финализаторы).
4. Делегаты.
5. События.
6. Перечисления.
7. Интерфейсы.
8. Свойства.
9. Индексаторы.
10. Методы.
11. Структуры.
12. Вложенные классы и записи.

Со всеми правилами StyleCop можно ознакомиться на сайте https://github.com/DotNetAnalyzers/StyleCopAnalyzers/blob/master/DOCUMENTATION.md.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 6.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Что такое делегат?
2. Что такое событие?
3. Как связаны базовый и производный классы и как производный может получить доступ к базовому?
4. В чем разница между операциями `is` и `as`?
5. С помощью какого ключевого слова можно предотвратить наследование класса и переопределение метода?
6. Какое ключевое слово используется для предотвращения создания экземпляра класса с помощью ключевого слова `new`?
7. Какое ключевое слово служит для переопределения члена?
8. Чем деструктор отличается от метода деконструктора?
9. Как выглядят сигнатуры конструкторов, которые должны иметь все исключения?
10. Что такое метод расширения и как его определить?

Упражнение 6.2. Создание иерархии наследования

Исследуйте иерархии наследования, выполнив следующие действия.

1. Создайте консольное приложение `Exercise02` и добавьте его в вашу рабочую область `Chapter06`.
2. Создайте класс `Shape` со свойствами `Height`, `Width` и `Area`.
3. Добавьте три унаследованных от него класса — `Rectangle`, `Square` и `Circle` — с любыми дополнительными членами, которые, по вашему мнению, подходят и правильно переопределяют и реализуют свойство `Area`.
4. В методе `Main` добавьте операторы для создания одного экземпляра каждой фигуры, как показано ниже:

```
Rectangle r = new(height: 3, width: 4.5);
WriteLine($"Rectangle H: {r.Height}, W: {r.Width}, Area: {r.Area}");

Square s = new(5);
WriteLine($"Square H: {s.Height}, W: {s.Width}, Area: {s.Area}");

Circle c = new(radius: 2.5);
WriteLine($"Circle H: {c.Height}, W: {c.Width}, Area: {c.Area}");
```

5. Запустите консольное приложение и убедитесь, что результат выглядит следующим образом:

```
Rectangle H: 3, W: 4.5, Area: 13.5
Square H: 5, W: 5, Area: 25
Circle H: 5, W: 5, Area: 19.6349540849362
```

Упражнение 6.3. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-6---implementing-interfaces-and-inheriting-classes, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы познакомились с локальными функциями и операциями, делегатами и событиями, реализацией интерфейсов, дженериками и получением типов с помощью наследования и объектно-ориентированного программирования. Вы также узнали о базовых и производных классах, о том, как переопределить член типа, использовать полиморфизм и выполнять приведение типов.

В главе 7 вы узнаете о том, как упаковывается и развертывается .NET, а в последующих главах — о том, какие типы она предоставляет для реализации общей функциональности: работы с файлами, доступа к базе данных, шифрования и многозадачности.

7 Упаковка и распространение типов .NET

Изучив данную главу, вы узнаете, как ключевые слова C# связаны с типами .NET, а также о взаимосвязи пространств имен и сборок. Кроме того, познакомитесь с тем, как упаковывать и публиковать кросс-платформенные приложения и библиотеки .NET, использовать устаревшие библиотеки .NET Framework в библиотеках .NET, а также с тем, как переносить устаревшие кодовые базы .NET Framework в .NET.

В этой главе:

- введение в .NET 6;
- компоненты .NET;
- публикация и развертывание ваших приложений;
- декомпиляция сборок .NET;
- упаковка библиотек для распространения NuGet;
- перенос приложений с .NET Framework на современную .NET.
- работа с функциями предварительного просмотра.

Введение в .NET 6

Данная часть книги посвящена функциональным возможностям *библиотек базовых классов* (Base Class Library, BCL) API, предоставляемых .NET, а также повторному использованию функциональных возможностей на всех различных платформах .NET с помощью .NET Standard.

В первую очередь рассмотрим новые версии и то, почему важно понимать работу более старых версий.

Поддержка средой .NET Core 2.0 и более поздними версиями стандарта .NET Standard 2.0 важна, поскольку он содержит многие API, отсутствующие в первой версии .NET Core. Среди накопленных за 15 лет библиотек и приложений, доступных разработчикам, использующим .NET Framework, были отобраны актуальные

для современной разработки и перенесены на .NET Core, так что теперь они могут работать на нескольких различных платформах в операционных системах Windows, macOS и Linux.

.NET Standard 2.1 добавил еще около 3000 новых API. Некоторые из них нуждаются в изменениях в среде выполнения, нарушающих обратную совместимость, вследствие чего .NET Framework 4.8 реализует только .NET Standard 2.0. Платформы .NET Core 3.0, Xamarin, Mono и Unity реализуют .NET Standard 2.1.

.NET 6 устраняет необходимость в .NET Standard, если все ваши проекты могут использовать .NET 6. Поскольку вам все еще может потребоваться создавать библиотеки классов для устаревших проектов .NET Framework или мобильных приложений Xamarin, то по-прежнему необходимо создавать библиотеки классов версий .NET Standard 2.0 и 2.1. В марте 2021 года я провел опрос среди опытных разработчиков, и половина отвечавших все еще нуждались в создании библиотек классов, совместимых с .NET Standard 2.0.

Теперь, когда версия .NET 6 уже выпущена с предварительной поддержкой мобильных и настольных приложений, созданных с использованием .NET MAUI, потребность в .NET Standard снизится еще больше.

Чтобы продемонстрировать прогресс .NET Core за последние пять лет, я сравнил основные версии .NET Core и современные версии .NET с эквивалентными версиями .NET Framework:

- *.NET Core 1.x* — содержит гораздо меньше API по сравнению с платформой .NET Framework 4.6.1, выпущенной в марте 2016 года;
- *.NET Core 2.x* — достигнут паритет API с платформой .NET Framework 4.7.1 для современных API, поскольку обе реализуют .NET Standard 2.0;
- *.NET Core 3.x* — содержит больше API по сравнению с платформой .NET Framework для современных API, поскольку .NET Framework 4.8 не поддерживает .NET Standard 2.1.
- *.NET 5* — еще более крупный API по сравнению с .NET Framework 4.8 для современных API со значительно улучшенной производительностью;
- *.NET 6* — окончательная унификация с поддержкой мобильных приложений в .NET MAUI, ожидалась к маю 2022 года.

.NET Core 1.0

Платформа .NET Core 1.0 вышла в июне 2016 года и была ориентирована на реализацию API, подходящего для создания современных кросс-платформенных приложений, включая веб- и облачные приложения, а также сервисы для Linux с использованием ASP.NET Core.

.NET Core 1.1

Платформа .NET Core 1.1 вышла в ноябре 2016 года и была направлена на исправление ошибок, увеличение количества поддерживаемых дистрибутивов Linux, поддержку .NET Standard 1.6 и повышение производительности, особенно в ASP.NET Core для веб-приложений и сервисов.

.NET Core 2.0

Платформа .NET Core 2.0 вышла в августе 2017 года и была сфокусирована на реализации .NET Standard 2.0, возможности использовать библиотеки .NET Framework и другие обновления.

Третье издание этой книги было выпущено в ноябре 2017 года, поэтому охватывало платформы .NET Core 2.0 и .NET Core для приложений *универсальной платформы Windows* (Universal Windows Platform, UWP).

.NET Core 2.1

Платформа .NET Core 2.1 вышла в мае 2018 года и была ориентирована на создание расширяемой системы инструментов, в которую были добавлены новые типы, такие как `Span<T>`, новые API для криптографии и сжатия, метапакет Windows Compatibility Pack с дополнительными 20 000 API для портирования устаревших Windows-приложений, преобразование значений в Entity Framework Core, преобразования `GroupBy` в LINQ, заполнение данными, типы запросов и даже дополнительные обновления производительности, включая темы, перечисленные в табл. 7.1.

Таблица 7.1. Дополнения в .NET Core 2.1

Функционал	Глава	Тема
Интервалы	8	Работа с интервалами, индексами и диапазонами
Алгоритм Brotli	9	Сжатие с помощью алгоритма Brotli (алгоритм сжатия данных с открытым исходным кодом)
Криптография	20	Новое в криптографии
Отложенная загрузка EF Core	10	Включение отложенной загрузки
Заполнение данными EF Core	10	Заполнение данными

.NET Core 2.2

Платформа .NET Core 2.2 вышла в декабре 2018 года и была направлена на усовершенствование диагностики для среды выполнения, необязательную многоуровневую компиляцию и добавление новых функций в ASP.NET Core и Entity Framework Core, таких как поддержка пространственных данных с использованием типов из библиотеки *NetTopologySuite (NTS)*, тегов запросов и коллекций собственных сущностей.

.NET Core 3.0

Платформа .NET Core 3.0 вышла в сентябре 2019 года и была ориентирована на добавление поддержки создания настольных приложений для операционной системы Windows с использованием Windows Forms (2001), Windows Presentation Foundation (WPF; 2006) и Entity Framework 6.3, параллельное и локальное развертывание приложений, быстрое чтение JSON-файлов, доступ к последовательному порту и другим контактам (пинам) для решений в сфере *Интернета вещей* (Internet of Things, IoT) и многоуровневую компиляцию по умолчанию, включая темы, перечисленные в табл. 7.2.

Таблица 7.2. Дополнения в .NET Core 3.0

Функционал	Глава	Тема
Встраивание .NET Core в приложение	7	Публикация и развертывание ваших приложений
`Index` и `Range`	8	Работа с интервалами, индексами и диапазонами
`System.Text.Json`	9	Высокопроизводительная обработка JSON
Асинхронные потоки	12	Работа с асинхронными потоками

Четвертое издание этой книги было опубликовано в октябре 2019 года, поэтому охватывало некоторые новые API, добавленные в более поздних версиях.

.NET Core 3.1

Платформа .NET Core 3.1 вышла в декабре 2019 года и была нацелена на исправление ошибок и улучшения, чтобы этот выпуск имел *долгосрочную поддержку* (Long Term Support, LTS) и не терял ее до декабря 2022 года.

.NET 5.0

Платформа .NET 5.0 вышла в ноябре 2020 года и была направлена на унификацию различных платформ .NET, доработку платформы и повышение производительности, включая темы, перечисленные в табл. 7.3.

Таблица 7.3. Дополнения в .NET 5.0

Функционал	Глава	Тема
Тип `Half`	8	Работа с числами
Улучшение производительности регулярных выражений	8	Улучшения производительности регулярных выражений
Улучшение `System.Text.Json`	9	Высокопроизводительная обработка JSON
EF Core, сгенерированный SQL	10	Получение сгенерированного SQL
Включение фильтрации EF Core	10	Фильтрация включенных сущностей
EF Core Scaffold-DbContext теперь сингуляризует с помощью Humanizer	10	Создание шаблонов с использованием существующей базы данных

.NET 6.0

Платформа .NET 6.0 вышла в ноябре 2021 года и была нацелена на унификацию с мобильной платформой, добавление дополнительных функций в EF Core для управления данными, а также на повышение производительности, включая темы, перечисленные в табл. 7.4.

Таблица 7.4. Дополнения в .NET 6.0

Функционал	Глава	Тема
Проверка статуса .NET SDK	7	Проверка пакетов SDK для .NET на наличие обновлений
Поддержка Apple Silicon	7	Создание консольного приложения для публикации
Связать режим обрезки по умолчанию	7	Уменьшение размера приложений с помощью обрезки
`DateOnly` и `TimeOnly`	8	Указание значений даты и времени
`EnsureCapacity` для `List<T>`	8	Повышение производительности за счет обеспечения емкости коллекции
Соглашения о настройке EF Core	10	Настройка предварительных моделей
Новые методы LINQ	11	Построение выражений LINQ с помощью класса `Enumerable`

Повышение производительности с .NET Core 2.0 до .NET 5

Компания Microsoft за последние несколько лет значительно улучшила производительность. Более подробную информацию вы можете прочитать на сайте https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/.

Проверка пакетов SDK для .NET на наличие обновлений

В .NET 6 включена команда для проверки версий .NET SDK и установленных сред выполнения и предупреждения о необходимости обновления. Например, вы вводите следующую команду:

```
dotnet sdk check
```

В результате вы получите следующий вывод, включая статус доступных обновлений:

```
.NET SDKs:
Version                              Status
---------------------------------------------------------------------------
3.1.412                              Up to date.
5.0.202                              Patch 5.0.206 is available.
...
```

Компоненты .NET

Платформа .NET Core состоит из нескольких частей.

- *Компиляторы языка* преобразуют ваш исходный код, написанный на таких языках, как C#, F# и Visual Basic, в код *промежуточного языка* (intermediate language, IL), хранящийся в сборках. В языке C# 6.0 и более поздних версиях компания Microsoft переключилась на компилятор с открытым исходным кодом, известный как Roslyn, который также используется в Visual Basic.
- *Общеязыковая исполняющая среда* (Common Language Runtime, CoreCLR) — загружает сборки, компилирует код на IL в нативный код для ЦПУ того компьютера, на котором запущено приложение, и выполняет этот код в среде, которая управляет такими ресурсами, как потоки и память.
- *Библиотеки базовых классов* (Base Class Libraries, BCL) *сборок в пакетах NuGet (CoreFX)* — это предварительно собранные сборки типов, упакованные и распространяемые с помощью NuGet для выполнения типовых задач при разработке

приложений. Вы можете использовать компоненты .NET Core для быстрого создания чего угодно, подобно тому как вы можете комбинировать детали LEGO™. Платформа .NET Core 2.0 реализовала .NET Standard 2.0 — расширенный набор всех предыдущих версий .NET Standard — и достигла паритета с .NET Framework и Xamarin. Платформа .NET Core 3.0 реализует .NET Standard 2.1, что добавляет новые возможности и позволяет повысить производительность по сравнению с функциональностью .NET Framework. Платформа .NET 6 реализует унифицированный BCL для всех типов приложений, включая мобильные.

Сборки, пакеты NuGet и пространства имен

Сборка — это место, где тип хранится в файловой системе. Сборки представляют собой механизм для развертывания кода. Например, сборка `System.Data.dll` содержит типы для управления данными. Чтобы использовать типы в других сборках, на них нужно сделать ссылку. Сборки могут быть статическими (предварительно созданными) или динамическими (генерируемыми во время выполнения). Динамические сборки — это расширенная функция, которую мы не будем рассматривать в этой книге. Сборки могут быть скомпилированы в один файл, как DLL (библиотека классов) или EXE (консольное приложение).

Сборки часто распространяются в виде *пакетов NuGet*, которые представляют собой файлы, скачиваемые из общедоступных онлайн-каналов, и могут содержать несколько сборок и других ресурсов. Вы также узнаете о *пакетах SDK для проектов, рабочих нагрузках и платформах*, которые представляют собой комбинации пакетов NuGet.

Канал Microsoft NuGet находится по следующему адресу: https://www.nuget.org/.

Пространства имен

Пространство имен — это адрес типа. Пространство имен представляет собой механизм уникальной идентификации типа, требующий полного адреса, а не просто короткого имени. В реальной жизни *Bob*, проживающий по адресу *34 Sycamore Street*, отличается от *Bob*, проживающего по адресу *12 Willow Drive*.

В среде .NET интерфейс `IActionFilter` пространства имен `System.Web.Mvc` отличается от интерфейса `IActionFilter` пространства имен `System.Web.Http.Filters`.

Зависимости между сборками

Если сборка компилируется в виде библиотеки классов (предоставляет типы другим сборкам), то ей присваивается расширение `.dll` (dynamic link library, библиотека динамической компоновки), и она не может выполняться автономно.

Если сборка компилируется как приложение, то ей присваивается расширение `.exe` (executable, *исполняемый* файл), и она может выполняться автономно. До выпуска версии .NET Core 3.0 консольные приложения компилировались в файлы `.dll` и должны были выполняться командой `dotnet run` или хостовым исполняемым файлом.

Любая сборка может ссылаться на одну или несколько сборок библиотеки классов, которые в этом случае станут ее зависимостями. Однако вы не можете использовать циклические ссылки. Поэтому сборка `B` не может ссылаться на сборку `A`, если та уже ссылается на сборку `B`. Компилятор оповестит вас, если вы попытаетесь добавить зависимость так, что при этом появится циклическая ссылка. Такие ссылки зачастую говорят о плохом коде. Если вы уверены, что вам нужна циклическая ссылка, то решить эту проблему можно с помощью интерфейса.

Платформа Microsoft .NET и пакет SDK

По умолчанию консольные приложения содержат ссылку на Microsoft .NET SDK. Эта специальная платформа содержит в пакетах NuGet тысячи типов, которые требуются при разработке практически всех приложений, например типы `System.Int32` и `System.String`.

При использовании .NET вы ссылаетесь на сборки-зависимости, пакеты NuGet и платформы, которые нужны вашему приложению в файле проекта.

Рассмотрим отношения между сборками и пространствами имен.

1. Откройте редактор кода и создайте рабочую область/решение `Chapter07`.
2. Создайте консольное приложение с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter07`;
   3) файл и папка проекта: `AssembliesAndNamespaces`.
3. Откройте файл `AssembliesAndNamespaces.csproj` и обратите внимание, что это типичный файл проекта для приложения .NET 6:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Пространства имен и типы в сборках

Наиболее распространенные типы .NET находятся в сборке `System.Runtime.dll`. Между сборками и пространствами имен не всегда существует однозначное соответствие. Одна сборка может содержать множество пространств имен, а пространство имен может содержаться во многих сборках. Вы можете увидеть связь между отдельными сборками и пространствами имен, для которых они предоставляют типы (табл. 7.5).

Таблица 7.5. Связь сборок и пространств имен

Сборки	Пример пространства имен	Пример типов
`System.Runtime.dll`	`System`, `System.Collections`, `System.Collections.Generic`	`Int32`, `String`, `IEnumerable<T>`
`System.Console.dll`	`System`	`Console`
`System.Threading.dll`	`System.Threading`	`Interlocked`, `Monitor`, `Mutex`
`System.Xml.Xdocument.dll`	`System.Xml.Linq`	`Xdocument`, `Xelement`, `Xnode`

Пакеты NuGet

Платформа .NET Core разделена на набор пакетов, которые распространяются с помощью технологии управления пакетами под названием NuGet, поддерживаемой Microsoft. Каждый из этих пакетов представляет собой одну сборку с таким же именем. Например, пакет `System.Collections` содержит сборку `System.Collections.dll`.

Преимущества пакетов таковы:

- можно легко распространять в общедоступных каналах;
- можно использовать повторно;
- поставка каждого из пакетов может производиться по собственному графику;
- пакеты могут быть протестированы независимо от других пакетов;
- могут поддерживать разные операционные системы и процессоры, так как могут включать несколько версий одной и той же сборки, созданной для разных операционных систем и процессоров;
- могут содержать зависимости, характерные только для одной библиотеки;
- приложения меньше, поскольку несвязанные пакеты не входят в состав дистрибутива.

В табл. 7.6 перечислены некоторые из наиболее значимых пакетов и их важные типы.

Таблица 7.6. Значимые пакеты и их важные типы

Пакеты	Важные типы
`System.Runtime`	`Object`, `String`, `Int32`, `Array`
`System.Collections`	`List<T>`, `Dictionary<Tkey, Tvalue>`
`System.NET.Http`	`HttpClient`, `HttpResponseMessage`
`System.IO.FileSystem`	`File`, `Directory`
`System.Reflection`	`Assembly`, `TypeInfo`, `MethodInfo`

Фреймворки

Существует двусторонняя связь между пакетами и фреймворками. Первые определяют API, а вторые группируют пакеты. Фреймворк без каких-либо пакетов не определит никакие API.

Каждый пакет .NET поддерживает набор фреймворков. Например, пакет `System.IO.FileSystem` версии 4.3.0 поддерживает следующие фреймворки:

- .NET Standard, версия 1.3 или более поздняя;
- .NET Framework, версия 4.6 или более поздняя;
- платформы Six Mono и Xamarin (например, Xamarin.iOS 1.0).

Более подробную информацию можно прочитать на сайте https://www.nuget.org/packages/System.IO.FileSystem/.

Импорт пространства имен в целях использования типа

Рассмотрим, как пространства имен связаны со сборками и типами.

1. В проекте `AssembliesAndNamespaces` в файле `Program.cs` введите следующий код:

```
XDocument doc = new();
```

2. Соберите проект и обратите внимание на сообщение компилятора об ошибке:

```
The type or namespace name 'XDocument' could not be found (are you missing
a using directive or an assembly reference?)
```

Тип `XDocument` не распознается, поскольку мы не сообщили компилятору пространство имен для этого типа. Хотя в данном проекте уже есть ссылка на сборку, содержащую тип, нам также необходимо либо добавить к имени типа пространство имен как префикс, либо импортировать пространство имен.

3. Щелкните кнопкой мыши на имени класса `XDocument`. Ваш редактор кода отобразит значок в виде лампочки, тем самым показывая, что программа распознает тип и может автоматически решить проблему.
4. Щелкните на значке лампочки и выберите в меню пункт using System.Xml.Linq;.

Таким образом вы *импортировали пространство имен*, добавив оператор `using` в начало файла. Как только пространство имен импортируется в начале файла с кодом, все типы в пространстве имен становятся доступными для использования в этом файле. Для этого достаточно указать имя типа, не прибегая к необходимости полностью его квалифицировать, задействуя префикс его пространства имен.

Иногда после импорта пространства имен рекомендуется добавлять комментарий с именем типа, чтобы напомнить себе, почему нужно импортировать это пространство:

```
using System.Xml.Linq; // XDocument
```

Связь ключевых слов языка C# с типами .NET

Один из распространенных вопросов, которые я получаю от неопытных программистов C#, звучит так: *«Отличаются ли чем-то типы string (строчная буква в начале) и String (прописная буква в начале)?»*

Короткий ответ — нет. Полный ответ заключается в том, что все ключевые слова C#, например, `string` или `int`, представляющие собой названия типов, — это псевдонимы для типа .NET в сборке библиотеки классов.

Когда вы используете ключевое слово `string`, компилятор преобразует его в тип `System.String`. А тип `int` он преобразует в тип `System.Int32`.

Рассмотрим пример.

1. В файле `Program.cs` объявите две переменные для хранения значений `string`: название первой начинается со строчной буквы, а название второй — с прописной:

```
string s1 = "Hello";
String s2 = "World";

WriteLine($"{s1} {s2}");
```

2. Запустите код и обратите внимание, что на данный момент обе переменные работают одинаково хорошо и имеют одинаковый смысл.

3. В файле `AssembliesAndNamespaces.csproj` добавьте записи, чтобы предотвратить глобальный импорт пространства имен `System`:

```
<ItemGroup>
  <Using Remove="System" />
</ItemGroup>
```

4. В файле `Program.cs` обратите внимание на сообщение компилятора об ошибке:

```
The type or namespace name 'String' could not be found (are you missing
a using directive or an assembly reference?)
```

5. В начале файла `Program.cs` импортируйте пространство имен `System` с оператором `using`, который исправит ошибку:

```
using System; // String
```

По возможности вместо фактического типа используйте ключевое слово C#, поскольку ключевые слова не нуждаются в импорте пространства имен.

Сопоставление псевдонимов C# с типами .NET

В табл. 7.7 приведены 18 ключевых слов C#, представляющих собой названия типов, и их фактические типы .NET.

Таблица 7.7. Ключевые слова C# и их фактические типы .NET

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
nint	System.IntPtr	nuint	System.UIntPtr
float	System.Single	double	System.Double
decimal	System.Decimal	bool	System.Boolean
object	System.Object	dynamic	System.Dynamic.DynamicObject

Другие компиляторы языков программирования .NET могут выполнять то же самое. Например, язык Visual Basic .NET содержит тип `Integer` — псевдоним для `System.Int32`.

Целые числа собственного размера

В C# 9 введены псевдонимы ключевых слов `nint` и `nuint` для *целых чисел собственного размера*, что означает, что размер хранилища для целочисленного значения зависит от платформы. Они хранят 32-битное целое число в 32-битном процессе, а операция `sizeof()` возвращает 4 байта; они хранят 64-битное целое число в 64-битном процессе, а `sizeof()` возвращает 8 байт. Псевдонимы представляют собой указатели на целочисленное значение в памяти, поэтому их имена .NET следующие: `IntPtr` и `UIntPtr`. Фактический тип хранилища будет либо `System.Int32`, либо `System.Int64`, в зависимости от процесса.

В 64-битном процессе содержится следующий код:

```
WriteLine($"int.MaxValue = {int.MaxValue:N0}");
WriteLine($"nint.MaxValue = {nint.MaxValue:N0}");
```

Результат его выполнения таков:

```
int.MaxValue = 2,147,483,647
nint.MaxValue = 9,223,372,036,854,775,807
```

Выявление расположения типа

Редакторы кода предоставляют встроенную документацию для типов .NET. Рассмотрим пример.

1. Щелкните правой кнопкой мыши на слове `XDocument` и выберите пункт меню Go to Definition (Перейти к определению).
2. Перейдите к началу файла и обратите внимание, что имя файла сборки — `System.Xml.XDocument.dll`, однако класс находится в пространстве имен `System.Xml.Linq` (рис. 7.1).

Рис. 7.1. Сборка и пространство имен, содержащие тип XDocument

3. Закройте вкладку XDocument [from metadata].
4. Щелкните правой кнопкой мыши на слове `string` или `String` и выберите пункт меню Go to Definition (Перейти к определению).

5. Перейдите к началу файла и обратите внимание, что имя файла сборки — `System.Runtime.dll`, однако класс находится в пространстве имен `System`.

В действительности ваш редактор кода технически передает вам неверную информацию. Если вы помните, в момент написания кода в главе 2, раскрывая объем словаря C#, мы обнаружили, что сборка `System.Runtime.dll` не содержит типов.

То, что она содержит, — это переадресаторы типов. Это специальные типы, в отношении которых кажется, что они существуют в сборке, однако на самом деле они реализованы в другом месте. В этом случае они реализуются глубоко внутри среды выполнения .NET с помощью высокооптимизированного кода.

Использование кода с устаревшими платформами, используя .NET Standard

До выпуска .NET Standard существовали *переносимые библиотеки классов* (Portable Class Libraries, PCL). С их помощью можно было создать библиотеку кода и явно указать, какие платформы должна поддерживать библиотека, например Xamarin, Silverlight и Windows 8. Ваша библиотека могла бы затем использовать пересечение API, поддерживаемых указанными платформами.

Сотрудники компании Microsoft поняли, что это нестабильный подход, и разработали .NET Standard — единый API, который будет поддерживаться всеми будущими платформами .NET. Существуют более старые версии .NET Standard, но только .NET Standard 2.0 и более поздние версии поддерживаются несколькими платформами .NET. Версия .NET Standard 2.1 была выпущена в конце 2019 года, но только .NET Core 3.0 и версия Xamarin того года поддерживают его новые функции. Далее я буду использовать термин .NET Standard для обозначения .NET Standard 2.0.

Идейно .NET Standard очень похож на HTML5 — это стандарты, которые должна поддерживать платформа. Так же как браузеры Google Chrome и Microsoft Edge реализуют стандарт HTML5, .NET Core, .NET Framework и Xamarin реализуют .NET Standard. Если вы хотите создать библиотеку типов, которая будет работать с различными версиями устаревшей .NET, то проще всего сделать это с помощью .NET Standard.

Многие API, добавленные в .NET Standard 2.1, требуют изменений в среде выполнения, а .NET Framework — устаревшая платформа Microsoft, которая должна оставаться неизменной, насколько это возможно. Поэтому платформа .NET Framework 4.8 останется на .NET Standard 2.0, а не на реализации NET Standard 2.1. Если вам необходимо поддерживать клиентов, работающих с .NET Framework, то следует разрабатывать библиотеки классов в .NET Standard 2.0, даже если это не самая последняя версия и не поддерживается все последние версии языка и новые функции BCL.

Выбор версии .NET Standard сводится к балансу между максимальным количеством поддерживаемых платформ и доступной функциональностью. Более низкая версия поддерживает больше платформ, но имеет меньший набор API. С более высокой версией ситуация обратная: меньше платформ, но больший набор API. Как правило, вы должны выбирать наименьшую версию, поддерживающую все необходимые вам API.

Общие сведения о значениях по умолчанию для библиотек классов с различными пакетами SDK

При использовании инструмента SDK `dotnet` для создания библиотеки классов полезно знать, какая целевая платформа будет использоваться по умолчанию (табл. 7.8).

Таблица 7.8. Целевые платформы

SDK	Целевая платформа по умолчанию для новых библиотек классов
.NET Core 3.1	`netstandard2.0`
.NET 5	`net5.0`
.NET 6	`net6.0`

Конечно, тот факт, что библиотека классов по умолчанию нацелена на определенную версию .NET, не означает, что вы не можете изменить ее после того, как будет создан проект библиотеки классов с помощью шаблона по умолчанию.

Вы можете вручную установить для целевой платформы значение, поддерживающее проекты, которым необходимо ссылаться на эту библиотеку (табл. 7.9).

Таблица 7.9. Целевые платформы библиотеки классов

Целевая платформа библиотеки классов	Может использоваться целевыми проектами
`netstandard2.0`	.NET Framework 4.6.1 или более поздних версий, .NET Core 2.0 или более поздних версий, .NET 5.0 или более поздних версий, Mono 5.4 или более поздних версий, Xamarin.Android 8.0 или более поздних версий, Xamarin.iOS 10.14 или более поздних версий
`netstandard2.1`	.NET Core 3.0 или более поздних версий, .NET 5.0 или более поздних версий, Mono 6.4 или более поздних версий, Xamarin. Android 10.0 или более поздних версий, Xamarin.iOS 12.16 или более поздних версий
`net5.0`	.NET 5.0 или более поздних версий
`net6.0`	.NET 6.0 или более поздних версий

Всегда проверяйте целевую структуру библиотеки классов, а затем при необходимости вручную меняйте ее на более подходящую. Проанализируйте и примите решение о том, какой она должна быть.

Создание библиотеки классов .NET Standard 2.0

Мы разработаем библиотеку классов с помощью .NET Standard 2.0, чтобы ее можно было использовать на всех важных устаревших платформах .NET и кроссплатформенно в операционных системах Windows, macOS и Linux, а также чтобы иметь доступ к широкому набору API платформы .NET.

1. Откройте редактор кода и создайте библиотеку классов `SharedLibrary` в рабочей области/решении `Chapter07`.
2. Если вы используете Visual Studio 2022, то в качестве целевой платформы выберите .NET Standard 2.0, а затем настройте стартовый проект для решения в соответствии с текущим выбором.
3. Если вы используете Visual Studio Code, то установите переключатель для целевой версии .NET Standard 2.0 следующим образом:

```
dotnet new classlib -f netstandard2.0
```

4. Если вы используете Visual Studio Code, то выберите `SharedLibrary` в качестве активного проекта OmniSharp.

Если вам нужно создать типы, которые используют новые функции .NET 6.0, а также типы, использующие только функции .NET Standard 2.0, то вы можете создать две отдельные библиотеки классов: одну — для .NET Standard 2.0, а вторую — для .NET 6.0. Эту тему мы будем изучать в главе 10.

Альтернатива ручному созданию двух библиотек классов — это создание одной библиотеки, имеющей многоцелевое назначение. Если вы хотите, чтобы я добавил раздел о многоцелевом назначении в следующее издание, то сообщите мне об этом. Более подробно о многоцелевом назначении вы можете прочитать на сайте https://docs.microsoft.com/en-us/dotnet/standard/library-guidance/cross-platform-targeting#multi-targeting.

Управление пакетом SDK для .NET

По умолчанию при выполнении команд `dotnet` используется самый последний установленный пакет SDK для .NET. Но может возникнуть необходимость контролировать, какой SDK используется.

Например, один пользователь, прочитав четвертое издание, хотел, чтобы его действия полностью соответствовали описанным в книге рекомендациям с учетом использования пакета SDK для .NET Core 3.1. Но этот человек также установил пакет SDK для .NET 5.0 и использовал его по умолчанию. Как было описано выше, при создании новых библиотек классов поведение было изменено на целевую версию .NET 5.0 вместо .NET Standard 2.0, что сбило пользователя с толку.

Вы можете управлять пакетом SDK для .NET, используемым по умолчанию, с помощью файла `global.json`. Команда `dotnet` выполняет поиск файла `global.json` в текущей папке и папках-предках.

1. Создайте подкаталог/подпапку ControlSDK в папке рабочей области `Chapter07`.
2. В Windows запустите Command Prompt (Командная строка) или Windows Terminal (Терминал Windows). В macOS используйте Terminal (Терминал). Если вы используете Visual Studio Code, то можете использовать встроенный терминал.
3. В папке ControlSDK в командной строке или терминале введите команду для создания файла `global.json`, который принудительно будет использовать последнюю версию .NET Core 3.1 SDK:

```
dotnet new globaljson --sdk-version 3.1.412
```

4. Откройте файл `global.json` и проанализируйте его содержимое:

```
{
  "sdk": {
    "version": "3.1.412"
  }
}
```

Номера версий последних пакетов SDK для .NET можно найти в таблице на сайте https://dotnet.microsoft.com/download/visual-studio-sdks.

5. В папке ControlSDK в командной строке или терминале введите команду для создания проекта библиотеки классов:

```
dotnet new classlib
```

6. Если у вас не установлен пакет SDK для .NET Core 3.1, то вы увидите сообщение об ошибке:

```
Could not execute because the application was not found or a compatible
.NET SDK is not installed.
```

7. Если у вас установлен пакет SDK для .NET Core 3.1, будет создан проект библиотеки классов, ориентированный на .NET Standard 2.0 по умолчанию.

Выполнять описанные выше шаги нет необходимости, но если вы хотите попрактиковать и еще не установили пакет SDK для .NET Core 3.1, то можете установить его, используя следующую ссылку: https://dotnet.microsoft.com/download/dotnet/3.1.

Публикация и развертывание ваших приложений

Если вы пишете роман и хотите, чтобы его прочитали другие люди, вам необходимо опубликовать его.

Большинство разработчиков пишут код, с помощью которого другие разработчики могут писать свои программы или который пользователи будут запускать в качестве приложения. Чтобы это произошло, необходимо опубликовать свой код в виде упакованных библиотек классов или исполняемых приложений.

Существует три способа публикации и развертывания приложения .NET:

1) *платформенно-зависимое развертывание* (framework-dependent deployment, FDD);
2) *зависящие от платформы исполняемые файлы* (framework-dependent executables, FDE);
3) автономное (self-contained) развертывание.

Если вы решили развернуть свое приложение и его зависимые пакеты, но не саму .NET, то вы будете полагаться на то, что эта платформа уже установлена на целевом компьютере. Это хорошо работает для веб-приложений, развернутых на сервере, поскольку .NET и множество других веб-приложений, вероятно, уже находятся на сервере.

Развертывание, зависящее от платформы (FDD), означает, что вы разворачиваете библиотеку DLL, которая должна выполняться инструментом командной строки `dotnet`. *Исполняемые файлы, зависящие от платформы* (FDE), означают, что вы разворачиваете EXE-файл, который можно запускать непосредственно из командной строки. В обоих вариантах требуется установка .NET.

Иногда вам необходимо передать USB-накопитель с вашим приложением другому пользователю и быть уверенными, что оно станет работать на другом компьютере. Вы хотите выполнить автономное развертывание. Даже если размер файлов развертывания больше, вы будете уверены, что все будет работать.

Разработка консольного приложения для публикации

Рассмотрим, как опубликовать консольное приложение.

1. Откройте редактор кода и создайте консольное приложение `DotNetCoreEverywhere` в рабочей области/решении `Chapter07`.
2. В Visual Studio Code выберите `DotNetEverywhere` в качестве активного проекта OmniSharp. Когда вы увидите всплывающее предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку Yes (Да), чтобы добавить их.
3. В файле `Program.cs` удалите комментарий и статически импортируйте класс `Console`.
4. В файле `Program.cs` добавьте оператор для вывода сообщения о том, что консольное приложение может работать везде, и информацию об операционной системе:

```
WriteLine("I can run everywhere!");

WriteLine($"OS Version is {Environment.OSVersion}.");

if (OperatingSystem.IsMacOS())
{
  WriteLine("I am macOS.");
}
else if (OperatingSystem.IsWindowsVersionAtLeast(major: 10))
{
  WriteLine("I am Windows 10 or 11.");
}
else
{
  WriteLine("I am some other mysterious OS.");
}

WriteLine("Press ENTER to stop me.");
ReadLine();
```

5. Откройте файл `DotNetEverywhere.csproj` и добавьте идентификаторы среды выполнения для трех операционных систем внутри элемента `<PropertyGroup>`, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
```

```
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RuntimeIdentifiers>
      win10-x64;osx-x64;osx.11.0-arm64;linux-x64;linux-arm64
    </RuntimeIdentifiers>
  </PropertyGroup>

</Project>
```

- Идентификатор `win10-x64` обозначает 64-битную версию Windows 10 или Windows Server 2016. Вы также можете использовать значение RID `win10-arm64` для развертывания на Microsoft Surface Pro X.
- Идентификатор `osx-x64` обозначает macOS Sierra 10.12 или более позднюю версию. Вы также можете указать значения RID для конкретной версии, таких как `osx.10.15-x64` (Catalina), `osx.11.0-x64` (Big Sur на Intel) или `osx.11.0-arm64` (Big Sur на Apple Silicon).
- Идентификатор `linux-x64` обозначает большинство настольных дистрибутивов Linux, таких как Ubuntu, CentOS, Debian или Fedora. Используйте `linux-arm` для 32-битной ОС Raspbian или Raspberry Pi. Используйте `linux-arm64` для Raspberry Pi с 64-битной версией Ubuntu.

Команды dotnet

Установка .NET SDK включает в себя *интерфейс командной строки* (command-line interface, CLI) `dotnet`.

Создание новых проектов

Интерфейс командной строки `dotnet` содержит команды, которые работают с текущей папкой в целях создания нового проекта с помощью шаблонов.

1. В Windows запустите Command Prompt (Командная строка) или Windows Terminal (Терминал Windows). В macOS — Terminal (Терминал). Если вы используете Visual Studio Code, то можете задействовать встроенный терминал.
2. Введите команду `dotnet new --list` или `dotnet new -l`, чтобы вывести список установленных шаблонов (рис. 7.2).

Большинство параметров командной строки dotnet имеют длинную и короткую версии. Например, --list или -l. Короткие тексты набираются быстрее, но с большей вероятностью могут быть неверно истолкованы вами или другими пользователями. Иногда необходим более четкий ввод текста.

```
Windows PowerShell
PS C:\Users\markj> dotnet new -l
Template Name                                  Short Name          Language    Tags
---------------------------------------------  ------------------  ----------  ---------------------
Console Application                            console             [C#],F#,VB  Common/Console
Class Library                                  classlib            [C#],F#,VB  Common/Library
WPF Application                                wpf                 [C#],VB     Common/WPF
WPF Class Library                              wpflib              [C#],VB     Common/WPF
WPF Custom Control Library                     wpfcustomcontrollib [C#],VB     Common/WPF
WPF User Control Library                       wpfusercontrollib   [C#],VB     Common/WPF
Windows Forms App                              winforms            [C#],VB     Common/WinForms
Windows Forms Control Library                  winformscontrollib  [C#],VB     Common/WinForms
Windows Forms Class Library                    winformslib         [C#],VB     Common/WinForms
Worker Service                                 worker              [C#],F#     Common/Worker/Web
MSTest Test Project                            mstest              [C#],F#,VB  Test/MSTest
NUnit 3 Test Item                              nunit-test          [C#],F#,VB  Test/NUnit
NUnit 3 Test Project                           nunit               [C#],F#,VB  Test/NUnit
xUnit Test Project                             xunit               [C#],F#,VB  Test/xUnit
Razor Component                                razorcomponent      [C#]        Web/ASP.NET
Razor Page                                     page                [C#]        Web/ASP.NET
MVC ViewImports                                viewimports         [C#]        Web/ASP.NET
MVC ViewStart                                  viewstart           [C#]        Web/ASP.NET
Blazor Server App                              blazorserver        [C#]        Web/Blazor
Blazor WebAssembly App                         blazorwasm          [C#]        Web/Blazor/WebAssembly
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                           webapp              [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
Razor Class Library                            razorclasslib       [C#]        Web/Razor/Library
ASP.NET Core Web API                           webapi              [C#],F#     Web/WebAPI
ASP.NET Core gRPC Service                      grpc                [C#]        Web/gRPC
dotnet gitignore file                          gitignore                       Config
global.json file                               globaljson                      Config
NuGet Config                                   nugetconfig                     Config
Dotnet local tool manifest file                tool-manifest                   Config
Web Config                                     webconfig                       Config
Solution File                                  sln                             Solution
Protocol Buffer File                           proto                           Web/gRPC
```

Рис. 7.2. Список установленных шаблонов новых проектов

Получение информации о платформе .NET и ее окружении

Важно видеть, какие пакеты SDK и среды выполнения для .NET установлены в настоящее время, а также информацию об операционной системе. Это можно узнать с помощью команды:

```
dotnet --info
```

Обратите внимание на результат:

```
.NET SDK (reflecting any global.json):
  Version:  6.0.100
  Commit:   22d70b47bc

Runtime Environment:
 OS Name:     Windows
 OS Version:  10.0.19043
```

```
 OS Platform: Windows
 RID:         win10-x64
 Base Path:   C:\Program Files\dotnet\sdk\6.0.100\

Host (useful for support):
  Version: 6.0.0
  Commit:  91ba01788d

.NET SDKs installed:
  3.1.412 [C:\Program Files\dotnet\sdk]
  5.0.400 [C:\Program Files\dotnet\sdk]
  6.0.100 [C:\Program Files\dotnet\sdk]

.NET runtimes installed:
  Microsoft.AspNetCore.All 2.1.29 [...\dotnet\shared\Microsoft.AspNetCore.All]
...
```

Управление проектами

Интерфейс командной строки `dotnet` содержит команды, управляющие проектом в текущей папке:

- `dotnet restore` — скачивает зависимости проекта;
- `dotnet build` — собирает, или компилирует, проект;
- `dotnet test` — собирает и запускает модульные тесты проекта;
- `dotnet run` — собирает и запускает проект;
- `dotnet pack` — создает пакет NuGet для проекта;
- `dotnet publish` — компилирует, а затем публикует проект с зависимостями или как отдельное приложение;
- `dotnet add` — добавляет ссылку на пакет или библиотеку классов в проект;
- `dotnet remove` — удаляет ссылку на пакет или библиотеку классов из проекта;
- `dotnet list` — перечисляет ссылки на пакет или библиотеки классов для проекта.

Публикация автономного приложения

Теперь, когда вы ознакомились с некоторыми примерами команд инструмента `dotnet`, мы можем опубликовать наше кросс-платформенное консольное приложение.

1. С помощью командной строки убедитесь, что находитесь в папке DotNetEverywhere.

2. Чтобы собрать и опубликовать версию выпуска консольного приложения для операционной системы Windows 10, введите следующую команду:

```
dotnet publish -c Release -r win10-x64
```

3. Обратите внимание, что механизм сборки восстанавливает все необходимые пакеты, компилирует исходный код проекта в DLL сборки и создает папку публикации:

```
Microsoft (R) Build Engine version 17.0.0+073022eb4 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  Restored C:\Code\Chapter07\DotNetEverywhere\DotNetEverywhere.csproj (in
46.89 sec).
  DotNetEverywhere -> C:\Code\Chapter07\DotNetEverywhere\bin\Release\
net6.0\win10-x64\DotNetEverywhere.dll
  DotNetEverywhere -> C:\Code\Chapter07\DotNetEverywhere\bin\Release\
net6.0\win10-x64\publish\
```

4. Введите следующие команды, чтобы собрать и опубликовать версии выпуска для операционных систем macOS и Linux:

```
dotnet publish -c Release -r osx-x64
dotnet publish -c Release -r osx.11.0-arm64
dotnet publish -c Release -r linux-x64
dotnet publish -c Release -r linux-arm64
```

Вы можете автоматизировать эти команды с помощью языка сценариев, например, PowerShell и выполнять их в любой операционной системе, используя кросс-платформенное ядро PowerShell. Просто создайте файл с расширением .ps1 и пятью командами. Затем выполните файл. Более подробно о PowerShell вы можете прочитать на этом сайте: https://github.com/markjprice/cs10dotnet6/tree/main/docs/powershell.

5. Откройте окно Finder (Файловый менеджер) в macOS или File Explorer (Проводник) Windows, перейдите к `DotNetEverywhere\bin\Release\net6.0` и обратите внимание на созданные в итоге папки для различных операционных систем.
6. В папке win10-x64 выберите папку publish, обратите внимание на все поддерживающие сборки, например `Microsoft.Csharp.dll`.
7. Выберите исполняемый файл `DotNetEverywhere` и обратите внимание, что его размер составляет 161 Кбайт (рис. 7.3).

Рис. 7.3. Исполняемый файл DotNetEverywhere для Windows 10 64-bit

8. Если вы работаете в операционной системе Windows, то дважды щелкните на исполняемом файле и обратите внимание на результат:

```
I can run everywhere!
OS Version is Microsoft Windows NT 10.0.19042.0.
I am Windows 10.
Press ENTER to stop me.
```

9. Обратите внимание, что размер папки `publish` и всех ее файлов составляет около 64,8 Мбайт.
10. В папке `osx.11.0-arm64` выберите папку `publish`, отметьте все вспомогательные сборки, а затем выберите исполняемый файл `DotNetEverywhere` и обратите внимание, что его размер составляет 126 Кбайт, а размер папки `publish` — 71,8 Мбайт.

Если вы скопируете любую из этих папок `publish` в соответствующую операционную систему, то запустится консольное приложение — это работает, так как мы создали автономное приложение .NET. Например, в macOS с Intel результаты будут выглядеть следующим образом:

```
I can run everywhere!
OS Version is Unix 11.2.3
I am macOS.
Press ENTER to stop me.
```

В этом примере использовалось консольное приложение, но вы могли легко создать сайт или веб-сервис ASP.NET Core, а также приложение Windows Forms или WPF. Конечно, вы можете развертывать настольные приложения Windows только на компьютерах с Windows, но не на Linux или macOS.

Публикация однофайлового приложения

Чтобы опубликовать однофайловое приложение, при публикации вы можете указать флаги. В .NET 5 однофайловые приложения были в первую очередь ориентированы на Linux, поскольку как в Windows, так и в macOS существуют ограничения, которые означают, что настоящая однофайловая публикация технически невозможна. Выпуск .NET 6 позволяет создавать правильные однофайловые приложения в Windows.

Если вы можете предположить, что .NET 6 уже установлена на компьютере, на котором вы хотите запустить свое приложение, то при публикации своего приложения для выпуска можете использовать дополнительные флаги, чтобы указать, что оно не должно быть автономным и что вы хотите опубликовать как один файл (если возможно), как показано в следующей команде (которую необходимо ввести в одной строке):

```
dotnet publish -r win10-x64 -c Release --self-contained=false
/p:PublishSingleFile=true
```

В результате будут созданы два файла: `DotNetEverywhere.exe` и `DotNetEverywhere.pdb`. Файл с расширением `.exe` — исполняемый. Файл с расширением `.pdb` представляет собой файл *базы данных отладки программы*, в котором хранится отладочная информация.

Для опубликованных приложений в macOS отсутствует расширение файла .exe, поэтому, если вы используете osx-x64 в приведенной выше команде, имя файла не будет иметь расширения.

Если вам необходимо, чтобы файл `.pdb` был встроен в исполняемый файл, добавьте элемент `<DebugType>` к элементу `<PropertyGroup>` в вашем файле `.csproj` и установите его как встроенный, как показано ниже:

```
<PropertyGroup>

  <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RuntimeIdentifiers>
      win10-x64;osx-x64;osx.11.0-arm64;linux-x64;linux-arm64
    </RuntimeIdentifiers>
  <DebugType>embedded</DebugType>

</PropertyGroup>
```

Если вы не можете предположить, что .NET 6 установлена на компьютере, то, хотя Linux также генерирует лишь два файла, ожидайте следующих дополнительных файлов для Windows: `coreclr.dll`, `clrjit.dll`, `clrcompression.dll` и `mscordaccore.dll`.

Рассмотрим пример для операционной системы Windows.

1. В командной строке введите команду, чтобы создать версию выпуска консольного приложения для операционной системы Windows 10:

```
dotnet publish -c Release -r win10-x64 /p:PublishSingleFile=true
```

2. Перейдите в папку DotNetEverywhere\bin\Release\net6.0\win10-x64\publish, выберите исполняемый файл `DotNetEverywhere` и обратите внимание, что размер исполняемого файла теперь составляет около 58,3 Мбайт, а также имеется файл `.pdb` размером 10 Кбайт. В вашей системе размер файлов будет отличаться.

Уменьшение размера приложений с помощью обрезки

Одна из проблем при развертывании приложения .NET как автономного заключается в том, что библиотеки .NET занимают слишком много места. Больше всего в уменьшении размера нуждаются компоненты Blazor WebAssembly, так как все библиотеки .NET необходимо загрузить в браузер.

К счастью, вы можете уменьшить этот размер, не упаковывая неиспользуемые сборки в свои развертывания. Система обрезки приложений, представленная в .NET Core 3.0, может определять сборки, необходимые для вашего кода, и удалять ненужные.

В .NET 5 обрезка удаляла отдельные типы и даже члены, такие как методы, из сборки, если они не использовались. Например, при работе с консольным приложением Hello World сборка `System.Console.dll` сокращена с 61,5 до 31,5 Кбайт. Для .NET 5 это экспериментальная функция, поэтому не используйте ее по умолчанию.

В .NET 6 Microsoft добавила аннотации в свои библиотеки, чтобы указать, каким образом их можно безопасно обрезать, поэтому обрезка типов и членов стандартная. Этот механизм называется *режимом обрезки ссылок*.

Проблема в том, насколько хорошо обрезка идентифицирует неиспользуемые сборки, типы и элементы. Если ваш код динамический, возможно использующий отражение, то может работать некорректно, поэтому Microsoft также допускает ручное управление.

Включение обрезки на уровне сборки

Существует два способа включить обрезку на уровне сборки.

Первый — добавить элемент в файл проекта:

```
<PublishTrimmed>true</PublishTrimmed>
```

Второй — добавить при публикации флаг:

```
dotnet publish ... -p:PublishTrimmed=True
```

Включение обрезки на уровне типов и членов

Существует два способа включить обрезку на уровне типов и элементов.

Первый — добавить два элемента в файл проекта:

```
<PublishTrimmed>true</PublishTrimmed>
<TrimMode>Link</TrimMode>
```

Второй — добавить при публикации два флага:

```
dotnet publish ... -p:PublishTrimmed=True -p:TrimMode=Link
```

Для .NET 6 режим обрезки ссылок используется по умолчанию, поэтому вам необходимо установить переключатель только в том случае, если вы хотите установить альтернативный режим обрезки, такой как `copyused`, что означает обрезку на уровне сборки.

Декомпиляция сборок .NET

Один из лучших способов научиться программировать для .NET — наблюдать, как это делают профессионалы.

Вы можете декомпилировать сборки других разработчиков и в целях, не связанных с обучением, таких как копирование их кода для использования в вашей собственной производственной библиотеке или приложении. Но всегда помните, что пользуетесь чьей-то интеллектуальной собственностью, и относитесь к этому с уважением.

Декомпиляция с помощью расширения ILSpy для Visual Studio 2022

В целях обучения вы можете декомпилировать любую сборку .NET с помощью такого инструмента, как *ILSpy*.

1. В Visual Studio 2022 для Windows выберите команду меню Extensions ▸ Manage Extensions (Расширения ▸ Управление расширениями).
2. В поле поиска введите `ilspy`.
3. Для расширения ILSpy нажмите Download (Скачать).
4. Нажмите кнопку Close (Закрыть).
5. Закройте Visual Studio, чтобы разрешить установку расширения.
6. Перезапустите Visual Studio и снова откройте решение `Chapter07`.
7. На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте DotNetEverywhere и выберите Open output in ILSpy (Открыть вывод в ILSpy).

8. Выберите команду меню File ▸ Open (Файл ▸ Открыть).
9. Выберите папку Code/Chapter07/DotNetEverywhere/bin/Release/net6.0/linux-x64.
10. Выберите сборку System.IO.FileSystem.dll и нажмите кнопку Open (Открыть).
11. В дереве сборок разверните сборку System.IO.FileSystem, разверните пространство имен System.IO, выберите класс `Directory` и дождитесь его декомпиляции.
12. В классе `Directory` щелкните на [+], чтобы развернуть метод `GetParent` (рис. 7.4).

Рис. 7.4. Декомпилированный метод GetParent класса Directory в Windows

13. Обратите внимание на проверку параметра `path` и генерации `ArgumentNullException`, если он имеет значение `null`, или `ArgumentException`, если он имеет нулевую длину.
14. Закройте ILSpy.

Декомпиляция с помощью расширения ILSpy для Visual Studio Code

Аналогичная возможность доступна на разных платформах в качестве расширения для Visual Studio Code.

1. Если вы еще не установили расширение *ILSpy .NET Decompiler* для Visual Studio Code, то найдите его и установите.
2. В macOS или Linux расширение зависит от Mono, поэтому вам также потребуется установить Mono, используя следующую ссылку: https://www.mono-project.com/download/stable/.

3. В программе Visual Studio Code выберите команду меню View ▸ Command Palette (Вид ▸ Панель команд).
4. Введите `ilspy`, а затем выберите ILSpy: Decompile IL Assembly (pick file).
5. Перейдите в папку Code/Chapter07/DotNetEverywhere/bin/Release/net6.0/linux-x64.
6. Выберите сборку System.IO.FileSystem.dll и нажмите пункт Select assembly (Выбрать сборку). Ничего не произойдет, но вы можете убедиться, что ILSpy работает, проанализировав окно Output (Вывод), выбрав ilspy-vscode в раскрывающемся списке и наблюдая за процессом (рис. 7.5).

Рис. 7.5. Вывод расширения ILSpy при выборе сборки для декомпиляции

7. На панели EXPLORER (Проводник) разверните ILSPY DECOMPILED MEMBERS (Декомпилированные члены ILSpy), выберите сборку, закройте окно Output (Вывод) и обратите внимание на два окна редактирования, в которых отображаются атрибуты текущей сборки, для чего используется язык C#, и ссылки на внешние DLL и сборки, для чего используется IL (рис. 7.6).
8. В коде IL справа обратите внимание на ссылку на сборку `System.Runtime`, включая номер версии, как показано ниже:

```
.module extern libSystem.Native
.assembly extern System.Runtime
{
  .publickeytoken = (
    b0 3f 5f 7f 11 d5 0a 3a
  )
  .ver 6:0:0:0
}
```

Рис. 7.6. Разворачивание ILSPY DECOMPILED MEMBERS

Элемент `.module extern libSystem.Native` означает, что эта сборка выполняет вызовы функций системных API Linux, как и следовало ожидать от кода, взаимодействующего с файловой системой. Если бы мы декомпилировали эквивалент этой сборки для Windows, то вместо нее использовался бы `.module extern kernel32.dll` — Win32 API.

9. На панели EXPLORER (Проводник) в ILSPY DECOMPILED MEMBERS (Декомпилированные члены ILSpy) разверните сборку, затем пространство имен `System.IO`, выберите пункт меню Directory (Каталог) и обратите внимание на два открывающихся окна редактирования, в которых отображается декомпилированный класс `Directory` как на языке C# (слева), так и на IL (справа) (рис. 7.7).

Рис. 7.7. Декомпилированный класс Directory в коде C# и IL

10. Сравните исходный код C# метода `GetParent`...

```
public static DirectoryInfo? GetParent(string path)
{
  if (path == null)
  {
    throw new ArgumentNullException("path");
  }
  if (path.Length == 0)
  {
    throw new ArgumentException(SR.Argument_PathEmpty, "path");
  }
  string fullPath = Path.GetFullPath(path);
  string directoryName = Path.GetDirectoryName(fullPath);
  if (directoryName == null)
  {
    return null;
  }
  return new DirectoryInfo(directoryName);
}
```

11. ...С эквивалентным исходным кодом метода `GetParent` на IL:

```
.method /* 06000067 */ public hidebysig static
  class System.IO.DirectoryInfo GetParent (
    string path
  ) cil managed
{
  .param [0]
    .custom instance void System.Runtime.CompilerServices
    .NullableAttribute::.ctor(uint8) = (
      01 00 02 00 00
    )
  // метод начинается с RVA 0x62d4
  // размер 64 (0x40)
  .maxstack 2
  .locals /* 1100000E */ (
    [0] string,
    [1] string
  )

  IL_0000: ldarg.0
  IL_0001: brtrue.s IL_000e

  IL_0003: ldstr "path" /* 700005CB */
  IL_0008: newobj instance void [System.Runtime]
    System.ArgumentNullException::.ctor(string) /* 0A000035 */
  IL_000d: throw

  IL_000e: ldarg.0
  IL_000f: callvirt instance int32 [System.Runtime]
    System.String::get_Length() /* 0A000022 */
```

```
    IL_0014: brtrue.s IL_0026
    IL_0016: call string System.SR::get_Argument_PathEmpty() /* 0600004C */
    IL_001b: ldstr "path" /* 700005CB */
    IL_0020: newobj instance void [System.Runtime]
      System.ArgumentException::.ctor(string, string) /* 0A000036 */
    IL_0025: throw IL_0026: ldarg.0
    IL_0027: call string [System.Runtime.Extensions]
      System.IO.Path::GetFullPath(string) /* 0A000037 */
    IL_002c: stloc.0 IL_002d: ldloc.0
    IL_002e: call string [System.Runtime.Extensions]
      System.IO.Path::GetDirectoryName(string) /* 0A000038 */
    IL_0033: stloc.1
    IL_0034: ldloc.1
    IL_0035: brtrue.s IL_0039 IL_0037: ldnull
    IL_0038: ret IL_0039: ldloc.1
    IL_003a: newobj instance void
      System.IO.DirectoryInfo::.ctor(string) /* 06000097 */
    IL_003f: ret
  } // конец метода Directory::GetParent
```

Окна редактирования кода IL не будут вам особенно полезны, пока вы не разберетесь в разработке приложений на C# и .NET достаточно хорошо — настолько, что уже станет действительно важно знать, как компилятор C# преобразует ваш исходный код в код IL. Более полезные окна редактирования содержат эквивалентный исходный код на C#, написанный экспертами Microsoft. Вы можете научиться многим важным навыкам, увидев, как профессионалы реализуют типы. Например, метод GetParent показывает, как проверять аргументы на null и другие исключения аргументов.

12. Закройте окна редактирования, не сохраняя изменения.
13. На панели EXPLORER (Проводник) в ILSPY DECOMPILED MEMBERS (Декомпилированные члены ILSpy) щелкните правой кнопкой мыши на сборке и выберите пункт меню Unload Assembly (Выгрузить сборку).

Нет, вы не можете технически предотвратить декомпиляцию

Иногда меня спрашивают, есть ли способ защитить скомпилированный код от декомпиляции. Мой ответ — нет, и если вы подумаете об этом, то поймете почему. Вы можете усложнить задачу, используя инструменты запутывания кода, такие как Dotfuscator, но в конечном итоге не сможете полностью предотвратить декомпиляцию.

Все скомпилированные приложения содержат инструкции для платформы, операционной системы и оборудования, на котором они работают. Эти инструкции

должны быть функционально такими же, как и оригинальный исходный код, но их сложнее читать. Эти инструкции должны быть удобочитаемыми для выполнения вашего кода, поэтому они должны быть понятны при выполнении декомпиляции. Если вы защитите свой код от декомпиляции с помощью специальных устройств, то при этом предотвратите и запуск своего кода!

Виртуальные машины имитируют аппаратное обеспечение и поэтому могут фиксировать все взаимодействия между вашим запущенным приложением и программным и аппаратным обеспечением, на которых, по их мнению, оно работает.

Если бы вы могли защитить свой код, то предотвратили бы также присоединение к нему с помощью отладчика и пошаговое выполнение. Если в скомпилированном приложении находится файл pdb, то можете прикрепить отладчик и выполнять операторы построчно. Даже без файла pdb вы все равно можете подключить отладчик и получить некоторое представление о работе кода.

Это подходит для всех языков программирования. Не только языки .NET, такие как C#, Visual Basic и F#, но и C, C++, Delphi, язык ассемблера — все они могут быть подключены для отладки, дизассемблирования или декомпиляции. Некоторые инструменты, используемые профессионалами, описаны в табл. 7.10.

Таблица 7.10. Инструменты, используемые профессионалами

Тип	Продукт	Описание
Виртуальная машина	VMware	Профессионалы, такие как аналитики вредоносных программ, всегда запускают программное обеспечение внутри виртуальной машины
Отладчик	SoftICE	Запускается под операционной системой обычно на виртуальной машине
Отладчик	WinDbg	Важен для понимания внутреннего устройства Windows, поскольку знает о структурах данных Windows больше, чем другие отладчики
Дизассемблер	IDA Pro	Используется профессиональными аналитиками вредоносных программ
Декомпилятор	HexRays	Декомпилирует приложения, разработанные на языке C. Плагин для IDA Pro
Декомпилятор	DeDe	Декомпилирует приложения, разработанные на языке Delphi
Декомпилятор	dotPeek	Декомпилятор .NET от JetBrains

Отладка, дизассемблирование и декомпиляция чужого программного обеспечения, скорее всего, противоречит его лицензионному соглашению и является незаконным во многих юрисдикциях. Зная законы, вы всегда сможете защитить свою интеллектуальную собственность.

Упаковка библиотек для распространения через NuGet

Прежде чем мы научимся создавать и упаковывать наши собственные библиотеки, рассмотрим, как проект может использовать существующий пакет.

Ссылка на пакет NuGet

Предположим, вы хотите добавить пакет, созданный сторонним разработчиком, например `Newtonsoft.Json` — популярный пакет для работы с форматом сериализации *JavaScript Object Notation (JSON)*.

1. В проекте `AssembliesAndNamespaces` добавьте ссылку на пакет NuGet `Newtonsoft.Json` либо с помощью GUI для Visual Studio 2022, либо с помощью команды `dotnet add package` для Visual Studio Code.
2. Откройте файл `AssembliesAndNamespaces.csproj`, и вы увидите, что ссылка на пакет добавлена, как показано ниже:

```
<ItemGroup>
  <PackageReference Include="newtonsoft.json" Version="13.0.1" />
</ItemGroup>
```

У вас может быть установлена более новая версия пакета `newtonsoft.json`, поскольку с момента публикации этой главы наверняка было написано обновление.

Фиксация зависимостей

Чтобы восстанавливать пакеты целостно и писать надежный код, важно *фиксировать зависимости*. Это обеспечит использование одного и то же семейства пакетов, выпущенных для конкретной версии .NET, например SQLite для .NET 6.0, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="6.0.0" />
  </ItemGroup>

</Project>
```

Чтобы зафиксировать зависимости, каждый пакет должен содержать одну версию без дополнительных спецификаторов, которые включают в себя бета-версии (`beta1`), релиз-кандидаты (`rc4`) и подстановочные символы (`*`).

Подстановочные символы позволяют автоматически ссылаться на будущие версии и использовать их, поскольку всегда представляют самый последний выпуск. Но эти символы особенно опасны, так как могут повлечь применение несовместимых пакетов, что повлечет неработоспособность приложения.

Поскольку новые версии пакетов выпускаются каждый месяц, то при написании книги вам придется постоянно обновлять ссылки на них (что очень неудобно), как я делал в течение 2021 года:

```
<PackageReference
  Include="Microsoft.EntityFrameworkCore.Sqlite"
  Version="6.0.0-preview.*" />.
```

Если вы используете команду `dotnet add package` или *управление пакетами NuGet* в Visual Studio, то по умолчанию будет использоваться самая последняя версия пакета. Но если вы копируете и вставляете конфигурацию из статьи блога или вручную добавляете ссылку, то можете включить квалификаторы с подстановочными знаками.

Зависимости, приведенные ниже, — пример ссылок на пакеты NuGet, которые *не являются* фиксированными, и поэтому их следует избегать, если вы не знаете последствий:

```
<PackageReference Include="System.NET.Http" Version="4.1.0-*" />
<PackageReference Include="Newtonsoft.Json" Version="12.0.3-beta1" />
```

Сотрудники Microsoft гарантируют совместную работу пакетов, если вы зафиксировали свои зависимости на версии, поставляемой с определенной версией .NET, например 6.0.0. Почти всегда фиксируйте свои зависимости.

Упаковка библиотеки для NuGet

Теперь упакуем ранее созданный проект `SharedLibrary`.

1. В проекте `SharedLibrary` переименуйте файл `Class1.cs` в `StringExtensions.cs`.
2. Измените его содержимое, чтобы предоставить некоторые полезные методы расширения для проверки различных текстовых значений с помощью регулярных выражений:

```
using System.Text.RegularExpressions;

namespace Packt.Shared
{
  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 шестнадцатеричных символов
      return Regex.IsMatch(input,
        "^#?([a-fA-F0-9]{3}|[a-fA-F0-9]{6})$");
    }
  }
}
```

О том, как писать регулярные выражения, вы узнаете в главе 8.

3. Измените содержимое файла `SharedLibrary.csproj`, как показано ниже, а также обратите внимание на следующие моменты:
   - идентификатор `PackageId` должен быть глобально уникальным, поэтому вы должны использовать другое значение, если хотите опубликовать данный пакет NuGet в общедоступном веб-канале www.nuget.org, чтобы другие могли ссылаться на него и скачивать его;
   - `PackageLicenseExpression` должно быть значением, взятым с spdx.org/licenses/, или вы можете указать собственную лицензию;
   - все остальные элементы не требуют объяснения.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>

    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
    <PackageId>Packt.CSdotnet.SharedLibrary</PackageId>
    <PackageVersion>6.0.0.0</PackageVersion>
    <Title>C# 10 and .NET 6 Shared Library</Title>
    <Authors>Mark J Price</Authors>
    <PackageLicenseExpression>
      MS-PL
    </PackageLicenseExpression>
    <PackageProjectUrl>
      https://github.com/markjprice/cs10dotnet6
    </PackageProjectUrl>
    <PackageIcon>packt-csdotnet-sharedlibrary.png</PackageIcon>
```

```
    <PackageRequireLicenseAcceptance>true</
    PackageRequireLicenseAcceptance>
    <PackageReleaseNotes>
      Example shared library packaged for NuGet.
    </PackageReleaseNotes>
    <Description>
      Three extension methods to validate a string value.
    </Description>
    <Copyright>
      Copyright © 2016-2021 Packt Publishing Limited
    </Copyright>
    <PackageTags>string extensions packt csharp dotnet</PackageTags>

  </PropertyGroup>

  <ItemGroup>
    <None Include="packt-csdotnet-sharedlibrary.png">
      <Pack>True</Pack>
      <PackagePath></PackagePath>
    </None>
  </ItemGroup>

</Project>
```

Значения свойств конфигурации, которые допускают значения true или false, не могут содержать пробелов, поэтому запись <PackageRequireLicenseAcceptance> не допускает возврата каретки и отступа, которые показаны в коде выше.

4. Скачайте файл, содержащий изображение значка, перейдя по следующей ссылке: https://github.com/markjprice/cs10dotnet6/blob/main/vs4win/Chapter07/SharedLibrary/packt-csdotnet-sharedlibrary.png, и сохраните его в папке SharedLibrary.
5. Соберите релиз:
   1) в Visual Studio на панели инструментов выберите пункт Release (Выпуск), а затем выберите команду меню Build ▸ Build SharedLibrary (Сборка ▸ Создать общую библиотеку);
   2) в Visual Studio Code на панели TERMINAL (Терминал) введите команду `dotnet build -c Release`.
6. Если бы мы в файле проекта не установили для `<GeneratePackageOnBuild>` значение `true`, то нам пришлось бы вручную создать пакет NuGet, выполнив следующие дополнительные шаги:
   1) в Visual Studio выбрав команду меню Build ▸ Pack SharedLibrary (Сборка ▸ Упаковать общую библиотеку);
   2) в Visual Studio Code на панели TERMINAL (Терминал) введя команду `dotnet pack -c Release`.

Публикация пакета в общедоступном канале NuGet

Если вы хотите, чтобы другие разработчики могли скачать и использовать пакет NuGet, то вам необходимо загрузить его в общедоступный канал NuGet, такой как Microsoft.

1. Запустите браузер и перейдите по следующей ссылке: https://www.nuget.org/packages/manage/upload.
2. Если вы хотите загрузить пакет NuGet, чтобы другие разработчики могли ссылаться на него как на пакет зависимостей, то вам необходимо войти в систему через учетную запись Microsoft по адресу https://www.nuget.org/.
3. Нажмите Browse (Обзор) и выберите файл `.nupkg`, созданный путем генерации пакета NuGet. Путь к папке должен выглядеть следующим образом: `Code\Chapter07\SharedLibrary\bin\Release`, а файл должен называться `Packt.CSdotnet.SharedLibrary.6.0.0.nupkg`.
4. Убедитесь, что информация, введенная вами в файл `SharedLibrary.csproj`, правильная, а затем нажмите кнопку Submit (Отправить).
5. Через несколько секунд появится сообщение об успешном завершении загрузки вашего пакета (рис. 7.8).

Рис. 7.8. Сообщение о загрузке пакета NuGet

Если вы получили сообщение об ошибке, то проверьте файл проекта на наличие ошибок или прочитайте дополнительную информацию о формате PackageReference на сайте https://docs.microsoft.com/ru-ru/nuget/reference/msbuild-targets.

Публикация пакета в частном веб-канале NuGet

Организации могут размещать собственные частные веб-каналы NuGet. Это может быть удобно для многих команд разработчиков. Более подробную информацию вы можете найти на сайте https://docs.microsoft.com/en-us/nuget/hosting-packages/overview.

Изучение пакетов NuGet с помощью инструмента

Разработчики компании Uno Platform создали удобный инструмент *NuGet Package Explorer*, позволяющий открывать пакет NuGet и просматривать дополнительные сведения о нем. Помимо того, что это сайт, его можно установить как кросс-платформенное приложение. Посмотрим, как он работает.

1. Запустите браузер и перейдите по следующей ссылке: https://nuget.info.
2. В поле поиска введите `Packt.CSdotnet.SharedLibrary`.
3. Выберите пакет v6.0.0, опубликованный Марком Дж. Прайсом, и нажмите кнопку Open (Открыть).
4. В разделе Contents (Содержание) разверните папку lib и папку netstandard2.0.
5. Выберите SharedLibrary.dll и обратите внимание на детали (рис. 7.9).

Рис. 7.9. Изучение моего пакета с помощью NuGet Package Explorer компании Uno Platform

6. Если вы в будущем хотите использовать этот инструмент локально, то нажмите в браузере кнопку установки.

7. Закройте браузер.

Не все браузеры поддерживают установку подобных веб-приложений. Для тестирования и разработки я рекомендую Chrome.

Тестирование пакета библиотеки классов

Теперь загруженный пакет необходимо протестировать, создав ссылку на него в проекте `AssembliesAndNamespaces`.

1. В проекте `AssembliesAndNamespaces` добавьте ссылку на ваш (или мой) пакет, как показано ниже:

```
<ItemGroup>
  <PackageReference Include="newtonsoft.json" Version="13.0.1" />
  <PackageReference Include="packt.csdotnet.sharedlibrary"
    Version="6.0.0" />
</ItemGroup>
```

2. Соберите консольное приложение.
3. В файле `Program.cs` импортируйте пространство имен `Packt.Shared`.
4. В файле `Program.cs` предложите пользователю ввести некоторые значения `string`, а затем проверьте их с помощью методов расширения в пакете:

```
Write("Enter a color value in hex: ");
string? hex = ReadLine(); // или "00ffc8"
WriteLine("Is {0} a valid color value? {1}",
  arg0: hex, arg1: hex.IsValidHex());

Write("Enter a XML element: ");
string? xmlTag = ReadLine(); // или "<h1 class=\"<\" />"
WriteLine("Is {0} a valid XML element? {1}",
  arg0: xmlTag, arg1: xmlTag.IsValidXmlTag());

Write("Enter a password: ");
string? password = ReadLine(); // или "secretsauce"
WriteLine("Is {0} a valid password? {1}",
  arg0: password, arg1: password.IsValidPassword());
```

5. Запустите код, введите некоторые значения по запросу и проанализируйте результат:

```
Enter a color value in hex: 00ffc8
Is 00ffc8 a valid color value? True
Enter an XML element: <h1 class="<" />
Is <h1 class="<" /> a valid XML element? False
Enter a password: secretsauce
Is secretsauce a valid password? True
```

Портирование приложений с .NET Framework на современной .NET

Если вы в настоящее время являетесь разработчиком на .NET Framework, то вам может быть интересно, стоит ли переносить существующие приложения на современную .NET. Подумайте, является ли перенос правильным решением для вашего кода, поскольку иногда лучше ничего не переносить.

Например, у вас может быть сложный проект сайта, работающий на платформе .NET Framework 4.8, но посещаемый лишь небольшим количеством пользователей. Если он работает и обрабатывает трафик пользователей на минимальном оборудовании, то, возможно, месяцы переноса его на .NET 6 могут оказаться пустой тратой времени. Но если для сайта в настоящее время требуется много дорогих серверов Windows, то стоимость переноса может в конечном итоге окупиться, если вы сможете перейти на меньшее количество менее дорогих серверов Linux.

Можете ли вы портировать

Современная .NET прекрасно поддерживает следующие типы приложений в операционных системах Windows, macOS и Linux:

- сайты *ASP.NET Core MVC*;
- веб-сервисы *ASP.NET Core Web API* (REST/HTTP);
- сервисы ASP.NET Core SignalR;
- интерфейсы командной строки *консольного приложения*.

Современная .NET прекрасно поддерживает следующие типы приложений в операционной системе Windows (поэтому они являются потенциальными кандидатами на перенос):

- приложения *Windows Forms*;
- приложения *Windows Presentation Foundation (WPF)*.

Современная .NET хорошо поддерживает следующие типы приложений на кроссплатформенных настольных и мобильных устройствах:

- приложения *Xamarin* для операционных систем *iOS* и *Android*;
- *NET MAUI* для настольных приложений Windows и macOS или мобильных приложений на iOS и Android.

Современная .NET не поддерживает следующие типы устаревших приложений Microsoft:

- веб-приложения *ASP.NET Web Forms*. Их лучше всего реализовать повторно с помощью *ASP.NET Core Razor Pages* или *Blazor*;
- сервисы *Windows Communication Foundation (WCF)* (но существует проект с открытым исходным кодом *CoreWCF*, который вы можете использовать в зависимости от требований). Сервисы WCF лучше реализовать повторно с помощью сервисов *ASP.NET Core gRPC*;
- приложения *Silverlight*. Их лучше всего реализовать заново с помощью *.NET MAUI*.

Приложения Silverlight и ASP.NET Web Forms никогда не смогут быть перенесены в современную .NET. Однако существующие приложения Windows Forms и WPF-приложения можно перенести в .NET в операционной системе Windows, чтобы использовать новые API и более высокую производительность.

Устаревшие веб-приложения ASP.NET MVC и веб-сервисы ASP.NET Web API, в настоящее время находящиеся в .NET Framework, могут быть перенесены в современную .NET, а затем размещены в операционных системах Windows, Linux или macOS.

Стоит ли портировать

Так *следует* ли выполнять перенос, если это *технически возможно*? Какие преимущества вы при этом получаете? Ниже перечислены некоторые из них.

- *Развертывание сайтов и веб-сервисов в Linux, Docker или Kubernetes*. Эти операционные системы являются облегченными и экономичными в использовании в качестве платформ сайтов и веб-сервисов, особенно по сравнению с более дорогой Windows Server.
- *Удаление зависимости от IIS и System.Web.dll*. Даже если вы продолжите развертывание на Windows Server, платформа ASP.NET Core может быть размещена на облегченных и высокопроизводительных веб-серверах Kestrel (или других).
- *Инструменты командной строки*. Инструменты, которые разработчики и администраторы используют для автоматизации своих задач, часто создаются как консольные приложения. Возможность запуска единого инструмента кросс-платформенно очень полезна.

Сравнение .NET Framework и современной .NET

В табл. 7.11 приведены три важных различия этих платформ.

Таблица 7.11. Различия .NET Framework и современной .NET

Современная .NET	.NET Framework
Распространяется в виде NuGet-пакетов, поэтому каждое приложение может быть развернуто с помощью отдельной локальной версии платформы .NET, которая необходима для запуска	Распространяется в виде общесистемного комплекта сборок (если буквально, то глобального кэша сборок (Global Assembly Cache, GAC))
Разделена на небольшие разноуровневые компоненты, чтобы можно было выполнить минимальное развертывание	Требует полноценного монолитного развертывания
Удалены старые технологии, такие как ASP.NET Web Forms, и не кросс-платформенные функции, например AppDomains, .NET Remoting и бинарная сериализация	Наряду с технологиями, аналогичными имеющимся в современной .NET, например ASP.NET Core MVC, здесь сохраняются некоторые более старые технологии, такие как ASP.NET Web Forms

.NET Portability Analyzer

Специалисты компании Microsoft создали полезный инструмент, который можно использовать для оценки возможности переноса существующих приложений и подготовки соответствующего отчета, — .NET Portability Analyzer (анализатор переносимости .NET). Демонстрация его работы доступна по адресу https://docs.microsoft.com/en-us/shows/seth-juarez/brief-look-net-portability-analyzer.

.NET Upgrade Assistant

.NET Upgrade Assistant (помощник по обновлению) — последний инструмент Microsoft для обновления устаревших проектов до современной .NET.

Мое основное место работы — компания Optimizely. У нас есть платформа Digital Experience Platform (DXP) корпоративного масштаба на основе .NET Framework, включающая систему управления контентом (Content Management System, CMS) и предназначенная для создания сайтов цифровой коммерции. Компания Microsoft нуждалась в сложном проекте миграции для разработки и тестирования .NET Upgrade Assistant, поэтому мы работали с ними над созданием полезного инструмента.

В настоящее время им поддерживаются следующие типы проектов .NET Framework (а позже будут добавлены и другие):

- ASP.NET MVC;
- Windows Forms;
- WPF;
- консольное приложение;
- библиотека классов.

Он устанавливается как глобальный инструмент `dotnet`:

```
dotnet tool install -g upgrade-assistant
```

Подробнее об этом инструменте и о его использовании вы можете прочитать на сайте https://docs.microsoft.com/en-us/dotnet/core/porting/upgrade-assistant-overview.

Использование библиотек, не скомпилированных для .NET Standard

Большинство существующих пакетов NuGet можно использовать с современной .NET, даже если они не скомпилированы для .NET Standard или современной версии, такой как .NET 6. Если вы нашли пакет, который официально не поддерживает .NET Standard, как показано на его веб-странице nuget.org, то не стоит беспокоиться. Просто проверьте, работает ли он.

Например, существует пакет пользовательских коллекций для обработки матриц, созданный компанией Dialect Software LLC, документацию которого можно найти по ссылке https://www.nuget.org/packages/DialectSoftware.Collections.Matrix/.

Последний раз пакет обновлялся в 2013 году, то есть задолго до появления NET Core или .NET 6, поэтому был создан для .NET Framework. Пока такой пакет сборки использует только API, доступные в .NET Standard, он может применяться в проекте современной .NET.

Попробуем его использовать.

1. В проекте `AssembliesAndNamespaces` добавьте ссылку на пакет Dialect Software:

```
<PackageReference
  Include="dialectsoftware.collections.matrix"
  Version="1.0.0" />
```

2. Соберите проект `AssembliesAndNamespaces` для восстановления пакетов.
3. В файле `Program.cs` добавьте операторы для импорта пространств имен `DialectSoftware.Collections` и `DialectSoftware.Collections.Generics`.
4. Добавьте операторы для создания экземпляров `Axis` и `Matrix<T>`, заполните их значениями и выведите:

```
Axis x = new("x", 0, 10, 1);
Axis y = new("y", 0, 4, 1);

Matrix<long> matrix = new(new[] { x, y });

for (int i = 0; i < matrix.Axes[0].Points.Length; i++)
{
  matrix.Axes[0].Points[i].Label = "x" + i.ToString();
}

for (int i = 0; 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]);
}
```

5. Запустите код, обратите внимание на предупреждающее сообщение и проанализируйте результат:

```
warning NU1701: Package 'DialectSoftware.Collections.Matrix
1.0.0' was restored using '.NETFramework,Version=v4.6.1,
.NETFramework,Version=v4.6.2, .NETFramework,Version=v4.7,
.NETFramework,Version=v4.7.1, .NETFramework,Version=v4.7.2,
.NETFramework,Version=v4.8' instead of the project target framework
'net6.0'. 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
...
```

Данный пакет был создан до того, как появилась .NET 6, и компилятор и среда выполнения не могут знать, сработает ли он, и поэтому отображают предупреждения; однако поскольку пакет вызывает только API, совместимые с NET Standard, он успешно работает.

Функции предварительного просмотра

Перед Microsoft стоит задача предоставить некоторые новые функции, влияющие на многие разделы .NET, такие как среда выполнения, языковые компиляторы и библиотеки API. Это классическая проблема курицы и яйца. Что вы делаете в первую очередь?

С практической точки зрения это означает следующее: компания Microsoft, возможно, завершила бóльшую часть работы по добавлению новой функции, однако все может быть готово только в самом конце их нынешних годовых выпусков .NET, а это слишком поздно для надлежащего тестирования в естественных условиях.

Таким образом, начиная с версии .NET 6, Microsoft будет включать предварительные функции в выпуски *общедоступной версии* (general availability, GA). Разработчики смогут выбрать эти функции предварительного просмотра и оставить Microsoft свой отзыв. В более позднем выпуске GA их можно будет включить для всех версий.

Функции предварительного просмотра не поддерживаются в рабочем коде и, скорее всего, перед окончательным выпуском будут иметь критические изменения. Включать эти функции можно на свой страх и риск.

Требование к функциям предварительного просмотра

Атрибут `[RequiresPreviewFeatures]` используется для указания используемых сборок, типов или членов, и поэтому необходимо отображать предупреждение о функциях предварительного просмотра. Затем анализатор кода сканирует эту сборку и при необходимости генерирует предупреждения. Если ваш код не использует какие-либо функции предварительного просмотра, вы не увидите никаких предупреждений. Если вы их используете, то ваш код должен предупреждать своих пользователей об этом.

Включение функций предварительного просмотра

Рассмотрим пример функции предварительного просмотра, доступной в .NET 6, возможности определения интерфейса с помощью статического абстрактного метода.

1. Откройте редактор кода и создайте консольное приложение `UsingPreviewFeatures` в рабочей области/решении `Chapter07`.
2. В Visual Studio Code выберите `UsingPreviewFeatures` в качестве активного проекта OmniSharp. Когда вы увидите всплывающее предупреждение о том, что необходимые ресурсы отсутствуют, нажмите кнопку Yes (Да), чтобы добавить их.
3. В файле проекта добавьте элемент для включения функций предварительного просмотра и элемент для включения языковых функций предварительного просмотра, как показано ниже (выделено жирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <EnablePreviewFeatures>true</EnablePreviewFeatures>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>

</Project>
```

4. В файле `Program.cs` удалите комментарий и статически импортируйте класс `Console`.
5. Добавьте операторы для определения интерфейса со статическим абстрактным методом, классом, который его реализует, а затем вызовите метод в программе верхнего уровня:

```
using static System.Console;

Doer.DoSomething();

public interface IWithStaticAbstract
{
  static abstract void DoSomething();
}

public class Doer : IWithStaticAbstract
{
  public static void DoSomething()
  {
```

```
    WriteLine("I am an implementation of a static abstract method.");
  }
}
```

6. Запустите консольное приложение и проанализируйте результат.

Математические операции с дженериками

Почему компания Microsoft добавила возможность определять статические абстрактные методы? Чем они полезны?

Долгое время разработчики просили Microsoft предоставить им возможность использовать для типов-дженериков такие операции, как *. Это позволило бы разработчикам определять математические методы для выполнения таких операций, как сложение, усреднение и т. д., для любых типов-дженериков, вместо того чтобы создавать десятки перегруженных методов для всех числовых типов, которые необходимо поддерживать. Поддержка статических абстрактных методов в интерфейсах — фундаментальная функция, позволяющая использовать математические операции с дженериками.

Более подробную информацию можно найти на сайте https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-7---packaging-and-distributing-net-types.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 7.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. В чем разница между пространством имен и сборкой?
2. Как вы ссылаетесь на другой проект в файле `.csproj`?
3. В чем преимущество такого инструмента, как ILSpy?
4. Какой тип .NET представлен псевдонимом `float` в C#?
5. Какой инструмент следует использовать перед переносом приложения с .NET Framework на .NET 6 и какой инструмент позволяет выполнить бóльшую часть работы по переносу?

6. В чем разница между платформенно-зависимым и автономным развертыванием современных приложений .NET?
7. Что такое RID?
8. Чем различаются команды `dotnet pack` и `dotnet publish`?
9. Какие типы приложений, написанных для .NET Framework, можно перенести на современную .NET?
10. Можете ли вы использовать пакеты, написанные для .NET Framework, с современной .NET?

Упражнение 7.2. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-7---understanding-and-packaging-net-types, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Упражнение 7.3. PowerShell

PowerShell — это язык сценариев Microsoft для автоматизации задач в любой операционной системе. Microsoft рекомендует Visual Studio Code с расширением PowerShell для написания сценариев PowerShell.

Поскольку PowerShell — это самостоятельный расширяемый язык сценариев, в данной книге он не описан. Вместо этого я создал несколько дополнительных страниц в репозитории книг на GitHub, чтобы познакомить вас с некоторыми ключевыми понятиями и показать несколько примеров: https://github.com/markjprice/cs10dotnet6/tree/main/docs/powershell.

Резюме

В этой главе вы познакомились с .NET 6, изучили взаимосвязь сборок и пространств имен, варианты публикации приложения для распространения в нескольких операционных системах, мы упаковали и распространили библиотеку классов, а также обсудили варианты переноса существующих кодовых баз, написанных на .NET Framework.

В следующей главе вы узнаете о некоторых распространенных типах библиотек базовых классов, включенных в современную .NET.

8 Работа с распространенными типами .NET

Данная глава посвящена распространенным типам .NET. В их число входят типы для работы с числами, текстом, коллекциями, доступом к сети, отражением, атрибутами; типы для улучшения работы с интервалами, индексами и диапазонами; типы для работы с изображениями, а также интернационализация.

В этой главе:

- работа с числами;
- работа с текстом;
- работа с датами и временем;
- сопоставление с образцом с помощью регулярных выражений;
- хранение нескольких объектов в коллекциях;
- работа с интервалами, индексами и диапазонами;
- работа с сетевыми ресурсами;
- работа с отражением и атрибутами;
- работа с изображениями;
- интернационализация кода.

Работа с числами

Числа — один из наиболее распространенных типов данных. В табл. 8.1 перечислены наиболее распространенные типы в .NET для работы с числами.

Начиная с .NET Framework 1.0, в .NET входили 32-битные типы с плавающей запятой и 64-битные типы `double`. Спецификация IEEE 754 также определяет 16-битный стандарт с плавающей запятой. Машинное обучение и другие алгоритмы выиграют от этого меньшего числового типа с более низкой точностью, поэтому Microsoft ввела тип `System.Half` в .NET 5 и более поздних версиях.

Таблица 8.1. Наиболее распространенные типы в .NET для работы с числами

Пространство имен	Пример типа (-ов)	Описание
`System`	`Sbyte`, `Int16`, `Int32`, `Int64`	Целые числа: ноль, положительные и отрицательные целые числа
`System`	`Byte`, `UInt16`, `UInt32`, `UInt64`	Кардинальные числа: ноль и положительные целые числа
`System`	`Half`, `Single`, `Double`	Действительные числа, то есть числа с плавающей запятой
`System`	`Decimal`	Точные действительные числа: используются в научных, инженерных или финансовых сценариях
`System.Numerics`	`BigInteger`, `Complex`, `Quaternion`	Целые числа произвольной величины, комплексные числа и кватернионы

В настоящее время язык C# не определяет псевдоним `half`, поэтому вам необходимо использовать тип .NET `System.Half`. Однако в дальнейшем это может измениться.

Большие целые числа

Наибольшее целое число, которое может храниться в типах .NET и у которого есть псевдоним в C#, составляет около 18,5 квинтиллиона и хранится в виде беззнакового целого числа `long`. Но что, если вам понадобится хранить бóльшие числа?

Изучим числовые данные.

1. Откройте редактор кода и создайте решение/рабочую область `Chapter08`.
2. Создайте проект консольного приложения с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter08`;
   3) файл и папка проекта: `WorkingWithNumbers`.
3. В файле `Program.cs` удалите имеющиеся операторы и добавьте оператор для импорта простанства имен `System.Numerics`:

```
using System.Numerics;
```

4. Добавьте операторы для вывода максимального значения типа `ulong` и числа из 30 знаков, используя `BigInteger`:

```
WriteLine("Working with large integers:");
WriteLine("----------------------------------");

ulong big = ulong.MaxValue;
WriteLine($"{big,40:N0}");
```

```
BigInteger bigger =
  BigInteger.Parse("123456789012345678901234567890");

WriteLine($"{bigger,40:N0}");
```

Число `40` в коде формата означает выравнивание 40 символов по правому краю, поэтому оба числа выровнены по правому краю. `N0` означает разделитель тысяч и нулевых десятичных знаков.

5. Запустите код и проанализируйте результат:

```
Working with large integers:
----------------------------------------
              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 выражено следующим образом: $a + bi$, где a и b — действительные числа, а i — мнимая единица, где $i^2 = -1$. Если действительная часть равна нулю, то это мнимое число. Если мнимая часть равна нулю, то это действительное число.

Комплексные числа используются во многих областях *STEM* (science, technology, engineering, mathematics — «наука, технология, инженерия, математика»). Кроме того, обратите внимание, что сложение для них производится путем независимого сложения действительных и мнимых частей слагаемых:

```
(a + bi) + (c + di) = (a + c) + (b + d)i
```

Рассмотрим комплексные числа.

1. В файле `Program.cs` введите операторы для добавления двух комплексных чисел:

```
WriteLine("Working with complex numbers:");
Complex c1 = new(real: 4, imaginary: 2);
Complex c2 = new(real: 3, imaginary: 7);
Complex c3 = c1 + c2;

// вывод с использованием реализации ToString по умолчанию
WriteLine($"{c1} added to {c2} is {c3}");

// вывод в пользовательском формате
WriteLine("{0} + {1}i added to {2} + {3}i is {4} + {5}i",
  c1.Real, c1.Imaginary,
  c2.Real, c2.Imaginary,
  c3.Real, c3.Imaginary);
```

2. Запустите код и проанализируйте результат:

```
Working with complex numbers:
(4, 2) added to (3, 7) is (7, 9)
4 + 2i added to 3 + 7i is 7 + 9i
```

Кватернионы

Кватернионы — система чисел, расширяющая систему комплексных чисел. Они образуют четырехмерную ассоциативную нормированную алгебру с делением над полем действительных чисел и, следовательно, также предметную область.

Звучит сложно? Я тоже не очень хорошо в этом разбираюсь. Не беспокойтесь! Мы не будем использовать кватернионы при написании кода. Достаточно того, что они применяются для записи пространственных вращений, поэтому их используют в движках видеоигр, компьютерных симуляциях и системах управления полетами.

Работа с текстом

Другой весьма распространенный тип данных для переменных — это текст. Самые распространенные типы .NET Standard для работы с текстом перечислены в табл. 8.2.

Таблица 8.2. Самые распространенные типы .NET Standard для работы с текстом

Пространство имен	Тип	Описание
`System`	`Char`	Хранит один текстовый символ
`System`	`String`	Хранит несколько текстовых символов
`System.Text`	`StringBuilder`	Эффективно управляет строками
`System.Text.RegularExpressions`	`Regex`	Эффективно сопоставляет строки с образцами

Извлечение длины строки

Рассмотрим некоторые распространенные задачи, возникающие при работе с текстом. Например, вам необходимо определить длину фрагмента текста, хранящегося в переменной `string`.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithText` в рабочей области/решении `Chapter08`:
   1) в Visual Studio настройте стартовый проект для решения в соответствии с текущим выбором;
   2) в Visual Studio Code выберите `WorkingWithText` в качестве активного проекта OmniSharp.
2. В проекте `WorkingWithText` в файле `Program.cs` добавьте следующие операторы, чтобы определить переменную для хранения названия города Лондон, а затем запишите его имя и длину в консоль:

```
string city = "London";
WriteLine($"{city} is {city.Length} characters long.");
```

3. Запустите код и проанализируйте результат:

```
London is 6 characters long.
```

Извлечение символов строки

Класс `string` для хранения текста использует внутренний массив `char`. Для этого класса определен индексатор, поэтому с помощью синтаксиса массива мы можем читать хранящиеся символы. Массивы индексируются с нуля, поэтому третий символ будет находиться под индексом 2.

Посмотрим, как это работает.

1. Добавьте оператор для записи символов в первой и третьей позициях переменной `string`:

```
WriteLine($"First char is {city[0]} and third is {city[2]}.");
```

2. Запустите код и проанализируйте результат:

```
First char is L and third is n.
```

Разделение строк

Иногда требуется разделить текст в местах, где используется определенный символ, например запятая.

1. Добавьте операторы для определения одной переменной `string`, содержащей названия городов, разделенные запятыми, а затем используйте метод `Split`

и укажите, что хотите обработать запятые в качестве символа-разделителя, а затем перечислите возвращенный массив значений `string`:

```
string cities = "Paris,Tehran,Chennai,Sydney,New York,Medellín";

string[] citiesArray = cities.Split(',');

WriteLine($"There are {citiesArray.Length} items in the array.");
foreach (string item in citiesArray)
{
  WriteLine(item);
}
```

2. Запустите код и проанализируйте результат:

```
There are 6 items in the array.
Paris
Tehran
Chennai
Sydney
New York
Medellín
```

Далее в этой главе вы научитесь обрабатывать более сложные сценарии.

Извлечение фрагмента строки

В некоторых случаях может потребоваться извлечь часть текста. У метода `IndexOf` есть девять перегрузок, возвращающих позицию индекса указанного символа или строки внутри строки. Метод `Substring` содержит две перегрузки:

- `Substring(startIndex, length)` возвращает подстроку, которая начинается с `startIndex` и содержит последующие `length` символов;
- `Substring(startIndex)` возвращает подстроку, которая начинается с `startIndex` и содержит все символы до конца строки.

Рассмотрим пример.

1. Добавьте следующие операторы, чтобы сохранить полное имя человека в переменной `string` с символом пробела между именем и фамилией, найти позицию пробела и извлечь имя и фамилию по отдельности, чтобы их можно было повторно объединить в другом порядке:

```
string fullName = "Alan Jones";
int indexOfTheSpace = fullName.IndexOf(' ');

string firstName = fullName.Substring(
  startIndex: 0, length: indexOfTheSpace);

string lastName = fullName.Substring(
  startIndex: indexOfTheSpace + 1);

WriteLine($"Original: {fullName}");
WriteLine($"Swapped: {lastName}, {firstName}");
```

2. Запустите код и проанализируйте результат:

```
Original: Alan Jones
Swapped: Jones, Alan
```

Будь формат исходного полного имени иным, к примеру `"Lastname, Firstname"`, то и код отличался бы. В качестве дополнительного упражнения попробуйте написать несколько операторов, которые изменят входные данные `"Jones, Alan"` на `"Alan Jones"`.

Проверка содержимого в строках

Иногда необходимо проверить наличие определенных символов в начале, внутри или в конце строки. Это возможно с помощью методов `StartsWith`, `EndsWith` и `Contains`.

1. Добавьте операторы для хранения значения `string`, а затем проверьте, начинается ли оно с некоторых значений `string` или содержит их:

```
string company = "Microsoft";
bool startsWithM = company.StartsWith("M");
bool containsN = company.Contains("N");
WriteLine($"Text: {company}");
WriteLine($"Starts with M: {startsWithM}, contains an N: {containsN}");
```

2. Запустите код и проанализируйте результат:

```
Text: Microsoft
Starts with M: True, contains an N: False
```

Конкатенация строк, форматирование и прочие члены типа string

В табл. 8.3 перечислены некоторые другие члены типа `string`.

Таблица 8.3. Некоторые другие члены типа string

Член	Описание
`Trim`, `TrimStart`, `TrimEnd`	Эти методы удаляют пробельные символы, такие как пробел, табуляция и возврат каретки в начале и/или конце
`ToUpper`, `ToLower`	Преобразуют символы в прописные или строчные
`Insert`, `Remove`	Эти методы добавляют или удаляют указанный текст
`Replace`	Замещает указанный текст
`string.Empty`	Можно использовать вместо выделения памяти каждый раз, когда вы используете литеральное значение `string` с помощью пары двойных кавычек без содержимого (`""`)
`string.Concat`	Конкатенирует две переменные `string`. Операция `+` делает то же самое при использовании между операндами `string`
`string.Join`	Конкатенирует одну или несколько переменных `string` с указанным символом между ними
`string.IsNullOrEmpty`	Проверяет, является ли переменная `string` пустой или `null`
`string.IsNullOrWhitespace`	Проверяет, является ли строковая переменная `null` или пробелом; то есть сочетание любого количества горизонтальных и вертикальных интервальных символов, например табуляция, пробел, возврат каретки, перевод строки и т. д.
`string.Format`	Альтернативный метод интерполяции строк для вывода форматированных значений `string`, в котором используются позиционированные параметры вместо именованных

Некоторые из предыдущих методов — статические. Это значит, метод может быть вызван только из типа, а не из экземпляра переменной. В табл. 8.3 я указал статические методы, поставив перед ними префикс `string.`, например, `string.Format`.

Рассмотрим некоторые из перечисленных методов.

1. Добавьте следующие операторы, чтобы взять массив строковых значений и заново объединить их в одну строковую переменную с разделителями, используя метод `Join`:

```
string recombined = string.Join(" => ", citiesArray);
WriteLine(recombined);
```

2. Запустите код и проанализируйте результат:

```
Paris => Tehran => Chennai => Sydney => New York => Medellín
```

3. Добавьте следующие операторы, чтобы использовать позиционированные параметры и синтаксис форматирования интерполированных строк для двойного вывода одних и тех же трех переменных:

```
string fruit = "Apples";
decimal price = 0.39M;
DateTime when = DateTime.Today;

WriteLine($"Interpolated: {fruit} cost {price:C} on {when:dddd}.");

WriteLine(string.Format("string.Format: {0} cost {1:C} on {2:dddd}.",
  arg0: fruit, arg1: price, arg2: when));
```

4. Запустите код и проанализируйте результат:

```
Interpolated: Apples cost £0.39 on Thursday.
string.Format: Apples cost £0.39 on Thursday.
```

Обратите внимание, что мы могли бы упростить второй оператор, поскольку `WriteLine` поддерживает те же коды формата, что и `string.Format`:

```
WriteLine("WriteLine: {0} cost {1:C} on {2:dddd}.",
  arg0: fruit, arg1: price, arg2: when);
```

Эффективное создание строк

Вы можете конкатенировать (сцепить) две строки, чтобы создать новую строку типа `string`, используя метод `String.Concat` или операцию `+`. Но оба варианта не рекомендуются, поскольку .NET необходимо будет создать совершенно новую строку типа `string` в памяти.

Последствий может и не быть, если вы объединяете только два значения `string`, но если конкатенацию проводить в цикле с большим количеством итераций, это может весьма негативно повлиять на производительность и использование памяти. В главе 12 вы научитесь эффективно конкатенировать переменные `string` с помощью типа `StringBuilder`.

Работа с датами и временем

Следующими по популярности после чисел и текста типами данных для работы являются даты и время. Ниже приведены два основных типа:

- `DateTime` — представляет собой комбинированное значение даты и времени для фиксированного момента времени;
- `TimeSpan` — представляет собой промежуток времени.

Эти два типа часто используются вместе. Например, если вычесть одно значение `DateTime` из другого, то результатом будет `TimeSpan`. Если вы прибавляете `TimeSpan` к `DateTime`, то результатом будет значение `DateTime`.

Указание значений даты и времени

Обычно указываются отдельные значения для таких компонентов даты и времени, как день и час (табл. 8.4).

Таблица 8.4. Параметры даты/времени и диапазон значений

Параметр даты/времени	Диапазон значений
`year`	От 1 до 9999
`month`	От 1 до 12
`day`	От 1 до количества дней в данном месяце
`hour`	От 0 до 23
`minute`	От 0 до 59
`second`	От 0 до 59

Альтернативой является предоставление значения в виде строки, которая будет проанализирована, но это может быть неправильно истолковано в зависимости от языка и региональных стандартов потока по умолчанию. Например, в Великобритании даты указываются как «день/месяц/год», в отличие от США, где даты указываются как «месяц/день/год».

Посмотрим, что можно сделать с датами и временем.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithTime` в рабочей области/решении `Chapter08`.
2. В программе Visual Studio Code выберите `WorkingWithTime` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` удалите существующие операторы, а затем добавьте операторы для инициализации некоторых специальных значений даты/времени:

```
WriteLine("Earliest date/time value is: {0}",
  arg0: DateTime.MinValue);

WriteLine("UNIX epoch date/time value is: {0}",
  arg0: DateTime.UnixEpoch);

WriteLine("Date/time value Now is: {0}",
  arg0: DateTime.Now);
```

```
WriteLine("Date/time value Today is: {0}",
  arg0: DateTime.Today);
```

4. Запустите код и проанализируйте результат:

```
Earliest date/time value is: 01/01/0001 00:00:00
UNIX epoch date/time value is: 01/01/1970 00:00:00
Date/time value Now is: 23/04/2021 14:14:54
Date/time value Today is: 23/04/2021 00:00:00
```

5. Добавьте операторы для определения дня Рождества (Christmas) в 2021 году (или укажите любой другой год) и отобразите его различными способами:

```
DateTime christmas = new(year: 2021, month: 12, day: 25);

WriteLine("Christmas: {0}",
  arg0: christmas); // дефолтный формат

WriteLine("Christmas: {0:dddd, dd MMMM yyyy}",
  arg0: christmas); // пользовательский формат

WriteLine("Christmas is in month {0} of the year.",
  arg0: christmas.Month);

WriteLine("Christmas is day {0} of the year.",
  arg0: christmas.DayOfYear);

WriteLine("Christmas {0} is on a {1}.",
  arg0: christmas.Year,
  arg1: christmas.DayOfWeek);
```

6. Запустите код и проанализируйте результат:

```
Christmas: 25/12/2021 00:00:00
Christmas: Saturday, 25 December 2021
Christmas is in month 12 of the year.
Christmas is day 359 of the year.
Christmas 2021 is on a Saturday.
```

7. Добавьте операторы для выполнения операций сложения и вычитания с аргументом `Christmas`:

```
DateTime beforeXmas = christmas.Subtract(TimeSpan.FromDays(12));
DateTime afterXmas = christmas.AddDays(12);

WriteLine("12 days before Christmas is: {0}",
  arg0: beforeXmas);
```

```
WriteLine("12 days after Christmas is: {0}",
  arg0: afterXmas);

TimeSpan untilChristmas = christmas - DateTime.Now;

WriteLine("There are {0} days and {1} hours until Christmas.",
  arg0: untilChristmas.Days,
  arg1: untilChristmas.Hours);

WriteLine("There are {0:N0} hours until Christmas.",
  arg0: untilChristmas.TotalHours);
```

8. Запустите код и проанализируйте результат:

```
12 days before Christmas is: 13/12/2021 00:00:00
12 days after Christmas is: 06/01/2022 00:00:00
There are 245 days and 9 hours until Christmas.
There are 5,890 hours until Christmas.
```

9. Добавьте операторы для определения времени в день Рождества, когда ваши дети могут проснуться, чтобы открывать подарки, и выведите его различными способами:

```
DateTime kidsWakeUp = new(
  year: 2021, month: 12, day: 25,
  hour: 6, minute: 30, second: 0);

WriteLine("Kids wake up on Christmas: {0}",
  arg0: kidsWakeUp);

WriteLine("The kids woke me up at {0}",
  arg0: kidsWakeUp.ToShortTimeString());
```

10. Запустите код и проанализируйте результат:

```
Kids wake up on Christmas: 25/12/2021 06:30:00
The kids woke me up at 06:30
```

Глобализация с учетом дат и времени

Текущие язык и региональные стандарты определяют, как интерпретируются даты и время.

1. В начале файла `Program.cs` импортируйте пространство имен `System.Globalization`.
2. Добавьте операторы для отображения текущего языка и региональных стандартов, которые используются для отображения значений даты и времени, а затем

проанализируйте День независимости США и отобразите его различными способами:

```
WriteLine("Current culture is: {0}",
  arg0: CultureInfo.CurrentCulture.Name);

string textDate = "4 July 2021";
DateTime independenceDay = DateTime.Parse(textDate);

WriteLine("Text: {0}, DateTime: {1:d MMMM}",
  arg0: textDate,
  arg1: independenceDay);

textDate = "7/4/2021";
independenceDay = DateTime.Parse(textDate);

WriteLine("Text: {0}, DateTime: {1:d MMMM}",
  arg0: textDate,
  arg1: independenceDay);

independenceDay = DateTime.Parse(textDate,
  provider: CultureInfo.GetCultureInfo("en-US"));

WriteLine("Text: {0}, DateTime: {1:d MMMM}",
  arg0: textDate,
  arg1: independenceDay);
```

3. Запустите код и проанализируйте результат:

```
Current culture is: en-GB
Text: 4 July 2021, DateTime: 4 July
Text: 7/4/2021, DateTime: 7 April
Text: 7/4/2021, DateTime: 4 July
```

На моем компьютере текущим языком и региональными стандартами является британский английский. Если дата указана как 4 июля 2021 года, то она будет правильно проанализирована независимо от того, является ли текущий региональный стандарт британским или американским. Но если дата указана как 7/4/2021, то будет неправильно проанализирована как 7 апреля. Вы можете переопределить текущий язык и региональные стандарты, указав правильный вариант в качестве поставщика при синтаксической обработке, как показано в третьем примере выше.

4. Добавьте операторы для цикла с 2020 по 2025 год, отображающие, является ли год високосным и сколько дней в феврале, а затем покажите, приходятся ли Рождество и День независимости на летнее время:

```
for (int year = 2020; year < 2026; year++)
{
  Write($"{year} is a leap year: {DateTime.IsLeapYear(year)}. ");
```

```
    WriteLine("There are {0} days in February {1}.",
      arg0: DateTime.DaysInMonth(year: year, month: 2), arg1: year);
  }

  WriteLine("Is Christmas daylight saving time? {0}",
    arg0: christmas.IsDaylightSavingTime());

  WriteLine("Is July 4th daylight saving time? {0}",
    arg0: independenceDay.IsDaylightSavingTime());
```

5. Запустите код и проанализируйте результат:

```
2020 is a leap year: True. There are 29 days in February 2020.
2021 is a leap year: False. There are 28 days in February 2021.
2022 is a leap year: False. There are 28 days in February 2022.
2023 is a leap year: False. There are 28 days in February 2023.
2024 is a leap year: True. There are 29 days in February 2024.
2025 is a leap year: False. There are 28 days in February 2025.
Is Christmas daylight saving time? False
Is July 4th daylight saving time? True
```

Обработка дат/времени по отдельности

В .NET 6 появились новые типы для работы со значением только даты или только времени, которые называются `DateOnly` и `TimeOnly`. Лучше использовать их, чем значение `DateTime` с нулевым временем для хранения значения только даты, поскольку это безопасно с точки зрения типов и позволяет избежать неправильного применения. Кроме того, тип `DateOnly` лучше сопоставляется с типами столбцов базы данных, например со столбцом `date` в SQL Server. Тип `TimeOnly` хорошо подходит для установки будильников и планирования регулярных встреч или событий, и он сопоставляется со столбцом `time` в SQL Server.

Запланируем с их помощью вечеринку для английской королевы.

1. Добавьте операторы для определения дня рождения королевы и времени начала ее вечеринки, а затем объедините эти два значения, чтобы создать запись в календаре и не пропустить вечеринку:

```
DateOnly queensBirthday = new(year: 2022, month: 4, day: 21);
WriteLine($"The Queen's next birthday is on {queensBirthday}.");

TimeOnly partyStarts = new(hour: 20, minute: 30);
WriteLine($"The Queen's party starts at {partyStarts}.");

DateTime calendarEntry = queensBirthday.ToDateTime(partyStarts);
WriteLine($"Add to your calendar: {calendarEntry}.");
```

2. Запустите код и проанализируйте результат:

```
The Queen's next birthday is on 21/04/2022.
The Queen's party starts at 20:30.
Add to your calendar: 21/04/2022 20:30:00.
```

Сопоставление с образцом при помощи регулярных выражений

Регулярные выражения полезны для проверки на допустимость ввода пользователя. Они очень эффективны и могут становиться крайне сложными. Почти все языки программирования поддерживают регулярные выражения и применяют универсальный набор специальных символов для их определения.

Рассмотрим несколько примеров регулярных выражений.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithRegularExpressions` в рабочей области/решении `Chapter08`.
2. В Visual Studio Code выберите `WorkingWithRegularExpressions` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` импортируйте следующее пространство имен:

```
using System.Text.RegularExpressions;
```

Проверка цифр, введенных в виде текста

Мы начнем с реализации распространенного примера проверки ввода чисел.

1. Добавьте операторы для ввода пользователем своего возраста, а затем с помощью регулярного выражения убедитесь, что он правильный:

```
Write("Enter your age: ");
string? input = ReadLine();

Regex ageChecker = new(@"\d");

if (ageChecker.IsMatch(input))
{
  WriteLine("Thank you!");
}
else
{
  WriteLine($"This is not a valid age: {input}");
}
```

Обратите внимание на следующие детали кода:

- символ `@` перед строкой отключает возможность использования escape-последовательностей в строке. Escape-последовательности предваряются префиксом в виде обратного слеша (`\`). К примеру, escape-последовательность `\t` обозначает горизонтальный отступ (табуляцию), а `\n` — новую строку. При использовании регулярных выражений нам нужно отключить эту функцию;
- после того как escape-последовательности отключены с помощью символа `@`, их можно интерпретировать с помощью регулярного выражения. Например, escape-последовательность `\d` обозначает цифру. Далее вы узнаете еще больше регулярных выражений с префиксом в виде обратного слеша.

2. Запустите код, введите целое число, например, для возраста — `34`, и проанализируйте результат:

```
Enter your age: 34
Thank you!
```

3. Снова запустите код, введите `carrots` и проанализируйте результат:

```
Enter your age: carrots
This is not a valid age: carrots
```

4. Снова запустите код, введите `bob30smith` и проанализируйте результат:

```
Enter your age: bob30smith
Thank you!
```

Регулярное выражение, которое мы использовали, — `\d`, обозначает одну цифру. Однако оно не ограничивает ввод значения *до* и *после* этой цифры. Данное регулярное выражение на русском языке можно объяснить так: «Введите любые пришедшие вам в голову символы, используя хотя бы одну цифру».

В регулярных выражениях начало ввода обозначается символом каретки `^`, а конец — символом доллара `$`. Мы воспользуемся этими символами, чтобы указать, что между началом и концом ввода не ожидаем ничего другого, кроме цифры.

5. Измените регулярное выражение на `^\d$`:

```
Regex ageChecker = new(@"^\d$");
```

6. Перезапустите код и обратите внимание, что он отклоняет любой ввод, кроме одной цифры. Мы же хотим, чтобы можно было указать одну цифру или больше. В этом случае нужно добавить символ `+` после выражения `\d`.

7. Измените регулярное выражение, как показано ниже:

```
Regex ageChecker = new(@"^\d+$");
```

8. Запустите код снова и обратите внимание, что регулярное выражение допускает ввод только нуля или положительных целых чисел любой длины.

Рост производительности регулярных выражений

Для работы с регулярными выражениями типы .NET используются на всей платформе .NET и во многих приложениях, созданных с ее помощью. Они оказывают значительное влияние на производительность, но до сих пор Microsoft не уделяла им особого внимания при оптимизации.

В .NET 5 и более поздних версиях пространство имен `System.Text.RegularExpressions` было переработано в целях достижения максимальной производительности. Стандартные тесты для регулярных выражений, использующие такие методы, как `IsMatch`, теперь выполняются в пять раз быстрее. И что самое важное, вам не нужно переписывать код, чтобы получить преимущества!

Синтаксис регулярных выражений

В табл. 8.5 приведены несколько универсальных комбинаций символов, которые вы можете использовать в регулярных выражениях.

Таблица 8.5. Символы регулярных выражений

Символ	Значение	Символ	Значение
`^`	Начало ввода	`$`	Конец ввода
`\d`	Одна цифра	`\D`	Одна не цифра (любой символ, не являющийся цифрой)
`\s`	Пробел	`\S`	Не пробел (любой символ, не являющийся пробелом)
`\w`	Словесные символы	`\W`	Не словесные символы
`[A-Za-z0-9]`	Диапазон символов	`\^`	Символ `^` (каретки)
`[aeiou]`	Набор символов	`[^aeiou]`	Любой символ, кроме входящего в набор
`.`	Любой одиночный символ	`\.`	Символ `.` (точка)

Кроме того, в табл. 8.6 приведены некоторые квантификаторы регулярных выражений, влияющие на предыдущие символы.

Таблица 8.6. Квантификаторы регулярных выражений

Символ	Значение	Символ	Значение
`+`	Один или более	`?`	Один или ноль
`{3}`	Ровно три	`{3,5}`	От трех до пяти
`{3,}`	Минимум три	`{,3}`	Более трех

Примеры регулярных выражений

В табл. 8.7 приведены примеры некоторых регулярных выражений с описанием их значения.

Таблица 8.7. Примеры регулярных выражений

Выражение	Значение
`\d`	Одна цифра где-либо во вводе
`a`	Символ `a` где-либо во вводе
`Bob`	Слово `Bob` где-либо во вводе
`^Bob`	Слово `Bob` в начале ввода
`Bob$`	Слово `Bob` в конце ввода
`^\d{2}$`	Строго две цифры
`^[0-9]{2}$`	Строго две цифры
`^[A-Z]{4,}$`	Не менее четырех прописных латинских букв только в наборе символов ASCII
`^[A-Za-z]{4,}$`	Не менее четырех прописных или строчных латинских букв только в наборе символов ASCII
`^[A-Z]{2}\d{3}$`	Строго две прописные латинские буквы в наборе символов ASCII и три цифры
`^[A-Za-z\u00c0-\u017e]+$`	Как минимум одна прописная или строчная латинская буква в наборе символов ASCII или европейская буква в наборе символов Unicode, как показано в списке ниже: • ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝ; • Þßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿıŒœŠšŸŽž
`^d.g$`	Буква `d`, затем любой символ, а затем буква `g`, то есть совпадет со словами наподобие `dig`, `dog` и другими с любым символом между буквами `d` и `g`
`^d\.g$`	Буква `d`, затем точка (`.`), а затем буква `g`, поэтому совпадет только с последовательностью `d.g`

Применяйте регулярные выражения для проверки пользовательского ввода. Одни и те же регулярные выражения можно повторно использовать в других языках, таких как JavaScript и Python.

Разбивка сложных строк, разделенных запятыми

Ранее в этой главе вы узнали, как разделить простую строковую переменную с помощью запятой. Но как быть с примером наподобие этого (перечислены названия фильмов)?

```
"Monsters, Inc.","I, Tonya","Lock, Stock and Two Smoking Barrels"
```

Значение `string` указывается в двойных кавычках в начале и конце названия каждого фильма. С их помощью мы можем определить, нужно ли нам разделять по запятой в конкретном месте. Метод `Split` так не умеет, поэтому вместо него мы можем прибегнуть к регулярному выражению.

Более полное объяснение вы можете прочитать в статье на сайте Stack Overflow по ссылке https://stackoverflow.com/questions/18144431/regex-to-split-a-csv.

Чтобы включить двойные кавычки в значение `string`, мы ставим перед ними обратную косую черту (обратный слеш).

1. Добавьте операторы для хранения сложной переменной `string`, разделенной запятыми, а затем разделите ее «глупым» способом с помощью метода `Split`:

```
string films = "\"Monsters, Inc.\",\"I, Tonya\",\"Lock, Stock and Two
Smoking Barrels\"";

WriteLine($"Films to split: {films}");

string[] filmsDumb = films.Split(',');

WriteLine("Splitting with string.Split method:");

foreach (string film in filmsDumb)
{
  WriteLine(film);
}
```

2. Добавьте следующие операторы для определения регулярного выражения, чтобы разделить заголовки фильма «умным» способом и вывести их на экран:

```
WriteLine();

Regex csv = new(
  "(?:^|,)(?=[^\"]|(\")?)\"?((?(1)[^\"]*|[^,\"]*))\"?(?=,|$)");

MatchCollection filmsSmart = csv.Matches(films);

WriteLine("Splitting with regular expression:");
foreach (Match film in filmsSmart)
{
  WriteLine(film.Groups[2].Value);
}
```

3. Запустите код и проанализируйте результат:

```
Splitting with string.Split method:
"Monsters
 Inc."
"I
 Tonya"
"Lock
 Stock and Two Smoking Barrels"

Splitting with regular expression:
Monsters, Inc.
I, Tonya
Lock, Stock and Two Smoking Barrels
```

Хранение нескольких объектов в коллекциях

Коллекции — еще один распространенный тип данных. Если в переменной требуется сохранить несколько значений, то вы можете использовать коллекции.

Коллекция — это структура данных в памяти, позволяющая управлять несколькими элементами различными способами, хотя все коллекции имеют общие функции.

В табл. 8.8 перечислены наиболее распространенные типы .NET для работы с коллекциями.

Таблица 8.8. Наиболее распространенные типы .NET для работы с коллекциями

Пространство имен	Пример типа (-ов)	Описание
System.Collections	IEnumerable, IEnumerable<T>	Интерфейсы и базовые классы, используемые коллекциями
System.Collections.Generic	List<T>, Dictionary<T>, Queue<T>, Stack<T>	Типы из этой сборки и пространства имен стали применяться в версии C# 2.0 с .NET Framework 2.0, поскольку позволяют указать тип, который вы хотите сохранить, с помощью параметра типа-дженерика (что более безопасно, быстро и эффективно)
System.Collections.Concurrent	BlockingCollection, ConcurrentDictionary, ConcurrentQueue	Типы из этой сборки и пространства имен безопасны для использования в многопоточных сценариях
System.Collections.Immutable	ImmutableArray, ImmutableDictionary, ImmutableList, ImmutableQueue	Типы из этой сборки и пространства имен предназначены для сценариев, в которых содержимое коллекции никогда не должно изменяться, хотя они могут создавать измененные коллекции как новый экземпляр

Общие свойства коллекций

Коллекции реализуют интерфейс `ICollection`. Это значит, все коллекции содержат свойство `Count`, возвращающее количество объектов, содержащихся в коллекции:

```
namespace System.Collections
{
  public interface ICollection : IEnumerable
  {
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo(Array array, int index);
  }
}
```

Например, если бы у нас была коллекция `passengers`, то мы могли бы сделать следующее:

```
int howMany = passengers.Count;
```

Все коллекции реализуют интерфейс `IEnumerable`; то есть с ними можно выполнить итерацию с помощью оператора `foreach`. Они должны содержать метод `GetEnumerator`, который возвращает объект, реализующий `IEnumerator`. Это значит, возвращаемый объект должен содержать методы `MoveNext` и `Reset` для навигации по коллекции и свойство `Current`, которое содержит текущий элемент коллекции:

```
namespace System.Collections
{
  public interface IEnumerable
  {
    IEnumerator GetEnumerator();
  }
}

namespace System.Collections
{
  public interface IEnumerator
  {
    object Current { get; }
    bool MoveNext();
    void Reset();
  }
}
```

Например, чтобы выполнить действие для каждого объекта в коллекции `passengers`, мы можем написать следующий код:

```
foreach (Passenger p in passengers)
{
  // выполняем действие над каждым пассажиром
}
```

Помимо интерфейсов коллекций, основанных на объектах, существуют также интерфейсы-дженерики и классы-дженерики, где тип-дженерик определяет тип, хранящийся в коллекции:

```
namespace System.Collections.Generic
{
  public interface ICollection<T> : IEnumerable<T>, IEnumerable
  {
    int Count { get; }
    bool IsReadOnly { get; }
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int index);
    bool Remove(T item);
  }
}
```

Повышение производительности за счет обеспечения пропускной способности коллекции

Начиная с .NET 1.1, такие типы, как `StringBuilder`, имеют метод `EnsureCapacity`, который может предварительно определить размер своего внутреннего массива хранения в соответствии с ожидаемым конечным размером строки. Это повышает производительность, поскольку не нужно многократно увеличивать размер массива по мере добавления новых символов.

Начиная с версии .NET Core 2.1, типы, такие как `Dictionary<T>` и `HashSet<T>`, тоже содержат метод `EnsureCapacity`.

В .NET 6 и более поздних версиях коллекции, например `List<T>`, `Queue<T>` и `Stack<T>`, теперь тоже имеют метод `EnsureCapacity`:

```
List<string> names = new();
names.EnsureCapacity(10_000);
// загружаем десять тысяч имен в список
```

Выбор коллекции

Существует несколько различных категорий коллекции, которые следует выбирать для разных ситуаций: списки, словари, стеки, очереди, множества и многие другие узкоспециализированные коллекции.

Списки

Списки, то есть тип, реализующий интерфейс `IList<T>`, являются *упорядоченными коллекциями*:

```
namespace System.Collections.Generic
{
  [DefaultMember("Item")] // он же индексатор
  public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
  {
    T this[int index] { get; set; }
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);
  }
}
```

Интерфейс `IList<T>` наследуется от интерфейса `ICollection<T>`, поэтому он содержит свойство `Count` и метод `Add` для добавления элемента в конец коллекции, а также методы `Insert` для вставки элемента в указанную позицию и `RemoveAt` для удаления элемента в указанной позиции.

Списки весьма уместны, если вы хотите вручную управлять порядком элементов в коллекции. Каждому элементу в списке автоматически присваивается уникальный индекс (позиция). Элементы могут быть любого типа, определяемого типом `T`, и дублироваться. Индексы имеют тип `int` и начинаются с 0, поэтому первый элемент в списке имеет индекс 0 (табл. 8.9).

Таблица 8.9. Индексы элементов

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 (табл. 8.10).

Таблица 8.10. Изменение индексов элементов

Индекс	Элемент
0	London
1	Paris
2	London
3	Santiago
4	Sydney

Словари

Словари будут удобны в случае, если каждое *значение* (или объект) имеет уникальное подзначение (или искусственно созданное значение), которое в дальнейшем можно использовать в качестве *ключа* для быстрого поиска значения в коллекции. Ключ должен быть уникальным. К примеру, при сохранении списка персон вы можете применить в качестве ключа номера паспортов.

Представьте, что ключ — это своего рода указатель в словаре в реальном мире. Он позволяет быстро найти определение сло́ва, поскольку слова́ (к примеру, ключи) сортируются, и если мы ищем определение слова *manatee*, то открыли бы словарь в середине, так как буква *M* находится в середине алфавита.

Словари в программировании ведут себя аналогичным образом при выполнении поиска. Они реализуют интерфейс `IDictionary<TKey, TValue>`:

```
namespace System.Collections.Generic
{
  [DefaultMember("Item")] // индексатор
  public interface IDictionary<TKey, TValue>
    : ICollection<KeyValuePair<TKey, TValue>>,
      IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
{
    TValue this[TKey key] { get; set; }
    ICollection<TKey> Keys { get; }
    ICollection<TValue> Values { get; }
    void Add(TKey key, TValue value);
    bool ContainsKey(TKey key);
    bool Remove(TKey key);
    bool TryGetValue(TKey key, [MaybeNullWhen(false)] out TValue value);
  }
}
```

Элементы в словаре являются экземплярами `struct`, он же тип значения `KeyValuePair<TKey, TValue>`, где `TKey` — это тип ключа, а `TValue` — тип значения:

```
namespace System.Collections.Generic
{
  public readonly struct KeyValuePair<TKey, TValue>
  {
    public KeyValuePair(TKey key, TValue value);
    public TKey Key { get; }
    public TValue Value { get; }
    [EditorBrowsable(EditorBrowsableState.Never)]
    public void Deconstruct(out TKey key, out TValue value);
    public override string ToString();
  }
}
```

В примере `Dictionary<string, Person>` в качестве ключа используется `string`, а в качестве значения — экземпляр `Person`. Тип `Dictionary<string, string>` использует значения `string` и для ключа, и для значения (табл. 8.11).

Таблица 8.11. Ключи и значения

Ключ	Значение
BSA	Bob Smith
MW	Max Williams
BSB	Bob Smith
AM	Amir Mohammed

Стеки

Стеки удобно использовать в тех случаях, если вы хотите реализовать поведение *«последним пришел — первым вышел»* (last-in, first-out, LIFO). С помощью стека вы можете получить прямой доступ или удалить только один элемент в его начале, хотя можете перечислить весь стек элементов. Вы не можете получить доступ, например, ко второму элементу в стеке.

Так, текстовые редакторы используют стек для хранения последовательности действий, которые вы недавно выполнили, а затем, когда вы нажимаете сочетание клавиш Ctrl+Z, программа отменяет последнее действие в стеке, затем предпоследнее и т. д.

Очереди

Очереди удобны, если вы хотите реализовать поведение *«первым пришел — первым вышел»* (first-in, first-out, FIFO). С помощью очереди вы можете получить прямой доступ или удалить только один элемент в ее начале, хотя можете перечислить всю цепочку элементов. Вы не можете получить доступ, например, ко второму элементу в очереди.

Так, 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.

В .NET 6 появилась функция `PriorityQueue`, где каждому элементу в очереди присваивается приоритет, а также его положение в очереди.

Множества

Множества — прекрасный выбор, если вы хотите выполнять операции над множествами между двумя коллекциями. Например, у вас могут быть две коллекции с названиями городов, и вы хотите выяснить, какие имена используются в обоих множествах (так называемое *пересечение* между множествами). Элементы множества должны быть уникальными.

Краткое описание методов коллекции

Каждая коллекция имеет свой набор методов для добавления и удаления элементов, как показано в табл. 8.12.

Таблица 8.12. Наборы методов для добавления и удаления элементов

Коллекция	Методы добавления	Методы удаления	Описание
Список	`Add`, `Insert`	`Remove`, `RemoveAt`	Списки упорядочены таким образом, чтобы элементы имели целочисленную позицию индекса. Метод `Add` добавит новый элемент в конец списка, а `Insert` добавит новый элемент в указанную позицию индекса
Словарь	`Add`	`Remove`	Словари не упорядочены, поэтому элементы не имеют позиций с целочисленным индексом. Вы можете проверить, был ли использован ключ, вызвав метод `ContainsKey`
Стек	`Push`	`Pop`	Стеки всегда добавляют новый элемент на вершину стека с помощью метода `Push`. Первый элемент находится внизу. Элементы всегда удаляются с вершины стека с помощью метода `Pop`. Вызовите метод `Peek`, чтобы увидеть это значение, не удаляя его
Очередь	`Enqueue`	`Dequeue`	Очереди всегда добавляют новый элемент в конец очереди с помощью метода `Enqueue`. Первый элемент находится в начале очереди. Элементы всегда удаляются из начала очереди с помощью метода `Dequeue`. Вызовите метод `Peek`, чтобы увидеть это значение, не удаляя его

Работа со списками

Рассмотрим пример работы со списками.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithCollections` в рабочей области/решении `Chapter08`.

2. В программе Visual Studio Code выберите `WorkingWithCollections` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` удалите существующие операторы, а затем определите функцию для вывода коллекции значений `string` с заголовком:

```
static void Output(string title, IEnumerable<string> collection)
{
  WriteLine(title);
  foreach (string item in collection)
  {
    WriteLine($" {item}");
  }
}
```

4. Определите статический метод `WorkingWithLists`, чтобы проиллюстрировать некоторые из распространенных способов определения и работы со списками:

```
static void WorkingWithLists()
{
  // простой синтаксис для создания списка и добавления трех элементов
  List<string> cities = new();
  cities.Add("London");
  cities.Add("Paris");
  cities.Add("Milan");

  /* альтернативный синтаксис, который преобразуется компилятором
  в три вышеприведенных вызова метода Add
  List<string> cities = new()
    { "London", "Paris", "Milan" };
  */

  /* альтернативный синтаксис, передающий массив строковых значений
  методу AddRange
  List<string> cities = new();
  cities.AddRange(new[] { "London", "Paris", "Milan" });
  */

  Output("Initial list", cities);

  WriteLine($"The first city is {cities[0]}.");
  WriteLine($"The last city is {cities[cities.Count - 1]}.");

  cities.Insert(0, "Sydney");

  Output("After inserting Sydney at index 0", cities);

  cities.RemoveAt(1);
  cities.Remove("Milan");

  Output("After removing two cities", cities);
}
```

5. В начале файла `Program.cs` после импорта пространства имен вызовите метод `WorkingWithLists`:

```
WorkingWithLists();
```

6. Запустите код и проанализируйте результат:

```
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

Рассмотрим пример работы со словарями.

1. В файле `Program.cs` определите статический метод `WorkingWithDictionaries`, чтобы проиллюстрировать некоторые из распространенных способов работы со словарями, например поиск определений слов:

```
static void WorkingWithDictionaries()
{
  Dictionary<string, string> keywords = new();

  // добавление с использованием именованных параметров
  keywords.Add(key: "int", value: "32-bit integer data type");

  // добавление с использованием позиционных параметров
  keywords.Add("long", "64-bit integer data type");
  keywords.Add("float", "Single precision floating point number");

  /* альтернативный синтаксис; компилятор преобразует в вызовы метода Add
  Dictionary<string, string> keywords = new()
  {
    { "int", "32-bit integer data type" },
    { "long", "64-bit integer data type" },
```

```
    { "float", "Single precision floating point number" },
  }; */

  /* альтернативный синтаксис; компилятор преобразует в вызовы метода Add
  Dictionary<string, string> keywords = new()
  {
    ["int"] = "32-bit integer data type",
    ["long"] = "64-bit integer data type",
    ["float"] = "Single precision floating point number",
    // последняя запятая необязательна
  }; */

  Output("Dictionary keys:", keywords.Keys);
  Output("Dictionary values:", keywords.Values);

  WriteLine("Keywords and their definitions");
  foreach (KeyValuePair<string, string> item in keywords)
  {
    WriteLine($" {item.Key}: {item.Value}");
  }

  // ищем значение по ключу
  string key = "long";
  WriteLine($"The definition of {key} is {keywords[key]}");
}
```

2. В начале файла `Program.cs` закомментируйте предыдущий вызов метода, а затем вызовите метод `WorkingWithDictionaries`:

```
// WorkingWithLists();
WorkingWithDictionaries();
```

3. Запустите код и проанализируйте результат:

```
Dictionary keys:
  int
  long
  float
Dictionary values:
  32-bit integer data type
  64-bit integer data type
  Single precision floating point number
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
```

Работа с очередями

Рассмотрим пример работы с очередями.

1. В файле `Program.cs` определите статический метод `WorkingWithQueues`, чтобы проиллюстрировать некоторые из распространенных способов работы с очередями, например обслуживание клиентов в очереди за кофе:

```
static void WorkingWithQueues()
{
  Queue<string> coffee = new();

  coffee.Enqueue("Damir"); // начало очереди
  coffee.Enqueue("Andrea");
  coffee.Enqueue("Ronald");
  coffee.Enqueue("Amin");
  coffee.Enqueue("Irina"); // конец очереди

  Output("Initial queue from front to back", coffee);

  // сервер обрабатывает следующего человека в очереди
  string served = coffee.Dequeue();
  WriteLine($"Served: {served}.");

  // сервер обрабатывает следующего человека в очереди
  served = coffee.Dequeue();
  WriteLine($"Served: {served}.");

  Output("Current queue from front to back", coffee);

  WriteLine($"{coffee.Peek()} is next in line.");

  Output("Current queue from front to back", coffee);
}
```

2. В начале файла `Program.cs` закомментируйте предыдущие вызовы методов и вызовите метод `WorkingWithQueues`.
3. Запустите код и проанализируйте результат:

```
Initial queue from front to back
  Damir
  Andrea
  Ronald
  Amin
  Irina
Served: Damir.
Served: Andrea.
```

```
Current queue from front to back
  Ronald
  Amin
  Irina
Ronald is next in line.
Current queue from front to back
  Ronald
  Amin
  Irina
```

4. Определите статический метод `OutputPQ`:

```
static void OutputPQ<TElement, TPriority>(string title,
  IEnumerable<(TElement Element, TPriority Priority)> collection)
{
  WriteLine(title);
  foreach ((TElement, TPriority) item in collection)
  {
    WriteLine($" {item.Item1}: {item.Item2}");
  }
}
```

Обратите внимание, что метод `OutputPQ` является дженериком. Вы можете указать два типа, используемых в кортежах, которые передаются в качестве коллекции `collection`.

5. Определите статический метод `WorkingWithPriorityQueues`:

```
static void WorkingWithPriorityQueues()
{
  PriorityQueue<string, int> vaccine = new();

   // добавляем несколько человек
   // 1 = высокоприоритетные люди в возрасте 70 лет или со слабым здоровьем
   // 2 = средний приоритет, например люди среднего возраста
   // 3 = низкий приоритет, например подростки и молодые люди
  vaccine.Enqueue("Pamela", 1); // моя мама (70 лет)
  vaccine.Enqueue("Rebecca", 3); // моя племянница (подросток)
  vaccine.Enqueue("Juliet", 2); // моя сестра (40 лет)
  vaccine.Enqueue("Ian", 1); // мой папа (70 лет)

  OutputPQ("Current queue for vaccination:", vaccine.UnorderedItems);

  WriteLine($"{vaccine.Dequeue()} has been vaccinated.");
  WriteLine($"{vaccine.Dequeue()} has been vaccinated.");

  OutputPQ("Current queue for vaccination:", vaccine.UnorderedItems);
  WriteLine($"{vaccine.Dequeue()} has been vaccinated.");
```

```
    vaccine.Enqueue("Mark", 2); // я (40 лет)
    WriteLine($"{vaccine.Peek()} will be next to be vaccinated.");

    OutputPQ("Current queue for vaccination:", vaccine.UnorderedItems);
  }
```

6. В начале файла `Program.cs` закомментируйте предыдущие вызовы методов и вызовите метод `WorkingWithPriorityQueues`.
7. Запустите код и проанализируйте результат:

```
Current queue for vaccination:
  Pamela: 1
  Rebecca: 3
  Juliet: 2
  Ian: 1
Pamela has been vaccinated.
Ian has been vaccinated.
Current queue for vaccination:
  Juliet: 2
  Rebecca: 3
Juliet has been vaccinated.
Mark will be next to be vaccinated.
Current queue for vaccination:
  Mark: 2
  Rebecca: 3
```

Сортировка коллекций

Класс `List<T>` можно отсортировать, вызвав вручную его метод `Sort` (но помните, что позиции индексов всех элементов будут изменены). Ручную сортировку списка значений `string` или других встроенных типов также довольно просто выполнить, но если вы создаете коллекцию элементов собственного типа, то он должен реализовать интерфейс `IComparable`. О том, как это сделать, вы узнали в главе 6.

Коллекции `Stack<T>` и `Queue<T>` не могут быть отсортированы, поскольку обычно это не требуется. Вы же, вероятно, никогда не будете сортировать очередь посетителей гостиницы. Но в некоторых случаях вам может понадобиться отсортировать словарь или множество.

Иногда бывает полезно иметь автоматически сортируемую коллекцию, то есть такую, которая по мере добавления или удаления элементов сама будет поддерживать их упорядоченность.

Существует несколько автоматически сортирующихся коллекций. Различия между ними часто незначительны, но могут влиять на загруженность памяти и производительность вашего приложения, поэтому рекомендуется выбирать коллекции, наиболее подходящие под ваши требования.

В табл. 8.13 приведены автоматически сортирующиеся коллекции, которые используются наиболее часто.

Таблица 8.13

Коллекция	Описание
`SortedDictionary<TKey, TValue>`	Представляет собой коллекцию пар «ключ — значение», которые сортируются по ключу
`SortedList<TKey, TValue>`	Представляет собой коллекцию пар «ключ — значение», которые сортируются по ключу
`SortedSet<T>`	Представляет собой коллекцию уникальных объектов, которые сопровождаются в отсортированном порядке

Более специализированные коллекции

Существует еще несколько коллекций для особых случаев.

Работа с компактным массивом битовых значений

Коллекция `System.Collections.BitArray` управляет компактным массивом битовых значений, которые представлены в виде булеанов, где `true` означает, что бит включен (значение равно 1), а `false` означает, что бит выключен (значение равно 0).

Работа с эффективными списками

Коллекция `System.Collections.Generics.LinkedList<T>` представляет собой двунаправленный список, в котором каждый элемент имеет ссылку на предыдущий и следующий элементы. Они обеспечивают более высокую производительность по сравнению с классом `List<T>` для сценариев, в которых вы будете часто вставлять и удалять элементы из середины списка. В `LinkedList<T>` элементы не нужно переставлять в памяти.

Использование неизменяемых коллекций

Иногда необходимо создать коллекцию *неизменяемой*. Это значит, ее члены не могут изменяться, то есть вы не можете добавлять или удалять их.

Если вы импортируете пространство имен `System.Collections.Immutable`, то у любой коллекции, реализующей интерфейс `IEnumerable<T>`, появится шесть методов расширения для преобразования ее в неизменяемый список, словарь, хеш-множество и т. д.

Рассмотрим простой пример.

1. В проекте `WorkingWithCollections` в файле `Program.cs` импортируйте пространство имен `System.Collections.Immutable`.
2. В методе `WorkingWithLists` добавьте операторы в конец метода, чтобы преобразовать список `cities` в неизменяемый, а затем добавьте в него новый город:

```
ImmutableList<string> immutableCities = cities.ToImmutableList();
ImmutableList<string> newList = immutableCities.Add("Rio");

Output("Immutable list of cities:", immutableCities);
Output("New list of cities:", newList);
```

3. В начале файла `Program.cs` закомментируйте предыдущие вызовы методов и раскомментируйте вызов метода `WorkingWithLists`.
4. Запустите код, проанализируйте результат и обратите внимание, что неизменяемый список городов не изменяется при вызове метода `Add`, он возвращает новый список с вновь добавленным городом:

```
Immutable list of cities:
  Sydney
  Paris
New list of cities:
  Sydney
  Paris
  Rio
```

Для повышения производительности многие приложения хранят общую копию часто используемых объектов в центральном кэше. Чтобы обезопасить работу нескольких потоков с этими объектами, зная, что они не изменятся, вы должны сделать их неизменяемыми или использовать тип многопоточной коллекции. Дополнительную информацию вы можете прочитать по следующей ссылке: https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.

Эффективные приемы работы с коллекциями

Допустим, вам нужно создать метод для обработки коллекции. Для максимальной гибкости вы можете объявить входной параметр как интерфейс `IEnumerable<T>` и сделать метод общим:

```
void ProcessCollection<T>(IEnumerable<T> collection)
{
  // обрабатываем элементы коллекции,
  // по возможности используя оператор foreach
}
```

Можно передать в этот метод массив, список, очередь, стек или что-либо еще, реализующее `IEnumerable<T>`, и он будет обрабатывать элементы. Однако за гибкость передачи любой коллекции в этот метод приходится платить производительностью.

Одна из проблем производительности `IEnumerable<T>` также является одним из его преимуществ: отложенное выполнение, также известное как отложенная загрузка. Типы, реализующие этот интерфейс, не обязаны реализовывать отложенное выполнение, но многие делают это.

Но худшая проблема производительности интерфейса `IEnumerable<T>` заключается в том, что при итерации приходится выделять объект в куче. Чтобы избежать этого выделения памяти, вы должны определить свой метод, используя конкретный тип, как показано ниже:

```
void ProcessCollection<T>(List<T> collection)
{
  // обрабатываем элементы коллекции,
  // по возможности используя оператор foreach
}
```

При этом будет использоваться метод `List<T>.Enumerator GetEnumerator()`, который возвращает `struct` вместо метода `IEnumerator<T> GetEnumerator()`, который возвращает ссылочный тип. Ваш код будет выполняться в два-три раза быстрее и потребует меньше памяти. Как и в случае со всеми рекомендациями, связанными с производительностью, вы должны убедиться в получении преимущества, выполнив тесты производительности на реальном коде в среде продукта. О том, как это сделать, вы узнаете в главе 12.

Работа с интервалами, индексами и диапазонами

Одной из целей компании Microsoft относительно .NET Core 2.1 было повышение производительности и улучшение использования ресурсов. Ключевая функция .NET, которая позволяет это сделать, — тип `Span<T>`.

Управление памятью с помощью интервалов

При работе с массивами вы часто будете создавать новые копии подмножеств из существующих объектов так, чтобы иметь возможность обрабатывать только подмножество. Это неэффективно, поскольку в памяти должны быть созданы дубликаты объектов.

Если вам нужно работать с подмножествами массива, то можно использовать *интервал* — он подобен «окну» в исходном массиве и является более эффективным

решением с точки зрения использования памяти и повышения производительности. Интервалы работают только с массивами, а не с коллекциями, поскольку память должна быть непрерывной.

Прежде чем мы рассмотрим интервалы более подробно, нам необходимо освоить некоторые новые объекты: индексы и диапазоны.

Идентификация позиций с помощью типа Index

В языке C# 8.0 представлены две новые функции, позволяющие идентифицировать индекс элемента в массиве и диапазон элементов с помощью двух индексов.

В предыдущем подразделе вы узнали, что к объектам в списке можно получить доступ, передав целое число в их индексатор:

```
int index = 3;
Person p = people[index]; // четвертый человек в массиве
char letter = name[index]; // четвертая буква имени
```

Тип значения `Index` позволяет более формально идентифицировать позицию, к тому же поддерживает отсчет с конца:

```
// два способа определить один и тот же индекс, 3 с начала
Index i1 = new(value: 3); // считаем с начала
Index i2 = 3; // с помощью операции неявного преобразования int

// два способа определить один и тот же индекс, 5 от конца
Index i3 = new(value: 5, fromEnd: true);
Index i4 = ^5; // с помощью операции каретки
```

Идентификация диапазонов с помощью типа Range

Тип значения `Range` использует значения `Index` для указания начала и конца диапазона, с помощью конструктора, синтаксиса языка C# или статических методов:

```
Range r1 = new(start: new Index(3), end: new Index(7));
Range r2 = new(start: 3, end: 7); // с помощью неявного целочисленного
преобразования
Range r3 = 3..7; // используя синтаксис C# 8.0 или более поздней версии
Range r4 = Range.StartAt(3); // от индекса 3 до последнего индекса
Range r5 = 3..; // от индекса 3 до последнего индекса
Range r6 = Range.EndAt(3); // от индекса 0 до индекса 3
Range r7 = ..3; // от индекса 0 до индекса 3
```

Для упрощения работы с диапазонами к значению `string` (которое внутренне использует массив `char`), массиву `int` и интервалам были добавлены мето-

ды расширения. Они принимают диапазон в качестве параметра и возвращают `Span<T>`, что делает их очень эффективными с точки зрения управления памятью.

Использование индексов, диапазонов и интервалов

Рассмотрим пример того, как с помощью индексов и диапазонов возвращать интервалы.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithRanges` в рабочей области/решении `Chapter08`.
2. В программе Visual Studio Code выберите `WorkingWithRanges` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` добавьте следующие операторы, чтобы сравнить использование метода `Substring` типа `string` и диапазонов для извлечения фрагментов чьего-либо имени:

```
string name = "Samantha Jones";

// использование метода Substring

int lengthOfFirst = name.IndexOf(, ,);
int lengthOfLast = name.Length - lengthOfFirst - 1;

string firstName = name.Substring(
  startIndex: 0,
  length: lengthOfFirst);

string lastName = name.Substring(
  startIndex: name.Length - lengthOfLast,
  length: lengthOfLast);

WriteLine($"First name: {firstName}, Last name: {lastName}");

// использование интервалов

ReadOnlySpan<char> nameAsSpan = name.AsSpan();
ReadOnlySpan<char> firstNameSpan = nameAsSpan[0..lengthOfFirst];
ReadOnlySpan<char> lastNameSpan = nameAsSpan[^lengthOfLast..^0];

WriteLine("First name: {0}, Last name: {1}",
  arg0: firstNameSpan.ToString(),
  arg1: lastNameSpan.ToString());
```

4. Запустите код и проанализируйте результат:

```
First name: Samantha, Last name: Jones
First name: Samantha, Last name: Jones
```

Работа с сетевыми ресурсами

Иногда возникает необходимость работать с сетевыми ресурсами. В табл. 8.14 описаны наиболее распространенные типы в .NET, которые предназначены для такой работы.

Таблица 8.14. Наиболее распространенные типы в .NET, предназначенные для работы с сетевыми ресурсами

Пространство имен	Пример типа (-ов)	Описание
`System.NET`	`Dns`, `Uri`, `Cookie`, `WebClient`, `IPAddress`	Предназначены для работы с DNS-серверами, URI, IP-адресами и т. д.
`System.NET`	`FtpStatusCode`, `FtpWebRequest`, `FtpWebResponse`	Предназначены для работы с FTP-серверами
`System.NET`	`HttpStatusCode`, `HttpWebRequest`, `HttpWebResponse`	Предназначены для работы с HTTP-серверами, то есть сайтами и сервисами. Типы из `System.NET.Http` проще в использовании
`System.NET.Http`	`HttpClient`, `HttpMethod`, `HttpRequestMessage`, `HttpResponseMessage`	Предназначены для работы с HTTP-серверами, то есть сайтами и сервисами. О том, как их использовать, вы узнаете в главе 16
`System.NET.Mail`	`Attachment`, `MailAddress`, `MailMessage`, `SmtpClient`	Предназначены для работы с SMTP-серверами, то есть отправки сообщений электронной почты
`System.NET.NET-workInformation`	`IPStatus`, `NetworkChange`, `Ping`, `TcpStatistics`	Предназначены для работы с низкоуровневыми сетевыми протоколами

Работа с URI, DNS и IP-адресами

Рассмотрим некоторые распространенные типы для работы с сетевыми ресурсами.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithNet-workResources` в рабочей области/решении `Chapter08`.
2. В программе Visual Studio Code выберите `WorkingWithNetworkResources` в качестве активного проекта OmniSharp.
3. В начале файла `Program.cs` импортируйте пространство имен для работы с сетью:

```
using System.NET; // IPHostEntry, Dns, IPAddress
```

4. Добавьте следующие операторы, чтобы предложить пользователю ввести адрес сайта, а затем с помощью типа `Uri` разделите его на части, включая схему (HTTP, FTP и т. д.), номер порта и имя хоста:

```
Write("Enter a valid web address: ");
string? url = ReadLine();

if (string.IsNullOrWhiteSpace(url))
{
  url = "https://stackoverflow.com/search?q=securestring";
}

Uri uri = new(url);

WriteLine($"URL: {url}");
WriteLine($"Scheme: {uri.Scheme}");
WriteLine($"Port: {uri.Port}");
WriteLine($"Host: {uri.Host}");
WriteLine($"Path: {uri.AbsolutePath}");
WriteLine($"Query: {uri.Query}");
```

Для удобства код также разрешает пользователю просто нажать клавишу Enter, чтобы использовать пример URL.

5. Запустите код, введите правильный адрес сайта или нажмите клавишу Enter и проанализируйте результат:

```
Enter a valid web address:
URL: https://stackoverflow.com/search?q=securestring
Scheme: https
Port: 443
Host: stackoverflow.com
Path: /search
Query: ?q=securestring
```

6. Добавьте операторы, чтобы получить IP-адрес для введенного сайта:

```
IPHostEntry entry = Dns.GetHostEntry(uri.Host);
WriteLine($"{entry.HostName} has the following IP addresses:");
foreach (IPAddress address in entry.AddressList)
{
  WriteLine($" {address} ({address.AddressFamily})");
}
```

7. Запустите код, введите правильный адрес сайта или нажмите клавишу Enter и проанализируйте результат:

```
stackoverflow.com has the following IP addresses:
  151.101.193.69 (InterNetwork)
  151.101.129.69 (InterNetwork)
  151.101.1.69 (InterNetwork)
  151.101.65.69 (InterNetwork)
```

Проверка соединения с сервером

Теперь необходимо добавить программный код для проверки соединения с веб-сервером.

1. Импортируйте пространство имен для получения дополнительной информации о сетях:

```
using System.NET.NETworkInformation; // Ping, PingReply, IPStatus
```

2. Добавьте операторы для проверки соединения с сайтом, адрес которого был введен:

```
try
{
  Ping ping = new();
  WriteLine("Pinging server. Please wait...");
  PingReply reply = ping.Send(uri.Host);

  WriteLine($"{uri.Host} was pinged and replied: {reply.Status}.");
  if (reply.Status == IPStatus.Success)
  {
    WriteLine("Reply from {0} took {1:N0}ms",
      arg0: reply.Address,
      arg1: reply.RoundtripTime);
  }
}
catch (Exception ex)
{
  WriteLine($"{ex.GetType().ToString()} says {ex.Message}");
}
```

3. Запустите код, нажмите клавишу Enter и проанализируйте результат (вывод в macOS):

```
Pinging server. Please wait...
stackoverflow.com was pinged and replied: Success.
Reply from 151.101.193.69 took 18ms took 136ms
```

4. Снова запустите код, но в этот раз введите адрес http://google.com:

```
Enter a valid web address: http://google.com
URL: http://google.com
Scheme: http
Port: 80
Host: google.com
Path: /
```

```
Query:
google.com has the following IP addresses:
  2a00:1450:4009:807::200e (InterNetworkV6)
  216.58.204.238 (InterNetwork)
Pinging server. Please wait...
google.com was pinged and replied: Success.
Reply from 2a00:1450:4009:807::200e took 24ms
```

Работа с отражением и атрибутами

Отражение (или *рефлексия*) означает процесс, в ходе которого программа может отслеживать и изменять собственную структуру и поведение во время выполнения. Сборка состоит из четырех частей:

- *метаданные и манифест сборки* — имя, сборка и версия файла, ссылки на сборки и т. д.;
- *метаданные типов* — 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 содержат информацию о вашем коде. Они автоматически генерируются из вашего кода (например, информация о типах и членах) или применяются к вашему коду с помощью атрибутов.

Атрибуты могут применяться на нескольких уровнях: к сборкам, типам и их членам:

```
// атрибут уровня сборки
[assembly: AssemblyTitle("Working with Reflection")]

// атрибут уровня типа
[Serializable]
public class Person
{
  // атрибут уровня члена
  [Obsolete("Deprecated: use Run instead.")]
  public void Walk()
  {
...
```

Программирование на основе атрибутов часто используется в таких моделях приложений, как ASP.NET Core, для обеспечения таких функций, как маршрутизация, безопасность и кэширование.

Версии сборок

Номера версий в .NET представляют собой комбинацию из трех чисел с двумя необязательными дополнениями. Если следовать принципам семантического версионирования, то эти три числа обозначают следующее:

- *мажорная версия* — критические изменения;
- *минорная версия* — некритические изменения, включая новые функции и исправления ошибок;
- *патч-версия* — некритические исправления ошибок.

При обновлении пакета NuGet, который вы уже используете в проекте, в целях безопасности вы должны указать дополнительный флаг, чтобы обновляться только до самой высокой минорной версии во избежание критических изменений, или до самой высокой патч-версии, если вы очень осторожны и хотите получать только исправления ошибок, как показано в следующих командах: Update-Package Newtonsoft.Json -ToHighestMinor или Update-Package Newtonsoft.Json -ToHighestPatch.

При необходимости версия может включать в себя:

- *предварительный выпуск* — неподдерживаемые предварительные версии;
- *номер сборки* — ночные сборки.

Следуйте правилам семантического версионирования, описанным на сайте http://semver.org.

Чтение метаданных сборки

Рассмотрим пример работы с атрибутами.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithReflection` в рабочей области/решении `Chapter08`.
2. В программе Visual Studio Code выберите `WorkingWithReflection` в качестве активного проекта OmniSharp.
3. В начале файла `Program.cs` импортируйте пространство имен для отражения:

```
using System.Reflection; // сборка
```

4. Введите следующие операторы, чтобы получить сборку консольного приложения, вывести ее имя и местоположение, а также получить все атрибуты уровня сборки и вывести их типы:

```
WriteLine("Assembly metadata:");
Assembly? assembly = Assembly.GetEntryAssembly();
if (assembly is null)
{
  WriteLine("Failed to get entry assembly.");
  return;
}

WriteLine($" Full name: {assembly.FullName}");
WriteLine($" Location: {assembly.Location}");

IEnumerable<Attribute> attributes = assembly.GetCustomAttributes();

WriteLine($" Assembly-level attributes:");
foreach (Attribute a in attributes)
{
  WriteLine($" {a.GetType()}");
}
```

5. Запустите код и проанализируйте результат:

```
Assembly metadata:
  Full name: WorkingWithReflection, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null
  Location: /Users/markjprice/Code/Chapter08/WorkingWithReflection/bin/
    Debug/net6.0/WorkingWithReflection.dll
  Assembly-level attributes:
    System.Runtime.CompilerServices.CompilationRelaxationsAttribute
    System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
    System.Diagnostics.DebuggableAttribute
    System.Runtime.Versioning.TargetFrameworkAttribute
    System.Reflection.AssemblyCompanyAttribute
    System.Reflection.AssemblyConfigurationAttribute
    System.Reflection.AssemblyFileVersionAttribute
    System.Reflection.AssemblyInformationalVersionAttribute
    System.Reflection.AssemblyProductAttribute
    System.Reflection.AssemblyTitleAttribute
```

Обратите внимание, что, поскольку полное имя сборки должно однозначно ее идентифицировать, оно представляет собой комбинацию таких элементов, как:

- *имя*, например `WorkingWithReflection`;
- *версия*, например `1.0.0.0`;
- *язык и региональные параметры*, например `neutral`;
- *токен открытого ключа*, хотя он может быть и `null`.

Теперь, зная некоторые атрибуты, определенные для сборки, мы можем специально их запросить.

6. Добавьте операторы, чтобы получить классы `AssemblyInformationalVersionAttribute` и `AssemblyCompanyAttribute`, а затем вывести их значения:

```
AssemblyInformationalVersionAttribute? version = assembly
  .GetCustomAttribute<AssemblyInformationalVersionAttribute>();

WriteLine($" Version: {version?.InformationalVersion}");

AssemblyCompanyAttribute? company = assembly
  .GetCustomAttribute<AssemblyCompanyAttribute>();

WriteLine($" Company: {company?.Company}");
```

7. Запустите код и проанализируйте результат:

```
  Version: 1.0.0
  Company: WorkingWithReflection
```

Итак, если вы не зададите версию, то по умолчанию она будет 1.0.0, а если не зададите компанию, то по умолчанию будет указано имя сборки. Явно установим данную информацию. Устаревший способ установки этих значений в .NET Framework заключался в добавлении атрибутов в файл исходного кода C#:

```
[assembly: AssemblyCompany("Packt Publishing")]
[assembly: AssemblyInformationalVersion("1.3.0")]
```

Компилятор Roslyn, используемый в .NET Core, устанавливает эти атрибуты автоматически, ввиду чего мы не можем применить старый способ. Вместо этого они должны быть установлены в файле проекта.

8. Отредактируйте файл проекта `WorkingWithReflection.csproj`, чтобы добавить элементы для версии и компании, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <Version>6.3.12</Version>
    <Company>Packt Publishing</Company>
  </PropertyGroup>

</Project>
```

9. Запустите код и проанализируйте результат:

```
  Version: 6.3.12
  Company: Packt Publishing
```

Создание пользовательских атрибутов

Вы можете определить собственные атрибуты, наследуя от класса `Attribute`.

1. Добавьте в проект файл класса `CoderAttribute.cs`.
2. Определите класс атрибута, который может дополнять либо классы, либо методы двумя свойствами для хранения имени кодировщика и даты последнего изменения:

```
namespace Packt.Shared;

[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);
  }
}
```

3. В файле `Program.cs` импортируйте несколько пространств имен:

```
using System.Runtime.CompilerServices; // CompilerGeneratedAttribute
using Packt.Shared; // CoderAttribute
```

4. В конце файла `Program.cs` добавьте класс с методом и дополните метод атрибутом `Coder` с данными о двух кодировщиках:

```
class Animal
{
  [Coder("Mark Price", "22 August 2021")]
  [Coder("Johnni Rasmussen", "13 September 2021")]
  public void Speak()
  {
    WriteLine("Woof...");
  }
}
```

5. В файле `Program.cs` над классом `Animal` добавьте код для получения типов, перечислите их элементы, прочитайте все атрибуты `Coder` этих элементов и выведите информацию:

```
WriteLine();
WriteLine($"* Types:");
Type[] types = assembly.GetTypes();
```

```
foreach (Type type in types)
{
  WriteLine();
  WriteLine($"Type: {type.FullName}");
  MemberInfo[] members = type.GetMembers();

  foreach (MemberInfo member in members)
  {
    WriteLine("{0}: {1} ({2})",
      arg0: member.MemberType,
      arg1: member.Name,
      arg2: member.DeclaringType?.Name);

    IOrderedEnumerable<CoderAttribute> coders =
      member.GetCustomAttributes<CoderAttribute>()
      .OrderByDescending(c => c.LastModified);

    foreach (CoderAttribute coder in coders)
    {
      WriteLine("-> Modified by {0} on {1}",
        coder.Coder, coder.LastModified.ToShortDateString());
    }
  }
}
```

6. Запустите код и проанализируйте результат:

```
* Types:
...
Type: Animal
Method: Speak (Animal)
-> Modified by Johnni Rasmussen on 13/09/2021
-> Modified by Mark Price on 22/08/2021
Method: GetType (Object)
Method: ToString (Object)
Method: Equals (Object)
Method: GetHashCode (Object)
Constructor: .ctor (Program)
...
Type: <Program>$+<>c
Method: GetType (Object)
Method: ToString (Object)
Method: Equals (Object)
Method: GetHashCode (Object)
Constructor: .ctor (<>c)
Field: <>9 (<>c)
Field: <>9__0_0 (<>c)
```

Что собой представляет тип `<Program>$+<>c`?

Это сгенерированный компилятором *класс отображения*. Оператор `<>` указывает на то, что он сгенерирован компилятором, а `c` — на то, что это класс отображения.

Это недокументированные детали реализации компилятора, которые могут измениться в любой момент. Вы можете игнорировать их, поэтому в качестве дополнительной задачи добавьте операторы в консольное приложение для фильтрации сгенерированных компилятором типов, пропустив типы, дополненные атрибутом `CompilerGeneratedAttribute`.

Возможности отражения

Выше мы рассмотрели небольшой пример того, чего можно достичь с помощью отражения. Мы использовали отражение только для чтения метаданных из нашего кода. Помимо этого, оно может динамически:

- загружать сборки, на которые в данный момент нет ссылок: https://docs.microsoft.com/ru-ru/dotnet/standard/assembly/unloadability-howto;
- выполнять код: https://docs.microsoft.com/ru-ru/dotnet/api/system.reflection.methodbase.invoke;
- генерировать новый код и сборки: https://docs.microsoft.com/ru-ru/dotnet/api/system.reflection.emit.assemblybuilder.

Работа с изображениями

ImageSharp — это сторонняя кросс-платформенная библиотека 2D-графики. Когда версия .NET Core 1.0 находилась в разработке, были отрицательные отзывы от сообщества об отсутствии пространства имен `System.Drawing` для работы с 2D-изображениями.

Проект *ImageSharp* был начат, чтобы восполнить этот пробел для современных приложений .NET.

В официальной документации по `System.Drawing` компания Microsoft сообщает: «Пространство имен `System.Drawing` не рекомендуется для новой разработки, так как оно не поддерживается в сервисах Windows или ASP.NET и не является кросс-платформенным. ImageSharp и SkiaSharp рекомендуются в качестве альтернативы».

Рассмотрим пример, чего можно достичь, используя ImageSharp.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithImages` в рабочей области/решении `Chapter08`.
2. В программе Visual Studio Code выберите `WorkingWithImages` в качестве активного проекта OmniSharp.
3. Создайте папку `images` и загрузите девять изображений по следующей ссылке: https://github.com/markjprice/cs9dotnet5/tree/master/Assets/Categories.

4. Добавьте ссылку на пакет для `SixLabors.ImageSharp`:

```
<ItemGroup>
  <PackageReference Include="SixLabors.ImageSharp" Version="1.0.3" />
</ItemGroup>
```

5. Соберите проект `WorkingWithImages`.
6. В начале файла `Program.cs` импортируйте несколько пространств имен для работы с изображениями:

```
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Processing;
```

7. В файле `Program.cs` введите операторы для преобразования всех файлов в папке изображений в миниатюры, представленные в оттенках серого с размером в одну десятую:

```
string imagesFolder = Path.Combine(
  Environment.CurrentDirectory, "images");

IEnumerable<string> images =
  Directory.EnumerateFiles(imagesFolder);

foreach (string imagePath in images)
{
  string thumbnailPath = Path.Combine(
    Environment.CurrentDirectory, "images",
    Path.GetFileNameWithoutExtension(imagePath)
    + "-thumbnail" + Path.GetExtension(imagePath));

  using (Image image = Image.Load(imagePath))
  {
    image.Mutate(x => x.Resize(image.Width / 10, image.Height / 10));
    image.Mutate(x => x.Grayscale());
    image.Save(thumbnailPath);
  }
}
WriteLine("Image processing complete. View the images folder.");
```

8. Запустите код.
9. В файловой системе откройте папку `images` и обратите внимание на гораздо меньшие по размеру в байтах миниатюры, представленные в оттенках серого, как показано на рис. 8.1.

ImageSharp также обладает пакетами NuGet для программного рисования изображений и работы с изображениями в Интернете:

- `SixLabors.ImageSharp.Drawing`;
- `SixLabors.ImageSharp.Web`.

Рис. 8.1. Изображения после обработки

Интернационализация кода

Интернационализация — это процесс, позволяющий вашему приложению правильно работать во всем мире. Он состоит из двух частей: *глобализации* и *локализации*.

Глобализация заключается в написании кода для поддержки разных языковых и региональных параметров. При разработке приложений важно учитывать язык и регион, поскольку, к примеру, форматы даты и валюты в Квебеке и Париже различаются, 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. Например, в коде `da-DK` символы `da` указывают на датский язык, а символы `DK` определяют страну — Данию; в коде `fr-CA` символы `fr` указывают на французский язык, а `CA` определяют страну — Канаду.

ISO — не аббревиатура. Это отсылка к греческому слову *isos*, которое в переводе означает «равный».

Локализация — это настройка пользовательского интерфейса для реализации поддержки языка, например изменение названия кнопки `Закрыть` на `Close (en)` или `Fermer (fr)`. Поскольку локализация больше связана с языком, ей не всегда нужно учитывать регион. Ирония, однако, в том, что само слово «стандартизация» в `en-US` (standardization) и `en-GB` (standardisation) намекает, что знать о регионе все же полезно.

Обнаружение и изменение региональных настроек

Интернационализация — огромная тема, по которой написаны целые книги. В этом разделе вы изучите самые основы, используя тип `CultureInfo` в пространстве имен `System.Globalization`.

Напишем код.

1. Откройте редактор кода и создайте консольное приложение `Internationalization` в рабочей области/решении `Chapter08`.
2. В программе Visual Studio Code выберите `Internationalization` в качестве активного проекта OmniSharp.
3. В начале файла `Program.cs` импортируйте пространство имен для использования типов глобализации:

```
using System.Globalization; // CultureInfo
```

4. Добавьте операторы для получения текущих языковых и региональных настроек глобализации и локализации и вывести часть информации о них, а затем предложите пользователю ввести новые настройки, чтобы показать, насколько это влияет на форматирование значений, таких как дата и валюта:

```
CultureInfo globalization = CultureInfo.CurrentCulture;
CultureInfo localization = CultureInfo.CurrentUICulture;

WriteLine("The current globalization culture is {0}: {1}",
  globalization.Name, globalization.DisplayName);

WriteLine("The current localization culture is {0}: {1}",
  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))
{
  CultureInfo ci = new(newCulture);

  // измените текущие языковые и региональные параметры
  CultureInfo.CurrentCulture = ci;
```

```
  CultureInfo.CurrentUICulture = ci;
}
WriteLine();

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(
  "{0} was born on a {1:dddd}, is {2:N0} minutes old, and earns {3:C}",
  name, date, minutes, earns);
```

При запуске приложения поток автоматически настраивается на использование языковых и региональных параметров в соответствии с операционной системой. Я запускаю свой код в Лондоне, поэтому для потока задан английский язык (Великобритания).

Код предлагает пользователю ввести другой ISO-код. Благодаря этому ваши приложения смогут менять языковые и региональные настройки по умолчанию во время выполнения.

Затем приложение использует стандартные коды формата для вывода дня недели (`dddd`), количества минут с разделителями тысяч (`N0`) и заработной платы с символом валюты. Настройка происходит автоматически на основе языковых и региональных настроек потока.

5. Запустите код, введите значение `en-GB` в качестве ISO-кода, а затем какие-нибудь данные для примера, включая 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
```

Если вы вводите en-US вместо en-GB, то должны указывать дату, используя формат «месяц/день/год».

6. Перезапустите код 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: 340000
Mikkel was born on a onsdag, is 18.656.640 minutes old, and earns
340.000,00 kr.
```

В этом примере только дата и зарплата глобализированы под Данию. Остальная часть текста жестко закодирована как английский язык. В настоящее время в этой книге не описано, как переводить текст с одного языка на другой. Если вы хотите, чтобы я включил данную тему в следующее издание, то, пожалуйста, дайте мне знать.

Подумайте, нужна ли вашему приложению интернационализация, и если да, то запланируйте ее, прежде чем начнете создавать программу! Запишите весь применяемый в пользовательском интерфейсе текст, который необходимо будет локализовать. Подумайте обо всех данных, которые подлежат глобализации (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 получить дополнительную информацию.

Упражнение 8.1. Проверочные вопросы

Пользуясь ресурсами в Интернете, ответьте на следующие вопросы.

1. Какое максимальное количество символов можно сохранить в переменной `string`?
2. В каких случаях и почему нужно использовать тип `SecureString`?
3. В каких ситуациях целесообразно применить класс `StringBuilder`?
4. В каких случаях следует задействовать класс `LinkedList<T>`?
5. Когда класс `SortedDictionary<T>` нужно использовать вместо класса `SortedList<T>`?

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* использовать официальный стандарт для адресов электронной почты при создании регулярного выражения, призванного проверять адрес электронной почты пользователя?

Упражнение 8.2. Регулярные выражения

В рабочей области/решении `Chapter08` создайте консольное приложение `Exercise02`, которое предлагает пользователю ввести сначала регулярное выражение, а затем еще некоторые данные, и сравнивает их на соответствие выражению до тех пор, 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 8.3. Методы расширения

В рабочей области/решении `Chapter08` создайте библиотеку классов `Exercise03`, которая определяет методы, расширяющие числовые типы, такие как `BigInteger` и `int`, с помощью метода `ToWords`, который возвращает строку, описывающую число. Например, `18,000,000` — восемнадцать миллионов, а `18,456,002,032,011,000,007` — восемнадцать квинтиллионов четыреста пятьдесят шесть квадриллионов два триллиона тридцать два миллиарда одиннадцать миллионов семь.

Более подробно об именах для больших чисел можно прочитать на сайте https://en.wikipedia.org/wiki/Names_of_large_numbers.

Упражнение 8.4. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-8---working-with-common-net-types, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали о некоторых вариантах типов, которые позволяют хранить и обрабатывать числа, даты, время и текст, включая регулярные выражения; а также о том, с помощью каких коллекций можно хранить группы элементов. Кроме того, вы рассмотрели индексы, диапазоны и интервалы, научились использовать некоторые сетевые ресурсы, выполнять отражение для кода и атрибутов, обрабатывать изображения с помощью рекомендованной Microsoft сторонней библиотеки, а также интернационализировать код.

В следующей главе вы узнаете, как управлять файлами и потоками, кодировать и декодировать текст и выполнять сериализацию.

9 Работа с файлами, потоками и сериализация

Данная глава посвящена чтению и записи в файлы и потоки, кодированию текста и сериализации.

В этой главе:

- управление файловой системой;
- чтение и запись с помощью потоков;
- кодирование и декодирование текста;
- сериализация графов объектов;
- управление обработкой JSON.

Управление файловой системой

Приложениям часто необходимо выполнять операции ввода и вывода с файлами и каталогами в разных средах. Пространства имен `System` и `System.IO` содержат классы, позволяющие решить эти задачи.

Работа с кросс-платформенными средами и файловыми системами

Рассмотрим, как работать с кросс-платформенными средами, учитывая различия между Windows, macOS или Linux. Пути различаются для Windows, macOS и Linux, поэтому мы начнем с изучения того, как .NET обрабатывает их.

1. Откройте редактор кода и создайте рабочую область/решение `Chapter09`.
2. Создайте проект консольного приложения с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter09`;
   3) файл и папка проекта: `WorkingWithFileSystems`.

3. В файле `Program.cs` добавьте операторы для статического импорта типов `System.Console`, `System.IO.Directory`, `System.Environment` и `System.IO.Path`:

```
using static System.Console;
using static System.IO.Directory;
using static System.IO.Path;
using static System.Environment;
```

4. В файле `Program.cs` создайте статический метод `OutputFileSystemInfo` и добавьте операторы для выполнения следующих действий:
   - вывод символов-разделителей для пути и каталогов;
   - вывод пути к текущему каталогу;
   - вывод нескольких специальных путей для системных и временных файлов и документов.

```
static void OutputFileSystemInfo()
{
  WriteLine("{0,-33} {1}", arg0: "Path.PathSeparator",
    arg1: PathSeparator);
  WriteLine("{0,-33} {1}", arg0: "Path.DirectorySeparatorChar",
    arg1: DirectorySeparatorChar);
  WriteLine("{0,-33} {1}", arg0: "Directory.GetCurrentDirectory()",
    arg1: GetCurrentDirectory());
  WriteLine("{0,-33} {1}", arg0: "Environment.CurrentDirectory",
    arg1: CurrentDirectory);
  WriteLine("{0,-33} {1}", arg0: "Environment.SystemDirectory",
    arg1: SystemDirectory);
  WriteLine("{0,-33} {1}", arg0: "Path.GetTempPath()",
    arg1: GetTempPath());

  WriteLine("GetFolderPath(SpecialFolder");
  WriteLine("{0,-33} {1}", arg0: " .System)",
    arg1: GetFolderPath(SpecialFolder.System));
  WriteLine("{0,-33} {1}", arg0: " .ApplicationData)",
    arg1: GetFolderPath(SpecialFolder.ApplicationData));
  WriteLine("{0,-33} {1}", arg0: " .MyDocuments)",
    arg1: GetFolderPath(SpecialFolder.MyDocuments));
  WriteLine("{0,-33} {1}", arg0: " .Personal)",
    arg1: GetFolderPath(SpecialFolder.Personal));
}
```

Тип Environment содержит множество других полезных членов, которые мы не использовали в этом коде, включая метод GetEnvironmentVariables и свойства OSVersion и ProcessorCount.

5. В файле `Program.cs` над функцией вызовите метод `OutputFileSystemInfo`:

```
OutputFileSystemInfo();
```

6. Запустите код и проанализируйте результат (рис. 9.1).

```
Microsoft Visual Studio Debug Console
Path.PathSeparator                  ;
Path.DirectorySeparatorChar         \
Directory.GetCurrentDirectory()     C:\Code\Chapter09\WorkingWithFileSystems\bin\Debug\net6.0
Environment.CurrentDirectory        C:\Code\Chapter09\WorkingWithFileSystems\bin\Debug\net6.0
Environment.SystemDirectory         C:\WINDOWS\system32
Path.GetTempPath()                  C:\Users\markj\AppData\Local\Temp\
GetFolderPath(SpecialFolder
 .System)                           C:\WINDOWS\system32
 .ApplicationData)                  C:\Users\markj\AppData\Roaming
 .MyDocuments)                      \\Mac\Home\Documents
 .Personal)                         \\Mac\Home\Documents
```

Рис. 9.1. Запуск вашего приложения для отображения информации о файловой системе в Windows

При запуске консольного приложения с помощью команды dotnet run в программе Visual Studio Code CurrentDirectory будет папкой проекта, а не папкой внутри корзины.

В операционной системе Windows в качестве разделителя каталогов используется символ обратной косой черты, или обратный слеш (\). В macOS и Linux — символ косой черты (слеш, /). Не стройте предположений о том, какой символ используется в вашем коде при объединении путей.

Управление дисками

Для управления дисками используйте класс типа `DriveInfo`, который содержит статический метод, возвращающий информацию обо всех дисках, подключенных к вашему компьютеру. Каждый диск имеет тип.

Рассмотрим диски.

1. Создайте метод `WorkWithDrives` и добавьте следующие операторы, чтобы получить все диски и вывести их имя, тип, размер, доступное свободное пространство и формат, но только если диск находится в состоянии готовности:

```
static void WorkWithDrives()
{
  WriteLine("{0,-30} | {1,-10} | {2,-7} | {3,18} | {4,18}",
    "NAME", "TYPE", "FORMAT", "SIZE (BYTES)", "FREE SPACE");

  foreach (DriveInfo drive in DriveInfo.GetDrives())
  {
    if (drive.IsReady)
    {
```

```
        WriteLine(
          "{0,-30} | {1,-10} | {2,-7} | {3,18:N0} | {4,18:N0}",
          drive.Name, drive.DriveType, drive.DriveFormat,
          drive.TotalSize, drive.AvailableFreeSpace);
      }
      else
      {
        WriteLine("{0,-30} | {1,-10}", drive.Name, drive.DriveType);
      }
    }
  }
```

Убедитесь, что диск находится в состоянии готовности, прежде чем использовать свойства, такие как TotalSize. В противном случае возможна ошибка, обычно возникающая при работе со съемными носителями.

2. В файле `Program.cs` закомментируйте предыдущий вызов метода и добавьте вызов метода `WorkWithDrives`, как показано ниже:

```
// OutputFileSystemInfo();
WorkWithDrives();
```

3. Запустите код и проанализируйте результат (рис. 9.2).

```
Microsoft Visual Studio Debug Console
NAME                           | TYPE       | FORMAT  |        SIZE (BYTES) |          FREE SPACE
C:\                            | Fixed      | NTFS    |    136,314,818,560  |     23,093,919,744
D:\                            | CDRom
E:\                            | CDRom
W:\                            | Network    | PrlSF   |    499,963,174,912  |     41,217,667,072
X:\                            | Network    | PrlSF   |    499,963,174,912  |     41,217,667,072
Y:\                            | Network    | PrlSF   |    499,963,174,912  |     41,217,667,072
Z:\                            | Network    | PrlSF   |    499,963,174,912  |     41,217,667,072
```

Рис. 9.2. Отображение информации о диске в Windows

Управление каталогами

Для управления каталогами используйте статические классы `Directory`, `Path` и `Environment`. Эти типы содержат множество членов для работы с файловой системой.

Создавая собственные пути в коде, вы должны быть очень внимательны и не допускать необоснованных предположений о платформе, например о том, какой символ применить в качестве разделителя каталогов.

1. Создайте метод `WorkWithDirectories` и добавьте операторы для выполнения следующих действий:

- определите собственный путь в корневом каталоге пользователя, создав массив строк для имен каталогов, а затем корректно скомбинировав их с помощью метода `Combine` типа `Path`;
- проверьте наличие пользовательского пути к каталогу, применив метод `Exists` класса `Directory`;
- создайте каталог, а затем удалите его, включая файлы и подкаталоги в нем с помощью методов `CreateDirectory` и `Delete` класса `Directory`.

```
static void WorkWithDirectories()
{
  // определяем путь к каталогу для новой папки,
  // начиная с папки пользователя
  string newFolder = Combine(
    GetFolderPath(SpecialFolder.Personal),
    "Code", "Chapter09", "NewFolder");

  WriteLine($"Working with: {newFolder}");

  // проверяем, существует ли она
  WriteLine($"Does it exist? {Exists(newFolder)}");

  // создаем каталог
  WriteLine("Creating it...");
  CreateDirectory(newFolder);
  WriteLine($"Does it exist? {Exists(newFolder)}");
  Write("Confirm the directory exists, and then press ENTER: ");
  ReadLine();

  // удаляем каталог
  WriteLine("Deleting it...");
Delete(newFolder, recursive: true);
WriteLine($"Does it exist? {Exists(newFolder)}");
}
```

2. В файле `Program.cs` закомментируйте предыдущий вызов метода и добавьте вызов метода `WorkWithDirectories`.
3. Запустите код, проанализируйте результат и используйте ваш любимый инструмент управления файлами, чтобы подтвердить создание каталога, прежде чем нажать клавишу Enter для его удаления:

```
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 файла, как мы делали это для типа каталога. Но для следующего примера мы этого не сделаем, поскольку он содержит некоторые из тех же методов, что и тип каталога, а это может привести к конфликту. Тип файла имеет достаточно короткое имя, чтобы это не имело значения в данном случае. Выполните следующие шаги.

1. Создайте метод `WorkWithFiles` и добавьте операторы для выполнения следующих действий:
   1) проверку существования файла;
   2) создания текстового файла;
   3) записи текстовой строки в файл;
   4) закрытия файла для освобождения системных ресурсов и блокировок файла (обычно это делается внутри блока операторов `try-finally`, чтобы обеспечить закрытие файла, даже если при записи в него возникает исключение);
   5) резервного копирования файла;
   6) удаления оригинального файла;
   7) чтения содержимого файла из резервной копии с последующим закрытием.

```
static void WorkWithFiles()
{
  // определяем путь к каталогу для выходных файлов,
  // начиная с папки пользователя
  string dir = Combine(
    GetFolderPath(SpecialFolder.Personal),
    "Code", "Chapter09", "OutputFiles");

  CreateDirectory(dir);

  // определяем пути к файлам
  string textFile = Combine(dir, "Dummy.txt");
  string backupFile = Combine(dir, "Dummy.bak");
  WriteLine($"Working with: {textFile}");

  // проверяем, существует ли файл
  WriteLine($"Does it exist? {File.Exists(textFile)}");

  // создаем новый текстовый файл и записываем в него строку
  StreamWriter textWriter = File.CreateText(textFile);
  textWriter.WriteLine("Hello, C#!");
  textWriter.Close(); // close file and release resources
  WriteLine($"Does it exist? {File.Exists(textFile)}");

  // копируем файл и перезаписываем, если он уже 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();

    // удаляем файл
    File.Delete(textFile);
    WriteLine($"Does it exist? {File.Exists(textFile)}");

    // считываем текстовый файл из резервной копии
    WriteLine($"Reading contents of {backupFile}:");
    StreamReader textReader = File.OpenText(backupFile);
    WriteLine(textReader.ReadToEnd());
    textReader.Close();
  }
```

2. В файле `Program.cs` закомментируйте предыдущий вызов и добавьте вызов метода `WorkWithFiles`.
3. Запустите код и проанализируйте результат:

```
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

В одних случаях вам нужно работать с частями путями, например, извлечь только имя папки, файла или расширение. В других понадобится создавать временные папки и имена файлов. Все это выполняется с помощью статических методов класса `Path`.

1. Добавьте следующие операторы в конец метода `WorkWithFiles`:

```
// управляем путями
WriteLine($"Folder Name: {GetDirectoryName(textFile)}");
WriteLine($"File Name: {GetFileName(textFile)}");
WriteLine("File Name without Extension: {0}",
  GetFileNameWithoutExtension(textFile));
WriteLine($"File Extension: {GetExtension(textFile)}");
WriteLine($"Random File Name: {GetRandomFileName()}");
WriteLine($"Temporary File Name: {GetTempFileName()}");
```

2. Запустите код и проанализируйте результат:

```
Folder Name: /Users/markjprice/Code/Chapter09/OutputFiles
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 нулевого размера 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 его размер или время последнего обращения к нему, вы можете создать экземпляр класса `FileInfo` или `DirectoryInfo`.

Эти классы наследуются от класса `FileSystemInfo`, поэтому оба имеют такие члены, как `LastAccessTime` и `Delete`, а также дополнительные члены, характерные только для них, как показано в табл. 9.1.

Таблица 9.1. Список свойств и методов для файлов и каталогов

Класс	Члены
`FileSystemInfo`	Поля: `FullPath`, `OriginalPath`. Свойства: `Attributes`, `CreationTime`, `CreationTimeUtc`, `Exists`, `Extension`, `FullName`, `LastAccessTime`, `LastAccessTimeUtc`, `LastWriteTime`, `LastWriteTimeUtc`, `Name`. Методы: `Delete`, `GetObjectData`, `Refresh`
`DirectoryInfo`	Свойства: `Parent`, `Root`. Методы: `Create`, `CreateSubdirectory`, `EnumerateDirectories`, `EnumerateFiles`, `EnumerateFileSystemInfos`, `GetAccessControl`, `GetDirectories`, `GetFiles`, `GetFileSystemInfos`, `MoveTo`, `SetAccessControl`
`FileInfo`	Свойства: `Directory`, `DirectoryName`, `IsReadOnly`, `Length`. Методы: `AppendText`, `CopyTo`, `Create`, `CreateText`, `Decrypt`, `Encrypt`, `GetAccessControl`, `MoveTo`, `Open`, `OpenRead`, `OpenText`, `OpenWrite`, `Replace`, `SetAccessControl`

Напишем код, использующий экземпляр `FileInfo` для эффективного выполнения нескольких действий с файлом.

1. Добавьте операторы в конец метода `WorkWithFiles`, чтобы создать экземпляр класса `FileInfo` для файла резервной копии и записать информацию о нем в консоль:

```
FileInfo info = new(backupFile);
WriteLine($"{backupFile}:");
WriteLine($"Contains {info.Length} bytes");
WriteLine($"Last accessed {info.LastAccessTime}");
WriteLine($"Has readonly set to {info.IsReadOnly}");
```

2. Запустите код и проанализируйте результат:

```
/Users/markjprice/Code/Chapter09/OutputFiles/Dummy.bak:
Contains 11 bytes
Last accessed 26/10/2021 09:08:26
Has readonly set to False
```

Количество байтов может различаться в вашей операционной системе, поскольку операционные системы могут использовать разные окончания строки.

Контроль работы с файлами

При работе с файлами часто возникает необходимость контролировать, как именно они открываются. Метод `File.Open` содержит перегрузки для указания дополнительных параметров с помощью значений `enum`.

Ниже приведены используемые типы `enum`:

- `FileMode` — контролирует ваши действия с файлом, например `CreateNew`, `OpenOrCreate` или `Truncate`;
- `FileAccess` — определяет, какой уровень доступа вам нужен, например `ReadWrite`;
- `FileShare` — управляет блокировками файла, чтобы разрешить другим процессам указанный уровень доступа, например `Read`.

Возможно, вы захотите открыть файл и прочитать его, а также разрешить его считывать другим процессам:

```
FileStream file = File.Open(pathToFile,
  FileMode.Open, FileAccess.Read, FileShare.Read);
```

Существует также тип `enum` для атрибутов файла:

- `FileAttributes` — используется для проверки свойства `Attributes` типов, производных от `FileSystemInfo` на наличие таких значений, как `Archive` и `Encrypted`.

Вы можете проверить атрибуты файла или каталога:

```
FileInfo info = new(backupFile);
WriteLine("Is the backup file compressed? {0}",
  info.Attributes.HasFlag(FileAttributes.Compressed));
```

Чтение и запись с помощью потоков

Поток (stream)[1] представляет собой последовательность байтов, которую можно считать или в которую можно записать некие данные. Хотя файлы могут обрабатываться во многом подобно массивам, с произвольным доступом по известной позиции байта в файле, считается полезным обрабатывать файлы как поток, в котором байты могут быть доступны в последовательном порядке.

Кроме того, потоки могут использоваться для обработки входных и выходных данных терминала и сетевых ресурсов, таких как сокеты и порты, которые не обеспечивают произвольный доступ и не могут выполнять поиск (то есть перемещение) позиции. Вы можете написать код для обработки произвольных байтов, не зная и не заботясь о том, откуда они берутся. Ваш код просто считывает или записывает в поток, а другой фрагмент кода определяет, где байты хранятся фактически.

Абстрактные и конкретные потоки

Существует абстрактный класс `Stream`, представляющий собой любой тип потока. Помните, что абстрактный класс `abstract` не может быть реализован с помощью ключевого слова `new`; он может быть только унаследован.

Есть множество конкретных классов, которые наследуются от этого базового, включая `FileStream`, `MemoryStream`, `BufferedStream`, `GZipStream` и `SslStream`, и потому все они работают одинаково. Все потоки реализуют интерфейс `IDisposable`, поэтому имеют метод `Dispose` для освобождения неуправляемых ресурсов.

Некоторые из универсальных членов класса `Stream` приведены в табл. 9.2.

Таблица 9.2. Некоторые из универсальных членов класса Stream

Член	Описание
`CanRead`, `CanWrite`	Эти свойства определяют, поддерживает ли текущий поток возможность чтения и записи
`Length`, `Position`	Эти свойства определяют длину потока в байтах и текущую позицию в нем. Могут вызвать исключение для некоторых типов потоков
`Dispose`	Этот метод закрывает поток и освобождает его ресурсы
`Flush`	Если поток имеет буфер, то этот метод записывает байты из буфера в поток, и буфер очищается
`CanSeek`	Это свойство определяет, можно ли использовать метод `Seek`

[1] В русскоязычной терминологии слово «поток» используется для обозначения англоязычных понятий Stream (дословно: «поток, струя, течение») и Thread (дословно: «нить»).

Член	Описание
`Seek`	Этот метод перемещает текущую позицию на ту, которая указана в его параметре
`Read`, `ReadAsync`	Эти методы считывают определенное количество байтов из потока в байтовый массив и перемещают позицию
`ReadByte`	Этот метод считывает байт из потока и перемещает позицию
`Write`, `WriteAsync`	Эти методы записывают содержимое массива байтов в поток
`WriteByte`	Этот метод записывает байт в поток

Запоминающие потоки

Некоторые *запоминающие потоки*, предоставляющие место хранения байтов представлены в табл. 9.3.

Таблица 9.3. Запоминающие потоки

Пространство имен	Класс	Описание
`System.IO`	`FileStream`	Байты, хранящиеся в файловой системе
`System.IO`	`MemoryStream`	Байты, хранящиеся в памяти в текущем процессе
`System.NET.Sockets`	`NetworkStream`	Байты, хранящиеся в сетевом расположении

Класс `FileStream` был переработан в .NET 6, чтобы обеспечить гораздо более высокую производительность и надежность в Windows.

Функциональные потоки

Некоторые *функциональные потоки* не могут существовать сами по себе, но их можно «подключить» к другим потокам, чтобы расширить их функциональность, как показано в табл. 9.4.

Таблица 9.4. Функциональные потоки

Пространство имен	Класс	Описание
`System.Security.Cryptography`	`CryptoStream`	Шифрует и дешифрует поток
`System.IO.Compression`	`GZipStream`, `DeflateStream`	Сжимают и распаковывают поток
`System.NET.Security`	`AuthenticatedStream`	Передает учетные данные через поток

Вспомогательные потоки

Иногда бывает необходимо работать с потоками на низком уровне. Чаще всего, однако, удается упростить задачу, добавив в цепочку специальные вспомогательные классы. Все вспомогательные типы для потоков реализуют интерфейс `IDisposable`, поэтому имеют метод `Dispose` для освобождения неуправляемых ресурсов.

Некоторые вспомогательные классы для обработки распространенных сценариев представлены в табл. 9.5.

Таблица 9.5. Вспомогательные классы

Пространство имен	Класс	Описание
`System.IO`	`StreamReader`	Считывает данные из потока в простом текстовом формате
	`StreamWriter`	Записывает данные в поток в простом текстовом формате
	`BinaryReader`	Считывает данные из потока в виде типов .NET. Например, метод `ReadDecimal` считывает следующие 16 байт из базового потока как значение `decimal`, а метод `ReadInt32` считывает следующие четыре байта как значение `int`
	`BinaryWriter`	Записывает данные в поток в виде типов .NET. Например, метод `Write` с параметром типа `decimal` записывает 16 байт в базовый поток, а метод `Write` с параметром типа `int` записывает четыре байта
`System.Xml`	`XmlReader`	Считывает данные из потока в XML-формате
	`XmlWriter`	Записывает данные в поток в XML-формате

Запись в текстовые потоки

Напишем код для записи текста в поток.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithStreams` в рабочей области/решении `Chapter09`:
   1) в программе Visual Studio настройте стартовый проект для решения в соответствии с текущим выбором;
   2) в программе Visual Studio Code выберите `WorkingWithStreams` в качестве активного проекта OmniSharp.
2. В проекте `WorkingWithStreams` в файле `Program.cs` импортируйте пространство имен `System.Xml`, статически импортируйте типы `System.Console`, `System.Environment` и `System.IO.Path`.

3. В конце файла `Program.cs` определите статический класс `Viper` и статический массив `Callsigns`, содержащий значения `string`:

```
static class Viper
{

  // определяем массив позывных пилотов
  public static string[] Callsigns = new[]
  {
    "Husker", "Starbuck", "Apollo", "Boomer",
    "Bulldog", "Athena", "Helo", "Racetrack"
  };
}
```

4. Над классом `Viper` определите метод `WorkWithText`, который перечисляет позывные Viper, записывая каждый из них на отдельной строке в одном текстовом файле:

```
static void WorkWithText()
{

  // определяем файл для записи
  string textFile = Combine(CurrentDirectory, "streams.txt");

  // создаем текстовый файл и возвращаем вспомогательную функцию записи
  StreamWriter text = File.CreateText(textFile);

  // перечисляем строки, записывая каждую
  // в поток на отдельной строке
  foreach (string item in Viper.Callsigns)
  {
    text.WriteLine(item);
  }
  text.Close(); // release resources

  // выводим содержимое файла
  WriteLine("{0} contains {1:N0} bytes.",
    arg0: textFile,
    arg1: new FileInfo(textFile).Length);

  WriteLine(File.ReadAllText(textFile));
}
```

5. Под импортом пространства имен вызовите метод `WorkWithText`.
6. Запустите код и проанализируйте результат:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.txt contains
60 bytes.
Husker
Starbuck
```

```
Apollo
Boomer
Bulldog
Athena
Helo
Racetrack
```

7. Откройте созданный файл и убедитесь, что он содержит список позывных.

Запись в XML-потоки

Существуют два способа написания XML-элемента:

- `WriteStartElement` и `WriteEndElement` — используются, когда элемент содержит дочерние элементы;
- `WriteElementString` — используется, когда у элемента нет дочерних элементов.

Теперь сохраним массив позывных пилота Viper в значениях `string` в XML-файле.

1. Создайте метод `WorkWithXml`, перечисляющий позывные пилота, записывая каждый из них как элемент в один файл XML:

```
static void WorkWithXml()
{
  // определяем файл для записи
  string xmlFile = Combine(CurrentDirectory, "streams.xml");

  // создаем файловый поток
  FileStream xmlFileStream = File.Create(xmlFile);

  // оборачиваем файловый поток во вспомогательную функцию XML
  // и создаем автоматический отступ для вложенных элементов
  XmlWriter xml = XmlWriter.Create(xmlFileStream,
    new XmlWriterSettings { Indent = true });

  // пишем XML-декларацию
  xml.WriteStartDocument();

  // записываем корневой элемент
  xml.WriteStartElement("callsigns");;

  // перечисляем строки, записывая каждую в поток
  foreach (string item in Viper.Callsigns)
  {
    xml.WriteElementString("callsign", item);
  }

  // записываем закрывающий корневой элемент
  xml.WriteEndElement();
```

```
    // закрываем вспомогательную функцию и поток
    xml.Close();
    xmlFileStream.Close();

    // выводим все содержимое файла
    WriteLine("{0} contains {1:N0} bytes.",
      arg0: xmlFile,
      arg1: new FileInfo(xmlFile).Length);

    WriteLine(File.ReadAllText(xmlFile));
  }
```

2. В файле `Program.cs` закомментируйте предыдущий вызов метода и добавьте вызов в метод `WorkWithXml`.
3. Запустите код и проанализируйте результат:

```
/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

Открывая файл для чтения или записи, вы используете ресурсы вне .NET. Они называются *неуправляемыми* и должны быть освобождены по окончании работы с ними. Чтобы гарантировать их освобождение, можно вызывать метод `Dispose` внутри блока `finally`.

Чтобы правильно распоряжаться неуправляемыми ресурсами, отредактируем наш предыдущий код, который работает с XML.

1. Измените метод `WorkWithXml`, как показано ниже (выделено жирным шрифтом):

```
static void WorkWithXml()
{
  FileStream? xmlFileStream = null;
  XmlWriter? xml = null;

  try
  {
```

```
    // определяем файл для записи
    string xmlFile = Combine(CurrentDirectory, "streams.xml");

    // создаем файловый поток
    xmlFileStream = File.Create(xmlFile);

    // оборачиваем файловый поток во вспомогательную функцию XML
    // и создаем автоматический отступ для вложенных элементов
    xml = XmlWriter.Create(xmlFileStream,
      new XmlWriterSettings { Indent = true });

    // пишем XML-декларацию
    xml.WriteStartDocument();

    // записываем корневой элемент
    xml.WriteStartElement("callsigns");

    // перечисляем строки, записывая каждую в поток
    foreach (string item in Viper.Callsigns)
    {
      xml.WriteElementString("callsign", item);
    }

    // записываем закрывающий корневой элемент
    xml.WriteEndElement();

    // закрываем вспомогательную функцию и поток
    xml.Close();
    xmlFileStream.Close();

    // выводим все содержимое файла
    WriteLine($"{0} contains {1:N0} bytes.",
      arg0: xmlFile,
      arg1: new FileInfo(xmlFile).Length);

    WriteLine(File.ReadAllText(xmlFile));
  }
  catch (Exception ex)
  {
    // если путь не существует, то будет выдано исключение
    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.");
      }
    }
  }
```

Вы также можете вернуться и изменить другие ранее созданные методы, но пусть это будет дополнительным упражнением для вас.

2. Запустите код и проанализируйте результат:

```
The XML writer's unmanaged resources have been disposed.
The file stream's unmanaged resources have been disposed.
```

Перед вызовом метода Dispose убедитесь, что объект не равен null.

Упрощение освобождения с помощью оператора using

Вы можете упростить код, проверяющий объект на `null`, а затем вызывающий его метод `Dispose` с помощью оператора `using`. Я бы рекомендовал использовать `using`, а не вызывать `Dispose` вручную, если вам не нужен более высокий уровень контроля.

Может немного сбить с толку то, что существует два варианта использования ключевого слова `using`: импорт пространства имен и генерация оператора `finally`, который вызывает метод `Dispose` для объекта, реализующего интерфейс `IDisposable`.

Компилятор изменяет блок оператора `using` на `try-finally` без оператора `catch`. Кроме того, вы можете использовать вложенные операторы `try`, так что при желании все еще можете перехватить какие-либо исключения. Рассмотрим это на примере следующего кода:

```
using (FileStream file2 = File.OpenWrite(
  Path.Combine(path, "file2.txt")))
{
  using (StreamWriter writer2 = new StreamWriter(file2))
  {
    try
    {
      writer2.WriteLine("Welcome, .NET!");
    }
    catch(Exception ex)
    {
```

```
      WriteLine($"{ex.GetType()} says {ex.Message}");
    }
  } // автоматически вызываем Dispose, если объект не равен null
} // автоматически вызываем Dispose, если объект не равен null
```

Вы можете еще больше упростить код, не указывая в явном виде фигурные скобки и отступы для операторов `using`:

```
using FileStream file2 = File.OpenWrite(
  Path.Combine(path, "file2.txt"));

using StreamWriter writer2 = new(file2);

try
{
  writer2.WriteLine("Welcome, .NET!");
}
catch(Exception ex)
{
  WriteLine($"{ex.GetType()} says {ex.Message}");
}
```

Сжатие потоков

Формат XML довольно объемный, поэтому занимает больше памяти в байтах, чем обычный текст. Мы можем сжать XML-данные, воспользовавшись знакомым всем алгоритмом сжатия, известным под названием *GZIP*.

1. В начале файла `Program.cs` импортируйте пространство имен для работы со сжатием:

   ```
   using System.IO.Compression; // BrotliStream, GZipStream, CompressionMode
   ```

2. Добавьте метод `WorkWithCompression`, который использует экземпляры `GZipStream` для создания сжатого файла, содержащего те же элементы XML, что и раньше, а затем распаковывает его при чтении и выводе в консоль:

   ```
   static void WorkWithCompression()
   {
     string fileExt = "gzip";

     // сжимаем XML-вывод
     string filePath = Combine(
       CurrentDirectory, $"streams.{fileExt}");

     FileStream file = File.Create(filePath);

     Stream compressor = new GZipStream(file, CompressionMode.Compress);
   ```

```
    using (compressor)
    {
      using (XmlWriter xml = XmlWriter.Create(compressor))
      {
        xml.WriteStartDocument();
        xml.WriteStartElement("callsigns");

        foreach (string item in Viper.Callsigns)
        {
          xml.WriteElementString("callsign", item);
        }

        // обычный вызов WriteEndElement не требуется,
        // поскольку, освобождая неуправляемые ресурсы,
        // используемые объектом XmlWriter,
        // автоматически завершает любые элементы любой глубины
      }
    } // также закрывает базовый поток

    // вывод всего содержимого сжатого файла
    WriteLine("{0} contains {1:N0} bytes.",
      filePath, new FileInfo(filePath).Length);

    WriteLine($"The compressed contents:");
    WriteLine(File.ReadAllText(filePath));

    // чтение сжатого файла
    WriteLine("Reading the compressed XML file:");
    file = File.Open(filePath, FileMode.Open);

    Stream decompressor = new GZipStream(file,
      CompressionMode.Decompress);

    using (decompressor)
    {
      using (XmlReader reader = XmlReader.Create(decompressor))
      {
        while (reader.Read()) // чтение следующего XML-узла
        {
          // проверяем, находимся ли мы на узле элемента с именем позывной
          if ((reader.NodeType == XmlNodeType.Element)
            && (reader.Name == "callsign"))
          {
            reader.Read(); // переходим к тексту внутри элемента
            WriteLine($"{reader.Value}"); // читаем его значение
          }
        }
      }
    }
  }
```

3. В файле `Program.cs` оставьте вызов метода `WorkWithXml` и добавьте вызов метода `WorkWithCompression`:

```
// WorkWithText();
WorkWithXml();
WorkWithCompression();
```

4. Запустите код и сравните размеры XML-файла и сжатого XML-файла. Как видите, сжатые XML-данные занимают вполовину меньше объема памяти по сравнению с таким же количеством несжатых:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.xml contains
310 bytes.
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.gzip contains
150 bytes.
```

Сжатие с помощью алгоритма Brotli

Выпуская платформу .NET Core 2.1, компания Microsoft представила реализацию алгоритма сжатия Brotli. По производительности он похож на алгоритм, используемый в DEFLATE и GZIP, но результат, как правило, сжат на 20 % больше.

Рассмотрим пример.

1. Измените метод `WorkWithCompression`, в котором необязательный параметр указывает, следует ли использовать алгоритм Brotli, и по умолчанию указывает, что его нужно применить, как показано ниже:

```
static void WorkWithCompression(bool useBrotli = true)
{
  string fileExt = useBrotli ? "brotli" : "gzip";

  // сжатие XML-вывода
  string filePath = Combine(
    CurrentDirectory, $"streams.{fileExt}");

  FileStream file = File.Create(filePath);

  Stream compressor;

  if (useBrotli)
  {
    compressor = new BrotliStream(file, CompressionMode.Compress);
  }
  else
  {
```

```
  compressor = new GZipStream(file, CompressionMode.Compress);
}

using (compressor)
{
  using (XmlWriter xml = XmlWriter.Create(compressor))
  {
    xml.WriteStartDocument();
    xml.WriteStartElement("callsigns");
    foreach (string item in Viper.Callsigns)
    {
      xml.WriteElementString("callsign", item);
    }
  }
} // закрытие основного потока

// выводим все содержимое сжатого файла в консоль
WriteLine("{0} contains {1:N0} bytes.",
  filePath, new FileInfo(filePath).Length);

WriteLine($"The compressed contents:");
WriteLine(File.ReadAllText(filePath));

// чтение сжатого файла
WriteLine("Reading the compressed XML file:");
file = File.Open(filePath, FileMode.Open);

Stream decompressor;
if (useBrotli)
{
  decompressor = new BrotliStream(
    file, CompressionMode.Decompress);
}
else
{
  decompressor = new GZipStream(
    file, CompressionMode.Decompress);
}

using (decompressor)
{
  using (XmlReader reader = XmlReader.Create(decompressor))
  {
    while (reader.Read())
    {
      // проверить, находимся ли мы на элементе с именем callsign
      if ((reader.NodeType == XmlNodeType.Element)
        && (reader.Name == "callsign"))
      {
```

```
            reader.Read(); // переход к тексту внутри элемента
            WriteLine($"{reader.Value}"); // чтение его значения
          }
        }
      }
    }
  }
```

2. В начале файла `Program.cs` дважды вызовите метод `WorkWithCompression`, один раз по умолчанию с помощью алгоритма Brotli и один раз с использованием утилиты GZIP:

```
WorkWithCompression();
WorkWithCompression(useBrotli: false);
```

3. Запустите код и сравните размеры двух сжатых файлов XML. Благодаря использованию алгоритма Бротли сжатые XML-данные занимают почти на 21 % меньше места:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.brotli
contains 118 bytes.
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.gzip contains
150 bytes.
```

Кодирование и декодирование текста

Текстовые символы могут быть представлены разными способами. Например, алфавит можно закодировать 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), представляющих кодовую точку в кодовом пространстве. Большинство кодовых точек представляют один текстовый символ, но некоторые могут иметь и другие значения, такие как форматирование.

Например, таблица ASCII имеет кодовое пространство с 128 кодовыми точками. Платформа .NET использует стандарт *Unicode* для внутреннего кодирования текста. Данный стандарт содержит более миллиона кодовых точек.

Иногда возникает необходимость переместить текст за пределы .NET, чтобы с ним могли работать системы, которые не используют стандарт Unicode или применяют другую его вариацию, поэтому важно научиться преобразовывать кодировки.

В табл. 9.6 перечислены некоторые альтернативные кодировки текста, обычно используемые на компьютерах.

Таблица 9.6. Альтернативные кодировки текста

Кодировка	Описание
ASCII	Кодирует ограниченный диапазон символов, используя семь младших битов байта
UTF-8	Представляет каждую кодовую точку Unicode в виде последовательности от одного до четырех байтов
UTF-7	Спроектирована как более эффективная, чем UTF-8, при работе с семибитовыми каналами, но имеет целый набор проблем с безопасностью и надежностью, так что UTF-8 является более предпочтительной
UTF-16	Представляет каждую кодовую точку Unicode в виде последовательности из одного или двух 16-битных целых чисел
UTF-32	Представляет каждую кодовую точку Unicode в виде 32-битного целого числа и, следовательно, является кодировкой фиксированной длины в отличие от других кодировок Unicode, являющихся кодировками переменной длины
Кодировки ANSI и ISO	Предоставляет поддержку ряда кодовых страниц для поддержки конкретного языка или группы языков

В большинстве случаев в настоящее время кодировка UTF-8 считается хорошим вариантом по умолчанию, и поэтому она и выбрана по умолчанию в .NET. То есть именно она представлена в Encoding.Default.

Кодировка строк в последовательности байтов

Рассмотрим примеры кодировки текста.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithEncodings` в рабочей области/решении `Chapter09`.
2. В программе Visual Studio Code выберите `WorkingWithEncodings` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` импортируйте пространство имен `System.Text` и статически импортируйте класс `Console`.
4. Добавьте операторы для кодирования строки, применяя выбранную пользователем кодировку. Переберите каждый байт, а затем декодируйте его обратно в строку и выведите результат:

```
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");

// выбираем кодировку
Write("Press a number to choose an encoding: ");
ConsoleKey number = ReadKey(intercept: false).Key;
WriteLine();
WriteLine();

Encoding encoder = number switch
{
  ConsoleKey.D1 or ConsoleKey.NumPad1 => Encoding.ASCII,
  ConsoleKey.D2 or ConsoleKey.NumPad2 => Encoding.UTF7,
  ConsoleKey.D3 or ConsoleKey.NumPad3 => Encoding.UTF8,
  ConsoleKey.D4 or ConsoleKey.NumPad4 => Encoding.Unicode,
  ConsoleKey.D5 or ConsoleKey.NumPad5 => Encoding.UTF32,
  _                                   => Encoding.Default
};

// определяем строку для кодировки
string message = "Café cost: £4.39";

// кодируем строку в массив байтов
byte[] encoded = encoder.GetBytes(message);

// проверяем, сколько байтов требуется для кодировки
WriteLine("{0} uses {1:N0} bytes.",
  encoder.GetType().Name, encoded.Length);
WriteLine();

// перечисляем каждый байт
WriteLine($"BYTE HEX CHAR");
foreach (byte b in encoded)
{
  WriteLine($"{b,4} {b.ToString("X"),4} {(char)b,5}");
}

// декодируем массив байтов обратно в строку и отображаем его
string decoded = encoder.GetString(encoded);
WriteLine(decoded);
```

5. Запустите код и обратите внимание на предупреждение о том, что следует избегать использования `Encoding.UTF7`, поскольку это небезопасно. Конечно, если вам нужно сгенерировать текст, используя данную кодировку для совместимости с другой системой, то она должна оставаться опцией в .NET.
6. Нажмите клавишу `1`, чтобы выбрать кодировку ASCII. Обратите внимание: при выводе байтов символ фунта стерлинга (`£`) и акцентированное е (`é`) не могут быть представлены в кодировке ASCII, поэтому вместо них используется знак вопроса (`?`):

```
BYTE  HEX  CHAR
  67   43    C
  97   61    a
 102   66    f
  63   3F    ?
  32   20
 111   6F    o
 115   73    s
 116   74    t
  58   3A    :
  32   20
  63   3F    ?
  52   34    4
  46   2E    .
  51   33    3
  57   39    9
Caf? cost: ?4.39
```

7. Перезапустите код и нажмите клавишу `3`, чтобы выбрать кодировку UTF-8. Обратите внимание: для хранения данных она требует два дополнительных байта для двух символов, которым нужно по два байта (всего 18 байт вместо 16), но может кодировать и декодировать символы `é` и `£`:

```
UTF8EncodingSealed uses 18 bytes.

BYTE  HEX  CHAR
  67   43    C
  97   61    a
 102   66    f
 195   C3    Ã
 169   A9    ©
  32   20
 111   6F    o
 115   73    s
 116   74    t
  58   3A    :
  32   20
 194   C2    Â
 163   A3    £
  52   34    4
  46   2E    .
  51   33    3
  57   39    9
Café cost: £4.39
```

8. Перезапустите код и нажмите клавишу `4`, чтобы выбрать кодировку Unicode (UTF-16). Обратите внимание: она требует два байта для каждого символа, то есть всего 32 байта, и может кодировать и декодировать символы `é` и `£`. Данная кодировка используется внутри .NET для хранения значений `char` и `string`.

Кодирование и декодирование текста в файлах

При использовании вспомогательных классов потоков, таких как `StreamReader` и `StreamWriter`, вы можете указать предпочтительную кодировку. При записи во вспомогательный поток строки будут кодироваться автоматически, а при чтении из него — автоматически декодироваться.

Чтобы указать кодировку, передайте ее в качестве второго параметра конструктору вспомогательного типа:

```
StreamReader reader = new(stream, Encoding.UTF8);
StreamWriter writer = new(stream, Encoding.UTF8);
```

Зачастую вы не сможете выбирать кодировку, так как 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 — процесс преобразования активного объекта в последовательность байтов с помощью выбранного формата. Обратный процесс называется *десериализацией*. Они позволяют сохранять текущее состояние активного объекта, чтобы его можно было воссоздать в будущем. Например, сохранение текущего состояния игры, чтобы завтра можно было продолжить игру с того же места. Сериализованные объекты обычно хранятся в файле или базе данных.

Существуют десятки доступных для выбора форматов, но два наиболее распространенных — это *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 устаревшими системами. 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, а мобильные приложения часто выполняют вызовы с ограниченной пропускной способностью, поэтому количество байтов важно.

Платформа .NET содержит несколько классов, которые умеют сериализовать в форматы XML и JSON (и из них). Мы начнем изучение с классов `XmlSerializer` и `JsonSerializer`.

XML-сериализация

Начнем с изучения XML, вероятно, наиболее используемого в мире формата сериализации (на данный момент). В качестве универсального примера мы определим пользовательский класс для хранения информации о человеке, а затем создадим граф объектов, задействуя список экземпляров `Person` с вложением.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithSerialization` в рабочей области/решении `Chapter09`.
2. В программе Visual Studio Code выберите `WorkingWithSerialization` в качестве активного проекта OmniSharp.
3. Добавьте класс `Person` со свойством `Salary`, которое объявлено защищенным (`protected`), то есть доступно только внутри самого класса и производных классов. Для указания зарплаты класс содержит конструктор с единственным параметром, устанавливающим начальный оклад:

```
namespace Packt.Shared;

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; }
}
```

4. В файле `Program.cs` импортируйте пространства имен для работы с сериализацией XML и статически импортируйте классы `Console`, `Environment` и `Path`:

```
using System.Xml.Serialization; // класс XmlSerializer
using Packt.Shared; // класс Person

using static System.Console;
using static System.Environment;
using static System.IO.Path;
```

5. Добавьте операторы для создания объектного графа экземпляров `Person`:

```
/создаем объектный граф
List<Person> people = new()
{
```

```
    new(30000M)
    {
       FirstName = "Alice",
       LastName = "Smith",
       DateOfBirth = new(1974, 3, 14)
    },
    new(40000M)
    {
       FirstName = "Bob",
       LastName = "Jones",
       DateOfBirth = new(1969, 11, 23)
    },
    new(20000M)
    {
       FirstName = "Charlie",
       LastName = "Cox",
       DateOfBirth = new(1984, 5, 4),
       Children = new()
      {
          new(0M)
          {
             FirstName = "Sally",
             LastName = "Cox",
             DateOfBirth = new(2000, 7, 12)
          }
       }
    }
};

// создаем объект, который будет форматировать список лиц как XML
XmlSerializer xs = new(people.GetType());

// создаем файл для записи
string path = Combine(CurrentDirectory, "people.xml");

using (FileStream stream = File.Create(path))
{
   // сериализуем объектный граф в поток
   xs.Serialize(stream, people);
}

WriteLine("Written {0:N0} bytes of XML to {1}",
  arg0: new FileInfo(path).Length,
  arg1: path);
WriteLine();

// отображаем сериализованный граф объектов
WriteLine(File.ReadAllText(path));
```

6. Запустите код и проанализируйте результат. Обратите внимание на вызываемое исключение:

```
Unhandled Exception: System.InvalidOperationException: Packt.Shared.Person
cannot be serialized because it does not have a parameterless constructor.
```

7. В файле `Person` добавьте оператор для определения конструктора без параметров:

```
public Person() { }
```

Обратите внимание: конструктору ничего не нужно делать, однако он должен присутствовать, чтобы класс `XmlSerializer` мог его вызвать для создания экземпляров `Person` в процессе десериализации.

8. Перезапустите код и проанализируйте результат. Обратите внимание, что граф объектов сериализуется в виде элементов XML, таких как `<FirstName>Bob</FirstName>`, и что свойство `Salary` не включено в результат, поскольку не является общедоступным:

```
Written 752 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>Cox</LastName>
    <DateOfBirth>1984-05-04T00:00:00</DateOfBirth>
    <Children>
      <Person>
        <FirstName>Sally</FirstName>
        <LastName>Cox</LastName>
        <DateOfBirth>2000-07-12T00:00:00</DateOfBirth>
      </Person>
    </Children>
  </Person>
</ArrayOfPerson>
```

Генерация компактного XML

Формат XML можно использовать более эффективно, если для некоторых полей вместо элементов применить атрибуты.

1. В классе `Person` импортируйте пространство имен `System.Xml.Serialization`, чтобы вы могли дополнить некоторые свойства атрибутом `[XmlAttribute]`.
2. Дополните свойства имени, фамилии и даты рождения атрибутом `[XmlAttribute]` и задайте короткое имя для каждого свойства, как показано ниже:

```
[XmlAttribute("fname")]
public string FirstName { get; set; }

[XmlAttribute("lname")]
public string LastName { get; set; }

[XmlAttribute("dob")]
public DateTime DateOfBirth { get; set; }
```

3. Запустите код и обратите внимание: размер файла был уменьшен с 752 до 462 байт, что экономит пространство более чем на треть, за счет вывода значений свойств в виде атрибутов XML:

```
Written 462 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 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="Cox" dob="1984-05-04T00:00:00">
    <Children>
      <Person fname="Sally" lname="Cox" dob="2000-07-12T00:00:00" />
    </Children>
  </Person>
</ArrayOfPerson>
```

XML-десериализация

Рассмотрим десериализацию файлов XML обратно в активные объекты в памяти.

1. Добавьте следующие операторы, чтобы открыть файл XML, а затем десериализовать его:

```
using (FileStream xmlLoad = File.Open(path, FileMode.Open))
{
```

```
    // десериализуем и приводим объектный граф в список лиц
    List<Person>? loadedPeople =
      xs.Deserialize(xmlLoad) as List<Person>;

    if (loadedPeople is not null)
    {
      foreach (Person p in loadedPeople)
      {
        WriteLine("{0} has {1} children.",
          p.LastName, p.Children?.Count ?? 0);
      }
    }
  }
```

2. Запустите код и обратите внимание, что информация о людях успешно загружается из XML-файла и затем перечисляется:

```
Smith has 0 children.
Jones has 0 children.
Cox has 1 children.
```

Есть множество других атрибутов, с помощью которых можно управлять создаваемым XML.

Если вы не используете никаких аннотаций, класс `XmlSerializer` при десериализации выполняет сопоставление без учета регистра, используя имя свойства.

При использовании класса XmlSerializer помните, что учитываются только открытые (public) поля и свойства, а тип должен содержать конструктор без параметров. Вывод можно настроить с помощью атрибутов.

JSON-сериализация

Одна из наиболее популярных библиотек .NET для работы с форматом сериализации JSON — `Newtonsoft.Json`, известная как *Json.NET*.

Рассмотрим пример ее работы.

1. В проекте `WorkingWithSerialization` добавьте ссылку на пакет для последней версии `Newtonsoft.Json`, как показано ниже:

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json"
    Version="13.0.1" />
</ItemGroup>
```

2. Соберите проект `WorkingWithSerialization` для восстановления пакетов.
3. В файле `Program.cs` добавьте операторы для создания текстового файла, а затем сериализуйте информацию о людях в файл в формате JSON:

```
// создаем файл для записи
string jsonPath = Combine(CurrentDirectory, "people.json");

using (StreamWriter jsonStream = File.CreateText(jsonPath))
{
  // создаем объект, который будет форматироваться как JSON
  Newtonsoft.Json.JsonSerializer jss = new();

  // сериализуем объектный граф в строку
  jss.Serialize(jsonStream, people);
}
WriteLine();
WriteLine("Written {0:N0} bytes of JSON to: {1}",
  arg0: new FileInfo(jsonPath).Length,
  arg1: jsonPath);

// отображаем сериализованный граф объектов
WriteLine(File.ReadAllText(jsonPath));
```

4. Запустите код и обратите внимание, что формат JSON занимает более чем вполовину меньше байтов памяти по сравнению с форматом XML, содержащим элементы. Его размер даже меньше, чем XML с атрибутами:

```
Written 366 bytes of JSON to: /Users/markjprice/Code/Chapter09/
WorkingWithSerialization/people.json [{"FirstName":"Alice","LastName":"Smi
th","DateOfBirth":"1974-03-
14T00:00:00","Children":null},{"FirstName":"Bob","LastName":"Jones","Date
OfBirth":"1969-11-23T00:00:00","Children":null},{"FirstName":"Charlie","L
astName":"Cox","DateOfBirth":"1984-05-04T00:00:00","Children":[{"FirstNam
e":"Sally","LastName":"Cox","DateOfBirth":"2000-07-12T00:00:00","Children
":null}]}]
```

Высокопроизводительная обработка JSON

Платформа .NET Core 3.0 представляет новое пространство имен для работы с JSON — `System.Text.Json`. Оно оптимизировано в целях повышения производительности благодаря использованию таких API, как `Span<T>`.

Кроме того, более старые библиотеки, такие как Json.NET, реализуются через чтение UTF-16. Было бы более эффективным читать и записывать документы JSON с помощью кодировки UTF-8, поскольку большинство сетевых протоколов, включая

HTTP, используют именно ее, и вы можете избежать перекодирования UTF-8 в значения `string` Unicode в Json.NET и обратно.

Выпуск новых API позволил Microsoft добиться улучшения в 1,3–5,0 раза в зависимости от сценария.

Первый автор Json.NET Джеймс Ньютон-Кинг присоединился к Microsoft и работал над разработкой новых типов JSON. Как он сказал в комментарии при обсуждении новых API для JSON, «Json.NET никуда не уходит» (рис. 9.3).

Рис. 9.3. Комментарий первого автора Json.NET

Рассмотрим, как новые JSON-API используются для десериализации файла JSON.

1. В проекте `WorkingWithSerialization` в файле `Program.cs` импортируйте новый класс JSON для выполнения сериализации с помощью псевдонима, чтобы избежать конфликта имен с ранее использованным Json.NET:

```
using NewJson = System.Text.Json.JsonSerializer;
```

2. Добавьте следующие операторы, чтобы открыть файл JSON, десериализовать его и вывести имена и количество детей указанных в файле людей:

```
using (FileStream jsonLoad = File.Open(jsonPath, FileMode.Open))
{
  // десериализуем объектный граф в список лиц
  List<Person>? loadedPeople =
    await NewJson.DeserializeAsync(utf8Json: jsonLoad,
      returnType: typeof(List<Person>)) as List<Person>;

  if (loadedPeople is not null)
  {
    foreach (Person p in loadedPeople)
    {
      WriteLine("{0} has {1} children.",
        p.LastName, p.Children?.Count ?? 0);
    }
  }
}
```

3. Запустите код и проанализируйте результат:

```
Smith has 0 children.
Jones has 0 children.
Cox has 1 children.
```

Выбирайте Json.NET, чтобы увеличить производительность труда разработчиков и иметь большой набор функций, а System.Text.Json — для того, чтобы повысить производительность кода.

Управление обработкой JSON

Существует множество вариантов управления обработкой JSON:

- включение и исключение полей;
- настройка политики регистра;
- чувствительность к регистру;
- выбор между компактными и упрощенными пробельными символами.

Рассмотрим некоторые из этих вариантов.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithJson` в рабочей области/решении `Chapter09`.
2. В программе Visual Studio Code выберите `WorkingWithJson` в качестве активного проекта OmniSharp.
3. В проекте `WorkingWithJson` в файле `Program.cs` удалите существующий код, импортируйте два основных пространства имен для работы с JSON, а затем статически импортируйте типы `System.Console`, `System.Environment` и `System.IO.Path`:

```
using System.Text.Json; // класс JsonSerializer
using System.Text.Json.Serialization; // атрибут [JsonInclude]

using static System.Console;
using static System.Environment;
using static System.IO.Path;
```

4. В конце файла `Program.cs` определите класс `Book`:

```
public class Book
{
  // конструктор для установки свойства, не допускающего null
  public Book(string title)
  {
    Title = title;
  }
```

```
  // свойства

  public string Title { get; set; }
  public string? Author { get; set; }

  // поля

  [JsonInclude] // включаем это поле
  public DateOnly PublishDate;

  [JsonInclude] // включаем это поле
  public DateTimeOffset Created;

  public ushort Pages;
}
```

5. Над классом `Book` добавьте операторы для создания экземпляра класса `Book` и сериализации его в JSON:

```
Book csharp10 = new(title:
  "C# 10 and .NET 6 - Modern Cross-platform Development")
{
  Author = "Mark J Price",
  PublishDate = new(year: 2021, month: 11, day: 9),
  Pages = 823,
  Created = DateTimeOffset.UtcNow,
};

JsonSerializerOptions options = new()
{
  IncludeFields = true, // включает в себя все поля
  PropertyNameCaseInsensitive = true,
  WriteIndented = true,
  PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
};

string filePath = Combine(CurrentDirectory, "book.json");

using (Stream fileStream = File.Create(filePath))
{
  JsonSerializer.Serialize<Book>(
  utf8Json: fileStream, value: csharp10, options);
}

WriteLine("Written {0:N0} bytes of JSON to {1}",
  arg0: new FileInfo(filePath).Length,
  arg1: filePath);

WriteLine();


// отображаем сериализованный граф объектов
WriteLine(File.ReadAllText(filePath));
```

6. Запустите код и проанализируйте результат:

```
Written 315 bytes of JSON to C:\Code\Chapter09\WorkingWithJson\bin\Debug\
net6.0\book.json

{
  "title": "C# 10 and .NET 6 - Modern Cross-platform Development",
  "author": "Mark J Price",
  "publishDate": {
    "year": 2021,
    "month": 11,
    "day": 9,
    "dayOfWeek": 2,
    "dayOfYear": 313,
    "dayNumber": 738102
  },
  "created": "2021-08-20T08:07:02.3191648+00:00",
  "pages": 823
}
```

Обратите внимание на следующие моменты:

- размер файла JSON составляет 315 байт;
- в именах членов используется верблюжий регистр, например `publishDate`. Он лучше всего подходит для последующей обработки в браузере с помощью JavaScript;
- все поля включены в соответствии с установленными опциями, в том числе `pages`;
- JSON-код оптимизирован в целях улучшения читабельности;
- значения `DateTimeOffset` сохранены в едином стандартном строковом формате;
- значения `DateOnly` сохранены в виде объекта с подсвойствами для таких элементов даты, как `year` и `month`.

7. В файле `Program.cs` при установке параметров `JsonSerializerOptions` закомментируйте настройки политики регистра, отступов и включения полей.

8. Запустите код и проанализируйте результат:

```
Written 230 bytes of JSON to C:\Code\Chapter09\WorkingWithJson\bin\Debug\
net6.0\book.json

{"Title":"C# 10 and .NET 6 - Modern Cross-platform
Development","Author":"Mark J Price","PublishDate":{"Year":2021,"Month
":11,"Day":9,"DayOfWeek":2,"DayOfYear":313,"DayNumber":738102},"Creat
ed":"2021-08-20T08:12:31.6852484+00:00"}
```

Обратите внимание на следующие моменты:

- файл JSON имеет размер 230 байт, а значит, уменьшился более чем на 25 %;
- в именах участников используется обычный регистр, например `PublishDate`;
- поле `Pages` отсутствует. Остальные поля включены благодаря атрибуту `[JsonInclude]` в полях `PublishDate` и `Created`;
- JSON компактен и содержит минимальное количество пробельных символов, что позволяет сократить затраты трафика при передаче и хранении данных.

Новые методы расширения JSON для работы с ответами HTTP

В .NET 5 Microsoft добавила уточнения к типам в пространстве имен `System.Text.Json`, например методы расширения для `HttpResponse`, которые вы изучите в главе 16.

Переход с Newtonsoft на новый JSON

Если у вас есть код, использующий библиотеку Newtonsoft Json.NET, и вы хотите перейти в новое пространство имен `System.Text.Json`, то Microsoft предоставляет для этого специальную документацию, с которой вы можете ознакомиться по следующей ссылке: https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-text-json-migrate-from-newtonsoft-how-to.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 9.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Чем использование класса `File` отличается от использования класса `FileInfo`?
2. Чем различаются методы потока `ReadByte` и `Read`?
3. В каких случаях используются классы `StringReader`, `TextReader` и `StreamReader`?
4. Для чего предназначен тип `DeflateStream`?

5. Сколько байтов на символ затрачивается при использовании кодировки UTF-8?
6. Что такое граф объектов?
7. Какой формат сериализации лучше всего минимизирует затраты памяти?
8. Какой формат сериализации позволяет обеспечить кросс-платформенную совместимость?
9. Почему не рекомендуется использовать значение `string` наподобие `"\Code\Chapter01"` для представления пути и что необходимо выполнить вместо этого?
10. Где можно найти информацию о пакетах NuGet и их зависимостях?

Упражнение 9.2. XML-сериализация

В рабочей области/решении `Chapter09` создайте консольное приложение `Exercise02`, которое генерирует список фигур, использует сериализацию для сохранения его в файловой системе с помощью XML, а затем десериализует его обратно:

```
// создаем список фигур для сериализации
List<Shape> listOfShapes = new()
{
  new Circle { Colour = "Red", Radius = 2.5 },
  new Rectangle { Colour = "Blue", Height = 20.0, Width = 10.0 },
  new Circle { Colour = "Green", Radius = 8.0 },
  new Circle { Colour = "Purple", Radius = 12.3 },
  new Rectangle { Colour = "Blue", Height = 45.0, Width = 18.0 }
};
```

Фигуры должны содержать свойство только для чтения `Area`, чтобы при десериализации вы могли выводить список фигур, включая их площадь:

```
List<Shape> loadedShapesXml =
  serializerXml.Deserialize(fileXml) as List<Shape>;

foreach (Shape item in loadedShapesXml)
{
  WriteLine("{0} is {1} and has an area of {2:N2}",
    item.GetType().Name, item.Colour, item.Area);
}
```

Вот как должен выглядеть ваш вывод при запуске консольного приложения:

```
Loading shapes from XML:
Circle is Red and has an area of 19.63
Rectangle is Blue and has an area of 200.00
Circle is Green and has an area of 201.06
Circle is Purple and has an area of 475.29
Rectangle is Blue and has an area of 810.00
```

Упражнение 9.3. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-9---working-with-files-streams-and-serialization, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали, как читать из текстовых и XML-файлов и записывать в них, сжимать и распаковывать файлы, кодировать/декодировать текст и сериализовать объекты в форматы JSON и XML (и десериализовать их обратно).

В следующей главе вы узнаете, как работать с базами данных с помощью Entity Framework Core.

10 Работа с данными с помощью Entity Framework Core

Текущая глава посвящена чтению и записи в такие базы данных, как Microsoft SQL Server, SQLite и Azure Cosmos DB с помощью технологии объектно-реляционного отображения данных *Entity Framework Core (EF Core)*.

В этой главе:

- современные базы данных;
- настройка EF Core;
- определение моделей EF Core;
- запрос данных из модели EF Core;
- загрузка шаблонов с помощью EF Core;
- управление данными с помощью EF Core;
- обработка транзакций;
- модели Code First EF Core.

Современные базы данных

Два наиболее распространенных места для хранения данных — это *реляционная система управления базами данных* (РСУБД; Relational Database Management System, RDBMS), такая как Microsoft SQL Server, PostgreSQL, MySQL и SQLite, или база данных *NoSQL*, например Microsoft Azure Cosmos DB, Redis, MongoDB и Apache Cassandra.

Устаревшая Entity Framework

Технология Entity Framework (EF) была впервые выпущена как часть .NET Framework 3.5 с Service Pack 1 в конце 2008 года. С тех пор эта технология значительно развилась, поскольку компания Microsoft пристально наблюдала за тем, как про-

граммисты используют этот инструмент *объектно-реляционного отображения* (object-relational mapping, ORM) при работе над своими продуктами.

Суть ORM в определении сопоставления между столбцами в таблицах и свойствами в классах. Затем разработчик может взаимодействовать с объектами различных типов привычным для него способом вместо того, чтобы изучать способы сохранения значений в реляционной таблице или другой структуре, предоставляемой хранилищем данных NoSQL.

Entity Framework 6 (EF6) — это версия EF, включенная в последнюю версию .NET Framework. Она полная, стабильная и поддерживает способ определения модели EDMX (XML-файл), а также сложные модели наследования и некоторые другие расширенные функции.

Версия EF 6.3 и более поздние версии были извлечены из .NET Framework в виде отдельного пакета, поэтому данные версии могут поддерживаться в .NET Core 3.0 и более поздних версиях. Это позволяет переносить существующие проекты, такие как веб-приложения и сервисы, и запускать их на разных платформах. Тем не менее EF6 следует считать устаревшей технологией, поскольку она имеет некоторые ограничения при работе с разными платформами и в нее не будут добавляться новые функции.

Устаревшие версии Entity Framework 6.3 и старше

Чтобы использовать устаревшую версию Entity Framework в проекте .NET Core 3.0 или более поздней версии, необходимо добавить ссылку на пакет в файл проекта:

```
<PackageReference Include="EntityFramework" Version="6.4.4" />
```

Используйте устаревшую версию EF6 только при необходимости, например при переносе приложения WPF, которое его использует. Данная книга посвящена современной кросс-платформенной разработке, поэтому в оставшейся части главы я буду рассматривать только современную Entity Framework Core. Вам не нужно будет ссылаться на устаревший пакет EF6, как показано выше в проектах для этой главы.

Entity Framework Core

По-настоящему кросс-платформенная версия EF Core отличается от устаревшей Entity Framework. Хотя EF Core имеет похожее имя, вы должны знать, чем оно отличается от EF6. Последняя версия EF Core — 6.0, соответствующая .NET 6.0.

EF Core 5 и более поздни версии поддерживают только .NET 5 и более поздние версии. EF Core 3.0 и более поздние версии работают только на платформах,

поддерживающих .NET Standard 2.1, то есть .NET Core 3.0 и более поздние версии. Она не будет работать на платформах .NET Standard 2.0, таких как .NET Framework 4.8.

Помимо традиционных РСУБД, EF Core поддерживает современные облачные, нереляционные хранилища данных без схемы, такие как Microsoft Azure Cosmos DB и MongoDB, иногда со сторонними поставщиками.

EF Core содержит так много улучшений, что я не могу описать их все в этой главе. Поэтому я сосредоточусь на основных принципах, которые должны знать все разработчики .NET, и на некоторых интересных новых функциях.

Существует два подхода к работе с EF Core:

1) *сначала база данных* — база данных уже существует, поэтому вы создаете модель, соответствующую ее структуре и функциональным возможностям;
2) *сначала код* — базы данных не существует, поэтому вы создаете модель, а затем используете EF Core для создания базы данных, соответствующей ее структуре и функциональным возможностям.

Мы начнем с использования EF Core с существующей базой данных.

Создание консольного приложения для работы с EF Core

Сначала мы создадим проект консольного приложения для этой главы.

1. Откройте редактор кода и создайте решение/рабочую область `Chapter10`.
2. Создайте проект консольного приложения с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter10`;
   3) файл и папка проекта: `WorkingWithEFCore`.

Использование образца реляционной базы данных

Чтобы научиться управлять реляционной базой данных с помощью .NET, было бы полезно иметь под рукой образец средней сложности, но с достойным количеством заготовленных записей. Microsoft предлагает несколько образцов БД, большинство

из которых слишком сложны для наших целей, поэтому мы воспользуемся базой данных Northwind, впервые созданной в начале 1990-х годов.

Ниже приведена схема базы данных Northwind, которую вы можете использовать для справки, когда мы будем в дальнейшем писать код и запросы (рис. 10.1).

Рис. 10.1. Таблицы и связи базы данных Northwind

Далее в этой и последующих главах мы напишем код для работы с таблицами `Categories` и `Products`. Но сначала обратите внимание на следующие моменты:

- каждая категория имеет уникальный идентификатор, имя, описание и изображение;
- каждый товар имеет уникальный идентификатор, имя, цену за единицу, единицы на складе и другие поля;
- каждый товар связан с категорией, сохраняя уникальный идентификатор категории;
- связи между `Categories` и `Products` представляют собой вариант «один ко многим», то есть каждая категория может иметь ноль или более товаров.

Использование Microsoft SQL Server в Windows

Microsoft выпустила различные версии своего популярного и функционального продукта SQL Server для Windows, Linux и контейнеров Docker. Мы будем использовать бесплатную версию SQL Server Developer Edition, которая может работать автономно. Вы также можете использовать версию Express или бесплатную версию SQL Server LocalDB, которая может быть установлена вместе с Visual Studio для Windows.

Если у вас нет компьютера под управлением Windows или вы хотите использовать кросс-платформенную систему баз данных, то можете перейти к подразделу «Использование SQLite» далее в этой главе.

Загрузка и установка SQL Server

Вы можете загрузить дистрибутив SQL Server разных версий по следующей ссылке: https://www.microsoft.com/en-us/sql-server/sql-server-downloads.

1. Загрузите версию Developer (для разработчиков).
2. Запустите программу установки.
3. Выберите тип установки Custom (пользовательская).
4. Выберите папку для установочных файлов, а затем нажмите кнопку Install (Установить).
5. Дождитесь, пока загрузятся 1,5 Гбайт установочных файлов.
6. В SQL Server Installation Center (Центр установки SQL Server) нажмите кнопку Installation (Установка), а затем выберите пункт New SQL Server stand-alone installation (Новая автономная установка SQL Server) или Add features to an existing installation (Добавление функций в существующую установку).
7. Выберите вариант Developer в качестве бесплатной версии, а затем нажмите кнопку Next (Далее).
8. Примите условия лицензии, а затем нажмите кнопку Next (Далее).
9. Ознакомьтесь с правилами установки, устраните возникшие проблемы и нажмите кнопку Next (Далее).
10. В разделе Feature Selection (Выбор характеристик) выберите вариант Database Engine Services (Сервисы ядра базы данных), а затем нажмите кнопку Next (Далее).
11. В разделе Instance Configuration (Конфигурация экземпляра) выберите вариант Default instance (Экземпляр по умолчанию), а затем нажмите кнопку Next (Далее). Если у вас уже настроен экземпляр по умолчанию, то можно создать именованный экземпляр, например `cs10dotnet6`.
12. В разделе Server Configuration (Конфигурация сервера) обратите внимание, что компонент SQL Server Database Engine настроен на запуск автоматически. Уста-

новите автоматический запуск компонента SQL Server Browser, а затем нажмите кнопку Next (Далее).

13. В разделе Database Engine Configuration (Конфигурация ядра базы данных) на вкладке Server Configuration (Конфигурация сервера) установите режим Authentication Mode (Режим аутентификации) на Mixed (Смешанный), смените пароль учетной записи sa на надежный пароль, нажмите кнопку Add Current User (Добавить текущего пользователя), а затем кнопку Next (Далее).
14. В разделе Ready to Install (Готов к установке) просмотрите действия, которые будут выполнены, а затем нажмите кнопку Install (Установить).
15. В разделе Complete (Выполнено) отметьте успешно выполненные действия, а затем нажмите кнопку Close (Закрыть).
16. В SQL Server Installation Center (Центр установки SQL Server) в разделе Installation (Установка) нажмите кнопку Install SQL Server Management Tools (Установить инструменты управления SQL Server).
17. С помощью браузера скачайте последнюю версию SSMS.
18. Запустите программу установки и нажмите кнопку Install (Установить).
19. После завершения работы программы установки нажмите кнопку Restart (Перезапустить) или Close (Закрыть).

Создание образца базы данных Northwind для SQL Server

Теперь мы можем запустить сценарий базы данных для создания образца базы данных Northwind.

1. Если вы ранее не скачивали или не клонировали хранилище данных GitHub для данной книги, то сделайте это сейчас, используя ссылку https://github.com/markjprice/cs10dotnet6/.
2. Скопируйте сценарий для создания базы данных Northwind для SQL Server из следующего пути в вашем локальном хранилище данных Git: `/sql-scripts/Northwind4SQLServer.sql` в папку WorkingWithEFCore.
3. Запустите SQL Server Management Studio.
4. В диалоговом окне Connect to Server (Подключиться к серверу) в поле Server name (Имя сервера) введите символ `.` (точка), означающий имя локального компьютера, а затем нажмите кнопку Connect (Подключиться).

Если вам нужно создать именованный экземпляр, например, cs10dotnet6, то введите .\cs10dotnet6.

5. Выберите команду меню File ▸ Open ▸ File (Файл ▸ Открыть ▸ Файл).
6. Выберите файл `Northwind4SQLServer.sql` и нажмите кнопку Open (Открыть).

7. На панели инструментов нажмите кнопку Execute (Выполнить) и обратите внимание на сообщение Command(s) completed successfully (Команда (-ы) выполнена успешно).
8. В окне Object Explorer (Проводник объектов) разверните базу данных Northwind, а затем разверните пункт Tables (Таблицы).
9. Щелкните правой кнопкой мыши на пункте Products, выберите команду меню Select Top 1000 Rows (Выбрать 1000 лучших строк) и обратите внимание на возвращенные результаты (рис. 10.2).

Рис. 10.2. Таблица Products в среде SQL Server Management Studio

10. На панели инструментов окна Object Explorer (Проводник объектов) нажмите кнопку Disconnect (Отключиться).
11. Завершите работу SQL Server Management Studio.

Управление образцом базы данных Northwind с помощью Server Explorer

Нам не обязательно использовать SQL Server Management Studio для выполнения сценария базы данных. Мы также можем использовать инструменты Visual Studio, включая SQL Server Object Explorer и Server Explorer.

1. В программе Visual Studio выберите команду меню View ▸ Server Explorer (Просмотр ▸ Проводник сервера).
2. В окне Server Explorer (Проводник сервера) щелкните правой кнопкой мыши на пункте Data Connections (Подключение данных) и выберите команду меню Add Connection (Добавить подключение).

3. Если откроется диалоговое окно Choose Data Source (Выбор источника данных), показанное на рис. 10.3, то выберите вариант Microsoft SQL Server и нажмите кнопку Continue (Продолжить).

Рис. 10.3. Выбор SQL Server в качестве источника данных

4. В диалоговом окне Add Connection (Добавить подключение) введите имя сервера `.` (точка), укажите имя базы данных как `Northwind`, а затем нажмите кнопку OK.
5. В окне Server Explorer (Проводник сервера) разверните подключение к данным и его таблицы. Вы должны увидеть 13 таблиц, включая Categories и Products.
6. Щелкните правой кнопкой мыши на таблице Products, выберите команду меню Show Table Data (Показать данные таблицы) и обратите внимание на 77 строк продуктов.
7. Чтобы просмотреть подробную информацию о столбцах и типах таблицы Products, щелкните правой кнопкой мыши на таблице Products и выберите команду меню Open Table Definition (Открыть определение таблицы) или дважды щелкните кнопкой мыши на таблице в окне Server Explorer (Проводник сервера).

Использование SQLite

SQLite — небольшая кросс-платформенная самодостаточная РСУБД, доступная в публичном домене. Это самая распространенная база данных для мобильных платформ, таких как iOS (iPhone и iPad) и Android. Даже если вы работаете в Windows и в предыдущем подразделе настроили SQL Server, вы можете настроить и SQLite. Код, который мы напишем, будет работать в обоих случаях, и может быть интересно увидеть тонкие различия.

Настройка SQLite в macOS

SQLite включена в каталог /usr/bin/ операционной системы macOS в виде консольного приложения `sqlite3`.

Настройка SQLite в Windows

Для работы в Windows нам необходимо добавить папку с SQLite в системный путь, чтобы программа была обнаружена при вводе команд в командной строке или терминале.

1. Запустите браузер и перейдите по ссылке www.sqlite.org/download.html.
2. Прокрутите страницу вниз до раздела Precompiled Binaries for Windows (Предварительно скомпилированные двоичные файлы для Windows).
3. Выберите файл `sqlite-tools-win32-x86-3360000.zip`. Обратите внимание, что после публикации этой книги в названии файла может быть указан более высокий номер версии.
4. Извлеките содержимое ZIP-файла в папку C:\Sqlite\. Убедитесь, что извлеченный файл `sqlite3.exe` находится непосредственно в папке C:\SQLite, иначе исполняемый файл не будет найден, когда вы попытаетесь его использовать.
5. Перейдите к окну Windows Settings (Настройки Windows).
6. Найдите `environment` и выберите пункт Edit the system environment variables (Изменить системные переменные среды). В неанглоязычных версиях Windows, пожалуйста, найдите эквивалентный параметр на вашем языке.
7. Нажмите кнопку Environment Variables (Переменные среды).
8. В разделе System variables (Системные переменные) выберите в списке пункт Path (Путь) и нажмите кнопку Edit (Редактировать).
9. Выберите пункт New (Новый), введите `C:\Sqlite` и нажмите клавишу Enter.
10. Далее трижды нажмите кнопку OK.
11. Закройте окно Windows Settings (Настройки Windows).

Настройка SQLite для других операционных систем

SQLite можно скачать и установить для других операционных систем по следующей ссылке: https://www.sqlite.org/download.html.

Создание образца базы данных Northwind для SQLite

Теперь мы можем создать образец базы данных Northwind для SQLite с помощью сценария SQL.

1. Если вы ранее не клонировали репозиторий GitHub для данной книги, то сделайте это сейчас, используя следующую ссылку: https://github.com/markjprice/cs10dotnet6/.

2. Скопируйте сценарий для создания базы данных Northwind для SQLite из пути в локальном репозитории Git `/sql-scripts/Northwind4SQLite.sql` в папку WorkingWithEFCore.
3. Запустите командную строку в папке WorkingWithEFCore:
   1) в Windows запустите File Explorer (Проводник), щелкните правой кнопкой мыши на папке WorkingWithEFCore и выберите пункт New Command Prompt at Folder (Открыть командную строку здесь) или Open in Windows Terminal (Открыть в терминале Windows);
   2) в операционной системе macOS запустите приложение Finder, щелкните правой кнопкой мыши на папке WorkingWithEFCore и выберите пункт New Terminal at Folder (Новый терминал в папке).
4. Введите команду для выполнения сценария SQL с помощью SQLite и создания базы данных `Northwind.db`, как показано ниже:

```
sqlite3 Northwind.db -init Northwind4SQLite.sql
```

5. Вам придется немного подождать, поскольку на создание всей структуры базы данных у этой команды может уйти некоторое время. В конечном итоге вы увидите командную строку SQLite:

```
-- Loading resources from Northwind4SQLite.sql
SQLite version 3.36.0 2021-08-24 15:20:15
Enter ".help" for usage hints.
sqlite>
```

6. Для выхода из командного режима SQLite нажмите сочетание клавиш Ctrl+C в Windows или Ctrl+D в macOS.
7. Оставьте окно терминала или оболочки командной строки открытым, так как вскоре снова будете им пользоваться.

Управление образцом базы данных Northwind в SQLiteStudio

Для простого управления базами данных SQLite вы можете использовать кросс-платформенный графический менеджер баз данных *SQLiteStudio*.

1. Перейдите по ссылке http://sqlitestudio.pl и скачайте и установите приложение в удобное для вас место.
2. Запустите SQLiteStudio.
3. В меню Database (База данных) выберите пункт Add a database (Добавить базу данных).
4. В диалоговом окне Database (База данных) в разделе File (Файл) щелкните на значке желтой папки, чтобы перейти к существующему файлу базы

данных на локальном компьютере. В папке WorkingWithEFCore выберите файл `Northwind.db` и нажмите кнопку OK.

5. Щелкните правой кнопкой мыши на базе данных Northwind и выберите Connect to the database (Подключиться к базе данных). Вы увидите десять таблиц, созданные с помощью сценария. (Сценарий для SQLite проще, чем для SQL Server; в нем не создается столько таблиц и других объектов базы данных.)
6. Щелкните правой кнопкой мыши на таблице Products и выберите команду Edit the table (Редактировать таблицу).
7. В окне редактора таблицы обратите внимание на структуру таблицы `Products`, включая имена столбцов, типы данных, ключи и ограничения (рис. 10.4).

Рис. 10.4. Редактор таблиц в SQLiteStudio, отображающий структуру таблицы Products

8. В окне редактора таблицы перейдите на вкладку Data (Данные). Вы увидите 77 строк с наименованием товаров (рис. 10.5).

Рис. 10.5. Вкладка Data (Данные) со строками в таблице Products

9. В окне Database (База данных) щелкните правой кнопкой мыши на пункте Northwind и выберите команду меню Disconnect from the database (Отключиться от базы данных).
10. Выйдите из SQLiteStudio.

Настройка EF Core

Прежде чем углубляться в практические аспекты управления базами данных с помощью Entity Framework Core, немного поговорим о выборе *поставщиков данных Entity Framework Core*.

Выбор поставщика данных Entity Framework Core

Чтобы управлять данными в конкретной БД, нам нужны классы, которые знают, как эффективно взаимодействовать с базой.

Поставщики данных Entity Framework Core — это наборы классов, оптимизированные для работы с конкретным хранилищем данных. Существует даже поставщик для хранения данных в памяти текущего процесса, который может быть полезен для высокопроизводительного модульного тестирования, поскольку ему не требуется общаться с внешней системой.

Поставщики распространяются в пакетах NuGet, как показано в табл. 10.1.

Таблица 10.1. Хранилища данных и пакеты NuGet, позволяющие управлять ими

Хранилище данных	NuGet-пакет
Microsoft SQL Server 2012 или более поздних версий	Microsoft.EntityFrameworkCore.SqlServer
SQLite 3.7 или более поздних версий	Microsoft.EntityFrameworkCore.SQLite
MySQL	MySQL.EntityFrameworkCore
In-memory	Microsoft.EntityFrameworkCore.InMemory
Azure Cosmos DB SQL API	Microsoft.EntityFrameworkCore.Cosmos
Oracle DB 11.2	Oracle.EntityFrameworkCore

Вы можете установить в одном проекте любое количество поставщиков баз данных EF Core. Каждый пакет включает в себя общие типы, а также специфические типы, зависящие от конкретного поставщика.

Подключение к базе данных

Чтобы подключиться к базе данных SQLite, необходимо знать только имя файла базы данных, заданное с помощью параметра `Filename`.

Для подключения к базе данных SQL Server нам необходимо знать следующие сведения:

- имя сервера (и экземпляра, если он есть);
- имя базы данных;
- политики безопасности, например имя пользователя и пароль, а также следует ли передавать учетные данные текущего пользователя автоматически.

Мы указываем эту информацию в *строке подключения*.

Для обеспечения обратной совместимости в строке подключения SQL Server можно использовать несколько возможных ключевых слов для различных параметров, как показано в следующем списке:

- ключевые слова `Data Source`, или `server`, или `addr` представляют собой имя сервера (и необязательный экземпляр). Вы можете использовать точку `.` для обозначения локального сервера;
- ключевые слова `Initial Catalog` или `database` являются именем базы данных;
- для ключевых слов `Integrated Security` или `trusted_connection` устанавливаются значения `true` или `SSPI` для передачи текущих учетных данных пользователя потока;
- ключевому слову `MultipleActiveResultSets` присваивается значение `true`, чтобы разрешить использовать одно соединение для одновременной работы с несколькими таблицами в целях повышения эффективности. Оно используется для ленивой загрузки строк из связанных таблиц.

Как описано в приведенном выше списке, когда вы пишете код для подключения к базе данных SQL Server, вам необходимо знать имя сервера. Оно зависит от редакции и версии SQL Server, к которому вы будете подключаться, как показано в табл. 10.2.

Таблица 10.2. Версии SQL Server и имена сервера

Версия SQL Server	**Имя сервера\Имя экземпляра**
LocalDB 2012	`(localdb)\v11.0`
LocalDB 2016 или более поздние версии	`(localdb)\mssqllocaldb`
Express	`.\sqlexpress`
Full/Developer (экземпляр по умолчанию)	`.`
Full/Developer (именованный экземпляр)	`.\cs10dotnet6`

Используйте символ точки (.) в качестве сокращения для имени локального компьютера. Помните, что имена серверов SQL Server состоят из двух частей: имени компьютера и имени экземпляра SQL Server. Имена экземпляров задаются во время пользовательской установки.

Определение контекста базы данных Northwind

Класс `Northwind` будет использоваться для представления базы данных. Чтобы использовать EF Core, класс должен наследоваться от `DbContext`. Этот класс понимает, как взаимодействовать с базами данных и динамически генерировать SQL-запросы для запроса и обработки данных.

Ваш класс, производный от `DbContext`, должен иметь переопределенный метод `OnConfiguring`, который будет настраивать подключение к базе данных.

Чтобы вам было проще освоить SQLite и SQL Server, мы создадим проект, который поддерживает оба варианта, с полем `string` для управления данными во время выполнения.

1. В проекте `WorkingWithEFCore` добавьте ссылки на пакет для поставщика данных EF Core для SQL Server и SQLite, как показано ниже:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Sqlite"
    Version="6.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="6.0.0" />
</ItemGroup>
```

2. Соберите проект для восстановления пакетов.
3. Добавьте файл класса `ProjectConstants.cs`.
4. В файле `ProjectConstants.cs` определите класс с константой `public string` для хранения имени поставщика базы данных, который вы хотите использовать:

```
namespace Packt.Shared;

public class ProjectConstants
{
  public const string DatabaseProvider = "SQLite"; // или "SQLServer"
}
```

5. В файле `Program.cs` импортируйте пространство имен `Packt.Shared` и выведите поставщика базы данных:

```
WriteLine($"Using {ProjectConstants.DatabaseProvider} database
provider.");
```

6. Добавьте файл класса `Northwind.cs`.
7. В файле `Northwind.cs` определите класс `Northwind`, импортируйте основное пространство имен EF Core, наследуйте класс от `DbContext`, а в методе `OnConfiguring` отметьте поле `provider`, чтобы использовать либо SQLite, либо SQL Server:

```
using Microsoft.EntityFrameworkCore; // DbContext, DbContextOptionsBuilder

using static System.Console;

namespace Packt.Shared;

// позволяет управлять подключением к базе данных
public class Northwind : DbContext
{
  protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
  {
    if (ProjectConstants.DatabaseProvider == "SQLite")
    {
      string path = Path.Combine(
        Environment.CurrentDirectory, "Northwind.db");

      WriteLine($"Using {path} database file.");
      optionsBuilder.UseSqlite($"Filename={path}");
    }
    else
    {
      string connection = "Data Source=.;" +
        "Initial Catalog=Northwind;" +
        "Integrated Security=true;" +
        "MultipleActiveResultSets=true;";

      optionsBuilder.UseSqlServer(connection);
    }
  }
}
```

Если вы используете Visual Studio для Windows, то скомпилированное приложение запускается в папке WorkingWithEFCore\bin\Debug\net6.0, поэтому оно не найдет файл базы данных.

8. На панели Solution Explorer (Проводник решений) щелкните правой кнопкой мыши на файле `Northwind.db` и выберите команду меню Properties (Свойства).
9. В окне Properties (Свойства) присвойте параметру Copy to Output Directory (Копировать в выходной каталог) значение Copy always (Копировать всегда).
10. Откройте файл `WorkingWithEFCore.csproj` и обратите внимание на новые элементы:

```
<ItemGroup>
  <None Update="Northwind.db">
```

```
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

Если вы используете программу Visual Studio Code, то скомпилированное приложение будет выполняться в папке WorkingWithEFCore, поэтому файл базы данных будет найден без его копирования.

11. Запустите консольное приложение и обратите внимание на выходные данные, показывающие, какой поставщик базы данных вы выбрали для использования.

Определение моделей EF Core

В EF Core для создания модели во время выполнения используется сочетание *соглашений*, *атрибутов аннотаций* и операторов *Fluent API* таким образом, что любое действие, произведенное над классами, впоследствии может быть автоматически транслировано в действие, совершаемое над действительной базой данных. Сущностный класс представляет собой структуру таблицы, а экземпляр класса — строку в этой таблице.

В первую очередь мы рассмотрим три метода определения модели с примерами кода, а затем создадим несколько классов, реализующих эти подходы.

Соглашения Entity Framework Core для определения модели

В программном коде, который мы станем писать, будут использоваться следующие соглашения:

- предполагается, что имя таблицы совпадает с именем свойства `DbSet<T>` в классе `DbContext`, например `Products`;
- предполагается, что имена столбцов совпадают с именами свойств в классе сущностной модели, например `ProductID`;
- предполагается, что тип `string` в .NET соответствует типу `nvarchar` в базе данных;
- предполагается, что тип .NET `int` — это тип `int` в базе данных;
- предполагается, что первичным ключом является свойство `Id` или `ID`, либо, если класс сущностной модели именуется `Product`, свойство может быть названо `ProductId` или `ProductID`. Если данное свойство имеет любой целочисленный тип или тип `Guid`, то также предполагается, что это свойство — `IDENTITY` (то есть его значение присваивается автоматически при добавлении строки в базу данных).

Существует большое количество других соглашений (более того — вы можете определить собственные), однако это выходит за рамки данной книги. Более подробную информацию можно узнать на сайте https://docs.microsoft.com/ru-ru/ef/core/modeling/.

Использование атрибутов аннотаций Entity Framework Core для определения модели

Зачастую соглашений недостаточно для полного сопоставления всех классов с объектами базы данных. Простой способ сделать вашу модель умнее — применить атрибуты аннотаций.

Некоторые наиболее часто встречающиеся атрибуты показаны в табл. 10.3.

Таблица 10.3. Атрибуты аннотаций

Атрибут	**Описание**
`[Required]`	Гарантирует, что значение не является `null`
`[StringLength(50)]`	Обеспечивает длину значения до 50 символов
`[RegularExpression(expression)]`	Обеспечивает соответствие значения заданному регулярному выражению
`[Column(TypeName = "money", Name = "UnitPrice")]`	Указывает тип и имя столбца, используемые в таблице

Например, в базе данных максимальная длина наименования товара равна `40`, и данное значение не может быть пустым (`null`), как показано в следующем коде *языка определения данных* (Data Definition Language, DDL), который определяет, как создать таблицу `Products` со столбцами, типами данных, ключами и другими ограничениями:

```
CREATE TABLE Products (
    ProductId       INTEGER       PRIMARY KEY,
    ProductName     NVARCHAR (40) NOT NULL,
    SupplierId      "INT",
    CategoryId      "INT",
    QuantityPerUnit NVARCHAR (20),
    UnitPrice       "MONEY"       CONSTRAINT DF_Products_UnitPrice DEFAULT (0),
    UnitsInStock    "SMALLINT"    CONSTRAINT DF_Products_UnitsInStock DEFAULT (0),
    UnitsOnOrder    "SMALLINT"    CONSTRAINT DF_Products_UnitsOnOrder DEFAULT (0),
    ReorderLevel    "SMALLINT"    CONSTRAINT DF_Products_ReorderLevel DEFAULT (0),
    Discontinued    "BIT" NOT NULL
                                  CONSTRAINT DF_Products_Discontinued DEFAULT (0),
    CONSTRAINT FK_Products_Categories FOREIGN KEY (
       CategoryId
    )
```

```
    REFERENCES Categories (CategoryId),
    CONSTRAINT FK_Products_Suppliers FOREIGN KEY (
        SupplierId
    )
    REFERENCES Suppliers (SupplierId),
    CONSTRAINT CK_Products_UnitPrice CHECK (UnitPrice >= 0),
    CONSTRAINT CK_ReorderLevel CHECK (ReorderLevel >= 0),
    CONSTRAINT CK_UnitsInStock CHECK (UnitsInStock >= 0),
    CONSTRAINT CK_UnitsOnOrder CHECK (UnitsOnOrder >= 0)
    );
```

Чтобы указать эти ограничения, в классе `Product` мы можем применить следующие атрибуты:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

Атрибуты могут использоваться при отсутствии очевидного соответствия между типами .NET и типами базы данных.

Например, в базе данных тип столбца `UnitPrice` таблицы `Product` — `money`. В .NET нет типа `money`, поэтому вместо него мы должны воспользоваться типом `decimal`:

```
[Column(TypeName = "money")]
public decimal? UnitPrice { get; set; }
```

Еще один пример — таблица `Categories`, как показано в следующем коде DDL:

```
CREATE TABLE Categories (
    CategoryId   INTEGER       PRIMARY KEY,
    CategoryName NVARCHAR (15) NOT NULL,
    Description  "NTEXT",
    Picture      "IMAGE"
);
```

Столбец `Description` может превышать максимум в 8000 символов, которые могут быть сохранены в переменной `nvarchar`, так что вместо этого его необходимо сопоставить с `ntext`:

```
[Column(TypeName = "ntext")]
public string Description { get; set; }
```

Использование Entity Framework Core Fluent API для определения модели

И последний способ, с помощью которого можно определить модель, — использовать *Fluent API*. Он может применяться вместо атрибутов или вдобавок к ним. Например, для определения свойства `ProductName` вместо того, чтобы дополнять

свойство двумя атрибутами, можно написать эквивалентный оператор Fluent API в методе `OnModelBuilding` класса контекста базы данных:

```
modelBuilder.Entity<Product>()
  .Property(product => product.ProductName)
  .IsRequired()
  .HasMaxLength(40);
```

Так можно упростить класс сущностной модели.

Заполнение таблиц базы данных с помощью Fluent API

Еще одним преимуществом Fluent API является предоставление исходных данных для заполнения базы данных. EF Core автоматически определяет, какие операции вставки, обновления или удаления должны быть выполнены.

Например, если бы мы хотели убедиться в существовании хотя бы одной строки в таблице `Product` в новой базе данных, то вызвали бы метод `HasData`:

```
modelBuilder.Entity<Product>()
  .HasData(new Product
  {
    ProductId = 1,
    ProductName = "Chai",
    UnitPrice = 8.99M
  });
```

Наша модель будет сопоставлена с существующей БД, которая уже заполнена данными, поэтому нам не придется использовать данную технику в нашем коде.

Создание модели Entity Framework Core для таблиц Northwind

Теперь, когда вы изучили способы определения модели EF Core, создадим модель для представления двух таблиц в базе данных Northwind.

Два сущностных класса будут ссылаться друг на друга, поэтому во избежание ошибок компилятора мы сначала создадим классы без каких-либо членов.

1. В проект `WorkingWithEFCore` добавьте два файла классов `Category.cs` и `Product.cs`.
2. В файле `Category.cs` определите класс `Category`:

   ```
   namespace Packt.Shared;

   public class Category
   {
   }
   ```

3. В файле `Product.cs` определите класс `Product`:

```
namespace Packt.Shared;

public class Product
{
}
```

Определение сущностных классов Category и Product

Класс `Category`, также известный как сущностная модель, будет использоваться для представления строки в таблице `Categories`. Эта таблица состоит из четырех столбцов, как показано в следующем DDL:

```
CREATE TABLE Categories (
    CategoryId   INTEGER       PRIMARY KEY,
    CategoryName NVARCHAR (15) NOT NULL,
    Description  "NTEXT",
    Picture      "IMAGE"
);
```

Мы будем использовать соглашения, чтобы определить:

- три из четырех свойств (не станем отображать столбец `Picture`);
- первичный ключ;
- связь «один ко многим» с таблицей `Products`.

Чтобы сопоставить столбец `Description` с правильным типом базы данных, нам необходимо дополнить соответствующее свойство типа `string` атрибутом `Column`.

Далее в этой главе с помощью Fluent API мы будем определять, что `CategoryName` не может иметь значение `null` и может содержать не более 15 символов.

Приступим.

1. Отредактируйте класс сущностной модели `Category`:

```
using System.ComponentModel.DataAnnotations.Schema; // [Column]

namespace Packt.Shared;

public class Category
{
  // эти свойства сопоставляются со столбцами в базе данных
  public int CategoryId { get; set; }
  public string? CategoryName { get; set; }

  [Column(TypeName = "ntext")]
  public string? Description { get; set; }
```

```
  // определяем свойство навигации для связанных строк
  public virtual ICollection<Product> Products { get; set; }

  public Category()
  {
    // чтобы разработчики могли добавлять продукты в категорию,
    // мы должны инициализировать свойство навигации в пустую коллекцию
    Products = new HashSet<Product>();
  }
}
```

Класс `Product` будет использоваться для представления строки в таблице `Products`, состоящей из десяти столбцов.

Вам не нужно включать все столбцы из таблицы в качестве свойств класса. Мы сопоставим только следующие шесть свойств: `ProductID`, `ProductName`, `UnitPrice`, `UnitsInStock`, `Discontinued` и `CategoryID`.

Столбцы, которые не сопоставлены со свойствами, не могут быть прочитаны или установлены с помощью экземпляров класса. При использовании класса для создания нового объекта новая строка в таблице будет иметь значение `null` или какое-либо другое по умолчанию для значений несопоставленных столбцов в этой строке. Вы должны убедиться, что отсутствующие столбцы являются необязательными или имеют значения по умолчанию, установленные базой данных, иначе во время выполнения будет выдано исключение. В этом сценарии строки уже содержат данные, и я решил, что у меня нет необходимости считывать их в этом приложении.

Мы можем переименовать столбец, определив свойство с другим именем, например `Cost`, а затем дополнить свойство атрибутом `[Column]` и указать имя его столбца, скажем `UnitPrice`.

Последнее свойство `CategoryID` связано со свойством `Category`, с помощью которого мы сопоставим каждый товар с его родительской категорией.

2. Отредактируйте класс `Product`:

```
using System.ComponentModel.DataAnnotations; // [Required], [StringLength]
using System.ComponentModel.DataAnnotations.Schema; // [Column]

namespace Packt.Shared;

public class Product
{
  public int ProductId { get; set; } // первичный ключ

  [Required]
  [StringLength(40)]
  public string ProductName { get; set; } = null!;

  [Column("UnitPrice", TypeName = "money")]
  public decimal? Cost { get; set; } // имя свойства != имя столбца
```

```
    [Column("UnitsInStock")]
    public short? Stock { get; set; }

    public bool Discontinued { get; set; }

    // эти два параметра определяют отношение внешнего ключа
    // к таблице Categories
    public int CategoryId { get; set; }
    public virtual Category Category { get; set; } = null!;
  }
```

Два свойства, `Category.Products` и `Product.Category`, связывающие две сущности, отмечены как виртуальные. Это позволяет EF Core наследовать и переопределять указанные свойства в целях предоставления дополнительной функциональности, например ленивой загрузки.

Добавление таблиц в контекстный класс базы данных Northwind

Внутри вашего класса, производного от `DbContext`, вы должны определить хотя бы одно свойство типа `DbSet<T>`. Эти свойства представляют таблицы. Чтобы сообщить EF Core, какие столбцы содержит каждая таблица, тип `DbSet<T>` использует дженерики для указания класса, представляющего строку таблицы. Этот класс сущностной модели имеет свойства, которые представляют столбцы таблицы.

Производный от `DbContext` класс может дополнительно иметь переопределенный метод `OnModelCreating`. В нем вы можете добавить операторы Fluent API как альтернативу дополнения атрибутами ваших сущностных классов.

Напишем код.

Отредактируйте класс `Northwind`, добавив операторы для определения двух свойств для двух таблиц и метод `OnModelCreating`, как показано ниже (выделено жирным шрифтом):

```
public class Northwind : DbContext
{
  // эти свойства сопоставляются с таблицами в базе данных
  public DbSet<Category>? Categories { get; set; }
  public DbSet<Product>? Products { get; set; }

  protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
  {
    ...
  }
```

```
  protected override void OnModelCreating(
    ModelBuilder modelBuilder)
  {
    // пример использования Fluent API вместо атрибутов,
    // чтобы ограничить длину имени категории 15 символами
    modelBuilder.Entity<Category>()
      .Property(category => category.CategoryName)
      .IsRequired() // NOT NULL
      .HasMaxLength(15);

    if (ProjectConstants.DatabaseProvider == "SQLite")
    {
      // добавлен патч для десятичной поддержки в SQLite
      modelBuilder.Entity<Product>()
        .Property(product => product.Cost)
        .HasConversion<double>();
    }
  }
}
```

В EF Core 3.0 и более поздних версиях тип decimal не поддерживается поставщиком базы данных SQLite для сортировки и других операций. Мы можем исправить это, указав модели, что значения decimal могут быть преобразованы в значения double при использовании поставщика базы данных SQLite. На самом деле в данном случае не происходит никакого преобразования во время выполнения.

Теперь, когда вы увидели несколько примеров определения сущностной модели вручную, рассмотрим технологию, которая может сделать часть работы за вас.

Настройка инструмента dotnet-ef

В .NET есть инструмент командной строки `dotnet`. Его можно расширить за счет возможностей, полезных для работы с EF Core. Он может выполнять задачи во время разработки, такие как создание и применение миграций из старой модели в более новую и генерация кода для модели из существующей базы данных.

Инструмент `dotnet-ef` не устанавливается автоматически. Вы должны установить этот пакет как *глобальный* или *локальный инструмент*. Если вы уже установили более старую версию инструмента, то вам следует удалить все существующие версии.

1. В командной строке или терминале проверьте, установлен ли `dotnet-ef` в качестве глобального инструмента:

```
dotnet tool list --global
```

2. Проверьте в результатах, была ли установлена более старая версия инструмента, например для .NET Core 3.1:

```
Package Id      Version     Commands
-------------------------------------
dotnet-ef       3.1.0       dotnet-ef
```

3. Если старая версия уже установлена, то удалите инструмент:

```
dotnet tool uninstall --global dotnet-ef
```

4. Установите последнюю версию:

```
dotnet tool install --global dotnet-ef --version 6.0.0
```

5. При необходимости следуйте инструкциям для конкретной операционной системы, чтобы добавить каталог dotnet tools в переменную среды PATH, как описано в выходных данных установки инструмента `dotnet-ef`.

Создание шаблонов с использованием существующей базы данных

Моделирование — это процесс использования инструмента для создания классов, представляющих модель существующей базы данных с использованием обратного проектирования. Хороший инструмент для создания шаблонов позволяет вам расширять автоматически сгенерированные классы, а затем регенерировать эти классы без потери этих расширенных классов.

Если вы знаете, что никогда не будете регенерировать классы с помощью данного инструмента, то можете изменять код для автоматически сгенерированных классов сколько угодно раз.

Рассмотрим пример и ответим на вопрос, генерирует ли инструмент ту же модель, что и мы вручную.

1. Добавьте пакет `Microsoft.EntityFrameworkCore.Design` в проект `WorkingWithEFCore`.
2. В командной строке или в терминале в папке WorkingWithEFCore создайте модель для таблиц `Categories` и `Products` в новой папке AutoGenModels, как показано в команде ниже:

```
dotnet ef dbcontext scaffold "Filename=Northwind.db" Microsoft.
EntityFrameworkCore.Sqlite --table Categories --table Products --outputdir
AutoGenModels --namespace WorkingWithEFCore.AutoGen --data-annotations
--context Northwind
```

Обратите внимание на следующие моменты:

- команда для выполнения — `dbcontext scaffold`;
- строка подключения — `"Filename=Northwind.db"`;
- поставщик базы данных — `Microsoft.EntityFrameworkCore.Sqlite`;
- таблицы для создания шаблонов — `--table Categories --table Products`;
- выходная папка — --output-dir AutoGenModels;
- пространством имен является `--namespace WorkingWithEFCore.AutoGen`;
- для использования аннотаций данных, а также Fluent API применяется `--data-annotations`;
- для переименования контекста из `[database_name]Context` используется `--context Northwind`.

Для SQL Server измените поставщик базы данных и строку подключения:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial
Catalog=Northwind;Integrated Security=true;" Microsoft.
EntityFrameworkCore.SqlServer --table Categories
--table Products --output-dir AutoGenModels --namespace
WorkingWithEFCore.AutoGen --data-annotations --context
Northwind
```

3. Обратите внимание на полученное сообщение и предупреждение сборки:

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string,
you should move it out of source code. You can avoid scaffolding the
connection string by using the Name= syntax to read it from configuration
- see https://go.microsoft.com/fwlink/?linkid=2131148. For more
guidance on storing connection strings, see http://go.microsoft.com/
fwlink/?LinkId=723263.
Skipping foreign key with identity '0' on table 'Products' since principal
table 'Suppliers' was not found in the model. This usually happens when
the principal table was not included in the selection set.
```

4. Откройте папку AutoGenModels и обратите внимание на автоматически созданные три файла классов: `Category.cs`, `Northwind.cs` и `Product.cs`.
5. Откройте файл `Category.cs` и обратите внимание на то, чем он отличается от файла, созданного вручную:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
```

```
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace WorkingWithEFCore.AutoGen
{
  [Index(nameof(CategoryName), Name = "CategoryName")]
  public partial class Category
  {
    public Category()
    {
      Products = new HashSet<Product>();
    }

    [Key]
    public long CategoryId { get; set; }

    [Required]
    [Column(TypeName = "nvarchar (15)")] // SQLite
    [StringLength(15)] // SQL Server
    public string CategoryName { get; set; }

    [Column(TypeName = "ntext")]
    public string? Description { get; set; }

    [Column(TypeName = "image")]
    public byte[]? Picture { get; set; }

    [InverseProperty(nameof(Product.Category))]
    public virtual ICollection<Product> Products { get; set; }
  }
}
```

Обратите внимание на следующие моменты:

- данный инструмент дополняет сущностный класс атрибутом `[Index]`, который был представлен в EF Core 5.0. Он указывает свойства, которые должны содержать индекс. В более ранних версиях для определения индексов поддерживался только Fluent API. Поскольку мы работаем с существующей базой данных, в этом нет необходимости. Но если мы захотим воссоздать новую пустую базу данных из нашего кода, то эта информация будет необходима;
- таблица в базе данных носит имя `Categories`, но инструмент `dotnet-ef` использует стороннюю библиотеку *Humanizer* для автоматического преобразования имени класса в единственное число в категорию `Category`, что является более естественным именем при создании отдельной сущности;
- сущностный класс объявляется с помощью ключевого слова `partial`, чтобы вы могли создать соответствующий частичный класс для добавления дополнительного кода. Это позволяет повторно запустить инструмент и регенерировать сущностный класс, не теряя экстракод;

- свойство `CategoryId` дополнено атрибутом `[Key]`, чтобы указать, что это первичный ключ для данной сущности. Тип данных для этого свойства — `int` для SQL Server и `long` для SQLite;
- свойство `Products` использует атрибут `[InverseProperty]` для определения отношения внешнего ключа к свойству `Category` в сущностном классе `Product`.

6. Откройте файл `Product.cs` и обратите внимание на то, чем он отличается от файла, созданного вручную.
7. Откройте файл `Northwind.cs` и обратите внимание на то, чем он отличается от файла, который вы создали вручную, как показано в следующем отредактированном коде:

```
using Microsoft.EntityFrameworkCore;

namespace WorkingWithEFCore.AutoGen
{
  public partial class Northwind : DbContext
  {
    public Northwind()
    {
    }

    public Northwind(DbContextOptions<Northwind> options)
      : base(options)
    {
    }

    public virtual DbSet<Category> Categories { get; set; } = null!;
    public virtual DbSet<Product> Products { get; set; } = null!;

    protected override void OnConfiguring(
      DbContextOptionsBuilder optionsBuilder)
    {
      if (!optionsBuilder.IsConfigured)
      {
      #warning To protect potentially sensitive information in your
      connection string, you should move it out of source code. You can
      avoid scaffolding the connection string by using the Name= syntax
      to read it from configuration - see https://go.microsoft.com/
      fwlink/?linkid=2131148. For more guidance on storing connection
      strings, see http://go.microsoft.com/fwlink/?LinkId=723263.
      optionsBuilder.UseSqlite("Filename=Northwind.db");
      }
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<Category>(entity =>
```

```
        {
          ...
        });

        modelBuilder.Entity<Product>(entity =>
        {
          ...
        });

        OnModelCreatingPartial(modelBuilder);
      }

      partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
    }
  }
```

Обратите внимание на следующие моменты:

- класс контекста данных `Northwind` является частичным (`partial`), что позволяет в дальнейшем расширять его и восстанавливать;
- класс содержит два конструктора: один без параметров по умолчанию и один позволяет передавать параметры. Это полезно в приложениях, где вы хотите указать строку подключения во время выполнения;
- для двух свойств `DbSet<T>`, представляющих таблицы `Categories` и `Products`, устанавливается значение `null-forgiving` (исключение нуля) с целью предотвратить появление предупреждений статического анализа компилятора во время компиляции. Это не имеет никакого эффекта во время выполнения;
- в методе `OnConfiguring`, если параметры не были указаны в конструкторе, по умолчанию используется строка подключения, которая ищет файл базы данных в текущей папке. Предупреждение компилятора напоминает, что вам не следует жестко кодировать информацию о безопасности в этой строке подключения;
- в методе `OnModelCreating` Fluent API используется для настройки двух классов сущностей, а затем вызывается частичный метод `OnModelCreatingPartial`. Это позволяет вам реализовать этот частичный метод в вашем частичном классе `Northwind` для того, чтобы добавить собственную конфигурацию Fluent API, которая не будет потеряна, если вы регенерируете классы модели.

8. Закройте автоматически сгенерированные файлы классов.

Настройка предварительных моделей

Наряду с поддержкой типов `DateOnly` и `TimeOnly` для использования с поставщиком базы данных SQLite, одной из новых функций, представленных в EF Core 6, является настройка предварительных моделей.

По мере усложнения моделей полагаться на соглашения для обнаружения типов сущностей и их свойств и успешного сопоставления их с таблицами и столбцами становится все труднее. Было бы полезно, если бы вы могли настраивать сами соглашения до того, как они будут использованы для анализа и построения модели.

Например, вы можете захотеть определить соглашение, согласно которому все свойства `string` по умолчанию должны иметь максимальную длину 50 символов или любые типы свойств, реализующие пользовательский интерфейс, не должны отображаться:

```
protected override void ConfigureConventions(
  ModelConfigurationBuilder configurationBuilder)
{
  configurationBuilder.Properties<string>().HaveMaxLength(50);
  configurationBuilder.IgnoreAny<IDoNotMap>();
}
```

В остальной части этой главы мы будем использовать классы, которые вы создали вручную.

Запрос данных из моделей EF Core

Теперь, когда у нас есть модель, сопоставляемая с базой данных Northwind и двумя ее таблицами, мы можем написать несколько простых запросов LINQ для извлечения данных. Намного больше информации о написании запросов LINQ вы получите в главе 11.

А пока просто напишите код и проанализируйте результат.

1. В начале файла `Program.cs` импортируйте основное пространство имен EF Core, чтобы включить использование метода расширения `Include` для предварительной выборки из соответствующей таблицы:

   ```
   using Microsoft.EntityFrameworkCore; // метод расширения Include
   ```

2. В конце файла `Program.cs` определите метод `QueryingCategories` и добавьте операторы для выполнения следующих действий:
   - создания экземпляра класса `Northwind`, который будет управлять базой данных. Экземпляры контекста БД рассчитаны на короткое время существования единицы работы. Их следует удалить как можно скорее, поэтому необходимо обернуть в оператор `using`. В главе 14 вы узнаете, как получить контекст базы данных с помощью внедрения зависимостей;
   - создания запроса для всех категорий, которые включают связанные товары;

- перечисления всех категорий с выводом названия и количества товаров в каждой из них.

```
static void QueryingCategories()
{
  using (Northwind db = new())
  {
    WriteLine("Categories and how many products they have:");

    // запрос на получение всех категорий и связанных с ними продуктов
    IQueryable<Category>? categories = db.Categories?
      .Include(c => c.Products);

    if (categories is null)
    {
      WriteLine("No categories found.");
      return;
    }

    // выполнение запроса и перечисление результатов
    foreach (Category c in categories)
    {
      WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
    }
  }
}
```

3. В начале файла `Program.cs` после вывода имени поставщика базы данных вызовите метод `QueryingCategories`, как показано ниже (выделено жирным шрифтом):

```
WriteLine($"Using {ProjectConstants.DatabaseProvider} database provider.");
QueryingCategories();
```

4. Запустите код и проанализируйте результат (при запуске с Visual Studio 2022 для Windows с использованием поставщика базы данных SQLite):

```
Using SQLite database provider.
Categories and how many products they have:
Using C:\Code\Chapter10\WorkingWithEFCore\bin\Debug\net6.0\Northwind.db
database file.
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.
```

Если вы работаете с Visual Studio Code, используя поставщик базы данных SQLite, то путь будет вести к папке WorkingWithEFCore. Если вы запускаетесь с помощью поставщика базы данных SQL Server, то путь к файлу базы данных не выводится.

Внимание! Если вы видите следующее исключение при использовании SQLite с Visual Studio 2022, то наиболее вероятная проблема заключается в том, что файл Northwind.db не копируется в выходной каталог. Убедитесь, что для параметра Copy to Output Directory (Копировать в выходной каталог) установлено значение Copy always (Копировать всегда):

```
Unhandled exception. Microsoft.Data.Sqlite.SqliteException
(0x80004005): SQLite Error 1: 'no such table: Categories'.
```

Фильтрация включенных сущностей

В EF Core 5.0 представлены *фильтруемые включения*; это значит, что вы можете указать лямбда-выражение в вызове метода `Include`, чтобы отфильтровать, какие сущности возвращаются в результатах.

1. В конце файла `Program.cs` определите метод `FilteredIncludes` и добавьте операторы для выполнения следующих действий:
   - создания экземпляра класса `Northwind`, который будет управлять базой данных;
   - предложения пользователю ввести минимальное значение для единиц на складе;
   - создания запроса для категорий, в которых содержатся продукты с минимальным количеством единиц на складе;
   - перечисления по категориям и продуктам с выводом названия и единиц на складе для каждой/каждого из них.

```
static void FilteredIncludes()
{
  using (Northwind db = new())
  {
    Write("Enter a minimum for units in stock: ");
    string unitsInStock = ReadLine() ?? "10";
    int stock = int.Parse(unitsInStock);

    IQueryable<Category>? categories = db.Categories?
      .Include(c => c.Products.Where(p => p.Stock >= stock));

    if (categories is null)
    {
      WriteLine("No categories found.");
      return;
    }
```

```
        foreach (Category c in categories)
        {
          WriteLine($"{c.CategoryName} has {c.Products.Count}
            products with a minimum of {stock} units in stock.");

          foreach(Product p in c.Products)
          {
            WriteLine($" {p.ProductName} has {p.Stock} units in stock.");
          }
        }
      }
    }
```

2. В файле `Program.cs` закомментируйте метод `QueryingCategories` и вызовите метод `FilteredIncludes`, как показано ниже:

```
WriteLine($"Using {ProjectConstants.DatabaseProvider} database provider.");
// QueryingCategories();
FilteredIncludes();
```

3. Запустите код, введите минимальное количество единиц, имеющееся на складе, например `100`, и проанализируйте результат:

```
Enter a minimum for units in stock: 100
Beverages has 2 products with a minimum of 100 units in stock.
  Sasquatch Ale has 111 units in stock.
  Rhönbräu Klosterbier has 125 units in stock.
Condiments has 2 products with a minimum of 100 units in stock.
  Grandma's Boysenberry Spread has 120 units in stock.
  Sirop d'érable has 113 units in stock.
Confections has 0 products with a minimum of 100 units in stock.
Dairy Products has 1 products with a minimum of 100 units in stock.
  Geitost has 112 units in stock.
Grains/Cereals has 1 products with a minimum of 100 units in stock.
  Gustaf's Knäckebröd has 104 units in stock.
Meat/Poultry has 1 products with a minimum of 100 units in stock.
  Pâté chinois has 115 units in stock.
Produce has 0 products with a minimum of 100 units in stock.
Seafood has 3 products with a minimum of 100 units in stock.
  Inlagd Sill has 112 units in stock.
  Boston Crab Meat has 123 units in stock.
  Röd Kaviar has 101 units in stock.
```

Символы Юникода в консоли Windows

В консоли, предоставляемой компанией Microsoft в версиях Windows до обновления Windows 10 Fall Creators Update, существует ограничение. По умолчанию консоль не может отображать символы Unicode, например такие, как в названии Rhönbräu.

Если у вас возникла эта проблема, то вы можете временно изменить кодовую страницу (также известную как набор символов) в консоли на Unicode UTF-8, введя следующую команду в командной строке перед запуском приложения:

```
chcp 65001
```

Фильтрация и сортировка товаров

Рассмотрим более сложный запрос, который фильтрует и сортирует данные.

1. В конце файла `Program.cs` определите метод `QueryingProducts` и добавьте операторы для выполнения следующих действий:
   - создания экземпляра класса `Northwind`, который будет управлять базой данных;
   - запроса у пользователя стоимости товаров. В отличие от предыдущего примера кода мы будем выполнять цикл до тех пор, пока введенная цена не станет действительной;
   - вывода товаров, стоимость которых выше указанной цены с помощью запроса LINQ;
   - циклической обработки результатов, вывода идентификатора, имени, стоимости (в долларах США) и количества единиц на складе.

```
static void QueryingProducts()
{
  using (Northwind db = new())
  {
    WriteLine("Products that cost more than a price, highest at top.");
    string? input;
    decimal price;

    do
    {
      Write("Enter a product price: ");
      input = ReadLine();
    } while (!decimal.TryParse(input, out price));

    IQueryable<Product>? products = db.Products?
      .Where(product => product.Cost > price)
      .OrderByDescending(product => product.Cost);

    if (products is null)
    {
      WriteLine("No products found.");
      return;
    }

    foreach (Product p in products)
    {
```

```
            WriteLine(
              "{0}: {1} costs {2:$#,##0.00} and has {3} in stock.",
              p.ProductId, p.ProductName, p.Cost, p.Stock);
            }
        }
    }
```

2. В файле `Program.cs` закомментируйте предыдущий метод и вызовите метод `QueryingProducts`:
3. Запустите код, введите число `50`, когда программа запросит указать стоимость товара, и проанализируйте результат:

```
Products that cost more than a price, highest at top.
Enter a product price: 50
38: Côte de Blaye costs $263.50 and has 17 in stock.
29: Thüringer Rostbratwurst costs $123.79 and has 0 in stock.
9: Mishi Kobe Niku costs $97.00 and has 29 in stock.
20: Sir Rodney's Marmalade costs $81.00 and has 40 in stock.
18: Carnarvon Tigers costs $62.50 and has 42 in stock.
59: Raclette Courdavault costs $55.00 and has 79 in stock.
51: Manjimup Dried Apples costs $53.00 and has 20 in stock.
```

Получение сгенерированного SQL-кода

Вам может быть интересно, насколько хорошо написаны операторы SQL, генерирующиеся из запросов C#, которые мы пишем. В EF Core 5.0 появился быстрый и простой способ увидеть сгенерированный SQL-код.

1. В методе `FilteredIncludes` перед использованием оператора `foreach` для перечисления запроса добавьте оператор для вывода сгенерированного SQL, как показано ниже:

```
WriteLine($"ToQueryString: {categories.ToQueryString()}");

foreach (Category c in categories)
```

2. В файле `Program.cs` закомментируйте вызов метода `QueryingProducts` и раскомментируйте вызов метода `FilteredIncludes`.
3. Запустите код, введите минимальное количество единиц, имеющихся на складе, например `99`, и проанализируйте результат (при работе с SQLite):

```
Enter a minimum for units in stock: 99
Using SQLite database provider.
ToQueryString: .param set @_stock_0 99

SELECT "c"."CategoryId", "c"."CategoryName", "c"."Description",
"t"."ProductId", "t"."CategoryId", "t"."UnitPrice", "t"."Discontinued",
```

```
"t"."ProductName", "t"."UnitsInStock"
FROM "Categories" AS "c"
LEFT JOIN (
    SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
    FROM "Products" AS "p"
    WHERE ("p"."UnitsInStock" >= @_stock_0)
) AS "t" ON "c"."CategoryId" = "t"."CategoryId"
ORDER BY "c"."CategoryId"
Beverages has 2 products with a minimum of 99 units in stock.
    Sasquatch Ale has 111 units in stock.
    Rhönbräu Klosterbier has 125 units in stock.
...
```

Обратите внимание, что для параметра SQL `@_stock_0` установлено минимальное складское значение `99`.

Для SQL Server генерируемый SQL немного другой, например, использует квадратные скобки вместо двойных кавычек вокруг имен объектов, как показано в следующем выводе:

```
Enter a minimum for units in stock: 99
Using SqlServer database provider.
ToQueryString: DECLARE @__stock_0 smallint = CAST(99 AS smallint);

SELECT [c].[CategoryId], [c].[CategoryName], [c].[Description],
[t].[ProductId], [t].[CategoryId], [t].[UnitPrice], [t].[Discontinued],
[t].[ProductName], [t].
[UnitsInStock]
FROM [Categories] AS [c]
LEFT JOIN (
    SELECT [p].[ProductId], [p].[CategoryId], [p].[UnitPrice], [p].
[Discontinued], [p].[ProductName], [p].[UnitsInStock]
    FROM [Products] AS [p]
    WHERE [p].[UnitsInStock] >= @__stock_0
) AS [t] ON [c].[CategoryId] = [t].[CategoryId]
ORDER BY [c].[CategoryId], [t].[ProductId]
```

Логирование EF Core с помощью провайдера логов для пользователя

Чтобы отслеживать взаимодействие EF Core и базы данных, нам потребуется включить логирование (журналирование). Для этого нужно выполнить следующие две задачи:

- регистрацию *провайдера логов*;
- реализацию *логгера*.

Рассмотрим пример.

1. Добавьте в проект файл `ConsoleLogger.cs`.
2. Отредактируйте код файла, определив два класса, реализующие интерфейсы `IloggerProvider` и `Ilogger`, как показано в коде ниже, и обратите внимание на следующие моменты:
   - класс `ConsoleLoggerProvider` возвращает экземпляр `ConsoleLogger`. Ему не нужны неуправляемые ресурсы, поэтому метод `Dispose` ничего не выполняет, но должен существовать;
   - класс `ConsoleLogger` отключен для уровней логирования `None`, `Trace` и `Information`. Включен для всех остальных уровней;
   - класс `ConsoleLogger` реализует свой метод `Log` путем записи в `Console`.

```
using Microsoft.Extensions.Logging; // ILoggerProvider, ILogger,
LogLevel

using static System.Console;

namespace Packt.Shared;

public class ConsoleLoggerProvider : ILoggerProvider
{
  public ILogger CreateLogger(string categoryName)
  {
    // у нас могут быть разные реализации регистратора
    // для разных значений categoryName, но у нас есть только одна
    return new ConsoleLogger();
  }

  // если средство ведения журнала использует неуправляемые ресурсы,
  // то вы можете освободить их здесь
  public void Dispose() { }
}

public class ConsoleLogger : ILogger
{
  // если средство ведения журнала использует неуправляемые ресурсы,
  // то можно вернуть здесь класс, реализующий Idisposable
  public IDisposable BeginScope<TState>(TState state)
  {
    return null;
  }

  public bool IsEnabled(LogLevel logLevel)
  {
    // чтобы избежать переполнения журнала, можно выполнить фильтрацию
    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)
    {
      // пишем в журнал уровень и идентификатор события
      Write($"Level: {logLevel}, Event Id: {eventId.Id}");

      // выводим только существующее состояние или исключение
      if (state != null)
      {
        Write($", State: {state}");
      }

      if (exception != null)
      {
        Write($", Exception: {exception.Message}");
      }
      WriteLine();
    }
  }
```

3. В начало файла `Program.cs` добавьте следующие операторы для импорта пространств имен:

```
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
```

4. Мы уже использовали метод `ToQueryString`, чтобы получить SQL для `FilteredIncludes`, поэтому нам не нужно добавлять логирование в этот метод. В методы `QueryingCategories` и `QueryingProducts` добавьте операторы в блоке `using` для контекста базы данных Northwind, которые получают сервис логирования и регистрируют наш консольный логер:

```
using (Northwind db = new())
{
  ILoggerFactory loggerFactory = db.GetService<ILoggerFactory>();
  loggerFactory.AddProvider(new ConsoleLoggerProvider());
```

5. В начале файла `Program.cs` закомментируйте вызов метода `FilteredIncludes` и раскомментируйте вызов метода `QueryingProducts`.
6. Запустите код и проанализируйте логи, которые частично показаны в следующем выводе:

```
...
Level: Debug, Event Id: 20000, State: Opening connection to database
'main' on server '/Users/markjprice/Code/Chapter10/WorkingWithEFCore/
Northwind.db'.
Level: Debug, Event Id: 20001, State: Opened connection to database 'main'
on server '/Users/markjprice/Code/Chapter10/WorkingWithEFCore/Northwind.
db'.
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[@__
price_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE "p"."UnitPrice" > @__price_0
ORDER BY "product"."UnitPrice" DESC
...
```

Ваши логи могут отличаться от приведенных выше в зависимости от выбранного вами поставщика базы данных и редактора кода, а также от будущих улучшений EF Core. Пока же обратите внимание, что разные события, такие как открытие соединения или выполнение команды, имеют разные идентификаторы.

Фильтрация логов по значениям, зависящим от конкретного поставщика

Значения идентификатора события и их смысл будут зависеть от поставщика данных .NET. Если мы хотим узнать, как запрос LINQ был преобразован в операторы SQL и как он выполняется, то идентификатор события для вывода должен иметь значение `Id`, равное `20100`.

1. Измените метод `Log` в классе `ConsoleLogger` для вывода только событий с `Id`, равным `20100`, как показано ниже (выделено жирным шрифтом):

```
public void Log<TState>(LogLevel logLevel, EventId eventId,
  TState state, Exception? exception,
  Func<TState, Exception, string> formatter)
{
  if (eventId.Id == 20100)
  {
    // пишем в журнал уровень и идентификатор события
```

```
    Write("Level: {0}, Event Id: {1}, Event: {2}",
      logLevel, eventId.Id, eventId.Name);

    // выводим только существующее состояние или исключение
    if (state != null)
    {
      Write($", State: {state}");
    }
    if (exception != null)
    {
      Write($", Exception: {exception.Message}");
    }
    WriteLine();
  }
}
```

2. В файле `Program.cs` раскомментируйте метод `QueryingCategories` и закомментируйте другие методы, чтобы можно было отслеживать операторы SQL, которые генерируются при объединении двух таблиц.
3. Запустите код и обратите внимание на следующие операторы SQL, которые подверглись логированию, как показано в следующем выводе (отредактирован для экономии места):

```
Using SQLServer database provider.
Categories and how many products they have:
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT [c].[CategoryId], [c].[CategoryName], [c].[Description], [p].
[ProductId], [p].[CategoryId], [p].[UnitPrice], [p].[Discontinued], [p].
[ProductName], [p].[UnitsInStock]
FROM [Categories] AS [c]
LEFT JOIN [Products] AS [p] ON [c].[CategoryId] = [p].[CategoryId]
ORDER BY [c].[CategoryId], [p].[ProductId]
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.
```

Логирование с помощью тегов запросов

При логировании запросов LINQ может быть сложно сопоставить сообщения журнала в сложных сценариях. В EF Core 2.2 добавлена функция тегов запросов, позволяющая добавлять комментарии SQL в журнал.

Вы можете аннотировать запрос LINQ с помощью метода `TagWith`:

```
IQueryable<Product>? products = db.Products?
  .TagWith("Products filtered by price and sorted.")
  .Where(product => product.Cost > price)
  .OrderByDescending(product => product.Cost);
```

Этот код добавит комментарий SQL в журнал, как показано в следующем выводе:

```
-- Products filtered by price and sorted.
```

Сопоставление с образцом с помощью оператора Like

EF Core поддерживает распространенные операторы SQL, включая `Like` для сопоставления с образцом.

1. В конце файла `Program.cs` добавьте метод `QueryingWithLike`, как показано в коде ниже, и обратите внимание на следующие моменты:
   - мы включили ведение журнала;
   - мы предлагаем пользователю ввести фрагмент наименования товара, а затем применить метод `EF.Functions.Like` для поиска в любом месте свойства `ProductName`;
   - для каждого сопоставляемого товара мы выводим его название, наличие на складе и информацию о том, снят ли он с производства.

```
static void QueryingWithLike()
{
  using (Northwind db = new())
  {
    ILoggerFactory loggerFactory = db.GetService<ILoggerFactory>();
    loggerFactory.AddProvider(new ConsoleLoggerProvider());

    Write("Enter part of a product name: ");
    string? input = ReadLine();

    IQueryable<Product>? products = db.Products?
    .Where(p => EF.Functions.Like(p.ProductName, $"%{input}%"));

    if (products is null)
    {
      WriteLine("No products found.");
      return;
    }

    foreach (Product p in products)
    {
```

```
            WriteLine("{0} has {1} units in stock. Discontinued? {2}",
              p.ProductName, p.Stock, p.Discontinued);
          }
        }
      }
```

2. В файле `Program.cs` закомментируйте существующие методы и вызовите метод `QueryingWithLike`.
3. Запустите код, введите фрагмент наименования товара, например `che`, и проанализируйте результат:

```
Using SQLServer database provider.
Enter part of a product name: che
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[@__
Format_1='?' (Size = 40)], 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
```

В EF Core 6.0 появилась еще одна полезная функция — EF.Functions.Random, которая сопоставляется с функцией базы данных, возвращающей псевдослучайное число от 0 до 1 без исключения. Например, вы можете умножить случайное число на количество строк в таблице, чтобы выбрать одну случайную строку из этой таблицы.

Определение глобальных фильтров

Продукция Northwind может быть снята с производства, поэтому возникает необходимость убедиться в том, что такие товары никогда не будут возвращаться в результате, даже если разработчик не использует `Where` для их фильтрации в своих запросах.

1. В файле `Northwind.cs` измените метод `OnModelCreating`, чтобы добавить глобальный фильтр для удаления снятых с производства товаров, как показано ниже (выделено жирным шрифтом):

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  ...
```

```
    // глобальный фильтр для удаления снятых с производства товаров
    modelBuilder.Entity<Product>()
      .HasQueryFilter(p => !p.Discontinued);
  }
```

2. Запустите код, введите фрагмент наименования товара `che`, проанализируйте результат и обратите внимание, что *Chef Anton's Gumbo Mix* теперь отсутствует, поскольку сгенерированный оператор SQL содержит фильтр для столбца `Discontinued`, как показано ниже (выделено жирным шрифтом):

```
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 шаблонов с помощью EF Core

EF Core предоставляет три наиболее популярные *схемы загрузки шаблонов*:

- *жадную* (немедленную) — загружает данные на ранней стадии;
- *ленивую* (отложенную) — автоматически загружает данные непосредственно перед тем, как они потребуются;
- *явную* — загружает данные вручную.

В этом разделе мы рассмотрим каждую из них.

Жадная загрузка сущностей

В методе `QueryingCategories` программного кода в настоящий момент свойство `Categories` служит для циклического прохождения каждой категории и вывода имени категории и количества товаров в ней.

Этот код работает потому, что при написании запроса мы включили жадную загрузку, вызвав метод `Include` для связанных товаров.

Посмотрим, что произойдет, если мы не вызовем метод `Include`.

1. Измените запрос, закомментировав вызов метода `Include`:

```
IQueryable<Category>? categories =
  db.Categories; //.Include(c => c.Products);
```

2. В файле `Program.cs` закомментируйте все методы, за исключением `QueryingCategories`.
3. Запустите код и проанализируйте результат:

```
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` — это экземпляр класса `Category`, имеющий свойство `Products`, которое, в свою очередь, представляет собой список товаров в данной категории. Поскольку исходный запрос осуществляет выборку только из таблицы `Categories`, значение данного свойства является пустым для каждой категории.

Включение ленивой загрузки

Ленивая загрузка была введена в EF Core 2.1 и может автоматически загружать отсутствующие связанные данные.

Чтобы разрешить ленивую загрузку, разработчики должны выполнить следующие действия:

- сослаться на пакет NuGet для прокси;
- настроить ленивую загрузку для использования прокси.

Посмотрим, как это работает.

1. В проекте `WorkingWithEFCore` добавьте ссылку на пакет для прокси-серверов EF Core:

```
<PackageReference
  Include="Microsoft.EntityFrameworkCore.Proxies"
  Version="6.0.0" />
```

2. Соберите проект для восстановления пакетов.
3. Откройте файл `Northwind.cs`, вызовите метод расширения, чтобы применить прокси с ленивой загрузкой в начале метода `OnConfiguring`, как показано ниже (выделено жирным шрифтом):

```
protected override void OnConfiguring(
  DbContextOptionsBuilder optionsBuilder)
```

```
{
  optionsBuilder.UseLazyLoadingProxies();
```

Теперь прокси будет автоматически проверять состояние загрузки элементов при каждом циклическом перечислении и попытке считывания свойства `Products`. Если требуемые элементы еще не загружены, то он «лениво» загрузит их для нас, выполнив оператор `SELECT` для подгрузки только набора товаров выбранной категории, после чего корректное количество товаров будет возвращено в программный вывод.

4. Запустите код и обратите внимание, что количество продуктов теперь правильное. Но вы увидите, что проблема ленивой загрузки заключается в необходимости каждый раз связываться с сервером базы данных для постепенной выборки всех данных, как показано в следующем фрагменте вывода:

```
Categories and how many products they have:
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryId", "c"."CategoryName", "c"."Description" FROM
"Categories" AS "c"
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[@
p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryId" = @ p_0)
Beverages has 11 products.
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[@
p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryId" = @ p_0)
Condiments has 11 products.
```

Явная загрузка сущностей

Еще один тип загрузки — явная загрузка. Работает так же, как ленивая, отличаясь только тем, что вы контролируете, какие именно связанные данные будут загружены в тот или иной момент времени.

1. В начале файла `Program.cs` импортируйте пространство имен отслеживания изменений, чтобы мы могли использовать класс `CollectionEntry` для ручной загрузки связанных объектов:

```
using Microsoft.EntityFrameworkCore.ChangeTracking; // класс
CollectionEntry
```

2. В методе `QueryingCategories` измените следующие операторы, чтобы отключить ленивую загрузку, а затем запросите у пользователя, хочет ли он применить жадную и явную загрузки:

```
IQueryable<Category>? categories;
  // = db.Categories;
  // .Include(c => c.Products);

db.ChangeTracker.LazyLoadingEnabled = false;

Write("Enable eager loading? (Y/N): ");
bool eagerloading = (ReadKey().Key == ConsoleKey.Y);
bool explicitloading = false;
WriteLine();

if (eagerloading)
{
  categories = db.Categories?.Include(c => c.Products);
}
else
{
  categories = db.Categories;

  Write("Enable explicit loading? (Y/N): ");
  explicitloading = (ReadKey().Key == ConsoleKey.Y);
  WriteLine();
}
```

3. В цикле `foreach` перед вызовом метода `WriteLine` добавьте следующие операторы, чтобы проверить, включена ли явная загрузка. Если да, то спросите пользователя, хочет ли он явно загрузить каждую отдельную категорию:

```
if (explicitloading)
{
  Write($"Explicitly load products for {c.CategoryName}? (Y/N): ");
  ConsoleKeyInfo key = ReadKey();
  WriteLine();
  if (key.Key == ConsoleKey.Y)
  {
    CollectionEntry<Category, Product> products =
      db.Entry(c).Collection(c2 => c2.Products);
    if (!products.IsLoaded) products.Load();
  }
}
WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
```

4. Запустите код:
   1) нажмите клавишу `N`, чтобы отключить жадную загрузку;
   2) затем нажмите клавишу `Y`, чтобы включить явную;
   3) по желанию для каждой категории нажмите клавишу `Y` или `N` для загрузки товаров, содержащихся в этой категории.

Я решил загружать товары, относящиеся только к двум из восьми категорий: Beverages и Seafood, как показано в следующем фрагменте, который был отредактирован для экономии места:

```
Categories and how many products they have:
Enable eager loading? (Y/N): n
Enable explicit loading? (Y/N): y
Level: Debug, Event Id: 20100, 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: 20100, State: Executing
DbCommand [Parameters=[@p_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryId" = @ p_0)
Beverages has 11 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): y
Level: Debug, Event ID: 20100, State: Executing
DbCommand [Parameters=[@ p_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryId" = @ p_0)
Seafood has 12 products.
```

Внимательно выбирайте схему загрузки данных, подходящую именно для вашего программного кода. Использование ленивой загрузки может буквально сделать вас ленивым разработчиком баз данных! Больше о схемах загрузки можно узнать на сайте https://docs.microsoft.com/ru-ru/ef/core/querying/related-data.

Управление данными с помощью EF Core

EF Core позволяет легко добавлять, обновлять и удалять сущности.

Класс `DbContext` автоматически поддерживает отслеживание изменений, что позволяет отслеживать множественные изменения сущностей локально, включая добавление новых, изменение существующих и их удаление. Когда вы будете готовы отправить эти изменения в основную базу данных, вызовите метод `SaveChanges`. Количество успешно измененных объектов будет возвращено.

Добавление сущностей

Добавим новую строку в таблицу.

1. В файле `Program.cs` создайте метод `AddProduct`:

```
static bool AddProduct(
  int categoryId, string productName, decimal? price)
{
  using (Northwind db = new())
  {
    Product p = new()
    {
      CategoryId = categoryId,
      ProductName = productName,
      Cost = price
    };

    // помечаем продукт как добавленный к отслеживанию изменений
    db.Products.Add(p);

    // сохранение отслеживаемых изменений в базе данных
    int affected = db.SaveChanges();
    return (affected == 1);
  }
}
```

2. В файле `Program.cs` создайте метод `ListProducts`, который выводит идентификатор, наименование, стоимость, запас и товары, снятые с производства, отсортированные по стоимости от самого дорогого:

```
static void ListProducts()
{
  using (Northwind db = new())
  {
    WriteLine("{0,-3} {1,-35} {2,8} {3,5} {4}",
      "Id", "Product Name", "Cost", "Stock", "Disc.");

    foreach (Product p in db.Products
      .OrderByDescending(product => product.Cost))
```

```
      {
        WriteLine("{0:000} {1,-35} {2,8:$#,##0.00} {3,5} {4}",
          p.ProductId, p.ProductName, p.Cost, p.Stock, p.Discontinued);
      }
    }
  }
```

Помните, что `1,-35` означает, что аргумент 1 должен выравниваться по левому краю в столбце шириной 35 символов, а `3,5` — что аргумент 3 должен выравниваться по правому краю в столбце шириной пять символов.

3. В файле `Program.cs` закомментируйте предыдущие вызовы методов, а затем вызовите методы `AddProduct` и `ListProducts`:

```
// QueryingCategories();
// FilteredIncludes();
// QueryingProducts();
// QueryingWithLike();

if (AddProduct(categoryId: 6,
  productName: "Bob's Burgers", price: 500M))
  {
    WriteLine("Add product successful.");
  }

ListProducts();
```

4. Запустите код, проанализируйте результат и обратите внимание, что новый товар был добавлен, как показано ниже во фрагменте вывода:

```
Add product successful.
Id Product Name              Cost Stock Disc.
078 Bob's Burgers         $500.00       False
038 Côte de Blaye         $263.50    17 False
020 Sir Rodney's Marmalade $81.00    40 False
...
```

Обновление сущностей

Изменим существующую строку в таблице.

1. В файле `Program.cs` добавьте метод для увеличения цены первого товара, название которого начинается с указанного значения (в нашем примере мы будем использовать `Bob`) на указанную сумму, например 20 долларов США:

```
static bool IncreaseProductPrice(
  string productNameStartsWith, decimal amount)
{
  using (Northwind db = new())
  {
```

```
    // получаем первый продукт, название которого начинается с имени
    Product updateProduct = db.Products.First(
      p => p.ProductName.StartsWith(productNameStartsWith));

    updateProduct.Cost += amount;

    int affected = db.SaveChanges();
    return (affected == 1);
  }
}
```

2. В файле `Program.cs` закомментируйте весь блок `if`, вызывающий метод `AddProduct`, и добавьте вызов метода `IncreaseProductPrice` перед вызовом списка товаров:

```
/*
if (AddProduct(categoryId: 6,
  productName: "Bob's Burgers", price: 500M))
{
  WriteLine("Add product successful.");
}
*/

if (IncreaseProductPrice(
  productNameStartsWith: "Bob", amount: 20M))
{
  WriteLine("Update product price successful.");
}

ListProducts();
```

3. Запустите код, проанализируйте результат и обратите внимание, что стоимость уже существующего товара `Bob's Burgers` увеличилась на 20 долларов:

```
Update product price successful.
Id Product Name             Cost Stock Disc.
078 Bob's Burgers         $520.00       False
038 Côte de Blaye         $263.50    17 False
020 Sir Rodney's Marmalade $81.00    40 False
...
```

Удаление сущностей

Вы можете удалить отдельные сущности, воспользовавшись методом `Remove`. Метод `RemoveRange` более эффективен, когда требуется удалить несколько сущностей.

Посмотрим, как удалить строки из таблицы.

1. В конце файла `Program.cs` добавьте метод для удаления всех товаров, название которых начинается с указанного значения (в нашем примере со слова `Bob`):

```
static int DeleteProducts(string productNameStartsWith)
{
  using (Northwind db = new())
  {
    IQueryable<Product>? products = db.Products?.Where(
      p => p.ProductName.StartsWith(productNameStartsWith));

    if (products is null)
    {
      WriteLine("No products found to delete.");
      return 0;
    }
    else
    {
      db.Products.RemoveRange(products);
    }

    int affected = db.SaveChanges();
    return affected;
  }
}
```

2. В файле `Program.cs` закомментируйте весь блок оператора `if`, вызывающий метод `IncreaseProductPrice`, и добавьте вызов метода `DeleteProducts`, как показано ниже:

```
int deleted = DeleteProducts(productNameStartsWith: "Bob");
WriteLine($"{deleted} product(s) were deleted.");
```

3. Запустите код и проанализируйте результат:

```
1 product(s) were deleted.
```

Если несколько названий товаров начинаются со слова `Bob`, то все они удаляются. В качестве дополнительной задачи измените операторы, добавив три новых товара, названия которых начинаются со слова `Bob`, а затем удалите их.

Объединение контекстов базы данных

Класс `DbContext` требует очистки неуправляемых ресурсов и спроектирован по принципу «одна единица работы» (single-unit-of-work). В предыдущих примерах кода мы создавали все экземпляры `Northwind`, наследуемые от класса `DbContext`, в блоке `using`, чтобы метод `Dispose` правильно вызывался в конце каждой единицы работы.

Функция ASP.NET Core, связанная с EF Core, делает ваш код более эффективным за счет объединения контекстов базы данных при создании сайтов и веб-сервисов. Это позволяет вам создавать и удалять сколько угодно наследуемых от класса `DbContext` объектов и знать, что ваш код по-прежнему максимально эффективен.

Работа с транзакциями

Каждый раз, когда вы вызываете метод `SaveChanges`, система запускает *неявную транзакцию* таким образом, что если нечто пойдет не так, то система автоматически отменит все внесенные изменения. Если же несколько изменений в транзакции завершаются успешно, то транзакция и все изменения считаются зафиксированными.

Транзакции позволяют сохранить целостность вашей 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 — «неделимый») — совершаются либо все операции текущей транзакции, либо ни одна из операций.
- *C* (consistent — «согласованный») — ваша база данных согласована как до, так и после совершения транзакции. Это зависит от логики вашего программного кода. Например, при переводе денег между банковскими счетами ваша бизнес-логика должна гарантировать, что, дебетуя 100 долларов США на одном счете, вы кредитуете 100 долларов США на другом.
- *I* (isolated — «изолированный») — во время выполнения транзакции вносимые изменения скрыты от других процессов. Существует несколько уровней изоляции, которые вы можете установить (табл. 10.4). Чем выше уровень, тем выше целостность данных. Однако для этого требуется применить множество блокировок, что негативно отразится на работе других процессов. Снимки состояния — особый уровень изоляции, при использовании которого создаются копии строк таблицы, позволяющие избежать блокировок, что приводит к увеличению размера базы данных при выполнении транзакций.
- *D* (durable — «надежный») — если при выполнении транзакции возникнет ошибка, то исходное состояние базы данных можно восстановить. Это часто реализуется в виде двухфазной фиксации и журналов транзакций. Как только транзакция зафиксирована, она гарантированно завершится, даже если впоследствии возникнут ошибки. В данном случае «надежный» — антоним «неустойчивого» (volatile).

Управление транзакциями с помощью уровней изоляции

Разработчик может контролировать транзакции, устанавливая *уровень изоляции* (см. табл. 10.4).

Таблица 10.4. Уровни изоляции

Уровень изоляции	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 вставка и удаление данных	Нет
`Snapshot`	Нет	Нет

Определение явной транзакции

Вы можете управлять явными транзакциями с помощью свойства `Database` контекста базы данных.

1. В файле `Program.cs` импортируйте пространство имен хранилища EF Core, чтобы использовать интерфейс `IDbContextTransaction`:

```
using Microsoft.EntityFrameworkCore.Storage;
// интерфейс IDbContextTransaction
```

2. В методе `DeleteProducts` после создания экземпляра переменной `db` добавьте следующие операторы, чтобы запустить явную транзакцию и вывести ее уровень изоляции. В конце метода подтвердите транзакцию и закройте фигурную скобку:

```
static int DeleteProducts(string name)
{
  using (Northwind db = new())
  {
    using (IDbContextTransaction t = db.Database.BeginTransaction())
    {
      WriteLine("Transaction isolation level: {0}",
        arg0: t.GetDbTransaction().IsolationLevel);
```

```
        IQueryable<Product>? products = db.Products?.Where(
          p => p.ProductName.StartsWith(name));

        if (products is null)
        {
        WriteLine("No products found to delete.");
          return 0;
        }
        else
        {
        db.Products.RemoveRange(products);
        }

        int affected = db.SaveChanges();
        t.Commit();
        return affected;
      }
    }
  }
```

3. Запустите код и проанализируйте результат с помощью SQL Server:

```
Transaction isolation level: ReadCommitted
```

4. Запустите код и проанализируйте результат с помощью SQLite:

```
Transaction isolation level: Serializable
```

Модели EF Core под названием Code First

Иногда у вас не будет существующей базы данных. Вместо этого вы определяете модель EF Core как Code First (сначала код), а затем EF Core может сгенерировать соответствующую базу данных, используя API для создания и сброса.

API для создания и удаления следует использовать только во время разработки. Вы же не захотите, чтобы приложение удалило производственную базу данных, как только вы его выпустите!

Например, нам может понадобиться создать приложение для управления студентами и курсами для академии. Один студент может записаться на несколько курсов. Один курс могут посещать несколько студентов. Это пример отношения «многие ко многим» между студентами и курсами.

Смоделируем этот пример.

1. Откройте редактор кода и создайте консольное приложение `CoursesAndStudents` в рабочей области/решении `Chapter10`.

2. В Visual Studio настройте стартовый проект для решения в соответствии с текущим выбором.
3. В Visual Studio Code выберите `CoursesAndStudents` в качестве активного проекта OmniSharp.
4. В проекте `CoursesAndStudents` добавьте ссылки на пакеты для следующих пакетов:
   - `Microsoft.EntityFrameworkCore.Sqlite`;
   - `Microsoft.EntityFrameworkCore.SqlServer`;
   - `Microsoft.EntityFrameworkCore.Design`.
5. Соберите проект `CoursesAndStudents` для восстановления пакетов.
6. Добавьте файлы `Academy.cs`, `Student.cs` и `Course.cs`.
7. Измените файл `Student.cs` и обратите внимание, что он является POCO (простой старый объект CLR) без атрибутов, дополняющих класс:

```
namespace CoursesAndStudents;

public class Student
{
  public int StudentId { get; set; }
  public string? FirstName { get; set; }
  public string? LastName { get; set; }

  public ICollection<Course>? Courses { get; set; }
}
```

8. Измените файл `Course.cs` и обратите внимание, что мы дополнили свойство `Title` некоторыми атрибутами, чтобы предоставить модели больше информации:

```
using System.ComponentModel.DataAnnotations;

namespace CoursesAndStudents;

public class Course
{
  public int CourseId { get; set; }
  [Required]
  [StringLength(60)]
  public string? Title { get; set; }

  public ICollection<Student>? Students { get; set; }
}
```

9. Измените файл `Academy.cs`:

```
using Microsoft.EntityFrameworkCore;

using static System.Console;
```

```
namespace CoursesAndStudents;

public class Academy : DbContext
{
  public DbSet<Student>? Students { get; set; }
  public DbSet<Course>? Courses { get; set; }

  protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
  {
    string path = Path.Combine(
      Environment.CurrentDirectory, "Academy.db");

    WriteLine($"Using {path} database file.");

    optionsBuilder.UseSqlite($"Filename={path}");

    // optionsBuilder.UseSqlServer(@"Data Source=.;Initial Catalog=
    Academy;Integrated Security=true;MultipleActiveResultSets=true;");
  }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    // правила проверки Fluent API

    modelBuilder.Entity<Student>()
      .Property(s => s.LastName).HasMaxLength(30).IsRequired();

      // заполнение базы данных образцами данных

      Student alice = new() { StudentId = 1,
        FirstName = "Alice", LastName = "Jones" };

      Student bob = new() { StudentId = 2,
        FirstName = "Bob", LastName = "Smith" };

      Student cecilia = new() { StudentId = 3,
        FirstName = "Cecilia", LastName = "Ramirez" };

      Course csharp = new()
      {
        CourseId = 1,
        Title = "C# 10 and .NET 6",
      };

      Course webdev = new()
      {
        CourseId = 2,
        Title = "Web Development",
      };
```

```
        Course python = new()
        {
          CourseId = 3,
          Title = "Python for Beginners",
        };

        modelBuilder.Entity<Student>()
          .HasData(alice, bob, cecilia);

        modelBuilder.Entity<Course>()
          .HasData(csharp, webdev, python);

        modelBuilder.Entity<Course>()
          .HasMany(c => c.Students)
          .WithMany(s => s.Courses)
          .UsingEntity(e => e.HasData(
            // все студенты записались на курс C#
            new { CoursesCourseId = 1, StudentsStudentId = 1 },
            new { CoursesCourseId = 1, StudentsStudentId = 2 },
            new { CoursesCourseId = 1, StudentsStudentId = 3 },
            // только Боб записался на Web Dev
            new { CoursesCourseId = 2, StudentsStudentId = 2 },
            // только Сесилия записалась на Python
            new { CoursesCourseId = 3, StudentsStudentId = 3 }
          ));
    }
}
```

Используйте анонимный тип для предоставления данных для промежуточной таблицы в отношениях «многие ко многим». Имена свойств соответствуют соглашению об именовании NavigationPropertyNamePropertyName, например, Courses — это имя навигационного свойства, а CourseId — имя свойства, поэтому CoursesCourseId будет именем свойства анонимного типа.

10. В начале файла `Program.cs` импортируйте пространство имен для EF Core и работы с задачами, а также статически импортируйте класс `Console`:

```
using Microsoft.EntityFrameworkCore; // для GenerateCreateScript()
using CoursesAndStudents; // БД Academy

using static System.Console;
```

11. В файле `Program.cs` добавьте операторы для создания экземпляра контекста базы данных `Academy` и используйте его для удаления базы данных, если она существует, создания базы данных из модели и вывода сценария SQL, который она использует, а затем перечислите студентов и их курсы:

```
using (Academy a = new())
{
```

```
    bool deleted = await a.Database.EnsureDeletedAsync();
    WriteLine($"Database deleted: {deleted}");

    bool created = await a.Database.EnsureCreatedAsync();
    WriteLine($"Database created: {created}");

    WriteLine("SQL script used to create database:");
    WriteLine(a.Database.GenerateCreateScript());

    foreach (Student s in a.Students.Include(s => s.Courses))
    {
      WriteLine("{0} {1} attends the following {2} courses:",
        s.FirstName, s.LastName, s.Courses.Count);

      foreach (Course c in s.Courses)
      {
        WriteLine($" {c.Title}");
      }
    }
  }
```

12. Запустите код и обратите внимание, что при первом его запуске не нужно будет удалять базу данных, поскольку она еще не существует:

```
Using C:\Code\Chapter10\CoursesAndStudents\bin\Debug\net6.0\Academy.db
database file.
Database deleted: False
Database created: True
SQL script used to create database:
CREATE TABLE "Courses" (
    "CourseId" INTEGER NOT NULL CONSTRAINT "PK_Courses" PRIMARY KEY
AUTOINCREMENT,
    "Title" TEXT NOT NULL
);

CREATE TABLE "Students" (
    "StudentId" INTEGER NOT NULL CONSTRAINT "PK_Students" PRIMARY KEY
AUTOINCREMENT,
    "FirstName" TEXT NULL,
    "LastName" TEXT NOT NULL

);
CREATE TABLE "CourseStudent" (
    "CoursesCourseId" INTEGER NOT NULL,
    "StudentsStudentId" INTEGER NOT NULL,
    CONSTRAINT "PK_CourseStudent" PRIMARY KEY ("CoursesCourseId",
"StudentsStudentId"),
    CONSTRAINT "FK_CourseStudent_Courses_CoursesCourseId" FOREIGN KEY
("CoursesCourseId") REFERENCES "Courses" ("CourseId") ON DELETE CASCADE,
```

```
    CONSTRAINT "FK_CourseStudent_Students_StudentsStudentId" FOREIGN
KEY ("StudentsStudentId") REFERENCES "Students" ("StudentId") ON DELETE
CASCADE
);

INSERT INTO "Courses" ("CourseId", "Title")
VALUES (1, 'C# 10 and .NET 6');

INSERT INTO "Courses" ("CourseId", "Title")
VALUES (2, 'Web Development');

INSERT INTO "Courses" ("CourseId", "Title")
VALUES (3, 'Python for Beginners');

INSERT INTO "Students" ("StudentId", "FirstName", "LastName")
VALUES (1, 'Alice', 'Jones');

INSERT INTO "Students" ("StudentId", "FirstName", "LastName")
VALUES (2, 'Bob', 'Smith');

INSERT INTO "Students" ("StudentId", "FirstName", "LastName")
VALUES (3, 'Cecilia', 'Ramirez');

INSERT INTO "CourseStudent" ("CoursesCourseId", "StudentsStudentId")
VALUES (1, 1);

INSERT INTO "CourseStudent" ("CoursesCourseId", "StudentsStudentId")
VALUES (1, 2);

INSERT INTO "CourseStudent" ("CoursesCourseId", "StudentsStudentId")
VALUES (2, 2);

INSERT INTO "CourseStudent" ("CoursesCourseId", "StudentsStudentId")
VALUES (1, 3);

INSERT INTO "CourseStudent" ("CoursesCourseId", "StudentsStudentId")
VALUES (3, 3);

CREATE INDEX "IX_CourseStudent_StudentsStudentId" ON "CourseStudent"
("StudentsStudentId");

Alice Jones attends the following 1 course(s):
  C# 10 and .NET 6
Bob Smith attends the following 2 course(s):
  C# 10 and .NET 6
  Web Development
Cecilia Ramirez attends the following 2 course(s):
  C# 10 and .NET 6
  Python for Beginners
```

Обратите внимание на следующие моменты:

- столбец `Title` (заголовок) является `NOT NULL` (не равен нулю), так как модель была дополнена с помощью `[Required]`;
- столбец `LastName` является `NOT NULL`, так как в модели использовался `IsRequired()`;
- была создана промежуточная таблица `CourseStudent` для хранения информации о том, какие студенты посещают те или иные курсы.

13. Используйте Visual Studio Server Explorer или SQLiteStudio для подключения к базе данных Academy и просмотра таблиц (рис. 10.6).

Рис. 10.6. Просмотр базы данных Academy в SQL Server с помощью Visual Studio 2022 Server Explorer

Миграции

После публикации проекта, использующего базу данных, может возникнуть вероятность, что позже вам понадобится изменить сущностную модель данных и, следовательно, структуру базы данных. На этом этапе вам не следует использовать методы `Ensure`. Вместо этого вам необходимо использовать систему, которая позволит вам постепенно обновлять схему базы данных, сохраняя все существующие данные в базе. Миграции EF Core являются такой системой.

Миграции быстро усложняются, поэтому выходят за рамки данной книги. Вы можете прочитать о них по следующей ссылке: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 10.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какой тип вы бы использовали для свойства, представляющего таблицу, например для свойства `Products` контекста базы данных?
2. Какой тип вы бы применили для свойства, которое представляет отношение «один ко многим», скажем свойство `Products` сущности `Category`?
3. Какое соглашение, касающееся первичных ключей, действует в EF?
4. Когда бы вы могли воспользоваться атрибутом аннотаций в сущностном классе?
5. Почему вы предпочли бы задействовать Fluent API, а не атрибуты аннотаций?
6. Что означает уровень изоляции транзакции `Serializable`?
7. Что возвращает в результате метод `DbContext.SaveChanges()`?
8. В чем разница между жадной и явной загрузками?
9. Как определить сущностный класс EF Core, чтобы он соответствовал следующей таблице?

```
CREATE TABLE Employees(
  EmpId INT IDENTITY,
  FirstName NVARCHAR(40) NOT NULL,
  Salary MONEY
)
```

10. Какие преимущества дает объявление свойств навигации сущностей с пометкой `virtual`?

Упражнение 10.2. Экспорт данных с помощью различных форматов сериализации

В рабочей области/решении `Chapter10` создайте консольное приложение `Exercise02`, которое запрашивает базу данных Northwind для всех категорий и товаров, а затем сериализует данные с помощью как минимум трех форматов сериализации, доступных для .NET. Какой формат сериализации использует наименьшее количество байтов?

Упражнение 10.3. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-10---working-with-data-using-entity-framework-core, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Упражнение 10.4. Изучение базы данных NoSQL

Эта глава была посвящена таким РСУБД, как SQL Server и SQLite. Если вы хотите узнать больше о базах данных NoSQL, таких как Cosmos DB и MongoDB, и о том, как использовать их в EF Core, то я рекомендую следующие ссылки:

- добро пожаловать в Azure Cosmos DB: https://docs.microsoft.com/en-us/azure/cosmos-db/introduction;
- использование баз данных NoSQL в качестве инфраструктуры сохраняемости: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/nosql-database-persistence-infrastructure;
- поставщики баз данных документов для Entity Framework Core: https://github.com/BlueshiftSoftware/EntityFrameworkCore.

Резюме

В этой главе вы узнали, как подключаться к существующей базе данных, выполнять простой запрос LINQ, обрабатывать результаты, использовать отфильтрованные включения. Вы научились добавлять, изменять и удалять данные, а также создавать сущностную модель данных для имеющейся БД, такой как Northwind. Вы также узнали, как определить модель Code First и использовать ее для создания новой базы данных и заполнения ее данными.

В следующей главе вы узнаете, как писать более сложные запросы LINQ, предназначенные для выбора, фильтрации, сортировки, объединения и группировки.

11 Создание запросов и управление данными с помощью LINQ

Данная глава посвящена технологии выражений *LINQ* (Language Integrated Query — «запрос, интегрированный в язык»). LINQ представляет собой набор языковых расширений, которые добавляют возможность работы с последовательностью элементов с их последующей фильтрацией, сортировкой и проецированием в различные структуры данных.

В этой главе:

- написание выражений LINQ;
- работа с множествами с помощью LINQ;
- использование LINQ с EF Core;
- подслащение синтаксиса LINQ с помощью синтаксического сахара;
- использование нескольких потоков и параллельного LINQ;
- создание собственных методов расширения LINQ;
- работа с поставщиком LINQ to XML.

Написание выражений LINQ

Мы уже написали несколько выражений LINQ в главе 10, однако в ней у нас были другие задачи, поэтому я не объяснил подробно работу технологии LINQ. Изучим ее сейчас.

Как работает LINQ

LINQ состоит из нескольких частей; одни из них обязательные, а другие — нет.

- *Методы расширения (обязательно)* — включают в себя `Where`, `OrderBy`, `Select`. Эти методы обеспечивают функциональность LINQ.
- *Поставщики данных LINQ (обязательно)* — включают в себя LINQ to Objects для обработки объектов в памяти, LINQ to Entities для обработки данных,

хранящихся во внешних базах данных и смоделированных с помощью EF Core, и LINQ to XML для обработки данных, хранящихся в формате XML. Эти поставщики выполняют выражения LINQ в соответствии с особенностями различных типов данных.

- *Лямбда-выражения (не обязательно)* — могут использоваться вместо именованных методов для упрощения запросов LINQ, например, для условной логики метода `Where`, предназначенного для фильтрации.
- *Синтаксис обработки запросов LINQ (не обязательно)* — включает в себя такие ключевые слова языка C#, как `from`, `in`, `where`, `orderby`, `descending`, `select`. Они являются псевдонимами для некоторых методов расширения LINQ. Использование этих псевдонимов поможет упростить написание запросов, особенно если у вас уже есть опыт работы с другими языками написания запросов, такими как *Structured Query Language (SQL)*.

При первом знакомстве с LINQ многие разработчики зачастую полагают, что синтаксис обработки запросов и есть LINQ. Однако это одна из необязательных частей LINQ!

Создание выражений LINQ с помощью класса Enumerable

Статический класс `Enumerable` предоставляет методы расширения LINQ, такие как `Where` и `Select`, любому типу, реализующему интерфейс `IEnumerable<T>`. Такой тип также называют *последовательностью*.

Например, массив любого типа реализует `IEnumerable<T>`, где `T` — это тип элемента массива. Это означает, что все массивы поддерживают создание запросов и управление с помощью LINQ.

Все дженерик-коллекции, такие как `List<T>`, `Dictionary<TKey, TValue>`, `Stack<T>` и `Queue<T>`, реализуют `IEnumerable<T>`, поэтому и для этих коллекций можно создавать запросы и управлять ими с помощью LINQ.

Класс `Enumerable` определяет более 50 методов расширения (табл. 11.1).

Таблица 11.1. Методы расширения, определенные классом Enumerable

Метод (-ы)	Описание
`First`, `FirstOrDefault`, `Last`, `LastOrDefault`	Получает первый или последний элемент в последовательности, либо генерирует исключение, либо возвращает значение по умолчанию для типа, например `0` для типа `int`, `null` для ссылочного типа при отсутствии первого или последнего элемента

Метод (-ы)	Описание
`Where`	Возвращает последовательность элементов, соответствующих указанному фильтру
`Single`, `SingleOrDefault`	Возвращает элемент, который соответствует определенному фильтру, или генерирует исключение, или возвращает значение по умолчанию для типа при отсутствии строго одного совпадения
`ElementAt`, `ElementAtOrDefault`	Возвращает элемент в указанной позиции индекса, или генерирует исключение, или возвращает значение по умолчанию для типа при отсутствии в этой позиции элемента. Новинка .NET 6 — перегрузки, которым можно передавать `Index` вместо `int`, что более эффективно при работе с последовательностями `Span<T>`
`Select`, `SelectMany`	Проецирует элементы в другую форму, а именно в другой тип, и делает плоской вложенную иерархию элементов
`OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`	Сортирует элементы по указанному полю или свойству
`Reverse`	Меняет порядок элементов
`GroupBy`, `GroupJoin`, `Join`	Группирует и/или объединяет две последовательности
`Skip`, `SkipWhile`	Пропускает несколько элементов или пропускает элементы, пока выражение `true` (верно)
`Take`, `TakeWhile`	Берет несколько элементов или берет элементы, пока выражение `true` (верно). Новинка .NET 6 — перегрузка `Take`, которой можно передать диапазон, например, `Take(range: 3..^5)`, что означает «взять подмножество, начинающееся на 3 элемента от начала и заканчивающееся на 5 элементов от конца», или вместо `Skip(4)` можно использовать `Take(4...)`
`Aggregate`, `Average`, `Count`, `LongCount`, `Max`, `Min`, `Sum`	Рассчитывает агрегированное значение
`TryGetNonEnumeratedCount`	Метод `Count()` проверяет, реализовано ли свойство `Count` в последовательности, и возвращает его значение, либо перечисляет всю последовательность для подсчета ее элементов. Новинка .NET 6 — этот метод, который проверяет только наличие `Count` (количество) и при его отсутствии возвращает значение `false` и устанавливает параметру `out` значение `0`, чтобы избежать потенциально неэффективной операции
`All`, `Any`, `Contains`	Возвращает значение `true`, если все или какие-то элементы соответствуют фильтру либо последовательность содержит указанный элемент
`Cast`	Приводит элементы к заданному типу. Это полезно для преобразования объектов, не являющихся дженериками, в типы-дженерики в сценариях, где компилятор в противном случае будет выдавать ошибки

Продолжение ⇨

Таблица 11.1 *(продолжение)*

Метод (-ы)	Описание
`OfType`	Удаляет элементы, не соответствующие указанному типу
`Distinct`	Удаляет повторяющиеся элементы
`Except`, `Intersect`, `Union`	Выполняет операции, которые возвращают множества. Сами множества не могут иметь повторяющиеся элементы. Входные данные этих методов могут являться любой последовательностью, поэтому могут иметь дубликаты, однако результат всегда является множеством
`Chunk`	Делит последовательность по количеству символов
`Append`, `Concat`, `Prepend`	Выполняет операции объединения последовательностей
`Zip`	Выполняет операцию сопоставления в зависимости от положения элементов, например, элемент в позиции 1 в первой последовательности соответствует элементу в позиции 1 во второй последовательности. Новинка .NET 6 — операция сопоставления трех последовательностей. Раньше для достижения той же цели вам пришлось бы дважды запускать перегрузку двух последовательностей
`ToArray`, `ToList`, `ToDictionary`, `ToHashSet`, `ToLookup`	Преобразовывает последовательность в массив или коллекцию. Это единственные методы расширения, которые выполняют выражения LINQ
`DistinctBy`, `ExceptBy`, `IntersectBy`, `UnionBy`, `MinBy`, `MaxBy`	Новинка .NET 6 — методы расширения `By`. Они позволяют выполнять сравнение не всего элемента, а его подмножества. Например, вместо удаления дубликатов путем сравнения всего объекта `Person` вы можете удалить дубликаты, сравнив только их `LastName` и `DateOfBirth`

Класс `Enumerable` также имеет некоторые методы, которые не являются методами расширения (табл. 11.2).

Таблица 11.2. Методы класса Enumerable, которые не являются методами расширения

Метод	Описание
`Empty<T>`	Возвращает пустую последовательность указанного типа `T`. Полезен для передачи пустой последовательности методу, которому нужен `IEnumerable<T>`
`Range`	Возвращает последовательность целых чисел из значения `start` с элементами `count`. Например, `Enumerable.Range(start: 5, count: 3)` будет содержать целые числа 5, 6 и 7
`Repeat`	Возвращает последовательность, содержащую один и тот же `element`, повторенный `count` раз. Например, `Enumerable.Repeat(element: "5", count: 3)` будет содержать значения `string` 5, 5 и 5

Принципы отложенного выполнения

LINQ использует *отложенное выполнение*. Важно понимать, что вызов большинства этих методов расширения не приводит к выполнению запроса и получению результатов. Большинство этих методов расширения возвращают выражение LINQ, которое представляет собой *вопрос*, а не *ответ*.

Рассмотрим пример.

1. Откройте редактор кода и создайте рабочую область/решение `Chapter11`.
2. Создайте проект консольного приложения с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter11`;
   3) файл и папка проекта: `LinqWithObjects`.
3. В файле `Program.cs` удалите существующий код и статически импортируйте `Console`.
4. Добавьте операторы для определения последовательности значений для людей `string`, которые работают в офисе:

```
// массив строк, реализующий IEnumerable<string>
string[] names = new[] { "Michael", "Pam", "Jim", "Dwight",
  "Angela", "Kevin", "Toby", "Creed" };

WriteLine("Deferred execution");

// Вопрос: какие имена начинаются с буквы M?
// (используем метод расширения LINQ)
var query1 = names.Where(name => name.EndsWith("m"));

// Вопрос: какие имена заканчиваются на букву M?
// (используем синтаксис написания запросов LINQ)
var query2 = from name in names where name.EndsWith("m") select name;
```

5. Чтобы задать вопрос и получить ответ, то есть выполнить запрос, вам необходимо *материализовать* его, либо вызвав один из методов `To`, например, `ToArray` или `ToLookup`, либо перечислив запрос:

```
// ответ возвращается в виде массива строк, содержащих Pam и Jim
string[] result1 = query1.ToArray();

// ответ возвращается в виде списка строк, содержащих Pam и Jim
List<string> result2 = query2.ToList();
```

```
// ответ возвращается по мере перечисления результата
foreach (string name in query1)
{
  WriteLine(name); // вывод Pam
  names[2] = "Jimmy"; // изменение Jim на Jimmy
  // на второй итерации Jimmy не заканчивается на букву M
}
```

6. Запустите код и проанализируйте результат:

```
Deferred execution
Pam
```

Благодаря отложенному выполнению после вывода первого результата, `Pam`, если исходные значения массива изменятся, к тому времени, когда мы вернемся в цикл, совпадений больше не будет, поскольку `Jim` стал `Jimmy` и не заканчивается на `M`, поэтому выводится только `Pam`.

Прежде чем погружаться в эту тему, сбавим темп и поочередно рассмотрим некоторые распространенные методы расширения LINQ и способы их использования.

Фильтрация элементов с помощью метода Where

Наиболее распространенная причина применения LINQ заключается в фильтрации элементов в последовательности с помощью метода расширения `Where`. Рассмотрим фильтрацию элементов, определив последовательность имен, а затем применив к ней операции LINQ.

1. В файле проекта закомментируйте элемент, позволяющий неявное использование, как показано ниже (выделено жирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <!--<ImplicitUsings>enable</ImplicitUsings>-->
  </PropertyGroup>

</Project>
```

2. В файле `Program.cs` попытайтесь вызвать метод расширения `Where` для массива имен:

```
WriteLine("Writing queries");

var query = names.W
```

3. Когда вы пытаетесь ввести название метода `Where`, обратите внимание, что его нет в списке членов массива `string` в подсказке IntelliSense (рис. 11.1).

Рис. 11.1. IntelliSense с отсутствующим методом расширения Where

Это происходит потому, что `Where` — *метод расширения*. Он не существует в типе массива. Чтобы сделать `Where` доступным, мы должны импортировать пространство имен `System.Linq`. Оно неявно импортируется по умолчанию в новые проекты .NET 6, но мы его отключили.

4. В файле проекта раскомментируйте элемент, разрешающий неявное использование.

5. Повторно введите название метода `Where` и обратите внимание, что подсказка IntelliSense теперь показывает методы расширения, добавленные классом `Enumerable` (рис. 11.2).

Рис. 11.2. IntelliSense теперь показывает методы расширения LINQ Enumerable

6. Теперь, когда вы вводите скобки после имени метода `Where`, IntelliSense сообщает, что для вызова метода `Where` вам необходимо передать его в экземпляре делегата `Func<string, bool>`.
7. Введите следующее выражение для создания нового экземпляра делегата `Func<string, bool>` и обратите внимание, что мы еще не указали имя метода, поскольку определим его позже:

```
var query = names.Where(new Func<string, bool>( ))
```

Делегат `Func<string, bool>` сообщает нам, что для каждой переменной `string`, передаваемой методу, он должен возвращать значение `bool`. Если метод возвращает `true`, то это значит, что нам следует включить `string` в результаты, а если `false` — исключить ее.

Ссылка на именованные методы

Определим метод, включающий только имена, длина которых превышает четыре символа.

1. В конце файла `Program.cs` определите метод, который будет включать только имена, длина которых превышает четыре символа:

```
static bool NameLongerThanFour(string name)
{
  return name.Length > 4;
}
```

2. Над методом `NameLongerThanFour` передайте имя метода в делегат `Func<string, bool>`, а затем переберите элементы запроса, как показано ниже (выделено жирным шрифтом):

```
var query = names.Where(
  new Func<string, bool>(NameLongerThanFour));

foreach (string item in query)
{
  WriteLine(item);
  // names[2] = "Jimmy"; // изменение Jim на Jimmy
  //  на второй итерации Jimmy не заканчивается на букву M
}
```

3. Запустите код и проанализируйте результат, отметив, что перечислены только имена, длина которых превышает четыре символа:

```
Writing queries
Michael
Dwight
```

```
Angela
Kevin
Creed
```

Упрощение кода за счет удаления явного создания экземпляра делегата

Мы можем упростить программный код, удалив явное создание экземпляра делегата `Func<string, bool>`, поскольку компилятор C# создаст этот экземпляр за нас.

1. Чтобы вы могли учиться, видя постепенно улучшающийся код, скопируйте и вставьте запрос.
2. Закомментируйте первый пример:

   ```
   // var query = names.Where(
   //   new Func<string, bool>(NameLongerThanFour));
   ```

3. Измените копию, чтобы удалить явное создание экземпляра делегата:

   ```
   var query = names.Where(NameLongerThanFour);
   ```

4. Запустите код и обратите внимание, что его поведение не изменилось.

Использование лямбда-выражений

Мы можем еще больше упростить код, используя *лямбда-выражения* вместо именованных методов.

Поначалу это может казаться сложным, однако лямбда-выражение, по сути, просто *безымянная функция*. В этих выражениях используется символ `=>` (читается как «идет в») для обозначения возвращаемого значения.

1. Скопируйте и вставьте запрос, закомментируйте второй пример и измените запрос:

   ```
   var query = names.Where(name => name.Length > 4);
   ```

   Обратите внимание: синтаксис лямбда-выражения включает в себя все важнейшие части метода `NameLongerThanFour`, но ничего, кроме них. Для лямбда-выражения требуется лишь определить:

   - имена входящих параметров: `name`;
   - выражение возвращаемого значения: `name.Length > 4`.

Тип входящего параметра `name` предполагается исходя из того, что последовательность содержит значения `string`. При этом для корректной работы метода `Where` тип возвращаемого значения, как это определено делегатом, должен быть `bool`, поэтому выражение после символа `=>` также должно возвращать значение типа `bool`.

Компилятор сделает большую часть работы за нас, поэтому наш код может быть настолько кратким, насколько возможно в принципе.

2. Запустите код и обратите внимание на то, что его поведение не изменилось.

Сортировка элементов

Для сортировки последовательности используются и такие популярные методы расширения, как `OrderBy` и `ThenBy`.

Методы расширения могут быть объединены в цепочку, если предыдущий метод возвращает другую последовательность, то есть тип, реализующий интерфейс `IEnumerable<T>`.

Сортировка по одному свойству с помощью OrderBy

Чтобы изучить сортировку, продолжим работу с текущим проектом.

1. Добавьте вызов метода `OrderBy` в конец уже написанного запроса:

```
var query = names
  .Where(name => name.Length > 4)
  .OrderBy(name => name.Length);
```

Для повышения удобочитаемости форматируйте оператор LINQ таким образом, чтобы вызов каждого метода расширения выполнялся на отдельной строке.

2. Запустите код 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

Нам может потребоваться выполнить сортировку по нескольким свойствам, например, расставить имена одинаковой длины в алфавитном порядке.

1. Добавьте вызов метода `ThenBy` в конце существующего запроса, как показано ниже (выделено жирным шрифтом):

```
var query = names
  .Where(name => name.Length > 4)
  .OrderBy(name => name.Length)
  .ThenBy(name => name);
```

2. Запустите код и обратите внимание на небольшое изменение в порядке сортировки. В группе имен с одинаковым количеством букв имена отсортированы по алфавиту по полному значению строки, поэтому имя `Creed` стоит перед именем `Kevin`, а `Angela` — перед `Dwight`:

```
Creed
Kevin
Angela
Dwight
Michael
```

Объявление запроса с помощью ключевого слова var или заданного типа

При написании выражения LINQ удобно использовать ключевое слово `var` для объявления объекта запроса. Это связано с тем, что тип часто меняется по мере работы над выражением LINQ. Например, наш запрос начинался как `IEnumerable<string>`, а в настоящее время является `IOrderedEnumerable<string>`.

1. Наведите указатель мыши на ключевое слово `var` и обратите внимание, что его типом является `IOrderedEnumerable<string>`.
2. Замените `var` на фактический тип, как показано ниже (выделено жирным шрифтом):

```
IOrderedEnumerable<string> query = names
  .Where(name => name.Length > 4)
  .OrderBy(name => name.Length)
  .ThenBy(name => name);
```

После завершения работы над запросом вы можете изменить объявленный тип с var на фактический тип, чтобы было понятнее. Это легко сделать, поскольку ваш редактор кода может подсказать вам, что это за тип.

Фильтрация по типу

Метод расширения `Where` отлично работает для фильтрации по значению, таких последовательностей как текст и числа. Но что, если последовательность содержит несколько типов и вы хотите отфильтровать по какому-то определенному типу с учетом иерархии наследования?

Представьте, что у вас есть последовательность исключений. Существуют сотни типов исключений, которые образуют сложную иерархию, как частично показано на рис. 11.3.

Рис. 11.3. Частичная иерархия наследования исключений

Рассмотрим фильтрацию по типу.

1. В файле `Program.cs` определите список объектов, наследуемых от исключений:

```
WriteLine("Filtering by type");

List<Exception> exceptions = new()
{
  new ArgumentException(),
  new SystemException(),
  new IndexOutOfRangeException(),
  new InvalidOperationException(),
  new NullReferenceException(),
  new InvalidCastException(),
  new OverflowException(),
  new DivideByZeroException(),
  new ApplicationException()
};
```

2. Добавьте операторы, используя метод расширения `OfType<T>`, чтобы удалить исключения, которые не являются арифметическими, и записать в консоль только арифметические исключения:

```
IEnumerable<ArithmeticException> arithmeticExceptionsQuery =
  exceptions.OfType<ArithmeticException>();
```

```
foreach (ArithmeticException exception in arithmeticExceptionsQuery)
{
  WriteLine(exception);
}
```

3. Запустите код и обратите внимание, что результаты включают только исключения типа `ArithmeticException` или наследуемых от него типов:

```
System.OverflowException: Arithmetic operation resulted in an overflow.
System.DivideByZeroException: Attempted to divide by zero.
```

Работа с множествами с помощью LINQ

Множества — одна из наиболее фундаментальных концепций в математике. *Множество* — это коллекция, состоящая из одного или нескольких уникальных объектов. *Мультимножество*, также известное как *множество с повторяющимися элементами*, представляет собой коллекцию из одного или нескольких объектов, которые могут иметь дубликаты.

Возможно, вы помните, как проходили в школе диаграммы Венна. Типовые операции с множествами — это *пересечение* и *объединение*.

Создадим консольное приложение, которое будет определять три массива значений `string` для группы учащихся, а затем выполнять над ними некоторые типовые операции для множеств и мультимножеств.

1. Откройте редактор кода и создайте консольное приложение `LinqWithSets` в рабочей области/решении `Chapter11`:
   1) в Visual Studio настройте стартовый проект для решения в соответствии с текущим выбором;
   2) в Visual Studio Code выберите `LinqWithSets` в качестве активного проекта OmniSharp.
2. В файле `Program.cs` удалите существующий код и статически импортируйте тип `Console`:

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

3. В конце файла `Program.cs` добавьте следующий метод для вывода в консоль последовательности переменных `string` в виде одной строки, разделенной запятыми, и дополнительным описанием:

```
static void Output(IEnumerable<string> cohort, string description = "")
{
  if (!string.IsNullOrEmpty(description))
  {
```

```
    WriteLine(description);
  }
  Write(" ");
  WriteLine(string.Join(", ", cohort.ToArray()));
  WriteLine();
}
```

4. Над методом `Output` добавьте операторы для определения трех массивов имен, выведите их, а затем выполните над ними различные операции с множествами:

```
string[] cohort1 = new[]
  { "Rachel", "Gareth", "Jonathan", "George" };

string[] cohort2 = new[]
  { "Jack", "Stephen", "Daniel", "Jack", "Jared" };

string[] cohort3 = new[]
  { "Declan", "Jack", "Jack", "Jasmine", "Conor" };

Output(cohort1, "Cohort 1");
Output(cohort2, "Cohort 2");
Output(cohort3, "Cohort 3");

Output(cohort2.Distinct(), "cohort2.Distinct()");
Output(cohort2.DistinctBy(name => name.Substring(0, 2)),
  "cohort2.DistinctBy(name => name.Substring(0, 2)):");
Output(cohort2.Union(cohort3), "cohort2.Union(cohort3)");
Output(cohort2.Concat(cohort3), "cohort2.Concat(cohort3)");
Output(cohort2.Intersect(cohort3), "cohort2.Intersect(cohort3)");
Output(cohort2.Except(cohort3), "cohort2.Except(cohort3)");
Output(cohort1.Zip(cohort2,(c1, c2) => $"{c1} matched with {c2}"),
  "cohort1.Zip(cohort2)");
```

5. Запустите код и проанализируйте результат:

```
Cohort 1
  Rachel, Gareth, Jonathan, George

Cohort 2
  Jack, Stephen, Daniel, Jack, Jared

Cohort 3
  Declan, Jack, Jack, Jasmine, Conor

cohort2.Distinct()
  Jack, Stephen, Daniel, Jared

cohort2.DistinctBy(name => name.Substring(0, 2)):
  Jack, Stephen, Daniel
```

```
cohort2.Union(cohort3)
  Jack, Stephen, Daniel, Jared, Declan, Jasmine, Conor

cohort2.Concat(cohort3)
  Jack, Stephen, Daniel, Jack, Jared, Declan, Jack, Jack, Jasmine, Conor

cohort2.Intersect(cohort3)
  Jack

cohort2.Except(cohort3)
  Stephen, Daniel, Jared

cohort1.Zip(cohort2)
  Rachel matched with Jack, Gareth matched with Stephen, Jonathan matched
with Daniel, George matched with Jack
```

Если в последовательностях имеется неравное количество элементов, то при использовании метода `Zip` некоторые из них останутся без подходящей пары. Элементы, не имеющие пары, такие как `Jared`, не будут включены в результат.

В примере `DistinctBy` вместо удаления дубликатов с помощью сравнения всего имени мы определяем лямбда-селектор ключа для удаления дубликатов путем сравнения первых двух символов, поэтому `Jared` будет удален, поскольку `Jack` уже является именем, начинающимся с `Ja`.

До сих пор мы использовали поставщик LINQ to Objects для работы с объектами в памяти. Далее мы будем использовать поставщик LINQ to Entities для работы с сущностями, хранящимися в базе данных.

Использование LINQ с EF Core

Мы рассмотрели запросы LINQ, которые выполняют фильтрацию и сортировку, но ни один из них не изменяет форму элементов в последовательности. Это называется *проекцией*, поскольку речь идет о проецировании элементов одной формы на другую форму. Изучать проекцию лучше с помощью более сложных типов, поэтому в нашем следующем проекте вместо последовательности `string` (строк) мы будем использовать последовательности сущностей из образца базы данных Northwind.

Я дам инструкции по использованию SQLite, поскольку он является кросс-платформенным, но вы можете использовать SQL Server, если предпочитаете его. Я включил закомментированный код, чтобы вы могли при желании включить SQL Server.

Создание модели EF Core

Для представления базы данных и таблиц, с которыми мы будем работать, нам нужно определить модель *EF Core*. Мы определим ее вручную, чтобы получить полный контроль и предотвратить автоматическое определение связей между таблицами `Categories` и `Products`. Позже вы будете использовать LINQ для соединения двух наборов сущностей.

1. Откройте редактор кода и создайте консольное приложение `LinqWithEFCore` в рабочей области/решении `Chapter11`.
2. В Visual Studio Code выберите `LinqWithEFCore` в качестве активного проекта OmniSharp.
3. В проекте `LinqWithEFCore` добавьте пакетную ссылку на поставщик EF Core для SQLite и/или SQL Server, как показано ниже:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Sqlite"
    Version="6.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="6.0.0" />
</ItemGroup>
```

4. Соберите проект для восстановления пакетов.
5. Скопируйте файл `Northwind4Sqlite.sql` в папку LinqWithEFCore.
6. В командной строке или терминале создайте базу данных Northwind, выполнив следующую команду:

```
sqlite3 Northwind.db -init Northwind4Sqlite.sql
```

7. Вам придется немного подождать, так как этой команде может потребоваться некоторое время для создания структуры базы данных. В конце концов вы увидите командную строку SQLite:

```
   -- Loading resources from Northwind.sql
SQLite version 3.36.0 2021-08-02 15:20:15
Enter ".help" for usage hints.
sqlite>
```

8. Нажмите сочетание клавиш cmd +D в macOS или Ctrl+C в Windows, чтобы выйти из командного режима SQLite.
9. Добавьте в проект три файла классов `Northwind.cs`, `Category.cs` и `Product.cs`.

10. Измените файл класса `Northwind.cs` следующим образом:

```
using Microsoft.EntityFrameworkCore; // DbContext, DbSet<T>

namespace Packt.Shared;

// управление соединением с базой данных
public class Northwind : DbContext
{

  // эти свойства сопоставляются с таблицами в базе данных
  public DbSet<Category>? Categories { get; set; }
  public DbSet<Product>? Products { get; set; }

  protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
  {
    string path = Path.Combine(
      Environment.CurrentDirectory, "Northwind.db");

    optionsBuilder.UseSqlite($"Filename={path}");

  /*
  string connection = "Data Source=.;" +
      "Initial Catalog=Northwind;" +
      "Integrated Security=true;" +
      "MultipleActiveResultSets=true;";

  optionsBuilder.UseSqlServer(connection);
  */
}

  protected override void OnModelCreating(
    ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Product>()
      .Property(product => product.UnitPrice)
      .HasConversion<double>();
  }
}
```

11. Измените файл `Category.cs` следующим образом:

```
using System.ComponentModel.DataAnnotations;

namespace Packt.Shared;

public class Category
{
  public int CategoryId { get; set; }
```

```
  [Required]
  [StringLength(15)]
  public string CategoryName { get; set; } = null!;

  public string? Description { get; set; }
}
```

12. Измените файл класса `Product.cs` следующим образом:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Packt.Shared;

public class Product
{
  public int ProductId { get; set; }

  [Required]
  [StringLength(40)]
  public string ProductName { get; set; } = null!;

  public int? SupplierId { get; set; }
  public int? CategoryId { get; set; }

  [StringLength(20)]
  public string? QuantityPerUnit { get; set; }

  [Column(TypeName = "money")] // требуется для SQL Server
  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; }
}
```

13. Соберите проект и исправьте все ошибки компилятора.

    Если вы используете Visual Studio 2022 для Windows, то скомпилированное приложение выполняется в папке LinqWithEFCore\bin\Debug\net6.0, поэтому не найдет файл базы данных, если мы не укажем, что его всегда следует копировать в выходной каталог.

14. На панели Solution Explorer (Проводник решений) щелкните правой кнопкой мыши на файле `Northwind.db` и выберите параметр Properties (Свойства).

15. В параметре Properties (Свойства) установите для параметра Copy to Output Directory (Копировать в выходной каталог) значение Copy always (Копировать всегда).

Фильтрация и сортировка последовательностей

Добавим операторы для фильтрации и сортировки последовательностей строк из таблиц.

1. В файле `Program.cs` статически импортируйте тип `Console` и пространства имен для работы с EF Core и вашей моделью сущностей с помощью LINQ:

```
using Packt.Shared; // Northwind, Category, Product
using Microsoft.EntityFrameworkCore; // DbSet<T>

using static System.Console;
```

2. В конце файла `Program.cs` добавьте метод для фильтрации и сортировки товаров:

```
static void FilterAndSort()
{
  using (Northwind db = new())
  {
    DbSet<Product> allProducts = db.Products;

    IQueryable<Product> filteredProducts =
      allProducts.Where(product => product.UnitPrice < 10M);

    IOrderedQueryable<Product> sortedAndFilteredProducts =
      filteredProducts.OrderByDescending(product => product.UnitPrice);

    WriteLine("Products that cost less than $10:");
    foreach (Product p in sortedAndFilteredProducts)
    {
      WriteLine("{0}: {1} costs {2:$#,##0.00}",
        p.ProductId, p.ProductName, p.UnitPrice);
    }
    WriteLine();
  }
}
```

Метод `DbSet<T>` реализует интерфейс `IEnumerable<T>`, поэтому LINQ можно использовать для запроса и управления коллекциями сущностей в моделях, созданных для EF Core. (На самом деле я должен был бы сказать `TEntity` вместо `T`, но имя этого общего типа не имеет никакого функционального эффекта. Единственное требование заключается в том, чтобы тип был классом. Имя просто указывает на то, что класс должен быть моделью сущности.)

Возможно, вы также заметили, что последовательности реализуют интерфейсы `IQueryable<T>` (или `IOrderedQueryable<T>` после вызова упорядочивающего метода LINQ) вместо интерфейсов `IEnumerable<T>` и `IOrderedEnumerable<T>`.

Это свидетельствует о том, что мы используем поставщик данных LINQ, который создает запрос в памяти с помощью деревьев выражений. Они представляют собой код в виде древовидной структуры данных и позволяют создавать динамические запросы, что полезно для построения запросов LINQ для внешних поставщиков данных, таких как SQLite.

Выражение LINQ будет преобразовано в другой язык запросов, такой как SQL. Перечисление запроса с помощью оператора `foreach` или вызов метода, такого как `ToArray`, приведет к немедленному исполнению запроса и материализации результатов.

3. После импорта пространства имен в файл `Program.cs` вызовите метод `FilterAndSort`.
4. Запустите код и проанализируйте результат:

```
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
```

Данный запрос распечатывает нужную нам информацию, однако делает это настолько неэффективно, что возвращает все столбцы таблицы `Products` вместо трех необходимых нам столбцов. Это эквивалентно следующему оператору SQL:

```
SELECT * FROM Products;
```

В главе 10 вы научились логировать команды SQL, выполненные для SQLite, чтобы вы могли убедиться в этом сами.

Проецирование последовательностей в новые типы

Прежде чем рассматривать проекцию, нам необходимо вспомнить синтаксис инициализации объекта. Имея определенный класс, вы можете создать экземпляр объекта, используя имя класса, оператор `new()` и фигурные скобки, чтобы установить начальные значения для полей и свойств:

```
public class Person
{
  public string Name { get; set; }
```

```
  public DateTime DateOfBirth { get; set; }
}

Person knownTypeObject = new()
{
  Name = "Boris Johnson",
  DateOfBirth = new(year: 1964, month: 6, day: 19)
};
```

C# 3.0 и более поздние версии позволяют создавать экземпляры *анонимных типов* с помощью ключевого слова `var`:

```
var anonymouslyTypedObject = new
{
  Name = "Boris Johnson",
  DateOfBirth = new DateTime(year: 1964, month: 6, day: 19)
};
```

Хотя мы не указали тип, компилятор может вывести анонимный тип из установки двух свойств `Name` и `DateOfBirth`. Компилятор может вывести типы двух свойств из присвоенных им значений: литеральной строки и нового экземпляра значения даты/времени.

Эта возможность особенно полезна при написании запросов LINQ для проецирования существующего типа в новый, не прибегая к явному определению последнего. Поскольку тип — анонимный, он может работать только с локальными переменными, объявленными с помощью ключевого слова `var`.

Сделаем команду SQL, исполняемую для таблицы базы данных, более эффективной, добавив вызов метода `Select` для проецирования экземпляров класса `Product` в экземпляры нового анонимного типа только с тремя свойствами.

1. В метод `FilterAndSort` добавьте оператор для расширения запроса LINQ для использования метода `Select` и возврата лишь необходимых нам трех свойств (столбцов таблицы), и измените оператор `foreach`, чтобы использовать ключевое слово `var` и проекционное выражение LINQ, как показано ниже (выделено жирным шрифтом):

```
IOrderedQueryable<Product> sortedAndFilteredProducts =
  filteredProducts.OrderByDescending(product => product.UnitPrice);

var projectedProducts = sortedAndFilteredProducts
  .Select(product => new // анонимный тип
  {
    product.ProductId,
    product.ProductName,
    product.UnitPrice
  });

WriteLine("Products that cost less than $10:");
foreach (var p in projectedProducts)
{
```

2. Наведите указатель мыши на ключевое слово `new` в вызове метода `Select` и ключевое слово `var` в операторе `foreach` и обратите внимание, что это анонимный тип (рис. 11.4).

Рис. 11.4. Анонимный тип, используемый при проецировании LINQ

3. Запустите код и убедитесь в том, что его вывод не изменился.

Объединение и группировка последовательностей

Существует два метода расширения для объединения и группировки:

- `Join` принимает четыре параметра: последовательность, с которой требуется объединиться; свойство или свойства *левой* последовательности, по которым нужно найти соответствие; свойство или свойства *правой* последовательности, по которым нужно найти соответствие; и проекцию;
- `GroupJoin` принимает те же параметры, только сочетает совпадения в групповой объект, где `Key` используется для совпавшего значения, а интерфейс `IEnumerable<T>` — для нескольких совпадений.

Объединение последовательностей

Рассмотрим эти методы на примере работы с двумя таблицами: `Categories` и `Products`.

1. В конце файла `Program.cs` создайте метод для выбора категорий и товаров, объедините их и выведите:

```
static void JoinCategoriesAndProducts()
{
  using (Northwind db = new())
```

```
    {
      // объедините продукты каждой категории, чтобы в результате
      // получить 77 сопоставлений
      var queryJoin = db.Categories.Join(
        inner: db.Products,
        outerKeySelector: category => category.CategoryId,
        innerKeySelector: product => product.CategoryId,
        resultSelector: (c, p) =>
          new { c.CategoryName, p.ProductName, p.ProductId });

      foreach (var item in queryJoin)
      {
        WriteLine("{0}: {1} is in {2}.",
          arg0: item.ProductId,
          arg1: item.ProductName,
          arg2: item.CategoryName);
      }
    }
  }
```

Объединяются две последовательности: *внешняя* и *внутренняя*. В предыдущем примере `categories` — внешняя последовательность, а `products` — внутренняя.

2. В начало файла `Program.cs` закомментируйте вызов метода `FilterAndSort` и вызовите метод `JoinCategoriesAndProducts`.
3. Запустите код и проанализируйте результат. Обратите внимание: каждый из 77 товаров выводится на отдельной строке (вывод отредактирован, чтобы включить только первые десять элементов):

```
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.
...
```

4. В конце существующего запроса вызовите метод `OrderBy` для сортировки по `CategoryName`:

```
.OrderBy(cp => cp.CategoryName);
```

5. Запустите код и проанализируйте результат. Обратите внимание, что для каждого из 77 товаров имеется отдельная строка вывода и в результатах

сначала отображаются все товары в категории `Beverages`, затем в категории `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.
...
```

Группировка и объединение последовательностей

1. В конце файла `Program.cs` создайте метод для группировки и объединения, отобразите сначала имя группы, а затем все элементы в каждой группе:

```
static void GroupJoinCategoriesAndProducts()
{
  using (Northwind db = new())
  {
    // сгруппируйте все товары в зависимости от их категории,
    // чтобы в результате получить восемь сопоставлений
    var queryGroup = db.Categories.AsEnumerable().GroupJoin(
      inner: db.Products,
      outerKeySelector: category => category.CategoryId,
      innerKeySelector: product => product.CategoryId,
      resultSelector: (c, matchingProducts) => new
      {
        c.CategoryName,
        Products = matchingProducts.OrderBy(p => p.ProductName)
      });

    foreach (var category in queryGroup)
    {
      WriteLine("{0} has {1} products.",
        arg0: category.CategoryName,
        arg1: category.Products.Count());

      foreach (var product in category.Products)
      {
```

```
            WriteLine($" {product.ProductName}");
          }
        }
      }
    }
```

2. Если бы мы не вызвали метод `AsEnumerable`, то возникло бы исключение времени выполнения:

```
Unhandled exception. System.ArgumentException: Argument type 'System.
Linq.IOrderedQueryable`1[Packt.Shared.Product]' does not match the
corresponding member type 'System.Linq.IOrderedEnumerable`1[Packt.Shared.
Product]' (Parameter 'arguments[1]')
```

   Это связано с тем, что не все методы расширения LINQ можно преобразовать из деревьев выражений в какой-то другой синтаксис запроса, такой как SQL. В подобных случаях мы можем преобразовать интерфейс `IQueryable<T>` в `Ienumerable<T>`, вызвав метод `AsEnumerable`, при котором в обработке запросов используется LINQ to EF Core только для передачи данных в приложение, а затем применяется LINQ to Objects для выполнения сложной обработки в памяти. Часто этот способ менее эффективен.

3. В начале файла `Program.cs` закомментируйте предыдущий вызов метода и вызовите метод `GroupJoinCategoriesAndProducts`.

4. Запустите код, проанализируйте результат. Обратите внимание, что все товары в каждой категории отсортированы по имени, как и было задано в запросе (вывод представлен частично):

```
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`. Напишем код, чтобы изучить, как некоторые из этих методов собирают информацию из таблицы `Products`.

1. В конце файла `Program.cs` создайте метод, показывающий применение методов расширения для агрегирования:

```
static void AggregateProducts()
{
  using (Northwind db = new())
  {
    WriteLine("{0,-25} {1,10}",
      arg0: "Product count:",
      arg1: db.Products.Count());

    WriteLine("{0,-25} {1,10:$#,##0.00}",
      arg0: "Highest product price:",
      arg1: db.Products.Max(p => p.UnitPrice));

    WriteLine("{0,-25} {1,10:N0}",
      arg0: "Sum of units in stock:",
      arg1: db.Products.Sum(p => p.UnitsInStock));

    WriteLine("{0,-25} {1,10:N0}",
      arg0: "Sum of units on order:",
      arg1: db.Products.Sum(p => p.UnitsOnOrder));

    WriteLine("{0,-25} {1,10:$#,##0.00}",
      arg0: "Average unit price:",
      arg1: db.Products.Average(p => p.UnitPrice));

    WriteLine("{0,-25} {1,10:$#,##0.00}",
      arg0: "Value of units in stock:",
      arg1: db.Products
        .Sum(p => p.UnitPrice * p.UnitsInStock));
  }
}
```

2. В начале файла `Program.cs` закомментируйте предыдущий метод и вызовите метод `AggregateProducts`.
3. Запустите код и проанализируйте результат:

```
Product count:                    77
Highest product price:       $263.50
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 LINQ с помощью синтаксического сахара

В версии языка C# 3.0 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*.

Рассмотрим следующий массив значений `string`:

```
string[] names = new[] { "Michael", "Pam", "Jim", "Dwight",
  "Angela", "Kevin", "Toby", "Creed" };
```

Для фильтрации и сортировки по имени вы можете использовать методы расширения и лямбда-выражения:

```
var query = names
  .Where(name => name.Length > 4)
  .OrderBy(name => name.Length)
  .ThenBy(name => name);
```

Или можете достичь тех же результатов, используя понятный синтаксис запросов:

```
var query = from name in names
  where name.Length > 4
  orderby name.Length, name
  select name;
```

Компилятор выполнит преобразование понятного синтаксиса запросов в эквивалентные 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, поскольку, если вы не вызовете `Select`, неявным образом будет выбран элемент целиком.

Не все методы расширения имеют эквивалентное ключевое слово на языке C#. В качестве примера таких методов можно привести `Skip` и `Take`, которые обычно используются для постраничного просмотра большого объема данных.

Запрос, использующий `Skip` и `Take`, не может быть написан только с помощью понятного синтаксиса запросов, поэтому мы можем написать запрос, используя исключительно методы расширения:

```
var query = names
  .Where(name => name.Length > 4)
  .Skip(80)
  .Take(10);
```

В качестве альтернативы можете заключить понятный синтаксис запроса в круглые скобки и перейти к использованию методов расширения:

```
var query = (from name in names
  where name.Length > 4
  select name)
  .Skip(80)
  .Take(10);
```

Вам следует изучить не только методы расширения и лямбда-выражения, но и понятный синтаксис запросов для написания запросов LINQ, поскольку, скорее всего, вам придется сопровождать код, в котором применяется все вышеперечисленное.

Использование нескольких потоков и параллельного LINQ

По умолчанию для выполнения одного запроса LINQ применяется только один поток. *Параллельный LINQ (parallel LINQ, PLINQ)* позволяет 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.

Разработка приложения с помощью нескольких потоков

Чтобы понять информацию данного подраздела, начнем с написания кода, который использует только один поток для вычисления чисел Фибоначчи для 45 целых чисел. Измерять изменения в производительности мы будем с помощью типа `StopWatch`.

Для наблюдения за центральным процессором и использованием его ядра мы используем инструменты операционной системы. Это упражнение будет малоэффективно, если у вас нет нескольких процессоров или хотя бы нескольких ядер!

1. Откройте редактор кода и создайте консольное приложение `LinqInParallel` в рабочей области/решении `Chapter11`.
2. В Visual Studio Code выберите `LinqInParallel` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` удалите существующие операторы и затем импортируйте пространство имен `System.Diagnostics`, чтобы мы могли применить тип `StopWatch`, а также статически импортируйте тип `System.Console`.

4. Добавьте следующие операторы, чтобы создать секундомер, записывающий временные интервалы, запустить таймер по нажатию клавиши, создать 45 целых чисел, вычислить последнее число Фибоначчи для каждого из них, остановить таймер и отобразить количество прошедших миллисекунд:

```
Stopwatch watch = new();
Write("Press ENTER to start. ");
ReadLine();
watch.Start();

int max = 45;

IEnumerable<int> numbers = Enumerable.Range(start: 1, count: max);

WriteLine($"Calculating Fibonacci sequence up to {max}. Please wait...");

int[] fibonacciNumbers = numbers
  .Select(number => Fibonacci(number)).ToArray();

watch.Stop();
WriteLine("{0:#,##0} elapsed milliseconds.",
  arg0: watch.ElapsedMilliseconds);

Write("Results:");
foreach (int number in fibonacciNumbers)
{
  Write($" {number}");
}

static int Fibonacci(int term) =>
  term switch
  {
    1 => 0,
    2 => 1,
    _ => Fibonacci(term - 1) + Fibonacci(term - 2)
  };
```

5. Запустите код, но пока *не нажимайте* клавишу Enter для запуска секундомера, поскольку нам нужно убедиться, что инструмент мониторинга показывает активность процессора.

Для Windows

1. Если вы используете Windows, то щелкните правой кнопкой мыши на кнопке Start (Пуск) или нажмите сочетание клавиш Ctrl+Alt+Del, а затем выберите пункт Task Manager (Диспетчер задач).
2. В нижней части окна Task Manager (Диспетчер задач) нажмите кнопку More details (Подробнее).
3. В верхней части окна Task Manager (Диспетчер задач) перейдите на вкладку Performance (Производительность).

4. Щелкните правой кнопкой мыши на графике CPU Utilization (Использование ЦП) и выберите пункты контекстного меню Change graph to ▸ Logical processors (Изменить график ▸ Логические процессоры).

Для macOS

1. Если вы используете macOS, то запустите приложение Activity Monitor (Монитор активности).
2. Выберите команду меню View ▸ Update Frequency ▸ Very often (1 sec) (Просмотр ▸ Частота обновления ▸ Очень часто (1 с)).
3. Чтобы увидеть графики центрального процессора, выберите команду меню Window ▸ CPU History (Окно ▸ История ЦП).

Для всех операционных систем

1. Переставьте инструмент мониторинга и редактор кода так, чтобы они находились рядом друг с другом.
2. Дождитесь стабилизации графиков центрального процессора и нажмите клавишу Enter, чтобы запустить секундомер и обработать запрос. В результате на экране отобразится количество прошедших миллисекунд:

```
Press ENTER to start.
Calculating Fibonacci sequence up to 45. Please wait...
17,624 elapsed milliseconds.
Results: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169
63245986 102334155 165580141 267914296 433494437 701408733
```

   Инструмент мониторинга, вероятно, покажет, что по большей части были использованы только один или два центральных процессора, чередующиеся во времени. Другие процессоры могут выполнять одновременно фоновые задачи, такие как очистка памяти от ненужных данных, поэтому графики этих процессоров тоже не будут плоскими. Но можно с уверенностью сказать, что работа не будет поровну распределена между всеми возможными центральными процессорами. Обратите также внимание, что некоторые логические процессоры достигают предела возможностей при работе на 100 %.

3. В файле `Program.cs` измените запрос так, чтобы он вызывал метод расширения `AsParallel` и сортировал полученную последовательность, поскольку при параллельной обработке результаты могут стать неупорядоченными:

```
int[] fibonacciNumbers = numbers.AsParallel()
  .Select(number => Fibonacci(number))
  .OrderBy(number => number)
  .ToArray();
```

Никогда не вызывайте AsParallel в конце запроса. Это ничего не дает. Вы должны выполнить хотя бы одну операцию после вызова AsParallel, чтобы эта операция была распараллелена. В .NET 6 появился анализатор кода, который предупредит о подобном неправильном использовании.

4. Запустите код, дождитесь нормализации графиков CPU и нажмите клавишу Enter, чтобы запустить секундомер и обработать запрос. В этот раз приложение должно справиться с задачей быстрее (впрочем, ускорение может быть не таким большим, как вам хотелось бы: управление несколькими потоками тоже требует усилий!):

```
Press ENTER to start.
Calculating Fibonacci sequence up to 45. Please wait...
9,028 elapsed milliseconds.
Results: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169
63245986 102334155 165580141 267914296 433494437 701408733
```

5. Инструмент мониторинга должен показать, что все процессоры использовались одинаково для выполнения запроса LINQ, и отметить, что ни один из логических процессоров не достигает максимума в 100 %, поскольку работа распределена более равномерно.

Принципы управления несколькими потоками вы более подробно изучите в главе 12.

Создание собственных методов расширения LINQ

Из главы 6 вы узнали, как создавать пользовательские методы расширения. Для создания методов расширения LINQ вам необходимо 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 качестве примера мы улучшим метод расширения `Average`. Любой школьник скажет вам, что *среднее значение* (average) может означать одно из трех понятий:

- *среднее арифметическое* — сумма чисел, деленная на их количество;
- *мода* — наиболее часто встречающееся число;
- *медиана* — число в середине упорядоченной последовательности чисел.

Метод расширения `Average` от Microsoft позволяет вычислить *среднее арифметическое*. Мы можем определить собственные методы расширения для вычисления моды и медианы.

1. В проект `LinqWithEFCore` добавьте файл `MyLinqExtensions.cs`.
2. Отредактируйте класс, чтобы его код выглядел так:

```
namespace System.Linq; // расширенное пространство имен от Microsoft

public static class MyLinqExtensions
{

  // цепной метод расширения LINQ
  public static IEnumerable<T> ProcessSequence<T>(
    this IEnumerable<T> sequence)
  {
    // здесь можно выполнить процесс обработки
    return sequence;
  }

  public static IQueryable<T> ProcessSequence<T>(
    this IQueryable<T> sequence)
  {
    // здесь можно выполнить процесс обработки
    return sequence;
  }

  // скалярный метод 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.OrderByDescending(
        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.OrderByDescending(
        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();
    }
  }
```

Если бы этот класс находился в отдельной библиотеке классов, то использовать ваши методы расширения LINQ можно было бы, лишь потребовав сослаться на сборку библиотеки класса, поскольку пространство имен `System.Linq` уже неявно импортировано.

Внимание! Все вышеперечисленные методы расширения, кроме одного, нельзя использовать с последовательностями IQueryable, подобными тем, которые используются в LINQ to SQLite или LINQ to SQL Server, поскольку мы не реализовали способ перевода нашего кода в базовый язык запросов, такой как SQL.

Использование цепного метода расширения

Сначала попробуем соединить метод `ProcessSequence` с другими методами расширения.

1. В файле `Program.cs` в методе `FilterAndSort` измените запрос LINQ для `Products`, чтобы вызвать ваш пользовательский метод цепного расширения, как показано ниже:

```
DbSet<Product>? allProducts = db.Products;

if (allProducts is null)
{
  WriteLine("No products found.");
  return;
}

IQueryable<Product> processedProducts = allProducts.ProcessSequence();

IQueryable<Product> filteredProducts = processedProducts
  .Where(product => product.UnitPrice < 10M);
```

2. В файле `Program.cs` раскомментируйте метод `FilterAndSort` и закомментируйте все вызовы других методов.
3. Запустите код и обратите внимание, что увидите тот же самый программный вывод, что и раньше, поскольку ваш метод не изменяет последовательность. Но зато теперь вы знаете, как расширять выражение LINQ с помощью созданной вами функциональности.

Использование методов Mode и Median

Теперь попробуем использовать методы `Mode` и `Median` для расчета других видов среднего значения.

1. В конце файла `Program.cs` добавьте метод для вывода среднего арифметического, медианы и моды для `UnitsInStock` и `UnitPrice` для товаров, используя собственные методы расширения и встроенный метод расширения `Average`:

```
static void CustomExtensionMethods()
{
  using (Northwind db = new())
  {
    WriteLine("Mean units in stock: {0:N0}",
      db.Products.Average(p => p.UnitsInStock));

    WriteLine("Mean unit price: {0:$#,##0.00}",
      db.Products.Average(p => p.UnitPrice));
```

```
    WriteLine("Median units in stock: {0:N0}",
      db.Products.Median(p => p.UnitsInStock));

    WriteLine("Median unit price: {0:$#,##0.00}",
      db.Products.Median(p => p.UnitPrice));

    WriteLine("Mode units in stock: {0:N0}",
      db.Products.Mode(p => p.UnitsInStock));

    WriteLine("Mode unit price: {0:$#,##0.00}",
      db.Products.Mode(p => p.UnitPrice));
  }
}
```

2. В файле `Program.cs` закомментируйте все предыдущие вызовы методов и вызовите метод `CustomExtensionMethods`.
3. Запустите код и проанализируйте результат:

```
Mean units in stock: 41
Mean unit price: $28.87
Median units in stock: 26
Median unit price: $19.50
Mode units in stock: 0
Mode unit price: $18.00
```

Итого в результате в списке четыре товара с ценой 18 долларов за единицу и пять товаров, количество которых составляет 0 штук.

Работа с LINQ to XML

LINQ to XML — это поставщик данных, позволяющий создавать запросы и управлять XML.

Генерация XML с помощью LINQ to XML

Создадим метод для преобразования таблицы `Products` в XML.

1. В проекте `LinqWithEFCore` в начале файла `Program.cs` импортируйте пространство имен `System.Xml.Linq`.
2. В конце файла `Program.cs` создайте метод для вывода товаров в формате XML:

```
static void OutputProductsAsXml()
{
  using (Northwind db = new())
  {
    Product[] productsArray = db.Products.ToArray();
```

```
    XElement xml = new("products",
    from p in productsArray
    select new XElement("product",
      new XAttribute("id", p.ProductId),
      new XAttribute("price", p.UnitPrice),
     new XElement("name", p.ProductName)));

    WriteLine(xml.ToString());
  }
}
```

3. В файле `Program.cs` закомментируйте предыдущий вызов метода и вызовите метод `OutputProductsAsXml`.
4. Запустите код и проанализируйте результат. Обратите внимание, что структура сгенерированного кода XML соответствует элементам и атрибутам, декларативно описанным с помощью оператора LINQ to XML в предыдущем листинге:

```
<products>
  <product id="1" price="18">
    <name>Chai</name>
  </product>
  <product id="2" price="19">
    <name>Chang</name>
  </product>
...
```

Чтение XML с помощью LINQ to XML

Возможно, вы захотите использовать LINQ to XML, чтобы создавать простые запросы к файлам XML или обрабатывать их.

1. В проект `LinqWithEFCore` добавьте файл `settings.xml`.
2. Измените его содержимое следующим 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>
```

Если вы используете Visual Studio 2022 для Windows, то скомпилированное приложение будет выполняться в папке LinqWithEFCore\bin\Debug\net6.0, поэтому не найдет файл `settings.xml`, если мы не укажем, что он всегда должен быть скопирован в выходной каталог.

3. На панели Solution Explorer (Проводник решений) щелкните правой кнопкой мыши на файле `settings.xml` и выберите параметр Properties (Свойства).
4. В параметре Properties (Свойства) установите для параметра Copy to Output Directory (Копировать в выходной каталог) значение Copy always (Копировать всегда).
5. В конце файла `Program.cs` создайте метод для выполнения следующих задач:
   - 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`;
   - перечислить массив для отображения результатов.

```
static void ProcessSettings()
{
  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}");
  }
}
```

6. В файле `Program.cs` закомментируйте предыдущий вызов метода и вызовите метод `ProcessSettings`.
7. Запустите код и проанализируйте результат:

```
color: red
size: large
price: 23.99
```

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 11.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Каковы две обязательные составные части LINQ?
2. Какой метод расширения LINQ вы использовали бы для возврата подмножества свойств из типа?
3. С помощью какого метода расширения LINQ вы бы выполнили фильтрацию последовательности?
4. Каковы пять методов расширения LINQ, выполняющих агрегацию данных?
5. Чем различаются методы расширения `Select` и `SelectMany`?
6. В чем разница между интерфейсами `IEnumerable<T>` и `IQueryable<T>` и как вы переключаетесь между ними?
7. Что представляет последний параметр типа `T` в делегатах-дженериках `Func`, таких как `Func<T1, T2, T>`?
8. В чем преимущество метода расширения LINQ, который заканчивается оператором `OrDefault`?
9. Почему понятный синтаксис запросов необязателен?
10. Каким образом вы можете создать собственные методы расширения LINQ?

Упражнение 11.2. Создание запросов LINQ

В рабочей области/решении `Chapter11` создайте консольное приложение `Exercise02`, которое запрашивает у пользователя название города, а затем перечисляет название компаний — 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, в качестве подсказки пользователю, прежде чем он 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
```

Упражнение 11.3. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-11---querying-and-manipulating-data-using-linq, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В текущей главе вы изучили, как писать запросы LINQ, предназначенные для выбора, проецирования, фильтрации, сортировки, объединения и группировки данных в различных форматах, в том числе XML, — все это задачи, которые вы будете выполнять каждый день.

В следующей главе вы будете использовать тип `Task` для улучшения производительности ваших приложений.

12 Улучшение производительности и масштабируемости с помощью многозадачности

Данная глава посвящена способам одновременного выполнения нескольких действий в целях повышения производительности, масштабируемости и эффективности работы конечных пользователей, работающих с созданными вами приложениями.

В этой главе:

- процессы, потоки и задачи;
- мониторинг производительности и использования ресурсов;
- асинхронное выполнение задач;
- синхронизация доступа к общим ресурсам;
- ключевые слова `async` и `await`.

Процессы, потоки и задачи

Процесс, одним из примеров которого является каждое из созданных нами консольных приложений, имеет ресурсы, такие как память и потоки, выделенные для него.

Поток выполняет написанный вами код, оператор за оператором. По умолчанию каждый процесс имеет только один поток, что может привести к проблемам, если нужно выполнять несколько задач одновременно. Потоки также отвечают за отслеживание информации об аутентифицированном в данный момент пользователе и правилах интернационализации, которые должны соблюдаться для текущего языка и региона.

Windows и большинство других современных операционных систем используют режим *вытесняющей многозадачности*, которая имитирует параллельное выполнение задач. Он делит процессорное время между потоками, выделяя *интервал времени* каждому потоку один за другим. Текущий поток приостанавливается, когда заканчивается его интервал времени. Затем процессор выделяет другому потоку еще один интервал.

Переключаясь с одного потока на другой, операционная система Windows сохраняет контекст потока и перезагружает ранее сохраненный контекст следующего потока в очереди потоков. На это требуются время и ресурсы.

Если вы как разработчик имеете дело со сложными действиями и хотите получить полный контроль над ними, то можете создавать отдельные экземпляры класса `Thread` и управлять ими. При наличии одного основного потока и нескольких небольших действий, которые можно выполнять в фоновом режиме, вы можете использовать класс `ThreadPool`. Это позволит добавить в очередь экземпляры делегатов, которые указывают на эти действия, реализованные в виде методов, и они будут автоматически распределены по потокам с помощью пула потоков.

В этой главе мы будем использовать тип `Task` для управления потоками на более высоком уровне абстракции.

Потокам, возможно, придется конкурировать и ждать доступа к общим ресурсам, таким как переменные, файлы и объекты баз данных. Для управления ими существуют типы, с работой которых вы ознакомитесь далее в этой главе.

В зависимости от задачи удвоение количества потоков (исполнителей) не уменьшает вдвое количество секунд, которое будет затрачено на выполнение задачи. Фактически это может даже увеличить длительность выполнения задачи.

Никогда не полагайтесь на то, что увеличение количества потоков повысит производительность! Выполните тесты производительности сначала базового кода без реализации нескольких потоков, а затем — кода с несколькими потоками. Выполняйте тесты производительности в промежуточной среде, максимально приближенной к рабочей.

Мониторинг производительности и использования ресурсов

Прежде чем мы сможем улучшить производительность любого кода, мы должны иметь возможность отслеживать его скорость и эффективность работы, чтобы получить базовые данные, на основании которых можем измерять улучшения.

Оценка эффективности типов

Какой тип лучше всего использовать для того или иного сценария? Для ответа на этот вопрос нам необходимо тщательно обдумать, что для нас *лучше всего*. Чтобы принять решение, нужно проверить следующие факторы:

- *функциональность* — представляет ли данный тип необходимые вам функции;
- *объем памяти* — сколько байтов памяти потребляет данный тип;
- *производительность* — насколько быстро работает данный тип;
- *потребности в будущем* — этот фактор зависит от изменений требований и удобства сопровождения.

Будут сценарии, например, при сохранении чисел, когда несколько типов имеют одинаковую функциональность, поэтому, прежде чем выбирать, следует сопоставить расход памяти и производительность.

Для сохранения миллионов чисел наиболее подходящим типом будет тот, который занимает меньше всего места в памяти. Если же необходимо хранить несколько чисел, но выполнять с ними большое количество операций, то лучше выбрать тип, который быстрее всего работает на конкретном ЦПУ.

Ранее упоминалась функция `sizeof()`, предназначенная 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 более хороший способ.

В Интернете и книгах можно прочитать множество рекомендаций, но единственный способ узнать, какой тип лучше подходит для вашей программы, — это сравнить их самостоятельно.

В следующем подразделе вы научитесь писать код для мониторинга действительных требований к памяти и производительности при использовании различных типов.

В настоящее время оптимальным выбором может стать переменная `short`, но, вероятно, будет лучше использовать переменную `int`, хотя она занимает в два раза больше места в памяти. Это связано с тем, что в дальнейшем потребуется хранить более широкий диапазон значений.

Существует важный показатель, о котором разработчики часто забывают: сопровождение. Он демонстрирует, сколько усилий потребуется приложить другому

разработчику, чтобы понять и отредактировать ваш код. Если вы используете неявный выбор типа, не объясняя данное действие с помощью комментария, то это может запутать разработчика, открывшего ваш код, чтобы исправить ошибку или добавить какую-либо функциональность.

Мониторинг производительности и памяти с помощью диагностики

В пространстве имен `System.Diagnostics` реализовано большое количество полезных типов для мониторинга вашего кода. В первую очередь мы рассмотрим тип `Stopwatch`.

1. Откройте редактор кода и создайте рабочую область/решение `Chapter12`.
2. Создайте проект библиотеки классов с такими настройками:
   1) шаблон проекта: Class Library/`classlib`;
   2) файл и папка рабочей области/решения: `Chapter12`;
   3) файл и папка проекта: `MonitoringLib`.
3. Создайте проект консольного приложения с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter12`;
   3) файл и папка проекта: `MonitoringLib`.
4. В программе Visual Studio настройте стартовый проект для решения в соответствии с текущим выбором.
5. В программе Visual Studio Code выберите `MonitoringApp` в качестве активного проекта OmniSharp.
6. В проекте `MonitoringLib` переименуйте файл `Class1.cs` в `Recorder.cs`.
7. В проекте `MonitoringApp` добавьте ссылку на библиотеку `MonitoringLib`, как показано ниже:

```
<ItemGroup>
  <ProjectReference
    Include="..\MonitoringLib\MonitoringLib.csproj" />
</ItemGroup>
```

8. Соберите проект `MonitoringApp`.

Полезные члены типов Stopwatch и Process

Тип `Stopwatch` содержит несколько полезных членов (табл. 12.1).

Таблица 12.1. Полезные члены типа Stopwatch

Член	Описание
Метод `Restart`	Обнуляет затраченное время и начинает измерение затраченного времени
Метод `Stop`	Останавливает измерение затраченного времени
Свойство `Elapsed`	Затраченное время, сохраненное в формате `TimeSpan` (например, часы:минуты:секунды)
Свойство `ElapsedMilliseconds`	Затраченное время в миллисекундах, хранящееся как значение `Int64`

Тип `Process` также содержит несколько полезных членов (табл. 12.2).

Таблица 12.2. Полезные члены типа Process

Член	Описание
`VirtualMemorySize64`	Отображает объем виртуальной памяти в байтах, выделенной для процесса
`WorkingSet64`	Отображает объем физической памяти в байтах, выделенной для процесса

Реализация класса Recorder

Мы создадим класс `Recorder`, который позволяет легко отслеживать затраты времени и памяти. Для реализации класса `Recorder` мы будем использовать классы `Stopwatch` и `Process`.

1. В файле `Recorder.cs` измените его содержимое, чтобы задействовать экземпляр класса `Stopwatch` для записи времени и текущий экземпляр класса `Process` для записи использованной памяти:

```
using System.Diagnostics; // Stopwatch

using static System.Console;
using static System.Diagnostics.Process; // GetCurrentProcess()

namespace Packt.Shared;

public static class Recorder
{
  private static Stopwatch timer = new();
```

```
  private static long bytesPhysicalBefore = 0;
  private static long bytesVirtualBefore = 0;

  public static void Start()
  {
    // инициируем очистку памяти сборщиками мусора
    // очищается память, на которую никто не ссылается
    GC.Collect();
    GC.WaitForPendingFinalizers();
    GC.Collect();

    // храним текущие затраты физической и виртуальной памяти
    bytesPhysicalBefore = GetCurrentProcess().WorkingSet64;
    bytesVirtualBefore = GetCurrentProcess().VirtualMemorySize64;
    timer.Restart();
  }

  public static void Stop()
  {
    timer.Stop();
    long bytesPhysicalAfter =
      GetCurrentProcess().WorkingSet64;

    long bytesVirtualAfter =
      GetCurrentProcess().VirtualMemorySize64;

    WriteLine("{0:N0} physical bytes used.",
      bytesPhysicalAfter - bytesPhysicalBefore);

    WriteLine("{0:N0} virtual bytes used.",
      bytesVirtualAfter - bytesVirtualBefore);

    WriteLine("{0} time span ellapsed.", timer.Elapsed);

    WriteLine("{0:N0} total milliseconds elapsed.",
      timer.ElapsedMilliseconds);
  }
}
```

В методе `Start` класса `Recorder` используется тип `GC` (garbage collector — «сборщик мусора»), позволяющий нам гарантировать, что вся выделенная в настоящий момент память будет собрана до записи количества использованной памяти. Это усовершенствованный метод, который почти никогда не следует использовать в коде приложения.

2. В файле `Program.cs` добавьте операторы для запуска и остановки класса `Recorder` при генерации массива из 10 000 целых чисел:

```
using Packt.Shared; // Recorder

using static System.Console;
```

```
WriteLine("Processing. Please wait...");
Recorder.Start();

// имитируем процесс использования ресурсов памяти...
int[] largeArrayOfInts = Enumerable.Range(
  start: 1, count: 10_000).ToArray();

// ...требуется некоторое время на выполнение
Thread.Sleep(new Random().Next(5, 10) * 1000);

Recorder.Stop();
```

3. Запустите код и проанализируйте результат:

```
Processing. Please wait...
655,360 physical bytes used.
536,576 virtual bytes used.
00:00:09.0038702 time span ellapsed.
9,003 total milliseconds ellapsed.
```

Помните, что прошедшее время вариабельно и составит от 5 до 10 секунд. Ваши результаты будут другими. Например, при запуске на моем Mac mini M1 использовалось меньше физической памяти, но больше виртуальной:

```
Processing. Please wait...
294,912 physical bytes used.
10,485,760 virtual bytes used.
00:00:06.0074221 time span ellapsed.
6,007 total milliseconds ellapsed.
```

Измерение эффективности обработки строк

Теперь, когда вы научились использовать типы `Stopwatch` и `Process` для мониторинга своего кода, с их помощью мы определим наилучший способ обработки переменных `string`.

1. В файле `Program.cs` закомментируйте предыдущий код, обернув его символами многострочных комментариев `/*` и `*/`.
2. Введите операторы для создания массива из 50 000 переменных типа `int`, а затем конкатенируйте их, используя в качестве разделителей запятые, с помощью классов `string` и `StringBuilder`:

```
int[] numbers = Enumerable.Range(
  start: 1, count: 50_000).ToArray();

WriteLine("Using string with +");
```

```
Recorder.Start();
string s = string.Empty; // i.e. ""
for (int i = 0; i < numbers.Length; i++)
{
  s += numbers[i] + ", ";
}
Recorder.Stop();

WriteLine("Using StringBuilder");
Recorder.Start();
System.Text.StringBuilder builder = new();
for (int i = 0; i < numbers.Length; i++)
{
  builder.Append(numbers[i]);
  builder.Append(", ");
}
Recorder.Stop();
```

3. Запустите код и проанализируйте результат:

```
Using string with +
14,883,072 physical bytes used.
3,609,728 virtual bytes used.
00:00:01.6220879 time span ellapsed.
1,622 total milliseconds ellapsed.
Using StringBuilder
12,288 physical bytes used.
0 virtual bytes used.
00:00:00.0006038 time span ellapsed.
0 total milliseconds ellapsed.
```

Исходя из результатов, мы можем сделать следующие выводы:

- класс `string` вместе с операцией `+` использовал около 14 Мбайт физической памяти, 1,5 Мбайт виртуальной и занял 1,5 с;
- класс `StringBuilder` использовал 12 Кбайт физической памяти, ноль виртуальной и занял меньше 1 мс.

В этом сценарии при конкатенации текста класс `StringBuilder` работает более чем в 1000 раз быстрее и приблизительно в 10 000 раз эффективнее по затратам ресурсов памяти! Это происходит потому, что конкатенация строк создает новую строку каждый раз, когда вы ее используете, поскольку значения строк неизменяемы, поэтому их можно безопасно объединять для повторного использования. Класс `StringBuilder` создает один буфер, пока добавляет больше символов.

Избегайте использования метода String.Concat или операции + внутри цикла. Вместо этого применяйте класс StringBuilder.

Теперь, когда вы научились измерять производительность и эффективность ресурсов вашего кода с помощью типов, встроенных в .NET, изучим пакет NuGet, который позволяет проводить более сложные измерения производительности.

Мониторинг производительности и памяти с помощью Benchmark.NET

Существует популярный .NET-пакет NuGet для тестирования производительности, результаты которого Microsoft приводит в постах блога об улучшении производительности. Поэтому разработчикам .NET полезно знать, как работает пакет, и использовать его для тестирования быстродействия. Посмотрим, как с его помощью мы можем сравнить производительность конкатенации строк и класса `StringBuilder`.

1. Откройте редактор кода и создайте консольное приложение `Benchmarking` в рабочей области/решении `Chapter12`.
2. В программе Visual Studio Code выберите `Benchmarking` в качестве активного проекта OmniSharp.
3. Добавьте ссылку на пакет `Benchmark.NET`. Помните, что вы можете установить более актуальную версию и использовать ее вместо моей:

```
<ItemGroup>
  <PackageReference Include="BenchmarkDotNet" Version="0.13.1" />
</ItemGroup>
```

4. Соберите проект для восстановления пакетов.
5. В файле `Program.cs` удалите существующие операторы, а затем импортируйте пространство имен для запуска теста производительности:

```
using BenchmarkDotNet.Running;
```

6. Добавьте новый файл класса `StringBenchmarks.cs`.
7. В файле `StringBenchmarks.cs` добавьте операторы для определения класса с методами для каждого теста производительности, который хотите запустить. В данном случае это два метода, которые объединяют 20 чисел, разделенных запятыми, используя либо конкатенацию строк, либо класс `StringBuilder`:

```
using BenchmarkDotNet.Attributes; // [Benchmark]

public class StringBenchmarks
{
  int[] numbers;
```

```
  public StringBenchmarks()
  {
    numbers = Enumerable.Range(
      start: 1, count: 20).ToArray();
  }

  [Benchmark(Baseline = true)]
  public string StringConcatenationTest()
  {
    string s = string.Empty; // например, ""
    for (int i = 0; i < numbers.Length; i++)
    {
      s += numbers[i] + ", ";
    }
    return s;
  }

  [Benchmark]
  public string StringBuilderTest()
  {
    System.Text.StringBuilder builder = new();
    for (int i = 0; i < numbers.Length; i++)
    {
      builder.Append(numbers[i]);
      builder.Append(", ");
    }
    return builder.ToString();
  }
}
```

8. В файл `Program.cs` добавьте оператор для запуска тестов производительности:

```
BenchmarkRunner.Run<StringBenchmarks>();
```

9. В программе Visual Studio 2022 на панели инструментов задайте параметру Solution Configurations (Конфигурации решения) значение Release.
10. В программе Visual Studio Code в терминале выполните команду `dotnet run --configuration Release`.
11. Запустите консольное приложение и обратите внимание на результаты, включая файлы отчетов, и — самое главное — на сводную таблицу, которая показывает, что конкатенация строк заняла в среднем 412 990 нс, а работа класса `StringBuilder` — 275 082 нс, как показано в следующем фрагменте кода и на рис. 12.1.

```
// ***** BenchmarkRunner: Finish *****

// * Export *
  BenchmarkDotNet.Artifacts\results\StringBenchmarks-report.csv
```

```
  BenchmarkDotNet.Artifacts\results\StringBenchmarks-report-github.md
  BenchmarkDotNet.Artifacts\results\StringBenchmarks-report.html

// * Detailed results *
StringBenchmarks.StringConcatenationTest: DefaultJob
Runtime = .NET 6.0.0 (6.0.21.37719), X64 RyuJIT; GC = Concurrent
Workstation
Mean = 412.990 ns, StdErr = 2.353 ns (0.57%), N = 46, StdDev = 15.957 ns
Min = 373.636 ns, Q1 = 413.341 ns, Median = 417.665 ns, Q3 = 420.775 ns,
Max = 434.504 ns
IQR = 7.433 ns, LowerFence = 402.191 ns, UpperFence = 431.925 ns
ConfidenceInterval = [404.708 ns; 421.273 ns] (CI 99.9%), Margin = 8.282
ns (2.01% of Mean)
Skewness = -1.51, Kurtosis = 4.09, MValue = 2
-------------------- Histogram --------------------
[370.520 ns ; 382.211 ns) | @@@@@@
[382.211 ns ; 394.583 ns) | @
[394.583 ns ; 411.300 ns) | @@
[411.300 ns ; 422.990 ns) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
[422.990 ns ; 436.095 ns) | @@@@@
---------------------------------------------------

StringBenchmarks.StringBuilderTest: DefaultJob
Runtime = .NET 6.0.0 (6.0.21.37719), X64 RyuJIT; GC = Concurrent
Workstation
Mean = 275.082 ns, StdErr = 0.558 ns (0.20%), N = 15, StdDev = 2.163 ns
Min = 271.059 ns, Q1 = 274.495 ns, Median = 275.403 ns, Q3 = 276.553 ns,
Max = 278.030 ns
IQR = 2.058 ns, LowerFence = 271.409 ns, UpperFence = 279.639 ns
ConfidenceInterval = [272.770 ns; 277.394 ns] (CI 99.9%), Margin = 2.312
ns (0.84% of Mean)
Skewness = -0.69, Kurtosis = 2.2, MValue = 2
-------------------- Histogram --------------------
[269.908 ns ; 278.682 ns) | @@@@@@@@@@@@@@@
---------------------------------------------------

// * Summary *

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19043.1165 (21H1/May2021Update)
11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores
.NET SDK=6.0.100
  [Host]     : .NET 6.0.0 (6.0.21.37719), X64 RyuJIT
  DefaultJob : .NET 6.0.0 (6.0.21.37719), X64 RyuJIT

|                  Method |     Mean |   Error |           StdDev | Ratio |
RatioSD |
|------------------------ |---------:|--------:|---------:|------:|------
--:|
| StringConcatenationTest | 413.0 ns | 8.28 ns | 15.96 ns | 1.00 |
0.00 |
```

```
| StringBuilderTest | 275.1 ns | 2.31 ns | 2.16 ns | 0.69 |
0.04 |

// * Hints *
Outliers
  StringBenchmarks.StringConcatenationTest: Default -> 7 outliers
were removed, 14 outliers were detected (376.78 ns..391.88 ns, 440.79
ns..506.41 ns)
  StringBenchmarks.StringBuilderTest: Default -> 2 outliers were
detected (274.68 ns, 274.69 ns)

// * Legends *
  Mean : Arithmetic mean of all measurements
  Error : Half of 99.9% confidence interval
  StdDev : Standard deviation of all measurements
  Ratio : Mean of the ratio distribution ([Current]/[Baseline])
  RatioSD : Standard deviation of the ratio distribution ([Current]/
[Baseline])
  1 ns : 1 Nanosecond (0.000000001 sec)

// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:01:13 (73.35 sec), executed benchmarks: 2

Global total time: 00:01:29 (89.71 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

Рис. 12.1. Сводная таблица, показывающая, что работа StringBuilder занимает 69 % времени по сравнению с конкатенацией строк

Раздел `Outliers` особенно интересен, поскольку показывает, что конкатенация строк не только медленнее, чем `StringBuilder`, но и более непостоянна по времени. Ваши результаты, конечно же, будут другими.

Теперь вы увидели два способа измерения производительности. Теперь посмотрим, как запускать задачи асинхронно, чтобы потенциально повысить производительность.

Асинхронное выполнение задач

Чтобы понять, каким образом несколько задач могут выполняться одновременно, мы напишем простое консольное приложение, выполняющее три метода: первый занимает три секунды, второй — две, а третий — одну. Имитировать такой процесс мы можем с помощью класса `Thread`, который передаст текущему потоку инструкцию заснуть на заданное количество миллисекунд.

Синхронное выполнение нескольких действий

Прежде чем запускать задачи одновременно, мы будем запускать их синхронно, то есть одну за другой.

1. Откройте редактор кода и создайте консольное приложение `WorkingWithTasks` в рабочей области/решении `Chapter12`.
2. В программе Visual Studio Code выберите `WorkingWithTasks` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` импортируйте пространство имен для работы с секундомером (пространства имен для работы с потоками и задачами импортируются неявно), а также статически импортируйте `Console`:

```
using System.Diagnostics; // Stopwatch

using static System.Console;
```

4. В конце файла `Program.cs` создайте метод для вывода информации о текущем потоке:

```
static void OutputThreadInfo()
{
  Thread t = Thread.CurrentThread;

  WriteLine(
    "Thread Id: {0}, Priority: {1}, Background: {2}, Name: {3}",
    t.ManagedThreadId, t.Priority,
    t.IsBackground, t.Name ?? "null");
}
```

5. В конце файла `Program.cs` добавьте три метода, имитирующие работу:

```
static void MethodA()
{
  WriteLine("Starting Method A...");
  OutputThreadInfo();
  Thread.Sleep(3000); // имитируем три секунды работы
```

```
  WriteLine("Finished Method A.");
}

static void MethodB()
{
  WriteLine("Starting Method B...");
  OutputThreadInfo();
  Thread.Sleep(2000); // имитируем две секунды работы
  WriteLine("Finished Method B.");
}

static void MethodC()
{
  WriteLine("Starting Method C...");
  OutputThreadInfo();
  Thread.Sleep(1000); // имитируем одну секунду работы
  WriteLine("Finished Method C.");
}
```

6. В начале файла `Program.cs` добавьте следующие операторы, чтобы вызвать метод, который выводит информацию о потоке, определить и запустить секундомер, вызвать три имитационных метода работы и вывести истекшие миллисекунды:

```
OutputThreadInfo();
Stopwatch timer = Stopwatch.StartNew();

WriteLine("Running methods synchronously on one thread.");
MethodA();
MethodB();
MethodC();

WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");
```

7. Запустите код и проанализируйте результат. Обратите внимание: при использовании только одного безымянного потока переднего плана выполнение программы займет немногим более 6 с:

```
Thread Id: 1, Priority: Normal, Background: False, Name: null
Running methods synchronously on one thread.
Starting Method A...
Thread Id: 1, Priority: Normal, Background: False, Name: null
Finished Method A.
Starting Method B...
Thread Id: 1, Priority: Normal, Background: False, Name: null
Finished Method B.
Starting Method C...
Thread Id: 1, Priority: Normal, Background: False, Name: null
Finished Method C.
6,017ms elapsed.
```

Асинхронное выполнение нескольких действий с помощью задач

Класс `Thread` был доступен уже в первой версии .NET и может использоваться для создания новых потоков и управления ими, но работать с ним напрямую сложно.

В версии .NET Framework 4.0 в 2010 году был добавлен класс `Task`, который представляет собой оболочку для потока, упрощающую создание и управление потоками. Управление несколькими потоками, обернутыми в задачи, даст нашему коду возможность выполняться асинхронно (в одно и то же время).

Каждая задача имеет свойства `Status` и `CreationOptions`. Класс `Task` содержит метод `ContinueWith`, который можно настроить с помощью перечисления `TaskContinuationOptions` и которым можно управлять, используя класс `TaskFactory`.

Запуск задач

Мы рассмотрим три способа запуска методов с помощью экземпляров класса `Task`. В репозитории GitHub есть ссылки на статьи, в которых обсуждаются плюсы и минусы этих способов. Каждый из них несколько отличается синтаксисом, но все они определяют `Task` и запускают его.

1. Закомментируйте вызовы трех методов и связанного с ними консольного сообщения. Добавьте операторы для создания и запуска трех задач, по одной для каждого метода, как показано ниже:

```
OutputThreadInfo();
Stopwatch timer = Stopwatch.StartNew();

/*
WriteLine("Running methods synchronously on one thread.");
MethodA();
MethodB();
MethodC();
*/

WriteLine("Running methods asynchronously on multiple threads.");

Task taskA = new(MethodA);
taskA.Start();

Task taskB = Task.Factory.StartNew(MethodB);

Task taskC = Task.Run(MethodC);

WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");
```

2. Запустите код и проанализируйте результат. Обратите внимание: количество прошедших миллисекунд выводится практически сразу, поскольку каждый из трех методов теперь выполняется тремя новыми фоновыми рабочими потоками, выделенными из пула потоков:

```
Thread Id: 1, Priority: Normal, Background: False, Name: null
Running methods asynchronously on multiple threads.
Starting Method A...
Thread Id: 4, Priority: Normal, Background: True, Name: .NET ThreadPool
Worker
Starting Method C...
Thread Id: 7, Priority: Normal, Background: True, Name: .NET ThreadPool
Worker
Starting Method B...
Thread Id: 6, Priority: Normal, Background: True, Name: .NET ThreadPool
Worker
6ms elapsed.
```

Возможно даже, что консольное приложение завершится до того, как одна или несколько задач смогут запуститься и записать результат в консоль!

Ожидание выполнения задач

Иногда нужно 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. 12.3).

Таблица 12.3. Методы ожидания

Метод	Описание
`t.Wait()`	Ожидает завершения задачи `t`
`Task.WaitAny(Task[])`	Ожидает завершения любой задачи в массиве
`Task.WaitAll(Task[])`	Ожидает завершения всех задач в массиве

Использование методов wait в задачах

Рассмотрим, каким образом мы можем использовать перечисленные в табл. 12.3 методы ожидания для решения проблем с нашим консольным приложением.

1. В файле `Program.cs` добавьте операторы сразу после кода создания трех задач и перед выводом истекшего времени, чтобы объединить в массив ссылки на три задачи и передать их методу `WaitAll`:

```
Task[] tasks = { taskA, taskB, taskC };
Task.WaitAll(tasks);
```

2. Запустите код и проанализируйте результат. Обратите внимание, что исходный поток приостанавливается на вызове `WaitAll`, ожидая завершения всех трех задач, прежде чем вывести значение затраченного времени, которое составляет чуть более трех секунд:

```
Id: 1, Priority: Normal, Background: False, Name: null
Running methods asynchronously on multiple threads.
Starting Method A...
Id: 6, Priority: Normal, Background: True, Name: .NET ThreadPool Worker
Starting Method B...
Id: 7, Priority: Normal, Background: True, Name: .NET ThreadPool Worker
Starting Method C...
Id: 4, Priority: Normal, Background: True, Name: .NET ThreadPool Worker
Finished Method C.
Finished Method B.
Finished Method A.
3,013ms elapsed.
```

Три новых потока исполняют свой код одновременно и потенциально могут запускаться в случайном порядке. `MethodC` завершается первым, поскольку длится всего 1 с, затем `MethodB`, который длится 2 с, и, наконец, `MethodA`, длящийся 3 с.

Тем не менее реально используемый центральный процессор существенно влияет на результат. Именно ЦП выделяет временные интервалы под каждый процесс, чтобы позволить им выполнять свои потоки. Вы не можете контролировать запуск методов.

Задачи продолжения

Если все три задачи могут быть выполнены одновременно, то все, что нам нужно сделать, — это дождаться завершения всех их. Однако часто задача зависит от результата выполнения другой задачи. Чтобы справиться с данной проблемой, нам нужно определить *задачи продолжения*.

Мы создадим несколько методов для имитации вызова веб-сервиса, возвращающего денежную сумму, которую затем необходимо использовать 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. На этот раз вместо того, чтобы ждать фиксированное количество времени, мы будем использовать класс `Random` для ожидания завершения каждого вызова метода со случайной длительностью в диапазоне от 2 до 4 с, чтобы имитировать работу.

1. В файле `Program.cs` добавьте следующие два метода, которые имитируют вызов веб-сервиса и хранимой процедуры базы данных:

```
static decimal CallWebService()
{
  WriteLine("Starting call to web service...");
  OutputThreadInfo();
  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...");
  OutputThreadInfo();
  Thread.Sleep((new Random()).Next(2000, 4000));
  WriteLine("Finished call to stored procedure.");
  return $"12 products cost more than {amount:C}.";
}
```

2. Закомментируйте предыдущие три задачи, обернув их код символами многострочного комментария /* */. Оставьте оператор, который выводит в результате количество затраченных миллисекунд.
3. Добавьте операторы перед уже существующим оператором, который выводит значение общего затраченного времени:

```
WriteLine("Passing the result of one task as an input into another.");

Task<string> taskServiceThenSProc = Task.Factory
  .StartNew(CallWebService) // returns Task<decimal>
  .ContinueWith(previousTask => // returns Task<string>
    CallStoredProcedure(previousTask.Result));

WriteLine($"Result: {taskServiceThenSProc.Result}");
```

4. Запустите код и проанализируйте результат:

```
Thread Id: 1, Priority: Normal, Background: False, Name: null
Passing the result of one task as an input into another.
Starting call to web service...
Thread Id: 4, Priority: Normal, Background: True, Name: .NET ThreadPool
Worker
Finished call to web service.
Starting call to stored procedure...
Thread Id: 6, Priority: Normal, Background: True, Name: .NET ThreadPool
Worker
Finished call to stored procedure.
Result: 12 products cost more than £89.99.
5,463ms elapsed.
```

Вы можете увидеть разные потоки, выполняющие вызовы веб-сервиса и хранимой процедуры, как в приведенных выше выходных данных (потоки 4 и 6), или один и тот же поток может быть использован повторно, поскольку он больше не занят.

Вложенные и дочерние задачи

Помимо определения зависимостей между задачами, вы можете определять вложенные и дочерние задачи. *Вложенная задача* создается внутри другой задачи. *Дочерняя* представляет собой вложенную задачу, которая должна быть завершена до того, как завершится ее родительская задача.

Рассмотрим пример работы этих задач.

1. Откройте редактор кода и создайте консольное приложение `NestedAndChildTasks` в рабочей области/решении `Chapter12`.
2. В программе Visual Studio Code выберите вариант `NestedAndChildTasks` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` удалите существующие операторы, статически импортируйте `Console` и затем добавьте два метода, один из которых запускает задачу для запуска второго:

```
static void OuterMethod()
{
  WriteLine("Outer method starting...");
  Task innerTask = Task.Factory.StartNew(InnerMethod);
  WriteLine("Outer method finished.");
}

static void InnerMethod()
{
  WriteLine("Inner method starting...");
  Thread.Sleep(2000);
  WriteLine("Inner method finished.");
}
```

4. Над методами добавьте следующие операторы, чтобы запустить задачу для запуска внешнего метода, и дождитесь ее завершения перед остановкой:

```
Task outerTask = Task.Factory.StartNew(OuterMethod);
outerTask.Wait();
WriteLine("Console app is stopping.");
```

5. Запустите код и проанализируйте результат:

```
Outer method starting...
Inner method starting...
Outer method finished.
Console app is stopping.
```

Обратите внимание: мы ждем завершения внешней задачи, однако внутренняя не должна при этом завершаться. На самом деле внешняя задача может завершиться и консольное приложение может закончить работу еще до того, как запустится внутренняя задача!

Связать две задачи в качестве родительской и дочерней необходимо с помощью специального параметра.

6. Измените существующий код, который определяет внутреннюю задачу, чтобы добавить перечисление `TaskCreationOption` с элементом `AttachedToParent`, как показано ниже:

```
Task innerTask = Task.Factory.StartNew(InnerMethod,
  TaskCreationOptions.AttachedToParent);
```

7. Запустите код и проанализируйте результат. Обратите внимание, что внутренняя задача должна завершиться раньше внешней:

```
Outer method starting...
Inner method starting...
Outer method finished.
Inner method finished.
Console app is stopping.
```

Метод `OuterMethod` может завершиться раньше метода `InnerMethod`, как показано в результате вывода в консоль, но мы все еще будем ждать завершения обоих методов. Это видно по тому, что консольное приложение не закончило работу до завершения как метода `OuterMethod`, так и `InnerMethod`.

Обертывание задач вокруг других объектов

Иногда у вас может быть метод, который вы хотите сделать асинхронным, но возвращаемый результат сам по себе не является задачей. Вы можете обернуть возвращаемое значение в успешно выполненную задачу, вернуть исключение или указать, что задача была отменена, используя один из методов, показанных в табл. 12.4.

Таблица 12.4. Дополнительные методы

Метод	Описание
`FromResult<TResult>(TResult)`	Создает объект `Task<TResult>`, свойством `Result` которого выступает результат, не относящийся к задаче, а свойством `Status` является `RanToCompletion`
`FromException<TResult>(Exception)`	Создает объект `Task<TResult>`, который завершается с указанным исключением
`FromCanceled<TResult> (CancellationToken)`	Создает задачу `Task<TResult>`, которая завершается из-за отмены с указанным маркером отмены

Эти методы будут полезны, если необходимо:

- реализовать интерфейс, который имеет асинхронные методы, но ваша реализация является синхронной. Это часто встречается на сайтах и в сервисах;
- имитировать асинхронные реализации во время модульного тестирования.

В главе 7 мы создали библиотеку классов с функциями для проверки правильности XML-данных, паролей и шестнадцатеричных кодов.

Если бы мы хотели, чтобы эти методы соответствовали интерфейсу, который требует возврата `Task<T>`, то могли бы использовать эти полезные методы:

```
using System.Text.RegularExpressions;

namespace Packt.Shared;

public static class StringExtensions
{
  public static Task<bool> IsValidXmlTagAsync(this string input)
  {
    if (input == null)
    {
      return Task.FromException<bool>(
        new ArgumentNullException("Missing input parameter"));
    }
    if (input.Length == 0)
    {
      return Task.FromException<bool>(
        new ArgumentException("input parameter is empty."));
    }
    return Task.FromResult(Regex.IsMatch(input,
      @"^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$"));
  }

  // прочие методы
}
```

Если метод, который вам нужно реализовать, возвращает `Task` (эквивалент `void` в синхронном методе), то вы можете вернуть предварительно определенный завершенный объект `Task`:

```
public Task DeleteCustomerAsync()
{
  // ...
  return Task.CompletedTask;
}
```

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 другому ресурсу и тем самым вызвать проблему. По этой причине вам следует внимательно изучить, как сделать код *потокобезопасным*.

Простейший механизм обеспечения потоковой безопасности заключается в использовании объектной переменной в качестве флага или светофора, указывающего, когда на общий ресурс наложена исключительная блокировка.

В книге Уильяма Голдинга «Повелитель мух» (William Golding, Lord of the Flies) Хрюша (Piggy) и Ральф (Ralph) находят раковину и используют ее для организации встречи. Мальчики принимают «правило раковины» и договариваются о том, что говорить имеет право только тот, кто держит ее в руках.

Мне понравилась идея назвать объектную переменную, которую я использую для реализации потокобезопасного кода, той самой «раковиной» (`conch`). Когда у потока есть «раковина», ни один другой поток не должен обращаться к общему ресурсу (или ресурсам), представленному этой «раковиной». Обратите внимание, что я говорю *«должен»*. Только тот код, который учитывает «раковину», обеспечивает синхронизированный доступ. «Раковина» не является блокировкой.

Мы рассмотрим типы, которые можно использовать для синхронизации доступа к ресурсам:

- класс `Monitor` — объект, который может использоваться несколькими потоками для проверки того, должны ли они получить доступ к общему ресурсу в рамках одного процесса;
- класс `Interlocked` — объект управляет простыми числовыми типами на уровне центрального процессора.

Доступ к ресурсу из нескольких потоков

1. Откройте редактор кода и создайте консольное приложение `SynchronizingResourceAccess` в рабочей области/решении `Chapter12`.
2. В программе Visual Studio Code выберите `SynchronizingResourceAccess` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` удалите существующие операторы, добавьте операторы для реализации следующих действий:
   - импортируйте пространство имен для диагностических типов, таких как `Stopwatch`;
   - статически импортируйте тип `Console`;
   - в конце файла `Program.cs` создайте статический класс с двумя полями:
     - полем для для генерации случайного значения времени ожидания;
     - полем `string` для хранения сообщения (общий ресурс);
   - над классом создайте два статических метода, которые добавляют букву `A` или `B` к общей переменной `string` пять раз в цикле и ожидающих каждую итерацию в течение случайного интервала времени в пределах до двух секунд.

```
static void MethodA()
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(SharedObjects.Random.Next(2000));
    SharedObjects.Message += "A";
    Write(".");
  }
}

static void MethodB()
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(SharedObjects.Random.Next(2000));
    SharedObjects.Message += "B";
    Write(".");
  }
}

static class SharedObjects
{
  public static Random Random = new();
  public static string? Message; // общий ресурс
}
```

4. После импорта пространства имен введите операторы для выполнения обоих методов в отдельных потоках с помощью пары задач и дождитесь их завершения, прежде чем выводить значение затраченного времени в миллисекундах:

```
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($"Results: {SharedObjects.Message}.");
WriteLine($"{watch.ElapsedMilliseconds:N0} elapsed milliseconds.");
```

5. Запустите код и проанализируйте результат:

```
Please wait for the tasks to complete.
..........
Results: BABABAABBA.
5,753 elapsed milliseconds.
```

Обратите внимание: в результате оба потока одновременно изменили сообщение. В реальном приложении это чревато проблемой. Мы можем предотвратить одновременный доступ, применив взаимоисключающую блокировку к объекту `conch` и запрограммировав два метода на добровольную проверку переменной `conch` перед изменением общего ресурса, что и сделаем в следующем подразделе.

Применение взаимоисключающей блокировки к «раковине»

Теперь воспользуемся «раковиной» для подтверждения того, что только один поток получает доступ к общему ресурсу одновременно.

1. В код класса `SharedObjects` добавьте экземпляр переменной `object`, выступающей в роли «раковины»:

```
public static object Conch = new();
```

2. В код обоих методов, `MethodA` и `MethodB`, добавьте оператор `lock` для «раковины», окружив им оператор `for`:

```
lock (SharedObjects.Conch)
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(SharedObjects.Random.Next(2000));
```

```
      SharedObjects.Message += "A";
      Write(".");
    }
  }
```

Обратите внимание: поскольку проверка переменной «раковины» является добровольной, если вы используете оператор блокировки lock только в одном из двух методов, доступ к общему ресурсу будет по-прежнему осуществляться обоими методами. Убедитесь, что все методы, которые обращаются к общему ресурсу, учитывают «раковину».

3. Запустите код и проанализируйте результат:

```
Please wait for the tasks to complete.
..........
Results: BBBBBAAAAA.
10,345 elapsed milliseconds.
```

Хотя было затрачено довольно много времени, только один метод одновременно мог получить доступ к общему ресурсу в каждый момент времени. Первым может быть запущен либо `MethodA`, либо `MethodB`. Только после того, как один метод завершает работу с общим ресурсом, «раковина» освобождается и другой метод получает возможность выполнять свою функцию.

Оператор lock

Вы можете задаться вопросом, что делает оператор `lock` при «блокировке» объектной переменной (подсказка: не блокирует объект!)?

```
lock (SharedObjects.Conch)
{
  // работа с общим ресурсом
}
```

Компилятор C# меняет оператор `lock` на оператор `try-finally`, который использует класс `Monitor` для входа и выхода из объекта «раковина» (переменной `conch`) (мне нравится ассоциация «*взять* и *освободить* предмет из “раковины”»):

```
try
{
  Monitor.Enter(SharedObjects.Conch);

  // работа с общим ресурсом
}
finally
{
  Monitor.Exit(SharedObjects.Conch);
}
```

Когда поток вызывает `Monitor.Enter` для любого объекта, также известного как ссылочный тип, он проверяет, не использует ли «раковину» какой-нибудь другой поток. Если да, то поток ожидает. Если нет, то поток берет «раковину» и продолжает работу над общим ресурсом. Завершив работу, поток вызывает `Monitor.Exit`, освобождая «раковину». Если другой поток находился в ожидании, то теперь он может взять «раковину» и выполнить свою задачу. Это требует от всех потоков учитывать «раковину», вызывая `Monitor.Enter` и `Monitor.Exit` соответствующим образом.

Избежание взаимоблокировок

Понимать, как компилятор переводит оператор `lock` в вызовы методов класса `Monitor`, тоже важно, поскольку его использование может привести к взаимоблокировке.

Взаимоблокировки могут возникать при наличии двух или более общих ресурсов (каждый из которых имеет «раковину» для отслеживания того, какой поток в данный момент выполняет работу над каждым общим ресурсом), после чего происходит следующая последовательность событий:

- поток `X` захватывает «раковину» `A` и начинает работать над общим ресурсом `A`;
- поток `Y` захватывает «раковину» `B` и начинает работать над общим ресурсом `B`;
- продолжая работать над ресурсом `A`, поток `X` должен также работать с ресурсом `B`, поэтому он пытается захватить «раковину» `B`, но блокируется, поскольку поток `Y` уже захватил ее;
- продолжая работать над ресурсом `B`, поток `Y` должен также работать с ресурсом `A`, поэтому пытается захватить «раковину» `A`, но блокируется, поскольку поток `X` уже захватил ее.

Один из способов предотвратить взаимоблокировки — указать время ожидания при попытке получить блокировку. Чтобы реализовать такое поведение, вам необходимо вручную использовать класс `Monitor` вместо оператора `lock`.

1. Измените код, заменив оператор `lock` кодом, который пытается войти в «раковину» с временем ожидания и выводит ошибку, а затем выходит из класса `Monitor`, позволяя другим потокам войти в него, как показано ниже:

```
try
{
  if (Monitor.TryEnter(SharedObjects.Conch, TimeSpan.FromSeconds(15)))
  {
    for (int i = 0; i < 5; i++)
    {
      Thread.Sleep(SharedObjects.Random.Next(2000));
```

```
        SharedObjects.Message += "A";
        Write(".");
      }
    }
    else
    {
      WriteLine("Method A timed out when entering a monitor on conch.");
    }
  }
  finally
  {
    Monitor.Exit(SharedObjects.Conch);
  }
```

2. Запустите код и проанализируйте результат. Вы должны увидеть те же результаты, что и раньше (хотя либо A, либо B захватит «раковину» первым). Однако полученный код лучше, поскольку предотвращает потенциальные взаимоблокировки.

Используйте ключевое слово lock только в том случае, если можете написать код так, чтобы избежать потенциальных взаимоблокировок. Если вы не можете избежать их, то всегда используйте метод Monitor.TryEnter вместо lock, в сочетании с оператором try-finally, чтобы задать время ожидания, и один из потоков выйдет из взаимоблокировки, если она возникнет. Больше информации об эффективных методах работы с потоками можно получить по следующей ссылке: https://docs.microsoft.com/en-us/dotnet/standard/threading/managed-threading-best-practices.

Синхронизация событий

В главе 6 вы изучили, как создавать и обрабатывать события. Однако события .NET не являются потокобезопасными, поэтому вам нужно избегать их использования в многопоточных сценариях и следовать стандартному коду создания событий, который я показал вам ранее.

Узнав, что события .NET не являются потокобезопасными, некоторые разработчики пытаются использовать эксклюзивные блокировки при добавлении и удалении обработчиков событий или при создании события, что показано ниже в коде:

```
// поле делегата события
public event EventHandler Shout;

// «раковина»
private object eventLock = new();
```

```
// метод
public void Poke()
{
  lock (eventLock) // неудачная идея
  {
    // если есть слушатель...
    if (Shout != null)
    {
      // ...вызываем делегат, чтобы создать событие
      Shout(this, EventArgs.Empty);
    }
  }
}
```

Более подробную информацию о событиях и потокобезопасности вы можете прочитать на сайте https://docs.microsoft.com/ru-ru/archive/blogs/cburrows/field-like-events-considered-harmful.

Но это сложно, как объясняет Стивен Клири в следующем сообщении блога: https://blog.stephencleary.com/2009/06/threadsafe-events.html.

Выполнение атомарных операций CPU

Слово «атомарный» происходит от греческого слова *atomos*, что означает *«неделимый»*. Важно понимать, какие операции являются атомарными при многопоточности, так как не атомарные может прервать другой поток на полпути в процессе их выполнения. Является ли операция инкремента C# атомарной, как показано в коде ниже?

```
Int x = 3;
x++; // Это атомарная процессорная операция?
```

Операция инкремента — не атомарная! Инкремент целого числа требует выполнения центральным процессором следующих трех операций:

1) загрузки значения из экземпляра переменной в регистр;
2) инкремента (увеличения) значения;
3) сохранения значения в экземпляре переменной.

Первый поток может быть выгружен после выполнения первых двух шагов. Затем второй поток может выполнить все три шага. Возобновив выполнение, первый поток перезапишет значение переменной, а результат инкремента или декремента, выполненный вторым, будет утерян!

Тип `Interlocked` может выполнять атомарные действия над типами значений, таких как целые числа и числа с плавающей запятой.

Рассмотрим пример.

1. Объявите другое поле в классе `SharedObjects`, которое будет подсчитывать количество выполненных операций:

```
public static int Counter; // иной общий ресурс
```

2. В коде методов A и B в операторе `for` после изменения значения `string` добавьте оператор для безопасного инкремента счетчика:

```
Interlocked.Increment(ref SharedObjects.Counter);
```

3. После вывода затраченного времени запишите текущее значение счетчика в консоль:

```
WriteLine($"{SharedObjects.Counter} string modifications.");
```

4. Запустите код и проанализируйте результат, показанный ниже:

```
Please wait for the tasks to complete.
..........
Results: BBBBBAAAAA.
13,531 elapsed milliseconds.
10 string modifications.
```

Внимательные читатели поймут, что существующий объект `conch` защищает *все* общие ресурсы, доступ к которым осуществляется в блоке программного кода, заблокированном «раковиной», и потому в этом конкретном примере фактически нет необходимости задействовать класс `Interlocked`. Но если бы мы еще не защитили другой общий ресурс, такой как `Message`, то использовать класс `Interlocked` было бы необходимо.

Использование других типов синхронизации

Классы `Monitor` и `Interlocked` представляют собой взаимоисключающие блокировки, простые и эффективные, но иногда требуются более сложные параметры для синхронизации доступа к общим ресурсам (табл. 12.5).

Таблица 12.5. Другие типы синхронизации

Тип (-ы)	Описание
`ReaderWriterLock` и `ReaderWriterLockSlim`	Позволяют нескольким потокам быть в *режиме чтения*, еще одному потоку быть в *режиме записи* с исключительным правом написания блокировки и еще одному потоку, имеющему доступ для чтения, быть в *повышаемом режиме чтения*, из которого поток может перейти в режим записи, не отказываясь от доступа для чтения к ресурсу

Тип (-ы)	Описание
`Mutex`	Подобно классу `Monitor`, обеспечивает эксклюзивный доступ к общему ресурсу, за исключением того, что используется для синхронизации между процессами
`Semaphore` и `SemaphoreSlim`	Ограничивают количество потоков, которые могут одновременно обращаться к ресурсу или пулу ресурсов, определяя слоты. Это называется дросселированием ресурсов, а не их блокировкой
`AutoResetEvent` и `ManualResetEvent`	Дескрипторы ожидания событий позволяют потокам синхронизировать действия, сигнализируя друг другу и ожидая сигналов друг от друга

Ключевые слова async и await

В версии C# 5 были введены два ключевых слова C# для работы с типом `Task`. Они особенно полезны для таких ситуаций, как:

- реализация многозадачности для *графического пользовательского интерфейса* (graphical user interface, GUI);
- улучшение масштабируемости веб-приложений и веб-сервисов.

В главе 15 мы изучим, как с помощью ключевых слов `async` и `await` улучшить масштабируемость сайтов.

В главе 19 мы рассмотрим, как ключевые слова `async` и `await` позволяют реализовывать многозадачность для графических пользовательских интерфейсов.

А пока узнаем, почему эти два ключевых слова были введены в язык C#, и воспользуемся ими на практике.

Увеличение скорости отклика консольных приложений

Одно из ограничений консольных приложений заключается в том, что слово `await` можно использовать только в методах, помеченных как `async`, при этом C# 7 и более ранние версии не позволяют методу `Main` быть асинхронным! К счастью, в C# 7.1 была введена новая функция — поддержка `async` для метода `Main`.

1. Откройте редактор кода и создайте консольное приложение `AsyncConsole` в рабочей области/решении `Chapter12`.
2. В Visual Studio Code выберите `AsyncConsole` в качестве активного проекта OmniSharp.

3. В файле `Program.cs` удалите существующие операторы и статически импортируйте класс `Console`:

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

4. Добавьте операторы для создания экземпляра `HttpClient`, сделайте запрос на главную страницу Apple и выведите ее размер в байтах:

```
HttpClient client = new();

HttpResponseMessage response =
  await client.GetAsync("http://www.apple.com/");

WriteLine("Apple's home page has {0:N0} bytes.",
  response.Content.Headers.ContentLength);
```

5. Соберите проект и обратите внимание, что он собран успешно. В .NET 5 и более ранних версиях вы бы увидели такое сообщение об ошибке:

```
Program.cs(14,9): 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/
Chapter12/AsyncConsole/AsyncConsole.csproj]
```

6. Вам бы пришлось добавить ключевое слово `async` к методу `Main` и изменить возвращаемый тип на `Task`. В .NET 6 и более поздних версиях шаблон проекта консольного приложения использует функцию программы верхнего уровня для автоматического определения класса `Program` с помощью асинхронного метода `Main`.
7. Запустите код и проанализируйте результат, который, вероятно, будет иметь другое количество байтов, поскольку Apple часто меняет свою главную страницу:

```
Apple's home page has 40,252 bytes.
```

Увеличение скорости отклика GUI-приложений

В этой книге мы создавали только консольные приложения. Жизнь разработчика усложняется при создании веб-приложений, веб-сервисов и приложений с графическим пользовательским интерфейсом, таких как настольные Windows-приложения и мобильные приложения.

Одна из причин усложнения заключается в том, что для GUI-приложения есть специальный поток *пользовательского интерфейса* (user interface, UI).

Существует два правила, регулирующие работу в GUI:

- не выполнять задачи с длительным временем выполнения в потоке пользовательского интерфейса;
- не обращаться к UI-элементам ни в одном потоке, кроме потока пользовательского интерфейса.

Для соблюдения этих правил разработчикам приходилось писать сложный код, чтобы обеспечить выполнение длительных задач потоком, не связанным с пользовательским интерфейсом, но чтобы при этом после завершения такой задачи результаты ее выполнения безопасно передавались UI-потоку для отображения пользователю. Это чревато тем, что все может пойти наперекосяк!

К счастью, в C# 5 и более поздних версиях были введены ключевые слова `async` и `await`. Они позволяют вам продолжать писать код, как если бы он был синхронным, что делает его чистым и легким для понимания. Однако компилятор C# скрытно от вас создает сложную машину состояний и отслеживает запущенные потоки. Это нечто удивительное!

Рассмотрим пример. Мы создадим настольное Windows-приложение на основе WPF, которое переводит данные о сотрудниках из базы данных Northwind в базу данных SQL Server, используя такие низкоуровневые типы, как `SqlConnection`, `SqlCommand` и `SqlDataReader`. Вы сможете выполнить это задание, только если у вас есть компьютер под управлением операционной системы Windows и база данных Northwind, хранящаяся в SQL Server. Это единственный раздел в книге, который не является кросс-платформенным и современным (WPF уже 16 лет!).

На данном этапе мы сосредоточимся на том, чтобы сделать приложение с графическим пользовательским интерфейсом отзывчивым. Вы узнаете о XAML и создании кросс-платформенных GUI-приложений в главе 19. Поскольку в этой книге WPF не рассматривается, я подумал, что это задание станет хорошей возможностью хотя бы взглянуть на пример приложения, созданного с помощью WPF, даже если мы не будем рассматривать его подробно.

Приступим!

1. Если вы работаете в Visual Studio 2022 для Windows, добавьте новый проект `WPF Application [C#]` с именем `WpfResponsive` в решение `Chapter12`. Если вы используете программу Visual Studio Code, то выполните команду `dotnet new wpf`.
2. Настройки файла проекта должны быть следующими: тип вывода — Windows EXE, целевой фреймворк — .NET 6 для Windows (программа не будет работать на других платформах, таких как macOS и Linux), и в проекте используется WPF.

3. Добавьте в проект ссылку на пакет `Microsoft.Data.SqlClient`:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net6.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Data.SqlClient" Version="3.0.0"/>
  </ItemGroup>

</Project>
```

4. Соберите проект для восстановления пакетов.
5. В файле `MainWindow.xaml` в элементе `<Grid>` добавьте элементы для определения двух кнопок, текстового поля и поля списка, расположенных вертикально на панели стека, как показано ниже:

```
<Grid>
  <StackPanel>
    <Button Name="GetEmployeesSyncButton"
            Click="GetEmployeesSyncButton_Click">
      Get Employees Synchronously</Button>
    <Button Name="GetEmployeesAsyncButton"
            Click="GetEmployeesAsyncButton_Click">
      Get Employees Asynchronously</Button>
    <TextBox HorizontalAlignment="Stretch" Text="Type in here" />
    <ListBox Name="EmployeesListBox" Height="400" />
  </StackPanel>
</Grid>
```

Программа Visual Studio 2022 для Windows обеспечивает хорошую поддержку разработки WPF-приложений и предоставляет функционал IntelliSense при редактировании кода и разметки XAML. В программе Visual Studio Code такого нет.

6. В файле `MainWindow.xaml.cs` в классе `MainWindow` импортируйте пространства имен `System.Diagnostics` и `Microsoft.Data.SqlClient`, создайте две константы `string` для строки подключения к базе данных и оператора SQL. Затем добавьте обработчики событий для нажатия двух кнопок, которые используют эти кон-

станты, чтобы открыть соединение с базой данных Northwind и заполнить поля списка идентификаторами и именами всех сотрудников:

```
private const string connectionString =
  "Data Source=.;" +
  "Initial Catalog=Northwind;" +
  "Integrated Security=true;" +
  "MultipleActiveResultSets=true;";

private const string sql =
  "WAITFOR DELAY '00:00:05';" +
  "SELECT EmployeeId, FirstName, LastName FROM Employees";

private void GetEmployeesSyncButton_Click(object sender, RoutedEventArgs e)
{
  Stopwatch timer = Stopwatch.StartNew();
  using (SqlConnection connection = new(connectionString))
  {
    connection.Open();
    SqlCommand command = new(sql, connection);
    SqlDataReader reader = command.ExecuteReader();

    while (reader.Read())
    {
      string employee = string.Format("{0}: {1} {2}",
        reader.GetInt32(0), reader.GetString(1), reader.GetString(2));

      EmployeesListBox.Items.Add(employee);
    }
    reader.Close();
    connection.Close();
  }
  EmployeesListBox.Items.Add($"Sync: {timer.ElapsedMilliseconds:N0}ms");
}

private async void GetEmployeesAsyncButton_Click(
  object sender, RoutedEventArgs e)
{
  Stopwatch timer = Stopwatch.StartNew();
  using (SqlConnection connection = new(connectionString))
  {
    await connection.OpenAsync();
    SqlCommand command = new(sql, connection);
    SqlDataReader reader = await command.ExecuteReaderAsync();

    while (await reader.ReadAsync())
    {
      string employee = string.Format("{0}: {1} {2}",
```

```
            await reader.GetFieldValueAsync<int>(0),
            await reader.GetFieldValueAsync<string>(1),
            await reader.GetFieldValueAsync<string>(2));

        EmployeesListBox.Items.Add(employee);
      }
      await reader.CloseAsync();
      await connection.CloseAsync();
    }
    EmployeesListBox.Items.Add($"Async: {timer.ElapsedMilliseconds:N0}ms");
  }
```

Обратите внимание на следующие моменты:

- в операторе SQL используется команда SQL Server `WAITFOR DELAY` для имитации обработки, которая занимает пять секунд. Затем он выбирает три столбца из таблицы `Employees`;
- обработчик события `GetEmployeesSyncButton_Click` использует синхронные методы, чтобы открыть соединение и получить строки с данными о сотрудниках;
- обработчик события `GetEmployeesAsyncButton_Click` помечен как асинхронный и использует асинхронные методы с ключевым словом `await`, чтобы открыть соединение и получить строки с данными о сотрудниках;
- оба обработчика событий используют секундомер для записи количества миллисекунд, затраченных на выполнение операции, и добавляют его в поле списка.

7. Запустите приложение WPF без отладки.
8. Щелкните в текстовом поле, введите текст и обратите внимание, как реагирует графический пользовательский интерфейс.
9. Нажмите кнопку Get Employees Synchronously (Получить данные о сотрудниках синхронно).
10. Попробуйте щелкнуть в текстовом поле и обратите внимание, что графический пользовательский интерфейс не реагирует.
11. Подождите не менее пяти секунд, пока поле списка не заполнится данными о сотрудниках.
12. Щелкните в текстовом поле, введите текст и обратите внимание, что графический пользовательский интерфейс снова стал реагировать.
13. Нажмите кнопку Get Employees Asynchronously (Получить данные о сотрудниках асинхронно).
14. Щелкните в текстовом поле, введите текст и обратите внимание на то, что графический пользовательский интерфейс по-прежнему реагирует, выполняя операцию. Продолжайте вводить текст до тех пор, пока список не заполнится данными о сотрудниках.

15. Обратите внимание на разницу во времени выполнения этих двух операций. При синхронной выборке данных пользовательский интерфейс блокируется, а при асинхронной остается способным быстро реагировать.
16. Закройте приложение WPF.

Улучшение масштабируемости клиент-серверных приложений

Ключевые слова `async` и `await` могут применяться на стороне сервера при разработке сайтов, веб-приложений и сервисов. С точки зрения клиентского приложения ничего не меняется (или даже может немного увеличиться время возврата запроса). Таким образом, с точки зрения одного клиента применение `async` и `await` для реализации многозадачности на стороне сервера негативно влияет на пользовательский опыт!

Однако на стороне сервера создаются дополнительные более дешевые рабочие потоки, которые ожидают завершения длительных задач, чтобы дорогостоящие потоки ввода-вывода могли обрабатывать запросы других клиентов, а не блокироваться. Это положительно сказывается на общей масштабируемости веб-приложения или сервиса. Одновременно обеспечивается поддержка большего количества клиентов.

Общие типы, поддерживающие многозадачность

В табл. 12.6 приведены наиболее общие типы, поддерживающие асинхронные методы.

Таблица 12.6. Общие типы, поддерживающие асинхронные методы

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 `FlushAs`

Каждый раз, встречая метод, оканчивающийся суффиксом Async, вы должны проверять, возвращает ли он метод Task или Task<T>. Если да, то вы можете использовать этот метод вместо синхронного метода без суффикса Async. Не забудьте, что вызывать его надо с ключевым словом await, дополнив ваш метод словом async.

Ключевое слово await в блоках catch

Когда `async` и `await` были впервые представлены в версии C# 5, ключевое слово `await` можно было использовать только в блоке `try`, но не в `catch`. В C# 6 и более поздних версиях вы можете применять ключевое слово `await` в обоих блоках: и в `try`, и в `catch`.

Работа с асинхронными потоками

В версии .NET Core 3.0 компания Microsoft представила асинхронную обработку потоков.

Обучающие материалы по асинхронным потокам можно получить по следующей ссылке: https://docs.Microsoft.com/en-us/dotnet/csharp/tutorials/generate-consume-asynchronous-stream.

До выпуска C# 8.0 и .NET Core 3.0 ключевое слово `await` можно было использовать только с задачами, возвращающими скалярные значения. Поддержка асинхронного потока в .NET Standard 2.1 позволила асинхронному методу возвращать последовательность значений.

Рассмотрим пример, возвращающий три случайных целых числа в виде асинхронного потока.

1. Откройте редактор кода и создайте консольное приложение `AsyncEnumerable` в рабочей области/решении `Chapter12`.
2. В программе Visual Studio Code выберите `AsyncEnumerable` в качестве активного проекта OmniSharp.
3. В файле `Program.cs` удалите существующие операторы и статически импортируйте класс `Console`:

```
using static System.Console; // WriteLine()
```

4. В конце файла `Program.cs` создайте метод, который использует ключевое слово `yield` для асинхронного возврата случайной последовательности из трех чисел:

```
async static IAsyncEnumerable<int> GetNumbersAsync()
{
  Random r = new();

  // имитация работы
  await Task.Delay(r.Next(1500, 3000));
  yield return r.Next(0, 1001);

  await Task.Delay(r.Next(1500, 3000));
  yield return r.Next(0, 1001);
```

```
  await Task.Delay(r.Next(1500, 3000));
  yield return r.Next(0, 1001);
}
```

5. Выше метода `GetNumbersAsync` добавьте операторы для перечисления последовательности чисел:

```
await foreach (int number in GetNumbersAsync())
{
  WriteLine($"Number: {number}");
}
```

6. Запустите код и проанализируйте результат:

```
Number: 509
Number: 813
Number: 307
```

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 12.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какую информацию о классе `Process` вы можете найти?
2. Насколько точен класс `Stopwatch`?
3. По соглашению, какой суффикс следует применять к методу, если он возвращает `Task` или `Task<T>`?
4. Какое ключевое слово нужно 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 применять класс `Interlocked`?
8. Когда класс `Mutex` следует задействовать вместо класса `Monitor`?
9. В чем преимущество использования ключевых слов `async` и `await` на сайте или в веб-сервисе?
10. Можно ли отменить задачу? Если да, то каким образом?

Упражнение 12.2. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-12---improving-performance-and-scalability-using-multitasking, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

Из этой главы вы узнали, как определять и запускать задачи, дожидаться завершения одной или нескольких задач и управлять порядком выполнения задач. Вы также узнали, как синхронизировать доступ к общим ресурсам и о том, что стоит за ключевыми словами `async` и `await`.

В последующих семи главах вы узнаете, как создавать приложения для *моделей приложений* (они же *рабочие нагрузки*), поддерживаемых .NET, таких как сайты, сервисы, а также кросс-платформенные настольные и мобильные приложения.

13 Реальные приложения на C# и .NET

Третья и финальная часть этой книги посвящена реальным приложениям, разработанным на C# и .NET. Вы узнаете, как создавать кросс-платформенные приложения, такие как веб-сайты, сервисы, мобильные и настольные приложения.

Корпорация Microsoft предоставляет платформы для создания приложений на основе *моделей приложений* или *рабочих нагрузок*.

В главах 1–18 и в 20-й для создания приложений вы можете использовать программу Visual Studio для вашей операционной системы или кросс-платформенные инструменты Visual Studio Code и JetBrains Rider. В главе 19 вы можете использовать программу Visual Studio Code для создания мобильного и настольного приложения, но это будет не очень удобно. Программа Visual Studio 2022 для Windows имеет лучшую поддержку .NET MAUI, чем Visual Studio Code (на данный момент).

Я рекомендую вам проработать эту и следующие главы последовательно, так как в них будут ссылки на проекты из предыдущих глав. Таким образом вы накопите достаточно знаний и навыков для решения более сложных задач, которые будут представлены в последующих главах.

В этой главе:

- модели приложений для C# и .NET;
- нововведения ASP.NET Core;
- структурирование проектов;
- использование различных шаблонов проектов;
- разработка сущностной модели данных для Northwind.

Модели приложений для C# и .NET

В оставшихся главах мы узнаем о моделях приложений, рассмотренных на примере реальных приложений, разработаных на C# 10 и .NET 6, которому и посвящена эта книга.

Компания Microsoft предоставляет подробные рекомендации по реализации моделей приложений. Они размещены в документации, посвященной архитектуре приложений .NET: https://www.microsoft.com/net/learn/architecture.

Разработка сайтов с помощью ASP.NET Core

Сайты состоят из нескольких страниц, которые статически загружаются из файловой системы или динамически генерируются технологией разработки кода на стороне сервера, такой как ASP.NET Core. Браузер выполняет запросы `GET`, используя *уникальные адреса ресурсов* (Unique Resource Locators, URL), идентифицирующие каждую страницу, и может управлять данными, хранящимися на сервере, с помощью запросов `POST`, `PUT` и `DELETE`.

Для большинства сайтов браузер рассматривается как уровень представления, причем почти вся обработка выполняется на стороне сервера. На стороне клиента для реализации некоторых функций представления могут использоваться некоторые элементы JavaScript, например карусель.

Платформа ASP.NET Core предоставляет три технологии для создания сайтов:

- *ASP.NET Core Razor Pages* и *библиотеки классов Razor* служат для динамического создания HTML для простых сайтов. Более подробно об этом вы узнаете в главе 14;
- *ASP.NET Core MVC* — реализация паттерна проектирования Model-View-Controller (*MVC*, «Модель — представление — контроллер»), часто применяемого при разработке сложных сайтов. Более подробную информацию об этом вы узнаете в главе 15;
- *Blazor* позволяет разрабатывать компоненты пользовательского интерфейса на языке C# и платформе .NET вместо фреймворков на основе JavaScript, таких как Angular, React и Vue. *Blazor WebAssembly* запускает ваш код в браузере, как это делает фреймворк на основе JavaScript. *Blazor Server* запускает ваш код на сервере и динамически обновляет веб-страницу. Подробно о Blazor вы узнаете в главе 17. Blazor предназначен не только для создания сайтов; его можно использовать и для разработки гибридных мобильных и настольных приложений.

Разработка сайтов с помощью системы управления контентом

Большинство сайтов содержит много контента, и если бы для изменения какого-либо материала приходилось бы каждый раз привлекать разработчиков, то сайты не могли бы хорошо масштабироваться. *Система управления контентом* (Content Management System, CMS) позволяет разработчикам определять структуру и шаблоны контента, чтобы обеспечить согласованность и хороший дизайн,

а не владеющему техническими нюансами владельцу контента дает возможность легко управлять фактическим наполнением сайта. Владелец может создавать новые страницы или блоки контента и обновлять существующий контент, зная, что ему не придется прикладывать особых усилий, чтобы сайт отлично смотрелся для конечного пользователя.

Существует множество CMS, доступных для всех веб-платформ, например WordPress для PHP или Django для Python. Среди CMS, поддерживающих современную .NET, стоит упомянуть Optimizely Content Cloud, Piranha CMS и Orchard Core.

Основное преимущество использования CMS состоит в том, что эта система обеспечивает дружественный UI управления контентом. Владельцы контента заходят на сайт и сами управляют контентом. Затем он визуализируется и возвращается пользователям с помощью контроллеров и представлений ASP.NET MVC или через конечные точки веб-сервиса, известные как headless CMS, чтобы быть представленным мобильным и настольным приложениям, средствам коммуникации в коммерции и клиентскому ПО, разработанному на фреймворках JavaScript или Blazor.

В этой книге не рассматриваются системы CMS .NET, поэтому в репозитории GitHub я опубликовал ссылки, по которым вы можете узнать о них больше: https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#net-content-management-systems.

Создание веб-приложений с помощью фреймворков SPA

Веб-приложения, также известные как *одностраничные приложения* (single page application, SPA), состоят из одной веб-страницы, созданной с помощью таких фронтенд-технологий, как Blazor WebAssembly, Angular, React, Vue, или проприетарной библиотеки JavaScript, которая может отправлять запросы к бэкенд-веб-сервису, чтобы получить при необходимости больше данных и публиковать обновленные данные, используя распространенные форматы сериализации, такие как XML и JSON. Классические примеры — это веб-приложения Google: Gmail, Maps и Docs.

В веб-приложении на стороне клиента для реализации сложных взаимодействий с пользователем применяются фреймворки JavaScript или Blazor WebAssembly, но бóльшая часть важной обработки и доступа к данным все еще происходит на стороне сервера, поскольку браузер имеет ограниченный доступ к ресурсам локальной системы.

JavaScript имеет слабую типизацию и не предназначен для сложных проектов, поэтому большинство библиотек JavaScript в настоящее время используют Microsoft TypeScript, который добавляет строгую типизацию в JavaScript и содержит множество современных языковых функций для обработки сложных реализаций.

.NET SDK содержит шаблоны проектов для SPA на основе JavaScript и TypeScript, но, хоть они и используются обычно с ASP.NET Core в качестве бэкенда, мы не станем их изучать, так как эта книга посвящена языку C#, а не другим языкам программирования.

Таким образом, C# и .NET можно использовать для создания сайтов как на стороне сервера, так и на стороне клиента (рис. 13.1)

Рис. 13.1. Использование C# и .NET для создания сайтов как на стороне сервера, так и на стороне клиента

Создание веб- и других сервисов

Даже если мы не будем изучать SPA на основе JavaScript, мы узнаем, как создать веб-сервис с помощью *ASP.NET Core Web API*, затем вызвать его из серверного кода на сайтах ASP.NET Core. Позже мы будем вызывать этот веб-сервис из веб-компонентов Blazor WebAssembly и кросс-платформенных мобильных и настольных приложений.

Формально градации не существует, но иногда сервисы различаются по своей сложности:

- *сервис* — вся функциональность, необходимая клиентскому приложению, реализована в одном монолитном сервисе;
- *микросервис* — несколько сервисов, каждый из которых предоставляет небольшой набор функций;
- *наносервис* — одна функция, предоставляемая как сервис. В отличие от сервисов и микросервисов, которые работают непрерывно, наносервисы часто неактивны до тех пор, пока к ним не обратятся, в целях сокращения ресурсов и затрат.

Помимо веб-сервисов, использующих HTTP в качестве базового протокола обмена данными и принципы проектирования API, мы научимся создавать сервисы с помощью других технологий и принципов проектирования, в том числе:

- *gRPC* для создания высокоэффективных и производительных сервисов с поддержкой практически любой платформы;
- *SignalR* для создания взаимодействия между компонентами в режиме реального времени;
- *OData* для обертывания Entity Framework Core и других моделей данных в веб-интерфейс API;
- *GraphQL*, позволяющего клиенту контролировать, какие данные извлекаются из нескольких источников данных;
- *Azure Functions* для размещения бессерверных наносервисов в облаке.

Создание мобильных и настольных приложений

Существует две основные мобильные платформы: iOS от Apple и Android от Google, каждая из которых имеет свои языки программирования и API платформы. Есть также две основные платформы для настольных компьютеров: macOS от Apple и Windows от Microsoft, каждая из которых имеет свои языки программирования и API платформы, как показано в следующем списке:

- *iOS*: Objective C или Swift и UIkit;
- *Android*: Java или Kotlin и Android API;
- *macOS*: Objective C или Swift и AppKit или Catalyst;
- *Windows*: C, C++ или многие другие языки, а также Win32 API или Windows App SDK.

Поскольку эта книга посвящена современной кросс-платформенной разработке с помощью C# и .NET, в ней не рассматривается создание настольных приложений с использованием *Windows Forms*, *Windows Presentation Foundation (WPF)* или *Universal Windows Platform (UWP)*, так как они предназначены только для Windows.

Кросс-платформенные мобильные и настольные приложения могут быть созданы один раз для платформы *.NET Multi-platform App User Interfaces (MAUI)*, а затем могут запускаться и работать на многих мобильных и настольных платформах.

.NET MAUI упрощает разработку таких приложений за счет совместного использования компонентов UI, а также бизнес-логики. Они могут ориентироваться на те же API .NET, которые используются консольными приложениями, сайтами и веб-сервисами. Приложение будет выполняться средой выполнения Mono на мобильных устройствах и средой выполнения CoreCLR на настольных устройствах. Среда выполнения Mono лучше оптимизирована для мобильных устройств по сравнению с обычной средой выполнения .NET CoreCLR. Blazor WebAssembly также использует среду выполнения Mono, поскольку, как и мобильные приложения, она ограничена в ресурсах.

Приложения могут существовать сами по себе, но обычно вызывают сервисы, чтобы обеспечить взаимодействие, которое распространяется на все ваши вычислительные устройства, от серверов и ноутбуков до телефонов и игровых систем.

Будущие обновления .NET MAUI будут поддерживать существующие паттерны MVVM и XAML и такие паттерны, как *Model-View-Update (MVU)* с C#, который похож на Swift UI от Apple.

Предпоследняя глава этого, шестого издания — глава 19 — посвящена использованию .NET MAUI для создания кросс-платформенных мобильных и настольных приложений.

Альтернативы .NET MAUI

До того как Microsoft создала .NET MAUI, третьи стороны создали инициативы с открытым исходным кодом, чтобы позволить .NET разработчикам создавать кросс-платформенные приложения с помощью XAML под названием *Uno* и *Avalonia*.

Платформа Uno

Как заявляет Uno на своем сайте, это «первая и единственная платформа пользовательского интерфейса для однокодовых приложений для Windows, WebAssembly, iOS, macOS, Android и Linux».

Разработчики могут повторно использовать 99 % бизнес-логики и слоя пользовательского интерфейса в нативных мобильных, веб- и настольных приложениях.

Платформа Uno использует инструментальную платформу Xamarin, но не Xamarin.Forms. Для WebAssembly Uno применяет среду исполнения Mono-WASM точно так же, как и Blazor WebAssembly. Для Linux Uno использует Skia, предназначенную для рисования пользовательского интерфейса на холсте.

Платформа Avalonia

Как указано на сайте .NET Foundation, Avalonia «представляет собой кросс-платформенную среду пользовательского интерфейса на основе XAML, которая обеспечивает гибкую систему стилей и поддерживает широкий спектр операционных систем, таких как Windows, Linux с помощью Xorg, macOS. Avalonia готова для разработки настольных приложений общего назначения».

Avalonia можно рассматривать как духовного преемника WPF. Разработчики WPF, Silverlight и UWP, знакомые с WPF, могут продолжать извлекать выгоду из своих многолетних знаний и навыков.

Компания JetBrains использовала Avalonia для модернизации своих инструментов на базе WPF и перевода их на кросс-платформенный уровень.

Расширение Avalonia для Visual Studio и глубокая интеграция с JetBrains Rider делает разработку проще и продуктивнее.

Нововведения ASP.NET Core

За последние несколько лет компания Microsoft быстро расширила возможности ASP.NET Core. Платформы .NET, которые поддерживаются на сегодняшний день, перечислены ниже:

- ASP.NET Core 1.0–2.2 работают на основе платформы .NET Core или .NET Framework;
- ASP.NET Core 3.0 работает только на основе платформы .NET Core 3.0 или более поздних версий.

ASP.NET Core 1.0

Версия ASP.NET Core 1.0 вышла в июне 2016 года и была ориентирована на реализацию минимального API, подходящего для создания современных кросс-платформенных веб-приложений и сервисов для операционных систем Windows, macOS и Linux.

ASP.NET Core 1.1

Версия ASP.NET Core 1.1 вышла в ноябре 2016 года и была сосредоточена на исправлении обнаруженных ошибок и общих улучшениях функционала и производительности.

ASP.NET Core 2.0

Версия ASP.NET Core 2.0 вышла в августе 2017 года и была ориентирована на добавление новых функций, таких как Razor Pages, объединение сборок в метапакет `Microsoft.AspNetCore.All`, поддержку .NET Standard 2.0, предоставление новой модели аутентификации и улучшение производительности. Самыми крупными новыми функциями, представленными в ASP.NET Core 2.0, являются ASP.NET Core Razor Pages, о которых рассказывается в главе 14, и поддержка ASP.NET Core OData, которой посвящена глава 18.

ASP.NET Core 2.1

Версия ASP.NET Core 2.1 вышла в мае 2018 года и представляла собой выпуск с *долгосрочной поддержкой* (Long Term Support, LTS), то есть версия поддерживалась в течение трех лет до 21 августа 2021 года (обозначение LTS было официально присвоено только в августе 2018 года с версией 2.1.3).

Особое внимание уделялось добавлению новых функций, таких как SignalR для взаимодействия в режиме реального времени, библиотеки классов Razor для повторного использования веб-компонентов и ASP.NET Core Identity для идентификации. Кроме того, эта версия фокусировалась на улучшении поддержки HTTPS и *Общего регламента ЕС по защите данных* (General Data Protection Regulation, GDPR), включая темы, перечисленные в табл. 13.1.

Таблица 13.1. Изменения в ASP.NET Core 2.1

Функционал	Глава	Тема
Библиотеки классов Razor	14	Использование библиотек классов Razor
Поддержка GDPR	15	Разработка и изучение сайта ASP.NET Core MVC
UI-библиотека Identity и создание шаблонов	15	Изучение сайта ASP.NET Core MVC
Интеграционное тестирование	15	Тестирование сайта ASP.NET Core MVC
`[ApiController]`, `ActionResult<T>`	16	Разработка проекта ASP.NET Core Web API
Сведения о проблеме	16	Реализация контроллера Web API
`IHttpClientFactory`	16	Настройка HTTP-клиентов с помощью `HttpClientFactory`
ASP.NET Core SignalR	18	Реализация взаимодействия в режиме реального времени с помощью SignalR

ASP.NET Core 2.2

Версия ASP.NET Core 2.2 вышла в декабре 2018 года и была направлена на улучшение построения RESTful HTTP API, обновлении шаблонов проектов до Bootstrap 4 и Angular 6, оптимизированную конфигурацию для хостинга в Azure и повышение производительности, включая темы, перечисленные в табл. 13.2.

Таблица 13.2. Изменения в ASP.NET Core 2.2

Функционал	Глава	Тема
HTTP/2 в Kestrel	14	Классическая ASP.NET против современной ASP.NET Core
Модель внутрипроцессного хостинга	14	Разработка проекта ASP.NET Core
Маршрутизация к конечным точкам	14	Настройки маршрутизации к конечным точкам
Health Check API	16	Реализация Health Check API
Анализаторы Open API	16	Реализация анализаторов и соглашений Open API

ASP.NET Core 3.0

Версия ASP.NET Core 3.0 вышла в сентябре 2019 года и была ориентирована на полноценное использование .NET Core 3.0 и .NET Standard 2.1. Это значит, что данная версия не могла поддерживать .NET Framework и была дополнена полезными обновлениями, включая темы, перечисленные в табл. 13.3.

Таблица 13.3. Изменения в ASP.NET Core 3.0

Функционал	Глава	Тема
Статическое содержимое в библиотеках классов Razor	14	Использование библиотек классов Razor
Новые настройки регистрации сервиса MVC	15	Запуск ASP.NET Core MVC
ASP.NET Core gRPC	18	Создание сервисов с помощью ASP.NET Core gRPC
Blazor Server	17	Создание компонентов с помощью Blazor Server

ASP.NET Core 3.1

Версия ASP.NET Core 3.1 вышла в декабре 2019 года и является выпуском LTS; это значит, он будет поддерживаться до 3 декабря 2022 года. Особое внимание уделяется таким улучшениям, как поддержка частичного класса для компонентов Razor и создание нового помощника тега `<component>`.

Blazor WebAssembly 3.2

Версия Blazor WebAssembly 3.2 вышла в мае 2020 года. Она являлась текущим выпуском; это значит, проекты должны были быть обновлены до версии .NET 5 в течение трех месяцев после выпуска .NET 5, то есть до 10 февраля 2021 года. Компания Microsoft наконец выполнила обещание о полноценной веб-разработке с помощью .NET, Blazor Server и Blazor WebAssembly, которые мы будем изучать в главе 17.

ASP.NET Core 5.0

Версия ASP.NET Core 5.0 вышла в ноябре 2020 года и была нацелена на исправление ошибок, улучшение производительности за счет кэширования для проверки подлинности сертификата, динамическое сжатие HPack заголовков ответов HTTP/2 в Kestrel. Кроме того, особое внимание уделялось аннотациям, допускающим значение `NULL`, для сборок ASP.NET Core и уменьшению размеров образа контейнера, включая темы, перечисленные в табл. 13.4.

Таблица 13.4. Изменения в ASP.NET Core 5.0

Функционал	Глава	Тема
Метод расширения, позволяющего анонимный доступ к конечной точке	16	Защита веб-сервисов
Методы расширения JSON для `HttpRequest` и `HttpResponse`	16	Получение клиентов в виде JSON в контроллере

ASP.NET Core 6.0

Версия ASP.NET Core 6.0 вышла в ноябре 2021 года и была направлена на улучшение производительности, в том числе минимизацию кода для реализации базовых сайтов и сервисов, .NET Hot Reload и новые возможности хостинга для Blazor, такие как гибридные приложения с использованием .NET MAUI, включая темы, перечисленные в табл. 13.5.

Таблица 13.5. Изменения в ASP.NET Core 6.0

Функционал	Глава	Тема
Новый шаблон пустого веб-проекта	14	Шаблон пустого веб-проекта
Промежуточное программное обеспечение для логирования HTTP	16	Включение логирования HTTP
Минимальные API	16	Реализация минимальных Web API
Границы ошибок Blazor	17	Определение границ ошибок Blazor
Blazor WebAssembly AOT (ahead-of-time — «заблаговременный»)	17	Включение заблаговременной компиляции Blazor WebAssembly
.NET Hot Reload	17	Исправление кода с помощью .NET Hot Reload
Приложения .NET MAUI Blazor	19	Размещение компонентов Blazor в приложениях .NET MAUI

Создание настольных приложений только для Windows

Технологии для создания настольных приложений только для Windows включают в себя:

- *Windows Forms*, 2002 год;
- *Windows Presentation Foundation (WPF)*, 2006 год;
- приложения магазина *Windows Store*, 2012 год;
- приложения для *универсальной платформы Windows (Universal Windows Platform, UWP)*, 2015 год;
- приложения *Windows App SDK* (ранее *WinUI 3* и *Project Reunion*), 2021 год.

Устаревшие платформы приложений Windows

С момента выхода Microsoft Windows 1.0 в 1985 году приложения Windows создавались только с помощью языка C и вызова функций в трех основных библиотеках DLL, названных kernel, user и GDI. Когда Windows стала 32-битной (это произошло с выходом Windows 95), DLL-библиотеки получили суффикс 32 и стали известны как *Win32 API*.

В 1991 году компания Microsoft выпустила Visual Basic, который предоставил разработчикам визуальный способ создания пользовательского интерфейса для

приложений Windows с помощью перетаскивания из одного набора элементов управления в другой. Visual Basic пользовался огромной популярностью, а одноименная среда выполнения и сегодня по-прежнему распространяется как часть Windows 10.

Выпустив первую версию C# и .NET Framework в 2002 году, компания Microsoft предоставила технологию *Windows Forms* для создания настольных приложений Windows. В тот период времени существовал аналог для веб-разработки, он назывался *Web Forms*, отсюда и созвучные названия. Код мог быть написан на языках Visual Basic или C#. Windows Forms имела аналогичный визуальный конструктор с перетаскиванием, хотя для определения пользовательского интерфейса он генерировал код на C# или Visual Basic, который может быть сложным для восприятия людьми и редактирования напрямую.

В 2006 году компания Microsoft выпустила более мощную технологию для создания настольных приложений Windows, названную *Windows Presentation Foundation (WPF)*, в качестве ключевого компонента .NET Framework 3.0 наряду с *Windows Communication Foundation (WCF)* и *Windows Workflow (WF)*.

Хотя приложение WPF можно создать, написав только операторы C#, оно может также использовать *расширяемый язык разметки приложений* (eXtensible Application Markup Language, XAML) для определения своего пользовательского интерфейса, который легко понять как человеку, так и коду. Visual Studio для Windows частично создана с помощью WPF.

В 2012 году компания Microsoft выпустила Windows 8 со своими приложениями Windows Store, работающими в защищенной изолированной среде.

В 2015 году компания Microsoft выпустила Windows 10 с обновленной концепцией приложений Windows Store под названием *Universal Windows Platform (UWP)*. Приложения UWP могут быть созданы с помощью языка C++ и пользовательского интерфейса DirectX, или JavaScript и HTML, или языка C# с использованием собственного ответвления кода современной .NET, который не является кросс-платформенным, но предоставляет полный доступ к базовым API WinRT.

Приложения UWP могут выполняться только на платформе Windows 10, а не в более ранних версиях Windows, но приложения UWP могут запускаться на гарнитурах смешанной реальности Xbox и Windows с контроллерами движения.

Многие разработчики Windows отказались от приложений Windows Store и UWP из-за ограниченного доступа к базовой системе. Недавно Microsoft создала *Project Reunion* и *WinUI 3*, которые работают вместе, чтобы позволить разработчикам Windows привнести некоторые преимущества современной разработки Windows в свои существующие приложения WPF и дать им возможность пользоваться благами системной интеграции, которые есть у приложений UWP. Эта инициатива теперь известна как *Windows App SDK*.

Современная поддержка .NET для устаревших платформ Windows

Дисковый размер .NET SDK для Linux и macOS составляет около 330 Мбайт, для Windows — около 440 Мбайт. Разница объясняется тем, что .NET SDK включает в себя Windows Desktop Runtime (среда выполнения Windows), которая позволяет запускать устаревшие платформы приложений Windows Forms и WPF на современной платформе .NET.

Существует множество корпоративных приложений, созданных с использованием Windows Forms и WPF, которые необходимо сопровождать или улучшать с помощью новых функций, но до недавнего времени они были привязаны к .NET Framework, которая теперь является устаревшей платформой. С помощью современной .NET и ее пакета Windows Desktop Pack эти приложения теперь могут использовать все современные возможности .NET.

Структурирование проектов

Как вы должны структурировать свои проекты? До сих пор мы создавали небольшие отдельные консольные приложения, чтобы показать возможности языка или библиотеки. В оставшейся части книги мы создадим несколько проектов, используя различные технологии, которые будут работать вместе в целях создания единого решения.

В больших и сложных решениях может быть трудно ориентироваться во всем коде. Поэтому основной причиной структурирования проектов становится упрощение поиска компонентов. Хорошо иметь общее название для вашего решения или рабочей области, которое отражает приложение или решение.

Мы создадим несколько проектов для вымышленной компании *Northwind*. Мы назовем решение или рабочую область `PracticalApps` и будем использовать название `Northwind` в качестве префикса для всех названий проектов.

Существует множество способов структурирования и присвоения имен проектам и решениям, например, можно использовать иерархию папок, а также соглашение об именовании. Если вы работаете в команде, то убедитесь, что знаете, как это делает ваша команда.

Структурирование проектов в решении или рабочей области

Хорошо иметь соглашение об именовании проектов в решении или рабочей области, чтобы любой разработчик мог сразу определить, чем занимается каждый из них. Обычно используется тип проекта, например библиотека классов, консольное приложение, сайт и т. д., как показано в табл. 13.6.

Таблица 13.6. Примеры названий

Название	Описание
`Northwind.Common`	Проект библиотеки классов для общих типов, таких как интерфейсы, перечисления, классы, записи и структуры, используемых в нескольких проектах
`Northwind.Common.EntityModels`	Проект библиотеки классов для общих моделей сущностей EF Core. Сущностные модели часто используются как на стороне сервера, так и на стороне клиента, поэтому лучше разделить зависимости от конкретных поставщиков баз данных
`Northwind.Common.DataContext`	Проект библиотеки классов для контекста базы данных EF Core с зависимостями от конкретных поставщиков баз данных
`Northwind.Web`	Проект ASP.NET Core для простого сайта, который использует смесь статических HTML-файлов и динамических страниц Razor Pages
`Northwind.Razor.Component`	Проект библиотеки классов для страниц Razor Pages, используемый в нескольких проектах
`Northwind.Mvc`	Проект ASP.NET Core для сложного сайта, который использует паттерн MVC и может легче пройти модульное тестирование
`Northwind.WebApi`	Проект ASP.NET Core для сервиса HTTP API. Хороший выбор для интеграции с сайтами, поскольку для взаимодействия с сервисом можно использовать любую библиотеку JavaScript или Blazor
`Northwind.OData`	Проект ASP.NET Core для сервиса HTTP API, реализующего стандарт OData, чтобы клиент мог управлять запросами
`Northwind.GraphQL`	Проект ASP.NET Core для сервиса HTTP API, который реализует стандарт GraphQL, позволяющий клиенту управлять запросами
`Northwind.gRPC`	Проект ASP.NET Core для сервиса gRPC. Хороший выбор для интеграции с приложениями, созданными на любом языке и платформе, поскольку gRPC имеет широкую поддержку и отличается высокой эффективностью и производительностью
`Northwind.SignalR`	Проект ASP.NET Core для взаимодействия в режиме реальном времени
`Northwind.AzureFuncs`	Проект ASP.NET Core для реализации бессерверного наносервиса для размещения в функциях Azure
`Northwind.BlazorServer`	Проект сервера ASP.NET Core Blazor Server
`Northwind.BlazorWasm.Client`	Проект клиентской части ASP.NET Core Blazor WebAssembly
`Northwind.BlazorWasm.Server`	Проект серверной части ASP.NET Core Blazor WebAssembly
`Northwind.Maui`	Проект .NET MAUI для кросс-платформенного настольного/мобильного приложения
`Northwind.MauiBlazor`	Проект .NET MAUI для размещения компонентов Blazor с нативной интеграцией с операционной системой

Использование других шаблонов проектов

Когда вы устанавливаете .NET SDK, в комплект входит множество шаблонов проектов.

1. В командной строке или терминале введите следующую команду:

```
dotnet new --list
```

2. Вы увидите список установленных в настоящее время шаблонов, включая шаблоны для разработки рабочего стола Windows, если вы работаете в этой ОС (рис. 13.2).

```
Windows PowerShell
PS C:\Users\markj> dotnet new --list
Template Name                                  Short Name          Language    Tags
---------------------------------------------  ------------------  ----------  ---------------------
Console Application                            console             [C#],F#,VB  Common/Console
Class Library                                  classlib            [C#],F#,VB  Common/Library
WPF Application                                wpf                 [C#],VB     Common/WPF
WPF Class library                              wpflib              [C#],VB     Common/WPF
WPF Custom Control Library                     wpfcustomcontrollib [C#],VB     Common/WPF
WPF User Control Library                       wpfusercontrollib   [C#],VB     Common/WPF
Windows Forms App                              winforms            [C#],VB     Common/WinForms
Windows Forms Class Library                    winformslib         [C#],VB     Common/WinForms
Windows Forms Control Library                  winformscontrollib  [C#],VB     Common/WinForms
Worker Service                                 worker              [C#],F#     Common/Worker/Web
MSTest Test Project                            mstest              [C#],F#,VB  Test/MSTest
NUnit 3 Test Project                           nunit               [C#],F#,VB  Test/NUnit
NUnit 3 Test Item                              nunit-test          [C#],F#,VB  Test/NUnit
xUnit Test Project                             xunit               [C#],F#,VB  Test/xUnit
MVC ViewImports                                viewimports         [C#]        Web/ASP.NET
Razor Component                                razorcomponent      [C#]        Web/ASP.NET
MVC ViewStart                                  viewstart           [C#]        Web/ASP.NET
Razor Page                                     page                [C#]        Web/ASP.NET
Blazor Server App                              blazorserver        [C#]        Web/Blazor
Blazor WebAssembly App                         blazorwasm          [C#]        Web/Blazor/WebAssembly
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                           webapp              [C#]        Web/MVC/Razor Pages
ASP.NET Core with React.js and Redux           reactredux          [C#]        Web/MVC/SPA
ASP.NET Core with Angular                      angular             [C#]        Web/MVC/SPA
ASP.NET Core with React.js                     react               [C#]        Web/MVC/SPA
Razor Class Library                            razorclasslib       [C#]        Web/Razor/Library
ASP.NET Core Web API                           webapi              [C#],F#     Web/WebAPI
ASP.NET Core gRPC Service                      grpc                [C#]        Web/gRPC
dotnet gitignore file                          gitignore                       Config
global.json file                               globaljson                      Config
NuGet Config                                   nugetconfig                     Config
Dotnet local tool manifest file                tool-manifest                   Config
Web Config                                     webconfig                       Config
Solution File                                  sln                             Solution
Protocol Buffer File                           proto                           Web/gRPC
```

Рис. 13.2. Список шаблонов проектов dotnet

3. Обратите внимание на шаблоны проектов, связанных с веб, включая шаблоны для создания SPA с использованием Blazor, Angular и React. Но отсутствует еще одна общая библиотека JavaScript SPA: Vue.

Установка дополнительных пакетов шаблонов

Разработчики могут установить множество дополнительных пакетов шаблонов.

1. Запустите браузер и перейдите по адресу http://dotnetnew.azurewebsites.NET/.
2. Введите vue в текстовое поле и обратите внимание на список доступных шаблонов для Vue.js, включая один, опубликованный Microsoft (рис. 13.3).

Рис. 13.3. Шаблон проекта для Vue.js от Microsoft

3. Выберите вариант ASP.NET Core with Vue.js от Microsoft и обратите внимание на инструкции по установке и использованию этого шаблона:

```
dotnet new --install "Microsoft.AspNetCore.SpaTemplates"
dotnet new vue
```

4. Выберите вариант View other templates in this package (Посмотреть другие шаблоны в этом пакете) и обратите внимание, что, помимо шаблона проекта для Vue.js, в нем также есть шаблоны проектов для Aurelia и Knockout.js.

Разработка сущностной модели данных для базы данных Northwind

Реальные приложения обычно должны работать с данными в реляционной БД или другом хранилище данных. В этой главе мы определим сущностную модель данных на основе БД Northwind, хранящейся в SQL Server или SQLite. Такая модель будет использоваться в большинстве приложений, которые мы создадим в последующих главах.

Файлы сценариев Northwind4SQLServer.sql и Northwind4SQLite.sql отличаются друг от друга. Сценарий для SQL Server создает 13 таблиц, а также связанные с ними представления и хранимые процедуры. Сценарий для SQLite представляет собой упрощенную версию, которая создает только десять таблиц, поскольку SQLite не поддерживает столько функций. В основных проектах этой книги требуются только эти десять таблиц, поэтому вы сможете выполнить все представленные здесь задания с любой из баз данных.

Инструкции по установке SQL Server и SQLite описаны в главе 10. В ней же вы найдете инструкции по установке инструмента `dotnet-ef`, который можно будет использовать для создания сущностной модели из существующей базы данных.

Вам следует создать отдельный проект библиотеки классов для ваших сущностных моделей данных. Это позволяет упростить взаимодействие внутренних веб-серверов и клиентских компьютеров, мобильных устройств и клиентских приложений Blazor WebAssembly.

Разработка библиотеки классов для сущностных моделей с помощью SQLite

Теперь вы определите сущностные модели данных в библиотеке классов, чтобы их можно было повторно использовать в других типах проектов, включая модели приложений на стороне клиента. Если вы не применяете SQL Server, то вам нужно будет создать библиотеку классов для SQLite. Если вы используете SQL Server, то можете создать библиотеку классов как для SQLite, так и для SQL Server, а затем переключаться между ними по своему усмотрению.

Мы автоматически сгенерируем некоторые сущностные модели с помощью инструмента командной строки EF Core.

1. Откройте редактор кода и создайте рабочую область/решение `PracticalApps`.
2. Создайте проект библиотеки классов с такими настройками:
   1) шаблон проекта: Class Library/`classlib`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.Common.EntityModels.Sqlite`.
3. В проекте `Northwind.Common.EntityModels.Sqlite` добавьте ссылки на пакеты для поставщика базы данных SQLite и поддержки EF Core во время разработки:

```
<ItemGroup>
  <PackageReference
```

```
    Include="Microsoft.EntityFrameworkCore.Sqlite"
    Version="6.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Design"
    Version="6.0.0">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
    buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

4. Удалите файл `Class1.cs`.
5. Соберите проект.
6. Создайте файл `Northwind.db` для SQLite, скопировав файл `Northwind4SQLite.sql` в папку PracticalApps, а затем в командной строке или в терминале введите следующую команду:

```
Sqlite3 Northwind.db -init Northwind4SQLite.sql
```

7. Вам придется немного подождать, так как этой команде может потребоваться некоторое время на создание структуры базы данных:

```
-- Loading resources from Northwind4SQLite.sql
SQLite version 3.35.5 2021-04-19 14:49:49
Enter ".help" for usage hints.
Sqlite>
```

8. Нажмите сочетание клавиш Ctrl+C в Windows или Cmd+D в macOS для выхода из командного режима SQLite.
9. Откройте командную строку или терминал для папки Northwind.Common.EntityModels.Sqlite.
10. В командной строке создайте модели классов сущностей для всех таблиц:

```
dotnet ef dbcontext scaffold "Filename=../Northwind.db" Microsoft.
EntityFrameworkCore.Sqlite --namespace Packt.Shared --data-annotations
```

Обратите внимание на следующие моменты:

- команда для выполнения — `dbcontext scaffold`;
- строки подключения — `"Filename=../Northwind.db"`;
- поставщик базы данных — `Microsoft.EntityFrameworkCore.Sqlite`;
- пространством имен является `--namespace Packt.Shared`;
- для использования аннотаций данных, а также API Fluent применяется `--data-annotations`.

11. Обратите внимание на сообщения и предупреждения сборки:

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string,
you should move it out of source code. You can avoid scaffolding the
connection string by using the Name= syntax to read it from configuration
- see https://go.microsoft.com/fwlink/?linkid=2131148. For more
guidance on storing connection strings, see http://go.microsoft.com/
fwlink/?LinkId=723263.
```

Улучшение сопоставления классов и таблиц

Инструмент командной строки `dotnet-ef` генерирует разный код для SQL Server и SQLite, поскольку они поддерживают разные уровни функциональности.

Например, текстовые столбцы SQL Server могут иметь ограничения на количество символов. SQLite этого не поддерживает. Поэтому `dotnet-ef` будет генерировать атрибуты проверки, чтобы убедиться, что свойства `string` ограничены заданным количеством символов для SQL Server, но не для SQLite:

```
// код, генерируемый поставщиком базы данных SQLite
[Column(TypeName = "nvarchar (15)")]
public string CategoryName { get; set; } = null!;

// код, генерируемый поставщиком базы данных SQL Server
[StringLength(15)]
public string CategoryName { get; set; } = null!;
```

Ни один из поставщиков баз данных не будет отмечать свойства `string`, не являющиеся нулевыми, как требуемые:

```
// отсутствие проверки выполнения на null
public string CategoryName { get; set; } = null!;
// свойство, допускающее значение null
public string? Description { get; set; }

// добавление атрибута для выполнения проверки
[Required]
public string CategoryName { get; set; } = null!;
```

Чтобы улучшить отображение сущностной модели и правил проверки для SQLite, мы внесем небольшие изменения.

1. Откройте файл `Customer.cs` и добавьте регулярное выражение для проверки значения его первичного ключа, чтобы разрешить использование только заглавных западных символов, как показано ниже (выделено жирным шрифтом):

   ```
   [Key]
   [Column(TypeName = "nchar (5)")]
   ```

```
[RegularExpression("[A-Z]{5}")]
Public string CustomerId { get; set; }
```

2. Активируйте функцию поиска и замены в редакторе кода (в программе Visual Studio 2022 выберите команду меню Edit ▸ Find and Replace ▸ Quick Replace (Редактирование ▸ Поиск и замена ▸ Быстрая замена)), включите опцию Use Regular Expressions (Использовать регулярные выражения), а затем введите регулярное выражение в поле поиска:

```
\[Column\(TypeName = "(nchar|nvarchar) \((.*)\)"\)\]
```

3. В поле замены введите заменяющее регулярное выражение:

```
$&\n    [StringLength($2)]
```

После символа новой строки \n я ввел четыре символа пробела для правильного отступа в моей системе, которая использует два символа пробела на уровень отступа. Вы можете вставить столько символов, сколько захотите.

4. Настройте поиск и замену для поиска файлов в текущем проекте.
5. Выполните поиск и замену, чтобы заменить все (рис. 13.4).

Рис. 13.4. Поиск и замена всех совпадений с помощью регулярных выражений в программе Visual Studio 2022

6. Измените любые свойства даты/времени, например, в файле `Employee.cs`, чтобы использовать тип данных `DateTime`, допускающий значение `NULL`, вместо массива байтов, как показано ниже:

```
// до
[Column(TypeName = "datetime")]
public byte[] BirthDate { get; set; }
```

```
// после
[Column(TypeName = "datetime")]
Public DateTime? BirthDate { get; set; }
```

Используйте функцию поиска в вашем редакторе кода для поиска "datetime", чтобы найти все свойства, которые нужно изменить.

7. Измените любые свойства `money`, например, в файле `Order.cs`, чтобы использовать десятичное число, допускающее значение `NULL`, вместо массива байтов, как показано ниже:

```
// до
[Column(TypeName = "money")]
public byte[] Freight { get; set; }

// после
[Column(TypeName = "money")]
public decimal? Freight { get; set; }
```

Используйте функцию поиска в вашем редакторе кода для поиска "money", чтобы найти все свойства, которые нужно изменить.

8. Измените любые свойства `bit`, например, в файле `Product.cs`, чтобы использовать логическое значение вместо массива байтов:

```
// до
[Column(TypeName = "bit")]
public byte[] Discontinued { get; set; } = null!;

// после
[Column(TypeName = "bit")]
Public bool Discontinued { get; set; }
```

Используйте функцию поиска в вашем редакторе кода для поиска "bit", чтобы найти все свойства, которые нужно изменить.

9. В файле `Category.cs` сделайте свойство `CategoryId` равным значению `int`:

```
[Key]
public int CategoryId { get; set; }
```

10. В файле `Category.cs` сделайте свойство `CategoryName` обязательным:

```
[Required]
[Column(TypeName = "nvarchar (15)")]
[StringLength(15)]
public string CategoryName { get; set; }
```

11. В файле `Customer.cs` сделайте свойство `CompanyName` обязательным:

```
[Required]
[Column(TypeName = "nvarchar (40)")]
[StringLength(40)]
public string CompanyName { get; set; }
```

12. В файле `Employee.cs` присвойте свойству `EmployeeId` значение `int`, а не `long`.
13. В файле `Employee.cs` сделайте свойства `FirstName` и `LastName` обязательными.
14. В файле `Employee.cs` присвойте свойству `ReportsTo` значение `int?` вместо `long?`.
15. В файле `EmployeeTerritory.cs` присвойте свойству `EmployeeId` значение `int` вместо `long`.
16. В файле `EmployeeTerritory.cs` сделайте свойство `TerritoryId` обязательным.
17. В файле `Order.cs` присвойте свойству `OrderId` значение `int` вместо `long`.
18. В файле `Order.cs` дополните свойство `CustomerId` регулярным выражением, чтобы обеспечить использование пяти символов в верхнем регистре.
19. В файле `Order.cs` присвойте свойству `EmployeeId` значение `int?` вместо `long?`.
20. В файле `Order.cs` присвойте свойству `ShipVia` значение `int?` вместо `long?`
21. В файле `OrderDetail.cs` присвойте свойству `OrderId` значение `int` вместо `long`.
22. В файле `OrderDetail.cs` присвойте свойству `ProductId` значение `int` вместо `long`.
23. В файле `OrderDetail.cs` присвойте свойству `Quantity` значение `short` вместо `long`.
24. В файле `Product.cs` присвойте свойству `ProductId` значение `int` вместо `long`.
25. В файле `Product.cs` сделайте свойство `ProductName` обязательным.
26. В файле `Product.cs` присвойте свойствам `SupplierId` и `CategoryId` значение `int?` вместо `long?`.
27. В файле `Product.cs` присвойте свойствам `UnitsInStock`, `UnitsOnOrder` и `ReorderLevel` значение `short?` вместо `long?`.
28. В файле `Shipper.cs` присвойте свойству `ShipperId` значение `int` вместо `long`.
29. В файле `Shipper.cs` сделайте свойство `CompanyName` обязательным.
30. В файле `Supplier.cs` присвойте свойству `SupplierId` значение `int`, а не `long`.
31. В файле `Supplier.cs` сделайте свойство `CompanyName` обязательным.
32. В файле `Territory.cs` присвойте свойству `RegionId` значение `int`, а не `long`.
33. В файле `Territory.cs` сделайте свойства `TerritoryId` и `TerritoryDescription` обязательными.

Теперь, когда у нас есть библиотека классов для классов сущностей, мы можем создать библиотеку классов для контекста базы данных.

Создание библиотеки классов для контекста базы данных Northwind

Теперь вы определите библиотеку классов контекста базы данных.

1. Создайте проект библиотеки классов в рабочей области/решении с такими настройками:
   1) шаблон проекта: Class Library/`classlib`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.Common.DataContext.Sqlite`.
2. В программе Visual Studio настройте стартовый проект для решения в соответствии с текущим выбором.
3. В программе Visual Studio Code выберите `Northwind.Common.DataContext.Sqlite` в качестве активного проекта OmniSharp.
4. В проекте `Northwind.Common.DataContext.Sqlite` добавьте ссылку на проект `Northwind.Common.EntityModels.Sqlite` и добавьте ссылку на пакет для поставщика данных EF Core для SQLite:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SQLite"
    Version="6.0.0" />
</ItemGroup>
<ItemGroup>
  <ProjectReference Include=
    "..\Northwind.Common.EntityModels.Sqlite\Northwind.Common
.EntityModels.Sqlite.csproj" />
</ItemGroup>
```

Путь к ссылке на проект не должен содержать разрыва строки в вашем файле проекта.

5. В проекте `Northwind.Common.DataContext.Sqlite` удалите файл класса `Class1.cs`.
6. Соберите проект `Northwind.Common.DataContext.Sqlite`.
7. Переместите файл `NorthwindContext.cs` из проекта/папки `Northwind.Common.EntityModels.Sqlite` в проект/папку `Northwind.Common.DataContext.Sqlite`.

Если вы перетаскиваете файл между проектами на панели Solution Explorer (Проводник решений) программы Visual Studio, то он будет скопирован. Если во время перетаскивания удерживать нажатой клавишу Shift, то файл будет перемещен. Если вы перетаскиваете файл между проектами на панели EXPLORER (Проводник) программы Visual Studio Code, то он будет перемещен. Если во время перетаскивания удерживать нажатой клавишу Ctrl, то файл будет скопирован.

8. В файле `NorthwindContext.cs` в методе `OnConfiguring` удалите предупреждение компилятора `#warning` о строке подключения.

Мы будем переопределять строку подключения к базе данных по умолчанию в любых проектах, таких как сайты, которые должны работать с базой данных Northwind, поэтому класс, производный от DbContext, должен иметь конструктор с параметром DbContextOptions, чтобы это работало:

```
public NorthwindContext(DbContextOptions<NorthwindConte
xt> options)
  : base(options)
{
}
```

9. В методе `OnModelCreating` удалите все операторы Fluent API, которые вызывают метод `ValueGeneratedNever`, чтобы настроить свойства первичного ключа, такие как `SupplierId`, чтобы они никогда не генерировали значение автоматически, или вызовите метод `HasDefaultValueSql`:

```
modelBuilder.Entity<Supplier>(entity =>
{
  entity.Property(e => e.SupplierId).ValueGeneratedNever();
});
```

Если мы не удалим конфигурацию, как указано в предыдущих операторах, то при добавлении новых поставщиков значение SupplierId всегда будет равно 0 и мы сможем добавить только один поставщик с таким значением, а все остальные попытки вызовут исключение.

10. Для сущности `Product` сообщите SQLite, что `UnitPrice` может быть преобразована из `decimal` (десятичной) в `double` (двойную). Метод `OnModelCreating` теперь должен быть значительно упрощен:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<OrderDetail>(entity =>
  {
    entity.HasKey(e => new { e.OrderId, e.ProductId });

    entity.HasOne(d => d.Order)
      .WithMany(p => p.OrderDetails)
      .HasForeignKey(d => d.OrderId)
      .OnDelete(DeleteBehavior.ClientSetNull);
```

```
    entity.HasOne(d => d.Product)
      .WithMany(p => p.OrderDetails)
      .HasForeignKey(d => d.ProductId)
      .OnDelete(DeleteBehavior.ClientSetNull);
  });

  modelBuilder.Entity<Product>()
    .Property(product => product.UnitPrice)
    .HasConversion<double>();

  OnModelCreatingPartial(modelBuilder);
}
```

11. Добавьте класс `NorthwindContextExtensions.cs` и измените его содержимое, чтобы определить метод расширения, который добавляет контекст базы данных Northwind в коллекцию сервисов зависимостей:

```
using Microsoft.EntityFrameworkCore; // UseSqlite
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace Packt.Shared;

public static class NorthwindContextExtensions
{
  /// <summary>
  /// Adds NorthwindContext to the specified IserviceCollection. Uses the
      Sqlite database provider.
  /// </summary>
  /// <param name="services"></param>
  /// <param name="relativePath">Set to override the default
      of ".."</param>
  /// <returns>An IServiceCollection that can be used to add more
      services.</returns>
  public static IServiceCollection AddNorthwindContext(
    this IServiceCollection services, string relativePath = "..")
  {
    string databasePath = Path.Combine(relativePath, "Northwind.db");

    services.AddDbContext<NorthwindContext>(options =>
      options.UseSqlite($"Data Source={databasePath}")
    );

    return services;
  }
}
```

12. Соберите две библиотеки классов и исправьте все ошибки компилятора.

Создание библиотеки классов для сущностных моделей с помощью SQL Server

Чтобы использовать SQL Server, вам не нужно ничего делать, если вы уже настроили базу данных Northwind в главе 10. Но теперь вы будете создавать сущностные модели с помощью инструмента `dotnet-ef`.

1. Откройте редактор кода и создайте рабочую область/решение `PracticalApps`.
2. Создайте проект библиотеки классов с такими настройками:
   1) шаблон проекта: Class Library/`classlib`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.Common.EntityModels.SqlServer`.
3. В проекте `Northwind.Common.EntityModels.SqlServer` добавьте ссылки на пакеты для поставщика базы данных SQL Server и поддержки EF Core во время разработки:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="6.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Design"
    Version="6.0.0">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
    buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

4. Удалите файл `Class1.cs`.
5. Соберите проект.
6. Откройте в командной строке или в терминале папку Northwind.Common.EntityModels.SqlServer.
7. В командной строке создайте модели классов сущностей для всех таблиц, как показано в следующих командах:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial
Catalog=Northwind;Integrated Security=true;" Microsoft.
EntityFrameworkCore.SqlServer --namespace Packt.Shared --data-annotations
```

Обратите внимание на следующие моменты:

- команда для выполнения — `dbcontext scaffold`;
- строки подключения — `"Data Source=.;Initial Catalog=Northwind;Integrated Security=true;"`;
- поставщик базы данных — `Microsoft.EntityFrameworkCore.SqlServer`;
- пространством имен является `--namespace Packt.Shared`;
- для использования аннотаций данных, а также API Fluent применяется `--data-annotations`.

8. В файле `Customer.cs` добавьте регулярное выражение для проверки значения его первичного ключа, чтобы разрешить только верхний регистр западных символов:

```
[Key]
[StringLength(5)]
[RegularExpression("[A-Z]{5}")]
public string CustomerId { get; set; } = null!;
```

9. В файле `Customer.cs` сделайте свойства `CustomerId` и `CompanyName` обязательными.
10. Создайте проект библиотеки классов в рабочей области/решении с такими настройками:
    1) шаблон проекта: Class Library/`classlib`;
    2) файл и папка рабочей области/решения: `PracticalApps`;
    3) файл и папка проекта: `Northwind.Common.DataContext.SqlServer`.
11. В программе Visual Studio Code выберите `Northwind.Common.DataContext.SqlServer` в качестве активного проекта OmniSharp.
12. В проекте `Northwind.Common.DataContext.SqlServer` добавьте ссылку на проект `Northwind.Common.EntityModels.SqlServer` и добавьте ссылку на пакет для поставщика данных EF Core для SQL Server:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="6.0.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include=
    "..\Northwind.Common.EntityModels.SqlServer\Northwind.Common
    .EntityModels.SqlServer.csproj" />
</ItemGroup>
```

13. В проекте `Northwind.Common.DataContext.SqlServer` удалите файл класса `Class1.cs`.
14. Соберите проект `Northwind.Common.DataContext.SqlServer`.
15. Переместите файл `NorthwindContext.cs` из проекта/папки `Northwind.Common.EntityModels.SqlServer` в проект/папку `Northwind.Common.DataContext.SqlServer`.
16. В файле `NorthwindContext.cs` удалите предупреждение компилятора о строке подключения.
17. Добавьте класс `NorthwindContextExtensions.cs` и измените его содержимое, чтобы определить метод расширения, который добавляет контекст базы данных Northwind в коллекцию сервисов зависимостей:

```
using Microsoft.EntityFrameworkCore; // UseSqlServer
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace Packt.Shared;

public static class NorthwindContextExtensions
{
  /// <summary>
  /// Adds NorthwindContext to the specified IServiceCollection. Uses the
      SqlServer database provider.
  /// </summary>
  /// <param name="services"></param>
  /// <param name="connectionString">Set to override the default.</param>
  /// <returns>An IServiceCollection that can be used to add more
      services.</returns>
  public static IServiceCollection AddNorthwindContext(
    this IServiceCollection services, string connectionString =
      "Data Source=.;Initial Catalog=Northwind;"
      + "Integrated Security=true;MultipleActiveResultsets=true;")
  {
    services.AddDbContext<NorthwindContext>(options =>
      options.UseSqlServer(connectionString));

    return services;
  }
}
```

18. Соберите две библиотеки классов и исправьте все ошибки компилятора.

Мы предоставили дополнительные аргументы для метода AddNorthwindContext, чтобы можно было переопределить жестко заданный путь к имени файла базы данных SQLite или строку подключения к базе данных SQL Server. Это позволит коду быть более гибким, например загружать эти значения из файла конфигурации.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 13.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. NET 6 является кросс-платформенной. Приложения Windows Forms и WPF могут работать на .NET 6. Могут ли они работать на macOS и Linux?
2. Как приложение Windows Forms определяет свой пользовательский интерфейс и почему это может быть проблемой?
3. Как приложение WPF или UWP может определять свой пользовательский интерфейс и почему это хорошо для разработчиков?

Упражнение 13.2. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-13---introducing-practical-applications-of-c-and-net, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы познакомились с отдельными моделями приложений и рабочими нагрузками, которые можно использовать для создания практических приложений с помощью C# и .NET.

Вы создали от двух до четырех библиотек классов для определения сущностных моделей данных для работы с базой данных Northwind, используя либо SQLite, либо SQL Server, либо и то и другое.

В следующих шести главах вы получите информацию о том, как разработать:

- простые сайты с помощью статических HTML-страниц и динамических веб-страниц Razor;
- сложные веб-приложения, используя паттерн проектирования Model-View-Controller (MVC);
- веб-сервисы, пригодные для вызова любой платформой, которая может выполнять HTTP-запрос, и клиентские сайты, вызывающие эти веб-сервисы;

- компоненты пользовательского интерфейса Blazor, которые могут размещаться на веб-сервере, в браузере или на гибридных веб-нативных мобильных и настольных приложениях;
- сервисы, реализующие удаленные вызовы процедур с помощью gRPC;
- сервисы, реализующие взаимодействие в режиме реального времени с помощью SignalR;
- сервисы, обеспечивающие простой и гибкий доступ к модели EF Core;
- бессерверные наносервисы, размещенные на платформе Azure Functions;
- кросс-платформенные нативные мобильные и настольные приложения с помощью .NET MAUI.

14 Разработка сайтов с помощью ASP.NET Core Razor Pages

Данная глава посвящена тому, как создавать сайты с современной HTTP-архитектурой на стороне сервера с помощью Microsoft ASP.NET Core. Вы научитесь создавать простые сайты, используя функцию Razor Pages платформы ASP.NET Core, представленную в ASP.NET Core 2.0, и функцию библиотеки классов Razor, представленную в ASP.NET Core 2.1.

В этой главе:

- веб-разработка;
- обзор ASP.NET Core;
- функция Razor Pages платформы ASP.NET Core;
- использование Entity Framework Core совместно с ASP.NET Core;
- применение библиотек классов Razor;
- настройка сервисов и конвейера HTTP-запросов.

Веб-разработка

Разработка для Интернета означает разработку с помощью *протокола передачи гипертекста* (Hypertext Transfer Protocol, HTTP), поэтому мы начнем с обзора этой важной основополагающей технологии.

Протокол передачи гипертекста

Для связи с веб-сервером клиент, также известный как *пользовательский агент*, совершает вызовы через Интернет с помощью протокола HTTP. Это своего рода технический фундамент Всемирной паутины. Таким образом, говоря о сайтах 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, идентифицируемого по *уникальному адресу ресурса* (Uniform Resource Locator, URL), и сервер отправляет HTTP-ответ, как показано на рис. 14.1.

Рис. 14.1. HTTP-запрос и ответ

Для записи запросов и ответов можно использовать Google Chrome и другие браузеры.

Браузер Google Chrome доступен для большего количества операционных систем, чем любой другой, и содержит эффективные встроенные инструменты разработчика, поэтому лучше всего подойдет для тестирования сайтов. Всегда тестируйте свое веб-приложение в Google Chrome и по крайней мере в двух других браузерах, например Firefox и Safari для macOS и iPhone. Microsoft Edge перешел от использования собственного механизма визуализации Microsoft к применению Chromium в 2019 году, поэтому проводить тестирование с его помощью не обязательно. Если браузер Microsoft Internet Explorer и используется, то обычно только внутри организаций и во внутренней электронной сети.

Состав URL

URL состоит из нескольких компонентов:

- *схема* — http (открытый текст) или https (зашифрованный);
- *домен* — для опубликованного сайта или сервиса *домен верхнего уровня* (top-level domain, TLD) может быть example.com. У вас могут быть поддомены, такие как www, jobs или extranet. Во время разработки вы обычно используете localhost для всех сайтов и сервисов;

- *номер порта* — для опубликованного сайта или сервиса используется 80 для http, 443 для https. Эти номера портов обычно определяются согласно схеме. Во время разработки обычно используются другие номера портов, например 5000, 5001 и т. д., чтобы различать сайты и сервисы, которые используют общий домен localhost;
- *путь* — относительный путь к ресурсу, например /customers/germany;
- *строка запроса* — способ передачи значений параметров, например ?country=Germany&searchtext=shoes;
- *фрагмент* — ссылка на элемент веб-страницы с использованием его id, например #toc.

Назначение номеров портов для проектов в этой книге

В этой книге используется домен localhost для всех сайтов и сервисов, поэтому с помощью номеров портов мы будем различать проекты, когда несколько из них должны работать одновременно (табл. 14.1).

Таблица 14.1. Проекты и номера портов

Проект	Описание	Номер (-а) порта
`Northwind.Web`	Сайт ASP.NET Core Razor Pages	`5000` HTTP, `5001` HTTPS
`Northwind.Mvc`	Сайт ASP.NET Core MVC	`5000` HTTP, `5001` HTTPS
`Northwind.WebApi`	Сервис ASP.NET Core Web API	`5002` HTTPS, `5008` HTTP
`Minimal.WebApi`	ASP.NET Core Web API (минимальный)	`5003` HTTPS
`Northwind.OData`	Сервис ASP.NET Core OData	`5004` HTTPS
`Northwind.GraphQL`	Сервис ASP.NET Core GraphQL	`5005` HTTPS
`Northwind.gRPC`	Сервис ASP.NET Core gRPC	`5006` HTTPS
`Northwind.AzureFuncs`	Наносервис платформы Azure Functions	`7071` HTTP

Использование браузера Google Chrome для выполнения HTTP-запросов

Рассмотрим применение браузера Google Chrome для выполнения HTTP-запросов.

1. Запустите браузер Google Chrome.
2. Выберите команду меню More tools ▸ Developer tools (Дополнительные инструменты ▸ Инструменты разработчика).

3. Перейдите на вкладку Network (Сеть). Браузер Google Chrome должен немедленно начать запись сетевого трафика между вашим браузером и любыми веб-серверами (обратите внимание на красный кружок), как показано на рис. 14.2.

Рис. 14.2. Фиксация сетевого трафика на панели Developer tools (Инструменты разработчика) браузера Google Chrome

4. В адресной строке браузера введите адрес сайта Microsoft, содержащего обучающие материалы по ASP.NET, как показано в следующем URL: https://dotnet.microsoft.com/learn/aspnet.
5. На панели Developer tools (Инструменты разработчика) в списке записанных запросов прокрутите вверх и выделите первую запись, строку, где в столбце Type (Тип) указано значение Document (Документ), как показано на рис. 14.3.

Рис. 14.3. Записанные запросы на панели Developer tools (Инструменты разработчика)

6. В правой части панели щелкните кнопкой мыши на вкладке Headers (Заголовки). Вы должны увидеть подробную информацию в разделах Request Headers (Заголовки запросов) и Response Headers (Заголовки ответов), как показано на рис. 14.4.

Рис. 14.4. Разделы Request Headers (Заголовки запросов) и Response Headers (Заголовки ответов)

Обратите внимание на следующие моменты.

- Применен *метод запроса* — `GET`. В число других HTTP-методов, которые вы можете здесь увидеть, входят `POST`, `PUT`, `DELETE`, `HEAD` и `PATCH`.
- Получен *код состояния* — `200 OK`. Это значит, сервер обнаружил ресурс, запрошенный браузером, и вернул его в теле ответа. Другие коды состояния, которые вы можете увидеть в ответе на запрос `GET`, включают `301 Moved Permanently` (перемещено навсегда), `400 Bad Request` (плохой, неверный запрос), `401 Unauthorized` (не авторизован) и `404 Not Found` (не найден).
- *Заголовки запроса*, отправляемые браузером на веб-сервер, включают в себя следующие параметры:
  - *accept*, перечисляющий форматы ресурсов, принимаемые браузером. В этом случае браузер сообщает, что поддерживает форматы HTML, XHTML, XML и некоторые форматы изображений, но будет принимать

все другие файлы (`*/*`). Веса, также известные как качественные значения, по умолчанию равны `1.0`. XML указан с качественным значением `0.9`, поэтому является менее предпочтительным, чем HTML или XHTML. Все другие типы файлов имеют качественное значение `0.8`, поэтому наименее предпочтительны;

  - *accept-encoding*, в котором перечислены алгоритмы сжатия, поддерживаемые браузером, в данном случае это GZIP, DEFLATE и Brotli;
  - *accept-language*, перечисляющий человеческие языки, которые браузер предпочитает использовать в контенте. В этом случае английский язык США имеет качественное значение по умолчанию `1.0`, любой диалект английского языка имеет явно заданное качественное значение `0.9`, а любой диалект шведского языка имеет явно заданное качественное значение `0.8`.

- *Заголовки ответа*, элемент content-encoding говорит о том, что сервер отправил обратно ответ веб-страницы HTML, сжатый с помощью алгоритма GZIP, поскольку знает, что клиент может распаковать этот формат. (Этого не видно на рис. 14.4, так как не хватает места для разворачивания раздела Response Headers).

7. Закройте браузер Google Chrome.

Технологии клиентской веб-разработки

При создании веб-приложений разработчик должен знать не только C# и .NET. На стороне клиента (то есть в браузере) вы будете применять комбинацию следующих технологий:

- *HTML5* — используется для определения контента и структуры веб-страницы;
- *CSS3* — служит для настройки стилей, применяемых к элементам на веб-странице;
- *JavaScript* — используется для кодирования любой бизнес-логики, необходимой на веб-странице, например проверки правильности ввода формы или выполнения вызовов веб-сервиса для получения дополнительных данных, необходимых веб-странице.

HTML5, CSS3 и JavaScript — фундаментальные компоненты фронтенд-веб-разработки, однако существует множество дополнительных технологий, которые могут повысить ее эффективность. К их числу относятся Bootstap и CSS-препроцессоры, такие как SASS и LESS для стилей, язык TypeScript от Microsoft для написания более надежного кода и библиотеки JavaScript, такие как jQuery, Angular, React и Vue. Все перечисленные высокоуровневые технологии в конечном итоге переводятся или компилируются в три основные технологии, поэтому работают во всех современных браузерах.

В рамках процесса сборки и развертывания вы, вероятно, будете использовать такие технологии, как Node.js, Node Package Manager (NPM, стандартный менеджер пакетов) и Yarn — менеджеры пакетов на стороне клиента; а также Webpack — популярный модульный компоновщик, инструмент для компиляции, преобразования и объединения исходных файлов сайтов.

Обзор ASP.NET Core

Технологии компании Microsoft, используемые для разработки сайтов и сервисов, эволюционировали на протяжении многих лет. Microsoft ASP.NET Core — их естественное продолжение.

- *ASP* (Active Server Pages — «активные серверные страницы») — технология, реализованная в 1996 году и ставшая первой попыткой Microsoft создать платформу для динамического выполнения кода сайта на стороне сервера. ASP-файлы содержат смесь HTML и выполняемого на сервере кода на языке VBScript.
- *ASP.NET Web Forms* — эта технология была представлена в 2002 году вместе с платформой .NET Framework и предназначена для того, чтобы разработчики, незнакомые с веб-программированием, но знающие такие языки, как Visual Basic, могли быстро создавать сайты, перетаскивая и удаляя визуальные компоненты и записывая событийно-управляемый код на языке Visual Basic или C#. В новых веб-проектах .NET Framework следует отдавать предпочтение не Web Forms, а ASP.NET MVC.
- *WCF* (Windows Communication Foundation) — платформа, выпущенная в 2006 году и позволяющая разработчикам создавать сервисы SOAP и REST. SOAP — эффективная, но сложная технология, поэтому ее следует избегать, если нет необходимости использовать расширенные функции, такие как распределенные транзакции и сложные топологии обмена сообщениями.
- *ASP.NET MVC* — платформа, выпущенная в 2009 году. Предназначалась для четкого разделения задач веб-разработчиков между *моделями*, временно хранящими данные, *представлениями*, которые визуализируют эти данные в различном виде в пользовательском интерфейсе, и *контроллерами*, осуществляющими выборку модели и передачу ее представлению. Такое разделение положительно сказывается при многократном использовании и модульном тестировании.
- *ASP.NET Web API* — инструмент, выпущенный в 2012 году и позволяющий разработчикам создавать HTTP-сервисы (также известные как REST), которые масштабируются проще и лучше, чем SOAP-сервисы.
- *ASP.NET SignalR* — технология, выпущенная в 2013 году и реализующая взаимодействие в режиме реальном времени на сайтах за счет абстрагирования

вспомогательных технологий и методов, таких как WebSockets и Long Polling. Позволяет использовать такие функции сайта, как чат в режиме реального времени или обновление чувствительных ко времени данных (например, биржевых курсов) в самых разных браузерах, даже если они не поддерживают базовую технологию, такую как WebSockets.

- *ASP.NET Core* — платформа, выпущенная в 2016 году. Она сочетает в себе современные реализации технологий .NET Framework, таких как MVC, Web-API и SignalR с более новыми технологиями, такими как Razor Pages, gRPC и Blazor, работающими на современной .NET. Благодаря этому является кросс-платформенной. Содержит множество шаблонов проектов, которые помогут вам начать работу с поддерживаемыми технологиями.

Для разработки сайтов и сервисов следует выбирать платформу ASP.NET Core, поскольку она включает в себя современные и кросс-платформенные веб-технологии.

Версии ASP.NET Core от 2.0 до 2.2 могут работать на версии .NET Framework 4.6.1 или более поздней (только для операционной системы Windows), а также на .NET Core 3.0 или более поздней версии (кросс-платформенной). ASP.NET Core 6.0 поддерживает лишь .NET 6.0.

Классический ASP.NET против современного ASP.NET Core

До сих пор ASP.NET была построена на основе крупной сборки `System.Web.dll` на платформе .NET Framework и была тесно связана с веб-сервером Microsoft, предназначенным только для операционной системы Windows, известным как *Internet Information Services (IIS)*. За прошедшие годы в сборке скопилось множество функций, не подходящих для современной кросс-платформенной разработки.

ASP.NET Core — результат крупной переработки кода ASP.NET. Устранена зависимость от сборки `System.Web.dll` и IIS, а платформа реализована в виде модульных легковесных пакетов, как и остальная часть современной .NET. Использование IIS в качестве веб-сервера по-прежнему поддерживается ASP.NET Core, но есть и лучший вариант.

Вы можете разрабатывать и запускать приложения ASP.NET Core кросс-платформенно в операционной системе Windows, macOS и Linux. Компания Microsoft даже разработала кросс-платформенный, суперпроизводительный веб-сервер *Kestrel*, а исходный код всего стека полностью открыт.

В проектах версии ASP.NET Core 2.2 или более поздних по умолчанию используется новая модель внутрипроцессного хостинга. Это увеличивает производи-

тельность на 400 % при хостинге в Microsoft IIS. Однако Microsoft по-прежнему рекомендует использовать Kestrel для еще большего повышения производительности.

Создание пустого проекта ASP.NET Core

Мы создадим проект ASP.NET Core, отображающий список поставщиков из базы данных Northwind.

Инструмент `dotnet` содержит множество шаблонов проектов, выполняющих за вас значительную часть работы. Однако зачастую трудно определить, какой из них лучше всего подходит для конкретной ситуации. И чтобы разобраться во всем этом, мы начнем с пустого шаблона проекта сайта и затем пошагово добавим функции.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: ASP.NET Core Empty/`web`;
   2) язык: `C#`;
   3) файл и папка рабочей области/решения: `PracticalApps`;
   4) файл и папка проекта: `Northwind.Web`;
   5) в программе Visual Studio 2022 оставьте все остальные параметры по умолчанию, например, флажок Configure for HTTPS (Настроить для HTTPS) установлен, а Enable Docker (Включить Docker) снят.
2. В программе Visual Studio Code выберите `Northwind.Web` в качестве активного проекта OmniSharp.
3. Соберите проект `Northwind.Web`.
4. Откройте файл `NorthwindWeb.csproj` и обратите внимание, что проект похож на библиотеку классов, за исключением того, что SDK — `Microsoft.NET.Sdk.Web`, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

5. Если вы используете Visual Studio 2022, то на панели Solution Explorer (Проводник решений) установите флажок Show All Files (Показывать все файлы).
6. Раскройте папку obj, папку Debug, папку net6.0, выберите файл `Northwind.Web.GlobalUsings.g.cs` и обратите внимание, что неявно импортированные пространства имен включают все пространства имен для консольного приложения

или библиотеки классов, а также некоторые ASP.NET Core, такие как `Microsoft.AspNetCore.Builder`:

```
// <autogenerated />
global using global::Microsoft.AspNetCore.Builder;
global using global::Microsoft.AspNetCore.Hosting;
global using global::Microsoft.AspNetCore.Http;
global using global::Microsoft.AspNetCore.Routing;
global using global::Microsoft.Extensions.Configuration;
global using global::Microsoft.Extensions.DependencyInjection;
global using global::Microsoft.Extensions.Hosting;
global using global::Microsoft.Extensions.Logging;
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.NET.Http;
global using global::System.NET.Http.Json;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

7. Сверните папку obj.
8. Откройте файл `Program.cs` и обратите внимание на следующие моменты:
   - проект ASP.NET Core похож на консольное приложение верхнего уровня, точкой входа которого является скрытый метод `Main`, аргумент которого передается с помощью имени `args`;
   - он вызывает `WebApplication.CreateBuilder`, который создает хост (узел) для сайта, используя параметры по умолчанию для веб-хоста, который затем собирается;
   - сайт будет отвечать на все запросы HTTP `GET` простым текстом: `Hello World!`;
   - вызов метода `Run` является блокирующим, поэтому скрытый метод `Main` не возвращается до тех пор, пока веб-сервер не прекратит работу:

     ```
     var builder = WebApplication.CreateBuilder(args);
     var app = builder.Build();

     app.MapGet("/", () => "Hello World!");

     app.Run();
     ```

9. В конце файла добавьте оператор для записи сообщения в консоль после вызова метода `Run` и, следовательно, после остановки веб-сервера:

   ```
   app.Run();

   Console.WriteLine("This executes after the web server has stopped!");
   ```

Тестирование и защита сайта

Проверим функциональность пустого проекта сайта ASP.NET Core. Кроме того, чтобы обеспечить конфиденциальность, мы включим шифрование всего трафика между браузером и веб-сервером, переключившись с HTTP на HTTPS. HTTPS — это безопасная зашифрованная версия HTTP.

1. В программе Visual Studio:
   1) убедитесь, что на панели инструментов выбран вариант Northwind.Web, а не IIS Express или WSL, и переключите Web Browser (Microsoft Edge) на Google Chrome, как показано на рис. 14.5;

Рис. 14.5. Выбор профиля Northwind.Web с веб-сервером Kestrel в программе Visual Studio

   2) выберите команду меню Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки);
   3) при первом запуске защищенного сайта вам будет предложено проверить, что ваш проект настроен на использование SSL, и, чтобы избежать предупреждений в браузере, вы можете доверять самоподписанному сертификату, который сгенерировал ASP.NET Core. Нажмите кнопку Yes (Да);
   4) когда появится диалоговое окно Security Warning (Предупреждение о безопасности), снова нажмите кнопку Yes (Да).
2. В программе Visual Studio Code на панели TERMINAL (Терминал) введите команду `dotnet run`.
3. Обратите внимание: в окне командной строки Visual Studio или в терминале Visual Studio Code веб-сервер Kestrel начал прослушивать произвольные порты

для HTTP и HTTPS. Вы можете нажать Ctrl+C, чтобы выключить веб-сервер Kestrel и назначить средой хостинга `Development`:

```
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
  Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
  Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
  Content root path: C:\Code\PracticalApps\Northwind.Web
```

Программа Visual Studio также автоматически запустит выбранный вами браузер. Если вы используете программу Visual Studio Code, то вам придется запустить браузер Google Chrome вручную.

4. Оставьте веб-сервер включенным.
5. В Chrome откройте панель Developer Tools (Инструменты разработчика) и перейдите на вкладку Network (Сеть).
6. Введите адрес `http://localhost:5000/` или любой другой номер порта, который был присвоен для HTTP, и обратите внимание, что ответ `Hello World!` выдан в виде обычного текста с кросс-платформенного веб-сервера Kestrel, как показано на рис. 14.6.

Рис. 14.6. Ответ в виде обычного текста от http://localhost:5000/

Браузер Google Chrome автоматически запрашивает файл favicon.ico для отображения на вкладке браузера, но он отсутствует, поэтому отображается ошибка 404 Not Found.

7. Введите адрес `https://localhost:5001/` или другой номер порта, который был назначен для HTTPS, и обратите внимание, что если вы не используете Visual Studio или нажали No (Нет), когда вам было предложено доверять SSL-сертификату, то в качестве ответа будет выдана ошибка конфиденциальности, как показано на рис. 14.7.

Рис. 14.7. Ошибка конфиденциальности, означающая, что шифрование SSL не защищено сертификатом SSL

Вы увидите эту ошибку, если не настроили сертификат, которому браузер может доверять при шифровании и дешифровании HTTP-трафика (а отсутствие ошибки означает, что вы уже настроили сертификат).

В рабочей среде вам пришлось бы заплатить компании, такой как Verisign, за SSL-сертификат, поскольку он обеспечивает защиту ответственности и возможность технической поддержки.

Если вы используете вариант Linux, который не может создавать самозаверяющие сертификаты, и готовы каждые 90 дней подавать заявку на новый сертификат, то можете получить бесплатный сертификат по следующей ссылке: https://letsencrypt.org.

В процессе разработки вы можете дать операционной системе указание доверять временному сертификату разработки, предоставленному ASP.NET Core.

8. В командной строке или на панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C, чтобы выключить веб-сервер, и обратите внимание на сообщение, которое появится:

```
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
This executes after the web server has stopped!

C:\Code\PracticalApps\Northwind.Web\bin\Debug\net6.0\Northwind.Web.exe
(process 19888) exited with code 0.
```

9. Если вам необходимо доверять локальному самоподписанному SSL-сертификату, то в командной строке или на панели TERMINAL (Терминал) введите команду `https dev-certs https --trust` и обратите внимание на сообщение Trusting the HTTPS development certificate was requested (Было запрошено доверие HTTPS-сертификату для разработки). Вам могут предложить ввести пароль, а правильный сертификат HTTPS может уже быть установлен.

Включение более надежной защиты и перенаправление на защищенное соединение

Рекомендуется включить более строгую защиту и автоматически перенаправлять HTTP-запросы через протокол HTTPS.

HTTP Strict Transport Security (HSTS) — это дополнительная функция безопасности, которую следует активировать в любом случае. Если сайт сообщает о ней, а браузер поддерживает, то HSTS принудительно передает все данные по протоколу HTTPS и не позволяет посетителю использовать недостоверные или недействительные сертификаты.

Сделаем это сейчас.

1. В файле `Program.cs` добавьте оператор `if`, чтобы включить HSTS (если он не находится в разработке), как показано ниже:

```
if (!app.Environment.IsDevelopment())
{
  app.UseHsts();
}
```

2. Добавьте оператор перед вызовом `app.MapGet`, чтобы перенаправить HTTP-запросы в HTTPS:

```
app.UseHttpsRedirection();
```

3. Запустите проект сайта `Northwind.Web`.
4. Если Chrome запущен, то закройте и перезапустите его.

5. В Chrome откройте панель Developer Tools (Инструменты разработчика) и перейдите на вкладку Network (Сеть).
6. Введите адрес `http://localhost:5000/` или любой другой номер порта, который был присвоен для HTTP. Обратите внимание, как сервер отвечает с помощью `307 Temporary Redirect` (временное перенаправление) на порт 5001 и что сертификат теперь действителен и достоверен, как показано на рис. 14.8.

Рис. 14.8. Теперь соединение защищено действующим сертификатом и временным перенаправлением 307

7. Закройте браузер Google Chrome.
8. Завершите работу веб-сервера.

Не забудьте останавливать работу веб-сервера Kestrel каждый раз, когда завершаете тестирование сайта.

Управление средой хостинга

В более ранних версиях ASP.NET Core в шаблоне проекта устанавливалось правило, согласно которому в режиме разработки любые необработанные исключения будут отображаться в окне браузера, чтобы разработчик мог увидеть подробную информацию об исключении:

```
if (app.Environment.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
}
```

В ASP.NET Core 6 и более поздних версиях этот код выполняется автоматически по умолчанию, поэтому недоступен в шаблоне проекта.

Как ASP.NET Core узнает, когда мы работаем в режиме разработки, чтобы метод `IsDevelopment` возвращал `true`? Попробуем разобраться.

ASP.NET Core может считывать переменные среды, чтобы определить, какую среду размещения использовать, например `DOTNET_ENVIRONMENT` или `ASPNETCORE_ENVIRONMENT`.

Во время локальной разработки эти настройки можно переопределить.

1. В папке Northwind.Web разверните папку Properties, откройте файл `launchSettings.json` и обратите внимание на профиль `Northwind.Web`, который задает в качестве среды размещения `Development`, как показано ниже (выделено жирным шрифтом):

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:56111",
      "sslPort": 44329
    }
  },
  "profiles": {
    "Northwind.Web": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

2. Измените случайно назначенные номера портов для HTTP на `5000` и HTTPS на `5001`.
3. Измените среду размещения на `Production`. В программе Visual Studio по желанию измените значение `launchBrowser` на `false`, чтобы предотвратить автоматический запуск браузера.

4. Запустите сайт и обратите внимание, что среда размещения — `Production`:

```
info: Microsoft.Hosting.Lifetime[0]
  Hosting environment: Production
```

5. Завершите работу веб-сервера.
6. В файле `launchSettings.json` измените среду размещения обратно на `Development`.

Файл launchSettings.json также содержит конфигурацию для использования IIS в качестве веб-сервера с помощью случайных номеров портов. В этой книге мы будем использовать в качестве веб-сервера только Kestrel, поскольку он является кросс-платформенным.

Разделение конфигурации для сервисов и конвейера

Разместить весь код для инициализации простого веб-проекта в файле `Program.cs` может быть хорошей идеей, особенно для веб-сервисов, поэтому мы еще раз рассмотрим этот стиль в главе 16.

Однако для проектов, более крупных, чем самый простой веб-проект, лучше выделить конфигурацию в отдельный класс `Startup` с помощью двух методов:

- `ConfigureServices(IServiceCollection services)` служит для добавления сервисов зависимостей в контейнер внедрения зависимостей, таких как поддержка Razor Pages, поддержка *совместного использования ресурсов между разными источниками* (Cross-Origin Resource Sharing, CORS) или контекст базы данных для работы с базой данных Northwind.
- `Configure(IApplicationBuilder app, IWebHostEnvironment env)` используется для настройки HTTP-конвейера, через который проходят запросы и ответы. Вызовите различные методы `Use` для параметра `app`, чтобы собрать конвейер в том порядке, в котором должны обрабатываться функции.

Рис. 14.9. Диаграмма методов стартового класса ConfigureServices и Configure

Оба метода будут вызываться автоматически во время выполнения.

Теперь создадим класс `Startup`.

1. Создайте файл класса `Startup.cs` в проекте `Northwind.Web`.
2. Измените содержимое файла `Startup.cs`:

```
namespace Northwind.Web;

public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
  }

  public void Configure(
    IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (!env.IsDevelopment())
    {
      app.UseHsts();
    }

    app.UseRouting(); // начало маршрутизации конечной точки

    app.UseHttpsRedirection();

    app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/", () => "Hello World!");
    });
  }
}
```

Обратите внимание на следующие моменты:

- метод `ConfigureServices` в настоящее время пуст, поскольку нам пока не нужно добавлять зависимые сервисы;
- метод `Configure` устанавливает конвейер HTTP-запросов и позволяет использовать маршрутизацию конечных точек. Он настраивает маршрутизируемую конечную точку на ожидание запросов с помощью одной и той же карты для каждого HTTP-запроса `GET` для корневого пути `/`, которая отвечает на эти запросы, возвращая простой текст `"Hello World!"`. О маршрутизируемых конечных точках и их преимуществах вы узнаете в конце этой главы.

Теперь мы должны указать, что хотим использовать класс `Startup` в точке входа приложения.

3. Измените файл `Program.cs`:

```
using Northwind.Web; // класс Startup

Host.CreateDefaultBuilder(args)
  .ConfigureWebHostDefaults(webBuilder =>
  {
    webBuilder.UseStartup<Startup>();
  }).Build().Run();

Console.WriteLine("This executes after the web server has stopped!");
```

4. Запустите сайт и обратите внимание, что его поведение не изменилось.
5. Завершите работу веб-сервера.

Во всех остальных проектах сайтов и сервисов, которые мы создадим в этой книге, мы будем использовать один файл Program.cs, созданный с помощью шаблонов проектов .NET 6. Прием с использованием файла Startup.cs рассматривается в данной главе.

Как позволить сайту обрабатывать статический контент

Сайт, возвращающий только одно текстовое сообщение, не очень полезен!

Как минимум он должен возвращать статические HTML-страницы, CSS, которые веб-страницы будут использовать для стилизации, и любые другие статические ресурсы, такие как изображения и видео.

По соглашению эти файлы следует хранить в каталоге wwwroot, чтобы отделить их от динамически выполняемых частей проекта вашего сайта.

Создание папки для статических файлов и веб-страницы

Создадим папку для статических ресурсов сайта и основную страницу индекса, использующую для стилизации Bootstrap.

1. Создайте в проекте/папке `Northwind.Web` папку wwwroot.
2. Добавьте в папку wwwroot новый HTML-файл `index.html`.
3. Измените содержимое файла, чтобы добавить ссылку на Bootstrap, размещенный на CDN, для стилизации и используйте современные передовые практики, такие как настройка области просмотра:

```
<!doctype html>
<html lang="en">
```

```
<head>
  <!-- Требуемые мета-теги -->
  <meta charset="utf-8" />
  <meta name="viewport" content=
    "width=device-width, initial-scale=1 " />

  <!-- Bootstrap CSS -->
  <link href=
"https://cdn.jsdelivr.NET/npm/bootstrap@5.1.0/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-KyZXEAg3QhqLMpG8r+8fhAXLRk2vvoC2f3B09zV
Xn8CA5QIVfZOJ3BCsw2P0p/We" crossorigin="anonymous">

  <title>Welcome ASP.NET Core!</title>
</head>

<body>
  <div class="container">
    <div class="jumbotron">
      <h1 class="display-3">Welcome to Northwind B2B</h1>
      <p class="lead">We supply products to our customers.</p>
      <hr />
      <h2>This is a static HTML page.</h2>
      <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 получить последний элемент <link> для Bootstrap, скопируйте его из документации (https://getbootstrap.com/docs/5.0/getting-started/introduction/#starter-template) и вставьте.

Включение статических файлов и файлов по умолчанию

Если бы вы запустили сайт прямо сейчас и ввели в адресной строке браузера адрес `http://localhost:5000/index.html`, то сайт вернул бы ошибку `404 Not Found`, тем самым сообщив, что веб-страница не найдена. Чтобы он мог возвращать статические файлы, такие как `index.html`, необходимо явно настроить данную функцию.

Даже после включения статических файлов, если вы запустите веб-приложение и введете в поле адреса `http://localhost:5000/`, сайт выдаст ошибку `404 Not Found`, поскольку веб-сервер не знает, что возвращать по умолчанию, если не запрашивается никакой именованный файл.

Включим статические файлы и явно сконфигурируем файлы по умолчанию и изменим зарегистрированный путь URL, который возвращает ответ в виде простого текста `Hello World!`.

1. В файле `Startup.cs` в методе `Configure` добавьте следующие операторы после включения перенаправления HTTPS, чтобы включить статические файлы и файлы по умолчанию, и измените оператор, который отправляет любой запрос `GET` для возврата ответа `Hello World!` в виде простого текста, как показано ниже:

```
app.UseHttpsRedirection();

app.UseDefaultFiles(); // index.html, default.html и т.п.

app.UseStaticFiles();

app.UseEndpoints(endpoints =>
{
  endpoints.MapGet("/hello", () => "Hello World!");
});
```

Вызов метода UseDefaultFiles должен следовать перед вызовом метода UseStaticFiles, в противном случае метод не сработает! Более подробно о порядке размещения промежуточного программного обеспечения и маршрутизации конечных точек вы узнаете в конце этой главы.

2. Запустите сайт.
3. Запустите браузер Google Chrome и отобразите панель Developer Tools (Инструменты разработчика).
4. В адресной строке браузера введите `http://localhost:5000/`. Обратите внимание: вы перенаправлены на HTTPS-адрес на порте `5001`, и файл `index.html` возвращается теперь по защищенному соединению, поскольку является одним из возможных файлов по умолчанию для этого сайта.
5. На панели Developer Tools (Инструменты разработчика) обратите внимание на запрос таблицы стилей Bootstrap.
6. В адресной строке браузера введите `http://localhost:5000/hello` и обратите внимание, что отображается простой текст `Hello World!`, как и раньше.
7. Закройте браузер Google Chrome и завершите работу веб-сервера.

Если все веб-страницы — статические, то есть изменяются только вручную веб-редактором, значит, работа по разработке нашего сайта завершена. Однако почти всем сайтам необходим динамический контент, то есть веб-страница, генерируемая во время работы сайта путем выполнения программного кода.

Проще всего добиться этого результата, использовав функцию ASP.NET Core под названием *Razor Pages*.

Функция Razor Pages от ASP.NET Core

Razor Pages позволяет разработчику легко смешивать операторы C# с разметкой HTML, чтобы сделать создаваемую веб-страницу динамической. Вот почему они используют расширение файла `.cshtml`.

По соглашению ASP.NET Core ищет Razor Pages в папке Pages.

Добавление Razor Pages

Скопируем статическую HTML-страницу и изменим ее на динамическую Razor Page, а затем добавим и включим сервис Razor Pages.

1. В папке проекта `Northwind.Web` создайте папку Pages.
2. Скопируйте файл `index.html` в папку Pages.
3. Для файла в папке Pages измените расширение файла с `.html` на `.cshtml`.
4. Удалите элемент `<h2>`, который означает, что это статическая HTML-страница.
5. В файле `Startup.cs` в методе `ConfigureServices` введите оператор для добавления ASP.NET Core Razor Pages и связанные с ними сервисы, такие как привязка модели, авторизация, защита от подделки, представления и вспомогательные теги к компоновщику:

```
Services.AddRazorPages();
```

6. В файле `Startup.cs` в методе `Configure` в конфигурации использования конечных точек добавьте оператор для вызова метода `MapRazorPages`, как показано ниже (выделено жирным шрифтом):

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapRazorPages();

  endpoints.MapGet("/hello", () => "Hello World!");
});
```

Добавление кода к Razor Pages

Синтаксис Razor в HTML-разметке веб-страницы обозначается символом `@`. Страницу, написанную с помощью Razor Pages, можно описать следующим образом.

- В начале файла требуется директива `@page`.
- В ней дополнительно может содержаться раздел `@functions`, определяющий:
  - свойства для хранения значений данных, как в определении класса. Экземпляр этого класса автоматически создается с именем `Model`, его свойства

могут быть установлены в специальных методах, и вы можете получить значения свойств в HTML;

- методы с именами `OnGet`, `OnPost`, `OnDelete` и т. д., которые работают при выполнении HTTP-запросов, таких как `GET`, `POST` и `DELETE`.

Преобразуем статическую HTML-страницу в Razor Page.

1. В папке Pages откройте файл `index.cshtml`.
2. Добавьте оператор `@page` в начало файла.
3. Добавьте блок оператора `@functions` после оператора `@page`.
4. Определите свойство для сохранения названия текущего дня в виде значения `string`.
5. Определите метод для установки переменной `DayName`, который работает при выполнении HTTP-запроса `GET` для страницы:

```
@page

@functions
{
  public string? DayName { get; set; }

  public void OnGet()
  {
    Model.DayName = DateTime.Now.ToString("dddd");
  }
}
```

6. Выведите название дня во втором абзаце HTML, как показано ниже (выделено жирным шрифтом):

```
<p>It's @Model.DayName! Our customers include restaurants, hotels, and
Cruise lines.</p>
```

7. Запустите сайт.
8. В адресной строке Chrome введите `https://localhost:5001/`. Вы увидите, что на странице отображается название текущего дня (рис. 14.10).

Рис. 14.10. Страница «Добро пожаловать в Northwind» отображает текущий день

9. В адресной строке Chrome введите `https://localhost:5001/index.html`, что точно соответствует имени статического файла, и обратите внимание, что он, как и раньше, возвращает статическую HTML-страницу.
10. В адресной строке Chrome введите `https://localhost:5001/hello`, что в точности соответствует конечному маршруту, который возвращает простой текст, и обратите внимание, что он, как и раньше, возвращает `Hello World!`.
11. Закройте браузер Google Chrome и завершите работу веб-сервера.

Использование общих макетов в Razor Pages

Большинство сайтов имеют несколько страниц. Если бы каждая из них содержала всю шаблонную разметку, которая в настоящее время находится в файле `index.cshtml`, то управление стало бы проблематичным. Поэтому ASP.NET Core содержит функцию под названием *«макеты»*.

Чтобы использовать макеты, следует создать Razor-файл, определяющий макет по умолчанию для всех Razor Pages (и всех представлений MVC), и сохранить его в папке Shared, таким образом облегчив его поиск по соглашению. Название этого файла может быть любым, поскольку мы укажем его, но обычно принято использовать вариант `_Layout.cshtml`.

Кроме того, чтобы установить макет по умолчанию для всех Razor Pages (и всех представлений MVC), необходимо создать файл с особым именем. Он должен называться `_ViewStart.cshtml`.

Посмотрим, как работают макеты.

1. В папке Pages создайте файл `_ViewStart.cshtml`. (Шаблон элемента Visual Studio называется Razor View Start).
2. Измените содержимое файла:

```
@{
  Layout = "_Layout";
}
```

3. В папке Pages создайте подпапку Shared.
4. В папке Shared создайте файл `_Layout.cshtml`. (Шаблон элемента Visual Studio называется Razor Layout).
5. Отредактируйте содержимое файла `_Layout.cshtml` (оно похоже на содержимое файла `index.cshtml`, поэтому вы можете просто скопировать из него разметку HTML и вставить ее):

```
<!doctype html>
<html lang="en">
```

```
<head>
  <!-- Требуемые мета-теги -->
  <meta charset="utf-8" />
  <meta name="viewport" content=
    "width=device-width, initial-scale=1, shrink-to-fit=no" />

  <!-- Bootstrap CSS -->
  <link href=
"https://cdn.jsdelivr.NET/npm/bootstrap@5.1.0/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-KyZXEAg3QhqLMpG8r+8fhAXLRk2vvoC2f3B09zV
Xn8CA5QIVfZOJ3BCsw2P0p/We" crossorigin="anonymous">

  <title>@ViewData["Title"]</title>
</head>

<body>
  <div class="container">
    @RenderBody()
    <hr />
    <footer>
      <p>Copyright &copy; 2021 - @ViewData["Title"]</p>
    </footer>
  </div>

  <!-- JavaScript-код для включения таких функций, как карусель -->
  <script src="https://cdn.jsdelivr.NET/npm/bootstrap@5.1.0/
dist/js/bootstrap.bundle.min.js" integrity="sha384-
U1DAWAznBHeqEIlVSCgzq+c9gqGAJn5c/t99JyeKa9xxaYpSvHU5awsuZVVFIhvj"
crossorigin="anonymous"></script>

  @RenderSection("Scripts", required: false)

</body>
</html>
```

При анализе этой разметки обратите внимание на следующие моменты:

- элемент `<title>` устанавливается динамически с помощью серверного кода из словаря `ViewData`. Это простой способ передачи данных между различными частями сайта ASP.NET Core. В этом случае данные будут установлены в файле класса Razor Page, а затем выведены в общий макет;
- метод `@RenderBody()` отмечает точку вставки для запрашиваемого представления;
- внизу каждой страницы будут отображаться горизонтальная линейка и нижний колонтитул;
- в нижней части макета находится сценарий для реализации кое-каких полезных функций Bootstrap, которые мы будем использовать позже, — например, карусели изображений;

- после элементов `<script>` для Bootstrap мы определили раздел `Scripts`, чтобы с помощью технологии Razor Page можно было вставить дополнительные необходимые сценарии.

6. Отредактируйте код файла `index.cshtml`: удалите всю HTML-разметку, за исключением `<div class="jumbotron">` и ее содержимого, и оставьте код C# в добавленном ранее блоке `@functions`.
7. Добавьте в метод `OnGet` оператор для сохранения заголовка страницы в словаре `ViewData` и измените кнопку для перехода на страницу поставщиков (которую мы создадим чуть позже), как показано ниже (выделено жирным шрифтом):

```
@page

@functions
{
  public string? DayName { get; set; }

  public void OnGet()
  {
    ViewData["Title"] = "Northwind B2B";

    Model.DayName = DateTime.Now.ToString("dddd");
  }
}
<div class="jumbotron">
  <h1 class="display-3">Welcome to Northwind B2B</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>
```

8. Запустите сайт, посетите его с помощью браузера Google Chrome. Вы увидите, что сайт работает так же, как раньше, хотя нажатие кнопки для перехода на страницу поставщиков приведет к ошибке `404 Not Found`, поскольку страница еще не создана.

Использование файлов с выделенным кодом в Razor Pages

Иногда лучше отделить HTML-разметку от данных и исполняемого кода, поэтому функция Razor Pages позволяет вам сделать это, поместив код C# в файлы классов *с выделенным кодом* (code-behind). Они называются так же, как и файл `.cshtml`, но имеют расширение `.cshtml.cs`.

Создадим страницу, которая отображает список поставщиков. В этом примере мы сконцентрируемся на изучении файлов с выделенным кодом. Чуть позже мы загрузим список поставщиков из базы данных, но в настоящий момент смоделируем это с помощью жестко запрограммированного массива значений `string`.

1. В папку Pages добавьте два новых файла: `Suppliers.cshtml` и `Suppliers.cshtml.cs`. (Шаблон элемента Visual Studio называется Razor Page - Empty, он создает оба файла.)
2. Добавьте операторы в файл с выделенным кодом `Suppliers.cshtml.cs`:

```
using Microsoft.AspNetCore.Mvc.RazorPages; // PageModel

namespace Northwind.Web.Pages;

public class SuppliersModel : PageModel
{
  public IEnumerable<string>? Suppliers { get; set; }

  public void OnGet()
  {
    ViewData["Title"] = "Northwind B2B - Suppliers";

    Suppliers = new[]
    {
      "Alpha Co", "Beta Limited", "Gamma Corp"
    };
  }
}
```

   При анализе этой разметки обратите внимание на следующие моменты:

   - класс `SuppliersModel` наследуется от `PageModel`, поэтому содержит такие элементы, как словарь `ViewData` для совместного использования данных. Вы можете щелкнуть правой кнопкой мыши на элементе `PageModel` и выбрать в контекстном меню команду Go To Definition (Перейти к определению), чтобы увидеть, что в классе содержится намного больше полезных элементов, таких как объект `HttpContext` текущего запроса;
   - класс `SuppliersModel` определяет свойство для хранения коллекции `Suppliers`, содержащей значения `string`;
   - когда для этой Razor-страницы выполняется HTTP-запрос `GET`, свойство `Suppliers` заполняется некоторыми именами поставщиков из массива значений `string`. Позже мы заполним его из базы данных Northwind.
3. Отредактируйте содержимое файла `Suppliers.cshtml`:

```
@page
@model Northwind.Web.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>
    @if (Model.Suppliers is not null)
    {
      @foreach(string name in Model.Suppliers)
      {
        <tr><td>@name</td></tr>
      }
    }
    </tbody>
  </table>
</div>
```

При анализе этой разметки обратите внимание на следующие моменты;

- типом модели для этой Razor-страницы выбран класс `SuppliersModel`;
- на странице отображается HTML-таблица со стилями Bootstrap;
- строки данных в таблице создаются циклически с помощью свойства `Suppliers` класса `Model`, если оно не равно `null`.

4. Запустите сайт и посетите его с помощью браузера Google Chrome.
5. Нажмите кнопку, позволяющую узнать больше информации о поставщиках, и обратите внимание на таблицу `Suppliers` (рис. 14.11).

Рис. 14.11. Таблица с информацией о поставщиках, загруженная из массива строк

Использование Entity Framework Core совместно с ASP.NET Core

С помощью Entity Framework Core можно естественным способом получить реальные данные на сайте. В главе 13 вы создали две пары библиотек классов: для сущностных моделей и для контекста базы данных Northwind либо для SQL Server, либо для SQLite, либо для обоих. Теперь вы будете использовать их в проекте вашего сайта.

Настройка Entity Framework Core как сервиса

Такие функциональные возможности, как контексты базы данных Entity Framework Core, необходимые для ASP.NET Core, во время запуска сайта должны быть зарегистрированы как сервис. В коде из репозитория GitHub и в примерах ниже используется SQLite, но вы можете легко изменить код под SQL Server.

Рассмотрим пример.

1. В проекте `Northwind.Web` добавьте ссылку на проект `Northwind.Common.DataContext` для SQLite или SQL Server:

```
<!-- при необходимости смените Sqlite на SqlServer -->
<ItemGroup>
  <ProjectReference Include="..\Northwind.Common.DataContext.Sqlite\
  Northwind.Common.DataContext.Sqlite.csproj" />
</ItemGroup>
```

Ссылку на проект нужно располагать на одной строке, не разрывая.

2. Соберите проект `Northwind.Web`.
3. В файле `Startup.cs` и импортируйте пространства имен для работы с вашими типами моделей сущностей:

```
using Packt.Shared; // метод расширения AddNorthwindContext
```

4. Добавьте в метод `ConfigureServices` оператор, чтобы зарегистрировать класс контекста базы данных Northwind:

```
services.AddNorthwindContext();
```

5. В проекте `Northwind.Web` в папке Pages откройте файл `Suppliers.cshtml.cs` и импортируйте пространство имен для нашего контекста базы данных:

```
using Packt.Shared; // NorthwindContext
```

6. В классе `SuppliersModel` добавьте скрытое поле для хранения контекста базы данных Northwind и конструктор для его установки:

```
private NorthwindContext db;

public SuppliersModel(NorthwindContext injectedContext)
{
  db = injectedContext;
}
```

7. Измените свойство `Suppliers`, чтобы оно содержало объекты `Supplier` вместо значений `string`.
8. В методе `OnGet` измените операторы для установки свойства `Suppliers` из свойства `Suppliers` контекста базы данных, отсортированного по стране, а затем по названию компании, как показано ниже (выделено жирным шрифтом):

```
public void OnGet()
{
  ViewData["Title"] = "Northwind B2B - Suppliers";

  Suppliers = db.Suppliers
    .OrderBy(c => c.Country).ThenBy(c => c.CompanyName);
}
```

9. Измените содержимое файла `Suppliers.cshtml`, чтобы импортировать пространство имен `Packt.Shared` и отобразить несколько столбцов для каждого поставщика:

```
@page
@using Packt.Shared
@model Northwind.Web.Pages.SuppliersModel
<div class="row">
  <h1 class="display-2">Suppliers</h1>
  <table class="table">
    <thead class="thead-inverse">
      <tr>
        <th>Company Name</th>
        <th>Country</th>
        <th>Phone</th>
      </tr>
    </thead>
    <tbody>
    @if (Model.Suppliers is not null)
    {
      @foreach(Supplier s in Model.Suppliers)
      {
        <tr>
          <td>@s.CompanyName</td>
          <td>@s.Country</td>
```

```
            <td>@s.Phone</td>
          </tr>
        }
      }
      </tbody>
    </table>
  </div>
```

10. Запустите сайт.
11. В адресной строке Chrome введите `https://localhost:5001/`.
12. Нажмите Learn more about our suppliers и обратите внимание, что таблица с информацией о поставщиках теперь загружается из базы данных, как показано на рис. 14.12.

Рис. 14.12. Таблица с информацией о поставщиках, загруженная из базы данных Northwind

Управление данными с помощью страниц Razor

Добавим функции для вставки нового поставщика.

Добавление модели возможности вставки объектов

В первую очередь необходимо изменить модель поставщика, чтобы она отвечала на HTTP-запросы `POST`, когда пользователь отправляет форму для вставки нового поставщика.

1. В проекте `Northwind.Web` в папке Pages откройте файл `Suppliers.cshtml.cs` и импортируйте следующее пространство имен:

```
using Microsoft.AspNetCore.Mvc; // [BindProperty], IactionResult
```

2. В класс `SuppliersModel` добавьте свойство для хранения одного поставщика и метод `OnPost`, который добавляет поставщика в таблицу `Suppliers` в базе данных Northwind, если его модель действительна:

```
[BindProperty]
public Supplier? Supplier { get; set; }

public IActionResult OnPost()
{
  if ((Supplier is not null) && ModelState.IsValid)
  {
    db.Suppliers.Add(Supplier);
    db.SaveChanges();
    return RedirectToPage("/suppliers");
  }
  else
  {
    return Page(); // возвращает исходную страницу
  }
}
```

При анализе этого кода обратите внимание на следующие моменты:

- мы добавили свойство `Supplier`, которое дополнено атрибутом `[BindProperty]`, чтобы упростить связь HTML-элементов на веб-странице со свойствами в классе `Supplier`;
- мы добавили метод, отвечающий на HTTP-запросы `POST`. Он проверяет все значения свойств на соответствие правилам проверки в сущностной модели класса `Supplier` (например, `[Required]` и `[StringLength]`), а затем добавляет поставщика в существующую таблицу и сохраняет изменения в контексте базы данных. Это сгенерирует оператор SQL для выполнения вставки в БД. Затем запрос перенаправляет на страницу `Suppliers` для отображения только что добавленного поставщика.

Определение формы для добавления нового поставщика

Далее необходимо изменить страницу Razor, чтобы определить форму, которую пользователь может заполнить и отправить для вставки нового поставщика.

1. В файле `Suppliers.cshtml` добавьте вспомогательные теги после объявления `@model`, чтобы мы могли использовать их на этой странице Razor (например, `asp-for`):

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

2. В конце файла добавьте форму для вставки нового поставщика и с помощью вспомогательного тега `asp-for` привяжите к полю ввода свойства `CompanyName`, `Country` и `Phone` класса `Supplier`:

```
<div class="row">
  <p>Enter details for a new supplier:</p>
  <form method="POST">
    <div><input asp-for="Supplier.CompanyName"
                placeholder="Company Name" /></div>
    <div><input asp-for="Supplier.Country"
                placeholder="Country" /></div>
    <div><input asp-for="Supplier.Phone"
                placeholder="Phone" /></div>
    <input type="submit" />
  </form>
</div>
```

При анализе этого кода обратите внимание на следующие моменты:

- элемент `<form>` с методом `POST` — обычный HTML-элемент, поэтому внутри него элемент `<input type="submit" />` отправит HTTP-запрос `POST` обратно на текущую страницу вместе со значениями любых других элементов внутри данной формы;
- элемент `<input>` с вспомогательным тегом `asp-for` обеспечивает привязку данных к модели на странице Razor.

3. Запустите сайт.
4. Нажмите Learn more about our suppliers, прокрутите вниз до конца страницы, введите `Bob's Burgers`, `USA` и `(603) 555-4567` и нажмите кнопку Submit (Отправить).
5. Обратите внимание: вы перенаправлены обратно в таблицу `Suppliers`, в которую добавлен новый поставщик.
6. Закройте браузер Google Chrome и завершите работу веб-сервера.

Внедрение сервиса зависимостей в страницу Razor

Если у вас есть страница Razor `.cshtml`, не имеющая файла с выделенным кодом, то вы можете внедрить сервис зависимостей, используя директиву `@inject`, вместо того чтобы внедрять параметры конструктора, а затем непосредственно ссылаться на внедренный контекст базы данных, используя синтаксис Razor в середине разметки.

Создадим простой пример.

1. В папке Pages создайте файл `Orders.cshtml`. (Шаблон элемента Visual Studio называется Razor Page - Empty, он создает два файла. Удалите файл `.cs`.)
2. В файле `Orders.cshtml` напишите код для вывода количества заказов в базе данных Northwind, как показано ниже:

```
@page
@using Packt.Shared
```

```
@inject NorthwindContext db
@{
  string title = "Orders";
  ViewData["Title"] = $"Northwind B2B - {title}";
}
<div class="row">
  <h1 class="display-2">@title</h1>
  <p>
    There are @db.Orders.Count() orders in the Northwind database.
  </p>
</div>
```

3. Запустите сайт.
4. Перейдите в раздел /orders и обратите внимание, что в базе данных Northwind имеется 830 заказов.
5. Закройте браузер Google Chrome и завершите работу веб-сервера.

Использование библиотек классов Razor

Все связанное со страницей Razor можно скомпилировать в библиотеку классов, чтобы облегчить повторное использование в разных проектах. В.NET Core 3.0 и более поздних версиях в библиотеку также можно включать статические файлы, такие как HTML, CSS, библиотеки JavaScript, и мультимедийные ресурсы, например файлы изображений. Сайт может использовать представление страницы Razor, определенное в библиотеке классов, или переопределить его.

Создание библиотеки классов Razor

Создадим новую библиотеку классов Razor.

Откройте редактор кода и создайте проект с такими настройками:

1) шаблон проекта: Razor Class Library/`razorclasslib`;
2) флажок/переключатель: Support pages and views (Поддержка страниц и представлений)/`-s`;
3) файл и папка рабочей области/решения: `PracticalApps`;
4) файл и папка проекта: `Northwind.Razor.Employee`.

Параметр -s является сокращением от названия переключателя --support-pages-and-views, который позволяет библиотеке классов использовать Razor Pages и представления файлов .cshtml.

Отключение компактного режима просмотра папок в Visual Studio Code

Прежде чем мы реализуем нашу библиотеку классов Razor, я хочу объяснить функцию Visual Studio Code, которая смутила некоторых читателей предыдущего издания, поскольку была добавлена после публикации книги.

Функция компактных папок означает, что вложенные папки, такие как /Areas/MyFeature/Pages/, отображаются в компактной форме, если промежуточные папки в иерархии не содержат файлов (рис. 14.13).

Рис. 14.13. Включен или отключен компактный режим просмотра папок

Если вы хотите отключить эту функцию, то выполните следующие действия.

1. В Windows выберите команду меню File ▸ Preferences ▸ Settings (Файл ▸ Параметры ▸ Настройки). В macOS выберите команду меню Code ▸ Preferences ▸ Settings (Код ▸ Параметры ▸ Настройки).
2. В поле настроек Search введите `compact`.
3. Снимите флажок Explorer: Compact Folders (рис. 14.14).
4. Закройте вкладку Settings (Настройки).

Рис. 14.14. Отключение компактного режима просмотра папок для Visual Studio Code

Реализация функции сотрудников с помощью EF Core

Теперь мы можем добавить ссылку на наши сущностные модели, чтобы имена сотрудников отображались в библиотеке классов Razor.

1. В проекте `Northwind.Razor.Employees` добавьте ссылку на проект `Northwind.Common.DataContext` для SQLite или SQL Server и обратите внимание, что указан SDK — Microsoft.NET.Sdk.Razor, как показано ниже (выделено жирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <!-- при необходимости смените Sqlite на SqlServer -->
  <ItemGroup>
    <ProjectReference Include="..\Northwind.Common.DataContext.Sqlite
    \Northwind.Common.DataContext.Sqlite.csproj" />
  </ItemGroup>

</Project>
```

Ссылку на проект нужно располагать на одной строке, не разрывая. Кроме того, не смешивайте проекты SQLite и SQL Server, иначе возникнут ошибки компиляции. Если вы использовали SQL Server в проекте Northwind.Web, то должны использовать SQL Server и в проекте Northwind.Razor.Employees.

2. Соберите проект `Northwind.Razor.Employees`.
3. В папке Areas щелкните правой кнопкой мыши на папке MyFeature, выберите в контекстном меню команду Rename (Переименовать), введите новое имя `PacktFeatures` и нажмите клавишу Enter.
4. В папке PacktFeatures во вложенной папке Pages создайте файл `_ViewStart.cshtml`. (Шаблон элемента Visual Studio называется Razor View Start. Или скопируйте его из проекта `Northwind.Web`.)
5. Измените содержимое файла, чтобы сообщить этой библиотеке классов, что все страницы Razor должны искать макет с тем же именем, которое используется в проекте `Northwind.Web`:

```
@{
  Layout = "_Layout";
}
```

Не нужно создавать файл _Layout.cshtml в этом проекте. Будет использоваться тот, который находится в основном проекте, например в проекте Northwind.Web.

6. В подпапке Pages переименуйте файл `Page1.cshtml` в `Employees.cshtml`, а файл `Page1.cshtml.cs` — в `Employees.cshtml.cs`.
7. Отредактируйте файл `Employees.cshtml.cs`, чтобы определить модель страницы с массивом сущностных экземпляров `Employee`, загруженных из базы данных Northwind:

```
using Microsoft.AspNetCore.Mvc.RazorPages; // PageModel
using Packt.Shared; // Employee, NorthwindContext

namespace PacktFeatures.Pages;

public class EmployeesPageModel : PageModel
{
  private NorthwindContext db;

  public EmployeesPageModel(NorthwindContext injectedContext)
  {
    db = injectedContext;
  }

  public Employee[] Employees { get; set; } = null!;

  public void OnGet()
  {
    ViewData["Title"] = "Northwind B2B - Employees";
    Employees = db.Employees.OrderBy(e => e.LastName)
      .ThenBy(e => e.FirstName).ToArray();
  }
}
```

8. Отредактируйте файл `employee.cshtml`:

```
@page
@using Packt.Shared
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model PacktFeatures.Pages.EmployeesPageModel
<div class="row">
  <h1 class="display-2">Employees</h1>
</div>
<div class="row">
@foreach(Employee employee in Model.Employees)
```

```
{
  <div class="col-sm-3">
    <partial name="_Employee" model="employee" />
  </div>
}
</div>
```

При анализе этого кода обратите внимание на следующие моменты:

- мы импортируем пространство имен `Packt.Shared`, чтобы в нем можно было использовать классы, такие как `Employee`;
- мы добавили поддержку для вспомогательного тега, чтобы могли применить элемент `<partial>`;
- мы объявили тип модели @model для нашей страницы Razor, чтобы задействовать только что определенный класс модели страницы;
- мы проходимся по элементам коллекции `Employees`, выводя каждый элемент с помощью частичного представления.

Реализация частичного представления для отображения одного сотрудника

Вспомогательный тег `<partial>` был представлен в ASP.NET Core 2.1. Частичное представление похоже на часть страницы Razor. Вы создадите его в этом разделе для отображения одного сотрудника.

1. В проекте `Northwind.Razor.Employees` в папке Pages создайте подпапку Shared.
2. В папке Shared создайте файл `_Employee.cshtml`. (Шаблон элемента Visual Studio называется Razor View - Empty.)
3. Отредактируйте файл `_Employee.cshtml`:

```
@model Packt.Shared.Employee
<div class="card border-dark mb-3" style="max-width: 18rem;">
  <div class="card-header">@Model?.LastName, @Model?.FirstName</div>
  <div class="card-body text-dark">
    <h5 class="card-title">@Model?.Country</h5>
    <p class="card-text">@Model?.Notes</p>
  </div>
</div>
```

При анализе этой разметки обратите внимание на следующие моменты:

- по соглашению имена частичных представлений начинаются с подчеркивания;
- если вы поместите частичное представление в папку Shared, то его можно будет найти автоматически;

- тип модели для этого частичного представления — отдельная сущность `Employee`;
- чтобы вывести информацию о каждом сотруднике, в нашем примере мы использовали стили карточек Bootstrap.

Использование и тестирование библиотеки классов Razor

Теперь в проекте сайта вы будете ссылаться на библиотеку классов Razor и использовать ее.

1. В проекте `Northwind.Web` добавьте ссылку на проект `Northwind.Razor.Employees`:

```
<ProjectReference Include=
  "..\Northwind.Razor.Employees\Northwind.Razor.Employees.csproj" />
```

2. Отредактируйте файл `Pages\index.cshtml`, чтобы добавить абзац со ссылкой на страницу сотрудников после ссылки на страницу поставщиков:

```
<p>
  <a class="btn btn-primary" href="packtfeatures/employees">
    Contact our employees
  </a>
</p>
```

3. Запустите сайт, посетите его с помощью браузера Google Chrome и нажмите кнопку Contact our employees (Свяжитесь с нашими сотрудниками), чтобы просмотреть карточки сотрудников (рис. 14.15).

Рис. 14.15. Список сотрудников из библиотеки классов Razor

Настройка сервисов и конвейера HTTP-запросов

Теперь, когда мы создали сайт, мы можем вернуться к конфигурации `Startup` и более подробно рассмотреть, как работают сервисы и конвейер HTTP-запросов.

Маршрутизация конечных точек

В предыдущих версиях ASP.NET Core система маршрутизации и расширяемая система промежуточного программного обеспечения не всегда успешно работали вместе; например, если вы хотели реализовать такую политику, как CORS, как в промежуточном ПО, так и в MVC. Компания Microsoft инвестировала средства в улучшение маршрутизации с помощью системы *маршрутизации конечных точек* (endpoint routing), представленной в ASP.NET Core 2.2.

Маршрутизация конечных точек заменяет маршрутизацию на основе IRouter, используемую в ASP.NET Core 2.1 и более ранних версиях. Microsoft рекомендует владельцам всех старых проектов ASP.NET Core по возможности перейти на маршрутизацию конечных точек.

Маршрутизация конечных точек получила свое название потому, что она представляет таблицу маршрутов в виде скомпилированного дерева конечных точек, по которым может эффективно перемещаться система маршрутизации. Одно из самых крупных улучшений касается производительности маршрутизации и выбора метода действий.

Данная технология включена по умолчанию в ASP.NET Core 2.2 и более поздней версии, если для совместимости установлено значение 2.2 или более поздней версии. Традиционные маршруты, зарегистрированные с помощью метода `MapRoute` или с помощью атрибутов, сопоставляются с новой системой.

Новая система маршрутизации включает сервис генерации ссылок, зарегистрированный как сервис зависимостей, которому не требуется `HttpContext`.

Настройка маршрутизации конечных точек

Технология маршрутизации конечных точек требует пары вызовов методов `UseRouting` и `UseEndpoints`:

- `UseRouting` отмечает позицию конвейера, в которой принимается решение о маршрутизации;
- `UseEndpoints` отмечает позицию конвейера, в которой выполняется выбранная конечная точка.

Промежуточное программное обеспечение, например для локализации, которое выполняется между этими методами, определяет выбранную конечную точку и при необходимости может переключиться на другую.

Технология маршрутизации конечных точек основана на том же синтаксисе, который используется в ASP.NET MVC с 2010 года, и использует атрибут `[Route]`, введенный в ASP.NET MVC 5 в 2013 году. Зачастую миграция требует внесения изменений лишь в конфигурацию `Startup`.

Контроллеры MVC, Razor Pages и фреймворки, такие как SignalR, раньше включались с помощью вызова `UseMvc` или аналогичных методов, но теперь добавляются внутри вызова метода `UseEndpoints`, поскольку все они интегрированы в одну и ту же систему маршрутизации вместе с промежуточным программным обеспечением.

Проверка конфигурации маршрутизации конечных точек

Проанализируйте файл класса `Startup.cs`:

```
using Packt.Shared; // метод расширения AddNorthwindContext

namespace Northwind.Web;

public class Startup
{
  public void ConfigureServices(IserviceCollection services)
  {
    services.AddRazorPages();

    services.AddNorthwindContext();
  }

  public void Configure(
    IapplicationBuilder app, IwebHostEnvironment env)
  {
    if (!env.IsDevelopment())
    {
      app.UseHsts();
    }

    app.UseRouting();

    app.UseHttpsRedirection();

    app.UseDefaultFiles(); // файлы index.html, default.html и т. п.
    app.UseStaticFiles();
```

```
    app.UseEndpoints(endpoints =>
    {
      endpoints.MapRazorPages();

      endpoints.MapGet("/hello", () => "Hello World!");
    });
  }
}
```

Класс `Startup` содержит два метода, которые автоматически вызываются хостом для настройки сайта.

Метод `ConfigureServices` регистрирует сервисы, которые затем можно получить, когда функциональность, которую они предоставляют, необходимо получить с помощью внедрения зависимостей. Наш код регистрирует два сервиса: Razor Pages и контекст базы данных EF Core.

Регистрация сервисов в методе ConfigureServices

В табл. 14.2 продемонстрированы общие методы, регистрирующие сервисы зависимостей, включая сервисы, которые объединяют вызовы других методов, регистрирующих сервисы.

Таблица 14.2. Методы, регистрирующие сервисы зависимостей

Метод	Регистрируемые сервисы
`AddMvcCore`	Минимальный набор сервисов, необходимых для маршрутизации запросов и вызова контроллеров. Большинству сайтов потребуются дополнительные настройки
`AddAuthorization`	Сервисы аутентификации и авторизации
`AddDataAnnotations`	Сервис аннотаций данных MVC
`AddCacheTagHelper`	Вспомогательный сервис тегов кеширования MVC
`AddRazorPages`	Сервис Razor Pages, включая механизм просмотра Razor. Обычно используется в простых проектах сайтов. Вызывает следующие дополнительные методы: • `AddMvcCore`; • `AddAuthorization`; • `AddDataAnnotations`; • `AddCacheTagHelper`
`AddApiExplorer`	Сервис обозревателя Web API
`AddCors`	Поддержка совместного использования ресурсов между разными источниками (CORS) в целях повышения безопасности

Метод	Регистрируемые сервисы
`AddFormatterMappings`	Сопоставления между форматом URL и соответствующим типом данных
`AddControllers`	Сервисы контроллера, но не сервисы для представлений или страниц. Обычно используется в проектах ASP.NET Core Web API. Вызывает следующие дополнительные методы: • `AddMvcCore`; • `AddAuthorization`; • `AddDataAnnotations`; • `AddCacheTagHelper`; • `AddApiExplorer`; • `AddCors`; • `AddFormatterMappings`
`AddViews`	Поддержка представлений `.cshtml`, включая соглашения по умолчанию
`AddRazorViewEngine`	Поддержка механизма просмотра Razor, включая обработку символа `@`
`AddControllersWithViews`	Контроллер, представления и сервисы страниц. Обычно используется в проектах сайтов ASP.NET Core MVC. Вызывает следующие дополнительные методы: • `AddMvcCore`; • `AddAuthorization`; • `AddDataAnnotations`; • `AddCacheTagHelper`; • `AddApiExplorer`; • `AddCors`; • `AddFormatterMappings`; • `AddViews`; • `AddRazorViewEngine`
`AddMvc`	Аналогичен `AddControllersWithViews`, но вы должны использовать его только для обратной совместимости
`AddDbContext<T>`	Ваш тип `DbContext` и его необязательный `DbContextOptions<TContext>`
`AddNorthwindContext`	Пользовательский метод расширения, который мы создали, чтобы упростить регистрацию класса `NorthwindContext` для SQLite или SQL Server в зависимости от проекта, на который ссылаются

Больше примеров использования этих методов расширения для регистрации сервисов вы увидите в следующих нескольких главах, когда мы будем работать с сервисами MVC и Web API.

Конфигурация конвейера HTTP-запросов в методе ConfigureServices

Метод `Configure` настраивает конвейер HTTP-запросов, состоящий из связанной последовательности делегатов, которые могут выполнять обработку, а затем принимать решение либо вернуть ответ самостоятельно, либо передать обработку следующему делегату в конвейере. Возвращающиеся ответы также можно обрабатывать.

Помните, что делегаты определяют сигнатуру метода, к которой может подключиться реализация делегата. Делегат для конвейера HTTP-запросов прост:

```
public delegate Task RequestDelegate(HttpContext context);
```

Вы можете видеть, что входным параметром является `HttpContext`. Благодаря этому вы получаете доступ ко всему, что вам может понадобиться для обработки входящего HTTP-запроса, включая путь URL, параметры строки запроса, файлы cookie и пользовательский агент.

Эти делегаты часто называются промежуточным программным обеспечением, так как находятся между клиентом браузера и сайтом или сервисом.

Делегаты промежуточного ПО можно настроить с помощью одного из указанных ниже методов или специального метода, который вызывает их сам:

- `Run` добавляет делегат промежуточного программного обеспечения, который завершает конвейер, немедленно возвращая ответ, вместо того чтобы вызывать следующий делегат промежуточного ПО;
- `Map` добавляет делегат промежуточного программного обеспечения, который создает ветвь в конвейере при наличии соответствующего запроса, основанного на пути URL, например `/hello`;
- `Use` добавляет делегат промежуточного программного обеспечения, который образует часть конвейера, чтобы определить необходимость передачи запроса следующему делегату в конвейере и изменить запрос и ответ до и после следующего делегата.

Для удобства использования существует множество методов расширения, которые упрощают построение конвейера, например `UseMiddleware<T>`, где `T` — это класс, содержащий:

1) конструктор с параметром `RequestDelegate`, который передается следующему компоненту конвейера;
2) метод `Invoke` с параметром `HttpContext` и возвращающий `Task`.

Обобщение ключевых методов расширения промежуточного программного обеспечения

Ключевые методы расширения промежуточного ПО, используемые в нашем коде, включают следующее:

- `UseDeveloperExceptionPage` захватывает синхронные и асинхронные экземпляры `System.Exception` из конвейера и генерирует ответы об ошибках HTML;
- `UseHsts` включает промежуточное программное обеспечение для использования механизма HSTS, добавляющего заголовок `Strict-Transport-Security`;
- `UseRouting` добавляет промежуточное ПО, которое определяет точку в конвейере, где принимаются решения о маршрутизации, и должно сочетаться с вызовом метода `UseEndpoints`, где затем выполняется обработка. Это означает, что для нашего кода любые URL-пути, соответствующие `/`, или `/index`, или `/suppliers`, будут сопоставлены с Razor Pages, а соответствующие `/hello` будут сопоставлены с анонимным делегатом. Любые другие URL-пути будут переданы следующему делегату для сопоставления, например, статических файлов. Вот почему, несмотря на то что сопоставление Razor Pages и `/hello` происходит после статических файлов в конвейере, на самом деле они имеют приоритет, так как вызов `UseRouting` происходит до вызова метода `UseStaticFiles`;
- `UseHttpsRedirection` добавляет промежуточное программное обеспечение для перенаправления HTTP-запросов на HTTPS, поэтому в нашем коде запрос `http://localhost:5000` будет изменен на `https://localhost:5001`;
- `UseDefaultFiles` добавляет промежуточное ПО, позволяющее отображать файлы по умолчанию на текущем пути, поэтому в нашем коде будет идентифицировать файлы, такие как `index.html`;
- `UseStaticFiles` добавляет промежуточное программное обеспечение, которое для возврата в ответе HTTP ищет в `wwwroot` статические файлы;
- `UseEndpoints` добавляет промежуточное ПО для генерации ответов от запросов, принятых ранее в конвейере. Добавляются две конечные точки, как показано ниже:
  - `MapRazorPages` добавляет промежуточное ПО, которое будет сопоставлять URL-пути, такие как `/suppliers`, с файлом `suppliers.cshtml` страницы Razor, расположенным в папке `/Pages`, и возвращать результаты в виде HTTP-ответа;
  - `MapGet` добавляет промежуточное ПО, которое будет сопоставлять URL-пути, такие как `/hello`, со встроенным делегатом, записывающим простой текст непосредственно в HTTP-ответ.

Визуализация HTTP-конвейера

Конвейер HTTP-запроса и ответа можно визуализировать как последовательность делегатов запроса, вызываемых один за другим. Диаграмма, показанная на рис. 14.16, исключает некоторые делегаты промежуточного программного обеспечения, такие как `UseHsts`.

Рис. 14.16. Конвейер HTTP-запросов и ответов

Как уже ранее упоминалось, методы `UseRouting` и `UseEndpoints` необходимо использовать вместе. Хотя код для определения сопоставленных путей, например `/hello`, написан в `UseEndpoints`, решение о том, совпадает ли URL-путь входящего HTTP-запроса и, следовательно, какую конечную точку выполнять, принимается в точке `UseRouting` в конвейере.

Реализация анонимного встроенного делегата в качестве промежуточного программного обеспечения

Делегат можно определить как встроенный анонимный метод. Мы зарегистрируем тот, который подключается к конвейеру после того, как будут приняты решения о маршрутизации для конечных точек.

Он выведет, какая конечная точка была выбрана, а также обработает один конкретный маршрут: `/bonjour`. Если маршрут совпадает, то выдаст в качестве ответа обычный текст, не вызывая никаких дальнейших действий в конвейере.

1. В файле `Startup.cs` статически импортируйте `Console`:

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

2. Добавьте операторы после вызова метода `UseRouting` и перед вызовом метода `UseHttpsRedirection` для использования анонимного метода в качестве делегата промежуточного программного обеспечения:

```
app.Use(async (HttpContext context, Func<Task> next) =>
{
  RouteEndpoint? Rep = context.GetEndpoint() as RouteEndpoint;
  if (rep is not null)
  {
    WriteLine($"Endpoint name: {rep.DisplayName}");
    WriteLine($"Endpoint route pattern: {rep.RoutePattern.RawText}");
  }

  if (context.Request.Path == "/bonjour")
  {
    // в случае совпадения URL-пути становится возвращаемым
    // завершающим делегатом, поэтому следующий делегат
    // не вызывается
    await context.Response.WriteAsync("Bonjour Monde!");
    return;
  }

  // можно изменить запрос перед вызовом следующего делегата
  await next();
  // можно изменить ответ после вызова следующего делегата
});
```

3. Запустите сайт.
4. В браузере Google Chrome перейдите по адресу https://localhost:5001/ и обратите внимание, что было совпадение на маршруте к конечной точке /, оно было обработано как `/index`, а страница Razor `Index.cshtml` была выполнена для возврата ответа:

```
Endpoint name: /index
Endpoint route pattern:
```

5. Перейдите по адресу https://localhost:5001/supplier и обратите внимание, что было совпадение на маршруте конечной точки `/supplier`, а страница Razor `Supplier.cshtml` была выполнена для возврата ответа:

```
Endpoint name: /Suppliers
Endpoint route pattern: Suppliers
```

6. Перейдите по адресу https://localhost:5001/index и обратите внимание, что было совпадение на маршруте конечной точки `/index`, а страница Razor `Index.cshtml` была выполнена для возврата ответа:

```
Endpoint name: /index
Endpoint route pattern: index
```

7. Перейдите по адресу https://localhost:5001/index.html и обратите внимание, что вывод не записывается в консоль, поскольку не было совпадения на маршруте конечной точки, но было совпадение для статического файла, поэтому он был возвращен в качестве ответа.
8. Перейдите по адресу https://localhost:5001/bonjour и обратите внимание, что в консоль не выводится результат, поскольку не было совпадения на маршруте конечной точки. Вместо этого наш делегат выполнил сопоставление с `/bonjour`, записал данные непосредственно в поток ответа и вернул их без дополнительной обработки.
9. Закройте браузер Google Chrome и завершите работу веб-сервера.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 14.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Каковы шесть методов, которые могут быть указаны в HTTP-запросе?
2. Какие шесть кодов состояния и их описания могут быть возвращены в HTTP-ответе?
3. Для чего в ASP.NET Core используется класс `Startup`?
4. Что такое HSTS и для чего он предназначен?
5. Как включить статические HTML-страницы для сайта?
6. Как вставить код C# в HTML, чтобы создать динамическую страницу?
7. Как определить общие макеты для Razor Pages?
8. Как отделить разметку от выделенного кода на странице Razor?
9. Как настроить контекст данных Entity Framework Core для использования с сайтом ASP.NET Core?
10. Как повторно использовать Razor Pages в ASP.NET Core 2.2 или более поздней версии?

Упражнение 14.2. Веб-страница, управляемая данными

Добавьте на сайт Northwind.Web страницу Razor, позволяющую пользователю просматривать список клиентов, сгруппированных по странам. Когда пользователь щелкает на записи клиента, отображается страница, на которой указаны полные контактные данные выбранного клиента и список его заказов.

Упражнение 14.3. Создание веб-страниц для консольных приложений

Реализуйте повторно как Razor Pages некоторые консольные приложения, рассмотренные в предыдущих главах, например в главе 4: предоставьте пользовательский веб-интерфейс для вывода таблиц умножения, расчета налогов и генерации факториалов и последовательности Фибоначчи.

Упражнение 14.4. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-14---building-websites-using-aspnet-core-razor-pages, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали об основах веб-разработки с помощью HTTP, научились создавать простой сайт, возвращающий статические файлы, и использовали Razor Pages в ASP.NET Core совместно с Entity Framework Core для создания веб-страниц, которые динамически генерировались из информации, расположенной в базе данных.

Мы рассмотрели конвейер HTTP-запросов и ответов, работу вспомогательных методов расширения и варианты добавления собственного промежуточного программного обеспечения, влияющего на обработку.

В следующей главе вы научитесь создавать более сложные сайты, используя ASP.NET Core MVC, который разделяет технические аспекты создания сайта на модели, представления и контроллеры, чтобы упростить управление ими.

15 Разработка сайтов с помощью паттерна MVC

Данная глава посвящена разработке сайтов с современной HTTP-архитектурой на стороне сервера с помощью Microsoft ASP.NET Core MVC. Вы узнаете о конфигурации запуска, аутентификации, авторизации, о маршрутах, моделях, представлениях и контроллерах, составляющих проект ASP.NET Core MVC.

В этой главе:

- настройка сайта ASP.NET Core MVC;
- изучение сайта ASP.NET Core MVC;
- добавление собственного функционала на сайт ASP.NET Core MVC;
- запрос к базе данных и использование шаблонов отображения;
- улучшение масштабируемости с помощью асинхронных задач.

Настройка сайта ASP.NET Core MVC

Технология Razor Pages от ASP.NET Core отлично подходит для разработки простых сайтов. Более сложным сайтам было бы лучше иметь более формальную структуру, позволяющую управлять этой сложностью.

Именно здесь будет полезен паттерн проектирования «Модель — представление — контроллер» (Model-View-Controller, MVC). Он использует технологии, аналогичные Razor Pages, но обеспечивает более четкое разделение между техническими компонентами, как показано в следующем перечне.

- *Модели* — классы, представляющие сущности данных и модели представления, которые используются на сайте.
- *Представления* — файлы Razor, то есть файлы `.cshtml`, которые преобразуют данные в моделях представлений в веб-страницы HTML. Фреймворк Blazor использует расширение файла `.razor`, но это не то же, что файлы Razor!
- *Контроллеры* — классы, которые выполняют код при поступлении HTTP-запроса на веб-сервер. Методы контроллера обычно создают модель представления, которая может содержать сущностные модели, и передают ее представлению, чтобы генерировать HTTP-ответ для отправки обратно в браузер или другой клиент.

Лучший способ разобраться в том, как использовать паттерн проектирования MVC для веб-разработки, — увидеть рабочий пример.

Создание сайтов ASP.NET Core MVC

С помощью шаблона проекта мы создадим проект сайта ASP.NET Core MVC с базой данных для аутентификации и авторизации пользователей. Программа Visual Studio 2022 по умолчанию использует базу данных учетных записей SQL Server LocalDB. Программа Visual Studio Code (или, точнее, инструмент `dotnet`) по умолчанию использует SQLite, и вы можете выбрать вариант SQL Server LocalDB вместо SQLite.

Посмотрим, как это работает.

1. Откройте редактор кода и создайте проект сайта MVC с учетными записями, хранящимися в базе данных, с такими настройками:
   1) шаблон проекта: ASP.NET Core Web App (Model-View-Controller)/`mvc`;
   2) язык: `C#`;
   3) файл и папка рабочей области/решения: `PracticalApps`;
   4) файл и папка проекта: `Northwind.Mvc`;
   5) параметры: Authentication Type: Individual Accounts (Тип аутентификации: Индивидуальные учетные записи)/`--auth Individual`;
   6) в программе Visual Studio оставьте все остальные параметры по умолчанию.
2. В программе Visual Studio Code выберите `Northwind.Mvc` в качестве активного проекта OmniSharp.
3. Соберите проект `Northwind.Mvc`.
4. В командной строке или в терминале используйте ключевое слово `help`, чтобы просмотреть другие параметры для этого шаблона проекта:

```
dotnet new mvc --help
```

5. Обратите внимание на результаты:

```
ASP.NET Core Web App (Model-View-Controller) (C#)
Author: Microsoft
Description: A project template for creating an ASP.NET Core application
with example ASP.NET Core MVC Views and Controllers. This template can
also be used for RESTful HTTP services.
This template contains technologies from parties other than Microsoft, see
https://aka.ms/aspnetcore/6.0-third-party-notices for details.
```

Существует множество опций, связанных с аутентификацией, как показано в табл. 15.1.

Таблица 15.1. Опции, связанные с аутентификацией

Переключатели	Описание
`-au|--auth`	Тип используемой аутентификации: • `None` (по умолчанию) — этот вариант также позволяет отключить HTTPS; • `Individual` — индивидуальная аутентификация, которая хранит зарегистрированных пользователей и их пароли в базе данных (по умолчанию SQLite). Мы будем использовать ее в проекте, который создадим для этой главы; • `IndividualB2C` — индивидуальная аутентификация с помощью Azure AD B2C; • `SingleOrg` — организационная аутентификация для одного клиента; • `MultiOrg` — организационная аутентификация для нескольких клиентов; • `Windows` — аутентификация Windows. В основном полезна для интрасетей
`-uld|--use-local-db`	Помогает установить, следует ли использовать SQL Server LocalDB вместо SQLite. Этот параметр применяется, только если выбран вариант `--auth Individual` или `--auth IndividualB2C`. Значение является необязательным `bool` и по умолчанию `false`
`-rrc|--razor-runtime-compilation`	Определяет, настроен ли проект на компиляцию времени выполнения Razor в отладочных сборках `Debug`. Это может повысить производительность запуска во время отладки, поскольку откладывается компиляция представлений Razor. Значение является необязательным `bool` и по умолчанию `false`
`-f|--framework`	Целевой фреймворк для проекта. Значения могут быть следующими: `net6.0` (по умолчанию), `net5.0` или `netcoreapp3.1`

Создание базы данных аутентификации для SQL Server LocalDB

Если вы создали проект MVC в программе Visual Studio 2022 или использовали команду `dotnet new mvc` с переключателем `-uld` или `--use-local-db`, то база данных для аутентификации и авторизации будет храниться в SQL Server LocalDB. Но база данных еще не создана. Сделаем это сейчас.

В командной строке или в терминале в папке Northwind.Mvc введите команду для запуска миграций базы данных, чтобы создать базу данных, используемую для хранения учетных данных для аутентификации:

```
dotnet ef database update
```

Если вы создали проект MVC с помощью команды `dotnet new`, то база данных для аутентификации и авторизации будет храниться в SQLite, а файл `app.db` уже создан.

Строка подключения для базы данных аутентификации называется `DefaultConnection` и хранится в файле `appsettings.json` в корневой папке проекта сайта MVC.

Для SQL Server LocalDB (с усеченной строкой подключения) воспользуйтесь следующим кодом:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-
    Northwind.Mvc-...;Trusted_Connection=True;MultipleActiveResultSets=true"
},
```

Для SQLite воспользуйтесь следующим кодом:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "DataSource=app.db;Cache=Shared"
},
```

Сайт ASP.NET Core MVC по умолчанию

Рассмотрим поведение шаблона проекта сайта ASP.NET Core MVC по умолчанию.

1. В проекте `Northwind.Mvc` раскройте папку Properties, откройте файл `launchSettings.json` и обратите внимание на случайные номера портов (ваши будут другими), настроенные для проекта для `HTTPS` и `HTTP`:

   ```
   "profiles": {
     "Northwind.Mvc": {
       "commandName": "Project",
       "dotnetRunMessages": true,
       "launchBrowser": true,
       "applicationUrl": "https://localhost:7274;http://localhost:5274",
       "environmentVariables": {
         "ASPNETCORE_ENVIRONMENT": "Development"
       }
     },
   ```

2. Измените номера портов на `5001` для HTTPS и `5000` для HTTP:

   ```
   "applicationUrl": "https://localhost:5001;http://localhost:5000",
   ```

3. Сохраните изменения в файле `launchSettings.json`.
4. Запустите сайт.
5. Запустите браузер Google Chrome и откройте панель Developer Tools (Инструменты разработчика).

6. Перейдите по адресу http://localhost:5000/. Обратите внимание на следующие моменты, показанные на рис. 15.1:
   - HTTP-запросы автоматически перенаправляются на HTTPS на порт 5001;
   - верхнее меню навигации содержит ссылки: Home (Главная страница), Privacy (Конфиденциальность), Register (Регистрация) и Login (Войти). Если ширина области просмотра составляет 575 пикселей или меньше, то навигация сворачивается в меню «гамбургер»;
   - название сайта, Northwind.Mvc, отображается в верхнем и нижнем колонтитулах.

Рис. 15.1. Главная страница сайта шаблона проекта ASP.NET Core MVC

Регистрация посетителей

По умолчанию пароли должны содержать хотя бы один не алфавитно-цифровой символ, одну цифру (от 0 до 9) и хотя бы одну прописную букву (A — Z). Я использую `Pa$$w0rd` в подобных ситуациях, когда просто изучаю тему.

Шаблон проекта MVC соответствует рекомендациям по *двойному согласию* (double-opt-in, DOI). Это значит, после ввода адреса электронной почты и пароля для регистрации на указанный адрес отправляется письмо. Пользователь должен нажать ссылку в нем, чтобы подтвердить регистрацию.

Мы еще не настроили поставщик электронной почты для отправки этого письма, поэтому должны симулировать данный шаг.

1. В верхней части меню навигации выберите пункт Register (Зарегистрироваться).
2. Введите адрес электронной почты и пароль, а затем нажмите кнопку Register (Зарегистрироваться). (Я использовал варианты `test@example.com` и `Pa$$w0rd`.)
3. Нажмите ссылку с текстом Click here to confirm your account (Нажмите здесь, чтобы подтвердить свою учетную запись) и обратите внимание, что вы перенаправ-

лены на веб-страницу Confirm email (Подтвердить адрес электронной почты), которую вы можете изменить.

4. В верхней части меню навигации нажмите кнопку Login (Войти), введите свой адрес электронной почты и пароль. (Обратите внимание, что дополнительный флаг служит для сохранения пароля. Здесь же содержатся ссылки на случай, если пользователь забыл пароль или хочет зарегистрироваться в качестве нового пользователя.) Затем нажмите кнопку Log in (Войти).
5. Нажмите свой адрес электронной почты в верхней части меню навигации. Вы перейдете на страницу управления учетной записью. Обратите внимание, что вы можете указать номер телефона, изменить адрес электронной почты, пароль, включить двухуровневую аутентификацию (если добавите приложение для аутентификации), скачать и удалить свои личные данные.
6. Закройте браузер Google Chrome и завершите работу веб-сервера.

Структура проекта сайта MVC

В редакторе кода программы Visual Studio на панели Solution Explorer (Проводник решений) (активируйте параметр Show All Files (Показывать все файлы)) или в программе Visual Studio Code на панели EXPLORER (Проводник) рассмотрите структуру проекта сайта MVC (рис. 15.2).

Рис. 15.2. Структура папок по умолчанию проекта ASP.NET Core MVC

Позже мы более подробно рассмотрим некоторые из представленных компонентов, а пока отметим следующие моменты:

- папка Areas содержит файлы, необходимые для интеграции проекта сайта с системой *ASP.NET Core Identity*, которая используется для проверки подлинности;
- папки bin и obj вмещают временные файлы, необходимые в процессе сборки, и скомпилированные сборки для проекта;
- папка Controllers содержит классы C# с методами (известными как действия), которые получают *модель* и передают ее *представлению*, например `HomeController.cs`;
- папка Data вмещает классы миграции Entity Framework Core, с помощью которых система ASP.NET Core Identity предоставляет хранилище данных для аутентификации и авторизации, например `ApplicationDbContext.cs`;
- папка Models содержит классы C#, которые представляют данные, собираемые контроллером и передаваемые в представление, например `ErrorViewModel.cs`;
- папка Properties включает файл конфигурации `launchSettings.json` для IIS или IIS Express в Windows и запуска сайта во время разработки. Этот файл используется только на локальной машине разработки и не может быть развернут на вашем рабочем сайте;
- папка Views содержит файлы Razor с расширением `.cshtml`, которые комбинируют код HTML и C# в целях динамического генерирования HTML-ответов. Файл `_ViewStart` устанавливает макет по умолчанию, а `_ViewImports` импортирует общие пространства имен, используемые во всех представлениях, например вспомогательные теги:
  - подпапка Home содержит файлы Razor для главной страницы и страниц конфиденциальности;
  - подпапка Shared вмещает файлы Razor для общего макета, страницу ошибок и два частичных представления для входа в систему и сценариев проверки;
- папка wwwroot содержит статический контент, используемый на сайте, такой как CSS-файлы для хранения стилей, библиотеки JavaScript, JavaScript для этого проекта сайта и файл `favicon.ico`. Вы также можете поместить сюда изображения и другие статические файловые ресурсы, такие как документы в формате PDF. Шаблон проекта включает в себя библиотеки Bootstrap и jQuery;
- в базе данных SQLite app.db хранятся зарегистрированные пользователи (если вы использовали SQL Server LocalDB, то она не понадобится);
- файлы `appsettings.json` и `appsettings.Development.json` содержат параметры, которые ваш сайт может загрузить во время выполнения, например строку подключения к базе данных для системы ASP.NET Identity и уровни логирования;
- файл `Northwind.Mvc.csproj` содержит параметры проекта, такие как использование веб-пакета SDK для .NET, запись для SQLite, гарантирующую, что файл

app.db будет скопирован в выходную папку сайта, и список пакетов NuGet, которые требуются вашему проекту, в том числе:

- Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore;
- Microsoft.AspNetCore.Identity.EntityFrameworkCore;
- Microsoft.AspNetCore.Identity.UI;
- Microsoft.EntityFrameworkCore.Sqlite или Microsoft.EntityFrameworkCore.SqlServer;
- Microsoft.EntityFrameworkCore.Tools;

- файл Program.cs определяет скрытый класс Program, который содержит основную точку входа. Он создает конвейер для обработки входящих HTTP-запросов, и использует для хостинга сайта параметры по умолчанию, такие как настройка веб-сервера Kestrel и загрузка настроек приложений. Файл добавляет и настраивает сервисы, необходимые вашему сайту, например ASP.NET Core Identity для аутентификации, SQLite или SQL Server для хранения идентификационных данных и т. д., а также определяет связи между компонентами в вашем приложении.

Обзор базы данных ASP.NET Core Identity

Откройте файл appsettings.json и найдите строку подключения, используемую для базы данных ASP.NET Core Identity (выделена жирным шрифтом в примере SQL Server LocalDB):

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-
    Northwind.Mvc-2F6A1E12-F9CF-480C-987D-FEFB4827DE22;Trusted_Connection=True;
    MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Если для хранения идентификационных данных вы использовали SQL Server LocalDB, то для подключения к базе данных можно использовать Server Explorer (Проводник сервера). Вы можете скопировать строку подключения из файла appsettings.json (но уберите второй обратный слеш между (localdb) и mssqllocaldb) и вставить ее.

Если вы установили инструмент SQLite, такой как SQLiteStudio, то можете открыть файл базы данных SQLite `app.db`.

Затем вы можете просмотреть таблицы, с помощью которых система ASP.NET Core Identity регистрирует пользователей и роли, включая таблицу `AspNetUsers`, применяемую для хранения данных о зарегистрированных пользователях.

Шаблон проекта ASP.NET Core MVC следует обычной практике, храня хеш пароля вместо самого пароля (мы поговорим об этом в главе 20).

Работа сайта ASP.NET Core MVC

Рассмотрим компоненты, из которых состоит современный сайт ASP.NET Core MVC.

Инициализация ASP.NET Core MVC

Начнем с изучения инициализации и конфигурации по умолчанию сайта MVC.

1. Откройте файл `Program.cs` и обратите внимание, что в нем используется функция программы верхнего уровня (поэтому имеется скрытый класс `Program` с методом `Main`). Можно считать, что этот файл разбит на четыре важных раздела сверху вниз.

В .NET 5 и более ранних проектах ASP.NET Core использовался класс Startup для разделения этих частей на отдельные методы, но в .NET 6 Microsoft рекомендует поместить все в один файл Program.cs.

2. В первом разделе импортируются некоторые пространства имен:

```
using Microsoft.AspNetCore.Identity; // IdentityUser
using Microsoft.EntityFrameworkCore; // UseSqlServer, UseSqlite
using Northwind.Mvc.Data; // ApplicationDbContext
```

Помните, что по умолчанию многие другие пространства имен импортируются с помощью функции неявного использования в .NET 6 и более поздних версиях. Соберите проект, после чего глобально импортированные пространства имен можно найти по следующему пути: obj\Debug\net6.0\Northwind.Mvc.GlobalUsings.g.cs.

3. Во втором разделе создается и настраивается конструктор веб-хостинга. Он регистрирует контекст базы данных приложения, используя SQL Server или

SQLite со строкой подключения к базе, загруженной из файла `appsettings.json`, далее добавляет ASP.NET Core Identity в целях аутентификации и настраивает ее для использования базы данных приложения. Кроме того, добавляется поддержка контроллеров MVC с представлениями:

```
var builder = WebApplication.CreateBuilder(args);

// добавление сервисов в контейнер
var connectionString = builder.Configuration
  .GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<ApplicationDbContext>(options =>
  options.UseSqlServer(connectionString)); // or UseSqlite

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
  options.SignIn.RequireConfirmedAccount = true)
  .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddControllersWithViews();
```

Объект `builder` имеет два часто используемых объекта:

- `Configuration` содержит объединенные значения из всех мест, где вы можете задать конфигурацию: файл `appsettings.json`, переменные среды, аргументы командной строки и т. д.;
- `Services` представляет собой коллекцию зарегистрированных сервисов зависимостей.

Вызов метода `AddDbContext` — пример регистрации сервиса зависимости. ASP.NET Core реализует паттерн проектирования *«Внедрение зависимостей»* (Dependency Injection, DI), чтобы другие компоненты, такие как контроллеры могли запрашивать необходимые сервисы через свои конструкторы.

Разработчики регистрируют сервисы в этом разделе файла `Program.cs` (или, если используется класс `Startup`, в его методе `ConfigureServices`).

4. В третьем разделе настраивается конвейер HTTP-запросов. В нем задается относительный путь URL для запуска миграции базы данных, если сайт находится в разработке, или более удобная страница ошибок и HSTS для производственной среды. Включены HTTPS-перенаправление, статические файлы, маршрутизация и ASP.NET Identity, а также настроены маршрут MVC по умолчанию и Razor Pages:

```
// настройка конвейера HTTP-запросов
if (app.Environment.IsDevelopment())
{
  app.UseMigrationsEndPoint();
}
```

```
else
{
  app.UseExceptionHandler("/Home/Error");
  // Значение HSTS по умолчанию — 30 дней. Вы можете изменить это значение
  // в рабочих сценариях, см. https://aka.ms/aspnetcore-hsts.
  app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapControllerRoute(
  name: "default",
  pattern: "{controller=Home}/{action=Index}/{id?}");

app.MapRazorPages();
```

В главе 14 мы изучили большинство из этих методов и функций.

Задаетесь вопросом, как функционирует метод расширения UseMigrationsEndPoint? Вы можете ознакомиться с официальной документацией, но это мало чем поможет. Например, в ней не сказано, какой относительный путь URL определяется по умолчанию: https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.aspnetcore.builder.migrationsendpointextensions.usemigrationsendpoint. К счастью, ASP.NET Core имеет открытый исходный код, поэтому мы можем проанализировать его и узнать, как он работает, по следующей ссылке: https://github.com/dotnet/aspnetcore/blob/main/src/Middleware/Diagnostics.EntityFrameworkCore/src/MigrationsEndPointOptions.cs#L18. Изучайте исходный код ASP.NET Core, чтобы разобраться, как он работает.

Помимо методов `UseAuthentication` и `UseAuthorization`, наиболее важным новым методом в этом разделе файла `Program.cs` является `MapControllerRoute`, настраивающий маршрут по умолчанию на использование MVC. Этот маршрут очень гибкий, поскольку подходит практически для любого входящего URL (мы рассмотрим эту тему чуть позже).

Хотя в этой главе мы не будем создавать Razor Pages, нам необходимо оставить вызов метода, отражающего поддержку страницы Razor, поскольку наш сайт MVC использует ASP.NET Core Identity для аутентификации и авторизации, а также библиотеку классов Razor для компонентов пользовательского интерфейса, таких как регистрация посетителей и вход в систему.

5. Четвертый и последний раздел содержит вызов метода с блокировкой потоков, который запускает сайт и ожидает ответа на входящие HTTP-запросы:

```
app.Run(); // блокировка вызова
```

Маршрутизация MVC по умолчанию

Маршрутизация заключается в том, чтобы обнаружить имя класса контроллера для создания экземпляра и метод действия для выполнения с необязательным параметром `id` для передачи в метод, который будет генерировать ответ HTTP.

Маршрут по умолчанию настроен для MVC:

```
endpoints.MapControllerRoute(
  name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Части маршрута, записанные в фигурных скобках `{}`, называются *сегментами* и подобны именованным параметрам метода. Значением этих сегментов может быть любая строка. Сегменты в URL не чувствительны к регистру.

Шаблон маршрута просматривает любой URL-путь, запрошенный браузером, и сопоставляет его, чтобы извлечь имя контроллера, имя действия и необязательное значение `id` (символ `?` делает его необязательным).

Если пользователь не указал эти данные, то применяются значения по умолчанию: `Home` в качестве контроллера и `Index` в качестве действия (присвоение через `=` устанавливает значение по умолчанию для именованного сегмента).

В табл. 15.2 приведены примеры URL и то, как маршрут по умолчанию будет обрабатывать имена контроллера и действия.

Таблица 15.2. Примеры URL и обработка маршрутом по умолчанию

URL	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`

Контроллеры и действия

В MVC символ C обозначает контроллер. По маршруту и входящему URL ASP.NET Core определит имя контроллера, поэтому будет искать класс, дополненный атрибутом `[Controller]` или наследуемый от класса, дополненного этим атрибутом, — например, от показанного ниже класса `ControllerBase`:

```
namespace Microsoft.AspNetCore.Mvc
{
  //
  // резюме:
  // базовый класс для контроллера MVC без поддержки представления
  [Controller]
  public abstract class ControllerBase
  {
...
```

Класс ControllerBase

Как видно из комментария XML, класс `ControllerBase` не поддерживает представления. Он используется для создания веб-сервисов, как вы увидите в главе 16.

Класс `ControllerBase` имеет множество полезных свойств для работы с текущим контекстом HTTP (табл. 15.3).

Таблица 15.3. Свойства класса ControllerBase

Свойство	Описание
`Request`	Только HTTP-запрос. Например, заголовки, параметры строки запроса, считываемое тело запроса в виде потока, тип и длина контента, а также cookie-файлы
`Response`	Только HTTP-ответ. Например, заголовки, подходящее для записи тело ответа в виде потока, тип и длина контента, код состояния и cookie-файлы. Кроме того, существуют делегаты, такие как `OnStarting` и `OnCompleted`, к которым можно подключить метод
`HttpContext`	Все о текущем контексте HTTP, включая запрос и ответ, информацию о соединении, набор функций, которые были включены на сервере с помощью промежуточного программного обеспечения, и объект `User` для аутентификации и авторизации

Класс Controller

Microsoft предоставила еще один класс `Controller`, который могут наследовать ваши классы, если нужна поддержка представления:

```
namespace Microsoft.AspNetCore.Mvc
{
```

```
  //
  // резюме:
  // базовый класс для контроллера MVC с поддержкой представления
  public abstract class Controller : ControllerBase,
    IActionFilter, IFilterMetadata, IAsyncActionFilter, IDisposable
  {
...
```

Класс `Controller` имеет множество полезных свойств для работы с представлениями (табл. 15.4).

Таблица 15.4. Свойства класса Controller

Свойство	Описание
`ViewData`	Словарь, в котором контроллер может хранить пары «ключ — значение», доступные в представлении. Время жизни словаря указано только для текущего запроса/ответа
`ViewBag`	Динамический объект, который обертывает `ViewData`, обеспечивая более удобный синтаксис для установки и получения значений словаря
`TempData`	Словарь, в котором контроллер может хранить пары «ключ — значение», доступные в представлении. Время жизни словаря указано для текущего запроса/ответа и следующего запроса/ответа для того же сеанса посетителя. Это полезно для хранения значения во время первоначального запроса, ответа с перенаправлением и дальнейшего считывания сохраненного значения в последующем запросе

Класс `Controller` имеет множество полезных методов для работы с представлениями (табл. 15.5).

Таблица 15.5. Методы класса Controller

Метод	Описание
`View`	Возвращает `ViewResult` после выполнения представления, которое отображает полный ответ, например динамически сгенерированную веб-страницу. Представление можно выбрать, используя соглашение или задав строковое имя. Представлению может быть передана модель
`PartialView`	Возвращает `PartialViewResult` после выполнения представления, которое является частью полного ответа, например динамически сгенерированного фрагмента HTML. Представление можно выбрать, используя соглашение или задав строковое имя. Представлению может быть передана модель
`ViewComponent`	Возвращает `ViewComponentResult` после выполнения компонента, который динамически генерирует HTML. Компонент следует выбрать, указав его тип или имя. В качестве аргумента может быть передан объект
`Json`	Возвращает `JsonResult`, содержащий объект, сериализованный в формате JSON. Это может быть полезно для реализации простого Web API в составе контроллера MVC, который в основном возвращает HTML для просмотра человеком

Обязанности контроллера

Контроллер должен:

- определять в конструкторе (или конструкторах) сервисы, необходимые контроллеру для того, чтобы он находился в допустимом состоянии и правильно функционировал;
- использовать имя действия для определения метода выполнения;
- извлекать параметры из HTTP-запроса;
- использовать параметры для получения любых дополнительных данных, необходимых для построения модели представления и передавать их в соответствующее представление для клиента. Например, если клиент — это браузер, то представление, которое отображает HTML, будет наиболее подходящим. Другие клиенты могут предпочесть альтернативные визуализации, такие как форматы документов, например PDF- или Excel-файл, либо форматы данных, например JSON или XML;
- возвращать результаты из представления клиенту в качестве HTTP-ответа с соответствующим кодом состояния.

Рассмотрим контроллер, который использовался для генерации главной страницы, а также страниц конфиденциальности и ошибок.

1. Разверните папку Controllers.
2. Откройте файл `HomeController.cs`.
3. Обратите внимание на следующие моменты, показанные в коде ниже:
   - импортируются дополнительные пространства имен, к которым я добавил комментарии, чтобы показать, для каких типов они необходимы;
   - скрытое поле, доступное только для чтения, объявляется для хранения ссылки на логер для класса `HomeController`, установленный в конструкторе;
   - все три метода действия вызывают метод `View()` и возвращают клиенту результаты в виде `IActionResult`;
   - метод действия `Error` передает модель представления в свое представление с идентификатором запроса, используемым для трассировки. Ответ об ошибке не будет кэшироваться.

```
using Microsoft.AspNetCore.Mvc; // Controller, IactionResult
using Northwind.Mvc.Models; // ErrorViewModel
using System.Diagnostics; // Activity

namespace Northwind.Mvc.Controllers;
public class HomeController : Controller
```

```
{
  private readonly Ilogger<HomeController> _logger;

  public HomeController(Ilogger<HomeController> logger)
  {
    _logger = logger;
  }

  public IactionResult Index()
  {
    return View();
  }

  public IactionResult Privacy()
  {
    return View();
  }

  [ResponseCache(Duration = 0,
    Location = ResponseCacheLocation.None, NoStore = true)]
  public IactionResult Error()
  {
    return View(new ErrorViewModel { RequestId =
      Activity.Current?.Id ?? HttpContext.TraceIdentifier });
  }
}
```

Если пользователь переходит по пути / или /Home, то получается эквивалент /Home/Index, поскольку это имена по умолчанию для контроллера и действия в маршруте по умолчанию.

Соглашение о пути поиска представлений

Методы Index и Privacy идентичны в реализации, но в результате возвращают разные веб-страницы. Причиной тому *соглашение*. Вызов метода View выполняет поиск файла Razor по разным путям для создания веб-страницы.

Мы намеренно изменим одно из имен страниц, чтобы увидеть пути, поиск по которым осуществляется по умолчанию.

1. В проекте Northwind.Mvc разверните папку Views, а затем папку Home.
2. Переименуйте файл Privacy.cshtml в Privacy2.cshtml.
3. Запустите сайт.
4. Запустите браузер Google Chrome, перейдите по адресу https://localhost:5001/, нажмите кнопку Privacy (Конфиденциальность) и обратите внимание на пути, по которым выполняется поиск представления для визуализации веб-страницы (в том числе в папках Shared для представлений MVC и Razor Pages) (рис. 15.3).

Рис. 15.3. Исключение, показывающее путь поиска по умолчанию для представлений

5. Закройте браузер Google Chrome и завершите работу веб-сервера.
6. Переименуйте файл `Privacy2.cshtml` обратно в `Privacy.cshtml`.

Теперь вы ознакомились с соглашением о пути поиска представлений:

- конкретное представление Razor — `/Views/{controller}/{action}.cshtml`;
- общее представление Razor — `/Views/Shared/{action}.cshtml`;
- общая страница Razor — `/Pages/Shared/{action}.cshtml`.

Ведение журнала

Вы только что видели, что некоторые ошибки перехватываются и записываются в консоль. Таким же образом вы можете выводить сообщения в консоли с помощью регистратора.

1. В папке Controllers, в файле `HomeController.cs`, в методе `Index` добавьте следующие операторы, чтобы использовать регистратор для записи в консоль некоторых сообщений различных уровней:

```
_logger.LogError("This is a serious error (not really!)");
_logger.LogWarning("This is your first warning!");
_logger.LogWarning("Second warning!");
_logger.LogInformation("I am in the Index method of the HomeController.");
```

2. Запустите проект сайта `Northwind.Mvc`.
3. Запустите браузер и перейдите на главную страницу сайта.
4. В командной строке или терминале обратите внимание на сообщения:

```
fail: Northwind.Mvc.Controllers.HomeController[0]
      This is a serious error (not really!)
warn: Northwind.Mvc.Controllers.HomeController[0]
      This is your first warning!
warn: Northwind.Mvc.Controllers.HomeController[0]
      Second warning!
info: Northwind.Mvc.Controllers.HomeController[0]
      I am in the Index method of the HomeController.
```

5. Закройте браузер Google Chrome и завершите работу веб-сервера.

Фильтры

Когда вам понадобится добавить некоторые общие функции для нескольких контроллеров и действий, вы можете использовать или определить собственные фильтры, реализованные в виде класса атрибутов.

Фильтры могут применяться на следующих уровнях:

- на уровне действия путем добавления атрибута к методу действия. Это повлияет только на один метод действия;
- на уровне контроллера путем добавления атрибута к классу контроллера. Это повлияет на все методы контроллера;
- на глобальном уровне путем добавления типа атрибута в коллекцию `Filters` экземпляра `MvcOptions`, который можно использовать для настройки MVC при вызове метода `AddControllersWithViews`:

```
builder.Services.AddControllersWithViews(options =>
  {
    options.Filters.Add(typeof(MyCustomFilter));
  });
```

Использование фильтра для защиты метода действия

Вы можете убедиться, что один конкретный метод действия в классе контроллера может вызываться только членами определенных ролей безопасности. Это можно сделать, дополнив метод атрибутом `[Authorize]`, как показано ниже:

- `[Authorize]` позволяет получить доступ к этому методу действия только аутентифицированным (не анонимным, вошедшим в систему) посетителям;

- `[Authorize(Roles = "Sales,Marketing")]` позволяет получить доступ к этому методу действия только посетителям, которые являются членами указанной роли (ролей).

Рассмотрим пример.

1. В файле `HomeController.cs` импортируйте пространство имен `Microsoft.AspNetCore.Authorization`.
2. Добавьте атрибут к методу `Privacy`, чтобы разрешить доступ только вошедшим в систему пользователям, которые являются членами группы/роли `Administrators`, как показано ниже:

```
[Authorize(Roles = "Administrators")]
public IActionResult Privacy()
```

3. Запустите сайт.
4. Нажмите кнопку Privacy (Конфиденциальность) и обратите внимание, что вы перенаправляетесь на страницу входа в систему.
5. Введите свой адрес электронной почты и пароль.
6. Нажмите кнопку Log in (Войти) и обратите внимание, что вам отказано в доступе.
7. Закройте браузер Google Chrome и завершите работу веб-сервера.

Включение управления ролями и создание роли программным путем

По умолчанию управление ролями не включено в проекте ASP.NET Core MVC, поэтому мы должны сначала включить его перед созданием ролей, а затем создадим контроллер, который программно создаст роль `Administrators` (если она еще не существует) и назначит тестового пользователя на эту роль.

1. В файле `Program.cs` в настройке идентификатора ASP.NET Core Identity и его базы данных добавьте вызов функции `AddRoles` для включения управления ролями, как показано ниже:

```
services.AddDefaultIdentity<IdentityUser>(
  options => options.SignIn.RequireConfirmedAccount = true)
  .AddRoles<IdentityRole>() // включить управление ролями
  .AddEntityFrameworkStores<ApplicationDbContext>();
```

2. В классе `Controllers` добавьте пустой класс контроллера `RolesController.cs` и измените его содержимое:

```
using Microsoft.AspNetCore.Identity; // RoleManager, UserManager
using Microsoft.AspNetCore.Mvc; // Controller, IActionResult
```

```
using static System.Console;

namespace Northwind.Mvc.Controllers;

public class RolesController : Controller
{
  private string AdminRole = "Administrators";

  private string UserEmail = "test@example.com";

  private readonly RoleManager<IdentityRole> roleManager;
  private readonly UserManager<IdentityUser> userManager;

  public RolesController(RoleManager<IdentityRole> roleManager,
    UserManager<IdentityUser> userManager)
  {
    this.roleManager = roleManager;
    this.userManager = userManager;
  }

  public async Task<IActionResult> Index()
  {
    if (!(await roleManager.RoleExistsAsync(AdminRole)))
    {
      await roleManager.CreateAsync(new IdentityRole(AdminRole));
    }

    IdentityUser user = await userManager.FindByEmailAsync(UserEmail);

    if (user == null)
    {
      user = new();
      user.UserName = UserEmail;
      user.Email = UserEmail;
      IdentityResult result = await userManager.CreateAsync(
        user, "Pa$$w0rd");

      if (result.Succeeded)
      {
        WriteLine($"User {user.UserName} created successfully.");
      }
      else
      {
        foreach (IdentityError error in result.Errors)
        {
          WriteLine(error.Description);
        }
      }
    }
```

```
    if (!user.EmailConfirmed)
    {
      string token = await userManager
        .GenerateEmailConfirmationTokenAsync(user);
      IdentityResult result = await userManager
        .ConfirmEmailAsync(user, token);

      if (result.Succeeded)
      {
        WriteLine($"User {user.UserName} email confirmed successfully.");
      }
      else
      {
        foreach (IdentityError error in result.Errors)
        {
          WriteLine(error.Description);
        }
      }
    }

    if (!(await userManager.IsInRoleAsync(user, AdminRole)))
    {
      IdentityResult result = await userManager
        .AddToRoleAsync(user, AdminRole);

      if (result.Succeeded)
      {
        WriteLine($"User {user.UserName} added to {AdminRole} successfully.");
      }
      else
      {
        foreach (IdentityError error in result.Errors)
        {
          WriteLine(error.Description);
        }
      }
    }

    return Redirect("/");
  }
}
```

Обратите внимание на следующие моменты:

- есть два поля для имени роли и электронной почты пользователя;
- конструктор получает и хранит сервисы зависимости зарегистрированного пользователя и диспетчера ролей;
- если роль `Administrators` не существует, то мы используем диспетчер ролей для ее создания;

- мы пытаемся найти тестового пользователя по его электронной почте, создаем его, если он не существует, а затем назначаем пользователя на роль администратора `Administrators`;
- поскольку сайт использует DOI, мы должны сгенерировать токен подтверждения электронной почты и использовать его для подтверждения адреса электронной почты новых пользователей;
- сообщения об успешном завершении и любых ошибках выводятся в консоль;
- вы будете автоматически перенаправлены на главную страницу.

3. Запустите сайт.
4. Нажмите кнопку Privacy (Конфиденциальность) и обратите внимание, что вы перенаправлены на страницу входа в систему.
5. Введите свои адрес электронной почты и пароль. (Я использовал адрес mark@example.com.)
6. Нажмите кнопку Log in (Войти) и обратите внимание, что вам, как и раньше, отказано в доступе.
7. Нажмите Home (Главная страница).
8. В адресной строке вручную введите `roles` в качестве относительного пути URL, как показано в следующей ссылке: https://localhost:5001/roles.
9. Просмотрите в консоли сообщения об успешном завершении:

```
User test@example.com created successfully.
User test@example.com email confirmed successfully.
User test@example.com added to Administrators successfully.
```

10. Нажмите кнопку Logout (Выйти), поскольку вы должны выйти из системы и снова войти в нее, чтобы загрузить свое членство в роли, если оно было создано после того, как вы уже вошли в систему.
11. Попробуйте еще раз получить доступ к странице Privacy (Конфиденциальность), введите адрес электронной почты нового пользователя, который был создан программно, например test@example.com, и пароль. Затем нажмите кнопку Log in (Войти), и у вас должен появиться доступ.
12. Закройте браузер Google Chrome и завершите работу веб-сервера.

Использование фильтра для кэширования ответа

Чтобы улучшить время отклика и масштабируемость, вы можете кэшировать HTTP-ответ, генерируемый методом действия, добавив атрибут `[ResponseCache]` к этому методу.

Вы контролируете, где и насколько долго кэшируется ответ, устанавливая параметры:

- `Duration` — в секундах; устанавливает параметр `max-age` заголовка ответа HTTP, измеряемый в секундах. Обычно выбирают один час (3600 с) и один день (86 400 с);
- `Location` — одно из значений `ResponseCacheLocation`: `Any`, `Client` или `None`; устанавливает заголовок ответа HTTP для управления кэшем;
- `NoStore` — если значение равно `true`, то параметры `Duration` и `Location` игнорируются и задается заголовок ответа HTTP для элемента управления кэшем, равный `no-store`.

Рассмотрим пример.

1. В файле `HomeController.cs` добавьте атрибут к методу `Index`, чтобы кэшировать ответ в течение 10 с в браузере или любых прокси-серверах между сервером и браузером, как показано ниже:

```
[ResponseCache(Duration = 10, Location = ResponseCacheLocation.Any)]
public IActionResult Index()
```

2. В разделе `Views` выберите пункт `Home`, откройте файл `Index.cshtml` и добавьте абзац для вывода текущего времени в длинном формате, чтобы учитывать секунды:

```
<p class="alert alert-primary">@DateTime.Now.ToLongTimeString()</p>
```

3. Запустите сайт.
4. Обратите внимание на время на главной странице.
5. Нажмите кнопку Register (Регистрация).
6. Нажмите кнопку Home (Главная страница) и обратите внимание, что время на главной странице совпадает, поскольку используется кэшированная версия страницы.
7. Нажмите кнопку Register (Регистрация). Подождите не менее 10 с.
8. Перейдите на главную страницу и обратите внимание, что время теперь обновлено.
9. Нажмите кнопку Log in (Войти), введите свой адрес электронной почты и пароль, а затем нажмите кнопку Log in (Войти).
10. Обратите внимание на время на главной странице.
11. Нажмите кнопку Privacy (Конфиденциальность).
12. Перейдите на главную страницу и обратите внимание, что страница не кэшируется.

13. Просмотрите консоль и обратите внимание на предупреждение, объясняющее, что ваше кэширование было переопределено, поскольку посетитель вошел в систему, и в этом сценарии ASP.NET Core использует токены для защиты от подделок, и они не должны кэшироваться:

```
warn: Microsoft.AspNetCore.Antiforgery.DefaultAntiforgery[8]
      The 'Cache-Control' and 'Pragma' headers have been overridden
and set to 'no-cache, no-store' and 'no-cache' respectively to prevent
caching of this response. Any response that uses antiforgery should not
be cached.
```

14. Закройте браузер Google Chrome и завершите работу веб-сервера.

Применение фильтра для переопределения маршрута

Возможно, вы захотите определить упрощенный маршрут для метода действия вместо того, чтобы использовать маршрут по умолчанию.

Например, для отображения страницы конфиденциальности в настоящее время требуется следующий URL, указывающий и контроллер, и действие:

```
https://localhost:5001/home/privacy
```

Мы можем упростить маршрут, как показано в следующей ссылке:

```
https://localhost:5001/private
```

Посмотрим, как это сделать.

1. В файле `HomeController.cs` добавьте атрибут к методу `Privacy`, чтобы определить упрощенный маршрут:

```
[Route("private")]
[Authorize(Roles = "Administrators")]
public IActionResult Privacy()
```

2. Запустите сайт.
3. В адресной строке введите следующий URL:

```
https://localhost:5001/private
```

4. Введите свой адрес электронной почты и пароль, нажмите кнопку Log in (Войти), и обратите внимание, что упрощенный путь показывает страницу Privacy (Конфиденциальность).
5. Закройте браузер Google Chrome и завершите работу веб-сервера.

Сущности и модели представлений

В MVC символ M обозначает модель. Это данные, необходимые для ответа на запрос. Обычно используются два типа моделей: сущностные модели и модели представлений.

Сущностные модели представляют собой сущности в базе данных, таком как SQL Server или SQLite. В зависимости от запроса может потребоваться извлечь оттуда одну или несколько сущностей. Сущностные модели определяются с помощью классов, поскольку могут изменяться и затем использоваться для обновления базового хранилища данных.

Все данные, которые мы хотим показать в ответ на запрос, — это модель MVC, которую иногда называют *моделью представления*, поскольку это *модель*, передающаяся в *представление* для его последующего перевода в формат ответа, такой как HTML или JSON. Модели представления должны быть неизменяемыми, поэтому их обычно определяют с помощью записей.

Например, этот HTTP-запрос `GET` может означать, что браузер запрашивает страницу сведений о товаре для товара под номером 3:

```
http://www.example.com/products/details/3
```

Контроллеру необходимо использовать значение маршрута идентификатора `3`, чтобы извлечь объект для этого товара и передать его представлению, которое затем превратит модель в HTML, чтобы отобразить в браузере.

Представьте: когда пользователь заходит на наш сайт, мы хотим показать ему карусель категорий, список товаров и количество посетителей за определенный месяц.

Мы будем ссылаться на сущностную модель данных Entity Framework Core для базы данных Northwind, созданной в главе 13.

1. В проекте `Northwind.Mvc` добавьте ссылку на проект `Northwind.Common.DataContext` для SQLite или SQL Server:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include=
  "..\Northwind.Common.DataContext.Sqlite\Northwind.Common.DataContext.
  Sqlite.csproj" />
</ItemGroup>
```

2. Соберите проект `Northwind.Mvc` для компиляции его зависимостей.
3. Если вы используете SQL Server или, возможно, захотите переключаться между SQL Server и SQLite, то в файле `appsettings.json` добавьте строку подключения для базы данных Northwind с помощью SQL Server, как показано ниже:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-
    Northwind.Mvc-DC9C4FAF-DD84-4FC9-B925-69A61240EDA7;Trusted_Connection
    =True;MultipleActiveResultSets=true",
    "NorthwindConnection": "Server=.;Database=Northwind;Trusted_Connection
    =True;MultipleActiveResultSets=true"
},
```

4. В файле `Program.cs` импортируйте пространство имен для работы с типами сущностной модели:

```
using Packt.Shared; // метод расширения AddNorthwindContext
```

5. Перед вызовом метода `builder.Build` добавьте операторы для загрузки соответствующей строки подключения, а затем и для регистрации контекста базы данных Northwind:

```
// если используете SQL Server
string sqlServerConnection = builder.Configuration
  .GetConnectionString("NorthwindConnection");
builder.Services.AddNorthwindContext(sqlServerConnection);

// если используете SQLite по умолчанию с ..\Northwind.db
builder.Services.AddNorthwindContext();
```

6. Добавьте в папку Models файл класса и назовите его `HomeIndexViewModel.cs`.

Хотя класс ErrorViewModel, созданный шаблоном проекта MVC, не соответствует этому соглашению, я рекомендую использовать соглашение об именовании {Controller}{Action}ViewModel для классов модели представления.

7. Измените операторы, чтобы определить три свойства для подсчета количества посетителей, получения списков категорий и товаров:

```
using Packt.Shared; // Category, Product

namespace Northwind.Mvc.Models;

public record HomeIndexViewModel
(
  int VisitorCount,
  IList<Category> Categories,
  IList<Product> Products
);
```

8. В файле `HomeController.cs` импортируйте пространство имен `Packt.Shared`:

```
using Packt.Shared; // NorthwindContext
```

9. Добавьте поле для хранения ссылки на экземпляр Northwind и инициализируйте его в конструкторе, как показано ниже:

```
public class HomeController : Controller
{
  private readonly ILogger<HomeController> _logger;
  private readonly NorthwindContext db;

  public HomeController(Ilogger<HomeController> logger,
    NorthwindContext injectedContext)
  {
    _logger = logger;
    db = injectedContext;
  }
...
```

ASP.NET Core будет использовать внедрение параметров конструктора для передачи экземпляра контекста базы данных NorthwindContext, используя строку подключения, которую вы указали в файле Program.cs.

10. Измените операторы в методе действия Index, чтобы создать для него экземпляр модели представления, имитируя счетчик посетителей с помощью класса Random для генерации числа от 1 до 1000 и базы данных Northwind для получения списков категорий и товаров, а затем передать модель в представление, как показано ниже:

```
[ResponseCache(Duration = 10, Location = ResponseCacheLocation.Any)]
public IActionResult Index()
{
  _logger.LogError("This is a serious error (not really!)");
  _logger.LogWarning("This is your first warning!");
  _logger.LogWarning("Second warning!");
  _logger.LogInformation("I am in the Index method of the HomeController.");

  HomeIndexViewModel model = new
   (
    VisitorCount: (new Random()).Next(1, 1001),
    Categories: db.Categories.ToList(),
    Products: db.Products.ToList()
  );
  return View(model); // передача модели представлению
}
```

Не забывайте о соглашении о поиске представления: когда метод View() вызывается в методе действия контроллера, ASP.NET Core MVC ищет в папке Views подпапку с тем же именем, что и у текущего контроллера, то есть Home. Затем ищет файл с тем же именем, что и у текущего действия, то есть Index.cshtml. Он также будет искать представления, соответствующие имени метода действия, в папке Shared и Razor Pages в папке Pages.

Представления

В MVC символ V обозначает представление. Оно несет ответственность за преобразование модели в HTML-код и другие форматы.

Для этого можно применить несколько *обработчиков представления*. Обработчик представления по умолчанию называется *Razor* и использует символ `@`, чтобы обозначить выполнение кода на стороне сервера. Функция Razor Pages, представленная в ASP.NET Core 2.0, использует тот же обработчик представления и поэтому может применять тот же синтаксис Razor.

Изменим представление главной страницы, чтобы отобразить списки категорий и товаров.

1. Разверните папку Views, а затем папку Home.
2. Откройте файл `Index.cshtml` и обратите внимание на код C#, заключенный в блок `@{}`. Этот блок будет выполнен в первую очередь и может использоваться для хранения данных, которые должны быть переданы в общий файл макета, такой как заголовок веб-страницы:

```
@{
  ViewData["Title"] = "Home Page";
}
```

3. Обратите внимание на статический HTML-контент в элементах `<div>`, для стилизации которых используется библиотека Bootstrap.

При необходимости определить собственные стили старайтесь создавать их на основе общей библиотеки, такой как Bootstrap, реализующей принципы адаптивного дизайна.

Как и в случае Razor Pages, существует файл `_ViewStart.cshtml`, который запускается из метода `View()`. Он используется для установки значений по умолчанию, применяемых ко всем представлениям.

Например, он задает для свойства `Layout` всех представлений общий файл макета:

```
@{
  Layout = "_Layout";
}
```

4. В папке Views откройте файл `_ViewImports.cshtml` и обратите внимание, что импортируются некоторые пространства имен, а затем добавляются вспомогательные теги ASP.NET Core:

```
@using Northwind.Mvc
@using Northwind.Mvc.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

5. В папке Shared откройте файл `_Layout.cshtml`.
6. Обратите внимание: заголовок читается из словаря `ViewData`, который был установлен ранее в представлении `Index.cshtml`:

```
<title>@ViewData["Title"] - Northwind.Mvc</title>
```

7. Обратите внимание на ссылки для поддержки Bootstrap и таблиц стилей сайта, где символ ~ обозначает папку wwwroot:

```
<link rel="stylesheet"
  href="~/lib/bootstrap/dist/css/bootstrap.css" />
<link rel="stylesheet" href="~/css/site.css" />
```

8. Обратите внимание на отображение панели навигации в заголовке:

```
<body>
  <header>
    <nav class="navbar ...">
```

9. Обратите внимание на отображение сворачиваемого элемента `<div>`, содержащего частичное представление для входа в систему и гиперссылки, позволяющие пользователям перемещаться между страницами с помощью вспомогательных тегов ASP.NET Core с атрибутами `asp-controller` и `asp-action`:

```
<div class=
  "navbar-collapse collapse d-sm-inline-flex justify-content-between">
  <ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area=""
        asp-controller="Home" asp-action="Index">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark"
        asp-area="" asp-controller="Home"
        asp-action="Privacy">Privacy</a>
    </li>
  </ul>
  <partial name="_LoginPartial" />
</div>
```

Для элементов `<a>` использованы вспомогательные теги с именами `asp-controller` и `asp-action`, чтобы указать имена контроллера и действия, которые должны выполниться при переходе по ссылке. Если хотите перейти к функции в библиотеке классов Razor, например к компоненту `employees`, который вы создали в предыдущей главе, используйте `asp-area`, чтобы указать имя функции.

10. Обратите внимание на отрисовку основной части страницы (body) внутри элемента `<main>`:

```
<div class="container">
  <main role="main" class="pb-3">
```

```
    @RenderBody()
  </main>
</div>
```

Метод `RenderBody` внедряет содержимое определенного представления Razor для страницы, например файла `Index.cshtml`, в общий макет.

11. Обратите внимание на элементы `<script>` в нижней части страницы — таким образом не замедляется отображение страницы, а вы можете добавлять собственные блоки сценариев в необязательный определенный раздел `scripts`, как показано в следующем фрагменте кода:

```
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js">
</script>
<script src="~/js/site.js" asp-append-version="true"></script>
@await RenderSectionAsync("scripts", required: false)
```

Когда вспомогательный тег `asp-append-version` указывается со значением `true` в любом элементе, например `<img>` или `<script>`, вместе с атрибутом `src`, вызывается Image Tag Helper (этот вспомогательный элемент назван неудачно, поскольку влияет не только на изображения!).

Его работа заключается в добавлении строки запроса с именем `v`, которое генерируется из хеша файла, указанного в `src`:

```
<script src="~/js/site.js? v=Kl_dqr9NVtnMdsM2MUg4qthUnWZm5T1fCEimBPWDNgM"></
script>
```

Изменение даже одного байта в файле `site.js` меняет его хеш, и, как следствие, если браузер или CDN кэшируют файл со сценарием, то кэшированная копия уничтожится и будет заменена новой версией.

Добавление собственного функционала на сайт ASP.NET Core MVC

Теперь, проанализировав структуру базового сайта MVC, вы сможете настроить и расширить его. Вы уже зарегистрировали модель EF Core для базы данных Northwind, и потому следующая задача — вывести часть этих данных на главную страницу.

Определение пользовательских стилей

На главной странице будет отображен список из 77 наименований товаров базы данных Northwind. Чтобы эффективно использовать пространство, мы

отобразим список в трех столбцах. Для этого нам необходимо настроить таблицу стилей для сайта.

1. В папке wwwroot\css откройте файл `site.css`.
2. В конце файла добавьте новый стиль, который будет применен к элементу с идентификатором `product-columns`:

```
#product-columns
{
  column-count: 3;
}
```

Настройка категории изображений

База данных Northwind включает в себя таблицу из восьми категорий, однако не содержит изображений, а сайты выглядят лучше, если на них есть яркие картинки.

1. В папке wwwroot создайте папку images.
2. В папку images добавьте восемь файлов изображений: `category1.jpeg`, `category2.jpeg` и т. д. до `category8.jpeg`.

Вы можете скачать изображения из репозитория GitHub для этой книги: https://github.com/markjprice/cs10dotnet6/tree/master/Assets/Categories.

Синтаксис Razor

Прежде чем настраивать вид главной страницы, рассмотрим пример файла Razor, имеющего начальный кодовый блок Razor, который создает экземпляр заказа с ценой и количеством, а затем выводит сведения о заказе на веб-странице:

```
@{
  Order order = new()
  {
    OrderId = 123,
    Product = "Sushi",
    Price = 8.49M,
    Quantity = 3
  };
}
<div>Your order for @order.Quantity of @order.Product has a total cost
of $@order.Price * @order.Quantity</div>
```

Этот Razor-файл приведет к неверному выводу:

```
Your order for 3 of Sushi has a total cost of $8.49 * 3
```

Хотя разметка Razor может включать значение любого отдельного свойства с помощью синтаксиса `@object.property`, выражения необходимо заключать в скобки:

```
<div>Your order for @order.Quantity of @order.Product has a total cost of $@
(order.Price * order.Quantity)</div>
```

Это выражение Razor приведет к корректному выводу:

```
Your order for 3 of Sushi has a total cost of $25.47
```

Определение типизированного представления

Если с помощью директивы `@model` определить ожидаемый представлением тип, то это улучшит работу IntelliSense при написании кода представления.

1. В папке Views\Home откройте файл `Index.cshtml`.
2. В начале файла добавьте оператор, чтобы установить `HomeIndexViewModel` в качестве используемого типа модели:

   ```
   @model HomeIndexViewModel
   ```

   Каждый раз при вводе `Model` в этом представлении ваш редактор кода будет знать подходящий тип и предоставит для него IntelliSense.

   Вводя код в представление, помните следующее:

   - чтобы объявить тип для модели, следует писать `@model` (со строчной буквой `m`);
   - чтобы взаимодействовать с экземпляром модели, следует писать `@Model` (с прописной буквой `M`).

   Продолжим переделывать главную страницу.

3. В исходном блоке кода Razor добавьте оператор, объявляющий переменную `string` для текущего элемента, и под существующим элементом `<div>` добавьте новую разметку, чтобы вывести категории в карусели и товары в виде неупорядоченного списка:

   ```
   @using Packt.Shared
   @model HomeIndexViewModel
   @{
     ViewData["Title"] = "Home Page";
     string currentItem = "";
   }

   <div class="text-center">
     <h1 class="display-4">Welcome</h1>
     <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building
     Web apps with ASP.NET Core</a>.</p>
   ```

```
    <p class="alert alert-primary">@DateTime.Now.ToLongTimeString()</p>
  </div>
  @if (Model is not null)
  {
  <div id="categories" class="carousel slide" data-ride="carousel"
       data-interval="3000" data-keyboard="true">
    <ol class="carousel-indicators">
    @for (int c = 0; c < Model.Categories.Count; c++)
    {
      if (c == 0)
      {
        currentItem = "active";
      }
      else
      {
        currentItem = "";
      }
      <li data-target="#categories" data-slide-to="@c"
          class="@currentItem"></li>
    }
    </ol>
    <div class="carousel-inner">
    @for (int c = 0; c < Model.Categories.Count; c++)
    {
      if (c == 0)
      {
        currentItem = "active";
      }
      else
      {
        currentItem = "";
      }
      <div class="carousel-item @currentItem">
        <img class="d-block w-100" src=
          "~/images/category@(Model.Categories[c].CategoryId).jpeg"
          alt="@Model.Categories[c].CategoryName" />
        <div class="carousel-caption d-none d-md-block">
          <h2>@Model.Categories[c].CategoryName</h2>
          <h3>@Model.Categories[c].Description</h3>
          <p>
            <a class="btn btn-primary"
              href="/category/@Model.Categories[c].CategoryId">View</a>
          </p>
        </div>
      </div>
    }
    </div>
    <a class="carousel-control-prev" href="#categories"
      role="button" data-slide="prev">
      <span class="carousel-control-prev-icon"
        aria-hidden="true"></span>
      <span class="sr-only">Previous</span>
```

```
    </a>
    <a class="carousel-control-next" href="#categories"
      role="button" data-slide="next">
      <span class="carousel-control-next-icon" 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>
      @if (Model is not null)
      {
      <h2>Products</h2>
      <div id="product-columns">
        <ul>
        @foreach (Product p in @Model.Products)
        {
          <li>
            <a asp-controller="Home"
              asp-action="ProductDetail"
              asp-route-id="@p.ProductId">
              @p.ProductName costs
              @(p.UnitPrice is null ? "zero" : p.UnitPrice.Value.ToString("C"))
            </a>
          </li>
        }
        </ul>
      </div>
      }
    </div>
  </div>
```

При анализе этой разметки Razor обратите внимание на следующие моменты:

- статические элементы HTML, такие как `<ul>` и `<li>`, можно легко смешивать с кодом на языке C#, чтобы выводить карусель категорий и список наименований товаров;
- к элементу `<div>` с атрибутом `id` в `product-columns` применен пользовательский стиль, определенный нами ранее, поэтому все содержимое данного элемента будет выведено в три столбца;
- в элементе `<img>` для каждой категории используются круглые скобки вокруг выражения Razor с целью гарантировать, что компилятор не рассматривает расширение `.jpeg` как часть выражения:

```
"~/images/category@(Model.Categories[c].CategoryID).jpeg"
```

- URL-пути в элементах `<a>` для ссылок на товары создаются с помощью вспомогательных тегов. Эти гиперссылки будут обрабатываться контроллером `HomeController` и его методом действия `ProductDetail`. Данный метод еще не существует, но мы его добавим в текущей главе позже. Идентификатор товара передается в виде сегмента маршрута `id`, как показано в следующем URL для Ipoh Coffee: https://localhost:5001/Home/ProductDetail/43.

Проверка измененной главной страницы

Посмотрим, как в итоге выглядит наша главная страница.

1. Запустите проект сайта `Northwind.Mvc`.
2. Обратите внимание: на главной странице имеется прокручивающаяся карусель, которая отображает категории, случайное количество посетителей и список товаров в трех столбцах (рис. 15.4).

Рис. 15.4. Обновленная главная страница сайта Northwind MVC

В настоящий момент нажатие любой из категорий или ссылок на товары приводит к ошибкам `404 Not Found`, поэтому мы изучим, как можно реализовать страницы, которые используют переданные параметры для отображения сведений о товаре или категории.

3. Закройте браузер Google Chrome и завершите работу веб-сервера.

Передача параметров с помощью значения маршрута

Один из способов передать простой параметр — использовать сегмент `id`, определенный в маршруте по умолчанию.

1. В классе `HomeController` добавьте метод действия `ProductDetail`:

```
public IActionResult ProductDetail(int? id)
{
  if (!id.HasValue)
  {
    return BadRequest("You must pass a product ID in the route, for example,
    /Home/ProductDetail/21");
  }

  Product? model = db.Products
    .SingleOrDefault(p => p.ProductId == id);

  if (model == null)
  {
    return NotFound($"ProductId {id} not found.");
  }

  return View(model); // передаем модель для просмотра и возвращаем результат
}
```

   Обратите внимание на следующие моменты:

   - в методе `ProductDetail` используется функция ASP.NET Core, называемая *привязкой модели*, для автоматического сопоставления идентификатора, переданного в маршруте, с идентификатором `id` в методе;
   - в методе проверяется, есть ли у идентификатора значение, и если да, то мы вызываем метод `BadRequest`, чтобы вернуть код состояния `400` и сообщение с правильным форматом URL;
   - в противном случае можно подключиться к базе данных и попытаться запросить товар по значению `id` (идентификатору);
   - если товар найден, то передается в представление; в противном случае мы вызываем метод `NotFound`, чтобы вернуть код состояния `404` и сообщение, объясняющее, что продукт с этим ID не был найден в базе данных.

2. Добавьте в папку Views/Home новый файл `ProductDetail.cshtml`.
3. Отредактируйте его содержимое:

```
@model Packt.Shared.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>
```

4. Запустите проект `Northwind.Mvc`.
5. Когда появится главная страница со списком товаров, выберите один из них, например товар под номером 2 — Chang.
6. Обратите внимание на URL в адресной строке браузера, заголовок страницы, отображаемый на вкладке браузера, и страницу сведений о товаре (рис. 15.5).

Рис. 15.5. Страница сведений о товаре Chang

7. Переключитесь на панель Developer tools (Инструменты разработчика).
8. Измените URL в адресной строке браузера, чтобы запросить идентификатор несуществующего товара, например, `99`, и обратите внимание на код состояния `404 Not Found` и сообщение об ошибке.

Тонкости привязки моделей

Привязка моделей (model binders) — эффективный инструмент и по умолчанию многое может выполнить автоматически. После того как маршрут по умолчанию идентифицирует класс контроллера для создания экземпляра и метод действия для вызова при наличии у данного метода параметров для них надо установить значения.

Для этого привязка моделей проанализирует значения переданных в HTTP-запросе параметров всех описанных ниже типов:

- *параметр маршрута*, такой как `id`, который мы создали чуть выше, как показано в следующем URL-пути: /Home/ProductDetail/2;
- *параметр строки запроса*, как показано в следующем URL-пути: /Home/ProductDetail?Id=2;
- параметр формы:

```
<form action="post" action="/Home/ProductDetail">
  <input type="text" name="id" value="2" />
  <input type="submit" />
</form>
```

Привязка моделей может установить значение практически любого типа:

- простых типов, таких как `int`, `string`, `DateTime` и `bool`;
- сложных типов, определяемых как `class`, `record` или `struct`;
- типов коллекций, таких как массивы и списки.

Создадим искусственный пример для демонстрации того, чего можно достичь с помощью привязки моделей по умолчанию.

1. В папку Models добавьте новый файл `Thing.cs`.
2. Отредактируйте его содержимое, чтобы определить класс с двумя свойствами: для нулевого целого числа `ID` и строковой переменной `color`:

```
namespace Northwind.Mvc.Models;

public class Thing
{
```

```
  public int? Id { get; set; }
  public string? Color { get; set; }
}
```

3. В `HomeController` добавьте два новых метода действия: для отображения страницы с формой и объекта с параметром, используя новый тип модели:

```
public IActionResult ModelBinding()
{
  return View(); // страница с формой
}

public IActionResult ModelBinding(Thing thing)
{
  return View(thing); // привязанный к модели объект
}
```

4. В папку Views\Home добавьте новый файл `ModelBinding.cshtml`.
5. Отредактируйте его содержимое:

```
@model Thing
@{
  ViewData["Title"] = "Model Binding Demo";
}
<h1>@ViewData["Title"]</h1>
<div>
  Enter values for your thing in the following form:
</div>
<form method="POST" action="/home/modelbinding?id=3">
  <input name="color" value="Red" />
  <input type="submit" />
</form>
@if (Model != null)
{
<h2>Submitted Thing</h2>
<hr />
<div>
  <dl class="dl-horizontal">
    <dt>Model.Id</dt>
    <dd>@Model.Id</dd>
    <dt>Model.Color</dt>
    <dd>@Model.Color</dd>
  </dl>
</div>
}
```

6. В папке Views/Home откройте файл `Index.cshtml` и в первом элементе `<div>` добавьте абзац текста со ссылкой на страницу привязки модели:

```
<p><a asp-action="ModelBinding" asp-controller="Home">Binding</a></p>
```

7. Запустите сайт.
8. На главной странице щелкните на Binding (Привязка).
9. Обратите внимание на необработанное исключение о неоднозначном совпадении (рис. 15.6).

Рис. 15.6. Необработанное исключение несоответствия метода неоднозначного действия

10. Закройте браузер Google Chrome и завершите работу веб-сервера.

Методы устранения неоднозначности действий

Компилятор C# может различать два метода, отметив, что сигнатуры различны, однако с точки зрения маршрутизации HTTP оба метода потенциально совпадают. Нам необходим специфический для HTTP способ устранения неоднозначности методов действия.

Мы могли бы сделать так, создав разные имена для действий или указав, что один метод должен использоваться для определенной HTTP-команды, такой как `GET`, `POST` или `DELETE`. Вот как мы решим эту проблему.

1. В классе `HomeController` добавьте ко второму методу действия `ModelBinding`, чтобы указать, что его следует использовать для обработки HTTP-запросов `POST`, то есть при отправке формы, как показано ниже (выделено жирным шрифтом):

```
[HttpPost]
Public IactionResult ModelBinding(Thing thing)
```

Другой метод действия ModelBinding будет неявно использоваться для всех других типов HTTP-запросов, таких как GET, PUT, DELETE и т. д.

2. Запустите сайт.
3. На главной странице щелкните на Binding (Привязка).

4. Нажмите кнопку Submit (Отправить) и обратите внимание, что значение свойства `ID` задается из параметра строки запроса, а свойства `color` — из параметра формы (рис. 15.7).

Рис. 15.7. Демонстрационная страница Model Binding

5. Закройте браузер Google Chrome и завершите работу веб-сервера.

Передача параметра маршрута

Теперь мы установим свойство с помощью параметра маршрута.

1. Измените действие для формы, чтобы передать значение `2` в качестве параметра маршрута (выделено жирным шрифтом):

```
<form method="POST" action="/home/modelbinding/2?id=3">
```

2. Запустите сайт.
3. На главной странице щелкните на Binding (Привязка).
4. Нажмите кнопку Submit (Отправить) и обратите внимание, что значение свойства `ID` задается из параметра маршрута, а свойства `color` — из параметра формы.
5. Закройте браузер Google Chrome и завершите работу веб-сервера.

Передача параметра формы

Теперь установим свойство с помощью параметра формы.

1. Измените действие для формы, чтобы передать значение `1` в качестве параметра формы, как показано ниже (выделено жирным шрифтом):

```
<form method="POST" action="/home/modelbinding/2?id=3">
  <input name="id" value="1" />
  <input name="color" value="Red" />
  <input type="submit" />
</form>
```

2. Запустите сайт.
3. На главной странице щелкните на Binding (Привязка).
4. Нажмите кнопку Submit (Отправить) и обратите внимание, что значения свойств `ID` и `color` задаются в параметрах формы.

При наличии нескольких параметров с одним и тем же именем помните, что параметры формы имеют самый высокий приоритет, а параметры строки запроса — самый низкий приоритет при автоматической привязке моделей.

Проверка модели

Процесс привязки модели может вызвать ошибки, например, преобразования типов данных или ошибки проверки, если модель была дополнена правилами проверки. Привязанные данные, а также все ошибки привязки или проверки сохраняются в `ControllerBase.ModelState`.

Рассмотрим, что мы можем сделать с состоянием модели, применив некоторые правила проверки к привязанной модели и затем показав сообщения о недопустимых данных в представлении.

1. В папке Models откройте файл `Thing.cs`.
2. Импортируйте пространство имен `System.ComponentModel.DataAnnotations`.
3. Дополните свойство `ID` атрибутом проверки, чтобы ограничить диапазон разрешенных чисел от 1 до 10 и убедиться в предоставлении цвета пользователем, и добавьте новое свойство `Email` с регулярным выражением для проверки, как показано ниже:

```
public class Thing
{
  [Range(1, 10)]
  public int? Id { get; set; }
```

```
  [Required]
  public string? Color { get; set; }

  [EmailAddress]
  public string? Email { get; set; }
}
```

4. В папке Models добавьте новый файл `HomeModelBindingViewModel.cs`.
5. Отредактируйте его содержимое, чтобы определить запись со свойствами для хранения связанной модели, флаг, указывающий на наличие ошибок, и последовательность сообщений об ошибках:

```
namespace Northwind.Mvc.Models;

public record HomeModelBindingViewModel
(
  Thing Thing,
  bool HasErrors,
  IEnumerable<string> ValidationErrors
);
```

6. В классе `HomeController` в методе `ModelBinding`, который обрабатывает HTTP-запрос `POST`, закомментируйте предыдущий оператор, который передал объект представлению, и вместо этого добавьте операторы для создания экземпляра модели представления. Проверьте модель и сохраните массив сообщений об ошибках, а затем передайте модель представлению, как показано ниже:

```
[HttpPost]
public IActionResult ModelBinding(Thing thing)
{
  HomeModelBindingViewModel model = new(
    thing,
    !ModelState.IsValid,
    ModelState.Values
      .SelectMany(state => state.Errors)
      .Select(error => error.ErrorMessage)
  );
  return View(model);
}
```

7. В папке Views\Home откройте файл `ModelBinding.cshtml`.
8. Измените объявление типа модели, чтобы использовать класс модели представления:

```
@model Northwind.Mvc.Models.HomeModelBindingViewModel
```

9. Добавьте элемент `<div>` для отображения всех ошибок проверки модели и измените вывод свойств объекта, поскольку модель представления изменилась, как показано ниже (выделено жирным шрифтом):

```
<form method="POST" action="/home/modelbinding/2?id=3">
  <input name="id" value="1" />
  <input name="color" value="Red" />
  <input name="email" value="test@example.com" />
  <input type="submit" />
</form>
@if (Model != null)
{
  <h2>Submitted Thing</h2>
  <hr />
  <div>
    <dl class="dl-horizontal">
      <dt>Model.Thing.Id</dt>
      <dd>@Model.Thing.Id</dd>
      <dt>Model.Thing.Color</dt>
      <dd>@Model.Thing.Color</dd>
      <dt>Model.Thing.Email</dt>
      <dd>@Model.Thing.Email</dd>
    </dl>
  </div>
  @if (Model.HasErrors)
  {
    <div>
      @foreach(string errorMessage in Model.ValidationErrors)
      {
        <div class="alert alert-danger" role="alert">@errorMessage</div>
      }
    </div>
  }
}
```

10. Запустите сайт.
11. На главной странице щелкните на Binding (Привязка).
12. Нажмите кнопку Submit (Отправить) и обратите внимание, что значения `1`, `Red` и `test@example.com` — допустимые.
13. Введите `13` в качестве значения `ID` (идентификатора), очистите цветное текстовое поле, удалите символ @ из адреса электронной почты, нажмите кнопку Submit (Отправить) и обратите внимание на сообщения об ошибках (рис. 15.8).
14. Закройте браузер Google Chrome и завершите работу веб-сервера.

О том, какое регулярное выражение использует компания Microsoft для реализации атрибута проверки EmailAddress, вы можете узнать по следующей ссылке: https://github.com/microsoft/referencesource/blob/5697-c29004a34d80acdaf5742d7e699022c64ecd/System.ComponentModel.DataAnnotations/DataAnnotations/EmailAddressAttribute.cs#L54.

Рис. 15.8. Демонстративная страница Model Binding с проверкой полей

Методы вспомогательного класса для представления

При создании представления для ASP.NET Core MVC вы можете использовать объект класса `Html` и его методы, чтобы создать разметку.

Ниже представлено описание некоторых полезных методов.

- `ActionLink` используется для создания якорного элемента `<a>`, содержащего путь URL к указанному контроллеру и действию. Например, `Html.ActionLink(linkText: "Binding", actionName: "ModelBinding", controllerName: "Home")` создаст `<a href="/ home/modelbinding">Binding</a>`. Вы можете достичь того же результата, используя вспомогательный тег привязки: `<a asp-action="ModelBinding" asp-controller="Home">Binding</a>`.
- `AntiForgeryToken` применяется внутри элемента `<form>`, чтобы вставить элемент `<hidden>`, содержащий маркер защиты от подделки, который будет проверен при отправке формы.
- `Display` и `DisplayFor` служат для создания HTML-разметки для выражения, относящегося к текущей модели, с помощью шаблона отображения. Существуют встроенные шаблоны отображения для типов .NET, а пользовательские

шаблоны можно создавать в папке DisplayTemplates. Имя папки чувствительно к регистру в определенных файловых системах.

- `DisplayForModel` используется для создания HTML-разметки для всей модели вместо одного выражения.
- `Editor` и `EditorFor` служат для создания HTML-разметки для выражения, относящегося к текущей модели, с помощью шаблона редактора. Существуют встроенные шаблоны редакторов для типов .NET, которые задействуют элементы `<label>` и `<input>`, а пользовательские шаблоны можно создавать в папке EditorTemplates. Имя папки чувствительно к регистру в определенных файловых системах.
- `EditorForModel` используется для создания HTML-разметки для всей модели вместо одного выражения.
- `Encode` применяется с целью безопасного кодирования объекта или строки в HTML. Например, строковое значение `"<script>"` будет закодировано в виде `"<script>"`. Обычно в этом нет необходимости, поскольку символ Razor `@` по умолчанию кодирует строковые значения.
- `Raw` используется для отображения строкового значения *без* кодирования в HTML.
- `PartialAsync` и `RenderPartialAsync` применяются для создания HTML-разметки для частичного представления. При желании вы можете передать модель и данные представления (`ViewData`).

Рассмотрим пример.

1. В папке Views/Home откройте файл `ModelBinding.cshtml`.
2. Измените отображение свойства `Email`, чтобы использовать метод `DisplayFor`, как показано ниже:

```
<dd>@Html.DisplayFor(model => model.Thing.Email)</dd>
```

3. Запустите сайт.
4. Щелкните на Binding (Привязка).
5. Нажмите кнопку Submit (Отправить).
6. Обратите внимание, что адрес электронной почты представляет собой активную гиперссылку, а не просто текст.
7. Закройте браузер Google Chrome и завершите работу веб-сервера.
8. В папке Models в файле `Thing.cs` закомментируйте атрибут `[EmailAddress]` над свойством `Email`.
9. Запустите сайт.
10. Щелкните на Binding (Привязка).

11. Нажмите кнопку Submit (Отправить).
12. Обратите внимание, что адрес электронной почты представляет собой просто текст.
13. Закройте браузер Google Chrome и завершите работу веб-сервера.
14. В папке Models в файле `Thing.cs` раскомментируйте атрибут `[EmailAddress]`.

Именно сочетание добавления к свойству `Email` атрибута проверки `[EmailAddress]` и отображения его с помощью метода `DisplayFor` сообщает ASP.NET Core о необходимости рассматривать значение как адрес электронной почты и, следовательно, отображать его в виде гиперссылки.

Отправка запросов в базу данных и использование шаблонов отображения

Создадим новый метод действия, которому будет передаваться параметр из строки запроса, и с его помощью запросим у базы данных Northwind информацию о товарах, стоимость которых превышает указанную цену.

В предыдущих примерах мы определили модель представления, содержащую свойства для каждого значения, которое необходимо отобразить в представлении. В этом примере будут два значения: список товаров и цена, которую ввел посетитель. Чтобы не определять класс или запись для модели представления, мы передадим список товаров в качестве модели и сохраним максимальную цену в коллекции `ViewData`.

Реализуем эту функцию.

1. В классе `HomeController` импортируйте пространство имен `Microsoft.EntityFrameworkCore`. Это нужно для того, чтобы мы могли использовать метод расширения `Include` и с его помощью включать связанные объекты, как описано в главе 10.
2. Добавьте новый метод действия:

```
public IactionResult ProductsThatCostMoreThan(decimal? Price)
{
  if (!price.HasValue)
  {
    return BadRequest("You must pass a product price in the query
      string, for example, /Home/ProductsThatCostMoreThan?price=50");
  }

  Ienumerable<Product> model = db.Products
    .Include(p => p.Category)
```

```
    .Include(p => p.Supplier)
    .Where(p => p.UnitPrice > price);

  if (!model.Any())
  {
    return NotFound(
      $"No products cost more than {price:C}.");
  }

  ViewData["MaxPrice"] = price.Value.ToString("C");
  return View(model); // передача модели представлению
}
```

3. В папке Views/Home создайте файл `ProductsThatCostMoreThan.cshtml`.
4. Отредактируйте его содержимое:

```
@using Packt.Shared
@model IEnumerable<Product>
@{
  string title =
    "Products That Cost More Than " + ViewData["MaxPrice"];
  ViewData["Title"] = title;
} <
h2>@title</h2>
@if (Model is null)
{
  <div>No products found.</div>
}
else
{
  <table class="table">
    <thead>
      <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>
    </thead>
    <tbody>
    @foreach (Product p in Model)
    {
      <tr>
        <td>
          @Html.DisplayFor(modelItem => p.Category.CategoryName)
        </td>
        <td>
          @Html.DisplayFor(modelItem => p.Supplier.CompanyName)
        </td>
```

```
        <td>
          @Html.DisplayFor(modelItem => p.ProductName)
        </td>
        <td>
          @Html.DisplayFor(modelItem => p.UnitPrice)
        </td>
        <td>
          @Html.DisplayFor(modelItem => p.UnitsInStock)
        </td>
      </tr>
    }
    <tbody>
  </table>
}
```

5. В папке Views/Home откройте файл `Index.cshtml`.
6. Добавьте следующий элемент формы после счетчика посетителей и перед заголовком `Products` и списком товаров. Таким образом пользователь получит форму для ввода цены. Затем он может нажать кнопку Submit (Отправить), чтобы вызвать метод действия, который выведет только товары, стоимость которых превышает введенную цену:

```
<h3>Query products by price</h3>
<form asp-action="ProductsThatCostMoreThan" method="GET">
  <input name="price" placeholder="Enter a product price" />
  <input type="submit" />
</form>
```

7. Запустите сайт.
8. На главной странице введите в форму цену товара, например `50`, а затем нажмите Submit (Отправить).
9. Обратите внимание на таблицу товаров, стоимость которых превышает введенную вами цену (рис. 15.9).
10. Закройте браузер Google Chrome и завершите работу веб-сервера.

Рис. 15.9. Отфильтрованный список товаров стоимостью более 50 фунтов стерлингов

Улучшение масштабируемости с помощью асинхронных задач

При создании настольного или мобильного приложения можно повысить скорость отклика, используя несколько задач (и потоков, на основе которых они работают), поскольку, пока один поток занят задачей, другой может обрабатывать взаимодействие с пользователем.

Задачи и их потоки также могут быть полезны на стороне сервера, особенно на сайтах, работающих с файлами или запрашивающих данные из хранилища или веб-сервиса, ведь ответ на такой запрос может потребовать какого-то времени. Однако они небезопасны для сложных вычислений, производимых процессором, поэтому выполняйте такие задачи синхронно, в обычном режиме.

Когда HTTP-запрос поступает на веб-сервер, поток из его пула выделяется для обработки запроса. Но если этот поток должен ждать ресурса, то блокируется от обработки всех входящих запросов. Если сайт получает больше одновременных запросов, чем количество потоков в его пуле, то на некоторые из этих запросов будет отправлено сообщение об ошибке времени ожидания сервера `503 Service Unavailable` (Сервис недоступен).

Блокированные потоки не выполняют полезной работы. Они *могут* обработать один запрос, но только если мы внедрим асинхронный код на наших сайтах.

Всякий раз, ожидая нужный ему ресурс, поток может вернуться в пул потоков и обработать другой входящий запрос, что улучшает масштабируемость сайта, то есть увеличивает количество одновременных запросов, которые он может обработать.

Почему бы просто не увеличить пул потоков? В современных операционных системах каждый поток в пуле имеет стек емкостью 1 Мбайт. Асинхронный метод использует меньший объем памяти. Благодаря этому также устраняется необходимость создания новых потоков в пуле, что требует времени. Скорость, с которой новые потоки добавляются в пул, обычно составляет один раз в каждые 2 с, что, по сравнению с переключением между асинхронными потоками, очень долго.

Сделайте методы действий вашего контроллера асинхронными.

Превращение методов действия контроллера в асинхронные

Существующий метод действия легко сделать асинхронным.

1. Измените метод действия `Index` на асинхронный, чтобы он возвращал задание и ожидал вызовы асинхронных методов для получения категорий и товаров, как показано ниже (выделено жирным шрифтом):

```
public async Task<IActionResult> Index()
{
  HomeIndexViewModel model = new
   (
    VisitorCount: (new Random()).Next(1, 1001),
    Categories: await db.Categories.ToListAsync(),
    Products: await db.Products.ToListAsync()
  );
  return View(model); // передача модели представлению
}
```

2. Аналогичным образом измените метод действия `ProductDetail`, как показано ниже (выделено жирным шрифтом):

```
public async Task<IActionResult> ProductDetail(int? Id)
{
  if (!id.HasValue)
  {
    return BadRequest("You must pass a product ID in the route,
    for example, /Home/ProductDetail/21");
  }

  Product? Model = await db.Products
    .SingleOrDefaultAsync(p => p.ProductId == id);

  if (model == null)
  {
    return NotFound($"ProductId {id} not found.");
  }
  return View(model); // передача модели представлению
                      // и возвращение результата
}
```

3. Запустите сайт и обратите внимание, что функциональные возможности сайта такие же. Но поверьте — масштабироваться он теперь будет лучше.
4. Закройте браузер Google Chrome и завершите работу веб-сервера.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 15.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Зачем нужны файлы со специальными именами `_ViewStart` и `_ViewImports`, созданные в папке Views?

2. Как называются три сегмента, определенные по умолчанию в ASP. NET Core MVC, и какие из них необязательны?
3. Что делает связыватель моделей по умолчанию и какие типы данных он может обрабатывать?
4. Как вывести содержимое текущего представления в общем файле макета, таком как `_Layout.cshtml`?
5. Как в общем файле макета, таком как `_Layout.cshtml`, вывести раздел, для которого содержимое может быть предоставлено текущим представлением, и как это представление выдает содержимое для данного раздела?
6. Какие пути ищутся для представления по соглашению при вызове метода `View` внутри метода действия контроллера?
7. Как выдать браузеру пользователя инструкцию кэшировать ответ в течение 24 ч?
8. Почему вы можете включить Razor Pages, даже если сами их не создаете?
9. Как ASP.NET Core MVC идентифицирует классы, которые могут действовать как контроллеры?
10. Каким образом ASP.NET Core MVC облегчает тестирование сайта?

Упражнение 15.2. Реализация MVC для страницы, содержащей сведения о категориях

В проекте `Northwind.Mvc` есть главная страница, на которой отображаются категории, но при нажатии кнопки View (Вид) сайт возвращает ошибку `404 Not Found`, например, для следующего URL: https://localhost:5001/category/1.

Дополните проект `Northwind.Mvc`, добавив возможность отображения страницы, содержащей подробные сведения о категории.

Упражнение 15.3. Улучшение масштабируемости за счет понимания и реализации асинхронных методов действий

Несколько лет назад Стивен Клири написал отличную статью для MSDN Magazine, объясняющую преимущества масштабируемости реализации асинхронных методов действия для ASP.NET. Те же принципы применимы к ASP.NET Core в еще большей степени, поскольку в отличие от устаревшей ASP.NET, как описано в статье, ASP.NET Core поддерживает асинхронные фильтры и другие компоненты.

Прочитайте статью: https://docs.microsoft.com/ru-ru/archive/msdn-magazine/2014/october/async-programming-introduction-to-async-await-on-asp-net.

Упражнение 15.4. Практика модульного тестирования контроллеров MVC

С помощью контроллеров функционирует бизнес-логика вашего сайта, поэтому важно проверять правильность ее работы, используя модульные тесты, как вы узнали в главе 4.

Напишите несколько модульных тестов для класса `HomeController`.

Подробнее о том, как проводить модульное тестирование контроллеров, вы можете прочитать по следующей ссылке: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/controllers/testing.

Упражнение 15.5. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-15---building-websites-using-the-model-view-controller-pattern, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы научились создавать большие и сложные сайты таким образом, чтобы можно было легко их тестировать путем регистрации и внедрения сервисов зависимостей, таких как контексты баз данных и регистраторы, и упрощать управление ими с помощью команд разработчиков, использующих ASP.NET Core MVC. Вы изучили конфигурацию запуска, аутентификацию, маршруты, модели, представления и контроллеры.

В следующей главе вы изучите способы создания и применения сервисов, использующих HTTP в качестве коммуникационного уровня, также известных как веб-сервисы.

16 Разработка и использование веб-сервисов

Эта глава посвящена тому, как разрабатывать веб-сервисы (они же HTTP- или REST-сервисы) с помощью Web API в ASP.NET Core и как использовать их с помощью HTTP-клиентов, которые могут быть .NET-приложениями любого другого типа, включая сайт, мобильное или настольное приложение.

В этой главе потребуются знания и навыки, которые вы получили в главах 10, 13, 14 и 15, посвященных практическому применению C# и .NET и созданию сайтов с помощью ASP.NET Core.

В этой главе:

- разработка веб-сервисов с помощью Web API в ASP.NET Core;
- документирование и тестирование веб-сервисов;
- обращение к веб-сервисам с помощью HTTP-клиентов;
- реализация расширенных функций для веб-сервисов;
- разработка веб-сервисов с помощью минимальных API.

Разработка веб-сервисов с помощью Web API в ASP.NET Core

Прежде чем мы создадим современный веб-сервис, нам необходимо обсудить некоторые основы, чтобы задать контекст для этой главы.

Аббревиатуры, типичные для веб-сервисов

Хотя изначально протокол HTTP проектировался в целях передачи запросов и ответов, содержащих HTML и других ресурсы для просмотра человеком, с его помощью также можно создавать сервисы.

Рой Филдинг, описывая в своей докторской диссертации архитектурный стиль *«передача состояния представления»* (Representational State Transfer, REST), объясняет, что стандарт HTTP хорошо подходит для построения сервисов, так как определяет следующие понятия:

- URL для уникальной идентификации ресурсов, например https://localhost:5001/api/products/23;
- методы для выполнения типовых задач, такие как `GET`, `POST`, `PUT` и `DELETE`;
- возможность согласовывать типы данных содержимого, которыми обмениваются в запросах и ответах, такие как XML и JSON. Согласование содержимого происходит, когда клиент указывает заголовок запроса, например, `Accept: application/xml,*/*;q=0.8`. Формат ответа по умолчанию, используемый Web API в ASP.NET Core, — JSON; это значит, одним из заголовков ответа будет `Content-Type: application/json; charset=utf-8`.

Веб-сервисы используют стандарт связи HTTP, вследствие чего их иногда называют HTTP-сервисами или сервисами RESTful. HTTP-сервисы или RESTful — главная тема данной главы.

Веб-сервисы также могут работать по *SOAP-протоколу* (Simple Object Access Protocol — простой протокол доступа к объектам), реализующему некоторые *стандарты WS-**. Эти стандарты позволяют клиентам и сервисам, реализованным на различных системах, взаимодействовать друг с другом. Стандарты WS-* были первоначально определены компанией IBM при участии других компаний, таких как Microsoft.

Windows Communication Foundation (WCF)

.NET Framework 3.0 и более поздние версии включают технологию *удаленного вызова процедур (remote procedure call, RPC)* под названием *Windows Communication Foundation (WCF)*. Технологии RPC позволяют коду в одной системе выполнять код в другой системе с передачей данных по сети.

WCF позволяет разработчикам легко создавать сервисы, включая сервисы SOAP, реализующие стандарты WS-*. Внедренная позднее поддержка сервисов Web/HTTP/REST хотя и была востребованной, но сама технология оказалась чересчур громоздкой.

Если у вас есть существующие сервисы WCF и вы хотели бы перенести их в современную .NET, то обратите внимание на проект с открытым исходным кодом, *релиз* (General Availability, GA) которого состоялся в феврале 2021 года. Вы можете прочитать о нем по ссылке https://corewcf.github.io/blog/2021/02/19/corewcf-ga-release.

Альтернатива WCF

Альтернатива WCF, рекомендуемая компанией Microsoft, — *gRPC*. Это современный кросс-платформенный RPC-фреймворк с открытым исходным кодом, созданный компанией Google (неофициально об этом сообщает буква g в аббревиатуре gRPC). Вы узнаете больше о gRPC в главе 18.

HTTP-запросы и ответы для Web API

В спецификации HTTP определены стандартные типы запросов и стандартные коды ответов. Большинство из них можно использовать для реализации сервисов Web API.

Наиболее распространенным типом запроса является `GET`, используемый для извлечения ресурса по уникальному пути, с дополнительными параметрами, такими как допустимый тип данных, заданными в качестве заголовка запроса, как показано в следующем примере:

```
GET /path/to/resource
Accept: application/json
```

Распространенные ответы включают успешное выполнение операции и несколько ошибок (табл. 16.1).

Таблица 16.1. Варианты ответов для запроса GET

Код состояния	Описание
`200 OK`	Путь корректный, ресурс был успешно найден, сериализован в приемлемый тип данных, а затем возвращен в теле ответа. В заголовках ответа указываются `Content-Type` (тип контента), `Content-Length` (размер контента) и `Content-Encoding` (кодировка контента), например GZIP
`301 Moved Permanently`	Со временем веб-сервис может изменить свою модель ресурсов, включая путь, используемый для идентификации существующего ресурса. Веб-сервис может сообщать новый путь, вернув соответствующий код состояния и заголовок ответа со значением `Location`, содержащим новый путь
`302 Found`	По аналогии с `301`
`304 Not Modified`	Если запрос включал заголовок `If-Modified-Since`, то веб-сервис может ответить этим кодом состояния. Тело ответа пустое, так как клиент должен использовать собственную кэшированную копию ресурса
`400 Bad Request`	Запрос был некорректным, например, использовался путь к ресурсу с целочисленным идентификатором, в котором отсутствует значение идентификатора

Продолжение ⇨

Таблица 16.1 *(продолжение)*

Код состояния	Описание
`401 Unauthorized`	Запрос допустим, ресурс найден, но клиент не предоставил учетные данные или не имеет доступа к этому ресурсу. Повторная аутентификация может разрешить доступ, например, путем добавления или изменения заголовка запроса `Authorization` (авторизация)
`403 Forbidden`	Запрос допустим, ресурс найден, но клиент не авторизован для доступа к этому ресурсу. Повторная аутентификация не решает проблему
`404 Not Found`	Запрос допустим, но ресурс не найден. Он может быть найден, если запрос будет выполнен позже. Чтобы указать, что ресурс никогда не будет найден, верните значение `410 Gone`
`406 Not Acceptable`	Возникает, если запрос содержит заголовок `Accept`, в котором перечислены типы данных, и они не поддерживаются веб-сервисом. Например, если клиент запрашивает данные в формате JSON, а веб-сервис возвращает только XML
`451 Unavailable for Legal Reasons`	Сайт, расположенный в США, может возвращать такой код в ответ на запросы из Европы, чтобы избежать необходимости соблюдать Общий регламент ЕС по защите данных (General Data Protection Regulation, GDPR). Номер был выбран как отсылка к роману «451 градус по Фаренгейту», в котором книги запрещаются и сжигаются
`500 Server Error`	Запрос допустим, но на стороне сервера во время его обработки произошел сбой. Повторная попытка позже может быть успешной
`503 Service Unavailable`	Веб-сервис занят и не может обработать запрос. Повторная попытка позже может быть успешной

Другие распространенные типы HTTP-запросов включают `POST`, `PUT`, `PATCH` или `DELETE`, которые создают, изменяют или удаляют ресурсы.

Чтобы создать новый ресурс, вы можете выполнить запрос `POST` с телом, содержащим новый ресурс:

```
POST /path/to/resource
Content-Length: 123
Content-Type: application/json
```

Чтобы создать новый ресурс или обновить существующий, вы можете выполнить запрос `PUT` с телом, содержащим совершенно новую версию существующего ресурса, и, если ресурс не существует, он будет создан, а если существует, то будет заменен (иногда это называют операцией обновления `upsert`):

```
PUT /path/to/resource
Content-Length: 123
Content-Type: application/json
```

Для более эффективного обновления существующего ресурса можно выполнить запрос PATCH с телом, содержащим объект только с теми свойствами, которые необходимо изменить:

```
PATCH /path/to/resource
Content-Length: 123
Content-Type: application/json
```

Чтобы удалить существующий ресурс, можно выполнить запрос DELETE:

```
DELETE /path/to/resource
```

Помимо ответов, показанных в табл. 16.1 для запроса GET, все типы запросов, которые создают, изменяют или удаляют ресурс, могут получить дополнительные распространенные ответы (табл. 16.2).

Таблица 16.2. Дополнительные распространенные ответы

Код состояния	Описание
201 Created	Новый ресурс создан успешно, заголовок ответа Location содержит путь к нему, а тело ответа содержит только что созданный ресурс. Немедленный запрос ресурса GET должен вернуть код 200
202 Accepted	Новый ресурс не может быть создан немедленно, поэтому запрос ставится в очередь для последующей обработки, и немедленный запрос ресурса GET может вернуть код 404. Тело запроса может содержать ресурс, указывающий на какую-либо форму проверки состояния или оценку того, когда ресурс станет доступным
204 No Content	Обычно используется в ответ на запрос DELETE, поскольку возвращать ресурс в теле после его удаления обычно не имеет смысла! Иногда используется в ответ на запросы POST, PUT или PATCH, если клиенту не нужно подтверждать, что запрос был обработан правильно
405 Method Not Allowed	Возвращается, если в запросе использован метод, который не поддерживается. Например, веб-сервис, предназначенный только для чтения, может явно запрещать операции PUT, DELETE и т. д.
415 Unsupported Media Type	Возвращается, если ресурс в теле запроса использует тип данных, который веб-сервис не может обработать. Например, если тело запроса содержит ресурс в формате XML, но веб-сервис может обрабатывать только данные в формате JSON

Разработка проекта Web API в ASP.NET Core

Создадим веб-сервис, предоставляющий способ работы с данными в базе данных Northwind, с помощью ASP.NET Core, чтобы данные могли использоваться любым клиентским приложением на любой платформе, которая может отправлять HTTP-запросы и получать HTTP-ответы.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: ASP.NET Core Web API/`webapi`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.WebApi`;
   4) другие параметры Visual Studio: Authentication Type (Тип аутентификации) — None (Нет), флажок Configure for HTTPS (Настроить для HTTPS) установлен, флажок Enable Docker (Включить Docker) снят, флажок Enable OpenAPI support (Включить поддержку OpenAPI) установлен.
2. В программе Visual Studio Code выберите `Northwind.WebApi` в качестве активного проекта OmniSharp.
3. Соберите проект `Northwind.WebApi`.
4. В папке Controllers откройте и проанализируйте файл `WeatherForecastController.cs`:

```
using Microsoft.AspNetCore.Mvc;

namespace Northwind.WebApi.Controllers;

[ApiController]
[Route("[controller]")]

public class WeatherForecastController : ControllerBase
{
  private static readonly string[] Summaries = new[]
  {
    "Freezing", "Bracing", "Chilly", "Cool", "Mild",
    "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
  };

  private readonly ILogger<WeatherForecastController> _logger;

  public WeatherForecastController(
    ILogger<WeatherForecastController> logger)
  {
    _logger = logger;
  }

  [HttpGet]
  public IEnumerable<WeatherForecast> Get()
  {
    return Enumerable.Range(1, 5).Select(index =>
      new WeatherForecast
      {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = Random.Shared.Next(-20, 55),
```

```
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .ToArray();
    }
}
```

Просматривая этот код, обратите внимание на следующие моменты:

- класс `Controller` наследуется от `ControllerBase`. Эта процедура проще, чем класс `Controller`, используемый в MVC, поскольку он не содержит таких методов, как `View`, для генерации HTML-ответов путем передачи модели представления в файл Razor;
- атрибут `[Route]` регистрирует относительный URL `/weatherforecast` для клиентов, чтобы использовать его для выполнения HTTP-запросов, которые будут обрабатываться этим контроллером. Например, данный контроллер будет обрабатывать HTTP-запрос для https://localhost:5001/weatherforecast/. Некоторым разработчикам нравится добавлять к имени контроллера префикс `api/`, что является соглашением, позволяющим в проектах различать MVC и Web API. Если вы введете `[controller]`, как показано в примере, то будут использованы символы из имени класса до слова `Controller`, в данном случае `WeatherForecast`, или вы можете просто ввести другое имя без скобок, например `[Route("api/forecast")]`;
- атрибут `[ApiController]` был введен в ASP.NET Core 2.1 и обеспечивает специфическое для REST поведение контроллеров, например автоматические HTTP-ответы с кодом состояния `400` для недопустимых моделей, как вы увидите далее в этой главе;
- атрибут `[HttpGet]` регистрирует метод `Get` в классе `Controller` для ответа на HTTP-запросы `GET`, а его реализация использует общий объект `Random`, чтобы возвратить массив объектов `WeatherForecast` со случайными температурами и сводками погоды, такими как `Bracing` или `Balmy`, в течение следующих пяти дней.

5. Добавьте второй метод `Get` с целочисленным параметром `days`, который при вызове позволяет указывать, за какое количество дней должен быть прогноз:
   - добавьте комментарий перед исходным методом, чтобы показать результаты метода действия и URL, на который он отвечает;
   - добавьте новый метод с целочисленным параметром `days`;
   - вырежьте и вставьте исходные операторы кода реализации метода `Get` в новый метод `Get`;
   - измените его для создания интерфейса `Ienumerable` целых чисел вплоть до запрошенного количества дней и отредактируйте исходный метод `Get` для вызова нового метода `Get` и передачи значения `5`.

Методы должны выглядеть так, как показано в коде ниже (выделено жирным шрифтом):

```
// GET /weatherforecast
[HttpGet]
public Ienumerable<WeatherForecast> Get() // исходный метод
{
  return Get(5); // пятидневный прогноз
}

// GET /weatherforecast/7
[HttpGet("{days:int}")]
public Ienumerable<WeatherForecast> Get(int days) // новый метод
{
  return Enumerable.Range(1, days).Select(index =>
    new WeatherForecast
    {
      Date = DateTime.Now.AddDays(index),
      TemperatureC = Random.Shared.Next(-20, 55),
      Summary = Summaries[Random.Shared.Next(Summaries.Length)]
    })
    .ToArray();
}
```

В атрибуте `[HttpGet]` обратите внимание на шаблон формата маршрута `{days:int}`, ограничивающий параметр `days` значениями `int`.

Функциональность веб-сервисов

Проверим функциональность веб-сервисов.

1. Если вы работаете в программе Visual Studio, на панели Properties (Свойства) откройте файл `launchSettings.json` и обратите внимание, что по умолчанию он запускает браузер и переходит на относительный URL-путь `/swagger`, как показано ниже:

```
"profiles": {
  "Northwind.WebApi": {
    "commandName": "Project",
    "dotnetRunMessages": "true",
    "launchBrowser": true,
    "launchUrl": "swagger",
    "applicationUrl": "https://localhost:5001;http://localhost:5000",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
```

2. Измените профиль `Northwind.WebApi`, присвоив параметру `launchBrowser` значение `false`.

3. Для параметра `applicationUrl` смените случайный номер порта для HTTP на `5000`, а для HTTPS — на `5001`.
4. Запустите проект веб-сервиса.
5. Запустите браузер Google Chrome.
6. Перейдите по адресу https:/localhost:5001/ и обратите внимание, что вы получите ответ с кодом состояния `404`, поскольку мы не включили статические файлы и отсутствует файл `index.html`, а также контроллер MVC с настроенным маршрутом. Данный проект не предназначен для просмотра и взаимодействия с пользователем, так что это ожидаемое поведение для веб-сервиса.

В моем примере на GitHub используется порт 5002. Позднее мы изменим его.

7. В браузере Google Chrome отобразите панель Developer tools (Инструменты разработчика).
8. Перейдите по адресу https:/localhost:5001/weatherforecast и обратите внимание, что сервис Web API должен возвращать документ JSON с пятью объектами случайного прогноза погоды в массиве (рис. 16.1).

Рис. 16.1. Запрос и ответ от веб-сервиса прогноза погоды

9. Закройте панель Developer Tools (Инструменты разработчика).
10. Перейдите по адресу https:/localhost:5001/weatherforecast/14 и обратите внимание на ответ при запросе двухнедельного прогноза погоды (рис. 16.2).
11. Закройте браузер Google Chrome и завершите работу веб-сервера.

Рис. 16.2. Двухнедельный прогноз погоды в формате документа JSON

Создание веб-сервиса для базы данных Northwind

В отличие от контроллеров MVC контроллеры Web API не вызывают представления Razor для возврата пользователям сайта HTML-ответов в их браузеры. Вместо этого контроллеры согласовывают контент с клиентским приложением, производящим HTTP-запрос, для возврата в HTTP-ответе данных в таких форматах, как XML, JSON или X-WWW-FORM-URLENCODED.

Затем клиентское приложение должно десериализовать данные из согласованного формата. В современных сервисах наиболее широко используется формат *JavaScript Object Notation (JSON)*, поскольку он достаточно компактный и отлично поддерживает JavaScript в браузере при создании *одностраничных приложений* (Single Page Applications, SPA) такими технологиями на стороне клиента, как Angular, React и Vue.

Мы будем ссылаться на сущностную модель данных Entity Framework Core для базы данных Northwind, созданную в главе 13.

1. В проекте `Northwind.WebApi` добавьте ссылку на проект `Northwind.Common.DataContext` для SQLite или SQL Server:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include=
  "..\Northwind.Common.DataContext.Sqlite\Northwind.Common.DataContext.
  Sqlite.csproj" />
</ItemGroup>
```

2. Соберите проект и исправьте возникшие ошибки компиляции в коде.
3. Откройте файл `Program.cs` и импортируйте пространства имен для работы с веб-медиаформатерами и общими классами `Packt`:

```
using Microsoft.AspNetCore.Mvc.Formatters;
using Packt.Shared; // метод расширения AddNorthwindContext

using static System.Console;
```

4. Добавьте оператор перед вызовом `AddControllers` для регистрации класса контекста базы данных Northwind (будет использоваться либо SQLite, либо SQL Server в зависимости от того, на какого поставщика базы данных вы сослались в файле проекта):

```
// добавление сервисов в контейнер
builder.Services.AddNorthwindContext();
```

5. В вызове `AddControllers` добавьте блок лямбда-выражений для вывода в консоль имен форматеров вывода по умолчанию и поддерживаемых ими типов данных, а затем добавьте средства форматирования сериализатора XML:

```
builder.Services.AddControllers(options =>
{
  WriteLine("Default output formatters:");
  foreach (IOutputFormatter formatter in options.OutputFormatters)
  {
    OutputFormatter? mediaFormatter = formatter as OutputFormatter;
    if (mediaFormatter == null)
    {
      WriteLine($" {formatter.GetType().Name}");
    }
    else // класс форматера вывода с поддерживаемыми медиаформатами
    {
      WriteLine(" {0}, Media types: {1}",
        arg0: mediaFormatter.GetType().Name,
        arg1: string.Join(", ",
          mediaFormatter.SupportedMediaTypes));
    }
  }
})
.AddXmlDataContractSerializerFormatters()
.AddXmlSerializerFormatters();
```

6. Запустите веб-сервис.
7. Обратите внимание, что в командной строке/терминале имеется четыре форматера вывода по умолчанию, в том числе те, которые преобразуют значения `null` в код состояния `204 No Content`, и те, которые поддерживают ответы в формате простого текста, потока байтов и JSON:

```
Default output formatters:
  HttpNoContentOutputFormatter
  StringOutputFormatter, Media types: text/plain
  StreamOutputFormatter
  SystemTextJsonOutputFormatter, Media types: application/json, text/json,
Application/*+json
```

8. Завершите работу веб-сервиса.

Создание репозиториев данных для сущностей

Для обеспечения операций CRUD отлично подходит определение и реализация хранилища данных (репозитория). Аббревиатура CRUD включает в себя следующие операции:

- *C* (Create) — создание;
- *R* (Retrieve, Read) — извлечение (или чтение);
- *U* (Update) — обновление;
- *D* (Delete) — удаление.

Создадим хранилище данных для таблицы `Customers` в базе данных Northwind. В этой таблице всего 91 клиент, поэтому мы будем хранить копию всей таблицы в памяти, чтобы улучшить масштабируемость и производительность при чтении записей о клиентах.

В реальном веб-сервисе вам необходимо использовать распределенный кэш, например Redis, хранилище структур данных с открытым исходным кодом, которое можно применять в качестве высокопроизводительной базы данных с высокой доступностью, кэша или брокера сообщений.

Будем следовать современным технологиям и сделаем API хранилища асинхронным. Мы будем использовать класс `Controller` с инъекцией параметров через конструктор, поэтому создается новый экземпляр контроллера для обработки каждого HTTP-запроса.

1. В проекте `Northwind.WebApi` создайте папку Repositories.
2. В папку Repositories добавьте два файла классов с именами `ICustomerRepository.cs` и `CustomerRepository.cs`.
3. Интерфейс `IcustomerRepository` определит пять методов:

```
using Packt.Shared; // Customer

namespace Northwind.WebApi.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);
}
```

4. Класс `CustomerRepository` будет реализовывать эти пять методов, помня, что методы, использующие внутри себя `await`, должны быть помечены как `async`:

```
using Microsoft.EntityFrameworkCore.ChangeTracking; // EntityEntry<T>
using Packt.Shared; // Customer
using System.Collections.Concurrent; // ConcurrentDictionary

namespace Northwind.WebApi.Repositories;

public class CustomerRepository : ICustomerRepository
{
  // используем статическое потокобезопасное поле словаря
  // для кэширования клиентов
  private static ConcurrentDictionary
    <string, Customer>? customersCache;

  // используем поле контекста данных экземпляра, поскольку
  // оно не должно кэшироваться из-за их внутреннего кэширования
  private NorthwindContext db;

  public CustomerRepository(NorthwindContext injectedContext)
  {
    db = injectedContext;

    // предварительно загружаем клиентов из базы данных как обычный
    // словарь с идентификатором клиента в качестве ключа,
    // затем преобразуем в потокобезопасный ConcurrentDictionary
    if (customersCache is null)
    {
      customersCache = new ConcurrentDictionary<string, Customer>(
        db.Customers.ToDictionary(c => c.CustomerId));
    }
  }

  public async Task<Customer?> CreateAsync(Customer c)
  {
    // нормализуем значения CustomerId в прописные
    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)
    {
      if (customersCache is null) return c;
      // нового клиента добавляем в кэш, иначе вызываем метод UpdateCache
      return customersCache.AddOrUpdate(c.CustomerId, c, UpdateCache);
    }
    else
```

```
    {
      return null;
    }
  }

  public Task<IEnumerable<Customer>> RetrieveAllAsync()
  {
    // в целях производительности извлекаем из кэша
    return Task.FromResult(customersCache is null
        ? Enumerable.Empty<Customer>() : customersCache.Values);
  }

  public Task<Customer?> RetrieveAsync(string id)
  {
    // в целях производительности извлекаем из кэша
    id = id.ToUpper();
    if (customersCache is null) return null!;
    customersCache.TryGetValue(id, out Customer? c);
    return Task.FromResult(c);
  }

  private Customer UpdateCache(string id, Customer c)
  {
    Customer? old;
    if (customersCache is not null)
    {
      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)
  {
    // нормализуем идентификатор клиента
    id = id.ToUpper();
    c.CustomerId = c.CustomerId.ToUpper();

    // обновляем в базе
    db.Customers.Update(c);
    int affected = await db.SaveChangesAsync();
    if (affected == 1)
    {
      // обновляем в кэше
      return UpdateCache(id, c);
    }
```

```
      return null;
    }

    public async Task<bool?> DeleteAsync(string id)
    {
      id = id.ToUpper();

      // удаляем из базы данных
      Customer? c = db.Customers.Find(id);
      if (c is null) return null;
      db.Customers.Remove(c);
      int affected = await db.SaveChangesAsync();
      if (affected == 1)
      {
        if (customersCache is null) return null;
        // удаляем из кэша
        return customersCache.TryRemove(id, out c);
      }
      else
      {
        return null;
      }
    }
  }
```

Реализация контроллера Web API

Существует несколько полезных атрибутов и методов для реализации контроллера, возвращающего данные вместо HTML.

В контроллерах MVC такой путь, как `/home/index`, сообщает нам имя класса контроллера и имя метода действия, например класс `HomeController` и метод действия `Index`.

В контроллерах Web API такой путь, как `/weatherforecast`, дает только указание использовать имя класса контроллера, например `WeatherForecastController`. Чтобы определить имя метода действия для выполнения, мы должны отобразить методы HTTP, такие как `GET` и `POST`, в методы в классе контроллера.

Чтобы указать методы HTTP, на которые должен производиться ответ, необходимо дополнить методы контроллера следующими методами действия:

- `[HttpGet]`, `[HttpHead]` — отвечают на запросы `GET` или `HEAD`, чтобы получить ресурс и вернуть либо ресурс и его заголовки ответа, либо только заголовки ответа;
- `[HttpPost]` — отвечает на запросы `POST` для создания нового ресурса или выполняет какое-либо другое действие, определенное сервисом;

- `[HttpPut]`, `[HttpPatch]` — отвечают на запросы `PUT` или `PATCH`, чтобы обновить существующий ресурс, либо заменив его, либо обновив подмножество его свойства;
- `[HttpDelete]` — отвечает на запросы `DELETE` для удаления ресурса;
- `[HttpOptions]` — отвечает на запросы `OPTIONS`.

Типы возвращаемых методов действия

Метод действия может возвращать типы .NET, такие как одно значение `string`, сложные объекты, определенные классом, записью или структурой, либо коллекции сложных объектов. Web API в ASP.NET Core сериализует их в запрошенный формат данных, заданный в заголовке `Accept` HTTP-запроса, например JSON, если был зарегистрирован подходящий сериализатор.

Осуществлять больший контроль над ответом помогают вспомогательные методы, которые возвращают оболочку `ActionResult` вокруг типа .NET.

Объявите тип возвращаемого значения метода действия как `IActionResult`, если он может возвращать разные типы на основе входных данных или других переменных. Объявите тип возвращаемого значения метода действия как `ActionResult<T>`, если он вернет только один тип, но с другими кодами состояния.

Дополните методы действий атрибутом [ProducesResponseType], чтобы указать все известные типы и коды состояния HTTP, которые клиент должен ожидать в ответе. Затем эту информацию можно публично предоставить для документирования сведений о взаимодействии клиента с вашим веб-сервисом. Это будет частью вашей официальной документации. Далее в главе вы узнаете, как можно установить анализатор кода, чтобы получать предупреждения, если вы не дополняете свои методы действий.

Например, метод действия, который получает товар на основе параметра `id`, будет дополнен тремя атрибутами: один для обозначения того, что он отвечает на запросы `GET` и имеет параметр `id`, и два для обозначения того, что происходит при успешном выполнении и при предоставлении клиентом неверного идентификатора товара, как показано ниже:

```
[HttpGet("{id}")]
[ProducesResponseType(200, Type = typeof(Product))]
[ProducesResponseType(404)]
Public IactionResult Get(string id)
```

Класс `ControllerBase` содержит методы, позволяющие легко возвращать разные ответы (табл. 16.3).

Таблица 16.3. Методы класса ControllerBase

Метод	Описание
`Ok`	Возвращает код состояния `200` с ресурсом, преобразованным в предпочтительный для клиента формат, такой как JSON или XML. Обычно используется в ответ на запрос `GET`
`CreatedAtRoute`	Возвращает код состояния `201` с указанием пути к новому ресурсу. Обычно используется в ответ на запрос `POST` для создания ресурса, который можно быстро отправить
`Accepted`	Возвращает код состояния `202` для обозначения, что запрос обрабатывается, но не завершен. Обычно используется в ответ на запрос `POST`, `PUT`, `PATCH` или `DELETE`, запускающий фоновый процесс, выполнение которого занимает много времени
`NoContentResult`	Возвращает код состояния `204`. Обычно используется в ответ на запрос `PUT`, `PATCH` или `DELETE`, когда ответ не должен содержать обновленный ресурс
`BadRequest`	Возвращает код состояния `400` и необязательную строку сообщения с более подробной информацией
`NotFound`	Возвращает код состояния `404` с автоматически заполненным телом `ProblemDetails` (требуется версия совместимости 2.2 или более поздняя)

Настройка репозитория клиента и контроллера Web API

Настроим хранилище так, чтобы оно могло вызываться из контроллера Web API.

При запуске веб-сервиса вы зарегистрируете реализацию сервиса как зависимость с заданной областью (scoped) для хранилища данных, а затем воспользуетесь внедрением параметров конструктора, чтобы получить его в новом контроллере Web API для работы с клиентами.

Чтобы показать пример различия между контроллерами MVC и Web API с помощью маршрутов, мы будем применять общеизвестное соглашение о добавлении к URL префикса `/api` для контроллера клиентов.

1. Откройте файл `Program.cs` и импортируйте пространство имен `Northwind.WebApi.Repositories`.
2. Добавьте оператор перед вызовом метода `Build`, который зарегистрирует `CustomerRepository` для использования во время выполнения в качестве зависимости с ограниченной областью действия, как показано ниже:

```
builder.Services.AddScoped<ICustomerRepository, CustomerRepository>();

var app = builder.Build();
```

Наше хранилище использует контекст базы данных, который зарегистрирован как scoped dependency (зависимость с ограниченной областью действия). Эти зависимости можно использовать только внутри других таких же зависимостей, поэтому мы не можем зарегистрировать хранилище как одноэлементное. Подробнее об этом можно прочитать по ссылке https://docs.microsoft.com/ru-ru/dotnet/core/extensions/dependency-injection#scoped.

3. В папку Controllers добавьте новый класс `CustomersController.cs`.
4. В файле класса `CustomersController` добавьте следующие операторы, чтобы определить класс контроллера Web API для работы с клиентами:

```
using Microsoft.AspNetCore.Mvc; // [Route], [ApiController],
                                   ControllerBase
using Packt.Shared; // Customer
using Northwind.WebApi.Repositories; // IcustomerRepository

namespace Northwind.WebApi.Controllers;

// базовый адрес: api/customers
[Route("api/[controller]")]
[ApiController]
public class CustomersController : ControllerBase
{

  private readonly ICustomerRepository repo;

  // конструктор внедряет репозиторий, зарегистрированный в Startup
  public CustomersController(ICustomerRepository repo)
  {
    this.repo = repo;
  }

  // GET: api/customers
  // GET: api/customers/?country=[country]
  // всегда возвращается список клиентов (может быть пустым)
  [HttpGet]
  [ProducesResponseType(200, Type = typeof(IEnumerable<Customer>))]
  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 = nameof(GetCustomer))] // именованный маршрут
[ProducesResponseType(200, Type = typeof(Customer))]
[ProducesResponseType(404)]
public async Task<IActionResult> GetCustomer(string id)
{
  Customer? c = await repo.RetrieveAsync(id);
  if (c == null)
  {
    return NotFound(); // 404 – ресурс не найден
  }
  return Ok(c); // 200 – OK, с клиентом в теле
}

// POST: api/customers
// BODY: Customer (JSON, XML)
[HttpPost]
[ProducesResponseType(201, Type = typeof(Customer))]
[ProducesResponseType(400)]
public async Task<IActionResult> Create([FromBody] Customer c)
{
  if (c == null)
  {
    return BadRequest(); // 400 – некорректный запрос
  }

  Customer? addedCustomer = await repo.CreateAsync(c);

  if (addedCustomer == null)
  {
    return BadRequest("Repository failed to create customer.");
  }
  else
  {
    return CreatedAtRoute( // 201 – ресурс создан
      routeName: nameof(GetCustomer),
      routeValues: new { id = addedCustomer.CustomerId.ToLower() },
      value: addedCustomer);
  }
}

// PUT: api/customers/[id]
// BODY: Customer (JSON, XML)
[HttpPut("{id}")]
[ProducesResponseType(204)]
```

```
    [ProducesResponseType(400)]
    [ProducesResponseType(404)]
    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 – некорректный запрос
      }

      Customer? existing = await repo.RetrieveAsync(id);
      if (existing == null)
      {
        return NotFound(); // 404 – ресурс не найден
      }

      await repo.UpdateAsync(id, c);

      return new NoContentResult(); // 204 – контент отсутствует
    }

    // DELETE: api/customers/[id]
    [HttpDelete("{id}")]
    [ProducesResponseType(204)]
    [ProducesResponseType(400)]
    [ProducesResponseType(404)]
    public async Task<IActionResult> Delete(string id)
    {
      Customer? existing = await repo.RetrieveAsync(id);
      if (existing == null)
      {
        return NotFound(); // 404 – ресурс не найден
      }

      bool? deleted = await repo.DeleteAsync(id);

      if (deleted.HasValue && deleted.Value) // короткое замыкание AND
      {
        return new NoContentResult(); // 204 – контент отсутствует
      }
      else
      {
        return BadRequest( // 400 – некорректный запрос
          $"Customer {id} was found but failed to delete.");
      }
    }
  }
```

При анализе этого класса контроллера Web API обратите внимание на следующие моменты:

- класс `Controller` регистрирует маршрут, имя которого начинается с `api/`, и включает имя контроллера, то есть `api/customer`;
- в конструкторе используется внедрение зависимостей для получения экземпляра зарегистрированного хранилища для клиентов;
- существует пять методов действия для выполнения операций CRUD для клиентов: два `GET` (для всех клиентов или одного), `POST` (создание), `PUT` (обновление) и `DELETE` (удаление);
- в метод `GetCustomers` может передаваться параметр `string` с названием страны. Если такой параметр отсутствует, то в результате возвращаются все клиенты. Если присутствует, то используется для фильтрации клиентов по странам;
- метод `GetCustomer` содержит маршрут с явным названием `GetCustomer`. Его можно использовать для создания URL после добавления нового клиента;
- оба метода `Create` и `Update` дополняют параметр `customer` атрибутом `[FromBody]`, чтобы дать привязке модели указание заполнить его значениями из тела запроса `POST`;
- метод `Create` возвращает ответ, который использует маршрут `GetCustomer`, чтобы клиент знал, как в дальнейшем получить вновь созданный ресурс. Мы сопоставляем эти методы, чтобы создать, а затем найти клиента;
- методам `Create` и `Update` не нужно проверять состояние модели клиента, переданного в теле HTTP-запроса, и при неверном запросе возвращают сообщение `400 Bad Request`, содержащее сведения об ошибках проверки модели, поскольку контроллер дополнен атрибутом `[ApiController]`, который делает это за вас.

Получая HTTP-запрос, сервис создает экземпляр класса контроллера, вызывает соответствующий метод действия, возвращает ответ в предпочитаемом клиентом формате и освобождает используемые контроллером ресурсы, включая хранилище и его контекст данных.

Указание сведений о проблеме

Функция, добавленная в ASP.NET Core 2.1 и более поздних версиях, представляет собой реализацию веб-стандарта для указания сведений о проблеме.

В контроллерах, дополненных атрибутом `[ApiController]` в проекте с включенной совместимостью ASP.NET Core 2.2 или более поздней версии, методы действия, возвращающие параметр `IActionResult` и клиентский код состояния (то есть `4xx`), будут автоматически включать в тело ответа сериализованный экземпляр класса `ProblemDetails`.

Если вы хотите взять на себя управление, то можете самостоятельно создать экземпляр `ProblemDetails` и включить дополнительную информацию.

Создадим неверный запрос, который требует возврата клиенту пользовательских данных.

1. В начале реализации метода `Delete` добавьте операторы, чтобы проверить, соответствует ли идентификатор буквальному строковому значению `"bad"`, и если да, то верните пользовательский объект сведений о проблеме:

```
// управление сведениями о проблеме
if (id == "bad")
{
  ProblemDetails problemDetails = new()
  {
    Status = StatusCodes.Status400BadRequest,
    Type = "https://localhost:5001/customers/failed-to-delete",
    Title = $"Customer ID {id} found but failed to delete.",
    Detail = "More details like Company Name, Country and so on.",
    Instance = HttpContext.Request.Path
  };
  return BadRequest(problemDetails); // 400 – некорректный запрос
}
```

2. Вы протестируете эту функциональность позже.

Управление сериализацией XML

В файл `Program.cs` добавим `XmlSerializer`, чтобы наш сервис Web API мог возвращать XML, а также JSON, если клиент делает такой запрос.

Однако `XmlSerializer` не может сериализовать интерфейсы. Для определения связанных дочерних сущностей наши сущностные классы используют `ICollection<T>`. Во время выполнения это вызовет предупреждение, например, для класса `Customer` и его свойства `Orders`, как показано ниже:

```
warn: Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerOutputFormatter[1]
An error occurred while trying to create an XmlSerializer for the type 'Packt.
Shared.Customer'.
System.InvalidOperationException: There was an error reflecting type 'Packt.
Shared.Customer'.
---> System.InvalidOperationException: Cannot serialize member 'Packt.
Shared.Customer.Orders' of type 'System.Collections.Generic.ICollection`1
[[Packt.Shared.Order, Northwind.Common.EntityModels, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null]]', see inner exception for more details.
```

Однако предупреждение можно предотвратить, исключив свойство `Orders` при сериализации класса `Customer` в XML.

1. В проектах `Northwind.Common.EntityModels.Sqlite` и `Northwind.Common.EntityModels.SqlServer` откройте файл `Customers.cs`.
2. Импортируйте пространство имен `System.Xml.Serialization`, чтобы мы могли использовать атрибут `[XmlIgnore]`.
3. Дополните свойство `Orders` атрибутом, чтобы игнорировать его при сериализации, как показано ниже:

```
[InverseProperty(nameof(Order.Customer))]
[XmlIgnore]
Public virtual Icollection<Order> Orders { get; set; }
```

4. В проекте `Northwind.Common.EntityModels.SqlServer` также дополните свойство `CustomerTypes` атрибутом `[XmlIgnore]`.

Документирование и тестирование веб-сервисов

Вы можете легко протестировать веб-сервис, отправляя в браузере HTTP-запросы `GET`. Чтобы проверить другие HTTP-методы, потребуется более совершенный инструмент.

Тестирование GET-запросов в браузерах

Вы будете применять браузер Google Chrome, чтобы тестировать три реализации запроса `GET`: для всех клиентов, для клиентов в указанной стране и для одного клиента, используя его уникальный идентификатор.

1. Запустите веб-сервис `Northwind.WebApi`.
2. Запустите браузер Google Chrome.
3. Перейдите по адресу http://localhost:5001/api/customers. Вы должны увидеть возвращенный JSON-файл, в котором содержатся все клиенты (91) в базе данных Northwind (без сортировки) (рис. 16.3).

Рис. 16.3. Список клиентов из базы данных Northwind в формате документа JSON

4. Перейдите по адресу localhost:5001/api/customers/?country=Germany. Вы должны увидеть возвращенный JSON-файл, содержащий только клиентов из Германии (рис. 16.4).

Рис. 16.4. Список клиентов, проживающих в Германии, в формате документа JSON

Если вы получите пустой массив, то убедитесь, что ввели название страны, используя правильный регистр, поскольку запрос к базе данных чувствителен к регистру. Например, сравните результаты uk и UK.

5. Перейдите по адресу localhost:5001/api/customers/alfki. Вы должны увидеть возвращенный JSON-файл, содержащий только одну строку — клиент Alfreds Futterkiste (рис. 16.5).

Рис. 16.5. Конкретная информация о клиенте в формате документа JSON

Вы не должны беспокоиться о регистре для значения идентификатора клиента, в отличие от названий стран, поскольку внутри класса `Controller` мы в программном коде нормализовали значение `string` в верхнем регистре.

Однако как мы можем проверить другие методы HTTP, такие как `POST`, `PUT` и `DELETE`? И как задокументировать наш веб-сервис, чтобы каждому было легко понять, как взаимодействовать с ним?

Чтобы решить первую проблему, мы можем установить расширение Visual Studio Code под названием *REST Client*. Чтобы решить вторую — можем использовать *Swagger*, самую популярную технологию для документирования и тестирования HTTP API. Однако в первую очередь рассмотрим возможности расширения Visual Studio Code.

Существует множество инструментов для тестирования Web API, например Postman. Невзирая на его популярность, я предпочитаю REST Client, поскольку он не скрывает от пользователя обработку. Мне кажется, что Postman излишне упрощен в плане интерфейса. Но я рекомендую вам изучить различные инструменты и найти подходящие именно вам. Вы можете узнать больше о Postman здесь: https://www.postman.com/.

Тестирование HTTP-запросов с помощью расширения REST Client

REST Client позволяет отправлять HTTP-запрос и просматривать ответ непосредственно в программе Visual Studio Code. Даже если вы предпочитаете использовать Visual Studio в качестве редактора кода, полезно установить программу Visual Studio Code, чтобы использовать такое расширение, как REST Client.

Выполнение GET-запросов с помощью REST Client

Мы начнем с создания файла для тестирования запросов `GET`.

1. Если вы еще не установили REST Client от Huachao Mao (`humao.rest-client`), то сейчас установите его в программе Visual Studio Code.
2. Откройте редактор кода и запустите веб-сервис проекта `Northwind.WebApi`.
3. В программе Visual Studio Code в папке PracticalApps создайте папку RestClientTests, а затем откройте ее.
4. В папке RestClientTests создайте файл `get-customers.http` и отредактируйте его код, чтобы в нем содержался HTTP-запрос `GET` для получения всех клиентов:

```
GET https://localhost:5001/api/customers/ HTTP/1.1
```

5. В программе Visual Studio Code выберите команду меню View ▸ Command Palette (Вид ▸ Панель команд), введите `rest client`, выберите команду Rest Client: Send Request (Rest Client: отправить запрос) и нажмите клавишу Enter (рис. 16.6).

Рис. 16.6. Отправка HTTP-запроса GET с помощью REST Client

6. Обратите внимание: вкладка Response (Ответ) отображается в новой области окна с вкладками, расположенными по вертикали, и вы можете перераспределить открытые вкладки по горизонтали, перетаскивая их.
7. Введите больше запросов `GET`, каждый из которых разделен тремя хеш-символами, чтобы проверить наличие клиентов в разных странах и получение информации об одном клиенте с помощью его ID:

```
###
GET https://localhost:5001/api/customers/?country=Germany HTTP/1.1
###
GET https://localhost:5001/api/customers/?country=USA HTTP/1.1
Accept: application/xml
###
GET https://localhost:5001/api/customers/ALFKI HTTP/1.1
###

GET https://localhost:5001/api/customers/abcxy HTTP/1.1
```

8. Щелкните на ссылке Send Request (Отправить запрос) над каждым запросом, чтобы отправить его; например, `GET` с заголовком запроса клиентов в США в формате XML вместо JSON, как показано на рис. 16.7.

Рис. 16.7. Отправка запроса на XML и получение ответа с помощью Rest Client

Выполнение прочих запросов с помощью REST Client

Далее мы создадим файл для тестирования других запросов, например `POST`.

1. В папке RestClientTests создайте файл `create-customer.http` и измените его содержимое, чтобы определить запрос `POST` для создания нового клиента. Обра-

тите внимание: REST Client будет отображать подсказки IntelliSense, если вы вводите обычные HTTP-запросы:

```
POST https://localhost:5001/api/customers/ HTTP/1.1
Content-Type: application/json
Content-Length: 266

{
  "customerID": "ABCXY",
  "companyName": "ABC Corp",
  "contactName": "John Smith",
  "contactTitle": "Sir",
  "address": "Main Street",
  "city": "New York",
  "region": "NY",
  "postalCode": "90210",
  "country": "USA",
  "phone": "(123) 555-1234"
}
```

2. Из-за разнообразия окончаний строк в разных операционных системах значение заголовка `Content-Length` будет разным в Windows и macOS или Linux. Если значение неверно, то запрос завершится ошибкой. Чтобы определить правильную длину содержимого, выделите тело запроса, а затем найдите в строке состояния количество символов (рис. 16.8).

Рис. 16.8. Проверка правильности длины содержимого

3. Отправьте запрос и обратите внимание, что в ответе возвращается код состояния `201 Created`. Обратите также внимание, что местоположением (то есть URL) вновь созданного клиента является https://localhost:5001/api/Customers/abcxy и оно включает вновь созданного клиента в тело ответа (рис. 16.9).

В качестве дополнительной задачи я оставлю вам создание файлов REST Client, которые проверяют обновления клиента (с помощью `PUT`) и удаление клиента (с помощью `DELETE`). Опробуйте их как на существующих, так и на несуществующих клиентах. Решения, касающиеся этой книги, находятся в репозитории GitHub.

Теперь, когда мы изучили быстрый и простой способ тестирования нашего сервиса, который также считается отличным способом изучения HTTP, как насчет внешних разработчиков? Мы хотим, чтобы им было максимально просто узнать, как работать с нашим сервисом, а затем вызвать его. Для этой цели мы будем использовать Swagger.

Рис. 16.9. Добавление нового клиента

Swagger

Самая важная часть Swagger — это спецификация OpenAPI, которая определяет контракт в стиле REST для вашего API, подробно описывая все его ресурсы и операции в удобном для пользователя и машины формате, что упрощает разработку, обнаружение и интеграцию.

С помощью спецификации OpenAPI для Web API разработчики могут автоматичеси создавать сильно типизированный код на стороне клиента на предпочитаемом ими языке или в библиотеке.

Еще одна полезная для нас функция — *Swagger UI*, автоматически генерирующая документацию для вашего API со встроенными возможностями визуального тестирования.

Добавим библиотеку Swagger для нашего веб-сервиса, используя пакет `Swashbuckle`.

1. Если веб-сервис работает, то завершите работу веб-сервера.
2. Откройте проект `Northwind.WebApi.csproj` и обратите внимание на ссылку на пакет для `Swashbuckle.AspNetCore`, как показано ниже:

```
<ItemGroup>
  <PackageReference Include="Swashbuckle.AspNetCore" Version="6.1.5" />
</ItemGroup>
```

3. Обновите версию пакета `Swashbuckle.AspNetCore` до последней (на момент написания этой книги в сентябре 2021 года это была версия 6.2.1).
4. В файле `Program.cs` обратите внимание на импорт пространства имен моделей OpenAPI от Microsoft:

```
using Microsoft.OpenApi.Models;
```

5. Импортируйте пространство имен SwaggerUI для Swashbuckle:

```
using Swashbuckle.AspNetCore.SwaggerUI; // SubmitMethod
```

6. Примерно в середине файла `Program.cs` обратите внимание на оператор для добавления поддержки Swagger, включая документацию для сервиса Northwind, указывающую, что это первая версия вашего сервиса:

```
builder.Services.AddSwaggerGen(c =>
  {
    c.SwaggerDoc("v1", new()
      { Title = "Northwind Service API", Version = "v1" });
  });
```

7. В разделе, где настраивается конвейер запросов HTTP, обратите внимание на операторы для использования Swagger и Swagger UI в режиме разработки и определите конечную точку для JSON-документа спецификации OpenAPI.
8. Добавьте код для явного перечисления методов HTTP, которые будут поддерживаться в нашем веб-сервисе, и измените имя конечной точки:

```
var app = builder.Build();

// настраиваем конвейер HTTP-запросов
if (builder.Environment.IsDevelopment())
{
  app.UseSwagger();
  app.UseSwaggerUI(c =>
  {
    c.SwaggerEndpoint("/swagger/v1/swagger.json",
      "Northwind Service API Version 1");

    c.SupportedSubmitMethods(new[] {
      SubmitMethod.Get, SubmitMethod.Post,
      SubmitMethod.Put, SubmitMethod.Delete });
  });
}
```

Тестирование запросов с помощью Swagger UI

Теперь вы готовы протестировать HTTP-запрос с помощью Swagger.

1. Запустите веб-сервис `Northwind.WebApi`.
2. В браузере Google Chrome перейдите по адресу https://localhost:5001/swagger/. Обратите внимание: были обнаружены и задокументированы контроллеры Web API Customer и WeatherForecast, а также Schemas, используемые API.
3. Щелкните на GET/api/Customers/{id}, чтобы развернуть эту конечную точку. Вы увидите необходимый параметр для идентификатора клиента (рис. 16.10).

Рис. 16.10. Проверка параметров GET-запроса в Swagger

4. Нажмите кнопку Try it out (Попробовать), введите идентификатор `ALFKI`, а затем нажмите широкую синюю кнопку Execute (Выполнить) (рис. 16.11).

Рис. 16.11. Ввод идентификатора клиента перед нажатием кнопки Execute (Выполнить)

5. Прокрутите вниз и обратите внимание на Request URL (URL запроса), Server response (Ответ сервера) с Code (Код) и Details (Сведения), включая Response body (Тело ответа) и Response headers (Заголовки ответа) (рис. 16.12).

Рис. 16.12. Информация об идентификаторе ALFKI при успешном запросе Swagger

6. Прокрутите вверх, выберите POST/api/Customers, чтобы развернуть этот раздел, затем нажмите кнопку Try it out (Попробовать).
7. Щелкните кнопкой мыши внутри поля Request body (Тело запроса) и измените JSON-файл, чтобы определить нового клиента:

```
{
  "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"
}
```

8. Нажмите кнопку Execute (Выполнить) и обратите внимание на Request URL (URL запроса), Server response (Ответ сервера) с Code (Код) и Details (Сведения), включая Response body (Тело ответа) и Response headers (Заголовки ответа), отмечая, что код ответа `201` означает успешное создание клиента (рис. 16.13).

Рис. 16.13. Успешное добавление нового клиента

9. Прокрутите вверх, выберите GET/api/Customers, нажмите кнопку Try it out (Попробовать), введите в качестве параметра страны `Denmark` и нажмите кнопку Execute (Выполнить), чтобы подтвердить добавление нового клиента в базу данных (рис. 16.14).

10. Выберите DELETE/api/Customers/{id}, затем нажмите кнопку Try it out (Попробовать). Введите `super` для идентификатора id и нажмите кнопку Execute (Выполнить). Обратите внимание: Server response Code (Код ответа сервера) — `204`, что указывает на успешное удаление (рис. 16.15).

11. Еще раз нажмите кнопку Execute (Выполнить). Обратите внимание: Server response Code (Код ответа сервера) `404` указывает на то, что клиент больше не существует, а Response body (Тело ответа) содержит JSON-документ с подробным описанием проблемы (рис. 16.16).

Рис. 16.14. Получение списка клиентов, проживающих в Дании, включая недавно добавленного клиента

Рис. 16.15. Успешное удаление клиента

Рис. 16.16. Удаленный клиент больше не существует

12. Введите bad для id, снова нажмите кнопку Execute (Выполнить). Обратите внимание: Server response Code (Код ответа сервера) `400` указывает на то, что клиент существует, но его не удалось удалить (в данном случае ввиду того, что веб-сервис имитирует эту ошибку), а Response body (Тело ответа) содержит JSON-документ с подробным описанием проблемы (рис. 16.17).

Рис. 16.17. Клиент действительно существует, но не может быть удален

13. Используйте методы `GET`, чтобы подтвердить удаление нового клиента из базы данных (первоначально в Дании было только два клиента).

Я оставлю читателю тестирование обновления для существующего клиента, используя метод PUT.

14. Закройте браузер Google Chrome и завершите работу веб-сервера.

Протоколирование HTTP

Протоколирование HTTP — это необязательный компонент промежуточного программного обеспечения, который регистрирует информацию о HTTP-запросах и HTTP-ответах, включая следующее:

- информацию о HTTP-запросе;
- заголовки;
- тело;
- информацию об HTTP-ответе.

Функция пригодится в веб-сервисах для аудита и отладки сценариев, но будьте осторожны, поскольку она может негативно сказаться на производительности. Вы также можете записывать в журнал *информацию личного порядка* (personally identifiable information, PII), что может вызвать проблемы с соответствием требованиям в некоторых юрисдикциях.

Посмотрим, как выполняется протоколирование HTTP.

1. В файле `Program.cs` импортируйте пространство имен для работы с протоколированием HTTP:

```
using Microsoft.AspNetCore.HttpLogging; // HttpLoggingFields
```

2. В разделе конфигурации сервисов добавьте оператор для настройки протоколирования HTTP:

```
builder.Services.AddHttpLogging(options =>
{
  options.LoggingFields = HttpLoggingFields.All;
  options.RequestBodyLogLimit = 4096; // по умолчанию 32 Кбайт
  options.ResponseBodyLogLimit = 4096; // по умолчанию 32 Кбайт
});
```

3. В разделе конфигурации HTTP-конвейера укажите оператор для добавления HTTP-протоколирования перед вызовом маршрутизации:

```
app.UseHttpLogging();
```

4. Запустите веб-сервис `Northwind.WebApi`.

5. Запустите браузер Google Chrome.
6. Перейдите по адресу https://localhost:5001/api/customers.
7. В командной строке/терминале обратите внимание, что запрос и ответ были зарегистрированы, как показано ниже:

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/1.1
      Method: GET
      Scheme: https
      PathBase:
      Path: /api/customers
      QueryString:
      Connection: keep-alive
      Accept: */*
      Accept-Encoding: gzip, deflate, br
      Host: localhost:5001

info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Content-Type: application/json; charset=utf-8
      ...
      Transfer-Encoding: chunked
```

8. Закройте браузер Google Chrome и завершите работу веб-сервера.

Обращение к веб-сервисам с помощью HTTP-клиентов

Теперь, создав и протестировав сервис Northwind, мы узнаем, как вызывать его из любого приложения .NET Core, используя класс `HttpClient` и его настройки.

Класс HttpClient

Самый простой способ обратиться к веб-сервису — это применить класс `HttpClient`. Однако многие разработчики используют его неправильно, поскольку он реализует интерфейс `IDisposable`, и даже документация Microsoft показывает его некорректное применение. В репозитории GitHub есть ссылки на статьи с более подробным обсуждением этого вопроса.

Обычно, когда тип реализует интерфейс `IDisposable`, вы должны создать его внутри оператора `using`, чтобы гарантировать его максимально быстрое удаление. Класс `HttpClient` отличается тем, что является разделяемым, повторно используемым и частично безопасным для потоков.

Проблема связана с управлением базовыми сетевыми сокетами. Суть в том, что вам необходимо использовать один экземпляр сокета для каждой конечной точки HTTP, которую вы потребляете в течение эксплуатации вашего приложения. Это позволит каждому экземпляру класса `HttpClient` установить значения по умолчанию, соответствующие конечной точке, с которой он работает, при этом эффективно управляя базовыми сетевыми сокетами.

Настройка HTTP-клиентов с помощью HttpClientFactory

Компания Microsoft знает об этой проблеме, и в ASP.NET Core 2.1 появился класс `HttpClientFactory`, помогающий решить часть распространенных проблем, с которыми могут столкнуться разработчики. Таким образом Microsoft продвигает использование лучших технологий для работы.

В следующем примере мы задействуем сайт MVC для Northwind в качестве клиента сервиса Web API для Northwind. Поскольку оба должны быть размещены на веб-сервере одновременно, нам в первую очередь необходимо настроить их для использования разных номеров портов:

- Northwind Web API будет прослушивать порт `5002` с помощью HTTPS;
- Northwind MVC продолжит прослушивать порт `5000` с помощью HTTP и порт `5002` с помощью HTTPS.

Настроим эти порты.

1. В проекте `Northwind.WebApi` в файле `Program.cs` добавьте вызов метода расширения `UseUrls`, чтобы указать номер порта `5002` для HTTPS, как показано ниже (выделено жирным шрифтом):

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseUrls("https://localhost:5002/");
```

2. В проекте `Northwind.Mvc` откройте файл `Program.cs` и импортируйте пространство имен для работы с фабрикой HTTP-клиентов:

```
using System.NET.Http.Headers; // MediaTypeWithQualityHeaderValue
```

3. Добавьте оператор, чтобы разрешить `HttpClientFactory` с именованным клиентом выполнять вызовы в сервис Web API Northwind с помощью HTTPS через порт `5002` и запрашивать JSON в качестве формата ответа по умолчанию:

```
builder.Services.AddHttpClient(name: "Northwind.WebApi",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5002/");
```

```
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

Получение контроллером списка клиентов в формате JSON

Теперь мы можем создать метод действия контроллера MVC, использующий фабрику для создания HTTP-клиента, делает запрос `GET` для клиентов и десериализует ответ JSON с помощью удобных методов расширения, представленных в .NET 5 в сборке `System.NET.Http.Json` и пространстве имен.

1. Откройте файл `Controllers/HomeController.cs` и объявите поле для хранения фабрики HTTP-клиентов:

```
Private readonly IhttpClientFactory clientFactory;
```

2. Установите поле в конструкторе, как показано ниже:

```
public HomeController(
  ILogger<HomeController> logger,
  NorthwindContext injectedContext,
  IHttpClientFactory httpClientFactory)
{
  _logger = logger;
  db = injectedContext;
  clientFactory = httpClientFactory;
}
```

3. Создайте метод действия для вызова сервиса Northwind Web API, выборки всех клиентов и передачи их в представление:

```
public async Task<IActionResult> Customers(string country)
{
  string uri;

  if (string.IsNullOrEmpty(country))
  {
    ViewData["Title"] = "All Customers Worldwide";
    uri = "api/customers/";
  }
  else
  {
    ViewData["Title"] = $"Customers in {country}";
    uri = $"api/customers/?country={country}";
  }
```

```
    HttpClient client = clientFactory.CreateClient(
      name: "Northwind.WebApi");

    HttpRequestMessage request = new(
      method: HttpMethod.Get, requestUri: uri);

    HttpResponseMessage response = await client.SendAsync(request);

    Ienumerable<Customer>? Model = await response.Content
      .ReadFromJsonAsync<Ienumerable<Customer>>();

    return View(model);
  }
```

4. В папке Views/Home создайте Razor-файл `Customers.cshtml`.
5. Отредактируйте файл Razor для отображения клиентов:

```
@using Packt.Shared
@model IEnumerable<Customer>
<h2>@ViewData["Title"]</h2>
<table class="table">
  <thead>
    <tr>
      <th>Company Name</th>
      <th>Contact Name</th>
      <th>Address</th>
      <th>Phone</th>
    </tr>
  </thead>
  <tbody>
    @if (Model is not null)
    {
      @foreach (Customer c in Model)
      {
        <tr>
          <td>
            @Html.DisplayFor(modelItem => c.CompanyName)
          </td>
          <td>
            @Html.DisplayFor(modelItem => c.ContactName)
          </td>
          <td>
            @Html.DisplayFor(modelItem => c.Address)
            @Html.DisplayFor(modelItem => c.City)
            @Html.DisplayFor(modelItem => c.Region)
            @Html.DisplayFor(modelItem => c.Country)
            @Html.DisplayFor(modelItem => c.PostalCode)
          </td>
```

```
            <td>
              @Html.DisplayFor(modelItem => c.Phone)
            </td>
          </tr>
        }
      }
    </tbody>
  </table>
```

6. В файле `Views/Home/Index.cshtml` добавьте форму после подсчета количества пользователей, позволяющую пользователям выбрать страну и увидеть клиентов:

```
<h3>Query customers from a service</h3>
<form asp-action="Customers" method="get">
  <input name="country" placeholder="Enter a country" />
  <input type="submit" />
</form>
```

Совместное использование ресурсов между разными источниками

Совместное использование ресурсов между разными источниками (Cross-Origin Resource Sharing, CORS) — это стандарт защиты веб-ресурсов на основе HTTP-заголовков, когда клиент и сервер находятся на разных доменах (источниках). Это позволяет серверу указывать, с каких источников (определяемых комбинацией домена, схемы и порта), кроме собственного, он будет разрешать загрузку ресурсов.

Поскольку наш веб-сервис размещен на порте `5002`, а наш сайт MVC размещен на портах `5000` и `5001`, они считаются разными источниками, и поэтому ресурсы не могут быть общими.

Было бы полезно включить совместное использование ресурсов между источниками на сервере и настроить наш веб-сервис так, чтобы он разрешал только запросы, исходящие с сайта MVC.

1. В проекте `Northwind.WebApi` откройте файл `Program.cs`.
2. Добавьте оператор в раздел конфигурации сервисов, чтобы добавить поддержку CORS:

```
builder.Services.AddCors();
```

3. Добавьте оператор в раздел конфигурации HTTP-конвейера перед вызовом `UseEndpoints`, чтобы использовать CORS и разрешить выполнение запросов `GET`, `POST`, `PUT` и `DELETE` для любого сайта, такого как MVC для Northwind, имеющего источник https://localhost:5001:

```
var app = builder.Build();

// Конфигурация HTTP-конвейера
app.UseCors(configurePolicy: options =>
{
  options.WithMethods("GET", "POST", "PUT", "DELETE");
  options.WithOrigins(
    "https://localhost:5001" // допускаются запросы от клиента MVC
  );
});
```

4. Запустите проект `Northwind.WebApi` и убедитесь, что веб-сервис прослушивает только порт `5002`:

```
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: https://localhost:5002
```

5. Запустите проект `NorthwindMvc` и убедитесь, что сайт прослушивает порты `5000` и `5002`:

```
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: http://localhost:5000
```

6. Запустите браузер Google Chrome.
7. В форме клиента введите название страны, например `Germany`, `UK` или `USA`, нажмите кнопку Submit (Отправить), и увидите список клиентов (рис. 16.18).

Рис. 16.18. Клиенты, проживающие в Великобритании

8. Нажмите кнопку Back (Назад) в браузере, очистите текстовое поле, в котором было введено название страны, нажмите кнопку Submit (Отправить) и просмотрите полученный список клиентов.

9. В командной строке/терминале обратите внимание, что класс `HttpClient` записывает каждый HTTP-запрос, который он делает, и HTTP-ответ, который он получает:

```
info: System.NET.Http.HttpClient.Northwind.WebApi.ClientHandler[100]
  Sending HTTP request GET https://localhost:5002/api/
customers/?country=UK
info: System.NET.Http.HttpClient.Northwind.WebApi.ClientHandler[101]
  Received HTTP response headers after 931.864ms - 200
```

10. Закройте браузер Google Chrome и завершите работу веб-сервера.

Вы успешно создали веб-сервис и вызвали его на сайте MVC.

Реализация расширенных функций веб-сервисов

Теперь, когда вы ознакомились с основами создания веб-сервиса и с последующим его вызовом из клиента, рассмотрим некоторые дополнительные функции.

Реализация проверки работоспособности API

Существует множество платных сервисов, выполняющих проверку доступности сайта с помощью банального пингования, некоторые делают более продвинутый анализ HTTP-ответа.

ASP.NET Core 2.2 и более поздние версии позволяют легко выполнять более подробные проверки работоспособности сайта. Например, ваш сайт может работать, но готов ли он полностью? Может ли он получить данные из своей базы данных?

Добавим в наш веб-сервис базовые возможности проверки работоспособности.

1. В проекте `Northwind.WebApi` добавьте ссылку на проект, чтобы включить проверки работоспособности базы данных Entity Framework Core:

```
<PackageReference Include=
  "Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore"
  Version="6.0.0" />
```

2. Соберите проект.
3. В файле `Program.cs`, в нижней части раздела конфигурации сервисов, добавьте оператор для добавления проверок работоспособности, в том числе в контекст базы данных Northwind:

```
builder.Services.AddHealthChecks()
  .AddDbContextCheck<NorthwindContext>();
```

По умолчанию проверка контекста базы данных вызывает метод EF Core CanConnectAsync. Вы можете настроить выполнение операции, используя метод AddDbContextCheck.

4. В разделе конфигурации HTTP-конвейера, перед вызовом `MapControllers` добавьте оператор для использования базовых проверок работоспособности:

```
App.UseHealthChecks(path: "/howdoyoufeel");
```

5. Запустите веб-сервис.
6. Запустите браузер Google Chrome.
7. Перейдите по адресу https://localhost:5002/howdoyoufeel и обратите внимание, что в ответ отображается простой текст: `Healthy`.
8. В командной строке или терминале обратите внимание на оператор SQL, который был выполнен в целях проверки работоспособности базы данных:

```
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT 1
```

9. Закройте браузер Google Chrome и завершите работу веб-сервера.

Реализация анализаторов и соглашений Open API

В этой главе вы узнали, как включить Swagger для документирования веб-сервиса, вручную дополнив класс `Controller` атрибутами.

В ASP.NET Core 2.2 или более поздних версиях содержатся API-анализаторы, использующие отражение для классов контроллеров, которые были помечены атрибутом `[ApiController]` в целях их автоматического документирования. Анализатор предполагает некоторые соглашения API.

Чтобы использовать анализатор, ваш проект должен включить OpenAPI Analyzers, как показано ниже (выделено жирным шрифтом):

```
<PropertyGroup>
  <TargetFramework>net6.0</TargetFramework>
  <Nullable>enable</Nullable>
  <ImplicitUsings>enable</ImplicitUsings>
  <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
</PropertyGroup>
```

После установки контроллеры, которые не были оформлены должным образом, будут содержать предупреждения (зеленый знак тильды) и предупреждения при компиляции исходного кода. Например, класс `WeatherForecastController`.

Автоматические исправления кода могут затем добавить соответствующие атрибуты `[Produces]` и `[ProducesResponseType]`, хотя в настоящее время это работает только в среде разработки Visual Studio. В программе Visual Studio Code вы увидите предупреждение о том, что, по мнению анализатора, вам следует добавить атрибуты, но вы должны добавить их самостоятельно.

Обработка проходных отказов

Веб-сервис, который вызывает клиентское приложение или сайт, может быть в другой части света. Проблемы с сетью между клиентом и сервером могут вызвать проблемы, которые не имеют ничего общего с вашим кодом реализации. Если клиент делает вызов и при этом возникает ошибка, то приложение должно продолжать работать. При повторной попытке проблема может быть решена. Необходим способ, чтобы справиться с такими временными отказами.

Чтобы справиться с этими проходными отказами, Microsoft рекомендует с помощью сторонней библиотеки Polly реализовывать автоматические повторные попытки с экспоненциальной задержкой. Вы определяете политику, а библиотека обрабатывает все остальное.

Более подробную информацию, касающуюся библиотеки Polly, вы можете узнать на сайте https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/implement-resilient-applications/implement-http-call-retries-exponential-backoff-polly.

Добавление HTTP-заголовков в целях безопасности

ASP.NET Core имеет встроенную поддержку общих HTTP-заголовков безопасности, например HSTS. Помимо этого, существует еще множество HTTP-заголовков, которые следует изучить.

Самый простой способ добавить эти заголовки — использовать класс промежуточного программного обеспечения.

1. В проекте/папке `Northwind.WebApi` создайте файл `SecurityHeadersMiddleware.cs` и измените его операторы:

```
using Microsoft.Extensions.Primitives; // StringValues

public class SecurityHeaders
{
  private readonly RequestDelegate next;
```

```
    public SecurityHeaders(RequestDelegate next)
    {
      this.next = next;
    }

    public Task Invoke(HttpContext context)
    {
      // добавьте сюда желаемые заголовки ответа HTTP
      context.Response.Headers.Add(
        "super-secure", new StringValues("enable"));

      return next(context);
    }
  }
```

2. В файле `Program.cs`, в разделе конфигурации HTTP-конвейера, перед вызовом `UseEndpoints` добавьте оператор для регистрации промежуточного программного обеспечения:

```
app.UseMiddleware<SecurityHeaders>();
```

3. Запустите веб-сервис.
4. Запустите браузер Google Chrome.
5. Выберите Developer Tools (Инструменты разработчика) и вкладку Network (Сеть) для записи запросов и ответов.
6. Перейдите по адресу https://localhost:5002/weatherforecast.
7. Обратите внимание на добавленный настраиваемый заголовок ответа HTTP `super-secure` (рис. 16.19).

Рис. 16.19. Добавление настраиваемого HTTP-заголовка super-secure

Создание веб-сервисов с помощью минимальных API

В .NET 6 компания Microsoft приложила много усилий по добавлению новых возможностей в язык C#10 и упрощению библиотек ASP.NET Core, чтобы обеспечить возможность создания веб-сервисов с помощью минимальных API.

Возможно, вы помните сервис прогноза погоды, который предоставляется в шаблоне проекта Web API. В нем продемонстрировано использование класса контроллера для возврата прогноза погоды на пять дней с помощью поддельных данных. Сейчас мы создадим такой же сервис погоды, используя минимальные API.

Для начала в сервисе погоды есть класс для представления единого прогноза погоды. Он понадобится нам в нескольких проектах, поэтому создадим библиотеку классов.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: Class library/`classlib`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.Common`.
2. Переименуйте файл `Class1.cs` в `WeatherForecast.cs`.
3. Измените файл `WeatherForecast.cs`:

```
namespace Northwind.Common
{
  public class WeatherForecast
  {
    public static readonly string[] Summaries = new[]
    {
      "Freezing", "Bracing", "Chilly", "Cool", "Mild",
      "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };

    public DateTime Date { get; set; }

    public int TemperatureC { get; set; }

    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

    public string? Summary { get; set; }
  }
}
```

Создание сервиса погоды с помощью минимальных API

Теперь воссоздадим этот сервис погоды, используя минимальные API. Он будет прослушивать порт `5003` и поддерживать CORS, чтобы запросы могли поступать только с сайта MVC и допускались только запросы `GET`.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: ASP.NET Core Empty/`web`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Minimal.WebApi`;
   4) другие параметры Visual Studio: Authentication Type (Тип аутентификации) — None (Нет), флажок Configure for HTTPS (Настроить для HTTPS) установлен, флажок Enable Docker (Включить Docker) снят, флажок Enable OpenAPI support (Включить поддержку OpenAPI) установлен.
2. В программе Visual Studio Code выберите `Minimal.WebApi` в качестве активного проекта OmniSharp.
3. В проекте `Minimal.WebApi` добавьте ссылку на проект `Northwind.Common`:

```
<ItemGroup>
  <ProjectReference Include="..\Northwind.Common\Northwind.Common.csproj"
/>
</ItemGroup>
```

4. Соберите проект `Minimal.WebApi`.
5. Измените содержимое файла `Program.cs`, как показано ниже:

```
using Northwind.Common; // WeatherForecast

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseUrls("https://localhost:5003");

builder.Services.AddCors();

var app = builder.Build();

// допускаются только клиент MVC и запросы GET
app.UseCors(configurePolicy: options =>
{
  options.WithMethods("GET");
  options.WithOrigins("https://localhost:5001");
});
```

```
app.MapGet("/api/weather", () =>
{
  return Enumerable.Range(1, 5).Select(index =>
    new WeatherForecast
    {
      Date = DateTime.Now.AddDays(index),
      TemperatureC = Random.Shared.Next(-20, 55),
      Summary = WeatherForecast.Summaries[
        Random.Shared.Next(WeatherForecast.Summaries.Length)]
    })
    .ToArray();
});

app.Run();
```

При разработке простых веб-сервисов избегайте создания класса контроллера и вместо этого используйте минимальные API, чтобы поместить всю конфигурацию и реализацию в одном месте — файле Program.cs.

6. В папке Properties измените файл `launchSettings.json`, чтобы настроить профиль `Minimal.WebApi` на запуск браузера с помощью порта `5003` в URL, как показано ниже:

```
"profiles": {
  "Minimal.WebApi": {
    "commandName": "Project",
    "dotnetRunMessages": "true",
    "launchBrowser": true,
    "applicationUrl": "https://localhost:5003",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
```

Тестирование минимального сервиса погоды

Прежде чем создавать клиента к сервису, проверим, что он возвращает прогнозы в формате JSON.

1. Запустите проект веб-сервиса.
2. Если вы не пользуетесь программой Visual Studio 2022, то запустите браузер Google Chrome и перейдите по адресу https://localhost:5003/api/weather.
3. Обратите внимание, что сервис Web API возвращает документ JSON с пятью случайными объектами прогноза погоды в массиве.
4. Закройте браузер Google Chrome и завершите работу веб-сервера.

Добавление прогнозов погоды на главную страницу сайта Northwind

Наконец, добавим HTTP-клиент на сайт Northwind, чтобы он мог вызывать сервис погоды и показывать прогнозы на главной странице.

1. В проекте Northwind.Mvc добавьте ссылку проекта на `Northwind.Common`, как показано ниже:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include="..\Northwind.Common.DataContext.Sqlite\
  Northwind.Common.DataContext.Sqlite.csproj" />
  <ProjectReference Include="..\Northwind.Common\Northwind.Common.csproj"
/>
</ItemGroup>
```

2. В файле `Program.cs` добавьте оператор для настройки HTTP-клиента на вызов минимального сервиса на порте `5003`:

```
builder.Services.AddHttpClient(name: "Minimal.WebApi",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5003/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

3. В файле `HomeController.cs` импортируйте пространство имен `Northwind.Common`, а в методе `Index` добавьте операторы для вызова сервиса погоды для получения прогнозов и сохранения их в словаре `ViewData` с помощью HTTP-клиента:

```
try
{
  HttpClient client = clientFactory.CreateClient(
    name: "Minimal.WebApi");

  HttpRequestMessage request = new(
    method: HttpMethod.Get, requestUri: "api/weather");

  HttpResponseMessage response = await client.SendAsync(request);

  ViewData["weather"] = await response.Content
    .ReadFromJsonAsync<WeatherForecast[]>();
}
catch (Exception ex)
{
  _logger.LogWarning($"The Minimal.WebApi service is not responding.
  Exception: {ex.Message}");
  ViewData["weather"] = Enumerable.Empty<WeatherForecast>().ToArray();
}
```

4. В папке Views/Home в файле `Index.cshtml` импортируйте пространство имен `Northwind.Common`, а затем в верхнем блоке кода извлеките прогнозы погоды из словаря `ViewData`, как показано ниже:

```
@{
  ViewData["Title"] = "Home Page";
  string currentItem = "";
  WeatherForecast[]? weather = ViewData["weather"] as WeatherForecast[];
}
```

5. В первом контейнере `<div>` после визуализации текущего времени добавьте разметку для перечисления прогнозов погоды, если их нет, и отображения их в виде таблицы:

```
<p>
  <h4>Five-Day Weather Forecast</h4>
  @if ((weather is null) || (!weather.Any()))
  {
    <p>No weather forecasts found.</p>
  }
  else
  {
  <table class="table table-info">
    <tr>
      @foreach (WeatherForecast w in weather)
      {
        <td>@w.Date.ToString("ddd d MMM") will be @w.Summary</td>
      }
    </tr>
  </table>
  }
</p>
```

6. Запустите сервис `Minimal.WebApi`.
7. Запустите сайт `Northwind.Mvc`.
8. Перейдите по адресу https://localhost:5001/ и обратите внимание на прогноз погоды (рис. 16.20).

Рис. 16.20. Пятидневный прогноз погоды на главной странице сайта Northwind

9. Проанализируйте командную строку или терминал для сайта MVC и обратите внимание на информационные сообщения, которые показывают, что запрос был отправлен в конечную точку минимального веб-сервиса API `api/weather` примерно за 83 мс:

```
info: System.NET.Http.HttpClient.Minimal.WebApi.LogicalHandler[100]
      Start processing HTTP request GET https://localhost:5003/api/weather
info: System.NET.Http.HttpClient.Minimal.WebApi.ClientHandler[100]
      Sending HTTP request GET https://localhost:5003/api/weather
info: System.NET.Http.HttpClient.Minimal.WebApi.ClientHandler[101]
      Received HTTP response headers after 76.8963ms - 200
info: System.NET.Http.HttpClient.Minimal.WebApi.LogicalHandler[101]
      End processing HTTP request after 82.9515ms – 200
```

10. Остановите сервис `Minimal.WebApi`, обновите браузер и обратите внимание, что через несколько секунд появляется главная страница сайта MVC без прогноза погоды.
11. Закройте браузер Google Chrome и завершите работу веб-сервера.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 16.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. От какого класса вы должны наследовать, чтобы создать класс контроллера для сервиса ASP.NET Core Web API?
2. Если вы дополнили свой класс `Controller` атрибутом `[ApiController]`, чтобы получить поведение по умолчанию, например автоматический ответ со статусом `400` для недопустимых моделей, что еще должны сделать?
3. Что вы должны сделать, чтобы указать, какой метод действия контроллера будет выполняться в ответ на HTTP-запрос?
4. Что вы должны сделать, чтобы указать, какие ответы следует ожидать при вызове метода действия?
5. Какие методы можно вызывать для возврата ответов с разными кодами состояния?
6. Каковы четыре способа тестирования веб-сервиса?

7. Почему вам не следует оборачивать класс `HttpClient` в оператор `using`, чтобы очистить его ресурсы, когда вы закончите с ним работать, даже если он реализует интерфейс `IDisposable`, и что вы должны использовать вместо этого?
8. Что такое CORS и почему важно включить ее в веб-сервис?
9. Как вы можете разрешить клиентам определять работоспособность вашего веб-сервиса в ASP.NET Core 2.2 и более поздних версиях?
10. Какие преимущества обеспечивает маршрутизация конечных точек?

Упражнение 16.2. Создание и удаление клиентов с помощью класса HttpClient

Дополните проект сайта `Northwind.Mvc`, добавив страницы, на которых у пользователя появится возможность заполнить форму для создания нового клиента или выполнить поиск клиента, а затем удалить его. Контроллер MVC должен вызывать веб-сервис Northwind для создания и удаления клиентов.

Упражнение 16.3. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-16---building-and-consuming-web-services, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали, как создать сервис ASP.NET Core Web API, пригодный для вызова любым приложением на любой платформе, которая может отправлять HTTP-запрос и обрабатывать HTTP-ответ. Вы также научились тестировать и документировать API веб-сервисов с помощью Swagger, а также эффективно использовать сервисы.

В следующей главе вы научитесь создавать пользовательские интерфейсы с помощью Blazor, новой компонентной технологии Microsoft, которая позволяет разработчикам создавать клиентские одностраничные приложения (single-page applications, SPA) для сайтов, используя C# вместо JavaScript, гибридные приложения для компьютеров и потенциально мобильные приложения.

17 Создание пользовательских интерфейсов с помощью Blazor

В этой главе рассказывается о том, как с помощью Blazor создавать пользовательские интерфейсы. Я расскажу, что такое Blazor, опишу его преимущества и недостатки.

Вы узнаете, как создавать компоненты Blazor, которые могут выполнять свой код на веб-сервере или в браузере. При размещении веб-интерфейсов на Blazor Server необходимые обновления пользовательского интерфейса в браузере обеспечиваются с помощью SignalR. При размещении с Blazor WebAssembly компоненты выполняют свой код на стороне клиента и для взаимодействия с сервером должны выполнять HTTP-вызовы.

В этой главе:

- знакомство с Blazor;
- сравнение шаблонов проектов Blazor;
- сборка компонентов с помощью Blazor Server;
- абстрагирование сервиса для компонента Blazor;
- сборка компонентов с использованием Blazor WebAssembly;
- улучшение приложений Blazor WebAssembly.

Знакомство с Blazor

Технология Blazor позволяет создавать общие компоненты и интерактивные пользовательские веб-интерфейсы на языке C# вместо JavaScript. В апреле 2019 года компания Microsoft объявила, что Blazor «больше не считается экспериментальной разработкой и реализуется в качестве поддерживаемой платформы веб-интерфейса, включая запуск на стороне клиента в браузере на WebAssembly». Blazor поддерживается всеми современными браузерами.

Недостатки JavaScript

Традиционно любой код, который должен выполняться в веб-браузере, пишется на JavaScript или на языке программирования более высокого уровня, который *транспилируется* (преобразуется или компилируется) в JavaScript. Это связано с тем, что все браузеры поддерживают JavaScript уже около двух десятилетий, поэтому данный язык стал де-факто стандартом для реализации бизнес-логики на стороне клиента.

Однако у JavaScript есть некоторые проблемы. Внешне он похож на языки в стиле C, такие как C# и Java, однако в действительности, если углубиться в детали, сильно отличается от них. Это динамически типизированный псевдофункциональный язык, который использует прототипы вместо наследования классов для многократного применения объектов. Как если бы вы думали, что перед вами человек, когда на самом деле это инопланетянин.

Разве не было бы замечательно, если бы мы могли использовать в браузере тот же язык и библиотеки, что и на стороне сервера?

Silverlight: C# и .NET на основе плагина

Компания Microsoft уже пыталась достичь этой цели с помощью технологии Silverlight. Когда в 2008 году увидел свет релиз Silverlight 2.0, разработчики на C# и .NET смогли создавать библиотеки и компоненты визуализации, которые выполнялись в браузере с помощью плагина Silverlight.

К 2011 году и выходу Silverlight 5.0 успешные продажи iPhone от Apple и ненависть Стива Джобса к плагинам для браузеров, таким как Flash, в конечном итоге привели к тому, что Microsoft отказалась от технологии Silverlight, поскольку, как и Flash, Silverlight блокировалась на iPhone и iPad.

WebAssembly: цель для Blazor

Новейшие достижения в разработке браузеров позволили Microsoft предпринять еще одну попытку. В 2017 году было принято *согласованное решение по выпуску WebAssembly*, и теперь его поддерживают все основные браузеры: Chromium (Chrome, Edge, Opera, Brave), Firefox и WebKit (Safari). Blazor не поддерживается программой Internet Explorer компании Microsoft, поскольку это устаревший браузер.

WebAssembly (Wasm) представляет собой формат бинарных инструкций для виртуальной машины, который позволяет запускать в Интернете код, написанный на нескольких языках, без потерь в скорости исполнения. Низкоуровневый язык Wasm разработан как портативная цель для компиляции языков высокого уровня, таких как C#.

Модели хостинга Blazor

Blazor — это единая модель программирования или приложения, имеющая несколько моделей хостинга.

- *Blazor Server* работает на стороне сервера, поэтому написанный вами код C# имеет полный доступ ко всем ресурсам, которые могут понадобиться вашей бизнес-логике, не прибегая к аутентификации. Затем Blazor Server использует SignalR для передачи обновлений пользовательского интерфейса клиентской стороне.
- Сервер должен поддерживать активное соединение SignalR с каждым клиентом и отслеживать текущее состояние каждого из них, поэтому Blazor Server плохо масштабируется при поддержке большого количества клиентов. Впервые он был выпущен в составе ASP.NET Core 3.0 в сентябре 2019 года и включен в .NET 5.0 и более поздние версии.
- *Blazor WebAssembly* работает на стороне клиента, поэтому ваш код C# имеет доступ только к ресурсам в браузере и должен выполнять HTTP-вызовы (которые могут потребовать аутентификации), прежде чем сможет получить доступ к ресурсам на сервере. Впервые он был выпущен в мае 2020 года как расширение для ASP.NET Core 3.1 в версии 3.2, поскольку это текущий выпуск и, следовательно, для него не предусмотрена долгосрочная поддержка ASP.NET Core 3.1. Версия Blazor WebAssembly 3.2 использовала среду выполнения Mono и библиотеки Mono; платформа .NET 5 и более поздние версии используют среду выполнения Mono и библиотеки .NET 5. Как сказал Дэниел Рот, *«Blazor WebAssembly работает на интерпретаторе .NET IL без JIT, поэтому он не дает преимуществ в скорости. Мы добились значительных улучшений скорости в .NET 5 и надеемся улучшить ситуацию в .NET 6»*.
- *Приложение .NET MAUI Blazor App*, оно же *Blazor Hybrid*, запускается в процессе .NET, отображает свой веб-интерфейс на элемент управления веб-представлением с помощью локального канала взаимодействия и размещается в приложении .NET MAUI. Концептуально это похоже на приложения Electron, которые используют Node.js. Мы рассмотрим эту модель хостинга в главе 19.

Эта многохостовая модель означает, что при тщательном планировании разработчик может написать компоненты Blazor один раз, а затем запустить их на стороне веб-сервера, веб-клиента или в настольном приложении.

Blazor Server поддерживается в Internet Explorer 11, а Blazor WebAssembly — нет.

Blazor WebAssembly имеет дополнительную поддержку для прогрессивных веб-приложений (Progressive Web Apps, PWA); это значит, пользователь сайта может использовать меню браузера, чтобы добавить приложение на свой рабочий стол и запустить в режиме офлайн.

Компоненты Blazor

Важно понимать, что Blazor используется для создания компонентов пользовательского интерфейса. Компоненты определяют, как визуализировать UI, реагировать на пользовательские события, и могут быть составлены, вложены и скомпилированы в библиотеку классов NuGet Razor для упаковки и распространения.

Например, вы можете создать компонент `Rating.razor`, как показано ниже:

```
<div>
@for (int i = 0; i < Maximum; i++)
{
  if (i < Value)
  {
    <span class="oi oi-star-filled" />
  }
  else
  {
    <span class="oi oi-star-empty" />
  }
}
</div>

@code {
  [Parameter]
  public byte Maximum { get; set; }

  [Parameter]
  public byte Value { get; set; }
}
```

Вместо одного файла с разметкой и блоком @code код можно хранить в отдельном файле кода программной части Rating.razor.cs. Класс в этом файле должен сопровождаться ключевым словом partial и иметь то же имя, что и компонент.

Затем вы можете использовать компонент на веб-странице следующим образом:

```
<h1>Review</h1>
<Rating id="rating" Maximum="5" Value="3" />
<textarea id="comment" />
```

Существует множество встроенных компонентов Blazor, включая компоненты для установки таких элементов, как `<title>` в разделе `<head>` веб-страницы, а также множество универсальных компонентов сторонних разработчиков.

В будущем Blazor может не ограничиваться только созданием компонентов пользовательского интерфейса с использованием веб-технологий. У Microsoft есть

экспериментальная технология, известная как *Blazor Mobile Bindings*, позволяющая разработчикам с помощью Blazor создавать компоненты мобильного UI. Вместо того чтобы использовать HTML и CSS для создания пользовательского веб-интерфейса, Blazor с помощью XAML и .NET MAUI создает кросс-платформенный графический пользовательский интерфейс.

Различия между Blazor и Razor

Вы можете задаться вопросом, почему компоненты Blazor используют файловое расширение `.razor`. Razor — это синтаксис разметки шаблонов, позволяющий одновременно использовать HTML и C#. В более старых технологиях, поддерживающих синтаксис Razor, расширение файла `.cshtml` используется для обозначения сочетания C# и HTML.

Синтаксис Razor используется для:

- *представлений* ASP.NET Core MVC и *частичных представлений* с расширением `.cshtml`. Бизнес-логика может быть абстрагирована в класс контроллера, рассматривающий представление в качестве шаблона для отправки модели представления с последующим выводом его на веб-страницу;
- *страниц Razor* с расширением `.cshtml`. Бизнес-логика может быть встроенной или содержать файл с расширением `.cshtml.cs`. Результат представляет собой веб-страницу;
- *компонентов Blazor* с расширением `.razor`. Результатом вывода не является веб-страница, хотя шаблоны могут использоваться для обертывания компонента, чтобы он отображался как веб-страница, а директива `@page` будет использоваться для назначения пути, определяющего URL для извлечения компонента как страницы.

Сравнение шаблонов проектов Blazor

Один из способов понять, какую модель размещения выбрать: Blazor Server или Blazor WebAssembly, — проанализировать различия в их шаблонах проектов по умолчанию.

Обзор шаблона проекта Blazor Server

Рассмотрим шаблон по умолчанию для проекта Blazor Server. Вы заметите, что он представляет собой то же самое, что и шаблон ASP.NET Core Razor Pages, но имеет несколько ключевых дополнений.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: Blazor Server App/`blazorserver`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.BlazorServer`;
   4) другие параметры Visual Studio: Authentication Type (Тип аутентификации) — None (Нет), флажок Configure for HTTPS (Настроить для HTTPS) установлен, флажок Enable Docker (Включить Docker) снят.
2. В программе Visual Studio Code выберите `Northwind.BlazorServer` в качестве активного проекта OmniSharp.
3. Соберите проект `Northwind.BlazorServer`.
4. В проекте/папке `Northwind.BlazorServer` откройте файл `Northwind.BlazorServer.csproj` и обратите внимание, что он идентичен проекту ASP.NET Core, который использует веб-пакет SDK и ориентирован на платформу .NET 6.0.
5. Откройте файл `Program.cs` и обратите внимание, что он идентичен проекту ASP.NET Core. Различия включают раздел, в котором настраиваются сервисы, содержащие вызов метода `AddServerSideBlazor`, как показано ниже (выделено жирным шрифтом):

```
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();
```

6. Обратите также внимание на раздел конфигурации HTTP-конвейера, включающий вызовы методов `MapBlazorHub` и `MapFallbackToPage`, которые настраивают приложение ASP.NET Core для приема входящих подключений SignalR для компонентов Blazor, в то время как другие запросы возвращаются к странице Razor `_Host.cshtml`, как показано ниже:

```
app.UseRouting();

app.MapBlazorHub();
app.MapFallbackToPage("/_Host");

app.Run();
```

7. В папке Pages откройте файл `_Host.cshtml` и обратите внимание, что он устанавливает общий макет `_Layout` и отображает компонент Blazor типа `App`, который предварительно визуализируется на сервере:

```
@page "/"
@namespace Northwind.BlazorServer.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
```

```
  Layout = "_Layout";
}

<component type="typeof(App)" render-mode="ServerPrerendered" />
```

8. В папке Pages откройте файл общего макета `_Layout.cshtml`, содержимое которого показано ниже:

```
@using Microsoft.AspNetCore.Components.Web
@namespace Northwind.BlazorServer.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport"
        content="width=device-width, initial-scale=1.0" />
  <base href="~/" />
  <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
  <link href="css/site.css" rel="stylesheet" />
  <link href="Northwind.BlazorServer.styles.css" rel="stylesheet" />
  <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
</head>
<body>
  @RenderBody()

  <div id="blazor-error-ui">
    <environment include="Staging,Production">
      An error has occurred. This application may no longer respond
      until reloaded.
    </environment>
    <environment include="Development">
      An unhandled exception has occurred. See browser dev tools
      for details.
    </environment>
    <a href="" class="reload">Reload</a>
    <a class="dismiss">×< /a>
  </div>

  <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

При анализе этой разметки обратите внимание на следующие моменты:

- код `<div id="blazor-error-ui">` позволяет показать ошибки Blazor, которые в случае возникновения отображаются в виде желтой полосы внизу веб-страницы;

- блок сценария для `blazor.server.js` управляет обратным подключением SignalR к серверу.

9. В папке Northwind.BlazorServer откройте файл `App.razor`. Обратите внимание, что файл определяет компонент `Router` для всех компонентов, найденных в текущей сборке:

```
<Router AppAssembly="@typeof(App).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData"
               DefaultLayout="@typeof(MainLayout)" />
    <FocusOnNavigate RouteData="@routeData" Selector="h1" />
  </Found>
  <NotFound>
    <PageTitle>Not found</PageTitle>
    <LayoutView Layout="@typeof(MainLayout)">
      <p>Sorry, there's nothing at this address.</p>
    </LayoutView>
  </NotFound>
</Router>
```

При анализе этой разметки обратите внимание на следующие моменты:

- если соответствующий путь найден, то выполняется метод `RouteView`, который устанавливает макет по умолчанию для компонента в `MainLayout` и передает любые параметры данных пути компоненту;
- если соответствующий путь не найден, то выполняется метод `LayoutView`, который отображает внутреннюю разметку (в данном случае простой элемент абзаца с сообщением, что по данному адресу ничего нет) внутри `MainLayout`.

10. В папке Shared откройте файл `MainLayout.razor` и обратите внимание, что он определяет тег `<div>` для боковой панели, содержащей меню навигации, который реализован в файле компонента `NavMenu.razor` в этом проекте, и HTML5-элемент, такой как `<main>` и `<article>` для содержимого:

```
@inherits LayoutComponentBase

<PageTitle>Northwind.BlazorServer</PageTitle>

<div class="page">
  <div class="sidebar">
    <NavMenu />
  </div>

  <main>
    <div class="top-row px-4">
```

```
      <a href="https://docs.microsoft.com/aspnet/"
         target="_blank">About</a>
    </div>
    <article class="content px-4">
      @Body
    </article>
  </main>
</div>
```

11. В папке Shared откройте файл `MainLayout.razor.css`. Обратите внимание, что он содержит изолированные стили CSS для каждого компонента.
12. В папке Shared откройте файл `NavMenu.razor`. Обратите внимание, что он содержит три пункта меню: `Home`, `Counter` и `Fetch data`. Они создаются с помощью компонента Microsoft Blazor под названием `NavLink`:

```
<div class="top-row ps-3 navbar navbar-dark">
  <div class="container-fluid">
    <a class="navbar-brand" href="">Northwind.BlazorServer</a>
    <button title="Navigation menu" class="navbar-toggler"
            @onclick="ToggleNavMenu">
      <span class="navbar-toggler-icon"></span>
    </button>
  </div>
</div>

<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
  <nav class="flex-column">
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
        <span class="oi oi-home" aria-hidden="true"></span> Home
      </NavLink>
    </div>
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="counter">
        <span class="oi oi-plus" aria-hidden="true"></span> Counter
      </NavLink>
    </div>
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="fetchdata">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data
      </NavLink>
    </div>
  </nav>
</div>

@code {
  private bool collapseNavMenu = true;
```

```
  private string? NavMenuCssClass => collapseNavMenu ? "collapse" : null;

  private void ToggleNavMenu()
  {
    collapseNavMenu = !collapseNavMenu;
  }
}
```

13. В папке Pages откройте файл `FetchData.razor`. Обратите внимание, что он определяет компонент, который извлекает прогноз погоды из внедренного зависимого сервиса, а затем отображает данные прогноза погоды в таблице:

```
@page "/fetchdata"

<PageTitle>Weather forecast</PageTitle>

@using Northwind.BlazorServer.Data
@inject WeatherForecastService ForecastService

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
  <p><em>Loading...</em></p>
}
else
{
  <table class="table">
    <thead>
      <tr>
        <th>Date</th>
        <th>Temp. (C)</th>
        <th>Temp. (F)</th>
        <th>Summary</th>
      </tr>
    </thead>
  <tbody>
  @foreach (var forecast in forecasts)
    {
      <tr>
        <td>@forecast.Date.ToShortDateString()</td>
        <td>@forecast.TemperatureC</td>
        <td>@forecast.TemperatureF</td>
        <td>@forecast.Summary</td>
      </tr>
    }
    </tbody>
```

```
  </table>
}

@code {
  private WeatherForecast[]? forecasts;

  protected override async Task OnInitializedAsync()
  {
    forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
  }
}
```

14. В папке Data откройте файл `WeatherForecastService.cs`. Обратите внимание, что это не класс контроллера Web API, а просто обычный класс, возвращающий случайные данные о погоде:

```
namespace Northwind.BlazorServer.Data
{
  public class WeatherForecastService
  {
    private static readonly string[] Summaries = new[]
    {
      "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm",
      "Balmy", "Hot", "Sweltering", "Scorching"
    };

    public Task<WeatherForecast[]> GetForecastAsync(DateTime startDate)
    {
      return Task.FromResult(Enumerable.Range(1, 5)
        .Select(index => new WeatherForecast
          {
            Date = startDate.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
          }).ToArray());
    }
  }
}
```

Изоляция CSS и JavaScript

Компонентам Blazor часто нужен отдельный код CSS, чтобы они могли применять стилизацию, или JavaScript для действий, которые не могут быть выполнены на чистом языке C#, например, для доступа к API браузера. Чтобы обеспечить отсутствие конфликтов с CSS и JavaScript на уровне сайта, Blazor поддерживает изоляцию CSS и JavaScript. Если у вас есть компонент `Index.razor`, то создайте файл CSS `Index.razor.css`. Стили, определенные в этом файле, будут переопределять все остальные стили в проекте.

Маршрутизация Blazor по компонентам страницы

Компонент `Router`, который мы видели в файле `App.razor`, обеспечивает маршрутизацию по компонентам. Разметка для создания экземпляра компонента выглядит как HTML-тег, где имя тега является типом компонента. Компоненты могут быть встроены в веб-страницу с помощью элемента, например, `<Rating Stars="5" />` или маршрутизированы, как страница Razor или контроллер MVC.

Как определить маршрутизируемый компонент страницы

Чтобы создать маршрутизируемый компонент страницы, добавьте директиву `@page` в начало файла компонента `.razor`, как показано ниже:

```
@page "customers"
```

Предыдущий код является эквивалентом контроллера MVC, дополненного атрибутом `[Route]`:

```
[Route("customers")]
public class CustomersController
{
```

Компонент `Router` сканирует сборку, указанную в параметре `AppAssembly`, на наличие компонентов, дополненных атрибутом `[Route]`, и регистрирует их URL.

Любой одностраничный компонент может иметь несколько директив `@page` для регистрации нескольких маршрутов.

Во время выполнения компонент страницы объединяется с любым указанным вами макетом точно так же, как это было бы с представлением MVC или страницей Razor. По умолчанию шаблон проекта Blazor Server определяет `MainLayout.razor` в качестве макета для компонентов страницы.

По соглашению компоненты маршрутизируемых страниц следует помещать в папку Pages.

Навигация по маршрутам Blazor

Компания Microsoft предоставляет сервис зависимостей `NavigationManager`, который понимает маршрутизацию Blazor и компонент `NavLink`.

Метод `NavigateTo` используется для перехода к указанному URL.

Передача параметров маршрутизации

Маршруты Blazor могут включать именованные параметры, не чувствительные к регистру, и ваш код может легко получить доступ к переданным значениям, привязав параметр к свойству в блоке кода с помощью атрибута `[Parameter]`, как показано ниже:

```
@page "/customers/{country}"

<div>Country parameter as the value: @Country</div>

@code {
  [Parameter]
  public string Country { get; set; }
}
```

Рекомендуемый способ обработки параметра, который должен иметь значение по умолчанию, если отсутствует, — это добавить параметр с помощью символа `?` и использовать оператор объединения с неопределенным значением в методе `OnParametersSet`, как показано ниже:

```
@page "/customers/{country?}"

<div>Country parameter as the value: @Country</div>

@code {
  [Parameter]
  public string Country { get; set; }

  protected override void OnParametersSet()
  {
    // если автоматически установленное свойство имеет значение null,
    // то укажите значение USA
    Country = Country ?? "USA";
  }
}
```

Базовые классы компонентов

Метод `OnParametersSet` определяется базовым классом `ComponentBase`, от которого наследуются компоненты по умолчанию:

```
using Microsoft.AspNetCore.Components;

public abstract class ComponentBase : IComponent, IHandleAfterRender,
IHandleEvent
{
  // члены не показаны
}
```

Класс `ComponentBase` имеет несколько полезных методов, которые вы можете вызывать и переопределять (табл. 17.1).

Таблица 17.1. Методы класса ComponentBase

Метод (-ы)	Описание
`InvokeAsync`	Вызовите этот метод для выполнения функции в контексте синхронизации ассоциируемого средства визуализации
`OnAfterRender`, `OnAfterRenderAsync`	Переопределите эти методы, чтобы вызывать код после каждой визуализации компонента
`OnInitialized`, `OnInitializedAsync`	Переопределите эти методы, чтобы вызвать код после того, как компонент получил свои начальные параметры от своего родительского объекта в дереве визуализации
`OnParametersSet`, `OnParametersSetAsync`	Переопределите эти методы, чтобы вызвать код после того, как компонент получил параметры и значения были присвоены свойствам
`ShouldRender`	Переопределите этот метод, чтобы указать, должен ли компонент визуализироваться
`StateHasChanged`	Вызовите этот метод, чтобы вызвать повторную визуализацию компонента

Компоненты Blazor могут иметь общие макеты аналогично представлениям MVC и страницам Razor.

Создайте файл компонента `.razor`, но сделайте его явным наследником от `LayoutComponentBase`, как показано ниже:

```
@inherits LayoutComponentBase

<div>
  ...
  @Body
  ...
</div>
```

Базовый класс имеет свойство `Body`, которое вы можете отобразить в коде в нужном месте макета.

Установите макет по умолчанию для компонентов в файле `App.razor` и его компоненте `Router`. Чтобы явно задать макет для компонента, используйте директиву `@layout`, как показано ниже:

```
@page "/customers"
@layout AlternativeLayout

<div>
  ...
</div>
```

Как использовать компонент навигационной ссылки с маршрутами

В HTML для определения навигационных ссылок используется элемент `<a>`:

```
<a href="/customers">Customers</a>
```

В Blazor используйте компонент `<NavLink>`, как показано ниже:

```
<NavLink href="/customers">Customers</NavLink>
```

Компонент `NavLink` лучше, чем элемент привязки, поскольку автоматически устанавливает свой класс как `active`, если его `href` совпадает с URL текущего местоположения. Если в вашем CSS используется другое имя класса, то вы можете задать имя класса в свойстве `NavLink.ActiveClass`.

По умолчанию в алгоритме сопоставления `href` является *префиксом* пути, поэтому если у компонента `NavLink` `href` имеет значение `/customers`, как показано в предыдущем примере кода, то он будет сопоставлять все следующие пути и устанавливать для всех них стиль класса `active`:

```
/customers
/customers/USA
/customers/Germany/Berlin
```

Чтобы убедиться, что алгоритм сопоставления выполняет сопоставление по *всем* путям, присвойте параметру `Match` значение `NavLinkMatch.All`:

```
<NavLink href="/customers" Match="NavLinkMatch.All">Customers</NavLink>
```

Если вы зададите другие атрибуты, такие как `target`, то они будут переданы в создаваемый элемент `<a>`.

Запуск шаблона проекта Blazor Server

Теперь, когда мы рассмотрели шаблон проекта и его важные компоненты, характерные для Blazor Server, мы можем запустить сайт и проанализировать результаты.

1. В папке Properties откройте файл `launchSettings.json`.
2. Измените значение свойства `applicationUrl`, указав порт `5000` для HTTP и порт `5001` для HTTPS:

```
"profiles": {
  "Northwind.BlazorServer": {
    "commandName": "Project",
```

```
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
```

3. Запустите сайт.
4. Запустите браузер Google Chrome.
5. Перейдите по адресу https://localhost:5001/.
6. В левом навигационном меню выберите пункт Fetch data (Получить данные), как показано на рис. 17.1.

Рис. 17.1. Получение данных о погоде в приложении Blazor Server

7. В адресной строке браузера измените путь маршрута на `/apples` и обратите внимание на сообщение об отсутствии компонента (рис. 17.2).
8. Закройте браузер Google Chrome и завершите работу веб-сервера.

Рис. 17.2. Сообщение об отсутствии компонента

Обзор шаблона проекта Blazor WebAssembly

Создадим проект Blazor WebAssembly. Я не буду показывать в книге код, если он такой же, как в проекте Blazor Server.

1. Откройте редактор кода и создайте в рабочей области/решении `PracticalApps` проект с такими настройками:
   1) шаблон проекта: Blazor WebAssembly App/`blazorwasm`;
   2) переключатели: `--pwa --hosted`;
   3) файл и папка рабочей области/решения: `PracticalApps`;
   4) файл и папка проекта: `Northwind.BlazorWasm`;
   5) параметр Authentication Type: None (Тип аутентификации: Нет);
   6) флажок Configure for HTTPS: (Настроить для HTTPS:) установлен;
   7) флажок ASP.NET Core hosted: (ASP.NET Core размещен:) установлен;
   8) флажок Progressive Web Application: (Прогрессивное веб-приложение:) установлен.

   При анализе созданных папок и файлов обратите внимание, что создаются три проекта:

   - `Northwind.BlazorWasm.Client` — проект Blazor WebAssembly в папке Northwind.BlazorWasm\Client;
   - `Northwind.BlazorWasm.Server` — сайт проекта ASP.NET Core в папке Northwind.BlazorWasm\Server для размещения сервиса погоды. Этот сайт содержит такую же реализацию для возврата случайных данных прогноза погоды, что и ранее, однако реализован как соответствующий класс контроллера Web API. Файл проекта содержит ссылки на проект `Shared` и `Client`, а также ссылку на пакет для поддержки Blazor WebAssembly на стороне сервера;
   - `Northwind.BlazorWasm.Shared` — библиотека классов в папке Northwind.BlazorWasm\Shared, содержащая модели для сервиса погоды.

   Структура папок упрощается, как показано на рис. 17.3.

Рис. 17.3. Структура папок для шаблона проекта Blazor WebAssembly

Существует два способа развертывания приложения Blazor WebAssembly. Вы можете развернуть только клиентский проект, разместив опубликованные файлы на любом статическом хостинге веб-сервера. Его можно настроить на вызов сервиса погоды, который вы создали в главе 16, или вы можете развернуть серверный проект, который ссылается на клиентское приложение и содержит как сервис погоды, так и приложение Blazor WebAssembly. Приложение размещается в папке wwwroot сайта сервера вместе с другими статическими ресурсами. Подробнее об этих вариантах можно прочитать здесь: https://docs.microsoft.com/ru-ru/aspnet/core/blazor/host-and-deploy/webassembly.

2. В папке Client откройте файл `Northwind.BlazorWasm.Client.csproj`. Обратите внимание, что файл использует пакет SDK Blazor WebAssembly и ссылается на два пакета WebAssembly и проект `Shared`, а также на исполнителя сервиса, необходимого для поддержки PWA:

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <ServiceWorkerAssetsManifest>service-worker-assets.js
        </ServiceWorkerAssetsManifest>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include=
      "Microsoft.AspNetCore.Components.WebAssembly"
      Version="6.0.0" />
    <PackageReference Include=
      "Microsoft.AspNetCore.Components.WebAssembly.DevServer"
      Version="6.0.0" PrivateAssets="all" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include=
      "..\Shared\Northwind.BlazorWasm.Shared.csproj" />
  </ItemGroup>

  <ItemGroup>
    <ServiceWorker Include="wwwroot\service-worker.js"
      PublishedContent="wwwroot\service-worker.published.js" />
  </ItemGroup>

</Project>
```

3. В папке Client откройте файл `Program.cs`. Обратите внимание, что разработчик хоста предназначен для `WebAssembly`, а не для серверного ASP.NET Core, и он регистрирует сервис зависимостей для выполнения HTTP-запросов, что является чрезвычайно распространенным требованием для приложений Blazor WebAssembly:

```
using Microsoft.AspNetCore.Components.Web;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;
using Northwind.BlazorWasm.Client;

var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");

builder.Services.AddScoped(sp => new HttpClient
  { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });

await builder.Build().RunAsync();
```

4. В папке wwwroot откройте файл `index.html`. Обратите внимание на файлы `manifest.json` и `service-worker.js`, поддерживающие автономную работу, а также на сценарий `blazor.webassembly.js`, который загружает все пакеты NuGet для Blazor WebAssembly:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0,
  maximum-scale=1.0, user-scalable=no" />
  <title>Northwind.BlazorWasm</title>
  <base href="/" />
  <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
  <link href="css/app.css" rel="stylesheet" />
  <link href="Northwind.BlazorWasm.Client.styles.css" rel="stylesheet" />
  <link href="manifest.json" rel="manifest" />
  <link rel="apple-touch-icon" sizes="512x512" href="icon-512.png" />
  <link rel="apple-touch-icon" sizes="192x192" href="icon-192.png" />
</head>

<body>
  <div id="app">Loading…</div>

  <div id="blazor-error-ui">
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
```

```
    <a class="dismiss">×< /a>
  </div>
  <script src="_framework/blazor.webassembly.js"></script>
  <script>navigator.serviceWorker.register('service-worker.js');</script>
</body>

</html>
```

5. Обратите внимание, что следующие файлы `.razor` идентичны файлам в проекте Blazor Server:
   - `App.razor`;
   - `Shared\MainLayout.razor`;
   - `Shared\NavMenu.razor`;
   - `Shared\SurveyPrompt.razor`;
   - `Pages\Counter.razor`;
   - `Pages\Index.razor`.
6. В папке Pages откройте файл `FetchData.razor`. Обратите внимание, что код аналогичен Blazor Server, за исключением внедренного сервиса зависимостей для выполнения HTTP-запросов:

```
@page "/fetchdata"
@using Northwind.BlazorWasm.Shared
@inject HttpClient Http

<h1>Weather forecast</h1>

...

@code {
  private WeatherForecast[]? Forecasts;

  protected override async Task OnInitializedAsync()
  {
    forecasts = await
      Http.GetFromJsonAsync<WeatherForecast[]>("WeatherForecast");
  }
}
```

7. Запустите проект `Northwind.BlazorWasm.Server`.
8. Обратите внимание, что приложение имеет те же функции, что и раньше. Код компонента Blazor выполняется внутри браузера, а не на сервере. Сервис погоды запускается на сервере.
9. Закройте браузер Google Chrome и завершите работу веб-сервера.

Сборка компонентов с помощью Blazor Server

В данном разделе создадим компонент для составления списка, создания и редактирования клиентов в базе данных Northwind. Мы создадим его сначала только для Blazor Server, а затем реорганизуем его для работы как с Blazor Server, так и с Blazor WebAssembly.

Определение и тестирование простого компонента

В существующий проект Blazor Server добавим новый компонент.

1. В проекте `Northwind.BlazorServer` (*не* проект `Northwind.BlazorWasm.Server`) в папке Pages добавьте новый файл `Customers.razor`. В программе Visual Studio элемент называется Razor Component.

Имена файлов компонентов должны начинаться с заглавной буквы. В противном случае появятся ошибки компиляции!

2. Добавьте операторы для вывода заголовка для компонента клиентов Customers и определите блок кода, определяющий свойство для хранения названия страны:

```
<h3>Customers@(string.IsNullOrWhiteSpace(Country) ? " Worldwide" : " in "
+ Country)</h3>

@code {
  [Parameter]
  public string? Country { get; set; }
}
```

3. В папке Pages в компоненте `Index.razor` добавьте в конце файла операторы, чтобы дважды создать компонент `Customers`, один раз передавая значение `Germany` в качестве параметра страны и один раз без указания страны:

```
<Customers Country="Germany" />
<Customers />
```

4. Запустите проект сайта `Northwind.BlazorServer`.
5. Закройте браузер Google Chrome.
6. Перейдите по адресу https://localhost:5001/ и обратите внимание на компоненты `Customers` (рис. 17.4).

Рис. 17.4. Компонент Customers с параметром Country, установленным со значением Germany, и без него

7. Закройте браузер Google Chrome и завершите работу веб-сервера.

Создание маршрутизируемого компонента страницы

Очень просто превратить этот компонент в маршрутизируемый компонент страницы с параметром маршрута для страны.

1. В папке Pages в компоненте `Customers.razor` добавьте в начале файла оператор, регистрирующий `/customers` в качестве маршрута с необязательным параметром маршрута страны, как показано ниже:

```
@page "/customers/{country?}"
```

2. В папке Shared откройте файл `NavMenu.razor` и добавьте два элемента списка для нашего компонента маршрутизируемой страницы, чтобы показать клиентов по всему миру и в Германии, которые используют значок с изображением людей:

```
<div class="nav-item px-3">
  <NavLink class="nav-link" href="customers" Match="NavLinkMatch.All">
    <span class="oi oi-people" aria-hidden="true"></span>
    Customers Worldwide
  </NavLink>
</div>
<div class="nav-item px-3">
  <NavLink class="nav-link" href="customers/Germany">
    <span class="oi oi-people" aria-hidden="true"></span>
    Customers in Germany
  </NavLink>
</div>
```

Для пункта меню «Клиенты» мы использовали значок с изображением людей. Вы можете посмотреть другие доступные значки по следующей ссылке: https://iconify.Design/icon-sets/oi/.

3. Запустите проект сайта.
4. Запустите браузер Google Chrome.
5. Перейдите по адресу https://localhost:5001/.
6. В левом меню навигации щелкните на ссылке Customers in Germany (Клиенты в Германии) и обратите внимание, что название страны правильно передается компоненту страницы и что компонент использует тот же общий макет, что и другие компоненты страницы, например `Index.razor`.
7. Закройте браузер Google Chrome и завершите работу веб-сервера.

Получение сущностей в компоненте

Теперь, когда вы ознакомились с минимальной реализацией компонента, мы можем добавить к нему некоторые полезные функции. В данном случае для извлечения клиентов из базы данных мы будем использовать контекст базы данных Northwind.

1. В файле `Northwind.BlazorServer.csproj` добавьте ссылку на проект контекста базы данных Northwind для SQL Server или SQLite, как показано ниже:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include="..\Northwind.Common.DataContext.Sqlite
  \Northwind.Common.DataContext.Sqlite.csproj" />
</ItemGroup>
```

2. Соберите проект `Northwind.BlazorServer`.
3. В файле `Program.cs` импортируйте пространство имен для работы с контекстом базы данных Northwind:

```
using Packt.Shared; // метод расширения AddNorthwindContext
```

4. В разделе настройки сервисов добавьте оператор для регистрации контекста базы данных Northwind в коллекции сервиса зависимостей:

```
builder.Services.AddNorthwindContext();
```

5. Откройте файл `_Imports.razor` и импортируйте пространства имен для работы с сущностями Northwind, чтобы создаваемым нами компонентам Blazor не нужно было импортировать пространства имен по отдельности:

```
@using Packt.Shared @* Northwind entities *@
```

Файл _Imports.razor влияет только на файлы .razor. Если для реализации компонентного кода вы используете сопутствующие файлы .cs, то в них пространства имен нужно импортировать отдельно или применить глобальное использование для неявного импорта пространства имен.

6. В папке Pages в файле `Customers.razor` добавьте операторы для ввода контекста базы данных Northwind и затем используйте его для вывода таблицы всех клиентов:

```
@using Microsoft.EntityFrameworkCore @* ToListAsync extension method *@
@page "/customers/{country?}"
@inject NorthwindContext db

<h3>Customers @(string.IsNullOrWhiteSpace(Country)
      ? "Worldwide" : "in " + Country)</h3>

@if (customers == null)
{<
p><em>Loading...</em></p>
}
else
{<
table class="table">
  <thead>
    <tr>
      <th>Id</th>
      <th>Company Name</th>
      <th>Address</th>
      <th>Phone</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
  @foreach (Customer c in customers)
  {
    <tr>
      <td>@c.CustomerId</td>
      <td>@c.CompanyName</td>
      <td>
        @c.Address<br/>
        @c.City<br/>
        @c.PostalCode<br/>
        @c.Country
      </td>
      <td>@c.Phone</td>
      <td>
        <a class="btn btn-info" href="editcustomer/@c.CustomerId">
          <i class="oi oi-pencil"></i></a>
        <a class="btn btn-danger"
            href="deletecustomer/@c.CustomerId">
          <i class="oi oi-trash"></i></a>
      </td>
```

```
      </tr>
    }
    </tbody>
  </table>
  }

  @code {
    [Parameter]
    public string? Country { get; set; }

    private IEnumerable<Customer>? customers;

    protected override async Task OnParametersSetAsync()
    {
      if (string.IsNullOrWhiteSpace(Country))
      {
        customers = await db.Customers.ToListAsync();
      }
      else
      {
        customers = await db.Customers
          .Where(c => c.Country == Country).ToListAsync();
      }
    }
  }
```

7. Запустите сайт проекта `Northwind.BlazorServer`.
8. Запустите браузер Google Chrome.
9. Введите адрес https://localhost:5001/.
10. В левом навигационном меню щелкните на ссылке Customers Worldwide (Клиенты по всему миру). Обратите внимание на таблицу клиентов, загружаемых из базы данных и отображаемых на веб-странице (рис. 17.5).

Рис. 17.5. Список клиентов по всему миру

11. В левом навигационном меню щелкните на ссылке Customers in Germany (Клиенты в Германии) и обратите внимание, что таблица клиентов отфильтрована и показывает только клиентов из Германии.
12. В адресной строке браузера измените слово `Germany` на `UK` и обратите внимание, что таблица клиентов отфильтрована, чтобы показывать только клиентов из Великобритании.
13. В левом меню навигации щелкните на ссылке Home (Главная) и обратите внимание, что компонент `customers` (клиенты) также работает правильно при использовании в качестве встроенного компонента на странице.
14. Нажмите любую из кнопок редактирования или удаления и обратите внимание, что они возвращают сообщение `Sorry, there's nothing at this address` (Извините, по этому адресу ничего нет), поскольку мы еще не реализовали эту функциональность.
15. Закройте браузер.
16. Завершите работу веб-сервера.

Абстрагирование сервиса для компонента Blazor

В настоящее время для выборки клиентов компонент Blazor напрямую вызывает контекст базы данных Northwind. Это прекрасно работает в Blazor Server, поскольку компонент выполняется на сервере. Однако данный компонент не будет работать при размещении в Blazor WebAssembly.

Чтобы обеспечить повторное использование компонентов, создадим локальный сервис зависимостей.

1. В проекте `Northwind.BlazorServer` в папке Data создайте файл `INorthwindService.cs`. (Шаблон элемента Visual Studio называется Interface).
2. Измените его содержимое, чтобы определить контракт для локального сервиса, который абстрагирует операции CRUD:

```
namespace Packt.Shared;

public interface INorthwindService
{
  Task<List<Customer>> GetCustomersAsync();
  Task<List<Customer>> GetCustomersAsync(string country);
  Task<Customer?> GetCustomerAsync(string id);
  Task<Customer> CreateCustomerAsync(Customer c);
  Task<Customer> UpdateCustomerAsync(Customer c);
  Task DeleteCustomerAsync(string id);
}
```

3. В папке Data создайте файл `NorthwindService.cs` и измените его содержимое, чтобы реализовать интерфейс `INorthwindService`, используя контекст базы данных Northwind:

```
using Microsoft.EntityFrameworkCore;

namespace Packt.Shared;

public class NorthwindService : InorthwindService
{
  private readonly NorthwindContext db;

  public NorthwindService(NorthwindContext db)
  {
    this.db = db;
  }

  public Task<List<Customer>> GetCustomersAsync()
  {
    return db.Customers.ToListAsync();
  }

  public Task<List<Customer>> GetCustomersAsync(string country)
  {
    return db.Customers.Where(c => c.Country == country).ToListAsync();
  }

  public Task<Customer?> GetCustomerAsync(string id)
  {
    return db.Customers.FirstOrDefaultAsync
      (c => c.CustomerId == id);
  }

  public Task<Customer> CreateCustomerAsync(Customer c)
  {
    db.Customers.Add(c);
    db.SaveChangesAsync();
    return Task.FromResult(c);
  }

  public Task<Customer> UpdateCustomerAsync(Customer c)
  {
    db.Entry(c).State = EntityState.Modified;
    db.SaveChangesAsync();
    return Task.FromResult(c);
  }

  public Task DeleteCustomerAsync(string id)
  {
    Customer? customer = db.Customers.FirstOrDefaultAsync
      (c => c.CustomerId == id).Result;
```

```
      if (customer == null)
      {
        return Task.CompletedTask;
      }
      else
      {
        db.Customers.Remove(customer);
        return db.SaveChangesAsync();
      }
    }
  }
```

4. В файле `Program.cs` в разделе, где настраиваются сервисы, добавьте операторы для регистрации `NorthwindService` в качестве временного сервиса, реализующего интерфейс `INorthwindService`, как показано ниже:

```
Builder.Services.AddTransient<InorthwindService, NorthwindService>();
```

5. В папке Pages откройте файл `Customers.razor`, замените директиву для внедрения контекста базы данных Northwind на директиву для внедрения зарегистрированного сервиса Northwind:

```
@inject INorthwindService service
```

6. Измените метод `OnParametersSetAsync`, чтобы вызвать сервис, как показано ниже:

```
protected override async Task OnParametersSetAsync()
{
  if (string.IsNullOrWhiteSpace(Country))
  {
    customers = await service.GetCustomersAsync();
  }
  else
  {
    customers = await service.GetCustomersAsync(Country);
  }
}
```

7. Запустите проект сайта `Northwind.BlazorServer` и убедитесь, что он сохраняет прежнюю функциональность.

Определение форм с помощью компонента EditForm

Для построения форм Microsoft предоставляет готовые компоненты. С их помощью мы будем предоставлять, создавать и редактировать функции для клиентов.

Чтобы упростить использование форм с Blazor, Microsoft предоставляет компонент `EditForm` и элементы форм, например `InputText`.

Компонент EditForm может иметь модель, настроенную для привязки ее к объекту со свойствами и обработчиками событий для настраиваемой проверки, а также для распознавания стандартных атрибутов проверки Microsoft в классе модели:

```
<EditForm Model="@customer" OnSubmit="ExtraValidation">
  <DataAnnotationsValidator />
  <ValidationSummary />
  <InputText id="name" @bind-Value="customer.CompanyName" />
  <button type="submit">Submit</button>
</EditForm>

@code {
  private Customer customer = new();

  private void ExtraValidation()
  {
    // выполнение дополнительной проверки
  }
}
```

Чтобы отобразить сообщение рядом с отдельным элементом формы, в качестве альтернативы компоненту ValidationSummary можно использовать компонент ValidationMessage.

Создание и использование компонента формы клиента

Теперь мы можем создать общий компонент для создания или редактирования информации о клиенте.

1. В папке Shared создайте файл CustomerDetail.razor. (Шаблон элемента Visual Studio называется Razor Component.) Этот компонент будет повторно использоваться в нескольких компонентах страницы.
2. Измените его содержимое, чтобы определить форму для редактирования свойств клиента:

```
<EditForm Model="@Customer" OnValidSubmit="@OnValidSubmit">
  <DataAnnotationsValidator />
  <div class="form-group">
    <div>
      <label>Customer Id</label>
      <div>
        <InputText @bind-Value="@Customer.CustomerId" />
        <ValidationMessage For="@(() => Customer.CustomerId)" />
      </div>
    </div>
  </div>
```

```
  <div class="form-group ">
    <div>
      <label>Company Name</label>
      <div>
        <InputText @bind-Value="@Customer.CompanyName" />
        <ValidationMessage For="@(() => Customer.CompanyName)" />
      </div>
    </div>
  </div>
  <div class="form-group ">
    <div>
      <label>Address</label>
      <div>
        <InputText @bind-Value="@Customer.Address" />
        <ValidationMessage For="@(() => Customer.Address)" />
      </div>
    </div>
  </div>
  <div class="form-group ">
    <div>
      <label>Country</label>
      <div>
        <InputText @bind-Value="@Customer.Country" />
        <ValidationMessage For="@(() => Customer.Country)" />
      </div>
    </div>
  </div>
  <button type="submit" class="btn btn-@ButtonStyle">
    @ButtonText
  </button>
</EditForm>

@code {
  [Parameter]
  public Customer Customer { get; set; } = null!;

  [Parameter]
  public string ButtonText { get; set; } = "Save Changes";

  [Parameter]
  public string ButtonStyle { get; set; } = "info";

  [Parameter]
  public EventCallback OnValidSubmit { get; set; }
}
```

3. В папке Pages создайте файл `CreateCustomer.razor`. Он будет компонентом страницы с возможностью маршрутизации.
4. Измените его содержимое, чтобы использовать компонент сведений о клиенте для создания нового клиента:

```
@page "/createcustomer"
@inject INorthwindService service
@inject NavigationManager navigation

<h3>Create Customer</h3>
<CustomerDetail ButtonText="Create Customer"
                Customer="@customer"
                OnValidSubmit="@Create" />

@code {
  private Customer customer = new();

  private async Task Create()
  {
    await service.CreateCustomerAsync(customer);
    navigation.NavigateTo("customers");
  }
}
```

5. В папке Pages найдите и откройте файл `Customers.razor` и после элемента `<h3>` добавьте элемент `<div>` с кнопкой для перехода к компоненту страницы `createcustomer`:

```
<div class="form-group">
  <a class="btn btn-info" href="createcustomer">
  <i class="oi oi-plus"></i> Create New</a>
</div>
```

6. В папке Pages создайте файл `EditCustomer.razor` и измените его содержимое, чтобы использовать компонент сведений о клиенте для редактирования и сохранения изменений для существующего клиента:

```
@page "/editcustomer/{customerid}"
@inject INorthwindService service
@inject NavigationManager navigation

<h3>Edit Customer</h3>
<CustomerDetail ButtonText="Update"
                Customer="@customer"
                OnValidSubmit="@Update" />

@code {
  [Parameter]
  public string CustomerId { get; set; }

  private Customer? Customer = new();

  protected async override Task OnParametersSetAsync()
  {
    customer = await service.GetCustomerAsync(CustomerId);
  }
```

```
  private async Task Update()
  {
    if (customer is not null)
    {
      await service.UpdateCustomerAsync(customer);
    }
    navigation.NavigateTo("customers");
  }
}
```

7. В папке `Pages` создайте файл `DeleteCustomer.razor` и измените его содержимое, чтобы использовать компонент сведений о клиенте, чтобы показать клиента, которого мы собираемся удалить:

```
@page "/deletecustomer/{customerid}"
@inject INorthwindService service
@inject NavigationManager navigation

<h3>Delete Customer</h3>
<div class="alert alert-danger">
  Warning! This action cannot be undone!
</div>
<CustomerDetail ButtonText="Delete Customer"
                ButtonStyle="danger"
                Customer="@customer"
                OnValidSubmit="@Delete" />

@code {
  [Parameter]
  public string CustomerId { get; set; }

  private Customer? customer = new();

  protected async override Task OnParametersSetAsync()
  {
    customer = await service.GetCustomerAsync(CustomerId);
  }

  private async Task Delete()
  {
    if (customer is not null)
    {
      await service.DeleteCustomerAsync(CustomerId);
    }
    navigation.NavigateTo("customers");
  }
}
```

Тестирование компонента формы клиента

Теперь мы можем протестировать компонент формы клиента и способы его использования для создания, редактирования и удаления клиентов.

1. Запустите проект сайта `Northwind.BlazorServer`.
2. Запустите браузер Google Chrome.
3. Перейдите по адресу https://localhost:5001/.
4. Перейдите к разделу Customers Worldwide (Клиенты по всему миру) и нажмите кнопку + Create New (+ Создать).
5. Введите недопустимый идентификатор клиента, например `ABCDEF`, оставьте текстовое поле и обратите внимание на сообщение проверки (рис. 17.6).

Рис. 17.6. Создание нового клиента и ввод неверного идентификатора клиента

6. Измените идентификатор клиента на `ABCDE`, введите значения для других текстовых полей и нажмите кнопку Create Customer (Создать клиента).
7. Когда появится список клиентов, прокрутите страницу вниз, чтобы увидеть нового клиента.
8. В строке клиента ABCDE нажмите кнопку со значком Edit (Изменить). Измените адрес, нажмите кнопку Update (Обновить) и обратите внимание, что выбранная запись обновлена.
9. В строке клиента ABCDE нажмите кнопку со значком Delete (Удалить). Обратите внимание на предупреждение, затем нажмите кнопку Delete Customer (Удалить клиента). Обратите внимание, что выбранная запись удалена.
10. Закройте браузер Google Chrome и завершите работу веб-сервера.

Создание компонентов с помощью Blazor WebAssembly

Чтобы четко увидеть основные различия, мы повторно используем ту же функциональность в проекте Blazor WebAssembly.

Поскольку мы абстрагировали локальный сервис зависимостей в интерфейсе `INorthwindService`, мы сможем повторно использовать все компоненты и этот интерфейс, а также классы сущностной модели. Единственное, что нам потребуется переписать, — это реализацию класса `NorthwindService`. Вместо того чтобы прямо вызывать класс `NorthwindContext`, он будет вызывать контроллер клиента Web API на стороне сервера, как показано на рис. 17.7.

Рис. 17.7. Сравнение реализаций с использованием Blazor Server и Blazor WebAssembly

Настройки сервера для Blazor WebAssembly

Во-первых, нам нужен веб-сервис, при вызове которого клиентское приложение сможет получать клиентов и управлять ими. Если вы прочитали главу 16 и выполнили упражнения в ней, то у вас есть сервис клиентов в проекте сервисов `Northwind.WebApi`, который вы можете использовать. Однако, чтобы сделать эту

главу более независимой, мы создадим контроллер клиента Web API в проекте `Northwind.BlazorWasm.Server`.

Внимание! В отличие от предыдущих проектов относительные ссылки на пути для общих проектов, таких как сущностные модели и база данных, находятся на два уровня выше, например «..\..».

1. В проекте/папке `Server` откройте файл `Northwind.BlazorWasm.Server.csproj` и добавьте операторы для ссылки на проект контекста базы данных Northwind для SQL Server или SQLite:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include="..\..\Northwind.Common.DataContext.Sqlite
  \Northwind.Common.DataContext.Sqlite.csproj" />
</ItemGroup>
```

2. Соберите проект `Northwind.BlazorWasm.Server`.
3. В проекте/папке `Server` откройте файл `Program.cs` и добавьте оператор, чтобы импортировать пространство имен для работы с контекстом базы данных Northwind:

```
using Packt.Shared;
```

4. В разделе настройки сервисов добавьте оператор для регистрации контекста базы данных Northwind для SQL Server или SQLite:

```
// если используете SQL Server
builder.Services.AddNorthwindContext();

// если используете SQLite
builder.Services.AddNorthwindContext(
  relativePath: Path.Combine("..", ".."));
```

5. В проекте `Server` в папке Controllers создайте файл `CustomersController.cs` и добавьте операторы для определения класса контроллера Web API с такими же методами CRUD, как и ранее:

```
using Microsoft.AspNetCore.Mvc; // [ApiController], [Route]
using Microsoft.EntityFrameworkCore; // ToListAsync, FirstOrDefaultAsync
using Packt.Shared; // NorthwindContext, Customer

namespace Northwind.BlazorWasm.Server.Controllers;

[ApiController]
[Route("api/[controller]")]
public class CustomersController : ControllerBase
{
  private readonly NorthwindContext db;
```

```
public CustomersController(NorthwindContext db)
{
  this.db = db;
}

[HttpGet]
public async Task<List<Customer>> GetCustomersAsync()
{
  return await db.Customers.ToListAsync();
}

[HttpGet("in/{country}")] // different path to disambiguate
public async Task<List<Customer>> GetCustomersAsync(string country)
{
  return await db.Customers
    .Where(c => c.Country == country).ToListAsync();
}

[HttpGet("{id}")]
public async Task<Customer?> GetCustomerAsync(string id)
{
  return await db.Customers
    .FirstOrDefaultAsync(c => c.CustomerId == id);
}

[HttpPost]
public async Task<Customer?> CreateCustomerAsync
  (Customer customerToAdd)
{
  Customer? existing = await db.Customers.FirstOrDefaultAsync
    (c => c.CustomerId == customerToAdd.CustomerId);

  if (existing == null)
  {
    db.Customers.Add(customerToAdd);
    int affected = await db.SaveChangesAsync();
    if (affected == 1)
    {
      return customerToAdd;
    }
  }
  return existing;
}

[HttpPut]
public async Task<Customer?> UpdateCustomerAsync(Customer c)
{
  db.Entry(c).State = EntityState.Modified;
  int affected = await db.SaveChangesAsync();
  if (affected == 1)
```

```
    {
      return c;
    }
    return null;
  }

  [HttpDelete("{id}")]
  public async Task<int> DeleteCustomerAsync(string id)
  {
    Customer? c = await db.Customers.FirstOrDefaultAsync
      (c => c.CustomerId == id);

    if (c != null)
    {
      db.Customers.Remove(c);
      int affected = await db.SaveChangesAsync();
      return affected;
    }
    return 0;
  }
}
```

Настройка клиента для Blazor WebAssembly

Во-вторых, мы можем повторно использовать компоненты из проекта Blazor Server. Поскольку компоненты будут идентичными, мы можем их скопировать, и нам необходимо будет только внести изменения в локальную реализацию абстрактного сервиса Northwind.

1. В проекте `Client` откройте файл `Northwind.BlazorWasm.Client.csproj` и добавьте операторы для ссылки на проект библиотеки сущностных моделей Northwind (не проект контекста базы данных) для SQL Server или SQLite, как показано ниже:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include="..\..\Northwind.Common.EntityModels.Sqlite\
Northwind.Common.EntityModels.Sqlite.csproj" />
</ItemGroup>
```

2. Соберите проект `Northwind.BlazorWasm.Client`.
3. В проекте `Client` откройте файл `_Imports.razor` и импортируйте пространство имен `Packt.Shared`, чтобы сделать типы сущностной модели Northwind доступными во всех компонентах Blazor:

```
@using Packt.Shared
```

4. В проекте `Client` в папке Shared откройте файл `NavMenu.razor`. Для клиентов по всему миру и во Франции добавьте элемент `NavLink`:

```
<div class="nav-item px-3">
  <NavLink class="nav-link" href="customers" Match="NavLinkMatch.All">
    <span class="oi oi-people" aria-hidden="true"></span>
    Customers Worldwide
  </NavLink>
</div>
<div class="nav-item px-3">
  <NavLink class="nav-link" href="customers/France">
    <span class="oi oi-people" aria-hidden="true"></span>
    Customers in France
  </NavLink>
</div>
```

5. Скопируйте компонент `CustomerDetail.razor` из папки Shared проекта `Northwind.BlazorServer` в папку Shared проекта `Northwind.BlazorWasm Client`.
6. Скопируйте следующие компоненты маршрутизируемой страницы из папки Pages проекта `Northwind.BlazorServer` в папку Pages проекта `Northwind.BlazorWasm Client`.
   - `CreateCustomer.razor`;
   - `Customers.razor`;
   - `DeleteCustomer.razor`;
   - `EditCustomer.razor`.
7. В проекте `Client` создайте папку Data.
8. Скопируйте файл `INorthwindService.cs` из папки Data проекта `Northwind.BlazorServer` в папку Data проекта `Client`.
9. В папке Data создайте файл `NorthwindService.cs`.
10. Измените его содержимое, чтобы реализовать интерфейс `InorthwindService`, используя `HttpClient` для вызова сервисов клиентов Web API:

```
using System.NET.Http.Json; // GetFromJsonAsync, ReadFromJsonAsync
using Packt.Shared; // Customer

namespace Northwind.BlazorWasm.Client.Data
{
  public class NorthwindService : InorthwindService
  {
    private readonly HttpClient http;

    public NorthwindService(HttpClient http)
    {
      this.http = http;
    }
```

```
    public Task<List<Customer>> GetCustomersAsync()
    {
      return http.GetFromJsonAsync
        <List<Customer>>("api/customers");
    }

    public Task<List<Customer>> GetCustomersAsync(string country)
    {
      return http.GetFromJsonAsync
        <List<Customer>>($"api/customers/in/{country}");
    }

    public Task<Customer> GetCustomerAsync(string id)
    {
      return http.GetFromJsonAsync
        <Customer>($"api/customers/{id}");
    }

    public async Task<Customer>
      CreateCustomerAsync (Customer c)
    {
      HttpResponseMessage response = await
        http.PostAsJsonAsync("api/customers", c);

      return await response.Content
        .ReadFromJsonAsync<Customer>();
    }

    public async Task<Customer> UpdateCustomerAsync(Customer c)
    {
      HttpResponseMessage response = await
        http.PutAsJsonAsync("api/customers", c);

      return await response.Content
        .ReadFromJsonAsync<Customer>();
    }

    public async Task DeleteCustomerAsync(string id)
    {
      HttpResponseMessage response = await
        http.DeleteAsync($"api/customers/{id}");
    }
  }
}
```

11. В файле `Program.cs` импортируйте пространства имен `Packt.Shared` и `Northwind.BlazorWasm.Client.Data`.
12. В разделе настройки сервисов добавьте оператор для регистрации сервиса зависимостей Northwind:

```
Builder.Services.AddTransient<InorthwindService, NorthwindService>();
```

Тестирование компонентов и сервиса Blazor WebAssembly

Теперь мы можем запустить проект хостинга сервера Blazor WebAssembly, чтобы проверить, работают ли компоненты с абстрактным сервисом Northwind, который вызывает сервис клиентов Web API.

1. В проекте/папке `Server` запустите проект сайта `Northwind.BlazorWasm.Server`.
2. Запустите браузер Google Chrome. Перейдите в Developer Tools (Инструменты разработчика) и выберите вкладку Network (Сеть).
3. Перейдите по адресу: https://localhost:5001/. Ваш номер порта будет другим, поскольку он назначается случайным образом. Просмотрите вывод консоли, чтобы увидеть подробную информацию.
4. Выберите вкладку Console (Консоль). Обратите внимание, что Blazor WebAssembly загрузил сборки .NET в кэш браузера и что они занимают около 10 Мбайт (рис. 17.8).

Рис. 17.8. Blazor WebAssembly загружает сборки .NET в кэш браузера

5. Выберите вкладку Network (Сеть).
6. В левом меню навигации щелкните на ссылке Customers Worldwide (Клиенты по всему миру). Обратите внимание на HTTP-запрос `GET` с ответом в формате JSON, содержащим все записи о клиентах (рис. 17.9).

Рис. 17.9. HTTP-запрос GET с ответом в формате JSON, содержащим информацию обо всех клиентах

7. Нажмите кнопку + Create New (+ Создать). Чтобы добавить нового клиента, заполните форму и обратите внимание на выполненный HTTP-запрос `POST` (рис. 17.10).

Рис. 17.10. HTTP-запрос POST для создания нового клиента

8. Повторите предыдущие шаги, чтобы отредактировать, а затем удалить вновь созданного клиента.
9. Закройте браузер Google Chrome и завершите работу веб-сервера.

Улучшение приложений Blazor WebAssembly

Существуют распространенные способы улучшения приложений Blazor WebAssembly. Сейчас мы рассмотрим самые популярные из них.

Включение Blazor WebAssembly AOT

По умолчанию среда выполнения .NET, используемая Blazor WebAssembly, выполняет интерпретацию IL с помощью интерпретатора, написанного на WebAssembly. В отличие от других приложений .NET он не использует динамический компилятор (JIT), поэтому производительность при выполнении рабочих нагрузок, требующих интенсивного использования ЦПУ, ниже, чем можно было бы ожидать.

В .NET 6 компания Microsoft добавила поддержку *опережающей/статической компиляции* (ahead-of-time, AOT), но вы должны явно согласиться на нее, поскольку, хотя она может значительно повысить производительность во время выполнения, компиляция с AOT может занять несколько минут в небольших проектах, подобных тем, о которых идет речь в этой книге, и потенциально гораздо больше времени в более крупных проектах. Размер скомпилированного приложения также больше, чем без AOT, — обычно в два раза. Поэтому решение об использовании AOT основывается на балансе между увеличением времени компиляции и загрузки браузера и потенциально гораздо более быстрым временем выполнения.

AOT была самой востребованной функцией в опросе Microsoft, а отсутствие AOT было названо основной причиной, по которой некоторые разработчики еще не используют .NET для разработки *одностраничных приложений* (single-page applications, SPA).

Установим дополнительную необходимую нагрузку для Blazor AOT под названием *.NET WebAssembly build tools*, а затем включим AOT для нашего проекта Blazor WebAssembly.

1. В командной строке или в терминале с правами администратора установите рабочую нагрузку Blazor AOT, как показано ниже:

```
dotnet workload install wasm-tools
```

2. Обратите внимание на сообщения:

```
...
Installing pack Microsoft.NET.Runtime.MonoAOTCompiler.Task version
6.0.0...
Installing pack Microsoft.NETCore.App.Runtime.AOT.Cross.browser-wasm
version 6.0.0...
Successfully installed workload(s) wasm-tools.
```

3. Измените файл проекта `Northwind.BlazorWasm.Client`, чтобы включить AOT, как показано ниже:

```
<PropertyGroup>
  <TargetFramework>net6.0</TargetFramework>
  <Nullable>enable</Nullable>
  <ImplicitUsings>enable</ImplicitUsings>
  <ServiceWorkerAssetsManifest>service-worker-assets.js
    </ServiceWorkerAssetsManifest>
  <RunAOTCompilation>true</RunAOTCompilation>
</PropertyGroup>
```

4. Опубликуйте проект `Northwind.BlazorWasm.Client`:

```
dotnet publish -c Release
```

5. Обратите внимание, что к 75 сборкам применена AOT:

```
  Northwind.BlazorWasm.Client -> C:\Code\PracticalApps\Northwind.
BlazorWasm\Client\bin\Release\net6.0\Northwind.BlazorWasm.Client.dll
  Northwind.BlazorWasm.Client (Blazor output) -> C:\Code\PracticalApps\
Northwind.BlazorWasm\Client\bin\Release\net6.0\wwwroot
  Optimizing assemblies for size, which may change the behavior of the
app. Be sure to test after publishing. See: https://aka.ms/dotnet-illink
  AOT'ing 75 assemblies
  [1/75] Microsoft.Extensions.Caching.Abstractions.dll -> Microsoft.
Extensions.Caching.Abstractions.dll.bc
  ...
  [75/75] Microsoft.EntityFrameworkCore.Sqlite.dll -> Microsoft.
EntityFrameworkCore.Sqlite.dll.bc
  Compiling native assets with emcc. This may take a while ...
  ...
  Linking with emcc. This may take a while ...
  ...
  Optimizing dotnet.wasm ...
  Compressing Blazor WebAssembly publish artifacts. This may take a while...
```

6. Дождитесь окончания процесса. Этот процесс может занять около 20 минут даже на современном многоядерном процессоре.
7. Перейдите в папку Northwind.BlazorWasm\Client\bin\release\net6.0\publish и обратите внимание на увеличение размера загрузки с 10 до 112 Мбайт.

Как видите, без AOT загруженное приложение Blazor WebAssembly занимало около 10 Мбайт. С AOT оно заняло около 112 Мбайт. Такое увеличение размера повлияет на восприятие посетителя сайта.

Использование AOT представляет собой баланс между более медленной начальной загрузкой и более быстрым потенциальным выполнением.

В зависимости от специфики вашего приложения, AOT может и не стоить того.

Поддержка прогрессивных веб-приложений

Поддержка *прогрессивного веб-приложения* (Progressive Web App, PWA) в проектах Blazor WebAssembly означает, что веб-приложение получает следующие преимущества:

- работает как обычная веб-страница, пока пользователь явно не решит перейти к полноценной работе с ним;
- когда оно будет установлено, запустите его из меню Start (Пуск) или используя значок на рабочем столе;
- визуально отображается в собственном окне вместо вкладки браузера;
- работает в офлайн-режиме;
- обновляется автоматически.

Разберемся, как работает поддержка PWA.

1. Запустите проект веб-хоста `Northwind.BlazorWasm.Server`.
2. Перейдите на https://localhost:5001/ или любой другой номер вашего порта.
3. В адресной строке браузера Google Chrome нажмите расположенный справа значок с всплывающей подсказкой Install Northwind.BlazorWasm (Установить Northwind Blazor Wasm) (рис. 17.11).

Рис. 17.11. Установка Northwind.BlazorWasm в качестве приложения

4. Нажмите кнопку Install (Установить).
5. Закройте браузер Google Chrome. Вам также может потребоваться закрыть приложение, если оно запускается автоматически.
6. Запустите приложение Northwind.BlazorWasm, используя меню Start (Пуск) операционной системы Windows или панель запуска macOS. Обратите внимание, что в нем имеются все возможности приложения.
7. Справа от строки заголовка щелкните на меню с тремя точками. Обратите внимание, что вы можете удалить приложение. Однако пока удалять его не следует.
8. Перейдите к панели Developer Tools (Инструменты разработчика). В Windows нажмите клавишу F12 или Ctrl+Shift+I. В macOS нажмите Cmd+Shift+I.

9. Выберите вкладку Network (Сеть) и затем в раскрывающемся списке Throttling (Регулирование) выберите инициализацию Offline (Офлайн).
10. В левом меню навигации нажмите ссылку Home (Главная), а затем нажмите Customers Worldwide (Клиенты по всему миру) и обратите внимание на то, что не удалось загрузить ни одного клиента, а вместо этого в нижней части окна приложения отображено сообщение об ошибке (рис. 17.12).

Рис. 17.12. Невозможно загрузить записи о клиентах, когда сеть отключена

11. На панели Developer Tools (Инструменты разработчика) снова установите для параметра Throttling (Регулирование) значение Disabled: No throttling (Отключено: нет регулирования).
12. Щелкните на ссылке Reload (Перезапуск), расположенной на желтой строке ошибок в нижней части приложения, и обратите внимание, что функциональность возвращается.
13. Теперь вы можете удалить приложение PWA или просто закрыть его.

Внедрение автономной поддержки для PWA

Мы могли бы усовершенствовать свой опыт, кэшируя HTTP-ответы `GET` от сервиса Web API локально, сохраняя новых, измененных или удаленных клиентов локально, а затем синхронизируя с сервером, выполняя сохраненные HTTP-запросы после восстановления сетевого подключения. Однако для хорошей реализации требуется много усилий, так что это выходит за рамки данной книги.

Анализатор совместимости браузеров для Blazor WebAssembly

В версии .NET 6 компания Microsoft унифицировала библиотеку .NET для всех рабочих нагрузок. В теории это означает, что приложение Blazor WebAssembly имеет полный доступ ко всем API .NET, однако на практике оно работает внутри изолированной программной среды браузера, поэтому существуют

ограничения. Если вы вызовете неподдерживаемый API, то это вызовет исключение `PlatformNotSupportedException`.

Чтобы получать предупреждения о неподдерживаемых API, вы можете добавить анализатор совместимости платформ, который уведомит вас, когда ваш код использует API, которые не поддерживаются браузерами.

Шаблоны проектов приложения Blazor WebAssembly App и библиотеки классов Razor Class Library автоматически включают проверку совместимости с браузерами.

Чтобы вручную активировать проверку совместимости браузеров, например, в проекте библиотеки классов, добавьте запись в файл проекта, как показано ниже:

```
<ItemGroup>
  <SupportedPlatform Include="browser" />
</ItemGroup>
```

Microsoft дополняет неподдерживаемые API:

```
[UnsupportedOSPlatform("browser")]
public void DoSomethingOutsideTheBrowserSandbox()
{
  ...
}
```

Если вы создаете библиотеки, которые не должны использоваться в приложениях Blazor WebAssembly, то вам следует дополнить свои API таким же образом.

Совместное использование компонентов Blazor в библиотеке классов

В настоящее время у нас есть компоненты, дублирующиеся в проекте Blazor Server и проекте Blazor WebAssembly. Было бы лучше определить их один раз в проекте библиотеки классов и ссылаться на них из двух других проектов Blazor.

Создадим новую библиотеку классов Razor.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: Razor class library/`razorclasslib`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.Blazor.Customers`;
   4) флажок Support pages and views установлен.
2. В проекте `Northwind.Blazor.Customers` добавьте ссылку на проект `Northwind.Common.EntityModels.Sqlite` или проект `SqlServer`.

3. В проект `Northwind.Blazor.Customers` добавьте запись для проверки совместимости с браузерами, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\Northwind.Common.EntityModels.Sqlite
    \Northwind.Common.EntityModels.Sqlite.csproj" />
  </ItemGroup>

  <ItemGroup>
    <SupportedPlatform Include="browser" />
  </ItemGroup>

</Project>
```

4. В проекте `Northwind.BlazorServer` добавьте ссылку на проект `Northwind.Blazor.Customers`.
5. Соберите проект `Northwind.BlazorServer`.
6. В проекте `Northwind.Blazor.Customers` удалите папку Areas и все ее содержимое.
7. Скопируйте файл `_Imports.razor` из корня проекта `Northwind.BlazorServer` в корень проекта `Northwind.Blazor.Customers`.
8. В файле `_Imports.razor` удалите два оператора импорта для пространства имен `Northwind.BlazorServer` и добавьте оператор импорта пространства имен, которое будет содержать наши общие компоненты Blazor:

```
@using Northwind.Blazor.Customers.Shared
```

9. Создайте три папки: Data, Pages и Shared.
10. Переместите файл `INorthwindService.cs` из папки Data проекта `Northwind.BlazorServer` в папку Data проекта `Northwind.Blazor.Customers`.
11. Переместите все компоненты из папки Shared проекта `Northwind.BlazorServer` в папку Shared проекта `Northwind.Blazor.Customers`.
12. Переместите компоненты `CreateCustomer.razor`, `Customers.razor`, `EditCustomer.razor` и `DeleteCustomer.razor` из папки Pages проекта `Northwind.BlazorServer` в папку Pages проекта `Northwind.Blazor.Customers`.

Мы не трогаем другие компоненты страницы, поскольку они имеют зависимости от сервиса погоды, который не был должным образом перепроектирован.

13. В проекте `Northwind.BlazorServer` в файле `_Imports.razor` удалите оператор `using` для `Northwind.BlazorServer.Shared` и добавьте операторы для импорта страницы и общих компонентов в библиотеку классов:

```
@using Northwind.Blazor.Customers.Pages
@using Northwind.Blazor.Customers.Shared
```

14. В проекте `Northwind.BlazorServer` в файле `App.razor` добавьте параметр, чтобы дать компоненту `Router` указание сканировать дополнительную сборку с целью установить маршруты для компонентов страницы в библиотеке классов, как показано ниже:

```
<Router AppAssembly="@typeof(App).Assembly"
        AdditionalAssemblies="new[] { typeof(Customers).Assembly }">
```

Не имеет значения, какой класс вы укажете, главное, чтобы он был во внешней сборке. Я выбрал Customers, поскольку это самый важный и очевидный класс компонента.

15. Запустите проект `Northwind.BlazorServer` и обратите внимание, что он ведет себя так же, как и раньше.

Теперь вы можете повторно применять компоненты Blazor в других проектах Blazor Server. Однако вы не можете использовать эту библиотеку классов в проектах Blazor WebAssembly, поскольку она зависит от полной рабочей нагрузки ASP.NET Core. Создание библиотек компонентов Blazor, работающих с обеими моделями хостинга, выходит за рамки этой книги.

Взаимодействие с JavaScript

По умолчанию компоненты Blazor не имеют доступа к возможностям браузера, таким как локальное хранилище, геолокация и сбор данных мультимедиа, а также к библиотекам JavaScript, таким как React или Vue. Если вам необходимо взаимодействовать с ними, то вы можете использовать JavaScript Interop.

Рассмотрим пример, в котором используется окно уведомлений окна браузера и локальное хранилище, которое может сохранять до 5 Мбайт данных для каждого посетителя в течение неограниченного времени.

1. В проекте `Northwind.BlazorServer` в папке wwwroot добавьте папку scripts.
2. В папке `scripts` добавьте файл `interop.js`.

3. Измените его содержимое:

```
function messageBox(message) {
  window.alert(message);
}

function setColorInStorage() {
  if (typeof (Storage) !== "undefined") {
    localStorage.setItem("color",
      document.getElementById("colorBox").value);
  }
}

function getColorFromStorage() {
  if (typeof (Storage) !== "undefined") {
    document.getElementById("colorBox").value =
      localStorage.getItem("color");
  }
}
```

4. В папке Pages в файле `_Layout.cshtml` после элемента `script`, реализующего поддержку Blazor Server, добавьте элемент `script`, ссылающийся на только что созданный файл JavaScript:

```
<script src="scripts/interop.js"></script>
```

5. В папке Pages в файле `Index.razor` удалите два экземпляра компонента `Customers`, а затем добавьте кнопку и блок кода, который использует сервис зависимостей времени выполнения Blazor JavaScript для вызова функции JavaScript:

```
<button type="button" class="btn btn-info" @onclick="AlertBrowser">
  Poke the browser</button>

<hr />

<input id="colorBox" />

<button type="button" class="btn btn-info" @onclick="SetColor">
  Set Color</button>

<button type="button" class="btn btn-info" @onclick="GetColor">
  Get Color</button>

@code {
  [Inject]
  public IJSRuntime JSRuntime { get; set; } = null!;

  public async Task AlertBrowser()
  {
    await JSRuntime.InvokeVoidAsync(
      "messageBox", "Blazor poking the browser");
  }
```

```
public async Task SetColor()
  {
    await JSRuntime.InvokeVoidAsync("setColorInStorage");
  }

  public async Task GetColor()
  {
    await JSRuntime.InvokeVoidAsync("getColorFromStorage");
  }
}
```

6. Запустите проект `Northwind.BlazorServer`.
7. Запустите браузер Google Chrome и перейдите по адресу https://localhost:5001/.
8. На главной странице в текстовом поле введите `red` (красный), а затем нажмите кнопку Set Color (Задать цвет).
9. Откройте панель Developer Tools (Инструменты разработчика), выберите вкладку Application (Приложение), разверните пункт Local Storage (Локальное хранилище), выберите вариант https://localhost:5001 и обратите внимание на пару «ключ — значение» `color-red` (рис. 17.13).

Рис. 17.13. Хранение цвета в локальном хранилище браузера с помощью JavaScript Interop

10. Закройте браузер Google Chrome и завершите работу веб-сервера.
11. Запустите проект `Northwind.BlazorServer`.
12. Запустите браузер Google Chrome и перейдите по адресу https://localhost:5001/.
13. На главной странице нажмите кнопку Get Color (Получить цвет) и обратите внимание, что в текстовом поле отображается значение `red` (красный), которое извлекается из локального хранилища между сеансами посетителей.
14. Закройте браузер Google Chrome и завершите работу веб-сервера.

Библиотеки компонентов Blazor

Существует множество библиотек компонентов Blazor. Платные библиотеки компонентов предоставлены такими компаниями, как Telerik, DevExpress и Syncfusion. Библиотеки компонентов Blazor с открытым исходным кодом включают следующие:

- Radzen Blazor Components: https://blazor.radzen.com/;
- Awesome Open Source Blazor Projects: https://awesomeopensource.com/projects/blazor.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 17.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какие две основные модели хостинга существуют в Blazor и чем они различаются?
2. Какая дополнительная конфигурация требуется в классе `Startup` в проекте сайта Blazor Server по сравнению с проектом сайта ASP.NET Core MVC?
3. Одним из преимуществ Blazor является возможность реализации компонентов на стороне клиента с помощью C# и .NET вместо JavaScript. Необходим ли компоненту Blazor какой-либо JavaScript?
4. Какова роль файла `App.razor` в проекте Blazor?
5. В чем преимущество использования компонента `<NavLink>`?
6. Каким образом значение передается компоненту?
7. В чем преимущество использования компонента `<EditForm>`?
8. Каким образом выполняются некоторые операторы при установленных параметрах?
9. Каким образом выполняются некоторые операторы при появлении компонента?
10. Каковы два ключевых различия в классе `Program` между сервером Blazor и проектом Blazor WebAssembly?

Упражнение 17.2. Создание компонента таблицы умножения

Создайте компонент, отображающий таблицу умножения на основе параметра `Number`, а затем протестируйте свой компонент двумя способами.

Сначала в файл `Index.razor` добавьте экземпляр компонента:

```
<timestable Number="6" />
```

Затем в адресной строке браузера введите следующий путь:

```
https://localhost:5001/timestable/6
```

Упражнение 17.3. Создание элемента навигации по стране

Добавьте метод действия в класс `CustomersController`, чтобы возвращать список названий стран.

В общем компоненте `NavMenu` вызовите веб-сервис клиента, чтобы получить список названий стран, и пройдитесь по ним, создавая пункт меню для каждой страны.

Упражнение 17.4. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-17---building-user-interfaces-using-blazor, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали, как создавать компоненты Blazor, размещенные как на сервере, так и на WebAssembly. Вы изучили некоторые ключевые различия между двумя моделями хостинга, например способы управления данными с помощью сервисов зависимостей.

Послесловие

Я хотел, чтобы эта книга отличалась от других по той же теме. Я надеюсь, она вам понравилась и вы оценили обилие примеров.

Дальнейшее изучение C# и .NET

Если вы хотите получить больше информации, чем я смог включить в эту книгу, то примечания, практические рекомендации и ссылки в репозитории GitHub укажут вам правильное направление: https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md.

Совершенствование навыков с помощью рекомендаций по проектированию

Теперь, когда вы изучили основы разработки с помощью C# и .NET, вы готовы улучшить качество своего кода, изучив более подробные рекомендации по проектированию.

Еще в начале эры .NET Framework компания Microsoft опубликовала книгу с рекомендациями по всем областям разработки .NET. Эти рекомендации до сих пор в значительной степени применимы к современной разработке .NET.

В книге рассматриваются следующие темы:

- рекомендации по присвоению имен;
- рекомендации по проектированию типов;
- рекомендации по проектированию элементов;
- проектирование с учетом расширяемости;
- рекомендации по проектированию исключений;
- рекомендации по использованию;
- общие паттерны проектирования.

Чтобы руководство было максимально простым для понимания, рекомендации обозначены просто терминами «делать», «учитывать», «избегать» и «не делать».

Компания Microsoft разместила выдержки из этой книги: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/.

Я настоятельно рекомендую вам просмотреть все рекомендации и применить их в своем коде.

Следующее издание

Я уже начал работу над седьмым изданием, которое мы планируем опубликовать с выходом .NET 7.0 в ноябре 2022 года. Я не ожидаю появления новых функций на уровне Blazor или .NET MAUI, однако рассчитываю, что .NET 7.0 внесет значительные улучшения во все аспекты .NET.

Если у вас есть предложения по темам, которые вы хотели бы рассмотреть или расширить, или вы заметили ошибки, которые нужно исправить в тексте или в коде, то, пожалуйста, сообщите мне подробности через репозитории GitHub для этой книги, который находится по следующей ссылке: https://github.com/markjprice/cs10dotnet6.

Желаю вам удачи во всех ваших проектах на C# и .NET!

Дополнительные материалы

Завершают книгу три дополнительные главы, посвященные созданию и использованию специализированных сервисов, .NET MAUI и защите ваших данных и приложений. Их вы сможете скачать в электронном виде по ссылке https://clck.ru/32EVw7.

Глава 18 «Создание и использование специализированных сервисов» рассказывает о создании сервисов с помощью gRPC, а также о том, как реализовывать взаимодействие между сервером и клиентом в режиме реального времени, используя SignalR. Кроме того, вы научитесь предоставлять данные в качестве веб-сервиса с помощью OData и размещать в облаке функции, реагирующие на триггеры, используя платформу Azure Functions.

Глава 19 «Разработка мобильных и настольных приложений с помощью .NET MAUI» знакомит вас с созданием кросс-платформенных мобильных и настольных приложений для Android, iOS, macOS и Windows. Вы изучите основы XAML, с помощью которых можно определять пользовательский интерфейс для графического приложения.

Глава 20 «Защита данных и приложений» посвящена тому, как с помощью шифрования защищать данные от просмотра злоумышленниками, а также тому, как применять хеширование и цифровые подписи для защиты данных от вмешательства или повреждения. Кроме того, вы узнаете, как с помощью аутентификации и авторизации защищать приложения от неавторизованных пользователей.

В приложении содержатся ответы на проверочные вопросы, приведенные в конце каждой главы.

18 Создание и использование специализированных сервисов

В этой главе вы познакомитесь с основами нескольких сервисных технологий, которые используются в более специализированных сценариях, чем веб-сервисы общего назначения. Как только вы поймете концепции и преимущества каждой из них, вы сможете углубиться в те из них, которые вас интересуют больше всего.

В этой главе:

- специализированные сервисные технологии;
- предоставление данных в качестве веб-сервиса с помощью OData;
- предоставление данных как сервиса с помощью GraphQL;
- реализация сервисов с помощью gRPC;
- реализация взаимодействия в режиме реального времени с помощью SignalR;
- реализация бессерверных сервисов с помощью платформы Azure Functions;
- сервисы идентификации;
- обзор вариантов специализированных сервисов.

Специализированные сервисные технологии

ASP.NET Core Web API часто лучше всего подходит для реализации веб-сервиса общего назначения, но это не единственная технология для реализации сервисов или обмена данными между компонентами распределенного приложения.

Мы не будем подробно описывать эти технологии, однако вам необходимо знать, как они функционируют и когда их следует использовать.

Windows Communication Foundation (WCF)

В 2006 году Microsoft представила релиз .NET Framework 3.0 с несколькими новыми базовыми платформами, одной из которых была *Windows Communication Foundation (WCF)*. Эта платформа абстрагировала реализацию бизнес-логики сервиса от инфраструктуры коммуникационных технологий, чтобы в будущем можно было легко перейти на альтернативный вариант или даже иметь несколько механизмов для связи с сервисом.

В WCF интенсивно применяется конфигурация XML для декларативного определения конечных точек, включая их адрес (address), привязку (binding) и контракт (contract). Это так называемые ABC конечных точек WCF. Разобравшись, как это сделать, вы поймете, что WCF — это эффективная и в то же время гибкая технология.

Компания Microsoft решила официально не переносить WCF на современную .NET, но существует проект OSS, принадлежащий сообществу и называемый *Core WCF*, которым управляет .NET Foundation. Если вам нужно перенести существующий сервис из .NET Framework на современную .NET или создать клиент для сервиса WCF, то можете использовать Core WCF. Имейте в виду: Core WCF никогда не станет полноценным портом, поскольку части WCF зависят от операционной системы Windows.

Такие технологии, как WCF, позволяют создавать распределенные приложения. Клиентское приложение может выполнять *удаленные вызовы процедур* (remote procedure calls, RPC) серверного приложения. Вместо того чтобы использовать порт WCF, мы могли бы задействовать альтернативную технологию RPC, например gRPC, которая рассматривается далее в этой главе.

Предоставление данных в виде веб-сервиса с помощью OData

Одним из самых распространенных применений веб-сервиса является предоставление базы данных клиентам, которые не понимают, как работать с собственной базой данных. Другим распространенным способом использования является предоставление упрощенного или абстрактного API, который предоставляет только аутентифицированный интерфейс для подмножества данных.

В главе 10 вы узнали, как создать модель EF Core, чтобы предоставить доступ к базе данных любому приложению .NET или сайту. Но как насчет приложений и сайтов, не относящихся к .NET? Я знаю, что это сложно представить, но не все разработчики используют .NET!

К счастью, все платформы разработки поддерживают HTTP, чтобы можно было вызывать веб-сервисы, а в ASP.NET Core есть пакет, позволяющий сделать это легко и просто с помощью стандарта OData.

OData

OData (Open Data Protocol, Открытый протокол данных) — это стандарт OASIS, одобренный ISO/IEC, который определяет набор рекомендаций по созданию и использованию RESTful API. Компания Microsoft создала его в 2007 году и выпустила версии 1.0, 2.0 и 3.0 в рамках своей программы Microsoft Open Specification Promise. Версия 4.0 была стандартизирована в OASIS и выпущена в 2014 году.

В ASP.NET Core OData используется OData версии 4.0.

OData основан на HTTP и имеет несколько конечных точек для поддержки различных версий и наборов сущностей.

В отличие от традиционных Web API, в которых сервис определяет все методы и возвращаемые данные, OData использует строки запросов, позволяя клиенту иметь больший контроль над возвращаемыми данными, и сводит к минимуму круговые задержки. Например, клиенту может понадобиться только два поля данных, `ProductName` и `Cost`, и соответствующий объект `Supplier`, причем только для продуктов, в которых `ProductName` содержит слово `burger`, а стоимость меньше 4,95, как показано ниже:

```
GET https://example.com/v1/products?$filter=contains(ProductName, 'burger') and
Cost lt 4.95&$orderby=Country,Cost&$select=ProductName,Cost&$expand=Supplier
```

Создание веб-сервиса, поддерживающего OData

Не существует шаблона `dotnet new` для ASP.NET Core OData, и, кроме того, в нем используются классы контроллера, поэтому мы будем использовать шаблон проекта Web API, а затем добавим ссылки на пакеты для поддержки OData.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: ASP.NET Core Web API/`webapi`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.OData`;
   4) другие параметры Visual Studio: Authentication Type: None (Тип аутентификации: Нет), флажок Configure for HTTPS (Настроить для HTTPS) установлен, флажок Enable Docker (Включить Docker) снят, флажок Enable OpenAPI support (Включить поддержку OpenAPI) установлен.

В программе Visual Studio Code выберите `Northwind.OData` в качестве активного проекта OmniSharp.

2. Добавьте ссылку на пакет для ASP.NET Core OData, как показано ниже:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.OData"
    Version="8.0.1" />
  <PackageReference Include="Swashbuckle.AspNetCore"
    Version="6.1.4" />
</ItemGroup>
```

Номера версий вышеуказанных пакетов NuGet, вероятно, изменятся в сторону увеличения после публикации книги. В качестве общего руководства рекомендуется использовать последнюю версию пакета.

3. Добавьте ссылку на проект контекста базы данных Northwind для SQLite или SQL Server:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include=
"..\Northwind.Common.DataContext.Sqlite\Northwind.Common.DataContext.
Sqlite.csproj" />
</ItemGroup>
```

4. В папке Northwind.OData удалите файл `WeatherForecast.cs`.
5. В папке Controllers удалите файл `WeatherForecastController.cs`.
6. В файле `Program.cs` настройте метод `UseUrls` на указание порта `5004` для HTTPS, как показано ниже (выделено жирным шрифтом):

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseUrls("https://localhost:5004");
```

7. Соберите проект `Northwind.OData`.

Определение моделей OData для моделей EF Core

Первая задача — определить, что мы хотим предоставить в веб-сервисе в качестве моделей OData. Все под вашим контролем, поэтому, если у вас есть существующая модель EF Core, как у нас для Northwind, вам не обязательно раскрывать ее всю. Вам даже не обязательно использовать модели EF Core. Источник данных может быть любым, хотя в этой книге мы рассмотрим его использование только в EF Core, поскольку это наиболее распространенный случай применения разработчиками .NET.

Определим две модели OData: одну для отображения каталога продукции Northwind, то есть таблиц категорий и продуктов, а другую — для отображения клиентов, их заказов и связанных таблиц.

1. В файле `Program.cs` импортируйте пространства имен для работы с OData и нашу модель EF Core для базы данных Northwind:

```
using Microsoft.AspNetCore.OData; // метод расширения AddOData
using Microsoft.OData.Edm; // IEdmModel
using Microsoft.OData.ModelBuilder; // ODataConventionModelBuilder
using Packt.Shared; // NorthwindContext и модели сущностей
```

2. В конце файла `Program.cs` добавьте метод для определения и возврата модели OData для каталога Northwind, которая будет отображать только наборы сущностей, то есть таблицы для категорий, товаров и поставщиков:

```
IEdmModel GetEdmModelForCatalog()
{
  ODataConventionModelBuilder builder = new();
  builder.EntitySet<Category>("Categories");
  builder.EntitySet<Product>("Products");
  builder.EntitySet<Supplier>("Suppliers");
  return builder.GetEdmModel();
}
```

3. Добавьте метод для определения модели OData для заказов клиентов Northwind и обратите внимание, что один и тот же набор сущностей может использоваться в нескольких моделях OData, как в случае с `Products`:

```
IEdmModel GetEdmModelForOrderSystem()
{
  ODataConventionModelBuilder builder = new();
  builder.EntitySet<Customer>("Customers");
  builder.EntitySet<Order>("Orders");
  builder.EntitySet<Employee>("Employees");
  builder.EntitySet<Product>("Products");
  builder.EntitySet<Shipper>("Shippers");
  return builder.GetEdmModel();
}
```

4. В разделе конфигурации служб после вызова метода `AddControllers` добавьте вызов метода расширения `AddOData`, чтобы определить две модели OData и использовать такие функции, как проекция, фильтрация и сортировка:

```
builder.Services.AddControllers()
  .AddOData(options => options
    // регистрация моделей OData различных версий
    .AddRouteComponents(routePrefix: "catalog",
      model: GetEdmModelForCatalog())
```

```
    .AddRouteComponents(routePrefix: "ordersystem",
      model: GetEdmModelForOrderSystem())

    // включение параметра запросов
    .Select() // включение $select для проекции
    .Expand() // включение $expand для навигации по сущностям
    .Filter() // включение $filter
    .OrderBy() // включение $orderby
    .SetMaxTop(100) // включение $top
    .Count() // включение $count
  );
```

5. Добавьте операторы перед вызовом метода `AddControllers`, чтобы зарегистрировать контекст базы данных `Northwind`:

```
builder.Services.AddNorthwindContext();
```

6. В папке Properties откройте файл `launchSettings.json`.
7. В профиле `Northwind.OData` измените `applicationUrl`, чтобы использовать порт `5004`, как показано ниже:

```
"applicationUrl": "https://localhost:5004",
```

Тестирование моделей OData

Теперь мы можем проверить, правильно ли определены модели OData.

1. Запустите веб-сервис `Northwind.OData`.
2. Запустите браузер Google Chrome.
3. Перейдите по адресу https://localhost:5004/swagger и обратите внимание, что сервис `Northwind.OData v1` задокументирован.
4. В разделе Metadata (Метаданные) выберите вариант GET/catalog (Получить/каталог), затем вариант Try it out (Попробовать), нажмите кнопку Execute (Выполнить) и обратите внимание на тело ответа, которое показывает имена и URL трех наборов сущностей в модели OData каталога:

```
{
  "@odata.context": "https://localhost:5004/catalog/$metadata",
  "value": [
    {
      "name": "Categories",
      "kind": "EntitySet",
      "url": "Categories"
    },
    {
      "name": "Products",
```

```
      "kind": "EntitySet",
      "url": "Products"
    },
    {
      "name": "Suppliers",
      "kind": "EntitySet",
      "url": "Suppliers"
    }
  ]
}
```

5. Щелкните на GET/catalog (Получить/каталог), чтобы свернуть этот раздел.
6. Выберите GET/catalog/$metadata (Получить/каталог/$метаданные), затем Try it out (Попробовать), нажмите кнопку Execute (Выполнить) и обратите внимание, что модель подробно описывает такие сущности, как `Category`, со свойствами и ключами, включая аспекты навигации для продуктов в каждой категории (рис. 18.1).

Рис. 18.1. Метаданные модели OData для каталога Northwind

7. Щелкните на GET/catalog/$metadata (Получить/каталог/$метаданные), чтобы свернуть этот раздел.
8. Закройте браузер Google Chrome и завершите работу веб-сервера.

Создание и тестирование контроллеров OData

Далее мы должны создать контроллеры OData, по одному для каждого типа сущностей, чтобы получить данные.

1. В папке Controllers добавьте пустой класс контроллера `CategoriesController`.
2. Измените его содержимое так, чтобы он наследовался от `ODataController`, получил экземпляр контекста базы данных Northwind с помощью введения параметров конструктора и определил два метода `Get` для извлечения всех категорий или одной категории по уникальному ключу:

```
using Microsoft.AspNetCore.Mvc; // IActionResult
using Microsoft.AspNetCore.OData.Query; // [EnableQuery]
using Microsoft.AspNetCore.OData.Routing.Controllers; // ODataController
using Packt.Shared; // NorthwindContext

namespace Northwind.OData.Controllers;

public class CategoriesController : ODataController
{
  private readonly NorthwindContext db;

  public CategoriesController(NorthwindContext db)
  {
    this.db = db;
  }

  [EnableQuery]
  public IActionResult Get()
  {
    return Ok(db.Categories);
  }

  [EnableQuery]
  public IActionResult Get(int key)
  {
    return Ok(db.Categories.Find(key));
  }
}
```

3. Повторите описанный выше шаг для `Products` и `Suppliers`. (На ваше усмотрение вы можете сделать то же самое для других сущностей, чтобы включить модель OData системы заказов. Обратите внимание, что `CustomerId` — это строка, а не целое число.)
4. Запустите веб-сервис `Northwind.OData`.
5. Запустите браузер Google Chrome.

6. Перейдите по адресу https://localhost:5004/swagger и обратите внимание, что наборы сущностей Categories, Products и Suppliers теперь можно использовать, поскольку вы создали для них контроллеры OData.
7. Выберите пункт GET/catalog/Categories (Получить/каталог/Категории), затем Try it out (Попробовать), нажмите кнопку Execute (Выполнить) и обратите внимание на тело ответа в формате JSON, содержащее все категории в наборе сущностей, как частично показано в этом выводе:

```
{
  "@odata.context": "https://localhost:5004/catalog/$metadata#Categories",
  "value": [
    {
      "CategoryId": 1,
      "CategoryName": "Beverages",
      "Description": "Soft drinks, coffees, teas, beers, and ales",
      "Picture": null
    },
    {
      "CategoryId": 2,
      "CategoryName": "Condiments",
      "Description": "Sweet and savory sauces, relishes, spreads,
      and seasonings",
      "Picture": null
    },
    ...
  ]
}
```

8. Закройте браузер Google Chrome и завершите работу веб-сервера.

Тестирование контроллеров OData с помощью REST Client

Использование пользовательского интерфейса Swagger для тестирования контроллеров OData может быстро стать неудобным. Лучшим инструментом является расширение для программы Visual Studio Code под названием REST Client.

1. Если вы еще не установили REST Client разработчика Huachao Mao (`humao.rest-client`), то установите его в программе Visual Studio Code прямо сейчас.
2. Откройте редактор кода и запустите веб-сервис проекта `Northwind.OData`.
3. В программе Visual Studio Code в каталоге PracticalApps создайте папку RestClientTests, если она еще не существует, а затем откройте ее.
4. В папке RestClientTests создайте файл `odata-catalog.http` и укажите в нем запрос на получение всех категорий:

```
GET https://localhost:5004/catalog/categories/ HTTP/1.1
```

5. Нажмите кнопку Send Request (Отправить запрос) и обратите внимание, что ответ совпадает с тем, который был возвращен Swagger, — документ JSON, содержащий все категории.
6. В файле `odata-catalog.http` добавьте дополнительные запросы, разделенные `###`, как показано в табл. 18.1.

Таблица 18.1. Дополнительные запросы

Запрос	Ответ
`https://localhost:5004/catalog/categories(3)`	`{ "@odata.context": "https://localhost:5004/catalog/$metadata#Categories/$entity", "CategoryId":3, "CategoryName":"Confections", "Description":"Desserts, candies, and sweet breads", "Picture":null }`
`https://localhost:5004/catalog/categories/3`	Аналогично тому, как указано выше
`https://localhost:5004/catalog/categories/$count`	`8`
`https://localhost:5004/catalog/products`	Документ JSON, содержащий все продукты
`https://localhost:5004/catalog/products/$count`	`77`
`https://localhost:5004/catalog/products(2)`	`{ "@odata.context": "https://localhost:5004/catalog/$metadata#Products/$entity", "ProductId": 2, "ProductName": "Chang", "SupplierId": 1, "CategoryId": 1, "QuantityPerUnit": "24 - 12 oz bottles", "UnitPrice": 19, "UnitsInStock": 17, "UnitsOnOrder": 40, "ReorderLevel": 25, "Discontinued": false }`
`https://localhost:5004/catalog/suppliers`	Документ JSON, содержащий данные обо всех поставщиках
`https://localhost:5004/catalog/suppliers/$count`	`29`

Запрос моделей OData

Для выполнения произвольных запросов к модели OData мы ранее включили выборку, фильтрацию и упорядочение. Одним из преимуществ OData является то, что в нем определены стандартные параметры запросов, как показано в табл. 18.2.

Таблица 18.2. Стандартные параметры запросов

Опция	Описание	Пример
`$select`	Выбирает свойства для каждой сущности	`$select=CategoryId,CategoryName`
`$expand`	Выбирает связанные сущности через навигационные свойства	`$expand=Products`
`$filter`	Выражение оценивается для каждого ресурса, и в ответ включаются только те объекты, для которых выражение истинно	`$filter=startswith(ProductName,'ch') or (UnitPrice gt 50)`
`$orderby`	Сортирует сущности по перечисленным свойствам в порядке возрастания (по умолчанию) или убывания	`$orderby=UnitPrice desc,ProductName`
`$top`	Пропускает указанное количество элементов. Берет указанное количество элементов	`$skip=40&$take=10`

По соображениям производительности пакетная обработка с помощью `$skip` и `$top` по умолчанию отключена.

Операторы OData

В OData есть операторы для использования с параметром `$filter`, как показано в табл. 18.3.

Таблица 18.3. Операторы для использования с параметром $filter

Операция	Описание
`eq`	Равно
`ne`	Не равно
`lt`	Меньше
`gt`	Больше
`le`	Меньше или равно
`ge`	Больше или равно

Продолжение ⇨

Таблица 18.3 *(продолжение)*

Операция	Описание
`and`	И
`or`	Или
`not`	Не
`add`	Арифметическое сложение для чисел и значений даты/времени
`sub`	Арифметическое вычитание для чисел и значений даты/времени
`mul`	Арифметическое вычитание для чисел
`div`	Арифметическое деление для чисел
`mod`	Арифметическое деление по модулю для чисел

Функции OData

В OData есть функции для использования с параметром `$filter`, как показано в табл. 18.4.

Таблица 18.4. Функции для использования с параметром $filter

Операция (-и)	Описание
`startswith(property, 'value')`	Текстовые значения, которые начинаются с указанного значения
`endswith(property, 'value')`	Текстовые значения, которые заканчиваются указанным значением
`concat(property, 'value')`	Конкатенация двух текстовых значений
`contains(property, 'value')`	Текстовые значения, содержащие указанное значение
`indexof(property, 'value')`	Возвращает позицию текстового значения
`length(property)`	Возвращает длину текстового значения
`substring`	Извлекает подстроку из текстового значения
`tolower`	Преобразует в строчные буквы
`toupper`	Преобразует в верхний регистр
`trim`	Обрезает пробелы до и после текстового значения
`now`	Текущая дата и время
`date`, `day`, `month`, `year`	Извлекает компоненты даты
`time`, `hour`, `minute`, `second`	Извлекает компоненты времени

Обзор запросов OData

Поэкспериментируем с некоторыми запросами OData.

1. В папке RestClientTests создайте файл `odata-catalog-queries.http` и добавьте в него запрос на получение всех категорий:

```
GET https://localhost:5004/catalog/categories/
  ?$select=CategoryId,CategoryName
```

2. Нажмите кнопку Send Request (Отправить запрос) и обратите внимание на ответ — документ JSON, содержащий все категории со свойствами `CategoryId` и `CategoryName`.
3. В файле `odata-catalog-queries.http` добавьте запрос на получение продуктов с названиями, начинающимися на Ch, таких как `Chai` и `Chef Anton's Gumbo Mix`, или имеющими цену за единицу более 50, например, `Mishi Kobe Niku` или `Sir Rodney's Marmalade`, как показано ниже:

```
GET https://localhost:5004/catalog/products/
  ?$filter=startswith(ProductName,'Ch') or (UnitPrice gt 50)
```

4. В файле `odata-catalog-queries.http` добавьте запрос для получения продуктов, отсортированных по цене, с самыми дорогими вверху, а затем отсортированных по названию продукта, и включите только свойства `ID`, названия и цены, как показано ниже:

```
GET https://localhost:5004/catalog/products/
  ?$orderby=UnitPrice desc,ProductName
  &$select=ProductId,ProductName,UnitPrice
```

5. В файле `odata-catalog-queries.http` добавьте запрос для получения категорий и связанных с ними продуктов:

```
GET https://localhost:5004/catalog/categories/
?$select=CategoryId,CategoryName
&$expand=Products
```

Логирование запросов OData

Как работают запросы OData? Выясним это, добавив функцию записи событий в контекст базы данных Northwind, чтобы увидеть реальные SQL-запросы, которые выполняются.

1. В проекте `Northwind.Common.DataContext.Sqlite` (и `SqlServer`) добавьте файл `ConsoleLogger.cs`.

2. Определите в файле три класса, один для реализации `ILoggerFactory`, один для реализации `ILoggerProvider` и один для реализации `ILogger`:

```
using Microsoft.Extensions.Logging;

using static System.Console;

namespace Packt.Shared;

public class ConsoleLoggerFactory : ILoggerFactory
{
  public void AddProvider(ILoggerProvider provider) { }

  public ILogger CreateLogger(string categoryName)
  {
    return new ConsoleLogger();
  }

  public void Dispose() { }
}

public class ConsoleLogger : ILogger
{
  public IDisposable BeginScope<TState>(TState state)
  {
    return null;
  }

  public bool IsEnabled(LogLevel logLevel)
  {
    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)
  {
    if (eventId.Id == 20100) // выполнить оператор SQL
    {
      Write($"Level: {logLevel}, Event Id: {eventId.Id}");
```

```
        // выводим состояние или исключение, только если оно существует
        if (state != null)
        {
          Write($", State: {state}");
        }

        if (exception != null)
        {
          Write($", Exception: {exception.Message}");
        }
        WriteLine();
      }
    }
  }
```

3. В файле `NorthwindContextExtensions.cs` в методе `AddNorthwindContext` после вызова `UseSqlServer` или `UseSqlite` вызовите `UseLoggerFactory` для регистрации вашего пользовательского консольного логера (средства ведения журнала), как показано ниже:

```
services.AddDbContext<NorthwindContext>(options =>
  options.UseSqlServer(connectionString) // or UseSqlite(...)
  .UseLoggerFactory(new ConsoleLoggerFactory())
);
```

4. Запустите веб-сервис `Northwind.OData`.
5. Запустите браузер Google Chrome.
6. Перейдите по адресу https://localhost:5004/catalog/products/?$filter=startswith(ProductName,'Ch') или (UnitPrice gt 50)&$select=ProductId,ProductName,UnitPrice.
7. В браузере Google Chrome обратите внимание на результат:

```
{"@odata.context":"https://localhost:5004/catalog/$metadata#Products
(ProductId,ProductName,UnitPrice)","value":[{"ProductId":1,"ProductName":
"Chai","UnitPrice":18.0000},{"ProductId":2,"ProductName":"Chang",
"UnitPrice":19.0000},{"ProductId":4,"ProductName":"Chef Anton's Cajun
Seasoning","UnitPrice":22.0000},{"ProductId":5,"ProductName":"Chef
Anton's Gumbo Mix","UnitPrice":21.3500},{"ProductId":9,"ProductName":
"Mishi Kobe Niku","UnitPrice":97.0000},{"ProductId":18,"ProductName":
"Carnarvon Tigers","UnitPrice":62.5000},{"ProductId":20,"ProductName":
"Sir Rodney's Marmalade","UnitPrice":81.0000},{"ProductId":29,
"ProductName":"Th\u00fcringer Rostbratwurst","UnitPrice":123.7900},
{"ProductId":38,"ProductName":"C\u00f4te deBlaye","UnitPrice":263.5000},
{"ProductId":39,"ProductName":"Chartreuse verte","UnitPrice":18.0000},
{"ProductId":48,"ProductName":"Chocolade","UnitPrice":12.7500},
{"ProductId":51,"ProductName":"Manjimup Dried Apples","UnitPrice":
53.0000},{"ProductId":59,"ProductName":"Raclette Courdavault",
"UnitPrice":55.0000}]}
```

8. В командной строке или в терминале обратите внимание на зарегистрированный оператор SQL, например, при использовании поставщика базы данных SQL Server:

```
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=
[@__TypedProperty_0='?' (Size = 4000), @__TypedProperty_1='?' (DbType =
Decimal)], CommandType='Text', CommandTimeout='30']
SELECT [p].[ProductId], [p].[ProductName], [p].[UnitPrice]
FROM [Products] AS [p]
WHERE ((@__TypedProperty_0 = N'') OR (LEFT([p].[ProductName],
LEN(@__TypedProperty_0)) = @__TypedProperty_0)) OR ([p].[UnitPrice] >
@__TypedProperty_1)
```

Может показаться, что метод действия Get в ProductsController возвращает всю таблицу Products, но на самом деле он возвращает объект IQueryable<Products>. Другими словами, он возвращает запрос LINQ, а не его результаты. Мы дополнили метод действия Get атрибутом [EnableQuery]. Это позволяет OData расширить запрос LINQ фильтрами, проекциями, сортировкой и т. д., и только после этого он выполняет запрос, сериализует результаты и возвращает их клиенту. Это делает сервисы OData максимально гибкими и эффективными.

Контроль версий контроллеров OData

Рекомендуется планировать будущие версии ваших моделей OData, которые могут иметь различные схемы и поведение.

Чтобы сохранить обратную совместимость, вы можете использовать префиксы URL OData для указания номера версии.

1. В проекте `Northwind.OData` в файле `Program.cs` в разделе конфигурации сервисов после добавления двух моделей OData для `catalog` и `ordersystem` добавьте третью модель OData, которая имеет номер версии:

```
.AddRouteComponents(routePrefix: "catalog",
  model: GetEdmModelForCatalog())
.AddRouteComponents(routePrefix: "ordersystem",
  model: GetEdmModelForOrderSystem())
.AddRouteComponents(routePrefix: "v{version}",
  model: GetEdmModelForCatalog())
```

2. В файле `ProductsController.cs` статически импортируйте `Console`, а затем измените методы `Get`, добавив строковый параметр `version`, и используйте его для изменения поведения методов, если в запросе указана версия 2, как показано ниже:

```
[EnableQuery]
public IActionResult Get(string version = "1")
{
  WriteLine($"ProductsController version {version}.");
  return Ok(db.Products);
}

[EnableQuery]
public IActionResult Get(int key, string version = "1")
{
  WriteLine($"ProductsController version {version}.");
  Product? p = db.Products.Find(key);
  if (p is null)
  {
    return NotFound($"Product with id {key} not found.");
  }
  if (version == "2")
  {
    p.ProductName += " version 2.0";
  }
  return Ok(p);
}
```

3. Откройте редактор кода и запустите веб-сервис проекта `Northwind.OData`.
4. В программе Visual Studio Code в файле `odata-catalog-queries.http` добавьте запрос на получение продукта с ID `50` с помощью модели v2 OData:

```
GET https://localhost:5004/v2/products(50)
```

5. Нажмите кнопку Send Request (Отправить запрос) и обратите внимание, что в ответ будет получен продукт, к названию которого добавлено `version 2.0`, как показано ниже:

```
{
  "@odata.context": "https://localhost:5004/v2/$metadata#Products/
  $entity",
  "ProductId": 50,
  "ProductName": "Valkoinen suklaa version 2.0",
  "SupplierId": 23,
  "CategoryId": 3,
  "QuantityPerUnit": "12 - 100 g bars",
  "UnitPrice": 16.2500,
  "UnitsInStock": 65,
  "UnitsOnOrder": 0,
  "ReorderLevel": 30,
  "Discontinued": false
}
```

Добавление сущностей с помощью POST

Чаще всего с помощью OData предоставляется Web API, поддерживающий пользовательские запросы. Вы также можете поддерживать операции CRUD, такие как вставка.

1. В файле `ProductsController.cs` добавьте метод действия для ответа на `POST`-запросы:

```
public IActionResult Post([FromBody] Product product)
{
  db.Products.Add(product);
  db.SaveChanges();
  return Created(product);
}
```

2. Запустите веб-сервис.
3. Создайте файл `odata-catalog-insert-product.http`, как показано в следующем HTTP-запросе:

```
POST https://localhost:5004/catalog/products
Content-Type: application/json
Content-Length: 234

{
  "ProductName": "Impossible Burger",
  "SupplierId": 7,
  "CategoryId": 6,
  "QuantityPerUnit": "Pack of 4",
  "UnitPrice": 40.25,
  "UnitsInStock": 50,
  "UnitsOnOrder": 0,
  "ReorderLevel": 30,
  "Discontinued": false
}
```

4. Нажмите кнопку Send Request (Отправить запрос).
5. Обратите внимание на успешный ответ:

```
HTTP/1.1 201 Created
Connection: close
Content-Type: application/json; odata.metadata=minimal; odata.
streaming=true
Date: Sat, 17 Jul 2021 12:01:57 GMT
Server: Kestrel
Location: https://localhost:5004/catalog/Products(80)
```

```
Transfer-Encoding: chunked
OData-Version: 4.0

{
  "@odata.context": "https://localhost:5004/catalog/$metadata
  #Products/$entity",
  "ProductId": 78,
  "ProductName": "Impossible Burger",
  "SupplierId": 7,
  "CategoryId": 6,
  "QuantityPerUnit": "Pack of 4",
  "UnitPrice": 40.25,
  "UnitsInStock": 50,
  "UnitsOnOrder": 0,
  "ReorderLevel": 30,
  "Discontinued": false
}
```

Создание клиента для OData

В этом подразделе рассмотрим, как клиент может вызывать веб-сервис OData.

Если мы хотим запросить у сервиса OData продукты, начинающиеся с букв `Cha`, то нам нужно отправить запрос `GET` с относительным URL, подобным приведенному ниже:

```
catalog/products/?$filter=startswith(ProductName, 'Cha')&$select=ProductId,
ProductName,UnitPrice
```

OData возвращает данные в формате JSON со свойством `value`, которое содержит полученные продукты в виде массива, как показано в следующем документе JSON:

```
{
  "@odata.context": "https://localhost:5004/catalog/$metadata#Products",
  "value": [
    {
      "ProductId": 1,
      "ProductName": "Chai",
      "SupplierId": 1,
      "CategoryId": 1,
      "QuantityPerUnit": "10 boxes x 20 bags",
      "UnitPrice": 18,
      "UnitsInStock": 39,
      "UnitsOnOrder": 0,
      "ReorderLevel": 10,
      "Discontinued": false
    },
```

Мы создадим класс модели, чтобы облегчить десериализацию ответа.

1. В проекте `Northwind.Mvc` в папке Models создайте файл класса `ODataProducts.cs`:

```
using Packt.Shared; // Product

namespace Northwind.Mvc.Models;

public class ODataProducts
{
  public Product[]? Value { get; set; }
}
```

2. В файле `Program.cs` добавьте операторы для регистрации HTTP-клиента для сервиса OData:

```
builder.Services.AddHttpClient(name: "Northwind.OData",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5004/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

Добавление страницы сервисов на сайт Northwind MVC

Далее мы создадим страницу сервисов.

1. В папке Controllers откройте файл `HomeController.cs` и добавьте новый метод действия для сервисов, который вызывает сервис OData для получения продуктов, начинающихся на `Cha`, и сохраняет результат в словаре `ViewData`:

```
public async Task<IActionResult> Services()
{
  try
  {
    HttpClient client = clientFactory.CreateClient(
      name: "Northwind.OData");

    HttpRequestMessage request = new(
      method: HttpMethod.Get, requestUri:
      "catalog/products/?$filter=startswith(ProductName, 'Cha')&$select=
      ProductId,ProductName,UnitPrice");

    HttpResponseMessage response = await client.SendAsync(request);
      ViewData["productsCha"] = (await response.Content
        .ReadFromJsonAsync<ODataProducts>())?.Value;
    }
    catch (Exception ex)
```

```
    {
      _logger.LogWarning($"Northwind.OData service exception: {ex.Message}");
    }

    return View();
  }
```

2. В разделе `Views/Shared` в файле `_Layout.cshtml` после элемента навигации `Privacy` добавьте новый элемент навигации, который ведет на страницу сервисов:

```
<li class="nav-item">
  <a class="nav-link text-dark" asp-area=""
    asp-controller="Home" asp-action="Services">Services</a>
</li>
```

3. В папке Views/Home добавьте новое пустое представление `Services.cshtml` и измените его содержимое для отображения продуктов, как показано ниже:

```
@using Packt.Shared
@using Northwind.Common
@{
  ViewData["Title"] = "Services";
  Product[]? products = ViewData["productsCha"] as Product[];
}
<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>
  @if (ViewData["productsCha"] != null)
  {
    <h2>Products that start with Cha using OData</h2>
    <p>
      @if (products is null)
      {
        <span class="badge badge-info">No products found.</span>
      }
      else
      {
        @foreach (Product p in products)
        {
          <span class="badge bg-info">
            @p.ProductId
            @p.ProductName
            @(p.UnitPrice is null ? "" : p.UnitPrice.Value.ToString("c"))
          </span>
        }
      }
    </p>
  }
</div>
```

4. При необходимости запустите проект `Minimal.Web` без отладки. (Если вы используете Visual Studio 2022, то выберите проект на панели Solution Explorer (Проводник решений), чтобы сделать его текущим выбором, а затем перейдите в Debug ▸ Start Without Debugging (Запуск ▸ Запуск без отладки).)
5. Запустите проект `Northwind.OData` без отладки.
6. Запустите проект `Northwind.Mvc` без отладки.
7. Запустите браузер Google Chrome.
8. Перейдите на страницу Services (Сервисы), нажав меню или перейдя по следующей ссылке: https://localhost:5001/home/services.
9. Обратите внимание, что из сервиса OData возвращаются три продукта, как показано на рис. 18.2.

Рис. 18.2. Три названия продуктов начинаются с Cha, полученного из сервиса OData

10. Закройте браузер Google Chrome и завершите работу всех веб-серверов.

Предоставление данных как сервиса с помощью GraphQL

Если вы предпочитаете использовать более современную технологию для предоставления данных как сервиса, то альтернативой OData является GraphQL.

GraphQL

Как и OData, GraphQL — это стандарт для описания ваших данных и последующего запроса к ним, который позволяет клиенту контролировать именно то, что ему нужно. GraphQL был разработан компанией Facebook в 2012 году, а в 2015 году

стал проектом с открытым исходным кодом и в настоящее время управляется GraphQL Foundation.

Некоторые преимущества GraphQL перед OData заключаются в том, что он не требует HTTP, поскольку является транспортно-агностическим (независимым), поэтому вы можете использовать альтернативные транспортные протоколы, такие как WebSockets, а GraphQL имеет единственную конечную точку, обычно просто `/graphql`.

GraphQL использует собственный формат запросов, который немного похож на JSON, но запросы GraphQL не требуют запятых между именами полей, как показано ниже:

```
{
  product (productId: 23) {
    productId
    productName
    cost
    supplier {
      companyName
      country
    }
  }
}
```

Официальным типом данных для документов запросов GraphQL является application/graphql.

Создание сервиса, поддерживающего GraphQL

Для GraphQL не существует шаблона проекта `dotnet new`, поэтому мы будем использовать шаблон проекта Web API (хотя GraphQL не обязательно должен быть размещен в веб-сервисе), а затем добавим ссылки на пакеты для поддержки GraphQL.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: ASP.NET Core Web API/`webapi`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.GraphQL`;
   4) другие параметры Visual Studio: Authentication Type: None (Тип аутентификации: Нет), флажок Configure for HTTPS (Настроить для HTTPS) установлен, флажок Enable Docker (Включить Docker) снят, флажок Enable OpenAPI support (Включить поддержку OpenAPI) снят.

Поскольку GraphQL не является традиционным сервисом Web API, Swagger должен быть выключен. Если вы используете командную строку, то добавьте следующий переключатель: dotnet new webapi --no-openapi.

2. В программе Visual Studio Code выберите `Northwind.GraphQL` в качестве активного проекта OmniSharp.
3. Добавьте ссылки на пакеты для основных компонентов сервера GraphQL и пользовательского интерфейса GraphQL Playground, как показано ниже:

```
<ItemGroup>
  <PackageReference
    Include="GraphQL.Server.Transports.AspNetCore"
    Version="5.0.2" />
  <PackageReference
    Include="GraphQL.Server.Transports.AspNetCore.SystemTextJson"
    Version="5.0.2" />
  <PackageReference
    Include="GraphQL.Server.Ui.Playground"
    Version="5.0.2" />
</ItemGroup>
```

Удалите пакет пользовательского интерфейса GraphQL Playground перед развертыванием сервиса в рабочей среде. Хотя вы можете применять Playground только в режиме разработки, любые неиспользуемые пакеты увеличивают потенциальную вероятность атаки злоумышленников.

4. Добавьте ссылку на проект контекста базы данных Northwind для SQLite или SQL Server:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include=
"..\Northwind.Common.DataContext.Sqlite\Northwind.Common.DataContext.
Sqlite.csproj" />
</ItemGroup>
```

5. В папке Northwind.GraphQL удалите файл `WeatherForecast.cs`.
6. В папке Controllers удалите файл `WeatherForecastController.cs`.
7. Удалите папку Controllers.
8. В файле `Program.cs` добавьте вызов метода расширения `UseUrls`, чтобы указать порт `5005` для HTTPS:

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseUrls("https://localhost:5005/");
```

9. В папке Properties откройте файл `launchSettings.json` и измените параметры `launchUrl` и `applicationUrl`:

```
"profiles": {
  "Northwind.GraphQL": {
    "commandName": "Project",
    "dotnetRunMessages": "true",
    "launchBrowser": true,
    "launchUrl": "ui/playground",
    "applicationUrl": "https://localhost:5005",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
```

10. Соберите проект `Northwind.GraphQL`.

Определение схемы GraphQL для Hello, World!

Первая задача — определить, что мы хотим представить в виде моделей GraphQL в веб-сервисе.

Определим модель GraphQL для самого простого примера — Hello, World!

1. В проект/папку `Northwind.GraphQL` добавьте файл класса `GreetQuery.cs`.
2. Измените класс, чтобы определить тип графа объектов `greet`, который в качестве ответа выдает обычный текст `"Hello, World!"`:

```
using GraphQL.Types; // ObjectGraphType

namespace Northwind.GraphQL;

public class GreetQuery : ObjectGraphType
{
  public GreetQuery()
  {
    Field<StringGraphType>(name: "greet",
      description: "A query type that greets the world.",
      resolve: context => "Hello, World!");
  }
}
```

3. В проект/папку `Northwind.GraphQL` добавьте файл класса `NorthwindSchema.cs`.
4. Измените класс, чтобы определить схему графа объектов, которая регистрирует класс `GreetQuery` как единственный тип запроса:

```
using GraphQL.Types; // схема

namespace Northwind.GraphQL;
```

```
public class NorthwindSchema : Schema
{
  public NorthwindSchema(IServiceProvider provider) : base(provider)
  {
    Query = new GreetQuery();
  }
}
```

Используйте внедрение параметров конструктора, чтобы получить IServiceProvider, который будет использоваться для получения зарегистрированных сервисов зависимостей.

5. В файле `Program.cs` импортируйте пространство имен для работы с GraphQL, а также запрос и схему GraphQL, которые вы только что определили:

```
using GraphQL.Server; // GraphQLOptions
using Northwind.GraphQL; // GreetQuery, NorthwindSchema
```

6. В разделе конфигурации сервисов после вызова метода `AddControllers` добавьте операторы для регистрации класса схемы в качестве сервиса зависимостей и для добавления поддержки GraphQL:

```
builder.Services.AddScoped<NorthwindSchema>();

builder.Services.AddGraphQL()
  .AddGraphTypes(typeof(NorthwindSchema), ServiceLifetime.Scoped)
  .AddDataLoader()
  .AddSystemTextJson(); // сериализовация ответов в формате JSON
```

Позже контекст базы данных Northwind будет зарегистрирован как scoped dependency service (сервис зависимостей с ограниченной областью действий), поэтому все сервисы, использующие его, также должны быть зарегистрированы как scoped, а не singleton.

7. В разделе настройки конвейера HTTP добавьте операторы для использования GraphQL со схемой Northwind и пользовательского интерфейса Playground, если вы находитесь в режиме разработки:

```
if (builder.Environment.IsDevelopment())
{
  app.UseGraphQLPlayground(); // путь по умолчанию: /ui/playground
}

app.UseGraphQL<NorthwindSchema>(); // путь по умолчанию: /graphql

app.UseHttpsRedirection();
```

8. Запустите проект сервиса `Northwind.GraphQL`.
9. Если вы используете Visual Studio Code, то запустите браузер Google Chrome и перейдите по адресу https://localhost:5005/ui/playground.
10. В левой части напишите безымянный запрос для запроса `greet`, как показано ниже:

```
{
  greet
}
```

11. Нажмите круглую серую кнопку воспроизведения и обратите внимание на URL конечной точки, используемый Playground, https://localhost:5005/graphql, и ответ, как показано в коде ниже и на рис. 18.3:

```
{
  "data": {
    "greet": "Hello, World!"
  },
  "extensions": {}
}
```

Рис. 18.3. Использование GraphQL Playground для выполнения приветственного запроса

12. Закройте браузер Google Chrome и завершите работу веб-сервера.

Мы также можем создать именованный запрос:

```
query QueryNameGoesHere {
  greet
}
```

Именованные запросы позволяют клиентам идентифицировать запросы и ответы в целях телеметрии, например, при размещении в облачных сервисах Microsoft Azure и мониторинге с помощью Application Insights.

Определение схемы GraphQL для моделей EF Core

Теперь, когда базовый сервис GraphQL успешно работает, добавим типы, чтобы обеспечить возможность запросов к базе данных Northwind.

1. В файле `Program.cs` импортируйте пространство имен для работы с моделью EF Core для базы данных Northwind:

```
using Packt.Shared; // метод AddNorthwindContext
```

2. Добавьте оператор в верхнюю часть раздела конфигурации сервисов, чтобы зарегистрировать класс контекста базы данных Northwind:

```
builder.Services.AddNorthwindContext();
```

3. В проект/папку `Northwind.GraphQL` добавьте файл класса `CategoryType.cs`. Он используется для описания класса сущности `Category` в системе GraphQL.
4. Измените класс `CategoryType`, чтобы определить тип графа объектов, соответствующий структуре модели сущности `Category`:

```
using GraphQL.Types; // ObjectGraphType<T>, ListGraphType<T>
using Packt.Shared; // Category

namespace Northwind.GraphQL;

public class CategoryType : ObjectGraphType<Category>
{
  public CategoryType()
  {
    Name = "Category";
    Field(c => c.CategoryId).Description("Id of the category.");
    Field(c => c.CategoryName).Description("Name of the category.");
    Field(c => c.Description).Description("Description of the category.");
    Field(c => c.Products, type: typeof(ListGraphType<ProductType>))
      .Description("Products in the category.");
  }
}
```

На данном этапе класс ProductType выдаст ошибку, поскольку мы его еще не создали.

5. В проект/папку `Northwind.GraphQL` добавьте файл класса `ProductType.cs`.
6. Определите в классе тип графа объектов, который соответствует структуре модели сущности `Product`:

```
using GraphQL.Types; // ObjectGraphType<T>, IntGraphType, DecimalGraphType
using Packt.Shared; // Category, Product

namespace Northwind.GraphQL;
```

```
public class ProductType : ObjectGraphType<Product>
{
  public ProductType()
  {
    Name = "Product";
    Field(p => p.ProductId).Description("Id of the product.");
    Field(p => p.ProductName).Description("Name of the product.");
    Field(p => p.CategoryId, type: typeof(IntGraphType))
      .Description("CategoryId of the product.");
    Field(p => p.Category, type: typeof(CategoryType))
      .Description("Category of the product.");
    Field(p => p.UnitPrice, type: typeof(DecimalGraphType))
      .Description("Unit price of the product.");
    Field(p => p.UnitsInStock, type: typeof(IntGraphType))
      .Description("Units in stock of the product.");
    Field(p => p.UnitsOnOrder, type: typeof(IntGraphType))
      .Description("Units on order of the product.");
  }
}
```

Простые типы, такие как int и string, автоматически распознаются GraphQL. Нулевые типы, такие как int? и decimal?, используемые свойствами CategoryId и UnitPrice, сложные типы, такие как Category, используемые свойством Category, и небольшие целые числа, такие как short, должны быть явно указаны, иначе GraphQL будет выбрасывать исключения во время выполнения.

7. В проект/папку `Northwind.GraphQL` добавьте файл класса `NorthwindQuery.cs`.
8. Измените класс, чтобы определить тип графа объектов, который имеет три типа запросов для возврата списка категорий, одной категории и товаров для категории:

```
using GraphQL; // метод расширения GetArgument
using GraphQL.Types; // ObjectGraphType, QueryArguments, QueryArgument<T>
using Microsoft.EntityFrameworkCore; // включить метод расширения
using Packt.Shared; // NorthwindContext

namespace Northwind.GraphQL;

public class NorthwindQuery : ObjectGraphType
{
  public NorthwindQuery(NorthwindContext db)
  {
    Field<ListGraphType<CategoryType>>(
      name: "categories",
      description: "A query type that returns a list of all categories.",
      resolve: context => db.Categories.Include(c => c.Products)
    );

    Field<CategoryType>(
```

```
        name: "category",
        description: "A query type that returns a category using its Id.",
        arguments: new QueryArguments(
          new QueryArgument<IntGraphType> { Name = "categoryId" }),
        resolve: context =>
        {
          Category? category = db.Categories.Find(
            context.GetArgument<int>("categoryId"));
          db.Entry(category).Collection(c => c.Products).Load();
          return category;
        }
      );

      Field<ListGraphType<ProductType>>(
        name: "products",
        arguments: new QueryArguments(
          new QueryArgument<IntGraphType> { Name = "categoryId" }),
        resolve: context =>
        {
          Category? category = db.Categories.Find(
            context.GetArgument<int>("categoryId"));
          db.Entry(category).Collection(c => c.Products).Load();
          return category.Products;
        }
      );
    }
  }
```

9. В файле `NorthwindSchema.cs` импортируйте пространство имен для получения сервиса с внедрением зависимостей и для контекста базы данных Northwind:

```
using Packt.Shared; // NorthwindContext
using Microsoft.Extensions.DependencyInjection; // GetRequiredService
```

10. В конструкторе закомментируйте оператор, который задает для `Query` использование `GreetQuery`, и добавьте оператор, который задает для него использование `NorthwindQuery`, получая и передавая требуемый контекст базы данных Northwind:

```
// Query = new GreetQuery();
Query = new NorthwindQuery(provider.GetRequiredService<NorthwindContext>());
```

Изучение запросов GraphQL с помощью Northwind

Теперь мы можем протестировать написание запросов GraphQL для базы данных Northwind.

1. Запустите проект сервиса `Northwind.GraphQL`.
2. Если вы работаете в программе Visual Studio Code, то запустите браузер Google Chrome и перейдите по адресу https://localhost:5005/ui/playground.

3. На Playground перейдите на вкладку Schema (Схема) с правой стороны и обратите внимание на определения схемы, запросов и типов, а также на подсказку IntelliSense, предоставляемую при вводе запроса (рис. 18.4).

Рис. 18.4. Схема для запроса категорий и продуктов Northwind с помощью GraphQL

4. Перейдите на вкладку Docs (Документы), а затем щелкните на categories (категории), category(...) (категория(...)) и products(...) (продукты(...)), чтобы просмотреть документацию.
5. Снова перейдите на вкладку Docs (Документы), чтобы свернуть панель.
6. В левой части введите именованный запрос, чтобы запросить все категории, как показано ниже:

```
query AllCategories {
  categories {
    categoryId
    categoryName
    description
  }
}
```

7. Нажмите кнопку воспроизведения и обратите внимание на ответ, показанный в следующем частичном выводе:

```
{
  "data": {
    "categories": [
      {
        "categoryId": 1,
```

```
        "categoryName": "Beverages",
        "description": "Soft drinks, coffees, teas, beers, and ales"
      },
      {
        "categoryId": 2,
        "categoryName": "Condiments",
        "description": "Sweet and savory sauces, relishes, spreads,
        and seasonings"
      },
      ...
```

8. Нажмите вкладку +, чтобы открыть новую вкладку, напишите запрос, чтобы запросить категорию с `Id 2`, включая `Id`, название и цену ее продуктов, как показано ниже:

```
query Condiments {
  category (categoryId: 2) {
    categoryId
    categoryName
    products {
      productId
      productName
      unitPrice
    }
  }
}
```

Убедитесь, что буква I в categoryId прописная.

9. Нажмите кнопку воспроизведения и обратите внимание на ответ, показанный в следующем частичном выводе:

```
{
  "data": {
    "category": {
      "categoryId": 2,
      "categoryName": "Condiments",
      "products": [
        {
          "productId": 3,
          "productName": "Aniseed Syrup",
          "unitPrice": 10
        },
        {
          "productId": 4,
          "productName": "Chef Anton's Cajun Seasoning",
```

```
          "unitPrice": 22
        },
        ...
```

10. Нажмите вкладку +, чтобы открыть новую вкладку, введите запрос, чтобы запросить `Id`, название и единицы на складе товаров в категории с `Id 1`, как показано ниже:

```
query BeverageProducts {
  products (categoryId: 1) {
    productId
    productName
    unitsInStock
  }
}
```

11. Нажмите кнопку воспроизведения и обратите внимание на ответ, показанный в следующем частичном выводе:

```
{
  "data": {
    "products": [
      {
        "productId": 1,
        "productName": "Chai",
        "unitsInStock": 39
      },
      {
        "productId": 2,
        "productName": "Chang",
        "unitsInStock": 17
      },
      ...
```

12. Закройте браузер Google Chrome и завершите работу веб-сервера.

Мутации и подписки GraphQL

Помимо запросов, другими стандартными функциями GraphQL являются мутации и подписки.

- *Мутации* дают возможность создавать, обновлять и удалять ресурсы.
- *Подписки* позволяют клиенту получать уведомления об изменении ресурсов. Они лучше всего работают с альтернативными коммуникационными технологиями, такими как WebSockets.

Если вы хотите, чтобы я добавил описание этих функций в следующее издание, то, пожалуйста, свяжитесь со мной и дайте мне знать.

Создание клиента для GraphQL

Наконец, в этом подразделе мы посмотрим, как клиент может вызывать сервис GraphQL.

Мы создадим класс модели, чтобы упростить десериализацию ответа.

1. В проекте `Northwind.Mvc` в папке Models создайте файл класса `GraphQLProducts.cs`:

```
using Packt.Shared; // Product

namespace Northwind.Mvc.Models;

public class GraphQLProducts
{
  public class DataProducts
  {
    public Product[]? Products { get; set; }
  }

  public DataProducts? Data { get; set; }
}
```

2. В файле `Program.cs` добавьте операторы для регистрации HTTP-клиента для сервиса GraphQL:

```
builder.Services.AddHttpClient(name: "Northwind.GraphQL",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5005/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

3. В папке Controllers в файле `HomeController.cs` импортируйте пространство имен для работы с текстовыми кодировками:

```
using System.Text; // кодировка
```

4. В методе действия `Services` добавьте операторы для вызова сервиса GraphQL и обратите внимание, что HTTP-запрос является `POST`-запросом, тип носителя — GraphQL, а запрос запрашивает все продукты в категории 8 (это морепродукты):

```
try
{
  HttpClient client = clientFactory.CreateClient(
    name: "Northwind.GraphQL");

  HttpRequestMessage request = new(
    method: HttpMethod.Post, requestUri: "graphql");
```

```
    request.Content = new StringContent(content: @"
      {
        products (categoryId: 8) {
          productId
          productName
          unitsInStock
        }
      }",
      encoding: Encoding.UTF8,
      mediaType: "application/graphql");

    HttpResponseMessage response = await client.SendAsync(request);

    if (response.IsSuccessStatusCode)
    {
      ViewData["seafoodProducts"] = (await response.Content
        .ReadFromJsonAsync<GraphQLProducts>())?.Data?.Products;
    }
    else
    {
      ViewData["seafoodProducts"] = Enumerable.Empty<Product>().ToArray();
    }
  }
  catch (Exception ex)
  {
    _logger.LogWarning($"Northwind.GraphQL service exception: {ex.Message}");
  }
```

5. В папке Views/Home в файле `Services.cshtml` добавьте новый раздел внутри самого контейнера `<div>` после раздела `@if`, который отображает продукты, начинающиеся с `Cha`, из сервиса OData, чтобы показать продукты из категории `Seafood`:

```
@{
  ViewData["Title"] = "Services";
  Product[]? products = ViewData["productsCha"] as Product[];
  Product[]? seafoodProducts = ViewData["seafoodProducts"] as Product[];
}
...
@if (seafoodProducts is not null)
{
  <h2>Seafood products using GraphQL</h2>
  <p>
    @foreach (Product p in seafoodProducts)
    {
      <span class="badge bg-success">
        @p.ProductId
        @p.ProductName
        -
        @(p.UnitsInStock is null ? "0" : p.UnitsInStock.Value) in stock
      </span>
    }
  </p>
}
```

6. При необходимости запустите проект `Minimal.Web` без отладки.
7. При необходимости запустите проект `Northwind.OData` без отладки.
8. Запустите проект `Northwind.GraphQL` без отладки.
9. Запустите проект `Northwind.Mvc`.
10. Перейдите на страницу Services (Сервисы): https://localhost:5001/home/services.
11. Обратите внимание, что продукты из категории `Seafood` успешно извлекаются с помощью GraphQL (рис. 18.5).

Рис. 18.5. Продукты в категории Seafood из сервиса GraphQL

12. Закройте браузер Google Chrome и завершите работу всех веб-серверов.

Реализация сервисов с помощью gRPC

gRPC — это современная высокопроизводительная RPC-платформа с открытым исходным кодом, которая может работать в любой среде.

gRPC

Клиент gRPC может вызывать методы в сервисе gRPC на другом сервере, как если бы это был локальный объект. Разработчик определяет интерфейс сервиса с методами, которые могут быть вызваны удаленно, включая их параметры и типы возврата. Сервер реализует этот интерфейс и запускает сервер gRPC для обработки клиентских вызовов. На стороне клиента сильно типизированный клиент gRPC предоставляет те же методы, что и на сервере.

Как и в случае с WCF, разработка API в gRPC построена на первичном определении контракта, что в данном случае позволяет избежать зависимости от языка

реализации. Вы определяете контракты в файлах `.proto`, которые обладают своим специфическим синтаксисом и затем используют инструменты для их преобразования в различные языки, такие как C#. Файлы `.proto` используются как сервером, так и клиентом для обмена сообщениями в правильном формате.

gRPC минимизирует использование сети, применяя двоичную сериализацию Protobuf, которая не читается человеком, в отличие от JSON или XML, используемых веб-сервисами. gRPC требует HTTP/2, который обеспечивает значительные преимущества в производительности по сравнению с предыдущими версиями, такие как двоичное кадрирование и сжатие, а также мультиплексирование вызовов HTTP/2 через одно соединение.

Основным ограничением gRPC является невозможность его использования в веб-браузерах, поскольку ни один браузер не обеспечивает уровень контроля, необходимый для поддержки клиента gRPC. Например, браузеры не позволяют вызывающей стороне требовать использования HTTP/2. Существует инициатива под названием *gRPC-Web*, которая добавляет дополнительный прокси-уровень, и прокси перенаправляет запросы на сервер gRPC.

Создание сервиса gRPC

Рассмотрим пример сервиса для управления поставщиками в базе данных Northwind.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: ASP.NET Core gRPC Service/`grpc`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.gRPC`.
2. В программе Visual Studio Code выберите `Northwind.gRPC` в качестве активного проекта OmniSharp.

Для работы с файлами .proto в программе Visual Studio Code необходимо расширение vscode-proto3 (zxh404.vscode-proto3).

3. В папке Protos откройте файл `greet.proto` и обратите внимание, что он определяет сервис `Greeter` с методом `SayHello`, который обменивается сообщениями `HelloRequest` и `HelloReply`:

```
syntax = "proto3";

option csharp_namespace = "Northwind.gRPC";

package greet;

// определение сервиса приветствия
service Greeter {
```

```
  // отправляем приветствие
  rpc SayHello (HelloRequest) returns (HelloReply);
}

// сообщение запроса, содержащее имя пользователя
message HelloRequest {
  string name = 1;
}

// ответное сообщение с приветствием
message HelloReply {
  string message = 1;
}
```

4. Откройте файл `Northwind.gRPC.csproj` и обратите внимание, что в нем содержится ссылка на пакет для реализации сервиса gRPC, размещенного в ASP.NET Core, а файл `.proto` зарегистрирован для использования на стороне сервера, как показано ниже:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>

<ItemGroup>
  <PackageReference Include="Grpc.AspNetCore" Version="2.32.0" />
</ItemGroup>
```

5. В папке Services откройте файл `GreeterService.cs` и обратите внимание, что в нем реализован контракт сервиса `Greeter`:

```
using Grpc.Core; // ServerCallContext
using Northwind.gRPC;

namespace Northwind.gRPC.Services;
public class GreeterService : Greeter.GreeterBase
{
  private readonly ILogger<GreeterService> _logger;

  public GreeterService(ILogger<GreeterService> logger)
  {
    _logger = logger;
  }

  public override Task<HelloReply> SayHello(
    HelloRequest request, ServerCallContext context)
  {
    return Task.FromResult(new HelloReply
    {
      Message = "Hello " + request.Name
    });
  }
}
```

6. В файле `Program.cs` в разделе конфигурации сервисов обратите внимание на вызов для добавления gRPC в коллекцию сервисов:

```
builder.Services.AddGrpc();
```

7. В файле `Program.cs` в разделе конфигурации HTTP-конвейера обратите внимание на вызов для сопоставления сервиса `Greeter`:

```
app.MapGrpcService<GreeterService>();
```

8. В файле `Program.cs` добавьте вызов метода расширения `UseUrls`, чтобы указать порт `5006` для HTTPS, как показано ниже:

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseUrls("https://localhost:5006/");
```

9. В папке Properties откройте файл `launchSettings.json` и измените параметр `applicationUrl`, чтобы использовать порт `5006`:

```
{
  "profiles": {
    "Northwind.gRPC": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": false,
      "applicationUrl": "https://localhost:5006",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

10. Соберите проект `Northwind.gRPC`.

Создание клиента gRPC

Мы добавим пакеты клиента gRPC в проект сайта Northwind MVC, чтобы он мог вызывать сервис gRPC.

1. В проект `Northwind.Mvc` добавьте ссылки на пакеты для формата Google Protobuf, клиента gRPC и инструментов, как показано ниже:

```
<PackageReference Include="Google.Protobuf" Version="3.17.3" />
<PackageReference Include="Grpc.NET.Client" Version="2.38.0" />
<PackageReference Include="Grpc.Tools" Version="2.38.1">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native; contentfiles;
    analyzers; buildtransitive</IncludeAssets>
</PackageReference>
```

Пакет Grpc.Tools используется только во время разработки, поэтому маркирован как PrivateAssets=all. Эта метка гарантирует, что инструментарий не будет опубликован на рабочем сайте.

2. Скопируйте папку Protos из проекта/папки `Northwind.gRPC` в проект/папку `Northwind.Mvc`. (В программе Visual Studio 2022 для копирования достаточно перетащить папку. В программе Visual Studio Code перетаскивайте папку, удерживая нажатой клавишу Ctrl или Cmd.)
3. В проекте `Northwind.Mvc` в файле `greet.proto` укажите пространство имен текущего проекта, чтобы автоматически создаваемые классы находились в том же пространстве имен:

```
option csharp_namespace = "Northwind.Mvc";
```

4. В файле проекта `Northwind.Mvc` добавьте группу элементов для регистрации файла `.proto` в качестве используемого на стороне клиента, как показано ниже:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Программа Visual Studio создаст для вас группу элементов, но по умолчанию для GrpcServices будет установлено значение Server, поэтому вы должны вручную изменить его на Client.

5. Соберите проект `Northwind.Mvc`, чтобы убедиться, что автоматически сгенерированные классы созданы.
6. В файле `HomeController.cs` импортируйте пространства имен для работы с gRPC в качестве клиента:

```
using Grpc.NET.Client; // GrpcChannel
```

7. В методе действия `Services` добавьте операторы для создания клиента gRPC и вызова метода `Greet`:

```
try
{
  using (GrpcChannel channel =
    GrpcChannel.ForAddress("https://localhost:5006"))
  {
    Greeter.GreeterClient greeter = new(channel);
    HelloReply reply = await greeter.SayHelloAsync(
      new HelloRequest { Name = "Henrietta" });
    ViewData["greeting"] = "Greeting from gRPC service: " + reply.Message;
  }
}
```

```
catch (Exception)
{
  _logger.LogWarning($"Northwind.gRPC service is not responding.");
}
```

8. В папке Views/Home в файле `Services.cshtml` добавьте код для отображения приветствия непосредственно под заголовком, перед отображением продуктов:

```
@if (ViewData["greeting"] != null)
{
  <p class="alert alert-primary">@ViewData["greeting"]</p>
}
```

Если вы очистите проект gRPC, то потеряете автоматически сгенерированные типы и увидите ошибки компиляции. Чтобы воссоздать их, внесите любые изменения в файл .proto или закройте и снова откройте проект/решение.

Тестирование клиента gRPC для сервиса gRPC

Теперь мы можем запустить сервис gRPC и проверить, может ли сайт Northwind MVC успешно вызвать его.

1. При необходимости запустите проект `Minimal.WebApi` без отладки.
2. При необходимости запустите проект `Northwind.OData` без отладки.
3. При необходимости запустите проект `Northwind.GraphQL` без отладки.
4. Запустите проект сервиса `Northwind.gRPC` без отладки.
5. Запустите проект `Northwind.Mvc`.
6. Перейдите на страницу Services (Сервисы): https://localhost:5001/home/services.
7. Обратите внимание на приветствие на странице сервисов (рис. 18.6).

Рис. 18.6. Страница сервисов после вызова сервиса gRPC для получения приветствия

8. В командной строке или в терминале для сервиса gRPC обратите внимание на информационные сообщения, в которых сказано, что HTTP/2 `POST` был обработан конечной точкой `greet.Greeter/SayHello` примерно за 41 мс:

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5006/greet.Greeter/
SayHello application/grpc -
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/2 POST https://localhost:5006/greet.Greeter/
SayHello application/grpc - - 200 - application/grpc 41.3434ms
```

9. Закройте браузер Google Chrome и завершите работу веб-сервера.

Реализация сервиса gRPC для модели EF Core

Теперь мы добавим в сервис gRPC поддержку работы с базой данных Northwind.

1. В проект `Northwind.gRPC` добавьте ссылку на проект контекста базы данных Northwind для SQLite или SQL Server:

```
<ItemGroup>
  <!-- при необходимости смените Sqlite на SqlServer -->
  <ProjectReference Include=
"..\Northwind.Common.DataContext.Sqlite\Northwind.Common.DataContext.
Sqlite.csproj" />
</ItemGroup>
```

2. В проекте `Northwind.gRPC` в папке Protos создайте файл `shipper.proto` (в программе Visual Studio шаблон элемента называется Protocol Buffer File):

```
syntax = "proto3";

option csharp_namespace = "Northwind.gRPC";

package shipr;

service Shipr {
  rpc GetShipper (ShipperRequest) returns (ShipperReply);
}

message ShipperRequest {
  int32 shipperId = 1;
}

message ShipperReply {
```

```
  int32 shipperId = 1;
  string companyName = 2;
  string phone = 3;
}
```

3. Откройте файл проекта и добавьте запись для включения файла `shipper.proto`:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  <Protobuf Include="Protos\shipper.proto" GrpcServices="Server" />
</ItemGroup>
```

4. Соберите проект `Northwind.gRPC`.
5. В папке Services создайте файл класса `ShipperService.cs` и измените его содержимое, чтобы определить сервис доставки, который использует контекст базы данных Northwind для возврата грузоотправителей:

```
using Grpc.Core; // ServerCallContext
using Packt.Shared; // NorthwindContext, Shipper

namespace Northwind.gRPC.Services;

public class ShipperService : Shipr.ShiprBase
{
  private readonly ILogger<ShipperService> _logger;
  private readonly NorthwindContext db;

  public ShipperService(ILogger<ShipperService> logger,
    NorthwindContext db)
  {
    _logger = logger;
    this.db = db;
  }

  public override async Task<ShipperReply> GetShipper(
    ShipperRequest request, ServerCallContext context)
  {
    return ToShipperReply(
      await db.Shippers.FindAsync(request.ShipperId));
  }

  private ShipperReply ToShipperReply(Shipper? shipper)
  {
    return new ShipperReply
    {
      ShipperId = shipper?.ShipperId ?? 0,
      CompanyName = shipper?.CompanyName ?? string.Empty,
      Phone = shipper?.Phone ?? string.Empty
    };
  }
}
```

6. В файле Program.cs импортируйте пространство имен для контекста базы данных Northwind:

```
using Packt.Shared; // метод расширения AddNorthwindContext
```

7. В разделе конфигурации сервисов добавьте вызов для регистрации контекста базы данных Northwind:

```
builder.Services.AddNorthwindContext();
```

8. В разделе конфигурации HTTP-конвейера после вызова для регистрации сервиса Greeter добавьте оператор для регистрации сервиса доставки:

```
app.MapGrpcService<ShipperService>();
```

Реализация клиента gRPC для модели EF Core

Теперь мы можем добавить клиентские возможности на сайт Northwind MVC.

1. Скопируйте файл shipper.proto из папки Protos проекта Northwind.gRPC в папку Protos проекта Northwind.MVC. (Удерживайте нажатой клавишу Ctrl или Cmd при перетаскивании, если используете программу Visual Studio Code.)
2. В проекте Northwind.Mvc в файле shipper.proto измените пространство имен соответствующим образом, чтобы автоматически сгенерированные классы находились в том же пространстве имен:

```
option csharp_namespace = "Northwind.Mvc";
```

3. В файле проекта Northwind.Mvc добавьте запись для регистрации файла .proto как используемого на стороне клиента:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
  <Protobuf Include="Protos\shipper.proto" GrpcServices="Client" />
</ItemGroup>
```

4. В папке Controllers в файле HomeController.cs в методе действия Services добавьте операторы для вызова gRPC-сервиса Shipper:

```
try
{
  using (GrpcChannel channel =
    GrpcChannel.ForAddress("https://localhost:5006"))
  {
    Shipr.ShiprClient shipr = new(channel);

    ShipperReply reply = await shipr.GetShipperAsync(
      new ShipperRequest { ShipperId = 3 });

    ViewData["shipr"] = new Shipper
    {
```

```
        ShipperId = reply.ShipperId,
        CompanyName = reply.CompanyName,
        Phone = reply.Phone
      };
    }
  }
  catch (Exception)
  {
    _logger.LogWarning($"Northwind.gRPC service is not responding.");
  }
```

5. В папке Views/Home в файле `Services.cshtml` добавьте код для отображения информации о грузоотправителе после приветствия, как показано ниже:

```
@if (ViewData["shipr"] != null)
{
  Shipper? shipper = ViewData["shipr"] as Shipper;
  <p class="alert alert-danger">
    ShipperId: @shipper?.ShipperId, CompanyName: @shipper?.CompanyName,
Phone: @shipper?.Phone
  </p>
}
```

6. При необходимости запустите проект `Minimal.WebApi` без отладки.
7. При необходимости запустите проект `Northwind.OData` без отладки.
8. При необходимости запустите проект `Northwind.GraphQL` без отладки.
9. Запустите проект сервиса `Northwind.gRPC` без отладки.
10. Запустите проект `Northwind.Mvc`.
11. Перейдите на страницу Services (Сервисы): https://localhost:5001/home/services.
12. Обратите внимание на информацию о грузоотправителе на странице сервисов (рис. 18.7).
13. Закройте браузер Google Chrome и завершите работу веб-сервера.

Рис. 18.7. Страница сервисов после вызова сервиса gRPC для получения грузоотправителя

Реализация взаимодействия в режиме реального времени с помощью SignalR

Интернет отлично подходит для создания сайтов и сервисов общего назначения, но он не был разработан для специализированных сценариев, требующих мгновенного обновления веб-страницы по мере поступления новой информации.

История взаимодействия в режиме реального времени в Интернете

Чтобы понять преимущества SignalR, необходимо знать историю HTTP и то, как организации работали над улучшением взаимодействия между клиентами и серверами в режиме реального времени.

На заре развития Интернета в 1990-х годах браузеры должны были отправлять полностраничный HTTP-запрос `GET` к веб-серверу, чтобы получить свежую информацию для показа посетителю.

XMLHttpRequest

В конце 1999 года Microsoft выпустила браузер Internet Explorer 5.0 с компонентом *XMLHttpRequest*, который мог выполнять асинхронные HTTP-запросы в фоновом режиме. Это, наряду с *динамическим HTML* (DHTML), позволяло плавно обновлять части веб-страницы при поступлении свежих данных.

Преимущества этой техники были очевидны, и вскоре все браузеры добавили такой же компонент.

AJAX

Компания Google максимально использовала данную возможность для создания умных веб-приложений, таких как Google Maps и Gmail. Несколько лет спустя эта техника стала известна как *асинхронный JavaScript и XML* (Asynchronous JavaScript and XML, AJAX).

Однако AJAX по-прежнему использует HTTP для обмена данными, и это имеет свои ограничения:

- во-первых, HTTP является протоколом взаимодействия «запрос — ответ»; это значит, сервер не может передавать данные клиенту. Он должен ждать, пока клиент сделает запрос;
- во-вторых, сообщения запроса и ответа HTTP содержат заголовки с большим количеством потенциально ненужных служебных данных;

- в-третьих, HTTP обычно требует создания нового базового TCP-соединения при каждом запросе.

WebSocket

WebSocket — полнодуплексная технология, то есть инициировать передачу новых данных может как клиент, так и сервер. WebSocket использует одно и то же TCP-соединение во время соединения. Она также более эффективна в плане размера отправляемых сообщений, поскольку минимальный размер фрейма составляет 2 байта.

WebSocket работает через HTTP-порты `80` и `443`, поэтому технология совместима с протоколом HTTP, а подтверждение WebSocket использует заголовок HTTP `Upgrade` для перехода от протокола HTTP к протоколу WebSocket.

Современные веб-приложения должны предоставлять актуальную информацию. Классическим примером является чат вживую, но существует множество других потенциальных приложений, от котировок акций до игр.

Всякий раз, когда серверу требуется передавать обновления на веб-страницу, вам нужна совместимая с Интернетом технология взаимодействия в режиме реального времени. Можно использовать WebSocket, но она поддерживается не всеми клиентами.

Знакомство с SignalR

ASP.NET Core SignalR — это библиотека с открытым исходным кодом, которая упрощает добавление веб-функций реального времени в приложения. Это своего рода абстракция над несколькими базовыми технологиями связи, что позволяет добавлять возможности взаимодействия в режиме реального времени с помощью сценариев на C#.

Разработчику не нужно понимать или реализовывать используемые базовые технологии, а SignalR будет автоматически переключаться между базовыми технологиями в зависимости от их поддержки браузером посетителя. Например, SignalR будет использовать WebSocket, когда эта технология доступна, а если нет, то плавно перейдет на другие, такие как длинный опрос AJAX, при этом код вашего приложения останется неизменным.

SignalR — это API для *удаленных вызовов процедур* (remote procedure calls, RPC) между сервером и клиентом. RPC вызывают функции JavaScript на клиентах из кода .NET на стороне сервера. SignalR имеет концентраторы для определения конвейера и автоматически обрабатывает отправку сообщений, используя два встроенных протокола концентраторов: JSON и бинарный протокол, основанный на MessagePack.

На стороне сервера SignalR работает везде, где поддерживается ASP.NET Core: серверы Windows, macOS или Linux. SignalR поддерживает следующие клиентские платформы:

- JavaScript-клиенты для современных браузеров, включая Chrome, Firefox, Safari, Edge и Internet Explorer 11;
- клиенты .NET, включая Blazor и Xamarin для мобильных приложений Android и iOS;
- Java 8 и более поздние версии.

Разработка сигнатур методов

При разработке сигнатур методов для сервиса SignalR лучше всего определять методы с одним параметром объекта, а не с несколькими параметрами простого типа. Например, определите класс с несколькими свойствами для использования в качестве типа одного параметра вместо передачи нескольких значений `string`:

```
// пример плохого варианта
public void SendMessage(string to, string body)

// пример лучшего решения
public class Message
{
  public string To { get; set; }
  public string Body { get; set; }
}

public void SendMessage(Message message)
```

Причина в том, что вы сможете вносить изменения в будущем, например добавить заголовок сообщения. В примере с плохим вариантом потребуется добавить третий строковый параметр `title`, и существующие клиенты будут получать ошибки, поскольку не отправляют дополнительное строковое значение. А использование примера лучшего решения не нарушит сигнатуру метода, поэтому существующие клиенты смогут продолжать вызывать его, как и до изменения. На стороне сервера дополнительное свойство `title` будет иметь нулевое значение, которое можно проверить и, возможно, установить в качестве значения по умолчанию.

Создание сервиса взаимодействия в реальном времени с помощью SignalR

Серверная библиотека SignalR включена в ASP.NET Core. Но клиентская библиотека JavaScript не включается в проект автоматически. Мы будем использовать инструмент *Library Manager CLI* для получения клиентской библиотеки из *unpkg*,

сети доставки контента (content delivery network, CDN), которая может доставлять все, что находится в менеджере пакетов Node.js.

Добавим серверный концентратор SignalR и клиентский JavaScript в проект Northwind MVC, чтобы реализовать функцию чата, позволяющую посетителям отправлять сообщения всем, кто в данный момент пользуется сайтом, динамически определяемым группам или одному указанному пользователю.

В рабочем решении было бы лучше разместить концентратор SignalR в отдельном веб-проекте, чтобы его можно было размещать и масштабировать независимо от остальной части сайта. Взаимодействие в реальном времени часто может создавать чрезмерную нагрузку на сайт.

Определение некоторых общих моделей

Сначала мы определим две общие модели, которые могут быть использованы как в серверном, так и в клиентском .NET-проектах, которые будут работать с нашим сервисом чата.

1. В проекте `Northwind.Common` добавьте файл класса `RegisterModel.cs` и измените его содержимое, чтобы определить модель для регистрации имени пользователя и групп, к которым он должен принадлежать:

```
namespace Northwind.Chat.Models;

public class RegisterModel
{
  public string? Username { get; set; }
  public string? Groups { get; set; }
}
```

2. В проекте `Northwind.Common` добавьте файл класса `MessageModel.cs` и измените его содержимое, чтобы определить модель сообщения со свойствами для тех, кому отправляется сообщение, и их тип (пользователь, группа или все), а также для тех, от кого было отправлено сообщение, и тела сообщения:

```
namespace Northwind.Chat.Models;

public class MessageModel
{
  public string? To { get; set; }
  public string? ToType { get; set; }
  public string? From { get; set; }
  public string? Body { get; set; }
}
```

Включение концентратора SignalR на стороне сервера

Далее мы включим концентратор SignalR на стороне сервера проекта Northwind MVC.

1. В проекте `Northwind.Mvc` добавьте ссылку на проект `Northwind.Common`, если вы не добавили ссылку на проект ранее.
2. В проекте `Northwind.Mvc` создайте папку `Hubs`.
3. В папке `Hubs` создайте файл класса `ChatHub.cs` и измените его содержимое так, чтобы он наследовался от класса `Hub` и реализовывал два метода, которые могут быть вызваны клиентом:

```
using Microsoft.AspNetCore.SignalR; // Hub
using Northwind.Chat.Models; // RegisterModel, MessageModel

namespace Northwind.Mvc.Hubs;

public class ChatHub : Hub
{
  // создаем новый экземпляр ChatHub для обработки каждого метода,
  // поэтому мы должны хранить имена пользователей и их идентификаторы
  // соединений в статическом поле
  private static Dictionary<string, string> users = new();

  public async Task Register(RegisterModel model)
  {
    // добавляем/обновляем словарь с именем пользователя и его connectionId
    users[model.Username] = Context.ConnectionId;

    foreach (string group in model.Groups.Split(','))
    {
      await Groups.AddToGroupAsync(Context.ConnectionId, group);
    }
  }

  public async Task SendMessage(MessageModel command)
  {
    MessageModel reply = new()
    {
      From = command.From,
      Body = command.Body
    };

    IClientProxy proxy;

    switch (command.ToType)
    {
      case "User":
        string connectionId = users[command.To];
        reply.To = $"{command.To} [{connectionId}]";
        proxy = Clients.Client(connectionId);
        break;
```

```
        case "Group":
          reply.To = $"Group: {command.To}";
          proxy = Clients.Group(command.To);
          break;

        default:
          reply.To = "Everyone";
          proxy = Clients.All;
          break;
      }

      await proxy.SendAsync(
        method: "ReceiveMessage", arg1: reply);
    }
  }
```

Обратите внимание на следующие моменты:

- `ChatHub` имеет два метода, которые может вызвать клиент: `Register` и `SendMessage`;
- `Register` имеет один параметр типа `RegisterModel`. Имя пользователя и его идентификатор соединения хранятся в статическом словаре, чтобы имя пользователя можно было использовать для поиска идентификатора соединения и отправки сообщений непосредственно этому пользователю;
- `SendMessage` имеет один параметр типа `MessageModel`. Метод создает экземпляр класса `MessageModel` — сообщение, которое он отправляет одному или нескольким клиентам. Затем он действует в зависимости от типа получателя. Для пользователя он ищет идентификатор соединения по имени пользователя, а затем вызывает метод `Client`, чтобы получить прокси, который будет взаимодействовать только с этим клиентом. Для группы он вызывает метод `Group`, чтобы получить прокси, который будет взаимодействовать только с членами этой группы. Во всех остальных случаях он вызывает метод `All`, чтобы получить прокси, который будет взаимодействовать с каждым клиентом. Наконец, отправляет сообщение асинхронно, используя прокси.

4. В файле `Program.cs` импортируйте пространство имен для вашего концентратора SignalR:

```
using Northwind.Mvc.Hubs; // ChatHub
```

5. В разделе конфигурации сервисов добавьте оператор для добавления поддержки SignalR в коллекцию сервисов:

```
builder.Services.AddSignalR();
```

6. В разделе конфигурации HTTP-конвейера после вызова для отображения Razor Pages добавьте оператор для отображения относительного URL-пути/чата на ваш концентратор SignalR:

```
app.MapHub<ChatHub>("/chat");
```

Добавление клиентской JavaScript-библиотеки SignalR

Далее мы добавим клиентскую JavaScript-библиотеку SignalR, чтобы можно было использовать ее на веб-странице.

1. Откройте командную строку или терминал для проекта `Northwind.Mvc`.
2. Установите инструмент Library Manager CLI, как показано в следующей команде:

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

Возможно, этот инструмент уже установлен. Чтобы обновить его до последней версии, повторите команду, но замените слово install (установить) на update (обновить).

3. Обратите внимание на сообщение об успешном завершении:

```
You can invoke the tool using the following command: libman
Tool 'microsoft.web.librarymanager.cli' (version '2.1.113') was
successfully installed.
```

4. Добавьте библиотеки `signalr.js` и `signalr.min.js` в проект из `unpkg`, как показано в следующей команде:

```
libman install @microsoft/signalr@latest -p unpkg -d wwwroot/js/signalr
--files dist/browser/signalr.js --files dist/browser/signalr.min.js
```

5. Обратите внимание на сообщение об успешном завершении:

```
Downloading file https://unpkg.com/@microsoft/signalr@latest/dist/browser/
signalr.js...
Downloading file https://unpkg.com/@microsoft/signalr@latest/dist/browser/
signalr.min.js...
wwwroot/js/signalr/dist/browser/signalr.js written to disk
wwwroot/js/signalr/dist/browser/signalr.min.js written to disk
Installed library "@microsoft/signalr@latest" to "wwwroot/js/signalr"
```

В программе Visual Studio есть графический интерфейс для добавления библиотек JavaScript на стороне клиента. Чтобы воспользоваться им, щелкните правой кнопкой мыши на веб-проекте и выберите команду меню Add ▸ Client Side Libraries (Добавить ▸ Библиотеки на стороне клиента).

Добавление страницы чата на сайт Northwind MVC

Далее мы создадим страницу чата.

1. В папке Controllers откройте файл `HomeController.cs` и добавьте новый метод действия для чата:

```
public IActionResult Chat()
{
  return View();
}
```

2. В папке Views/Shared в файле `_Layout.cshtml` добавьте новый элемент навигации после элемента навигации `Services`, который переходит на страницу чата, как показано ниже:

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area=""
      asp-controller="Home" asp-action="Chat">Chat</a>
</li>
```

3. В папке Views/Home добавьте новое пустое представление `Chat.cshtml` и измените его содержимое:

```
@{
  ViewData["Title"] = "Chat";
}
<div class="container">
  <h1>@ViewData["Title"]</h1>
  <div class="row">
    <div class="col-12">
      <h2>Register</h2>
    </div>
  </div>
  <div class="row">
    <div class="col-4">My name</div>
    <div class="col-8"><input type="text" id="from" /></div>
  </div>
  <div class="row">
    <div class="col-4">My groups</div>
    <div class="col-8"><input type="text" id="groups" value="Sales,IT"/>
    </div>
  </div>
  <div class="row">
    <div class="col-12">
      <input type="button" id="registerButton" value="Register" />
    </div>
  </div>
  <div class="row">
    <div class="col-12">
      <h2>Message</h2>
```

```
      </div>
    </div>
    <div class="row">
      <div class="col-4">To type</div>
      <div class="col-8">
        <select id="toType">
          <option selected>Everyone</option>
          <option>Group</option>
          <option>User</option>
        </select>
      </div>
    </div>
    <div class="row">
      <div class="col-4">To</div>
      <div class="col-8"><input type="text" id="to" /></div>
    </div>
    <div class="row">
    <div class="col-4">Body</div>
      <div class="col-8"><input type="text" id="body" /></div>
    </div>
    <div class="row">
      <div class="col-12">
        <input type="button" id="sendButton" value="Send" />
      </div>
    </div>
    <div class="row">
      <div class="col-12">
        <hr />
      </div>
    </div>
    <div class="row">
      <div class="col-12">
        <h2>Messages received</h2>
      </div>
    </div>
    <div class="row">
      <div class="col-12">
        <ul id="messages"></ul>
      </div>
    </div>
  </div>
  <script src="~/js/signalr/dist/browser/signalr.js"></script>
  <script src="~/js/chat.js"></script>
```

Обратите внимание на следующие моменты:

- на странице есть три раздела: Register (Регистрация), Message (Сообщение) и Messages received (Полученные сообщения);
- раздел Register (Регистрация) содержит два ввода для имени посетителя и разделенного запятыми списка групп, членом которых он хочет быть, а также кнопку, которую нужно нажать для регистрации;

- раздел Message (Сообщение) содержит три ввода для типа получателя, имени получателя и тела сообщения, а также кнопку, которую нужно нажать для отправки сообщения;
- раздел Messages received (Полученные сообщения) содержит элемент неупорядоченного списка, который будет динамически заполняться элементами списка при получении сообщения;
- за двумя элементами сценария для клиентской библиотеки SignalR следует реализация клиента чата.

4. В каталоге wwwroot/js создайте JavaScript-файл `chat.js` и измените его содержимое:

```
"use strict";

var connection = new signalR.HubConnectionBuilder()
  .withUrl("/chat").build();

document.getElementById("registerButton").disabled = true;
document.getElementById("sendButton").disabled = true;

connection.start().then(function () {
  document.getElementById("registerButton").disabled = false;
  document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
  return console.error(err.toString());
});

connection.on("ReceiveMessage", function (received) {
  var li = document.createElement("li");
  document.getElementById("messages").appendChild(li);
  // обратите внимание на использование обратной кавычки `,
  // чтобы включить форматированную строку
  li.textContent =
    `${received.from} says ${received.body} (sent to ${received.to})`;
});

document.getElementById("registerButton").addEventListener("click",
  function (event) {
  var registermodel = {
    username: document.getElementById("from").value,
    groups: document.getElementById("groups").value
  };
  connection.invoke("Register", registermodel).catch(function (err) {
    return console.error(err.toString());
  });
  event.preventDefault();
});

document.getElementById("sendButton").addEventListener("click",
```

```
  function (event) {
  var messageToSend = {
    to: document.getElementById("to").value,
    toType: document.getElementById("toType").value,
    from: document.getElementById("from").value,
    body: document.getElementById("body").value
  };
  connection.invoke("SendMessage", messageToSend).catch(function (err) {
    return console.error(err.toString());
  });
  event.preventDefault();
});
```

Обратите внимание на следующие моменты:

- сценарий создает конструктор соединения с концентратором SignalR, указывающий на относительный путь URL к концентратору чата на сервере/в чате;
- сценарий отключает кнопки Register (Регистрация) и Send (Отправить) до тех пор, пока соединение с концентратором на стороне сервера не будет успешно установлено;
- когда соединение получает вызов `ReceiveMessage` от концентратора на стороне сервера, оно добавляет элемент списка в неупорядоченный список `messages`. Содержимое элемента списка содержит подробную информацию о сообщении, такую как `from` (от кого), `to` (кому) и `body` (тело сообщения). Обратите внимание, что JavaScript использует верблюжий регистр;
- к кнопке Register (Регистрация) добавляется обработчик события нажатия кнопки мыши, который создает модель регистрации с именем пользователя и его группами, а затем вызывает метод `Register` на стороне сервера;
- обработчик события нажатия кнопки мыши добавляется к кнопке Send (Отправить), которая создает модель сообщения с параметрами `from`, `to`, `type` и `body`, а затем вызывает метод `SendMessage` на стороне сервера.

Тестирование функции чата

Теперь мы готовы попробовать отправить сообщения в чате между несколькими посетителями сайта.

1. Запустите сайт проекта `Northwind.Mvc`.
2. Запустите браузер Google Chrome.
3. Перейдите по адресу https://localhost:5001/home/chat.
4. Введите значения `Alice` для имени, `Sales, IT` для групп, а затем нажмите кнопку Register (Регистрация).
5. Откройте новое окно браузера или запустите другой браузер, например Firefox или Edge.

6. Перейдите по адресу https://localhost:5001/home/chat.
7. Введите значения `Bob` для имени, `Sales` для групп, а затем нажмите кнопку Register (Регистрация).
8. Откройте новое окно браузера или запустите другой браузер, например Firefox или Edge.
9. Перейдите по адресу https://localhost:5001/home/chat.
10. Введите значения `Charlie` для имени, `IT` для групп, а затем нажмите кнопку Register (Регистрация).
11. Расположите окна браузера так, чтобы вы могли видеть все три одновременно.
12. В браузере Алисы выберите Group (Группа), введите `Sales`, введите `Sell more!` и нажмите кнопку Send (Отправить).
13. Обратите внимание, что Алиса и Боб получили сообщение (рис. 18.8).

Рис. 18.8. Алиса отправляет сообщение группе Sales (Продажи)

14. В браузере Боба выберите Group (Группа), введите `IT`, введите `Fix more bugs!` и нажмите кнопку Send (Отправить).
15. Обратите внимание, что Алиса и Чарли получили сообщение (рис. 18.9).
16. В браузере Алисы выберите User (Пользователь), введите `Bob`, введите `Bonjour Bob!` и нажмите кнопку Send (Отправить).
17. Обратите внимание, что сообщение получает только Боб (рис. 18.10).

Рис. 18.9. Боб отправляет сообщение группе IT

Рис. 18.10. Алиса отправляет сообщение Бобу

18. В браузере Чарли оставьте для параметра To type (Кому) значение Everyone (Все), для параметра To (Кому) оставьте значение пустым, введите любое сообщение, а затем нажмите кнопку Send (Отправить) и обратите внимание, что все получили сообщение.
19. Закройте браузеры и завершите работу веб-сервера.

Создание консольного приложения клиента чата

Теперь создадим клиент .NET для SignalR. Мы будем использовать консольное приложение, хотя для любого типа проекта .NET потребуется одна и та же ссылка на пакет и код реализации.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.SignalR.ConsoleClient`.
2. Добавьте ссылку на пакет для клиента ASP.NET Core SignalR и ссылку на проект для `Northwind.Common`:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.SignalR.Client"
                    Version="6.0.0" />
</ItemGroup>
<ItemGroup>
  <ProjectReference
    Include="..\Northwind.Common\Northwind.Common.csproj" />
</ItemGroup>
```

3. В файле `Program.cs` импортируйте пространства имен для работы с SignalR в качестве клиента и модели чата, а затем добавьте операторы для создания соединения с концентратором, предложите пользователю ввести имя пользователя и группы для регистрации, а затем прослушивайте полученные сообщения:

```
using Microsoft.AspNetCore.SignalR.Client; // HubConnection
using Northwind.Chat.Models; // RegisterModel, MessageModel

using static System.Console;

Write("Enter a username: ");
string? username = ReadLine();
```

```
Write("Enter your groups: ");
string? groups = ReadLine();

HubConnection hubConnection = new HubConnectionBuilder()
  .WithUrl("https://localhost:5001/chat")
  .Build();

hubConnection.On<MessageModel>("ReceiveMessage", message =>
{
  WriteLine($"{message.From} says {message.Body} (sent to {message.To})");
});

await hubConnection.StartAsync();

WriteLine("Successfully started.");

RegisterModel registration = new()
{
  Username = username,
  Groups = groups
};

await hubConnection.InvokeAsync("Register", registration);

WriteLine("Successfully registered.");
WriteLine("Listening... (press ENTER to stop.)");
ReadLine();
```

4. Запустите сайт проекта `Northwind.Mvc` без отладки.
5. Запустите браузер Google Chrome.
6. Перейдите по адресу https://localhost:5001/home/chat.
7. Введите значения `Alice` для имени, `Sales, IT` для групп, а затем нажмите кнопку Register (Регистрация).
8. Запустите проект `Northwind.SignalR.ConsoleClient`, а затем введите свое имя и группы `Sales,Admins`.
9. Расположите окна браузера и консольного приложения так, чтобы вы могли видеть оба одновременно.
10. В браузере Алисы в разделе Message (Сообщения) выберите пункт Group (Группа), введите `Sales`, затем `Go team!`, нажмите кнопку Send (Отправить) и обратите внимание, что Алиса и вы получили сообщение.
11. Попробуйте отправить сообщения только себе, только членам группы `Admins` и всем, как показано на рис. 18.11.
12. В консольном приложении нажмите клавишу Enter, чтобы остановить его.
13. Закройте браузер Google Chrome и завершите работу веб-сервера.

Рис. 18.11. Алиса отправляет сообщения различным типам получателей

Реализация бессерверных сервисов с помощью Azure Functions

Azure Functions — это управляемая событиями платформа для бессерверных вычислений. Вы можете создавать и отлаживать их локально, а затем развертывать в облаке Microsoft Azure. Платформу можно реализовать на многих языках, а не только на C# и .NET. Она имеет расширения для Visual Studio и Visual Studio Code.

Зачем нужно создавать сервис без сервера? «Бессерверный» не означает буквально «без сервера». «Бессерверный» означает отсутствие постоянно работающего сервера, как правило, бóльшую часть времени.

Например, в организациях часто есть бизнес-функции, которые необходимо выполнять только раз в месяц или на разовой основе. Возможно, организация печатает чеки для выплаты зарплаты своим сотрудникам в конце месяца. Для этих чеков может потребоваться конвертация сумм заработной платы в слова для печати на чеке. Функция для преобразования чисел в слова может быть реализована как бессерверный сервис.

Платформа Azure Functions — это нечто большее, чем просто отдельные функции. С помощью *Durable Functions* поддерживаются сложные рабочие процессы с отслеживанием состояния и решения, управляемые событиями. Мы не будем

рассматривать эти функции в данной книге, поэтому, если вам интересно, вы можете узнать о них больше здесь: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

Модель Azure Functions

Платформа Azure Functions имеет модель программирования, основанную на триггерах и связях, которые позволяют вашим бессерверным приложениям реагировать на события и подключаться к другим сервисам, например к хранилищам данных.

Триггеры и привязки Azure Functions

Триггеры и привязки — ключевые понятия Azure Functions. Триггеры — это то, что заставляет функцию выполняться. Каждая функция должна иметь один и только один триггер. Наиболее распространенные триггеры представлены ниже:

- *HTTP* — реагирует на входящий HTTP-запрос;
- *Queue* — реагирует на поступление в очередь сообщения, готового к обработке;
- *Timer* — реагирует на наступление определенного времени;
- *Event Grid* — реагирует на наступление предопределенного события.

Привязки позволяют функциям иметь входы и выходы. Каждая функция может иметь ноль, одну или несколько привязок. Вот некоторые распространенные привязки:

- *Blob storage* — чтение или запись в любой файл, хранящийся в виде *двоичного большого объекта* (binary large object, BLOB);
- *Cosmos DB* — чтение или запись документов в облачное хранилище данных;
- *SignalR* — получение или выполнение удаленных вызовов методов;
- *Queue* — запись сообщения в очередь;
- *SendGrid* — отправка сообщения электронной почты;
- *Twilio* — отправка СМС;
- *IoT Hub* — запись на подключенное к Интернету устройство.

С полным списком поддерживаемых привязок вы можете ознакомиться здесь: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=csharp#supported-bindings.

Триггеры и привязки настраиваются по-разному для разных языков. Для C# и Java вы дополняете методы и параметры атрибутами. Для других языков вы настраиваете файл `function.json`.

Версии и языки Azure Functions

Azure Functions поддерживает четыре версии узлов выполнения и несколько языков (табл. 18.5).

Таблица 18.5. Версии узлов и языки, поддерживаемые Azure Functions

Язык (-и)	Версия 1 (v1)	Версия 2 (v2)	Версия 3 (v3)	Версия 4 (v4)
C#, F#	.NET Framework 4.8	.NET Core 2.1	.NET Core 3.1, .NET 5.0[2]	.NET 6.0[2]
JavaScript[1]	Node 6	Node 8, 10	Node 10, 12, 14	—
Java	—	Java 8	Java 8, 11	—
PowerShell	—	PowerShell Core 6	PowerShell Core 6, 7	—
Python	—	Python 3.6, 3.7	Python 3.6, 3.7, 3.8, 3.9	—

В этой книге мы рассмотрим только реализацию Azure Functions с помощью C# и .NET.

Модели хостинга Azure Functions

У Azure Functions есть две модели хостинга: внутрипроцессный и изолированный.

- *Внутрипроцессный* — ваша функция реализуется в библиотеке классов, которая запускается в том же процессе, что и хост. Ваши функции должны работать на той же версии .NET, что и среда выполнения Azure Functions.
- *Изолированный* — ваша функция реализована в консольном приложении, которое запускается в собственном процессе. Поэтому ваша функция может выполняться на текущих выпусках, таких как .NET 5.0, которые не поддерживаются средой выполнения Azure Functions, разрешающей только LTS-выпуски в процессе.

Azure Functions изначально поддерживает только одну LTS-версию .NET. Например, для Azure Functions v3 ваша функция должна использовать .NET Core 3.1 внутрипроцессно. Для Azure Functions v4 ваша функция должна использовать .NET 6.0 внутрипроцессно. Если вы создаете изолированную функцию, то можете выбрать любую версию .NET.

---

[1] Azure Functions поддерживает язык TypeScript путем компиляции в код на другом языке программирования (преобразования/компиляции) в JavaScript.

[2] .NET 5.0 поддерживается только в модели изолированного хостинга, поскольку это текущая версия. Платформа .NET 6.0 поддерживается как в изолированном, так и в текущем режиме, поскольку это выпуск с долгосрочной поддержкой.

Настройка локальной среды разработки для Azure Functions

Для начала, вам нужно установить последнюю версию *Azure Functions Core Tools*, которой на момент написания этой книги является v4, по следующей ссылке: https://www.npmjs.com/package/azure-functions-core-tools.

Azure Functions Core Tools (Основные инструменты от Azure Functions) предоставляет основную среду выполнения и шаблоны для создания функций, которые позволяют осуществлять локальную разработку на Windows, macOS и Linux с использованием любого редактора кода.

Azure Functions Core Tools включен в рабочую нагрузку разработки Azure в программе Visual Studio 2022, поэтому, возможно, он у вас уже установлен.

Создание проекта Azure Functions для локального запуска

Теперь мы можем создать проект Azure Functions. Хотя их можно создать в облаке с помощью портала Azure, разработчикам удобнее создавать и запускать их локально. После тестирования функции на собственном компьютере ее можно развернуть в облаке.

Каждый редактор кода имеет немного разные возможности для начала работы с проектом Azure Functions.

В программе Visual Studio 2022

Если вы предпочитаете Visual Studio, то создать проект Azure Functions можно следующим образом.

1. Откройте редактор кода и создайте проект с такими настройками:
   1) шаблон проекта: Azure Functions;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.AzureFuncs`.
2. В программе Visual Studio выберите .NET 6 (Isolated) (.NET 6 (Изолированный)), Http trigger (Триггер Http), Storage emulator (Эмулятор хранилища), а для Authorization level (Уровень авторизации) выберите Anonymous (Анонимный), затем нажмите Create (Создать) (рис. 18.12).

Рис. 18.12. Выбор параметров для проекта Azure Functions в программе Visual Studio 2022

В программе Visual Studio Code

Если вы предпочитаете Visual Studio Code, то создать проект Azure Functions можно следующим образом.

1. В программе Visual Studio Code перейдите в раздел Extensions (Расширения) и найдите расширение для Azure Functions (`ms-azuretools.vscode-azurefunctions`). Оно имеет зависимости от двух других расширений: Azure Account (`ms-vscode.azure-account`) и Azure Resources (`ms-azuretools.vscode-azureresourcegroups`), поэтому они тоже будут установлены.
2. В папке PracticalApps создайте папку Northwind.AzureFuncs и добавьте ее в рабочую область `PracticalApps`.
3. Закройте рабочую область `PracticalApps`, а затем откройте папку Northwind.AzureFuncs. (Следующие шаги допустимы только за пределами рабочей области.)
4. В расширении Azure в разделе FUNCTIONS (Функции) нажмите кнопку Create new project (Создать новый проект), а затем выберите папку Northwind.AzureFuncs (рис. 18.13).

Рис. 18.13. Выбор папки для проекта Azure Functions

5. В подсказках сделайте следующее:
   1) выберите язык для проекта функций: C#;
   2) выберите .NET 6 LTS в качестве среды выполнения .NET, к сожалению не показанной на рис. 18.14, поскольку на момент написания этой книги она еще не была выпущена;

Рис. 18.14. Выбор целевой среды выполнения .NET для проекта Azure Functions

   3) выберите шаблон для первой функции проекта: HTTP trigger;
   4) укажите имя функции: `NumbersToWordsFunction`;
   5) укажите пространство имен: `Northwind.AzureFuncs`;
   6) выберите уровень авторизации: Anonymous (Анонимный).
6. В меню File (Файл) программы Visual Studio Code закройте папку.
7. Откройте рабочую область `PracticalApps`.

С помощью func CLI

Если вы предпочитаете командную строку и какой-либо другой редактор кода, то создать проект Azure Functions можно следующим образом.

1. В папке PracticalApps создайте папку Northwind.AzureFuncs и добавьте ее в рабочую область `PracticalApps`.
2. В командной строке или в терминале в папке Northwind.AzureFuncs создайте проект Azure Functions с помощью C#, как показано в следующей команде:

```
func init --csharp
```

3. В командной строке или в терминале в папке Northwind.AzureFuncs создайте функцию Azure Functions с помощью HTTP-триггера, которую можно вызывать анонимно:

```
func new --name NumbersToWordsFunction --template "HTTP trigger"
--authlevel "anonymous"
```

4. При необходимости вы можете запустить функцию локально, как показано в следующей команде:

```
func start
```

Обзор проекта

Прежде чем писать функцию, мы рассмотрим, из чего состоит проект Azure Functions.

1. Откройте файл проекта и обратите внимание на версию Azure Functions и ссылки на пакеты, необходимые для реализации Azure Function, отвечающей на HTTP-запросы:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <AzureFunctionsVersion>v4</AzureFunctionsVersion>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Sdk.Functions"
      Version="3.0.13" />
  </ItemGroup>
  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
  </ItemGroup>
</Project>
```

2. Откройте файл `local.settings.json` и обратите внимание, что во время локальной разработки ваш проект будет использовать локальное хранилище разработки и изолированный процесс:

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet"
  }
}
```

Реализация функции

Теперь мы можем реализовать функцию для преобразования чисел в слова.

1. Если вы выполнили упражнение из главы 8 по написанию функции, преобразующей числа в слова, то используйте свою реализацию. Если нет, то используйте

класс, доступный по следующей ссылке: https://github.com/markjprice/cs10dotnet6/blob/master/vscode/PracticalApps/Northwind.AzureFuncs/NumbersToWords.cs.

2. Если вы используете Visual Studio, то в проекте `Northwind.AzureFuncs` щелкните правой кнопкой мыши на файле `Function1.cs` и переименуйте его в `NumbersToWordsFunction.cs`.
3. Откройте файл `NumbersToWordsFunction.cs` и измените его содержимое, чтобы реализовать Azure Function для преобразования суммы в виде числа в слова:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs; // [FunctionName], [HttpTrigger]
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using System.Numerics; // BigInteger
using Packt.Shared; // метод расширения ToWords
using System.Threading.Tasks; // Task

namespace Northwind.AzureFuncs;

public static class NumbersToWordsFunction
{
  [FunctionName(nameof(NumbersToWordsFunction))]
  public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")]
    HttpRequest req, ILogger log)
  {
    log.LogInformation($"C# HTTP trigger function processed a request.");

    string amount = req.Query["amount"];

    if (BigInteger.TryParse(amount, out BigInteger number))
    {
      return new OkObjectResult(number.ToWords());
    }
    else
    {
      return new BadRequestObjectResult($"Failed to parse: {amount}");
    }
  }
}
```

Тестирование функции

Теперь мы можем протестировать эту функцию.

1. Запустите проект `Northwind.AzureFuncs`. Если вы используете Visual Studio Code, то вам нужно перейти на панель Run and Debug (Запуск и отладка), убе-

диться, что выбрана опция Attach to .NET Functions (Подключиться к функциям .NET), а затем нажать кнопку Run (Выполнить).

2. Обратите внимание, что запускается эмулятор Azure Storage.
3. В Windows, если вы видите предупреждение о безопасности Windows от брандмауэра Windows Defender, нажмите Allow access (Разрешить доступ).
4. Обратите внимание, что Azure Functions Core Tools размещает вашу функцию обычно на порте `7071`:

```
Azure Functions Core Tools
Core Tools Version:       4.0.3743 Commit hash:
44e84987044afc45f0390191bd5d70680a1c544e (64-bit)
Function Runtime Version: 4.0.16281

Functions:
        NumbersToWordsFunction: [GET,POST] http://localhost:7071/api/
NumbersToWordsFunction

For detailed output, run func with --verbose flag.
[2021-09-12T18:44:47.499Z] Worker process started and initialized.
[2021-09-12T18:44:51.038Z] Host lock lease acquired by instance ID '000000
000000000000000000011150C3D'.
```

5. Выберите URL вашей функции и скопируйте его в буфер обмена.
6. Запустите браузер Google Chrome.
7. Вставьте URL в адресное поле, добавьте строку запроса: ?amount=123456 и обратите внимание на успешный ответ (рис. 18.15).

Рис. 18.15. Успешный вызов локально запущенной функции Azure Function

8. В командной строке или терминале обратите внимание, что функция была вызвана успешно, как показано ниже:

```
[2021-09-14T05:58:27.357Z] Executing 'Functions.NumbersToWordsFunction'
(Reason='This function was programmatically called via the host APIs.',
Id=c2c98c67-bf9f-4121-8f7b-701dbc9c0bad)
[2021-09-14T05:58:27.417Z] C# HTTP trigger function processed a request.
[2021-09-14T05:58:27.461Z] Executed 'Functions.NumbersToWordsFunction'
(Succeeded, Id=c2c98c67-bf9f-4121-8f7b-701dbc9c0bad, Duration=111ms)
```

9. Попробуйте вызвать функцию без суммы в строке запроса или с нецелым значением для суммы и обратите внимание, что функция возвращает код состояния `400`, указывающий на плохой запрос (рис. 18.16).

Рис. 18.16. Неверный запрос к локально запущенной функции Azure Function

10. Закройте браузер Google Chrome и завершите работу веб-сервера (или в программе Visual Studio Code остановите отладку).

Публикация проекта Azure Functions в облако

Теперь создадим приложение функции и связанные с ним ресурсы в подписке Azure, затем развернем вашу функцию в облаке и запустим ее там.

Если у вас еще нет учетной записи Azure, то вы можете бесплатно зарегистрироваться по следующей ссылке: https://azure.microsoft.com/en-us/free/.

В программе Visual Studio 2022

Visual Studio имеет графический пользовательский интерфейс для публикации в Azure.

1. На панели Solution Explorer (Проводник решений) щелкните правой кнопкой мыши на проекте `Northwind.AzureFuncs` и выберите команду меню Publish (Опубликовать).

2. Выберите Azure и нажмите кнопку Next (Далее).
3. Выберите Azure Function App (Windows) (Приложение Azure Function (Windows)) и нажмите кнопку Next (Далее).
4. Войдите в систему и введите свои учетные данные.
5. Выберите подписку.
6. В разделе Function Instance (Экземпляр функции) нажмите кнопку +, на которой есть всплывающая подсказка с надписью Create a new Azure Function (Создать новую функцию Azure).
7. Заполните диалоговое окно следующим образом (рис. 18.17):
   1) Name (Имя) — оно должно быть уникальным во всем мире;
   2) Subscription name (Имя подписки) — ваша подписка;
   3) Resource group (Группа ресурсов) — создайте группу ресурсов, чтобы потом упростить последующее удаление всего. Я ввел `cs10dotnet6projects`;
   4) Plan Type: Consumption (Тип плана: Потребление) (платите только за то, что используете);

Рис. 18.17. Создание приложения Azure Function

5) Location (Местоположение) — ближайший к вам центр обработки данных. Я выбрал UK South (Юг Великобритании);
6) Azure Storage (Хранилище Azure) — создайте учетную запись `cs10dotnet6projects` (или другую, уникальную в мировом масштабе — попробуйте добавить свои инициалы) в ближайшем к вам центре обработки данных и выберите тип учетной записи Standard - Locally Redundant Storage (Стандартный — локально резервируемое хранилище).

8. Нажмите кнопку Create (Создать). Этот процесс может занять минуту или больше.
9. В диалоговом окне Publish (Публикация) нажмите Finish (Готово).
10. В окне Publish (Публикация) нажмите кнопку Publish (Опубликовать) (рис. 18.18).

Рис. 18.18. Приложение Azure Function готово к публикации

11. Проанализируйте окно вывода:

```
Build started...
2>------ Publish started: Project: Northwind.AzureFuncs, Configuration:
Release Any CPU ------
2>Northwind.AzureFuncs -> C:\Code\PracticalApps\Northwind.AzureFuncs\bin\
Release\net6.0\Northwind.AzureFuncs.dll
2>Northwind.AzureFuncs -> C:\Code\PracticalApps\Northwind.AzureFuncs\obj\
Release\net6.0\PubTmp\Out\
2>Publishing C:\Code\PracticalApps\Northwind.AzureFuncs\obj\Release\
net6.0\PubTmp\Northwind.AzureFuncs - 20210911153432123.zip to https://
northwindazurefuncs20210911151522.scm.azurewebsites.NET/api/zipdeploy...
2>Zip Deployment succeeded.
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
==========
========== Publish: 1 succeeded, 0 failed, 0 skipped ==========
Waiting for function app ready....
Finished waiting for function app to be ready
```

12. Протестируйте функцию в браузере (рис. 18.19).

Рис. 18.19. Вызов функции Azure Function в облаке

Очистка ресурсов Azure

Вы можете удалить функциональные приложения и связанные с ним ресурсы, чтобы избежать дальнейших расходов. Для этого выполните следующие шаги.

1. В программе Visual Studio Code выберите команду меню View ▸ Command Palette (Вид ▸ Набор команд).
2. Найдите и выберите пункт Azure Functions: Open in portal (Azure Functions: Открыть в портале).
3. Выберите функциональное приложение.
4. На портале Azure в окне функционального приложения Overview (Обзор) выберите Resource Group (Группа ресурсов).
5. Убедитесь, что она содержит только те ресурсы, которые вы хотите удалить.
6. Нажмите кнопку Delete resource group (Удалить группу ресурсов) и примите все остальные подтверждения.

Сервисы идентификации

Сервисы идентификации используются для аутентификации и авторизации пользователей. Важно, чтобы эти сервисы реализовывали открытые стандарты, чтобы вы могли интегрировать разрозненные системы. К распространенным стандартам относятся *OpenID Connect* и *OAuth 2.0*.

Популярной бесплатной реализацией этих стандартов идентификации с открытым исходным кодом является *IdentityServer4*. Он позволяет разработчикам интегрировать аутентификацию на основе токенов, единую регистрацию и контроль доступа к API на сайтах, в сервисах и приложениях.

Microsoft не планирует официально поддерживать IdentityServer4, поскольку «создание и поддержка сервера аутентификации подразумевает занятость 24/7, а у Microsoft уже есть команда и продукт в этой области, Azure Active Directory, который позволяет бесплатно использовать 500 000 объектов».

Вы можете прочитать документацию по IdentityServer4 по следующей ссылке: https://identityserver4.readthedocs.io/.

Краткое описание вариантов выбора специализированных сервисов

Используйте рекомендации для различных сценариев в качестве руководства (табл. 18.6).

Таблица 18.6. Рекомендации для различных сценариев

Сценарий	Рекомендации
Общедоступные сервисы	Сервисы REST, также известные как сервисы на основе HTTP, лучше всего подходят для сервисов, которые должны быть общедоступными, особенно если их нужно вызывать из браузера или даже с мобильного устройства
Общедоступные сервисы данных	OData и GraphQL хорошо подходят для раскрытия сложных иерархических наборов данных, которые могут поступать из различных хранилищ данных. OData разработан и поддерживается компанией Microsoft в официальных пакетах .NET. GraphQL разработан компанией Facebook и поддерживается пакетами сторонних разработчиков
Обмен данными между сервисами	gRPC разработан для взаимодействия с низкой задержкой и высокой пропускной способностью. Отлично подходит для легковесных внутренних микросервисов, где эффективность имеет решающее значение
Взаимодействие «точка — точка» в режиме реального времени	gRPC имеет отличную поддержку двунаправленной потоковой передачи. Службы gRPC могут передавать сообщения в режиме реального времени без опроса. SignalR также подходит для многих видов взаимодействия в режиме реального времени, хотя менее эффективен, чем gRPC
Широковещательное взаимодействие в режиме реального времени	SignalR имеет отличную поддержку для широковещательного взаимодействия в режиме реального времени со многими клиентами
Многоязычные среды	Инструментарий gRPC поддерживает все популярные языки разработки, благодаря чему gRPC хорошо подходит для мультиязычных и платформенных сред
Среды с ограниченной пропускной способностью	Сообщения gRPC сериализуются с помощью Protobuf, облегченного формата сообщений. Сообщение gRPC всегда меньше, чем эквивалентное сообщение JSON
Наносервисы	Azure Functions не нуждается в круглосуточном размещении, поэтому хорошо подходит для наносервисов, которые обычно не нуждаются в постоянной работе

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 18.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. У вас есть приложение, которое взаимодействует с сервисом, созданным с помощью устаревшего сервиса Windows Communication Foundation. Каковы два возможных варианта переноса сервиса и клиента на современную .NET?
2. Какой транспортный протокол использует сервис OData?
3. Почему веб-сервис OData является более гибким, чем традиционный веб-сервис ASP.NET Core Web API?
4. Что нужно сделать с методом действия в контроллере OData, чтобы включить строки запроса для настройки того, что он возвращает?
5. Какой транспортный протокол использует сервис GraphQL?
6. Как определяются контракты в gRPC?
7. Благодаря каким трем преимуществам gRPC хорошо подходит для реализации сервисов?
8. Какие средства передачи данных использует SignalR и какой из них применяется по умолчанию?
9. В чем разница между моделями внутрипроцессного и изолированного хостинга для Azure Functions?
10. Что рекомендуется делать при разработке сигнатур методов RPC?

Упражнение 18.2. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-18---building-and-consuming-other-services, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали, как создавать более специализированные типы сервисов с помощью различных технологий, включая gRPC, SignalR, OData, GraphQL и Azure Functions.

В следующей главе вы узнаете, как создавать кросс-платформенные мобильные и настольные приложения с помощью .NET MAUI.

19 Разработка мобильных и настольных приложений с помощью .NET MAUI

Эта глава посвящена тому, как создавать приложения с *графическим пользовательским интерфейсом* (graphical user interface, GUI) путем создания кросс-платформенного мобильного и настольного приложения для iOS и Android, macOS Catalyst и Windows с использованием .NET MAUI (пользовательского интерфейса мультиплатформенного приложения).

Внимание! Примеры из этой главы были протестированы с помощью .NET Release Candidate 2, .NET MAUI Preview 9 и Visual Studio 2022 Preview 5. В будущих предварительных версиях разработчики, вероятно, исправят некоторые недочеты, а возможно, удалят некий функционал, до выхода версии GA. Для получения последних обновлений, пожалуйста, ознакомьтесь с обновленной главой (на английском) в режиме онлайн по следующей ссылке: https://github.com/markjprice/cs10dotnet6/tree/main/docs/chapter19.

Вы узнаете, как *расширяемый язык разметки приложений* (eXtensible Application Markup Language, XAML) упрощает определение пользовательского интерфейса для графического приложения.

Рассказать все о кросс-платформенной разработке GUI в одной главе невозможно, однако, как и веб-разработка, она настолько важна в наши дни, что я хотел бы познакомить вас с ее основами. Рассматривайте эту главу как введение, в котором вы познакомитесь с базовыми положениями, а более подробную информацию сможете найти в книге, посвященной разработке мобильных и настольных приложений.

Приложение позволит вносить списки клиентов в базу данных Northwind и управлять ими. Мобильное приложение будет обращаться к веб-сервису Northwind, который вы создали с помощью ASP.NET Core Web API в главе 16. Если вы еще не сделали этого, то вернитесь к данной главе и создайте сейчас или скачайте из репозитория GitHub по ссылке: https://github.com/markjprice/cs10dotnet6.

Хотя вы можете создать проект .NET MAUI в командной строке, а затем изменить его с помощью Visual Studio Code, официальных инструментов, которые могли бы вам помочь, пока нет. Ожидается, что они появятся в .NET 7.0 в конце 2022 года.

В этой главе:

- замечания по поводу отложенного выпуска .NET MAUI;
- знакомство с XAML;
- знакомство с .NET MAUI;
- разработка мобильных и настольных приложений с помощью .NET MAUI;
- взаимодействие приложения .NET MAUI с веб-сервисами.

Замечания по поводу отложенного выпуска .NET MAUI

Четырнадцатого сентября 2021 года компания Microsoft объявила, что выпуск .NET MAUI будет отложен. «К сожалению, .NET MAUI не будет готов к выпуску вместе с .NET 6 GA в ноябре», — прокомментировал Скотт Хантер, директор по управлению программами .NET. Вы можете прочитать объявление Скотта по следующей ссылке: https://devblogs.microsoft.com/dotnet/update-on-dotnet-maui/.

Ниже приводится приблизительный график выпуска предварительных версий и релиз-кандидатов, а затем и выхода .NET MAUI во втором квартале 2022 года:

- 12 октября 2021 года: .NET MAUI Preview 9 и .NET 6 Release Candidate 2, которые использовались для этой главы, опубликованы в печатных и электронных изданиях этой книги;
- 9 ноября 2021 года: .NET MAUI Preview 10 и .NET 6 GA;
- декабрь 2021 года: .NET MAUI Preview 11;
- январь 2022 года: .NET MAUI Preview 12;
- февраль 2022 года: .NET MAUI Preview 13;
- март 2022 года: .NET MAUI Release Candidate 1;
- апрель 2022 года: .NET MAUI Release Candidate 2;
- май 2022 года: .NET MAUI General Availability на Microsoft Build;
- ноябрь 2022 года: .NET MAUI включен в .NET 7.

Я хотел включить эту главу в печатную книгу, несмотря на то что после публикации некоторые ее части могут измениться. Чтобы поддерживать актуальность главы по мере выхода предварительных версий .NET MAUI, я планирую обновлять ее в репозитории GitHub для этой книги вплоть до выхода GA. Онлайн-версия данной главы на английском языке доступна по следующей ссылке: https://github.com/markjprice/cs10dotnet6/tree/main/docs/chapter19.

Начнем с рассмотрения языка разметки, используемого в .NET MAUI.

Знакомство с XAML

В 2006 году Microsoft выпустила *Windows Presentation Foundation (WPF)*, которая стала первой технологией, использующей *XAML* (eXtensible Application Markup Language, расширяемый язык разметки приложений). Вскоре после этого была разработана технология Silverlight для веб- и мобильных приложений, но она больше не поддерживается Microsoft. WPF до настоящего времени используется для создания настольных приложений Windows; в качестве примера можно привести Visual Studio для Windows.

XAML можно использовать для создания частей таких приложений, как:

- *приложения .NET MAUI* для мобильных и настольных устройств, включая Android, iOS, Windows и macOS. Они являются продуктом развития технологии *Xamarin.Forms*;
- *приложения WinUI 3* для устройств с Windows 10 и 11;
- *приложения Universal Windows Platform (UWP)* для устройств с Windows 10 и 11, Xbox One и Mixed Reality;
- *WPF-приложения* для рабочего стола Windows, включая версии Windows 7 и более поздние;
- *приложения Avalonia и Uno Platform*, использующие кросс-платформенные технологии сторонних производителей.

Упрощение кода с помощью XAML

XAML упрощает код C#, особенно при создании пользовательского интерфейса.

Представьте, что для создания панели инструментов вам необходимо добавить две или более кнопки, расположенные горизонтально.

В C# вы можете написать следующий код:

```
StackPanel toolbar = new();
toolbar.Orientation = Orientation.Horizontal;

Button newButton = new();
newButton.Content = "New";
newButton.Background = new SolidColorBrush(Colors.Pink);
toolbar.Children.Add(newButton);

Button openButton = new();
openButton.Content = "Open";
openButton.Background = new SolidColorBrush(Colors.Pink);
toolbar.Children.Add(openButton);
```

В XAML данный код можно упростить. В процессе обработки этого XAML устанавливаются эквивалентные свойства и вызываются методы для достижения той же цели, которая стояла перед кодом C#:

```
<StackPanel Name="toolbar" Orientation="Horizontal">
  <Button Name="newButton" Background="Pink">New</Button>
  <Button Name="openButton" Background="Pink">Open</Button>
</StackPanel>
```

Вы можете рассматривать XAML как альтернативный и более легкий способ объявления и создания экземпляров типов .NET, особенно при определении пользовательского интерфейса и ресурсов, которые он использует.

XAML позволяет объявлять ресурсы, такие как кисти, стили и темы, на разных уровнях, например на уровне элемента UI, страницы или глобально для приложения, чтобы обеспечить совместное использование ресурсов.

XAML также позволяет выполнять привязку данных между элементами пользовательского интерфейса или между элементами UI и объектами и коллекциями.

Выбор общих элементов управления

Существует множество предопределенных элементов управления, с помощью которых вы можете создавать общие сценарии пользовательского интерфейса. Почти все диалекты XAML поддерживают эти элементы управления (табл. 19.1).

Таблица 19.1. Элементы управления

Элементы управления	Описание
`Button`, `ImageButton`, `Menu`, `Toolbar`	Выполнение определенных 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, которые по-разному влияют на свои дочерние элементы
`Label`, `TextBlock`	Отображение текста только для чтения
`RichTextBox`, `TextBox`	Редактирование текста
`Image`, `MediaElement`	Встраивание изображений, видео и аудиофайлов
`DataGrid`	Просмотр и редактирование данных максимально быстро и легко
`Scrollbar`, `Slider`, `StatusBar`	Различные элементы пользовательского интерфейса

Расширения разметки

Для поддержки некоторых дополнительных функций XAML использует расширения разметки. Некоторые из наиболее важных из них включают привязку элементов и данных, а также повторное использование ресурсов:

- `{Binding}` связывает элемент со значением из другого элемента или источника данных;
- `{StaticResource}` связывает элемент с общим ресурсом;
- `{ThemeResource}` связывает элемент с общим ресурсом, определенным в теме.

В данной главе вы изучите несколько практических примеров расширений разметки.

Знакомство с .NET MAUI

Создать мобильное приложение, работающее только на iPhone, вы можете с помощью Objective-C или Swift и UIKit, используя инструмент разработки Xcode.

Создать мобильное приложение, работающее только на телефонах с Android, вы можете с помощью Java или Kotlin и Android SDK, используя инструмент разработки Android Studio.

Но что, если вам необходимо создать мобильное приложение, которое может работать на iPhone *и* телефонах с Android? И вы хотите создать его только один раз, используя знакомый вам язык программирования и платформу разработки? А что, если, приложив чуть больше усилий для адаптации пользовательского интерфейса к экранам размером с настольный монитор, вы могли бы также ориентироваться на настольные компьютеры macOS и Windows?

.NET MAUI позволяет разработчикам создавать на языке C# кросс-платформенные мобильные приложения для Apple iOS (iPhone) и iPadOS, macOS с помощью Catalyst, для Windows с помощью WinUI 3 и для Google Android с помощью C# и .NET, которые затем компилируются в собственные API и запускаются на собственных телефонных и настольных платформах.

Уровень бизнес-логики можно написать один раз и использовать на всех платформах. Взаимодействие с пользовательским интерфейсом и API различается на разных мобильных и настольных платформах, поэтому уровень UI иногда настраивается для каждой платформы.

Подобно WPF- и UWP-приложениям, в .NET MAUI язык разметки XAML служит для однократного определения пользовательского интерфейса для всех платформ сразу, абстрагируя платформенно-зависимые компоненты UI. Приложения, созданные с помощью .NET MAUI, формируют UI на основе собственных виджетов

платформы, поэтому приложения удобны и привлекательны, а также идеально подходят под целевую мобильную платформу.

UI, созданный с помощью .NET MAUI, не будет идеально подходить для конкретной платформы так же, как если бы это было сделано на заказ с использованием собственных инструментов для этой платформы, но для мобильных и настольных приложений этого вполне достаточно.

Инструменты разработки для мобильных и облачных технологий

Мобильные приложения часто взаимодействуют с облачными сервисами.

Сатья Наделла, генеральный исполнительный директор Microsoft, однажды сказал: *«Для меня стратегия mobile first означает не мобильность устройств, а мобильность индивидуального опыта. [...] Единственный способ организовать мобильность этих приложений и данных — использовать облако»*.

Как и раньше, воспользуемся Visual Studio Code для создания сервиса ASP.NET Core Web API для поддержки мобильного приложения. Создавать приложения .NET MAUI разработчики могут, используя программу Visual Studio 2022 для Windows или Visual Studio 2022 для Mac.

При установке Visual Studio 2022 необходимо установить флажок .NET MAUI (Preview), который относится к нагрузке Mobile development with .NET, как показано на рис. 19.1.

Рис. 19.1. Выбор рабочей нагрузки .NET MAUI для Visual Studio 2022

Создание приложений для iOS и macOS в Windows

Если вы хотите с помощью Visual Studio 2022 для Windows создавать мобильное приложение для iOS или настольное приложение для macOS Catalyst, то можете подключиться по сети к хосту сборки Mac. Инструкции можно найти по следующей ссылке: https://docs.microsoft.com/en-us/xamarin/ios/get-started/installation/windows/connecting-to-mac/.

Дополнительная функциональность

Создадим кросс-платформенное мобильное и настольное приложение, используя множество навыков и знаний, полученных в предыдущих главах. Кроме того, применим некоторые незнакомые функции.

MVVM

Model-View-ViewModel (MVVM) — это паттерн проектирования, подобный MVC. Буквы в аббревиатуре означают следующее:

- *Model* (Модель) — сущностный класс, который представляет объект данных в хранилище, например в реляционной базе данных;
- *View* (Представление) — способ представления данных в графическом пользовательском интерфейсе, включающий поля для отображения и редактирования данных, а также кнопки и другие элементы для взаимодействия с данными;
- *ViewModel* — класс, представляющий поля данных, действия и события, которые затем могут быть привязаны к таким элементам, как текстовые поля и кнопки в представлении.

В MVC модели, передаваемые в представление, доступны только для чтения, поскольку передаются в представление только в одном направлении. Вот почему неизменяемые записи хорошо подходят для моделей MVC. Но классы ViewModel — совсем другие. Они должны поддерживать двустороннее взаимодействие, и если исходные данные изменяются в течение жизни объекта, представление должно динамически обновляться.

Интерфейс INotifationPropertyChanged

Интерфейс `INotifyPropertyChanged` позволяет классу модели поддерживать двустороннюю привязку данных. Он требует от класса реализации события `PropertyChanged` с параметром типа `PropertyChangedEventArgs`:

```
namespace System.ComponentModel
{
  public class PropertyChangedEventArgs : EventArgs
```

```
  {
    public PropertyChangedEventArgs(string? propertyName);
    public virtual string? PropertyName { get; }
  }

  public delegate void PropertyChangedEventHandler(
    object? sender, PropertyChangedEventArgs e);

  public interface INotifyPropertyChanged
  {
    event PropertyChangedEventHandler PropertyChanged;
  }
}
```

Внутри каждого свойства в классе при установке нового значения вам необходимо инициировать событие (если оно не равно `null`) с экземпляром `PropertyChangedEventArgs`, содержащим имя свойства в виде значения `string`:

```
private string companyName;

public string CompanyName
{
  get => companyName;
  set
  {
    companyName = value; // сохраняем новое установленное значение
    PropertyChanged?.Invoke(this,
      new PropertyChangedEventArgs(nameof(CompanyName)));
  }
}
```

Будучи привязанным к свойству данных, элемент управления пользовательского интерфейса автоматически обновится, чтобы показать новое значение при его изменении.

Чтобы упростить реализацию, мы можем использовать функцию компилятора для получения имени свойства, дополнив параметр `string` атрибутом `[CallerMemberName]`:

```
private void NotifyPropertyChanged(
  [CallerMemberName] string propertyName = "")
{
  // если установлен обработчик события, вызываем делегат
  // и передаем имя свойства
  PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs(propertyName));
}

public string CompanyName
{
```

```
  get => companyName;
  set
  {
    companyName = value; // сохраняем новое установленное значение
    NotifyPropertyChanged(); // имя вызывающего абонента: "CompanyName"
  }
}
```

Класс ObservableCollection

С `INotifyPropertyChanged` связан интерфейс `INotifyCollectionChanged`, который реализуется классом `ObservableCollection<T>`. Он выдает уведомления при добавлении, удалении элементов или при обновлении коллекции. При привязке к элементам управления типа `ListView` или `TreeView` пользовательский интерфейс будет динамически обновляться, отражая изменения.

Сервисы зависимостей

Мобильные платформы, такие как iOS и Android, и настольные платформы, такие как Windows и macOS, по-разному реализуют общие функции, вследствие чего необходим способ реализовать общие функции на уровне платформы. Мы можем сделать это с помощью сервиса зависимостей.

- Определите интерфейс для общей функции, например `IDialer` для компонента набора телефонного номера на телефонном устройстве или `INotificationManager` для всплывающего локального уведомления на настольных и мобильных устройствах.
- Реализуйте интерфейс для всех платформ, которые вам необходимо поддерживать, например iOS и Android для компонента набора телефонного номера, и зарегистрируйте реализации с помощью атрибута:

```
[assembly: Dependency(typeof(PhoneDialer))]

namespace Northwind.Maui.iOS
{
  public class PhoneDialer : IDialer
```

- Получите платформенно-зависимую реализацию интерфейса, используя сервис зависимостей:

```
IDialer dialer = DependencyService.Get<IDialer>();
```

.NET MAUI Essentials включает компонент PhoneDialer, поэтому мы будем использовать его в нашем проекте вместо того, чтобы определять собственный сервис зависимости для набора телефонного номера.

Компоненты пользовательского интерфейса .NET MAUI

.NET MAUI содержит некоторые общие элементы управления, предназначенные для создания пользовательских интерфейсов. Эти элементы делятся на четыре категории:

- *страницы* представляют кросс-платформенные экраны приложений, например `ContentPage`, `NavigationPage`, `FlyoutPage`, и `TabbedPage`;
- *макеты* представляют структуру комбинации других компонентов пользовательского интерфейса, например `Grid`, `StackLayout` и `FlexLayout`;
- *представления* определяют отдельный компонент пользовательского интерфейса, например `CarouselView`, `CollectionView`, `Label`, `Entry`, `Editor` и `Button`;
- *ячейки* представляют отдельный элемент в списке или таблице, например `TextCell`, `ImageCell`, `SwitchCell` и `EntryCell`.

Вы можете отслеживать статус выполнения миграции компонентов .NET MAUI по следующей ссылке: https://github.com/dotnet/maui/wiki/Status.

Представление ContentPage

Представление `ContentPage` предназначено для простых пользовательских интерфейсов. Оно содержит свойство `ToolbarItems`, отображающее действия, которые пользователь может выполнять на платформе. Каждый параметр `ToolbarItem` может содержать значок и текст:

```
<ContentPage.ToolbarItems>
  <ToolbarItem Text="Add" Activated="Add_Activated"
    Order="Primary" Priority="0" />
  ...
</ContentPage.ToolbarItems>
```

Элемент управления ListView

Элемент управления `ListView` используется для длинных списков значений одного типа, связанных по данным. Может содержать верхние и нижние колонтитулы, а его элементы могут быть сгруппированы.

Этот элемент содержит ячейки для каждого элемента списка. Существует два встроенных типа ячеек: текст и изображение. Разработчики могут определять пользовательские типы ячеек.

Для ячеек могут быть определены действия контекста, которые появляются, когда ячейка пролистывается смахиванием влево на iPhone или долгим нажатием

в Android. Контекстное действие, которое является деструктивным, может быть отображено красным цветом.

```
<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>
```

Элементы управления Entry и Editor

Элементы управления `Entry` и `Editor` применяются для редактирования текстовых значений и часто привязаны по данным к свойству модели объекта:

```
<Editor Text="{Binding CompanyName, Mode=TwoWay}" />
```

Используйте элемент `Entry` для одной строки текста, а элемент `Editor` — для нескольких строк.

Обработчики .NET MAUI

В .NET MAUI элементы управления XAML определены в пространстве имен `Microsoft.Maui.Controls`. Компоненты, называемые *обработчиками*, сопоставляют эти общие элементы управления с нативными элементами управления на каждой платформе. В iOS обработчик сопоставляет элемент .NET MAUI `Button` с `UIButton`, определенным UIKit для iOS. В macOS `Button` сопоставляется с `NSButton`, определенным AppKit. В Android `Button` сопоставляется с нативным для Android элементом `AppCompatButton`.

Обработчики имеют свойство `NativeView`, которое раскрывает базовый нативный элемент управления. Это позволяет вам работать с платформенно-зависимыми функциями, такими как свойства, методы и события, и настраивать все экземпляры нативного элемента управления.

Написание платформенно-зависимого кода

Если вам нужно написать операторы, которые будут выполняться только для определенной платформы, например Android, вы можете использовать директивы компилятора.

Например, по умолчанию элементы управления вводом `Entry` в Android сопровождают символы подчеркивания.

Если вы хотите скрыть подчеркивание, то можете написать Android-зависимый код, чтобы получить обработчик для элемента управления `Entry`, использовать его

свойство `NativeView` для доступа к базовому нативному элементу управления, а затем установить свойство, управляющее этой функцией, в значение `false`:

```
#if __ANDROID__
  Handlers.EntryHandler.EntryMapper[nameof(IEntry.BackgroundColor)] = (h, v) =>
  {
    (h.NativeView as global::Android.Views.Entry).UnderlineVisible = false;
  };
#endif
```

Предопределенные константы компилятора включают в себя:

- `__ANDROID__`;
- `__IOS__`;
- `WINDOWS`.

Синтаксис оператора `#if` компилятора немного отличается от синтаксиса оператора `if` в C#:

```
#if __IOS__
  // iOS-зависимые операторы
#elif __ANDROID__
  // Android-зависимые операторы
#elif WINDOWS
  // Windows-зависимые операторы
#endif
```

Разработка мобильных и настольных приложений с помощью .NET MAUI

Создадим мобильное и настольное приложение для управления клиентами в Northwind.

Если вы никогда не запускали среду разработки Xcode, то запустите ее сейчас, чтобы увидеть окно Start (Пуск) и убедиться, что все необходимые компоненты установлены и зарегистрированы. Если не запускаете Xcode, то можете позже получить ошибки в среде Visual Studio для Mac.

Создание виртуального устройства Android для локального тестирования приложений

Для разработки под Android необходимо установить хотя бы один Android SDK. Установка по умолчанию Visual Studio с рабочей нагрузкой на разработку мобильных устройств уже включает в себя один Android SDK, но часто это довольно

старая версия, предназначенная для поддержки как можно большего количества устройств с Android.

Чтобы использовать последние функции .NET MAUI, вам необходимо установить более позднюю версию Android SDK.

1. В Windows запустите программу Visual Studio 2022.
2. Выберите команду меню Tools ▸ Android ▸ Android Device Manager (Инструменты ▸ Android ▸ Диспетчер устройств Android).
3. В Android Device Manager (Диспетчер устройств Android) нажмите кнопку + New (Создать), чтобы создать новое устройство.
4. В диалоговом окне New Device (Новое устройство) выберите следующие параметры:
   1) Base Device (Основное устройство): Pixel 2 (+ Store);
   2) Processor (Процессор): x86;
   3) OS (ОС): Pie 9.0 – API 28.
5. Нажмите кнопку Create (Создать).
6. Примите лицензионные соглашения.
7. Дождитесь скачивания всех необходимых файлов.
8. В Android Device Manager (Диспетчер устройств Android) в списке устройств в строке для устройства, которое вы только что создали, нажмите кнопку Start (Пуск).
9. Когда устройство Android завершит запуск, откройте браузер и проверьте, есть ли у него доступ к сети.
10. Закройте эмулятор.
11. Перезапустите Visual Studio 2022, чтобы убедиться, что программа поддерживает новый эмулятор.

Создание решения .NET MAUI

Сейчас мы создадим проект для кросс-платформенного мобильного и настольного приложения.

1. В программе Visual Studio для Windows создайте проект с такими настройками:
   1) шаблон проекта: .NET MAUI App (Preview)/`maui`;
   2) файл и папка рабочей области/решения: `PracticalApps`;
   3) файл и папка проекта: `Northwind.Maui.Customers`.

2. Откройте файл проекта и раскомментируйте элемент для включения таргетинга на Windows, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>net6.0-ios;net6.0-android;net6.0-maccatalyst</
    TargetFrameworks>
    <TargetFrameworks Condition="$([MSBuild]::IsOSPlatform('windows'))
    and '$(MSBuildRuntimeType)' == 'Full'">$(TargetFrameworks);net6.0-
    windows10.0.19041</TargetFrameworks>
    <OutputType>Exe</OutputType>
    <RootNamespace>Northwind.Maui.Customers</RootNamespace>
    <UseMaui>true</UseMaui>
    <SingleProject>true</SingleProject>
```

3. Справа от кнопки Run (Запуск) на панели инструментов присвойте параметру Framework значение net6.0-android и выберите образ эмулятора Pixel 2 - API 28 (Android 9.0 - API 28), который вы создали ранее (рис. 19.2).

Рис. 19.2. Выбор Android в качестве цели для запуска

4. Нажмите кнопку Run (Запуск) на панели инструментов и дождитесь, пока эмулятор устройства запустит операционную систему Android и ваше мобильное приложение.
5. В приложении .NET MAUI нажмите кнопку Click me (Нажми меня), чтобы увеличить показания счетчика в три раза (рис. 19.3).
6. Обратите внимание на окно XAML Live Preview (Динамический просмотр XAML) в программе Visual Studio и на то, что подключена функция XAML Hot Reload (Горячая перезагрузка XAML), чтобы вы могли вносить изменения в XAML и видеть их отражение в приложении, не перезапуская его. Например, попробуйте изменить текст метки Hello, World! на какой-нибудь другой, сохраните XAML-файл и нажмите кнопку Hot Reload (Горячая перезагрузка) на панели инструментов.

Рис. 19.3. Увеличение показаний счетчика в приложении Android .NET MAUI

7. Закройте симулятор устройства Android.
8. Выберите команду меню Build ▸ Configuration Manager (Сборка ▸ Диспетчер конфигурации).
9. В строке для проекта Northwind.Maui.Customers установите флажок Deploy (Развертывание) (рис. 19.4).

Рис. 19.4. Включение приложения Windows для развертывания на компьютере Windows

10. Справа от кнопки Run (Запуск) на панели инструментов присвойте параметру Framework значение net6.0-windows, а затем выберите Windows Machine.
11. Убедитесь, что выбрана конфигурация Debug (Отладка), а затем нажмите кнопку запуска с зеленым треугольником и меткой Windows Machine.
12. Через несколько мгновений обратите внимание, что приложение Windows отображается с той же кнопкой Click me (Нажми меня) и функцией счетчика (рис. 19.5).
13. Закройте приложение Windows.

Рис. 19.5. Увеличение показаний счетчика в приложении Windows .NET MAUI

Создание модели представления с двусторонней привязкой данных

Нам нужно создать модель представления, которая позволит нам показывать и изменять сущностный класс `Customer`, поэтому он должен реализовать двустороннюю привязку данных.

1. В папке проекта `Northwind.Maui.Customers` создайте два класса, один — `CustomerDetailViewModel.cs` для отображения сведений об одном клиенте, а другой — `CustomersListViewModel.cs` для отображения списка клиентов.
2. В классе `CustomerDetailViewModel.cs` измените операторы так, чтобы определить класс, который реализует интерфейс `INotifyPropertyChanged` и содержит шесть свойств:

```
using System.ComponentModel; // INotifyPropertyChanged
using System.Runtime.CompilerServices; // [CallerMemberName]

namespace Northwind.Maui.Customers;

public class CustomerDetailViewModel : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  private string customerId;
  private string companyName;
  private string contactName;
  private string city;
  private string country;
  private string phone;
```

```
// этот атрибут устанавливает параметр propertyName,
// используя контекст, в котором вызывается этот метод
private void NotifyPropertyChanged(
  [CallerMemberName] string propertyName = "")
{
  // если установлен обработчик события, вызываем делегат
  // и передаем имя свойства
  PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs(propertyName));
}

public string CustomerId
{
  get => customerId;
  set
  {
    customerId = value;
    NotifyPropertyChanged();
  }
}

public string CompanyName
{
  get => companyName;
  set
  {
    companyName = value;
    NotifyPropertyChanged();
  }
}

public string ContactName
{
  get => contactName;
  set
  {
    contactName = value;
    NotifyPropertyChanged();
  }
}

public string City
{
  get => city;
  set
  {
    city = value;
    NotifyPropertyChanged();
    NotifyPropertyChanged(nameof(Location));
  }
}
```

```
  public string Country
  {
    get => country;
    set
    {
      country = value;
      NotifyPropertyChanged();
      NotifyPropertyChanged(nameof(Location));
    }
  }

  public string Phone
  {
    get => phone;
    set
    {
      phone = value;
      NotifyPropertyChanged();
    }
  }

  public string Location
  {
    get => $"{City}, {Country}";
  }
}
```

Обратите внимание на следующие моменты:

- класс реализует интерфейс `INotifyPropertyChanged`, в связи с чем элемент управления с двухсторонней привязкой, такой как `Editor`, будет обновлять свойство, и наоборот. Событие `PropertyChanged` возникает всякий раз, когда одно из свойств изменяется с помощью частного метода `NotifyPropertyChanged`, упрощающего реализацию;
- помимо свойств для сохранения значений, полученных из HTTP-сервиса, класс определяет свойство `Location`, доступное только для чтения. Оно будет использоваться для привязки в сводном списке клиентов в целях отображения местоположения каждого из них. При изменении свойства `City` или `Country` нам также необходимо уведомлять об изменении `Location`, иначе все представления, привязанные к `Location`, будут обновляться некорректно.

3. В классе `CustomersListViewModel.cs` измените операторы, чтобы определить класс, который наследуется от `ObservableCollection<T>` и имеет метод для заполнения данных выборки:

```
using System.Collections.ObjectModel; // ObservableCollection<T>

namespace Northwind.Maui.Customers;

public class CustomersListViewModel :
```

```
  ObservableCollection<CustomerDetailViewModel>
{
  // тестирование перед вызовом веб-сервиса
  public void AddSampleData(bool clearList = true)
  {
    if (clearList) Clear();

    Add(new CustomerDetailViewModel
    {
      CustomerId = "ALFKI",
      CompanyName = "Alfreds Futterkiste",
      ContactName = "Maria Anders",
      City = "Berlin",
      Country = "Germany",
      Phone = "030-0074321"
    });

    Add(new CustomerDetailViewModel
    {
      CustomerId = "FRANK",
      CompanyName = "Frankenversand",
      ContactName = "Peter Franken",
      City = "München",
      Country = "Germany",
      Phone = "089-0877310"
    });

    Add(new CustomerDetailViewModel
    {
      CustomerId = "SEVES",
      CompanyName = "Seven Seas Imports",
      ContactName = "Hari Kumar",
      City = "London",
      Country = "UK",
      Phone = "(171) 555-1717"
    });
  }
}
```

Обратите внимание на следующие моменты:

- после загрузки из сервиса, который будет реализован позже в этой главе, клиенты кэшируются локально с помощью `ObservableCollection<T>`. Так поддерживаются уведомления для любых связанных компонентов пользовательского интерфейса, таких как `ListView`, чтобы UI мог перерисовать себя, когда базовые данные добавляют или удаляют элементы из коллекции;
- в целях тестирования, когда HTTP-сервис недоступен, существует статический метод для заполнения трех образцов клиентов.

Создание представлений для списка клиентов и подробной информации о клиенте

Заменим существующий код в файле `MainPage` на представление для отображения списка клиентов и сведений о клиенте.

1. В проекте `Northwind.Maui.Customers` удалите файл `MainPage.xaml`.
2. Откройте файл `App.xaml` и добавьте стиль для применения того же цвета фона и семейства шрифта к элементам управления `Entry`, которые применяются к элементам управления `Label`, как показано ниже:

```
<Style TargetType="Entry">
  <Setter Property="TextColor" Value="{DynamicResource PrimaryTextColor}"/>
  <Setter Property="FontFamily" Value="OpenSansRegular" />
  <Setter Property="HorizontalOptions" Value="StartAndExpand" />
  <Setter Property="WidthRequest" Value="300" />
</Style>
```

Реализация представления списка клиентов

Сначала мы создадим два представления для списка клиентов и показа подробной информации об одном клиенте, а затем реализуем список клиентов.

1. Щелкните правой кнопкой мыши на папке проекта `Northwind.Maui.Customers`, выберите команду меню Add ▸ New Item (Добавить ▸ Новый элемент), выберите Content Page (Страница содержимого), введите имя `CustomersListPage` и нажмите кнопку Add (Добавить) (рис. 19.6).

Рис. 19.6. Добавление нового элемента страницы содержимого XAML

2. Щелкните правой кнопкой мыши на папке Views, выберите команду меню Add ▸ New Item (Добавить ▸ Новый элемент), выберите Content Page (Страница содержимого), введите имя `CustomerDetailPage` и нажмите кнопку Add (Добавить).

На момент написания этой книги в программе Visual Studio 2022 нет шаблонов элементов проекта для .NET MAUI. Шаблон элемента проекта ContentPage предназначен для более старой версии Xamarin.Forms. На следующем этапе мы все равно заменим почти всю разметку и код, так что это не проблема.

3. Откройте файл `CustomersListPage.xaml` и измените его содержимое:

```
<ContentPage
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="Northwind.Maui.Customers.CustomersListPage"
  BackgroundColor="{DynamicResource PageBackgroundColor}"
  Title="List">
    <ContentPage.Content>
      <ListView ItemsSource="{Binding .}"
                VerticalOptions="Center"
                HorizontalOptions="Center"
                IsPullToRefreshEnabled="True"
                ItemTapped="Customer_Tapped"
                Refreshing="Customers_Refreshing">
        <ListView.Header>
          <StackLayout Orientation="Horizontal">
            <Label Text="Northwind Customers"
                   FontSize="Subtitle" Margin="10" />
            <Button Text="Add" Clicked="Add_Clicked" />
          </StackLayout>
        </ListView.Header>
        <ListView.ItemTemplate>
          <DataTemplate>
            <TextCell Text="{Binding CompanyName}"
                      Detail="{Binding Location}"
                      TextColor="{DynamicResource PrimaryTextColor}"
                      DetailColor="{DynamicResource PrimaryTextColor}" >
            <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>
```

Обратите внимание на следующие моменты:

- класс `ContentPage` содержит атрибут `Title` со значением `List`;
- класс `ListView` содержит атрибут `IsPullToRefreshEnabled` со значением `true`;
- были написаны обработчики для следующих событий:
  - `Customer_Tapped` — производится касание записи клиента, чтобы посмотреть сведения о нем;
  - `Customers_Refreshing` — список тянется вниз для обновления элементов;
  - `Customer_Phoned` — ячейка смахивается влево на iPhone или долго нажимается в Android, и далее производится касание кнопки `Phone` (Позвонить);
  - `Customer_Deleted` — ячейка смахивается влево на iPhone или долго нажимается в Android, и далее производится касание кнопки `Delete` (Удалить);
  - `Add_Clicked` — производится нажатие кнопки `Add` (Добавить);
- шаблон данных определяет способ отображения сведений о каждом клиенте: более крупный шрифт для названия компании и мелкий — для ее адреса;
- кнопка `Add` (Добавить) находится в заголовке представления списка, что позволяет пользователям перейти к представлению подробной информации и добавить нового клиента.

4. Найдите и откройте файл `CustomersListPage.xaml.cs` и измените его содержимое:

```
using Microsoft.Maui.Controls; // ContentPage, ListView
using Microsoft.Maui.Essentials; // PhoneDialer
using System;
using System.Threading.Tasks;

namespace Northwind.Maui.Customers;

public partial class CustomersListPage : ContentPage
{
  public CustomersListPage()
  {
    InitializeComponent();

    CustomersListViewModel viewModel = new();
    viewModel.AddSampleData();
    BindingContext = viewModel;
  }

  async void Customer_Tapped(object sender, ItemTappedEventArgs e)
  {
    if (e.Item is not CustomerDetailViewModel c) return;
```

```
    // переходим к подробному представлению и показываем
    // подключенного клиента
    await Navigation.PushAsync(new CustomerDetailPage(
      BindingContext as CustomersListViewModel, c));
  }

  async void Customers_Refreshing(object sender, EventArgs e)
  {
    if (sender is not ListView listView) return;

    listView.IsRefreshing = true;

    // имитируем обновление
    await Task.Delay(1500);

    listView.IsRefreshing = false;
  }

  void Customer_Deleted(object sender, EventArgs e)
  {
    MenuItem menuItem = sender as MenuItem;
    if (menuItem.BindingContext is not CustomerDetailViewModel c) return;
    (BindingContext as CustomersListViewModel).Remove(c);
  }

  async void Customer_Phoned(object sender, EventArgs e)
  {
    MenuItem menuItem = sender as MenuItem;
    if (menuItem.BindingContext is not CustomerDetailViewModel c) return;

    if (await DisplayAlert("Dial a Number",
      "Would you like to call " + c.Phone + "?",
      "Yes", "No"))
    {
      PhoneDialer.Open(c.Phone);
    }
  }

  async void Add_Clicked(object sender, EventArgs e)
  {
    await Navigation.PushAsync(new CustomerDetailPage(
      BindingContext as CustomersListViewModel));
  }
}
```

Обратите внимание на следующие моменты:

- значение `BindingContext` устанавливается на экземпляр `CustomersViewModel`, который заполняется данными образца в конструкторе страницы;
- при касании записи о клиенте в представлении списка пользователь попадает в представление сведений (которое вы реализуете на следующем шаге);

- опускаясь, представление списка запускает смоделированное обновление, которое занимает 1,5 с;
- при удалении из представления списка клиент удаляется из связанной модели представления клиентов;
- когда запись о пользователе в представлении списка смахивается, а кнопка Phone (Позвонить) нажата, в диалоговом окне пользователю будет предложено набрать номер. Если он согласится, то встроенная в платформу реализация будет получена с помощью разрешения зависимостей, а затем использована для набора номера;
- при нажатии кнопки Add (Добавить) пользователь перенаправляется на страницу сведений о клиенте для ввода сведений о новом клиенте.

Реализация подробного представления о клиенте

Далее мы реализуем подробное представление о клиенте.

1. Откройте файл `CustomerDetailPage.xaml` и измените его содержимое, как показано в коде ниже. Обратите внимание на следующие моменты:
   - элемент `Title` страницы содержимого имеет значение `Edit`;
   - макет построен на элементе `Grid` для вывода клиентов, который содержит два столбца и шесть строк;
   - представление `Entry` двусторонне связано по данным со свойствами класса `CustomerViewModel`;
   - для `InsertButton` определен обработчик событий для выполнения кода, добавляющего нового клиента.

```
<ContentPage
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="Northwind.Maui.Customers.Views.CustomerDetailPage"
  BackgroundColor="{DynamicResource PageBackgroundColor}"
  Title="Edit">

  <ContentPage.Content>
    <StackLayout VerticalOptions="Fill" HorizontalOptions="Fill">
      <Grid ColumnDefinitions="Auto,Auto"
            RowDefinitions="Auto,Auto,Auto,Auto,Auto,Auto">
        <Label Text="Customer Id" VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding CustomerId, Mode=TwoWay}" Grid.Column="1"
               MaxLength="5" TextTransform="Uppercase" />
        <Label Text="Company Name" Grid.Row="1"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding CompanyName, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="1" />
        <Label Text="Contact Name" Grid.Row="2"
```

```
                VerticalOptions="Center" Margin="6" />
            <Entry Text="{Binding ContactName, Mode=TwoWay}"
                Grid.Column="1" Grid.Row="2" />
            <Label Text="City" Grid.Row="3"
                VerticalOptions="Center" Margin="6" />
            <Entry Text="{Binding City, Mode=TwoWay}"
                Grid.Column="1" Grid.Row="3" />
            <Label Text="Country" Grid.Row="4"
                VerticalOptions="Center" Margin="6" />
            <Entry Text="{Binding Country, Mode=TwoWay}"
                Grid.Column="1" Grid.Row="4" />
            <Label Text="Phone" Grid.Row="5"
                VerticalOptions="Center" Margin="6" />
            <Entry 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>
```

2. Откройте файл `CustomerDetailPage.xaml.cs` и измените его содержимое:

```
using Microsoft.Maui.Controls;
using System;
using System.Threading.Tasks;

namespace Northwind.Maui.Customers;

public partial class CustomerDetailPage : ContentPage
{
  private CustomersListViewModel customers;

  public CustomerDetailPage(CustomersListViewModel customers)
  {
    InitializeComponent();

    this.customers = customers;
    BindingContext = new CustomerDetailViewModel();
    Title = "Add Customer";
  }

  public CustomerDetailPage(CustomersListViewModel customers,
    CustomerDetailViewModel customer)
  {
    InitializeComponent();

    this.customers = customers;
    BindingContext = customer;
    InsertButton.IsVisible = false;
  }
```

```
    async void InsertButton_Clicked(object sender, EventArgs e)
    {
      customers.Add((CustomerDetailViewModel)BindingContext);
      await Navigation.PopAsync(animated: true);
    }
  }
```

Обратите внимание на следующие моменты:

- конструктор по умолчанию устанавливает контекст привязки для нового экземпляра `customer`, а заголовок представления изменяется на `Add Customer`;
- конструктор с параметром `customer` устанавливает контекст привязки для этого экземпляра и скрывает кнопку Insert (Вставить), поскольку из-за двусторонней привязки данных она не нужна при редактировании существующего клиента;
- при нажатии кнопки Insert (Вставить) новый клиент добавляется к модели представления клиентов и навигация асинхронно возвращается к предыдущему виду.

Настройка главной страницы для мобильного приложения

Наконец, необходимо отредактировать код мобильного приложения, чтобы использовать список клиентов, обернутый в страницу навигации, в качестве главной страницы вместо удаленной, которая была создана с помощью шаблона проекта.

1. Откройте файл `App.xaml.cs`.
2. В конструкторе `App` измените оператор, создающий `MainPage`, чтобы вместо него создать экземпляр `CustomersListPage`, обернутый в экземпляр `NavigationPage`, как показано ниже:

```
public App()
{
  InitializeComponent();

  MainPage = new NavigationPage(new CustomersListPage());
}
```

Тестирование мобильного приложения в среде iOS

Протестируем мобильное приложение с помощью эмулятора устройства Android.

1. В программе Visual Studio справа от кнопки Run (Запуск) на панели инструментов установите целевой параметр Framework на значение net6.0-android и выберите эмулятор Android.

2. Начните с отладки проекта. Проект будет собран, а через несколько мгновений появится эмулятор устройства Android с запущенным приложением .NET MAUI (рис. 19.7).

Рис. 19.7. Эмулятор устройства Android, на котором запущено приложение Northwind Customers .NET MAUI

3. Щелкните кнопкой мыши на Seven Seas Imports и измените Company Name (Название компании) на Seven Oceans Imports (рис. 19.8).

Рис. 19.8. Редактирование названия компании на странице сведений о клиенте

4. Нажмите кнопку возврата, чтобы вернуться к списку клиентов, и обратите внимание, что название компании было обновлено из-за двусторонней привязки данных.
5. Щелкните на ссылке Add (Добавить) и заполните поля ввода для добавления записи о новом клиенте.

По умолчанию в эмуляторе устройств Android при наборе текста на физической клавиатуре отображается виртуальная. Чтобы скрыть ее, щелкните на значке клавиатуры справа от квадратной программной кнопки Android, а затем установите флажок Show virtual keyboard (Показывать виртуальную клавиатуру).

6. На странице сведений о клиенте нажмите кнопку Insert Customer (Вставить клиента) и после возвращения к списку клиентов обратите внимание, что новый клиент был добавлен в нижнюю часть списка. (На момент написания этой книги при использовании .NET MAUI Preview 9 существует ошибка, из-за которой представление списка не обновляется должным образом. Нажмите, удерживайте и перетащите вниз представление списка, а затем отпустите, чтобы обновить его.)
7. Нажмите и удерживайте нажатие на одном из клиентов, чтобы открыть кнопки действий Phone (Позвонить) и Delete (Удалить), как показано на рис. 19.9.

Рис. 19.9. Дополнительные команды для выбранного клиента

8. Щелкните на кнопке Phone (Позвонить) и обратите внимание на появившееся уведомление с подтверждением телефонного вызова клиента (кнопки Yes (Да) и No (Нет)).
9. Нажмите кнопку No (Нет).
10. Нажмите и удерживайте нажатие на одном из клиентов, чтобы открыть кнопки действий Phone (Позвонить) и Delete (Удалить). Щелкните кнопкой мыши на ссылке Delete (Удалить) и обратите внимание, что сведения о клиенте были удалены.
11. Нажмите, удерживайте и перетащите список вниз, а затем отпустите и обратите внимание на эффект анимации для обновления списка. Помните, что данная функция симулируется, поэтому список не изменяется.
12. Закройте эмулятор устройства Android.

Теперь мы заставим приложение вызвать сервис `Northwind.WebApi` для получения списка клиентов.

Взаимодействие мобильных приложений с веб-сервисами

Функция платформ Apple *App Transport Security (ATS)* заставляет разработчиков использовать передовой опыт, включая безопасные соединения между приложением и веб-сервисом. Она включена по умолчанию, и при небезопасном подключении мобильные приложения будут выдавать ошибку.

Вызвать веб-сервис, который защищен самоподписанным сертификатом, таким как наш сервис `Northwind.WebApi`, возможно, однако не слишком легко. Для простоты мы разрешим небезопасные подключения к веб-сервису и отключим проверки безопасности в мобильном приложении.

Разрешение небезопасных запросов в веб-сервисе

В первую очередь разрешим веб-сервису обрабатывать небезопасные соединения по новому URL.

1. В проекте `Northwind.WebApi` в файле `Program.cs` в разделе, который настраивает HTTP-конвейер, закомментируйте перенаправление HTTPS:

```
// закомментировано для проекта приложения .NET MAUI
// app.UseHttpsRedirection();
```

2. В файле `Program.cs` в методе `UseUrls` добавьте небезопасный URL, как показано ниже (выделено жирным шрифтом):

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseUrls(
    "https://localhost:5002"
    , "http://localhost:5008" // для клиента .NET MAUI
);
```

3. Запустите проект веб-сервиса `Northwind.WebApi` без отладки.
4. Запустите браузер Google Chrome и убедитесь, что веб-сервис возвращает клиентов в формате JSON, перейдя по адресу http://localhost:5008/api/customers/.
5. Закройте браузер Google Chrome, но оставьте веб-сервис работать.

Разрешение небезопасных подключений в приложении для iOS

Настроим проект `Northwind.Maui.Customers`, чтобы отключить функцию ATS и разрешить незащищенные HTTP-запросы к веб-сервису.

1. В проекте `Northwind.Maui.Customers` в папке Platforms/iOS откройте файл `Info.plist`, щелкнув правой кнопкой мыши, и откройте его с помощью XML (Text) Editor.
2. В конце словаря добавьте новый ключ NSAppTransportSecurity, который является словарем, и добавьте в него ключ NSAllowsArbitraryLoads со значением `true`, как показано ниже:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
          "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>LSRequiresIPhoneOS</key>
  <true/>
  ...
  <key>NSAppTransportSecurity</key>
  <dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
  </dict>
</dict>
</plist>
```

3. Сохраните и закройте файл `Info.plist`.

Разрешение небезопасных подключений в приложении для Android

Аналогично для Apple и ATS в Android 9 (уровень API 28) поддержка открытого текста (то есть без HTTPS) по умолчанию отключена.

Настроим проект, чтобы включить функцию открытого текста и разрешить незащищенные HTTP-запросы к веб-сервису.

1. В папке Platforms/Android, находящейся в папке Properties, откройте файл `MainApplication.cs`.
2. В атрибуте `Application` включите открытый текст, как показано ниже:

```
namespace Northwind.Maui.Customers
{
  [Application(UsesCleartextTraffic = true)]
  public class MainApplication : MauiApplication
```

Получение данных о клиентах с помощью сервиса

Теперь мы можем изменить страницу со списком клиентов, чтобы получать ее список из веб-сервиса вместо того, чтобы использовать образцы данных.

1. В проекте `Northwind.Maui.Customers` откройте файл `CustomersListPage.xaml.cs`.
2. Импортируйте следующие дополнительные пространства имен:

```
using System.Collections.Generic; // IEnumerable<T>
using System.Linq; // OrderBy
using System.NET.Http; // HttpClient
using System.NET.Http.Headers; // MediaTypeWithQualityHeaderValue
using System.NET.Http.Json; // ReadFromJsonAsync<T>
```

3. Измените код конструктора `CustomersListPage` так, чтобы загружать список клиентов через прокси сервиса и вызывать метод `AddSampleData` только при возникновении исключения:

```
public CustomersListPage()
{
  InitializeComponent();
  CustomersListViewModel viewModel = new();

  try
  {
    HttpClient client = new()
    {
      BaseAddress = new Uri("http://localhost:5008/")
    };

    client.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue("application/json"));

    HttpResponseMessage response = client
      .GetAsync("api/customers").Result;

    response.EnsureSuccessStatusCode();

    IEnumerable<CustomerDetailViewModel> customersFromService =
      response.Content.ReadFromJsonAsync
      <IEnumerable<CustomerDetailViewModel>>().Result;

    foreach (CustomerDetailViewModel c in customersFromService
      .OrderBy(customer => customer.CompanyName))
    {
      viewModel.Add(c);
    }
  }
```

```
    catch (Exception ex)
    {
      DisplayAlert(title: "Exception",
        message: $"App will use sample data due to: {ex.Message}",
        cancel: "OK");

      viewModel.AddSampleData();
    }

    BindingContext = viewModel;
  }
```

4. Выберите команду меню Build ▸ Clean Northwind.Maui.Customers (Сборка ▸ Очистить Northwind.Maui.Customers), поскольку изменения в файле `Info.plist`, такие как разрешение небезопасных соединений, иногда требуют чистой сборки.
5. Выберите команду меню Build ▸ Build Northwind.Maui.Customers (Сборка ▸ Собрать Northwind.Maui.Customers).
6. Запустите проект `Northwind.Maui.Customers` на эмуляторе Android и обратите внимание, что сведения о 91 клиенте загружены с помощью веб-сервиса.
7. Закройте эмулятор устройства Android.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 19.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Каковы четыре категории компонентов пользовательского интерфейса .NET MAUI и что они собой представляют?
2. Каковы четыре типа ячеек?
3. Каким образом вы можете разрешить пользователю выполнять действия с ячейками в `ListView`?
4. В каком случае элемент `Entry` используется вместо элемента `Editor`?
5. В чем заключается эффект установки `IsDestructive` в значение `true` для пункта меню в контекстных действиях ячейки?
6. Когда выполняется вызов методов `PushAsync` и `PopAsync` в приложении .NET MAUI?

7. В чем разница между `Margin` и `Padding` для такого элемента, как `Button`?
8. Как обработчики событий прикрепляются к объекту с помощью XAML?
9. Что делают стили XAML?
10. Где можно определить ресурсы?

Упражнение 19.2. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-19---building-mobile-and-desktop-apps-using-net-maui, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали, как создать кросс-платформенное мобильное и настольное приложение с помощью .NET MAUI, которое использует данные из веб-сервиса.

В следующей главе вы научитесь защищать данные и файлы с помощью хеширования, шифрования цифровой подписи, аутентификации и авторизации.

20 Защита данных и приложений

Эта глава посвящена тому, как с помощью шифрования защищать данные от просмотра злоумышленниками, а также тому, как применять хеширование и цифровые подписи для защиты данных от вмешательства или повреждения.

В .NET Core 2.1 компания Microsoft представила криптографические API, основанные на `Span<T>`, для хеширования, генерации случайных чисел, генерации и обработки асимметричной подписи и шифрования *RSA* (Rivest — Shamir — Adleman, Ривест — Шамир — Адлеман).

Криптографические операции реализуются операционной системой, поэтому при исправлении уязвимости системы безопасности в операционной системе приложения .NET немедленно получают преимущество. Однако это означает, что эти .NET-приложения могут использовать только те функции, которые поддерживает ОС. О том, какие функции и какой операционной системой поддерживаются, вы можете прочитать на сайте https://docs.microsoft.com/en-us/dotnet/standard/security/cross-platform-cryptography.

В этой главе:

- терминология безопасности;
- шифрование и дешифрование данных;
- хеширование данных;
- подписывание данных;
- генерация случайных чисел;
- аутентификация и авторизация пользователей.

Внимание! Код в этой главе соответствует примитивным сценариям безопасности и приводится только для базовых образовательных целей. Вы не должны использовать ее код для разработки настоящих библиотек и приложений. Работайте только с профессионально написанными библиотеками, которые были созданы с использованием принципов безопасности и усилены для реального применения в соответствии с актуальными рекомендациями по обеспечению безопасности.

Терминология безопасности

Существует множество способов защиты данных; несколько самых популярных из них перечислены ниже. В этой главе вы ознакомитесь с их подробным описанием и практическими реализациями.

- *Шифрование и дешифрование* — двусторонний процесс преобразования читаемого текста в шифрованный и обратно.
- *Хеши* — односторонний процесс генерации хеш-значения для безопасного хранения паролей или обнаружения вредоносных изменений либо повреждений данных. Простые хеши не дожны использоваться для паролей. Вы должны применять PBKDF2, bcrypt или scrypt, поскольку они гарантируют, что не может быть двух входных данных, которые генерируют одинаковый хеш.
- *Цифровые подписи* — метод проверки источника поступивших данных, то есть что данные были получены из заявленного источника путем верификации цифровой подписи, которая была применена к некоторым данным, с помощью открытого ключа.
- *Аутентификация* — метод идентификации пользователя путем проверки его учетных данных.
- *Авторизация* — метод выдачи 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 простые 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 представлены байтовыми массивами различного размера. Ключи используются для таких целей, как:

- шифрование и дешифрование: AES, 3DES, RC2, Rijndael, RSA;
- подписание и проверка: RSA, ECDSA, DSA;
- аутентификация и проверка подлинности сообщений: HMAC;
- согласование ключей: Diffie-Hellman, Elliptical Curve Diffie-Hellman.

Выберите больший размер ключа для более надежной защиты. Это чрезмерное упрощение, поскольку некоторые реализации RSA поддерживают до 16 384-битных ключей, на создание которых может потребоваться несколько дней, и в большинстве сценариев это будет лишним; 2048-битного ключа должно быть достаточно до 2030 года, после чего вам следует перейти на 3192-битные ключи.

Ключи для шифрования и дешифрования могут быть *симметричными* (также известны как *общие* или *секретные*, поскольку один и тот же ключ используется для шифрования и дешифрования и поэтому должен храниться в безопасности) и *асимметричными* (пара из открытого и закрытого ключей, в которой для шифрования используется открытый, а для дешифрования — только закрытый).

Алгоритмы шифрования с помощью симметричных ключей работают быстро и могут шифровать большие объемы потоковых данных. Алгоритмы шифрования с помощью асимметричных ключей функционируют медленно и могут шифровать только небольшие массивы байтов. Чаще всего асимметричные ключи используются для создания и проверки подписи.

В своих проектах применяйте оба способа, используя симметричное шифрование для защиты самих данных и асимметричное — для распространения симметричного ключа. По такому принципу работал криптографический протокол *Secure Sockets Layer (SSL)* 2.0 в Интернете в 1995 году. Сегодня то, что все еще часто называют SSL, на самом деле является механизмом *безопасности транспортного уровня* (Transport Layer Security, TLS), который использует согласование ключей, а не зашифрованные RSA сеансовые ключи.

Ключи в шифровании представлены массивами байтов разного размера.

Векторы инициализации и размеры блоков

Вполне вероятно, что при шифровании больших объемов данных повторяются некоторые из 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
```

Мы можем избежать повторения последовательностей, разделив данные на *блоки*. После шифрования блока из него генерируется значение массива байтов, которое

передается в следующий блок в целях настройки алгоритма. Следующий блок шифруется таким образом, что выходные данные отличаются даже для того же входа, что и в предыдущем блоке. Зашифровать первый блок можно при наличии массива байтов для выполнения задачи. Это так называемый *вектор инициализации* (initialization vector, IV).

Вектор инициализации должен:

- генерироваться случайным образом вместе с каждым зашифрованным сообщением;
- передаваться вместе с зашифрованным сообщением;
- сам по себе не быть секретом.

Соли

Соль представляет собой случайный массив байтов, который используется как дополнительный ввод для односторонней хеш-функции. Если вы не применяете соль при генерации хешей, то при условии, что многие из ваших пользователей регистрируются, указывая `123456` в качестве пароля (по данным на 2016 год, примерно 8 % пользователей так и делают!), все они имеют одно и то же хеш-значение и их учетная запись будет уязвима для внешнего доступа через подбор пароля по словарю.

Когда пользователь регистрируется, соль должна генерироваться случайным образом и конкатенироваться с указанным пользователем паролем до того, как будет хеширована. Результат этой операции (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.

Даже «соления» паролей недостаточно для действительно безопасного хранения. Вы должны поработать гораздо больше, например сделать PBKDF2, bcrypt или scrypt, но такая работа выходит за рамки этой книги.

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, однако надежный обмен байтовыми массивами может быть затруднен.

Вы можете надежно генерировать ключи и векторы инициализации, используя стандарт *формирования ключа на основе пароля* (password-based key derivation function, PBKDF2). Прекрасно подойдет класс `Rfc2898DeriveBytes`, который принимает пароль, соль, счетчик итераций и алгоритм хеширования (по умолчанию используется SHA-1, который больше не рекомендуется). Затем он генерирует ключи и векторы инициализации, вызывая метод `GetBytes`. Количество итераций — это то, сколько раз пароль хешируется в процессе работы. Чем больше итераций, тем сложнее взломать пароль.

Хотя класс `Rfc2898DeriveBytes` можно использовать для генерации IV и ключа, IV должен генерироваться каждый раз случайным образом и передаваться вместе с зашифрованным сообщением в виде открытого текста, поскольку оно не обязательно должно быть секретным.

Размер соли должен составлять 8 байт или больше, а количество итераций должно быть таким, чтобы для генерации ключа и IV для алгоритма шифрования на целевой машине требовалось около 100 мс. Это значение будет увеличиваться с течением времени по мере совершенствования процессоров. В приведенном ниже примере кода мы используем 150 000, но это значение уже будет слишком низким для некоторых компьютеров к тому времени, когда вы будете это читать.

Шифрование и дешифрование данных

На платформе .NET доступно несколько алгоритмов шифрования.

В устаревшей .NET Framework некоторые алгоритмы реализуются операционной системой, и их имена имеют суффикс `CryptoServiceProvider` или `Cng`. Есть алгоритмы, реализованные в .NET BCL, и их имена имеют суффикс `Managed`.

В современном .NET все алгоритмы реализуются операционной системой. Если алгоритмы ОС сертифицированы федеральными стандартами обработки информации *(Federal Information Processing Standards, FIPS)*, то .NET будет использовать алгоритмы, сертифицированные FIPS.

Как правило, для получения экземпляра алгоритма вы всегда будете использовать абстрактный класс, такой как `Aes` и его метод `Create`, поэтому вам не нужно знать, применяете ли вы `CryptoServiceProvider` или `Managed`.

Одни алгоритмы используют симметричные ключи, а другие — асимметричные. Главный асимметричный алгоритм шифрования — RSA. Рон Ривест, Ади Шамир и Леонард Адлеман описали этот алгоритм в 1977 году. Аналогичный алгоритм был разработан в 1973 году Клиффордом Коксом, английским математиком, работавшим в Центре правительственной связи Британского разведывательного

управления, но был рассекречен только в 1997 году, поэтому Ривест, Шамир и Адлеман получили признание и увековечили свои имена в аббревиатуре RSA.

Алгоритмы симметричного шифрования используют `CryptoStream` для эффективного шифрования или дешифрования большого количества байтов. Асимметричные алгоритмы могут обрабатывать только небольшое количество байтов, хранящихся в байтовом массиве вместо потока.

Ниже перечислены наиболее распространенные алгоритмы симметричного шифрования. Они наследуются от абстрактного класса `SymmetricAlgorithm`:

- AES;
- DESCryptoServiceProvider;
- TripleDES;
- RC2CryptoServiceProvider;
- RijndaelManaged.

Если вам нужно написать код для расшифровки неких данных, отправленных внешней системой, то вам придется прибегнуть к тому алгоритму, который внешняя система использовала для шифрования данных. Если вам нужно отправить зашифрованные данные в систему, которая может дешифровать только с помощью специального алгоритма, то вы снова не сможете выбрать алгоритм.

Если же ваш код выполняет и шифрование, и дешифрование, то можете выбрать алгоритм, который наилучшим образом соответствует вашим требованиям к надежности, производительности и т. д.

Выберите симметричный алгоритм блочного шифрования (Advanced Encryption Standard, AES), основанный на алгоритме Rijndael, для симметричного шифрования. Выберите RSA (алгоритм шифрования с открытым ключом) для асимметричного шифрования. Не путайте RSA с DSA. Алгоритм цифровой подписи (Digital Signature Algorithm, DSA) не может зашифровать данные. Он может генерировать только хеши и подписи.

Симметричное шифрование с помощью алгоритма AES

Чтобы упростить повторное использование обеспечивающего безопасность кода в нескольких проектах, мы создадим статический класс `Protector` в своей библиотеке классов и затем будем ссылаться на него в консольном приложении.

Приступим!

1. Откройте редактор кода и создайте рабочую область/решение `Chapter20`.
2. Создайте проект консольного приложения с такими настройками:

   1) шаблон проекта: Console Application/`console`;
   2) файл и папка рабочей области/решения: `Chapter20`;
   3) файл и папка проекта: `EncryptionApp`.
3. Создайте библиотеку классов `CryptographyLib` в рабочей области/решении `Chapter20`:
   1) в программе Visual Studio настройте стартовый проект для решения в соответствии с текущим выбором;
   2) в программе Visual Studio Code выберите `EncryptionApp` в качестве активного проекта OmniSharp.
4. В проекте `CryptographyLib` переименуйте файл `Class1.cs` в `Protector.cs`.
5. В проект `EncryptionApp` добавьте проектную ссылку на библиотеку `CryptographyLib`, как показано ниже:

```
<ItemGroup>
  <ProjectReference
    Include="..\CryptographyLib\CryptographyLib.csproj" />
</ItemGroup>
```

6. Соберите проект `EncryptionApp` и убедитесь в отсутствии ошибок компиляции.
7. Откройте файл `Protector.cs` и измените его содержимое так, чтобы определить статический класс `Protector` с полями для хранения массива байтов соли и большого количества итераций, а также методами `Encrypt` и `Decrypt`:

```
using System.Diagnostics;
using System.Security.Cryptography;
using System.Security.Principal;
using System.Text;
using System.Xml.Linq;

using static System.Console;
using static System.Convert;

namespace Packt.Shared
{
  public static class Protector
  {
    // размер соли должен быть не менее 8 байт, мы будем использовать 16 байт
    private static readonly byte[] salt =
      Encoding.Unicode.GetBytes("7BANANAS");

    // количество итераций должно быть достаточно высоким, чтобы
    // на генерацию ключа и IV на целевой машине уходило не менее 100 мс.
    // 150 000 итераций занимают 139 мс на моем процессоре Intel
    // Core i7-1165G7 11-го поколения с тактовой частотой 2,80 ГГерц.
    private static readonly int iterations = 150_000;
```

```
public static string Encrypt(
  string plainText, string password)
{
  byte[] encryptedBytes;
  byte[] plainBytes = Encoding.Unicode.GetBytes(plainText);

  using (Aes aes = Aes.Create()) // фабричный метод абстрактных классов
  {
    // пишем, сколько времени требуется для генерации ключа и IV
    Stopwatch timer = Stopwatch.StartNew();

    using (Rfc2898DeriveBytes pbkdf2 = new(
      password, salt, iterations))
    {
      aes.Key = pbkdf2.GetBytes(32); // устанавливаем 256-битный ключ
      aes.IV = pbkdf2.GetBytes(16); // устанавливаем 128-битный IV
    }

    timer.Stop();

    WriteLine("{0:N0} milliseconds to generate Key and IV using {1:N0}
    iterations.",
      arg0: timer.ElapsedMilliseconds,
      arg1: iterations);

    using (MemoryStream ms = new())
    {
      using (ICryptoTransform transformer = aes.CreateEncryptor())
      {
        using (CryptoStream cs = new(
          ms, transformer, CryptoStreamMode.Write))
        {
          cs.Write(plainBytes, 0, plainBytes.Length);
        }
      }
      encryptedBytes = ms.ToArray();
    }
  }

  return ToBase64String(encryptedBytes);
}

public static string Decrypt(
  string cipherText, string password)
{
  byte[] plainBytes;
  byte[] cryptoBytes = FromBase64String(cipherText);

  using (Aes aes = Aes.Create())
  {
    using (Rfc2898DeriveBytes pbkdf2 = new(
```

```
            password, salt, iterations))
          {
            aes.Key = pbkdf2.GetBytes(32);
            aes.IV = pbkdf2.GetBytes(16);
          }

          using (MemoryStream ms = new())
          {
            using (ICryptoTransform transformer = aes.CreateDecryptor())
            {
              using (CryptoStream cs = new(
                ms, transformer, CryptoStreamMode.Write))
              {
                cs.Write(cryptoBytes, 0, cryptoBytes.Length);
              }
            }
            plainBytes = ms.ToArray();
          }
        }

        return Encoding.Unicode.GetString(plainBytes);
      }
    }
  }
```

Обратите внимание на следующие моменты:

- хотя размер соли и количество итераций могут быть жестко закодированными (но их предпочтительнее хранить в самом сообщении), пароль *должен* передаваться в качестве параметра 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 зашифрованные массивы байтов переводятся в кодировку Base64 и из нее, чтобы упростить их чтение для человека.

Никогда не кодируйте жестко пароль в исходном коде, поскольку даже после компиляции пароль в сборке можно прочитать с помощью инструментов дизассемблера.

8. В проекте `EncryptionApp` откройте файл `Program.cs`, а затем импортируйте пространство имен для класса `Protector`, пространство имен для класса `CryptographicException` и статически импортируйте класс `Console`:

```
using System.Security.Cryptography; // CryptographicException
using Packt.Shared; // Protector

using static System.Console;
```

9. В файле `Program.cs` добавьте следующие операторы, 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();

if ((password is null) || (message is null))
{
  WriteLine("Message or password cannot be null.");
  return;
}

string cipherText = Protector.Encrypt(message, password);

WriteLine($"Encrypted text: {cipherText}");

Write("Enter the password: ");
string? password2Decrypt = ReadLine();

if (password2Decrypt is null)
{
  WriteLine("Password to decrypt cannot be null.");
  return;
}

try
{
  string clearText = Protector.Decrypt(cipherText, password2Decrypt);
  WriteLine($"Decrypted text: {clearText}");
}
catch (CryptographicException ex)
{
  WriteLine("{0}\nMore details: {1}",
    arg0: "You entered the wrong password!",
    arg1: ex.Message);
}
catch (Exception ex)
{
  WriteLine("Non-cryptographic exception: {0}, {1}",
    arg0: ex.GetType().Name,
    arg1: ex.Message);
}
```

10. Запустите код, попробуйте ввести сообщение и пароль для шифрования, введите тот же пароль для расшифровки и проанализируйте результат:

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
139 milliseconds to generate Key and IV using 150,000 iterations.
Encrypted text: eWt8sgL7aSt5DC9g74ONEPO7mjd55lXB/MmCZpUsFE0=
Enter the password: secret
Decrypted text: Hello Bob
```

Если ваш вывод показывает количество миллисекунд меньше 100, то увеличивайте количество итераций до тех пор, пока количество миллисекунд не станет 100 или больше. Обратите внимание, что другое количество итераций повлияет на хешированное значение, поэтому оно будет выглядеть иначе, чем показано выше.

11. Перезапустите код и вновь введите сообщение и пароль для шифрования, только на этот раз указав в пароле для расшифровки намеренную ошибку. Проанализируйте результат:

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
134 milliseconds to generate Key and IV using 150,000 iterations.
Encrypted text: eWt8sgL7aSt5DC9g74ONEPO7mjd55lXB/MmCZpUsFE0=
Enter the password: 123456
You entered the wrong password!
More details: Padding is invalid and cannot be removed.
```

Для поддержки будущих обновлений шифрования записывайте информацию о том, что вы выбрали, например AES-256, режим CBC с подстановкой PKCS#7, PBKDF2 и его алгоритм хеширования и количество итераций. Это известно как криптографическая гибкость.

Хеширование данных

На платформе .NET доступно несколько алгоритмов хеширования. Одни не требуют использования каких-либо ключей, вторым необходимы симметричные ключи, а третьим — асимметричные.

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. 20.1 представлены некоторые универсальные алгоритмы хеширования без ключа.

Таблица 20.1. Типичные алгоритмы хеширования без ключа

Алгоритм	Размер хеша	Описание
MD5	16 байт	Используется чаще всего, поскольку быстр в работе, но неустойчив к коллизиям
SHA1	20 байт	Применение алгоритма SHA1 в Интернете не рекомендуется с 2011 года
SHA256 SHA384 SHA512	32 байта 48 байт 64 байта	Алгоритмы из семейства «Безопасный алгоритм хеширования 2-го 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 бóльшим размером хеша, чтобы уменьшить вероятность повторения хешей. Первая известная коллизия алгоритма MD5 произошла в 2010 году, а первая известная коллизия алгоритма SHA1 — в 2017-м. Более подробно можно прочитать на сайте https://arstechnica.co.uk/information-technology/2017/02/at-deaths-door-for-years-widely-used-sha1-function-is-now-dead/.

Хеширование с помощью алгоритма SHA256

Добавим класс для представления пользователя, хранящегося в памяти, файле или базе данных. Мы будем использовать словарь для хранения нескольких пользователей в памяти.

1. В проекте библиотеки классов `CryptographyLib` добавьте файл класса `User.cs` и задайте ему три свойства для хранения имени пользователя, случайного значения соли, а также его соленого и хешированного пароля:

```
namespace Packt.Shared;

public class User
{
  public string Name { get; set; }
  public string Salt { get; set; }
  public string SaltedHashedPassword { get; set; }

  public User(string name, string salt,
    string saltedHashedPassword)
  {
```

```
    Name = name;
    Salt = salt;
    SaltedHashedPassword = saltedHashedPassword;
  }
}
```

2. В класс `Protector` добавьте следующие операторы, чтобы объявить словарь для хранения в памяти информации о нескольких пользователях. Применяются два метода: для регистрации нового пользователя и для проверки введенного пароля при последующей авторизации:

```
private static Dictionary<string, User> Users = new();

public static User Register(
  string username, string password)
{
  // генерируем случайную соль
  RandomNumberGenerator rng = RandomNumberGenerator.Create();
  byte[] saltBytes = new byte[16];
  rng.GetBytes(saltBytes);
  string saltText = ToBase64String(saltBytes);

  // генерируем соленый и хешированный пароль
  string saltedhashedPassword = SaltAndHashPassword(password, saltText);

  User user = new(username, saltText, saltedhashedPassword);

  Users.Add(user.Name, user);

  return user;
}

// проверяем пароль пользователя, который хранится
// в приватном статическом словаре Users
public static bool CheckPassword(string username, string password)
{
  if (!Users.ContainsKey(username))
  {
    return false;
  }

  User u = Users[username];

  return CheckPassword(password,
    u.Salt, u.SaltedHashedPassword);
}

// проверяем пароль пользователя, используя соль и хешированный пароль
public static bool CheckPassword(string password,
  string salt, string hashedPassword)
{
```

```
    // повторно генерируем соленый и хешированный пароль
    string saltedhashedPassword = SaltAndHashPassword(
      password, salt);

    return (saltedhashedPassword == hashedPassword);
  }

  private static string SaltAndHashPassword(string password, string salt)
  {
    using (SHA256 sha = SHA256.Create())
    {
      string saltedPassword = password + salt;
      return ToBase64String(sha.ComputeHash(
        Encoding.Unicode.GetBytes(saltedPassword)));
    }
  }
```

3. Откройте редактор кода и создайте консольное приложение `HashingApp` в рабочей области/решении `Chapter20`.
4. В программе Visual Studio Code выберите `HashingApp` в качестве активного проекта OmniSharp.
5. В проекте `HashingApp` добавьте ссылку на проект `CryptographyLib`.
6. Соберите проект `HashingApp` и убедитесь в отсутствии ошибок компиляции.
7. В файле `Program.cs` импортируйте пространство имен `Packt.Shared`.
8. В файле `Program.cs` добавьте операторы для регистрации пользователя и запроса регистрации другого пользователя, а также запроса авторизации под одним из логинов и проверки пароля:

```
WriteLine("Registering Alice with Pa$$w0rd:");
User alice = Protector.Register("Alice", "Pa$$w0rd");

WriteLine($"  Name: {alice.Name}");
WriteLine($"  Salt: {alice.Salt}");
WriteLine("  Password (salted and hashed): {0}",
  arg0: alice.SaltedHashedPassword);
WriteLine();

Write("Enter a new user to register: ");
string? username = ReadLine();

Write($"Enter a password for {username}: ");
string? password = ReadLine();

if ((username is null) || (password is null))
{
  WriteLine("Username or password cannot be null.");
  return;
}
```

```
WriteLine("Registering a new user:");
User newUser = Protector.Register(username, password);
WriteLine($" Name: {newUser.Name}");
WriteLine($" Salt: {newUser.Salt}");
WriteLine(" Password (salted and hashed): {0}",
  arg0: newUser.SaltedHashedPassword);
WriteLine();

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();

  if ((loginUsername is null) || (loginPassword is null))
  {
    WriteLine("Login username or password cannot be null.");
    return;
  }

  correctPassword = Protector.CheckPassword(
    loginUsername, loginPassword);

  if (correctPassword)
  {
    WriteLine($"Correct! {loginUsername} has been logged in.");
  }
  else
  {
    WriteLine("Invalid username or password. Try again.");
  }
}
```

9. Запустите код, зарегистрируйте нового пользователя с тем же паролем, что и у `Alice` (Алисы), и проанализируйте результат:

```
Registering Alice with Pa$$w0rd:
  Name: Alice
  Salt: I1I1dzIjkd7EYDf/6jaf4w==
  Password (salted and hashed): pIoadjE4W/XaRFkqS3br3UuAuPv/3LVQ8kzj6mvcz+s=

Enter a new user to register: Bob
Enter a password for Bob: Pa$$w0rd
Registering a new user:
  Name: Bob
  Salt: 1X7ym/UjxTiuEWBC/vIHpw==
  Password (salted and hashed):
```

```
DoBFtDhKeN0aaaLVdErtrZ3mpZSvpWDQ9TXDosTq0sQ=

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: 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 этих пользователей будут различаться.

Подписывание данных

В качестве доказательства, что полученные данные на самом деле присланы доверенным отправителем, используются цифровые подписи. Вы подписываете не сами данные, а только их хеш, поскольку все алгоритмы подписи сначала хешируют данные в качестве шага реализации. Они также позволяют сократить этот шаг и предоставить уже хешированные данные.

Для генерации и подписывания хеша мы воспользуемся сочетанием алгоритмов RSA и SHA256.

Мы могли бы применить алгоритм DSA как для хеширования, так и для подписывания. DSA генерирует подпись быстрее, чем RSA, но медленнее проверяет ее. Поскольку подпись генерируется один раз, но проверяется многократно, лучше проводить быстрее проверку, а не генерацию.

Сегодня DSA используется редко. Улучшенным эквивалентом является эллиптическая кривая DSA (Elliptic Curve DSA). Хотя ECDSA работает медленнее, чем RSA, она генерирует более короткую подпись с тем же уровнем безопасности.

Подписывание с помощью алгоритмов SHA256 и RSA

Рассмотрим подписывание данных и проверку подписи с помощью алгоритма шифрования с открытым ключом.

1. В классе `Protector` добавьте следующие операторы, чтобы объявить поле для хранения открытого ключа в виде значения `string`, а также два метода для генерации и проверки подписи:

```
public static string? PublicKey;

public static string GenerateSignature(string data)
{
  byte[] dataBytes = Encoding.Unicode.GetBytes(data);
  SHA256 sha = SHA256.Create();
  byte[] hashedData = sha.ComputeHash(dataBytes);
  RSA rsa = RSA.Create();

  PublicKey = rsa.ToXmlString(false); // exclude private key

  return ToBase64String(rsa.SignHash(hashedData,
    HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1));
}

public static bool ValidateSignature(
  string data, string signature)
{
  if (PublicKey is null) return false;
  byte[] dataBytes = Encoding.Unicode.GetBytes(data);
  SHA256 sha = SHA256.Create();
  byte[] hashedData = sha.ComputeHash(dataBytes);
  byte[] signatureBytes = FromBase64String(signature);
  RSA rsa = RSA.Create();
  rsa.FromXmlString(PublicKey);
  return rsa.VerifyHash(hashedData, signatureBytes,
    HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
}
```

Обратите внимание на следующие моменты:

- перед использованием кода проверки подписи должен быть доступен только открытый ключ из пары «открытый — закрытый ключ», чтобы при вызове метода `ToXmlString` мы могли передать значение `false`. Закрытый ключ требуется для подписи данных и должен храниться в секрете, поскольку любой, кто имеет доступ к этому ключу, может подписывать данные от вашего имени!
- алгоритм хеширования, применяемый для генерации хеша из данных с помощью вызова метода `SignHash`, должен соответствовать алгоритму хеширования, используемому при вызове метода `VerifyHash`. В вышеприведенном примере мы задействовали алгоритм `SHA256`.

Теперь мы можем протестировать подписывание каких-нибудь данных и проверку подписи.

2. Откройте редактор кода и создайте консольное приложение `SigningApp` в рабочей области/решении `Chapter20`.
3. В программе Visual Studio Code выберите `SigningApp` в качестве активного проекта OmniSharp.
4. В проекте `SigningApp` добавьте ссылку на проект `CryptographyLib`.

5. Соберите проект `SigningApp` и убедитесь в отсутствии ошибок компиляции.
6. В файле `Program.cs` импортируйте пространство имен `Packt.Shared`.
7. Добавьте следующие операторы, чтобы предложить пользователю ввести некий текст, подписать его, проверить подпись, а затем изменить ее и снова проверить, чтобы намеренно вызвать несоответствие:

```
Write("Enter some text to sign: ");
string? data = ReadLine();

string 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.");
}

// имитируем поддельную подпись, заменив первый символ на X или Y
string fakeSignature = signature.Replace(signature[0], 'X');
if (fakeSignature == signature)
{
  fakeSignature = signature.Replace(signature[0], 'Y');
}

if (Protector.ValidateSignature(data, fakeSignature))
{
  WriteLine("Correct! Signature is valid.");
}
else
{
  WriteLine($"Invalid signature: {fakeSignature}");
}
```

8. Запустите код и введите любой текст, как показано в следующем выводе (отредактированном по длине):

```
Enter some text to sign: The cat sat on the mat.
Signature: BXSTdM...4Wrg==
Public key used to check signature:
<RSAKeyValue><Modulus>nHtwl3...mw3w==</Modulus><Exponent>AQAB</Exponent></
RSAKeyValue>
Correct! Signature is valid.
Invalid signature: XXSTdM...4Wrg==
```

Генерация случайных чисел

Иногда необходимо сгенерировать случайные числа, возможно при создании игры, имитирующей бросок игральных костей, либо для дальнейшего использования с криптографией при шифровании или подписывании. В .NET есть несколько классов, генерирующих случайные числа.

Генерация случайных чисел для игр и подобных приложений

В сценариях, не требующих истинно случайных чисел, например в играх, вы можете создать экземпляр класса `Random`, как показано в примере ниже:

```
Random r = new();
```

В конструкторе класса `Random` содержится параметр для указания начального значения, используемого для инициализации генератора псевдослучайных чисел:

```
Random r = new(Seed: 46378);
```

Как вы помните из главы 2, имена параметров необходимо задавать с помощью так называемого верблюжьего регистра. Разработчик, определивший конструктор класса `Random`, нарушил это соглашение! Имя параметра следовало задать как `seed`, а не `Seed`.

Общие начальные значения работают как секретные ключи, так что если вы воспользуетесь одинаковым алгоритмом генерации случайных чисел с одинаковым начальным значением в двух приложениях, то они могут генерировать одинаковые последовательности «случайных» чисел. Иногда это необходимо, например при синхронизации GPS-приемника со спутником. Но обычно следует держать начальное значение в секрете.

Создав объект `Random`, вы можете вызывать его методы для генерации случайных чисел, как показано в следующих примерах кода:

```
// minValue — включающая нижняя граница, то есть 1 — возможное значение
// maxValue — исключающая верхняя граница, то есть 7 — невозможное значение
int dieRoll = r.Next(minValue: 1, maxValue: 7); // возвращает от 1 до 6

double randomReal = r.NextDouble(); // возвращает от 0,0 до менее 1,0

byte[] arrayOfBytes = new byte[256];
r.NextBytes(arrayOfBytes); // 256 случайных байт 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 превышающее максимальное значение. Аналогичным образом, значение, возвращаемое методом `NextDouble`, больше или равно `0.0` и меньше `1.0`.

Генерация случайных чисел для криптографии

Класс `Random` генерирует криптографически слабые *псевдослучайные* числа. Для криптографии его недостаточно! Если случайные числа — неслучайные, то они будут предсказуемыми и взломщик может сломать вашу защиту.

Для криптографически сильных псевдослучайных чисел необходимо использовать тип, наследующий от `RandomNumberGenerator`, например `RNGCryptoServiceProvider`.

Создадим метод для генерации истинно случайного байтового массива, который можно использовать в таких алгоритмах, как шифрование, для получения ключей и значений вектора инициализации.

1. В классе `Protector` добавьте следующие операторы, чтобы определить метод, генерирующий случайный ключ или вектор инициализации для использования в шифровании:

   ```
   public static byte[] GetRandomKeyOrIV(int size)
   {
     RandomNumberGenerator r = RandomNumberGenerator.Create();
     byte[] data = new byte[size];
     r.GetBytes(data);

     // data представляет собой массив, который теперь заполнен
     // криптографически стойкими случайными байтами
     return data;
   }
   ```

   Теперь мы можем протестировать случайные байты, сгенерированные для истинно случайного ключа шифрования или вектора инициализации.

2. Откройте редактор кода и создайте консольное приложение `RandomizingApp` в рабочей области/решении `Chapter20`.
3. В программе Visual Studio Code выберите `RandomizingApp` в качестве активного проекта для OmniSharp.
4. В проекте `RandomizingApp` добавьте ссылку на проект `CryptographyLib`.
5. Соберите проект `RandomizingApp` и убедитесь в отсутствии ошибок компиляции.
6. В файле `Program.cs` импортируйте пространство имен `Packt.Shared`.
7. Добавьте следующие операторы, чтобы предложить пользователю ввести размер байтового массива, а затем сгенерируйте случайные байтовые значения и запишите их в консоль:

```
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();
```

8. Запустите код, введите размерность ключа, например, 256 и проанализируйте случайно сгенерированный ключ:

```
How big do you want the key (in bytes): 256
Key as byte array:
f1 57 3f 44 80 e7 93 dc 8e 55 04 6c 76 6f 51 b9
e8 84 59 e5 8d eb 08 d5 e6 59 65 20 b1 56 fa 68
...
```

Аутентификация и авторизация пользователей

Аутентификация — это процесс проверки подлинности личности пользователя путем проверки его учетных данных через некоторый удостоверяющий центр. Учетные данные могут включать в себя имя пользователя и пароль, или отпечаток пальца, или скан лица. Проведя аутентификацию, удостоверяющий центр может выдавать утверждения о пользователе, например, каков его адрес электронной почты и к каким группам или ролям он принадлежит.

Авторизация — это процесс определения принадлежности группе или роли, проводимый перед выдачей доступа к таким ресурсам, как функции приложения или его данные. Хотя авторизация может основываться на индивидуальной идентификации, рекомендуется проводить ее на основе принадлежности роли или группе (что можно указать в утверждениях), даже если в группе или в роли имеется только один пользователь, поскольку это позволяет в дальнейшем менять информацию о принадлежности пользователя, не прибегая к повторному назначению индивидуальных прав доступа.

Например, вместо того, чтобы назначать права доступа к Букингемскому дворцу *Елизавете Александре Марии Виндзор* (пользователю), вы можете назначить права доступа *Монарху Соединенного Королевства Великобритании и Северной Ирландии и других королевств и территорий* (роль), а затем добавить Елизавету в качестве единственного участника этой роли. Тогда в какой-то момент в будущем вам не нужно будет изменять права доступа для роли *Монарх*; вы просто удалите Елизавету и добавите следующего человека в порядок престолонаследования. И конечно же, вы реализуете этот порядок в виде очереди.

Механизмы аутентификации и авторизации

Существует несколько механизмов аутентификации и авторизации. Все они реализуют пару интерфейсов `IIdentity` и `IPrincipal` в пространстве имен `System.Security.Principal`.

Идентификация пользователя

Интерфейс `IIdentity` представляет пользователя, поэтому имеет свойства `Name` и `IsAuthenticated`, которые указывают, является ли пользователь анонимным или успешно прошедшим аутентификацию по своим учетным данным:

```
namespace System.Security.Principal
{
  public interface IIdentity
  {
    string? AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string? Name { get; }
  }
}
```

Распространенным классом, реализующим этот интерфейс, является `GenericIdentity`, который наследуется от класса `ClaimsIdentity`:

```
namespace System.Security.Principal
{
  public class GenericIdentity : ClaimsIdentity
  {
    public GenericIdentity(string name);
    public GenericIdentity(string name, string type);
    protected GenericIdentity(GenericIdentity identity);
    public override string AuthenticationType { get; }
    public override IEnumerable<Claim> Claims { get; }
    public override bool IsAuthenticated { get; }
    public override string Name { get; }
    public override ClaimsIdentity Clone();
  }
}
```

Объекты `Claim` имеют свойство `Type`, указывающее, к чему относится утверждение: к имени пользователя, его принадлежности к роли или группе, к дате рождения и т. д.:

```
namespace System.Security.Claims
{
  public class Claim
  {
    // различные конструкторы
```

```
    public string Type { get; }
    public ClaimsIdentity? Subject { get; }
    public IDictionary<string, string> Properties { get; }
    public string OriginalIssuer { get; }
    public string Issuer { get; }
    public string ValueType { get; }
    public string Value { get; }
    protected virtual byte[]? CustomSerializationData { get; }
    public virtual Claim Clone();
    public virtual Claim Clone(ClaimsIdentity? identity);
    public override string ToString();
    public virtual void WriteTo(BinaryWriter writer);
    protected virtual void WriteTo(BinaryWriter writer, byte[]? userData);
  }

  public static class ClaimTypes
  {
    public const string Actor = "http://schemas.xmlsoap.org/ws/2009/09/
    identity/claims/actor";
    public const string NameIdentifier = "http://schemas.xmlsoap.org/
    ws/2005/05/identity/claims/nameidentifier";
    public const string Name = "http://schemas.xmlsoap.org/ws/2005/05/
    identity/claims/name";
    public const string PostalCode = "http://schemas.xmlsoap.org/ws/2005/05/
    identity/claims/postalcode";

    // ... многие другие строковые константы

    public const string MobilePhone = "http://schemas.xmlsoap.org/ws/2005/05/
    identity/claims/mobilephone";
    public const string Role = "http://schemas.microsoft.com/ws/2008/06/
    identity/claims/role";
    public const string Webpage = "http://schemas.xmlsoap.org/ws/2005/05/
    identity/claims/webpage";
  }
}
```

Принадлежность пользователя

Интерфейс `IPrincipal` служит для связи пользователя с ролями и группами, членом которых он является, поэтому `IPrincipal` можно использовать для целей авторизации:

```
namespace System.Security.Principal
{
  public interface IPrincipal
  {
    IIdentity? Identity { get; }
    bool IsInRole(string role);
  }
}
```

Текущий поток, выполняющий ваш код, содержит свойство `CurrentPrincipal`, которому может быть присвоен в качестве значения любой объект, реализующий интерфейс `IPrincipal`. В дальнейшем система будет обращаться к `CurrentPrincipal` за разрешением на выполнение защищенного действия.

Наиболее распространенным классом, реализующим этот интерфейс, является `GenericPrincipal`, который наследуется от `ClaimsPrincipal`:

```
namespace System.Security.Principal
{
  public class GenericPrincipal : ClaimsPrincipal
  {
    public GenericPrincipal(IIdentity identity, string[]? roles);
    public override IIdentity Identity { get; }
    public override bool IsInRole([NotNullWhen(true)] string? role);
  }
}
```

Реализация аутентификации и авторизации

Реализуем пример пользовательского механизма аутентификации и авторизации и рассмотрим эти процессы более подробно.

1. В проекте `CryptographyLib` добавьте свойство в класс `User` для хранения массива ролей:

   ```
   public string[]? Roles { get; set; }
   ```

2. В файле `User.cs` добавьте параметр для установки `Roles` в конструкторе.
3. Измените метод `Register` в классе `Protector` так, чтобы разрешить передачу массива ролей в качестве необязательного параметра, как показано в коде ниже (выделено жирным шрифтом):

   ```
   public static User Register(
     string username, string password,
     string[]? roles = null)
   ```

4. В методе `Register` добавьте параметр, чтобы установить массив ролей в новом объекте `User`:

   ```
   User user = new(username, saltText,
     saltedhashedPassword, roles);
   ```

5. В проекте `CryptographyLib` добавьте операторы в класс `Protector`, чтобы определить метод `LogIn` для входа в систему пользователя, и если имя пользователя и пароль действительны, то создайте общие идентификатор и приниципал

и назначьте их текущему потоку, указав, что тип аутентификации был пользовательским и назывался `PacktAuth`:

```
public static void LogIn(string username, string password)
{
  if (CheckPassword(username, password))
  {
    GenericIdentity gi = new(
      name: username, type: "PacktAuth");

    GenericPrincipal gp = new(
      identity: gi, roles: Users[username].Roles);

    // установка принципала в текущем потоке, чтобы он по умолчанию
    // использовался для авторизации
    Thread.CurrentPrincipal = gp;
  }
}
```

6. Откройте редактор кода и создайте консольное приложение `SecureApp` в рабочей области/решении `Chapter20`.
7. В программе Visual Studio Code выберите `SecureApp` в качестве активного проекта OmniSharp.
8. В проекте `SecureApp` добавьте ссылку на проект `CryptographyLib`.
9. Соберите проект `SecureApp` и убедитесь в отсутствии ошибок компиляции.
10. В файле `Program.cs` импортируйте необходимые пространства имен для работы с аутентификацией и авторизацией:

```
using Packt.Shared; // Protector
using System.Security; // SecurityException
using System.Security.Principal; // IPrincipal
using System.Security.Claims; // ClaimsPrincipal, Claim

using static System.Console;
```

11. Добавьте операторы для регистрации трех пользователей `Alice` (Алиса), `Bob` (Боб) и `Eve` (Ева), имеющих различные роли. Затем попросите пользователя войти в систему и выведите сведения о них:

```
Protector.Register("Alice", "Pa$$w0rd",
  roles: new[] { "Admins" });

Protector.Register("Bob", "Pa$$w0rd",
  roles: new[] { "Sales", "TeamLeads" });

// Ева не имеет никаких ролей
Protector.Register("Eve", "Pa$$w0rd");
```

```
// предлагаем пользователю ввести имя пользователя и пароль
// для входа под одним из этих трех пользователей

Write($"Enter your user name: ");
string? username = ReadLine();

Write($"Enter your password: ");
string? password = ReadLine();

if ((username == null) || (password == null))
{
  WriteLine("Username or password is null. Cannot login.");
  return;
}

Protector.LogIn(username, password);

if (Thread.CurrentPrincipal == null)
{
  WriteLine("Log in failed.");
  return;
}

IPrincipal 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:");

  IEnumerable<Claim>? claims = (p as ClaimsPrincipal)?.Claims;

  if (claims is not null)
  {
    foreach (Claim claim in claims)
    {
      WriteLine($"{claim.Type}: {claim.Value}");
    }
  }
}
```

12. Запустите код, войдите в систему под именем `Alice` (Алиса) с паролем `Pa$$w0rd` и проанализируйте результат:

```
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
```

13. Запустите код, войдите в систему под именем `Alice` (Алиса) с паролем `secret` и проанализируйте результат:

```
Enter your user name: Alice
Enter your password: secret
Log in failed.
```

14. Запустите код, войдите в систему под именем `Bob` (Боб) с паролем `Pa$$word` и проанализируйте результат:

```
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
```

Способы защиты приложения

Рассмотрим, как с помощью авторизации можно предотвратить доступ некоторых пользователей к тем или иным функциям приложения.

1. В конце файла `Program.cs` добавьте метод, защищенный проверкой разрешения внутри метода, и сгенерируйте соответствующие исключения, если пользователь анонимный или не член роли `Admins`:

```
static void SecureFeature()
{
  if (Thread.CurrentPrincipal == null)
  {
    throw new SecurityException(
```

```
      "A user must be logged in to access this feature.");
  }

  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.");
}
```

2. Над методом `SecureFeature` добавьте операторы для вызова метода `SecureFeature` в операторе `try`:

```
try
{
  SecureFeature();
}
catch (Exception ex)
{
  WriteLine($"{ex.GetType()}: {ex.Message}");
}
```

3. Запустите код, войдите в систему под именем `Alice` (Алиса) с паролем `Pa$$word` и проанализируйте результат:

```
You have access to this secure feature.
```

4. Запустите код, войдите в систему под именем `Bob` (Боб) с паролем `Pa$$word` и проанализируйте результат:

```
System.Security.SecurityException: User must be a member of Admins to
access this feature.
```

Аутентификация и авторизация в реальном мире

Хотя очень полезно увидеть примеры того, как могут работать аутентификация и авторизация, в реальном мире вам не следует создавать собственные системы безопасности, поскольку слишком велика вероятность того, что вы можете внести в них ошибки.

Вместо этого вам следует обратиться к коммерческим или открытым реализациям. Обычно они реализуют такие стандарты, как OAuth 2.0 и OpenID Connect. Популярным вариантом с открытым исходным кодом является *IdentityServer4*, но он будет поддерживаться только до ноября 2022 года. Полукоммерческий вариант — Duende IdentityServer.

Официальная позиция Microsoft заключается в следующем: «У Microsoft уже есть команда и продукт в этой области, Azure Active Directory, который позволяет бесплатно использовать 500 000 объектов». Более подробно об этом вы можете прочитать по следующей ссылке: https://devblogs.microsoft.com/aspnet/asp-net-core-6-and-authentication-servers/.

Практические задания

Проверьте полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 20.1. Проверочные вопросы

Ответьте на следующие 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, лучше всего подойдет для асимметричного шифрования?
3. Что такое радужная атака?
4. При использовании алгоритмов шифрования лучше применять блоки большого или малого размера?
5. Что такое криптографическое хеширование данных?
6. Что такое криптографическая подпись?
7. В чем разница между симметричным и асимметричным шифрованием?
8. Что такое RSA?
9. Почему пароли должны быть засолены перед сохранением?
10. SHA-1 — это алгоритм хеширования, разработанный Национальным агентством безопасности США. Почему его не следует использовать?

Упражнение 20.2. Защита данных с помощью шифрования и хеширования

В рабочей области/решении `Chapter10` создайте консольное приложение `Exercise02`, предназначенное для защиты конфиденциальных данных, таких как номер кредитной карты или пароль, хранящиеся в XML-файле, как показано в следующем примере:

```
<?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>
```

Обратите внимание: номер банковской карты и пароль клиента хранятся в открытом виде. Номер карты должен быть зашифрован, чтобы ее можно было расшифровать и использовать позже, а пароль следует засолить и хешировать.

Вы не должны хранить номера кредитных карт в своих приложениях. Это лишь пример секретных данных, которые вы, возможно, захотите защитить. Если вам приходится хранить номера кредитных карт, то вы должны приложить гораздо больше усилий, чтобы соответствовать требованиям индустрии платежных карт (Payment Card Industry, PCI).

Упражнение 20.3. Дешифрование данных

В рабочей области/решении `Chapter20` создайте консольное приложение `Exercise03`, открывающее XML-файл, защитой которого вы занимались в предыдущем упражнении, и расшифровывающее номер банковской карты.

Упражнение 20.4. Дополнительные ресурсы

Воспользуйтесь ссылками на странице https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-20---protecting-your-data-and-applications, чтобы получить дополнительную информацию по темам, приведенным в данной главе.

Резюме

В этой главе вы узнали, как зашифровать и дешифровывать данные с помощью симметричного шифрования, генерировать соленый хеш, подписывать данные и проверять цифровые подписи, генерировать истинно случайные числа и использовать аутентификацию и авторизацию для защиты ваших приложений.

Приложение
Ответы на проверочные вопросы

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

Глава 1. Привет, C#! Здравствуй, .NET!

1. Среда разработки Visual Studio 2022 превосходит Visual Studio Code?

   Ответ. Нет. Каждая из них оптимизирована для разных задач. Visual Studio 2022 для Windows — большая, тяжеловесная и может создавать приложения с графическими пользовательскими интерфейсами, например Windows Forms, WPF, UWP и приложения .NET MAUI, но доступна только на Windows. Visual Studio 2022 — это интерактивная среда разработки (Interactive Development Environment, IDE), а не редактор кода. Visual Studio Code меньше, она легковесная, ориентирована на код, поддерживает больше языков и является кросс-платформенной. Ожидается, что в ноябре 2022 года, с выходом .NET 7, Visual Studio Code получит расширение, которое облегчает создание пользовательских интерфейсов для приложений .NET MAUI с помощью паттерна проектирования MVU.

2. Платформа .NET 6 лучше .NET Framework?

   Ответ. Для современной разработки — да, но это зависит от того, что вам нужно. .NET 6 — это современная, кросс-платформенная, ориентированная на производительность версия устаревшего, зрелого .NET Framework. Ее чаще улучшают. .NET Framework лучше поддерживает устаревшие приложения; однако текущая версия 4.8 будет последним крупным выпуском. Она никогда не будет поддерживать некоторые языковые возможности C# 8 и более поздних версий.

3. Что такое .NET Standard и почему он все еще важен?

   Ответ. .NET Standard определяет API, также известный как контракт, который может реализовать платформа .NET. Последние версии .NET Framework, Xamarin и современной .NET реализуют .NET Standard 2.0, чтобы обеспечить

единый стандартный API, который разработчики могут повторно использовать любое количество раз. .NET Core 3.0 и более поздние версии, включая .NET 6, реализуют .NET Standard 2.1, в котором есть некоторые новые возможности, не поддерживаемые .NET Framework. Если вы хотите создать новую библиотеку классов, поддерживающую все платформы .NET, то она должна быть совместима с .NET Standard 2.0.

4. 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 код на промежуточном языке (IL). Затем этот IL-код с помощью общеязыковой исполняющей среды (CLR) преобразуется в машинные инструкции для процессора CPU во время выполнения.

5. Как называется метод точки входа консольного приложения .NET и как его объявить?

   Ответ. Метод `Main`. Рекомендуется использовать массив `string` для аргументов командной строки и тип возвращаемого значения `int`, хоть это и не обязательно. Метод может быть объявлен так, как показано в коде ниже:

```
public static void Main() // минимум
public static int Main(string[] args) // рекомендовано
```

6. Что такое программа верхнего уровня и как получить доступ к любым аргументам командной строки?

   Ответ. Программа верхнего уровня — это проект, в котором нет необходимости явно определять класс `Program` с точкой входа в метод `Main` с параметром `args` для доступа к любым аргументам командной строки. Они определяются для вас неявно, так что вы можете вводить операторы без шаблонного кода.

7. Что вы вводите в подсказке, чтобы создать и выполнить исходный код на языке C#?

   Ответ. В папке с файлом `.csproj` вы вводите `dotnet run`.

8. Каковы преимущества использования расширения .NET Interactive Notebooks для написания кода на языке C#?

   Ответ. К числу преимуществ можно отнести смешивание языков в одном документе и смешивание кода с разметкой для обогащенного текста.

9. Где найти справочную информацию по ключевому слову C#?

   Ответ. На сайте Microsoft Docs. В частности, документация по ключевым словам C# приведена на этой странице: https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/language-reference/keywords/.

10. Где найти решения общих проблем программирования?

    Ответ. На сайте https://stackoverflow.com/.

Глава 2. Говорим на языке C#

Упражнение 2.1. Проверочные вопросы

1. Какой оператор можно ввести в файл C#, чтобы узнать версию компилятора и языка?

   Ответ. Оператор `#error version`.

2. Каковы два типа комментариев в C#?

   Ответ. Однострочный комментарий с префиксом `//` и многострочный комментарий, начинающийся с `/*` и заканчивающийся `*/`.

3. В чем разница между дословной и интерполированной строкой?

   Ответ. Дословная строка начинается с символа `@`, и каждый символ (кроме `"`) интерпретируется как таковой; например, обратная косая черта `\` — это обратная косая черта `\`. Интерполированная строка начинается с символа `$` и может включать выражения, заключенные в фигурные скобки, например `{выражение}`.

4. Почему следует быть осторожными при использовании значений `float` и `double`?

   Ответ. Потому что их точность не гарантируется, особенно при выполнении сравнения на равенство.

5. Как определить, сколько байтов памяти использует такой тип, как `double`?

   Ответ. С помощью оператора `sizeof()`, например, `sizeof(double)`.

6. Когда следует использовать ключевое слово `var`?

   Ответ. Ключевое слово `var` нужно использовать только для объявления локальных переменных, когда вы не можете указать известный тип. Этим словом легко злоупотребить на начальном этапе написания кода, поскольку оно очень удобное, но его использование может усложнить последующее сопровождение кода.

7. Каков новейший способ создания экземпляра класса, такого как `XmlDocument`?

   Ответ. Этот способ заключается в использовании выражения `new` с целевым типом:

   ```
   XmlDocument doc = new();
   ```

8. Почему следует быть осторожными при использовании типа `dynamic`?

   Ответ. Потому что тип объекта, хранящегося в нем, не проверяется до времени выполнения, что может привести к возникновению исключений времени выполнения, если вы попытаетесь использовать член, который не существует в типе.

9. Как выровнять строку формата по правому краю?

   Ответ. После индекса или выражения добавьте запятую и целочисленное значение, чтобы указать ширину столбца, в пределах которого нужно выровнять

значение. Положительные целые числа означают выравнивание вправо, а отрицательные целые числа — выравнивание влево.

10. Какой символ разделяет аргументы в консольном приложении?

    Ответ. Символ пробела.

Упражнение 2.2. Проверочные вопросы о числовых типах

Какой тип следует 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 лучшей производительности на большинстве процессоров или `byte` (0–255) для наименьшего размера.

4. Размер оклада.

   Ответ. Тип `decimal`.

5. Международный стандартный книжный номер.

   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 или среднего бизнеса (примерно 50 000 сотрудников на предприятие).

    Ответ. Поскольку существуют сотни тысяч малых и средних предприятий, нам необходимо использовать размер памяти в качестве определяющего фактора, поэтому выберите тип `ushort`, так как он занимает всего 2 байта, в отличие от `int`, который занимает 4 байта.

Глава 3. Управление потоком исполнения, преобразование типов и обработка исключений

Упражнение 3.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Что произойдет, если разделить переменную `int` на `0`?

   Ответ. При делении целого или дробного числа вызывается исключение `DivideByZeroException`.

2. Что произойдет, если разделить переменную `double` на `0`?

   Ответ. Тип `double` содержит особое значение `Infinity`. Экземпляры чисел с плавающей запятой могут иметь специальные значения: `NaN` (не число) или, в случае деления на `0`, либо `PositiveInfinity`, либо `NegativeInfinity`.

3. Что происходит при переполнении переменной `int`, то есть когда вы присваиваете ей значение, выходящее за пределы допустимого диапазона?

   Ответ. Она будет работать в цикле, если вы не поместите оператор в блок `checked`. В последнем случае будет вызвано исключение `OverflowException`.

4. В чем разница между операторами `x = y++;` и `x = ++y;`?

   Ответ. В операторе `x = y++;` текущее значение `y` будет присвоено `x`, а затем инкрементировано. В операторе `x = ++y;` значение `y` сначала будет инкрементировано, а затем результат будет присвоен `x`.

5. В чем разница между операторами `break`, `continue` и `return` при использовании в операторах цикла?

   Ответ. Оператор `break` завершит весь цикл и продолжит выполнять код после цикла, оператор `continue` завершит текущую итерацию цикла и продолжит выполнение в начале блока цикла для следующей итерации, а оператор `return` завершит текущий вызов метода и продолжит выполнение после вызова метода.

6. Из каких трех частей состоит оператор `for` и какие из них обязательны?

   Ответ. Три части оператора `for` — это выражения инициализатора, условия и инкремента. Выражение условия должно быть логическим выражением, которое возвращает значение `true` или `false`, а две другие части необязательны.

7. В чем разница между операциями `=` и `==`?

   Ответ. С помощью операции `=` можно присвоить значения переменным, а `==` — операция проверки на равенство, возвращающая значение `true` или `false`.

8. Будет ли скомпилирован следующий оператор?

```
for ( ; true; ) ;
```

Ответ. Да. Для оператора `for` требуется только выражение условия, возвращающее как `true`, так и `false`. Выражения инициализатора и инкремента необязательны и могут быть опущены. В данном случае оператор `for` никогда не прекратит свое выполнение. Это пример бесконечного цикла.

9. Что представляет символ подчеркивания _ в выражении `switch`?

   Ответ. Возвращаемое по умолчанию значение.

10. Какой интерфейс должен реализовать объект, чтобы его можно было перечислить с помощью оператора `foreach`?

    Ответ. Интерфейс `IEnumerable`. Он должен иметь правильные методы с правильными подписями, даже если объект фактически не реализует интерфейс.

Упражнение 3.2. Циклы и переполнение

Что произойдет при выполнении кода, приведенного ниже?

```
int max = 500;
for (byte i = 0; i < max; i++)
{
  WriteLine(i);
}
```

Ответ. Код будет циклически работать без остановки, так как значение переменной `i` может быть только в диапазоне от `0` до `255`, поэтому, когда оно инкрементируется с шагом `255`, то возвращается к `0` и поэтому всегда будет меньше, чем `max` (`500`).

Чтобы предотвратить непрерывное выполнение цикла, вы можете обернуть код оператором `checked`. Это приведет к тому, что после достижения значения `255` из-за переполнения будет вызываться исключение:

```
254
255
System.OverflowException says Arithmetic operation resulted in an overflow.
```

Упражнение 3.5. Проверка знания операций

Каковы будут значения переменных `x` и `y` после выполнения следующих операторов?

1. Операции инкремента и сложения:

   ```
   x = 3;
   y = 2 + ++x;
   ```

   Ответ. `x` равен `4`; `y` равен `6`.

2. Операции бинарного сдвига:

```
x = 3 << 2;
y = 10 >> 1;
```

Ответ. x равен 12; y равен 5.

3. Побитовые операции:

```
x = 10 & 8;
y = 10 | 7;
```

Ответ. x равен 8; y равен 15.

Глава 4. Разработка, отладка и тестирование функций

1. Что в языке C# означает ключевое слово void?

Ответ. Оно указывает на то, что метод не имеет возвращаемого значения.

2. В чем разница между императивным и функциональным стилями программирования?

Ответ. Императивный стиль программирования означает написание последовательности операторов, которые исполняющая среда выполняет шаг за шагом, как рецепт. Написанный код сообщает среде выполнения, как именно выполнять задачу. Например, сначала необходимо выполнить шаг 1, затем шаг 2. Это парадигма программирования, в которой используются переменные, которые означают, что состояние программы может измениться в любой момент, в том числе вне текущей функции. Императивное программирование вызывает побочные эффекты, изменяя значение состояния вашей программы. Побочные эффекты сложно отладить. Стиль функционального программирования описывает то, чего вы хотите достичь, а не способ достижения результата. Данный стиль также можно назвать декларативным. Однако есть наиболее важный момент: во избежание побочных эффектов языки функционального программирования по умолчанию делают все состояния неизменяемыми.

3. В программе Visual Studio Code или Visual Studio какова разница между нажатием клавиш F5, Ctrl или Cmd+F5, Shift+F5 и Ctrl или Cmd+Shift+F5?

Ответ. Клавиша F5 сохраняет, компилирует, запускает и присоединяет отладчик; сочетание клавиш Ctrl или Cmd+F5 сохраняет, компилирует и запускает отладчик; сочетание клавиш Shift+F5 останавливает отладчик и работающее приложение, а сочетание клавиш Ctrl или Cmd+Shift+F5 перезапускает приложение без подключенного отладчика.

4. Куда записывает выходные данные метод `Trace.WriteLine`?

   Ответ. В любые настроенные прослушиватели трассировки. По умолчанию это терминал или командная строка, но также можно указать текстовый файл или любой пользовательский прослушиватель.

5. Каковы пять уровней трассировки?

   Ответ. `0` = `None`, `1` = `Error` (Ошибка), `2` = `Warning` (Предупреждение), `3` = `Info` (Информация) и `4` = `Verbose` (Подробная информация).

6. Чем различаются классы `Debug` и `Trace`?

   Ответ. Класс `Debug` активен только во время разработки. Класс `Trace` активен во время разработки и после выпуска в рабочую среду.

7. Как называются три А модульного теста?

   Ответ. Arrange, Act, Assert — «размещение, действие, утверждение».

8. При написании модульного теста с помощью xUnit каким атрибутом вы должны дополнять методы тестирования?

   Ответ. Атрибутом `[Fact]` или `[Theory]`.

9. Какая команда `dotnet` выполняет тесты xUnit?

   Ответ. Команда `dotnet test`.

10. Какой оператор следует использовать для повторного создания перехваченного исключения `ex` без потери трассировки стека?

    Ответ. Используйте оператор `throw;`. Не используйте `throw ex;`, так как это приведет к потере информации о трассировке стека.

Глава 5. Создание пользовательских типов с помощью объектно-ориентированного программирования

1. Какие шесть модификаторов доступа вы знаете и для чего они используются?

   Ответ. Шесть модификаторов доступа и их действие представлены ниже:

   - `private` — доступ ограничен содержащим классом;
   - `internal` — доступ ограничен текущей сборкой;
   - `protected` — доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса;
   - `internal protected` — доступ ограничен содержащим классом, производным классом или текущей сборкой;

- `private protected` — доступ ограничен содержащим или производным классом, находящимися в текущей сборке;
- `public` — неограниченный доступ.

2. Чем различаются ключевые слова `static`, `const` и `readonly` при их применении к члену типа?

   Ответ. Разница описана ниже:

   - `static` делает член общим для всех экземпляров, и доступ к нему должен осуществляться через тип, а не через экземпляр типа;
   - `const` устанавливает поле как фиксированное литеральное значение, которое никогда не должно меняться, так как в процессе компиляции сборки, которые используют это поле, копируют литеральное значение во время компиляции;
   - `readonly` создает поле, которое может быть назначено только для использования конструктора или инициализатора поля во время выполнения.

3. Для чего используется конструктор?

   Ответ. Для выделения памяти и инициализации значений полей.

4. Зачем применять атрибут `[Flags]` к типу `enum`, когда нужно хранить комбинированные значения?

   Ответ. Если не применить атрибут `[Flags]` к типу `enum` при сохранении скомбинированных значений, то сохраненное значение `enum`, представляющее собой комбинацию, будет возвращаться вызовом `ToString` в качестве сохраненного целочисленного значения вместо одного или нескольких списков текстовых значений, разделенных запятыми.

5. В чем польза ключевого слова `partial`?

   Ответ. Оно позволяет разделить определение типа на несколько файлов.

6. Что вы знаете о кортежах?

   Ответ. Кортеж — это структура данных, состоящая из нескольких частей. Они используются при необходимости сохранить несколько значений, при этом не определяя тип.

7. Для чего служит ключевое слово `record`?

   Ответ. Оно определяет структуру данных, которая по умолчанию является неизменной, что обеспечивает более функциональный стиль программирования. Как и класс, `record` может иметь свойства и методы, но значения свойств можно установить только во время инициализации.

8. Что такое перегрузка?

   Ответ. Перегрузка — это когда вы определяете несколько методов с одинаковым именем и разными входными параметрами.

9. В чем разница между полем и свойством?

   Ответ. Поле — это место хранения данных, на которое можно ссылаться. Свойство — это один или пара методов, которые получают и/или устанавливают значение. Значение свойства часто хранится в закрытом поле.

10. Как сделать параметр метода необязательным?

    Ответ. Нужно присвоить ему значение по умолчанию в сигнатуре метода.

Глава 6. Реализация интерфейсов и наследование классов

1. Что такое делегат?

   Ответ. Делегат — это типобезопасный указатель на метод. Его можно использовать для выполнения любого метода с соответствующей сигнатурой.

2. Что такое событие?

   Ответ. Событие — это поле, представляющее собой делегат, к которому применено ключевое слово `event`. Оно гарантирует, что используются только операторы `+=` и `-=`, позволяющие безопасно объединить несколько делегатов без замены каких-либо существующих обработчиков событий.

3. Как связаны базовый и производный классы и как производный может получить доступ к базовому?

   Ответ. Производный класс (подкласс) — это класс, унаследованный от базового класса (суперкласса). Чтобы получить доступ к классу, от которого наследуется подкласс, внутри производного класса необходимо использовать ключевое слово `base`.

4. В чем разница между операциями `is` и `as`?

   Ответ. Операция `is` возвращает значение `true`, если объект может быть приведен к типу; в противном случае возвращает значение `false`. Операция `as` возвращает указатель, если объект может быть приведен к типу; в противном случае возвращается значение `null`.

5. С помощью какого ключевого слова можно предотвратить наследование класса и переопределение метода?

   Ответ. Ключевое слово `sealed`.

6. Какое ключевое слово используется для предотвращения создания экземпляра класса с помощью нового ключевого слова `new`?

   Ответ. Ключевое слово `abstract`.

7. Какое ключевое слово служит для переопределения члена?

   Ответ. Ключевое слово `virtual`.

8. Чем деструктор отличается от метода деконструктора?

   Ответ. Деструктор, также известный как финализатор, должен использоваться для выпуска ресурсов, принадлежащих объекту. Метод деконструктора — функция C# 7 или более поздних версий, позволяющая разбивать сложный объект на более мелкие части. Это особенно полезно при работе с кортежами.

9. Как выглядят сигнатуры конструкторов, которые должны иметь все исключения?

   Ответ. Описания этих сигнатур приведены ниже:

   - конструктор без параметров;
   - конструктор с параметром `string`, обычно называемым `message`;
   - конструктор с параметром `string`, обычно называемым `message`, и параметром `Exception`, 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 другого типа. Вы определяете, какой тип хотите расширить, добавляя перед первым параметром этого типа в методе ключевое слово `this`.

Глава 7. Упаковка и распространение типов .NET

1. В чем разница между пространством имен и сборкой?

   Ответ. Пространство имен — это логический контейнер типа. Сборка — это физический контейнер типа. Чтобы использовать тип, разработчик должен ссылаться на его сборку. При желании разработчик может импортировать свое пространство имен или указать пространство имен при именовании типа.

2. Как вы ссылаетесь на другой проект в файле `.csproj`?

   Ответ. Нужно добавить элемент `<ProjectReference>`, который задает для атрибута `Include` путь к файлу эталонного проекта внутри элемента `<ItemGroup>`:

```
<ItemGroup>
  <ProjectReference Include="..\Calculator\Calculator.csproj" />
</ItemGroup>
```

3. В чем преимущество такого инструмента, как ILSpy?

   Ответ. Он позволяет научиться писать код на C# для платформы .NET, наблюдая за тем, как пишутся другие пакеты. Конечно, не используйте их интеллектуальную собственность. Но особенно полезно ознакомиться с тем, как разработчики Microsoft реализовали ключевые компоненты библиотек базовых классов. Декомпиляция также может быть полезна при вызове сторонней библиотеки, но, чтобы правильно вызывать ее, сначала лучше разобраться, как это делать.

4. Какой тип .NET представлен псевдонимом `float` в C#?

   Ответ. Тип `System.Single`.

5. Какой инструмент следует использовать перед переносом приложения с .NET Framework на .NET 6 и позволяет выполнить бóльшую часть работы по переносу?

   Ответ. Перед переносом приложения с .NET Framework на .NET 6 следует использовать .NET Portability Analyzer. Вы можете использовать .NET Upgrade Assistant для выполнения большей части работы по переносу.

6. В чем разница между платформенно-зависимым и автономным развертыванием современных приложений .NET?

   Ответ. Современные приложения .NET, зависящие от платформы, требуют установки .NET, чтобы операционная система могла их выполнить. Автономные приложения .NET включают в себя все необходимое для самостоятельного выполнения.

7. Что такое RID?

   Ответ. RID — сокращение от Runtime Identifier (идентификатор среды выполнения). Значения RID используются для определения целевых платформ, на которых выполняется приложение .NET.

8. Чем различаются команды `dotnet pack` и `dotnet publish`?

   Ответ. С помощью команды `dotnet pack` создается пакет NuGet, который затем может быть загружен в канал NuGet, такой как Microsoft. Команда `dotnet publish` позволяет поместить приложение и его зависимости в папку для развертывания в хост-системе.

9. Какие типы приложений, написанных для .NET Framework, можно перенести на современную .NET?

   Ответ. Консоль, ASP.NET MVC, ASP.NET Web API, Windows Forms и приложения Windows Presentation Foundation (WPF).

10. Можете ли вы использовать пакеты, написанные для .NET Framework, с современной .NET?

    Ответ. Да, при условии, что они вызывают только API в .NET Standard 2.0.

Глава 8. Работа с распространенными типами .NET

1. Какое максимальное количество символов можно сохранить в переменной `string`?

   Ответ. Максимальный размер переменной `string` составляет 2 Гбайт или примерно 1 миллиард символов, поскольку каждый символ занимает 2 байта из-за внутреннего использования кодировки Unicode (UTF-16) для символов в строке.

2. В каких случаях и почему нужно использовать тип `SecureString`?

   Ответ. Тип `string` хранит текстовые данные в памяти слишком долго, и при этом они не защищены. Тип `SecureString` шифрует текст и гарантирует немедленное освобождение памяти. Например, в WPF элемент управления `PasswordBox` сохраняет пароль в виде переменной `SecureString`, а при запуске нового процесса параметр `Password` должен быть переменной `SecureString`.

3. В каких ситуациях целесообразно применить класс `StringBuilder`?

   Ответ. При конкатенации свыше трех переменных `string` с помощью класса `StringBuilder` можно снизить потребление ресурсов памяти и улучшить производительность в сравнении со способами, предусматривающими использование метода `string.Concat` и операции `+`.

4. В каких случаях следует задействовать класс `LinkedList<T>`?

   Ответ. Каждый элемент в связанном списке содержит ссылку на предыдущие и следующие одноуровневые элементы, а также на сам список, поэтому его следует использовать, когда элементы необходимо вставлять в позиции в списки и удалять из них, не перемещая в памяти.

5. Когда класс `SortedDictionary<T>` нужно использовать вместо класса `SortedList<T>`?

   Ответ. Класс `SortedList<T>` использует меньше памяти, чем `SortedDictionary<T>`, а тот, в свою очередь, быстрее выполняет операции добавления и удаления несортированных данных. Если список заполняется уже отсортированными данными, то `SortedList<T>` работает быстрее, чем `SortedDictionary<T>`.

6. Каков ISO-код языковых и региональных параметров ISO для валлийского языка?

   Ответ. `cy-GB`.

7. В чем разница между локализацией, глобализацией и интернационализацией?

   Ответ

   - Локализация влияет на пользовательский интерфейс приложения. Определяется нейтральными (только языковыми) или специфичными (языковыми и региональными) настройками. Вы предоставляете текст и другие значения на нескольких языках. К примеру, метка текстового поля с именем может быть отображена как *First name* на английском языке и *Prénom* на французском.
   - Глобализация влияет на данные вашего 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 должны быть конкретными, чтобы форматирование десятичных значений денежных единиц было правильным, например канадские доллары вместо французских евро.
   - Интернационализация — это сочетание локализации и глобализации.

8. Что означает символ `$` в регулярных выражениях?

   Ответ. Конец ввода.

9. Как в регулярных выражениях представить цифровые символы?

   Ответ. Нужно использовать `\d` или `[0-9]`.

10. Почему *нельзя* использовать официальный стандарт для адресов электронной почты при создании регулярного выражения, призванного проверять адрес 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.

Глава 9. Работа с файлами, потоками и сериализация

1. Чем использование класса `File` отличается от использования класса `FileInfo`?

   Ответ. Класс `File` содержит методы `static`, 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. Чем различаются методы потока `ReadByte` и `Read`?

   Ответ. Метод `ReadByte` при каждом вызове возвращает один байт, а метод `Read` заполняет временный массив байтами до указанной длины. Обычно рекомендуется использовать метод `Read` для обработки сразу последовательности байтов.

3. В каких случаях используются классы `StringReader`, `TextReader` и `StreamReader`?

   Ответ

   - Класс `StringReader` используется для эффективного чтения из строки, хранящейся в памяти.
   - `TextReader` — это абстрактный класс, который наследуют классы `StringReader` и `StreamReader`, чтобы иметь совместную функциональность.
   - 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. Тип `DeflateStream` 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 на символ затрачивается при использовании кодировки UTF-8?

   Ответ. Количество байтов на символ, используемое кодировкой UTF-8, зависит от символа. Большинство символов латинского алфавита хранятся с помощью одного байта. Другим символам может потребоваться два и более байта.

6. Что такое граф объектов?

   Ответ. Графом объектов является любой экземпляр классов в памяти, которые ссылаются друг на друга, тем самым формируя набор связанных объектов. Например, объект `Customer` может иметь свойство, которое ссылается на коллекцию экземпляров `Order`.

7. Какой формат сериализации лучше всего минимизирует затраты памяти?

   Ответ. В объектной нотации JavaScript (JSON) хорошо сочетаются требования к пространству и практические факторы, такие как удобочитаемость. Однако буферы протокола лучше всего подходят для минимизации требований к пространству.

8. Какой формат сериализации позволяет обеспечить кросс-платформенную совместимость?

   Ответ. Расширяемый язык разметки (XML) все еще пригоден, если вам нужна максимальная совместимость, особенно с устаревшими системами, хотя JSON более эффективен, особенно если вам необходимо интегрироваться с веб-системами или буферами протокола, чтобы обеспечить лучшую производительность и использовать минимальную полосу пропускания.

9. Почему не рекомендуется использовать значение `string` наподобие `"\Code\Chapter01"` для представления пути и что необходимо выполнить вместо этого?

   Ответ. Использовать подобное значение для представления пути неправильно, так как предполагается, что символ «обратный слеш» служит в качестве разделителя папок во всех операционных системах. Вместо этого следует использовать метод `Path.Combine` и передавать отдельные значения `string` или массив `string` для каждой папки, как показано ниже:

```
string path = Path.Combine(new[] { "Code", "Chapter01" });
```

10. Где можно найти информацию о пакетах NuGet и их зависимостях?

    Ответ. На сайте https://www.nuget.org/.

Глава 10. Работа с данными с помощью Entity Framework Core

1. Какой тип вы бы использовали для свойства, представляющего таблицу, например для свойства `Products` контекста базы данных?

   Ответ. Тип `DbSet<T>`, где `T` — тип объекта, например `Product`.

2. Какой тип вы бы применили для свойства, которое представляет отношение «один ко многим», скажем свойство `Products` объекта `Category`?

   Ответ. Тип `ICollection<T>`, где `T` — тип объекта, например `Product`.

3. Какое соглашение, касающееся первичных ключей, действует в EF?

   Ответ. Предполагается, что свойство `ID`, или `Id`, или `ClassNameID`, или `ClassNameId` является 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 воспользовались атрибутом аннотаций в классе сущности?

   Ответ. Когда соглашения не могут обеспечить правильное сопоставление между классами и таблицами. Например, если имя класса не соответствует имени таблицы или имя свойства не соответствует имени столбца. Вы также можете определить ограничения, такие как максимальная длина символов в текстовом значении или диапазон числовых значений, добавив атрибуты проверки. Такие технологии, как ASP.NET Core MVC и Blazor, могут считывать их, чтобы предоставлять пользователям предупреждения об автоматической проверке.

5. Почему вы предпочли бы задействовать Fluent API, а не атрибуты аннотации?

   Ответ. Вы можете выбрать Fluent API, а не атрибуты аннотации, если требуется, чтобы классы элементов были свободными от постороннего кода, который не нужен во всех сценариях. Например, при создании библиотеки классов .NET Standard 2.0 для классов сущностей вам может потребоваться использовать только атрибуты проверки, чтобы эти метаданные могли считываться Entity Framework Core и такими технологиями, как проверка привязки модели ASP.NET Core и настольные и мобильные приложения .NET MAUI. Однако вы можете использовать Fluent API для определения специальных возможностей Entity Framework Core, таких как сопоставление с именем другой таблицы или столбца.

6. Что означает уровень изоляции транзакции `Serializable`?

   Ответ. Максимальные блокировки применяются для обеспечения полной изоляции от любых других процессов, работающих с затронутыми данными.

7. Что возвращает в результате метод `DbContext.SaveChanges()`?

   Ответ. Значение `int` для количества затронутых объектов.

8. В чем разница между жадной и явной загрузками?

   Ответ. Жадная загрузка означает, что связанные объекты включены в исходный запрос к базе данных, поэтому их не нужно загружать позже. Явная загрузка означает, что связанные объекты не включены в исходный запрос к БД и должны быть явно загружены непосредственно перед тем, как понадобятся.

9. Как определить сущностный класс EF Core, чтобы он соответствовал следующей таблице?

```
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. Какие преимущества дает объявление свойств навигации сущностей как `virtual`?

    Ответ. Это позволяет включить ленивую загрузку.

Глава 11. 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 вы должны импортировать пространство имен `System.Linq`, а затем обращаться к сборке поставщика данных того типа данных, с которыми хотите работать, за исключением провайдеров LINQ to Objects и LINQ to XML, которые встроены в .NET.

2. Какой метод расширения LINQ вы использовали бы для возврата подмножества свойств из типа?

   Ответ. Метод `Select` позволяет осуществлять проекцию (выбор) свойств.

3. С помощью какого метода расширения LINQ вы бы выполнили фильтрацию последовательности?

   Ответ. Метод `Where` позволяет выполнить фильтрацию, предоставляя делегат (или лямбда-выражение), который возвращает логическое значение, показывающее, нужно ли включать значение в результаты.

4. Каковы пять методов расширения LINQ, выполняющих агрегацию данных?

   Ответ. Любые пять из следующих: `Max`, `Min`, `Count`, `LongCount`, `Average`, `Sum` и `Aggregate`.

5. Чем различаются методы расширения `Select` и `SelectMany`?

   Ответ. Метод `Select` возвращает именно то, что вы указали. 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` и объединит их в одну последовательность.

6. В чем разница между интерфейсами `IEnumerable<T>` и `IQueryable<T>` и как вы переключаетесь между ними?

   Ответ. Интерфейс `IEnumerable<T>` указывает поставщик LINQ, который будет выполнять запрос локально, например LINQ to Objects. Эти поставщики не имеют ограничений, но могут быть менее эффективными.

   Интерфейс `IQueryable<T>` указывает поставщик LINQ, который сначала создает дерево выражений для представления запроса, а затем преобразует его в другой синтаксис запроса перед его выполнением, подобно тому как Entity Framework Core преобразует запросы LINQ в операторы SQL. Эти поставщики иногда имеют ограничения, например отсутствие поддержки определенных выражений, и могут создавать исключения. Вы можете превратить из поставщика `IQueryable<T>` в поставщик `IEnumerable<T>`, вызвав метод `AsEnumerable`.

7. Что представляет последний параметр типа `T` в делегатах-дженериках `Func`, таких как `Func<T1, T2, T>`?

   Ответ. Тип возвращаемого значения. Например, для `Func<string, int, bool>` используемая функция делегата или лямбда-функции должны возвращать логическое значение.

8. В чем преимущество метода расширения LINQ, который заканчивается оператором `OrDefault`?

   Ответ. Он возвращает значение по умолчанию, а не выдает исключение, если не может вернуть значение. Например, вызов метода `First` для последовательности значений `int` вызовет исключение, если коллекция пуста, но метод `FirstOrDefault` вернет в результате `0`.

9. Почему понятный синтаксис запросов необязателен?

   Ответ. Это просто синтаксический сахар. Он облегчает чтение кода разработчиками, но не добавляет никакой дополнительной функциональности, кроме ключевого слова `let`.

10. Каким образом вы можете создать собственные методы расширения LINQ?

    Ответ. Создайте класс `static`, содержащий метод `static` с параметром `IEnumerable<T>` и префиксом `this`, как показано ниже:

```
namespace System.Linq
{
  public static class MyLinqExtensionMethods
```

```
    {
      public static IEnumerable<T> MyChainableExtensionMethod<T>(
        this IEnumerable<T> sequence)
      {
        // возвращает значение IEnumerable<T>
      }

      public static int? MyAggregateExtensionMethod<T>(
        this IEnumerable<T> sequence)
      {
        // возвращает некое целочисленное значение
      }
    }
  }
```

Глава 12. Улучшение производительности и масштабируемости с помощью многозадачности

1. Какую информацию о классе `Process` вы можете найти?

   Ответ. Класс `Process` содержит ряд свойств, включая: `ExitCode`, `ExitTime`, `Id`, `MachineName`, `PagedMemorySize64`, `ProcessorAffinity`, `StandardInput`, `StandardOutput`, `StartTime`, `Threads` и `TotalProcessorTime`.

2. Насколько точен класс `Stopwatch`?

   Ответ. С точностью до наносекунды (миллиардной доли секунды), но все равно может быть погрешность.

3. По соглашению, какой суффикс следует применять к методу, если он возвращает `Task` или `Task<T>`?

   Ответ. Суффикс `Async`. Например, при использовании синхронного метода `Open` используйте `OpenAsync` для эквивалента, возвращающего `Task` или `Task<T>`.

4. Какое ключевое слово нужно применить к объявлению метода, чтобы в нем можно было использовать ключевое слово `await`?

   Ответ. Ключевое слово `async`.

5. Как создать дочернюю задачу?

   Ответ. 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. Используйте метод `Monitor.Enter` и передайте аргумент `TimeSpan` в качестве тайм-аута, а затем явно вызовите метод `Monitor.Exit` для снятия блокировки в конце работы.

7. В каких случаях нужно применять класс `Interlocked`?

   Ответ. Когда в коде используются целые числа и числа с плавающей запятой, распределяемые между несколькими потоками.

8. Когда класс `Mutex` следует задействовать вместо класса `Monitor`?

   Ответ. Используйте класс `Mutex`, когда вам необходимо поделиться ресурсом, выходя за границы процесса. Класс `Monitor` работает только с ресурсами внутри текущего процесса.

9. В чем преимущество использования ключевых слов `async` и `await` на сайте или в веб-сервисе?

   Ответ. Это улучшает масштабируемость, но не производительность конкретного запроса, поскольку требуются дополнительные усилия по обработке взаимодействия между потоками.

10. Вы можете отменить задачу? Если да, то каким образом?

    Ответ. Да, вы можете отменить задачу. Информация о том, как это сделать, представлена на странице https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/async/cancel-an-async-task-or-a-list-of-tasks.

Глава 13. Реальные приложения на C# и .NET

1. .NET 6 является кросс-платформенной. Приложения Windows Forms и WPF могут работать на .NET 6. Могут ли эти приложения работать на macOS и Linux?

   Ответ. Нет. Хотя приложения Windows Forms и WPF могут работать на .NET 6, они также должны вызывать Win32 API, поэтому ограничены возможностями Windows. Когда вы загружаете .NET SDK для Windows, он включает дополнительную рабочую нагрузку, чтобы обеспечить поддержку приложений WPF и Windows Forms.

2. Как приложение Windows Forms определяет свой пользовательский интерфейс и почему это может быть проблемой?

   Ответ. Приложение Windows Forms определяет свой пользовательский интерфейс с помощью кода C#. Поэтому важно использовать визуальный конструктор Windows Forms с панелью инструментов и поддержкой перетаскивания. Непосредственно редактировать сгенерированный код напрямую довольно сложно.

3. Как приложение WPF или UWP может определять свой пользовательский интерфейс и почему это хорошо для разработчиков?

   Ответ. Приложение WPF или UWP может определять свой пользовательский интерфейс с помощью XAML, который разработчикам понять и написать проще, чем код C#, даже без визуального конструктора. XAML также используется в .NET MAUI.

Глава 14. Разработка сайтов с помощью ASP.NET Core Razor Pages

1. Каковы шесть методов, которые могут быть указаны в HTTP-запросе?

   Ответ. `GET`, `HEAD`, `POST`, `PUT`, `PATCH`, `DELETE`. Другие включают `TRACE`, `OPTIONS` и `CONNECT`.

2. Какие шесть кодов состояния и их описания могут быть возвращены в HTTP-ответе?

   Ответ. `200 OK`, `201 Created`, `301 Moved Permanently`, `400 Bad Request`, `404 Not Found` (отсутствует ресурс) и `500 Internal Server Error`. Другие включают `101 Switching Protocols` (например, с HTTP на WebSocket), `202 Accepted`, `204 No Content`, `304 Not Modified`, `401 Unauthorized`, `403 Forbidden`, `406 Not Acceptable` (например, запрашивается формат ответа, который не поддерживается сайтом), `503 Service Unavailable`.

3. Для чего в ASP.NET Core используется класс `Startup`?

   Ответ. Чтобы отделить добавление и настройку сервисов зависимости от настройки промежуточного программного обеспечения в конвейере запросов и ответов, таких как обработка ошибок, параметры безопасности, статические файлы, файлы по умолчанию, маршрутизация конечных точек, Razor Pages и MVC и контексты данных Entity Framework Core. Шаблоны проектов по умолчанию для ASP.NET Core 6 не используют класс `Startup` и вместо этого помещают весь код конфигурации в файл `Program.cs`.

4. Что такое HSTS и для чего он предназначен?

   Ответ. HTTP Strict Transport Security (HSTS) — дополнительный механизм улучшения безопасности. Если сайт указывает его и браузер его поддерживает, то он принудительно передает все данные по HTTPS и не позволяет посетителю использовать ненадежные или недействительные сертификаты.

5. Как включить статические HTML-страницы для сайта?

   Ответ. Необходимо добавить операторы в метод `Configure` класса `Startup`, чтобы использовать файлы по умолчанию, а затем использовать статические файлы (этот порядок важен!):

```
app.UseDefaultFiles(); // index.html, default.html и т. п.
app.UseStaticFiles();
```

6. Как вставить код C# в HTML, чтобы создать динамическую страницу?

   Ответ. Чтобы смешать код языка C# с HTML и создать динамическую страницу, вы можете создать файл Razor с расширением `.cshtml`. Затем вы можете добавить префикс любых выражений C# с символом `@`, а операторы C# заключить в фигурные скобки или создать раздел `@functions`, как показано ниже:

```
@page
@functions
{
  public string[] DaysOfTheWeek
  {
    get => System.Threading.Thread.CurrentThread
      .CurrentCulture.DateTimeFormat.DayNames;
  }

  public string WhatDayIsIt
  {
    get => System.DateTime.Now.ToString("dddd");
  }
}
<html>
  <head>
    <title>Today is @WhatDayIsIt</title>
  </head>
  <body>
    <h1>Days of the week in your culture</h1>
    <ul>
    @{
      // чтобы добавить блок операторов: используйте фигурные скобки
      foreach (string dayName in DaysOfTheWeek)
      {
        <li>@dayName</li>
      }
    }
    </ul>
  </body>
</html>
```

7. Как определить общие макеты для Razor Pages?

   Ответ. Создайте как минимум два файла: `_Layout.cshtml` для определения разметки для общего макета и `_ViewStart.cshtml` для установки макета по умолчанию:

```
@{
  Layout = "_Layout";
}
```

8. Как отделить разметку от выделенного кода на странице Razor?

   Ответ. Создайте два файла: `MyPage.cshtml`, который содержит разметку, и `MyPage.cshtml.cs`, который содержит класс, наследуемый от `PageModel`. В файле `MyPage.cshtml` установите модель для использования класса, как показано ниже:

```
@page
@model MyProject.Pages.MyPageModel
```

9. Как настроить контекст данных Entity Framework Core для использования с сайтом ASP.NET Core?

   Ответ. Необходимо выполнить следующие действия:

   - в файле проекта указать ссылку на сборку, определяющую класс контекста данных;
   - в файле Program.cs или в классе `Startup` импортировать пространства имен для `Microsoft.EntityFrameworkCore` и класс контекста данных;
   - в методе `ConfigureServices` или в разделе `Program.cs`, в котором настраиваются сервисы, добавить оператор, который настраивает контекст данных со строкой подключения к базе данных для использования с указанным поставщиком базы данных, таким как SQLite или SQL Server:

     ```
     services.AddDbContext<MyDataContext>(options => // or UseSqlServer()
       options.UseSqlite("my database connection string"));
     ```

   - в классе модели Razor Page или в разделе `@functions` объявить скрытое поле для хранения контекста данных, а затем установить его в конструкторе:

     ```
     private MyDataContext db;
     public SuppliersModel(MyDataContext injectedContext)
     {
       db = injectedContext;
     }
     ```

10. Как повторно использовать Razor Pages в ASP.NET Core 2.2 или более поздней версии?

    Ответ. Все, что связано со страницей Razor, можно скомпилировать в библиотеку классов, используя следующую команду:

    ```
    dotnet new razorclasslib -s
    ```

Глава 15. Разработка сайтов с помощью паттерна MVC

1. Зачем нужны файлы со специальными именами `_ViewStart` и `_ViewImports`, созданные в папке Views?

   Ответ

   - Файл `_ViewStart` содержит блок операторов, которые выполняются при выполнении метода `View`, когда метод действия контроллера передает модель в представление, например, для установки макета по умолчанию.

- Файл `_ViewImports` содержит операторы `@using` для импорта пространств имен для всех представлений, чтобы избежать необходимости добавлять одни и те же операторы импорта в начале всех представлений.

2. Как называются три сегмента, определенные по умолчанию в ASP. NET Core MVC, что они представляют и какие из них необязательны?

   Ответ

   - `{controller}` — например, `/shippers`, представляет класс контроллера для создания экземпляра, например `HomeController`. Необязателен, так как может использовать значение по умолчанию `Home`.
   - `{action}` — например, `/privacy`, представляет метод действия для выполнения, например, `Privacy`. Необязателен, так как может использовать значение по умолчанию `Index`.
   - `{id}` — например, `/5` представляет параметр в методе действия, например `int id`. Необязателен, так как к нему добавляется символ `?`.

3. Что делает связыватель модели по умолчанию и какие типы данных он может обрабатывать?

   Ответ. Связыватель модели по умолчанию устанавливает параметры в методе действия. Он может обрабатывать следующие типы данных:

   - простые типы, такие как `int`, `string`, `DateTime`, включая типы, допускающие значение `null`;
   - сложные типы, такие как `Person`, `Product`;
   - типы коллекции, такие как `ICollection<T>` или `IList<T>`.

4. Как вывести содержимое текущего представления в общем файле макета, таком как `_Layout.cshtml`?

   Ответ. Вызовите метод `RenderBody`:

```
@RenderBody()
```

5. Как в общем файле макета, таком как `_Layout.cshtml`, вывести раздел, для которого содержимое может быть предоставлено текущим представлением, и как это представление выдает содержимое для данного раздела?

   Ответ. Чтобы вывести содержимое раздела в общем макете, вызовите метод `RenderSection`, указав при необходимости имя раздела:

```
@RenderSection("Scripts", required: false)
```

   Чтобы определить содержимое раздела в представлении, создайте именованный раздел, как показано ниже:

```
@section Scripts
{
```

```
<script>
  alert('Hello, Mr. Page!');
</script>
}
```

6. Какие пути ищутся для представления по соглашению при вызове метода `View` внутри метода действия контроллера?

   Ответ. При вызове метода `View` внутри метода действия контроллера выполняется поиск трех путей для представления по умолчанию на основе сочетаний имен контроллера и метода действия и специальной папки Shared:

```
InvalidOperationException: The view 'Index' was not found. The following
locations were searched:
/Views/Home/Index.cshtml
/Views/Shared/Index.cshtml
/Pages/Shared/Index.cshtml
```

   Эти три пути можно обобщить следующим образом:

   - `/Views/[controller]/[action].cshtml`;
   - `/Views/Shared/[action].cshtml`;
   - `/Pages/Shared/[action].cshtml`.

7. Как выдать браузеру пользователя инструкцию кэшировать ответ в течение 24 ч?

   Ответ. Дополните класс контроллера или метод действия атрибутом `[ResponseCache]` и установите для параметра `Duration` значение `86400` с, а для параметра `Location` — `ResponseCacheLocation.Client`.

8. Почему вы можете включить Razor Pages, даже если сами их не создаете?

   Ответ. Если вы использовали такие функции, как ASP.NET Core Identity UI, то для этого требуются Razor Pages.

9. Как ASP.NET Core MVC идентифицирует классы, которые могут действовать как контроллеры?

   Ответ. ASP.NET Core MVC проверяет, не дополнен ли класс (или класс, из которого он происходит) атрибутом `[Controller]`.

10. Каким образом ASP.NET Core MVC облегчает тестирование сайта?

    Ответ. Паттерн проектирования Model-View-Controller (MVC) разделяет технические аспекты формы данных (модели), исполняемых операторов для обработки входящего запроса и исходящего ответа. Затем он генерирует ответ в формате, запрошенном пользовательским агентом, например HTML или JSON. Это облегчает написание модульных тестов. ASP.NET Core также упрощает реализацию паттернов проектирования Inversion-of-Control (IoC) и Dependency Injection (DI) для удаления зависимостей при тестировании компонента, такого как контроллер.

Глава 16. Разработка и использование веб-сервисов

1. От какого класса вы должны наследовать, чтобы создать класс контроллера для сервиса ASP.NET Core Web API?

   Ответ. Вы должны наследовать от класса `ControllerBase`. Не наследуйте от `Controller`, как в MVC, поскольку этот класс включает такие методы, как `View`, использующие файлы Razor для визуализации HTML, которые не нужны для веб-сервиса.

2. Если вы дополнили свой класс `Controller` атрибутом `[ApiController]`, чтобы получить поведение по умолчанию, например автоматический ответ со кодом статуса `400` для недопустимых моделей, то что еще должны сделать?

   Ответ. Вам следует также вызвать метод `SetCompatibilityVersion` в классе `Startup`.

3. Что вы должны сделать, чтобы указать, какой метод действия контроллера будет выполняться в ответ на HTTP-запрос?

   Ответ. Вы должны дополнить метод действия атрибутом. Например, чтобы ответить на HTTP-запрос `POST`, дополните метод действия атрибутом `[HttpPost]`.

4. Что вы должны сделать, чтобы указать, какие ответы следует ожидать при вызове метода действия?

   Ответ. Дополните метод действия атрибутом `[ProducesResponseType]`, как показано ниже:

```
// GET: api/customers/[id]
[HttpGet("{id}", Name = nameof(Get))] // именованный маршрут
[ProducesResponseType(200, Type = typeof(Customer))]
[ProducesResponseType(404)]
public IActionResult Get(string id)
{
```

5. Какие методы можно вызывать для возврата ответов с разными кодами состояния?

   Ответ. Это методы:

   - `Ok` — возвращает код состояния `200` и объект, переданный этому методу в теле кода;
   - `CreatedAtRoute` — возвращает код состояния `201` и объект, переданный этому методу в теле кода;
   - `NoContentResult` — возвращает код состояния `204` и пустое тело кода;

- `BadRequest` — возвращает код состояния `400` и необязательное сообщение об ошибке;
- `NotFound` — возвращает код состояния `404` и необязательное сообщение об ошибке.

6. Каковы четыре способа тестирования веб-сервиса?

   Ответ. Описание способов представлено ниже:

   - использование браузера для проверки простых HTTP-запросов `GET`;
   - установка расширения REST Client для Visual Studio Code;
   - установка пакета Swagger NuGet в свой проект веб-сервиса, включение Swagger и применение пользовательского интерфейса тестирования Swagger;
   - установка инструмента Postman по следующей ссылке: https://www.postman.com.

7. Почему вам не следует оборачивать `HttpClient` в оператор `using`, чтобы очистить его ресурсы, когда вы закончите с ним работать, даже если он реализует интерфейс `IDisposable`, и что вы должны использовать вместо этого?

   Ответ. `HttpClient` является общим, реентерабельным и частично поточно-ориентированным, поэтому его сложно правильно использовать во многих сценариях. Следует применять `HttpClientFactory`, представленный в .NET Core 2.1.

8. Что такое CORS и почему важно включить ее в веб-сервис?

   Ответ. Аббревиатура CORS расшифровывается как Cross-Origin Resource Sharing (совместное использование ресурсов между разными источниками). Это систему важно включить для веб-сервиса, поскольку в целях повышения безопасности по умолчанию политика для одного и того же браузера не позволяет коду, загруженному из одного источника, получать доступ к ресурсам, скачанным из другого источника.

9. Как вы можете разрешить клиентам определять работоспособность вашего веб-сервиса в ASP.NET Core 2.2 и более поздних версиях?

   Ответ. Вы можете установить API проверки работоспособности, включая проверки работоспособности базы данных для контекстов данных Entity Framework Core. Чтобы сообщить клиенту подробную информацию, проверку работоспособности можно расширить.

10. Какие преимущества обеспечивает маршрутизация конечных точек?

    Ответ. Маршрутизация конечных точек обеспечивает улучшенную производительность при маршрутизации и выборе метода действия, а также сервис генерации ссылок.

Глава 17. Создание пользовательских интерфейсов с помощью Blazor

1. Какие две основные модели хостинга существуют в Blazor и чем они различаются?

   Ответ. Две основные модели хостинга для Blazor — это Server и WebAssembly.

   - Blazor Server выполняет код на стороне сервера. Это означает, что код имеет полный и простой доступ к ресурсам на стороне сервера, таким как базы данных. Это может упростить реализацию функциональности. Пользовательский интерфейс обновляется с помощью SignalR. Это значит, между браузером и сервером необходимо постоянное соединение, что, в свою очередь, ограничивает масштабируемость.
   - Blazor WebAssembly выполняет код на стороне клиента. Это значит, код имеет доступ только к ресурсам в браузере, что может усложнить реализацию, поскольку всякий раз, когда требуются новые данные, должен выполняться обратный вызов на сервер.

2. Какая дополнительная конфигурация требуется в классе `Startup` в проекте сайта Blazor Server по сравнению с проектом сайта ASP.NET Core MVC?

   Ответ. В классе `Startup` в методе `ConfigureServices` необходимо вызвать метод `AddServerSideBlazor`, а в методе `Configure` при настройке конечных точек необходимо вызвать `MapBlazorHub` и `MapFallbackToPage`.

3. Одним из преимуществ Blazor является возможность реализации компонентов пользовательского интерфейса с помощью C# и .NET вместо JavaScript. Необходим ли Blazor какой-либо JavaScript?

   Ответ. Да, компонентам Blazor требуется минимальный JavaScript. Для Blazor Server этот вопрос можно решить, используя файл `_framework/blazor.server.js`, для Blazor WebAssembly — файл `_framework/blazor.webassembly.js`. Blazor WebAssembly с PWA также использует рабочий файл сервиса JavaScript `service-worker.js`. JavaScript также необходим для вызова браузера и других API на стороне клиента.

4. Какова роль файла `App.razor` в проекте Blazor?

   Ответ. Файл `App.razor` настраивает маршрутизатор, используемый всеми компонентами Blazor в текущей сборке. Например, он устанавливает общий макет по умолчанию для компонентов, которые соответствуют маршруту, и представление, которое будет использоваться, если совпадения не обнаружены.

5. В чем преимущество использования компонента `<NavLink>`?

   Ответ. Он интегрируется с системой маршрутизации Blazor, поэтому может автоматически применять текущий стиль для визуального указания, когда текущий маршрут соответствует компоненту `<NavLink>`.

6. Каким образом значение передается компоненту?

   Ответ. Вы можете передать значение в компонент, дополнив общедоступное свойство в компоненте атрибутом `[Parameter]`, а затем установив атрибут в компоненте при его использовании:

```
// определение компонента
@code {
  [Parameter]
  public string ButtonText { get; set; };
}

// использование компонента
<CustomerDetail ButtonText="Create Customer" />
```

7. В чем преимущество использования компонента `<EditForm>`?

   Ответ. Он позволяет получать автоматические сообщения проверки.

8. Каким образом выполняются некоторые операторы при установленных параметрах?

   Ответ. Когда параметры установлены, вы можете выполнить некоторые операторы, определив метод `OnParametersSetAsync` для обработки события.

9. Каким образом выполняются некоторые операторы при появлении компонента?

   Ответ. При обнаружении компонента вы можете выполнить некоторые операторы, определив метод `OnInitializedAsync` для обработки события.

10. Каковы два ключевых различия в классе `Program` между сервером Blazor и проектом Blazor WebAssembly?

    Ответ. Различия выглядят следующим образом: 1) использование `WebAssemblyHostBuilder` вместо `Host.CreateDefaultBuilder` и 2) регистрация `HttpClient` с базовым адресом среды хоста.

Глава 18. Создание и использование специализированных сервисов

1. У вас есть приложение, которое взаимодействует с сервисом, созданным с помощью устаревшего сервиса Windows Communication Foundation. Каковы два возможных варианта переноса сервиса и клиента на современную .NET?

   Ответ. Первый вариант: использование проекта Core WCF с открытым исходным кодом. Второй вариант: повторная реализация сервиса и клиента с помощью gRPC.

2. Какой транспортный протокол использует сервис OData?

   Ответ. Транспортный протокол HyperText Transport Protocol (HTTP).

3. Почему веб-сервис OData является более гибким, чем традиционный веб-сервис ASP.NET Core Web API?

   Ответ. OData использует строки запросов для своих запросов, что позволяет клиенту контролировать то, что возвращается, и сводит к минимуму циклы полного обхода. Традиционный Web API определяет все методы и то, что возвращается.

4. Что нужно сделать с методом действия в контроллере OData, чтобы включить строки запроса для настройки того, что он возвращает?

   Ответ. Нужно дополнить метод действия в контроллере OData атрибутом `[EnableQuery]`.

5. Какой транспортный протокол использует сервис GraphQL?

   Ответ. HTTP или другие протоколы, например WebSocket.

6. Как определяются контракты в gRPC?

   Ответ. С помощью файлов `.proto`.

7. Благодаря каким трем преимуществам gRPC хорошо подходит для реализации сервисов?

   Ответ. Эти преимущества описаны ниже:

   1) его двоичная сериализация Protobuf, которая минимизирует использование сети;
   2) его требование HTTP/2, которое обеспечивает значительные преимущества в производительности;
   3) его поддержка почти всеми языками и платформами.

8. Какие средства передачи данных использует SignalR и какой из них применяется по умолчанию?

   Ответ. SignalR предпочитает использовать WebSockets в качестве своего средства передачи данных, затем он будет возвращаться к событиям на стороне сервера, и, наконец, будет использовать длинный опрос, если ни один из них не поддерживается клиентом и сервером.

9. В чем разница между моделями внутрипроцессного и изолированного хостингов для Azure Functions?

   Ответ. Модель внутрипроцессного хостинга требует, чтобы ваша функция Azure загружалась вместе с другим кодом и была нацелена на заранее определенную версию LTS-выпуска, например .NET Core 3.1 или .NET 6. Модель изолированного хостинга позволяет вашей функции Azure загружаться в собственный процесс и использовать любую версию .NET, которую вы выберете.

10. Что рекомендуется делать при разработке сигнатур методов RPC?

    Ответ. Важно определять один параметр с помощью сложного типа. Это позволяет в будущем добавлять к типу дополнительные свойства, не нарушая контракт между клиентом и сервисов.

Глава 19. Разработка мобильных и настольных приложений с помощью .NET MAUI

1. Каковы четыре категории компонентов пользовательского интерфейса .NET MAUI и что они собой представляют?

   Ответ. Эти категории перечислены ниже:

   - *страницы* представляют экраны мобильных приложений;
   - *макеты* представляют структуру комбинации других компонентов пользовательского интерфейса;
   - *представления* определяют отдельный компонент пользовательского интерфейса;
   - *ячейки* представляют отдельный элемент в виде списка или таблицы.

2. Каковы четыре типа ячеек?

   Ответ. `TextCell`, `SwitchCell`, `EntryCell` и `ImageCell`.

3. Каким образом вы можете разрешить пользователю выполнять действия с ячейками в `ListView`?

   Ответ. Вы можете установить некоторые контекстные действия, представляющие собой пункты меню, которые вызывают событие, как показано ниже:

```
<TextCell Text="{Binding CompanyName}"
          Detail="{Binding Location}"
          TextColor="{DynamicResource PrimaryTextColor}"
          DetailColor="{DynamicResource PrimaryTextColor}" >
  <TextCell.ContextActions>
    <MenuItem Clicked="Customer_Phoned" Text="Phone" />
    <MenuItem Clicked="Customer_Deleted" Text="Delete"
              IsDestructive="True" />
  </TextCell.ContextActions>
</TextCell>
```

4. В каком случае элемент `Entry` используется вместо элемента `Editor`?

   Ответ. Элемент `Entry` используется для одной строки текста, а `Editor` — для нескольких строк.

5. В чем заключается эффект установки `IsDestructive` в значение `true` для пункта меню в контекстных действиях ячейки?

   Ответ. Пункт меню окрашен в красный цвет в качестве предупреждения для пользователя.

6. Когда выполняется вызов методов `PushAsync` и `PopAsync` в приложении .NET MAUI?

   Ответ. Чтобы обеспечить переход между экранами со встроенной поддержкой возврата к предыдущему экрану, при первом запуске приложения оберните первый экран оператором `NavigationPage`:

```
MainPage = new NavigationPage(new CustomersListPage());
```

Чтобы перейти к следующему экрану, добавьте следующую страницу в объекте `Navigation`:

```
await Navigation.PushAsync(new CustomerDetailPage(c));
```

Чтобы вернуться к предыдущему экрану, откройте страницу из объекта `Navigation`, как показано ниже:

```
await Navigation.PopAsync();
```

7. В чем разница между `Margin` и `Padding` для такого элемента, как `Button`?

   Ответ. Разница заключается в том, что `Margin` находится вне `Border`, а `Padding` — внутри `Border`.

8. Как обработчики событий прикрепляются к объекту с помощью XAML?

   Ответ. С помощью присвоения атрибуту имени события имена метода в классе кода программной части:

```
<Button Clicked="SaveButton_Clicked">
```

9. Что делают стили XAML?

   Ответ. Позволяют установить одно или несколько свойств.

10. Где можно определить ресурсы?

    Ответ. Вы можете определить ресурсы в любом элементе в зависимости от того, где хотите использовать их:

    - для совместного использования ресурсов во всем приложении определите ресурсы в элементе `<Application. Resources>`;
    - для совместного использования ресурсов только в пределах страницы определите ресурсы в ее элементе `<Page.Resources>`;
    - для совместного использования ресурсов только в одном элементе, таком как кнопка, определите ресурсы в его элементе `<Button.Resources>`.

Глава 20. 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, лучше всего подойдет для асимметричного шифрования?

   Ответ. Алгоритм RSA.

3. Что такое радужная атака?

   Ответ. Радужная атака использует таблицу предварительно рассчитанных хешей паролей. Когда база данных хешей паролей украдена, злоумышленник может быстро сравнить хеши радужной таблицы и определить оригинальные пароли.

4. При использовании алгоритмов шифрования лучше применять блоки большого или малого размера?

   Ответ. Малого.

5. Что такое криптографическое хеширование данных?

   Ответ. Криптографический хеш — это вывод фиксированного размера, который получается в результате ввода произвольного размера, обрабатываемого хеш-функцией. Хеш-функции — односторонние, то есть единственный способ воссоздать исходные данные — это перебрать все возможные входные данные и сравнить результаты.

6. Что такое криптографическая подпись?

   Ответ. Криптографическая подпись — это значение, добавляемое к цифровому документу для подтверждения его подлинности. Действительная подпись сообщает получателю, что документ был создан известным отправителем и не был изменен.

7. В чем разница между симметричным и асимметричным шифрованием?

   Ответ. Симметричное шифрование использует секретный общий ключ для шифрования и дешифрования. Асимметричное шифрование использует открытый ключ для шифрования и закрытый ключ для дешифрования.

8. Что означает RSA?

   Ответ. Rivest — Shamir — Adleman (Ривест — Шамир — Адлеман) — фамилии трех создателей криптографического алгоритма, созданного в 1978 году.

9. Почему пароли должны быть засолены перед сохранением?

   Ответ. Чтобы замедлить радужные словарные атаки.

10. SHA-1 — это алгоритм хеширования, разработанный Национальным агентством безопасности США. Почему его не следует использовать?

    Ответ. Алгоритм SHA-1 больше не является безопасным. Все современные браузеры перестали принимать SSL-сертификаты алгоритма SHA-1.

Марк Прайс

C# 10 и .NET 6. Современная кросс-платформенная разработка

Перевел с английского С. Черников

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректор	*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: 12.2022.

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 06.10.22. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 68,370. Тираж 500. Заказ 0000.

Харрисон Ферроне

Изучаем C# через разработку игр на Unity. 5-е издание

Изучение C# через разработку игр на Unity — популярный способ ускоренного освоения мощного и универсального языка программирования, используемого для решения прикладных задач в широком спектре предметных областей. Эта книга дает вам возможность с нуля изучить программирование на C# без зубодробительных терминов и непонятной логики программирования, причем процесс изучения сопровождается созданием простой игры на Unity.

В пятом издании изложены последние версии всех современных функций C# на примерах из игрового движка Unity, а также добавлена новая глава о промежуточных типах коллекций. Вы начнете с основ программирования и языка C#, узнаете основные концепции программирования на C#, включая переменные, классы и объектно-ориентированное программирование. Освоив программирование на C#, переключитесь непосредственно на разработку игр на Unity и узнаете, как написать сценарий простой игры на C#. На протяжении всей книги описываются лучшие практики программирования, которые помогут вам вывести свои навыки Unity и C# на новый уровень. В результате вы сможете использовать язык C # для создания собственных реальных проектов игр на Unity.

КУПИТЬ

Эндрю Стиллмен,
Дженнифер Грин

Head First. Изучаем C#. 4-е изд.

Стиль Head First позволяет сразу приступить к созданию собственного кода на C#, даже если у вас нет никакого опыта программирования. Не нужно тратить время на изучение скучных спецификаций и примеров! Вы освоите необходимый минимум инструментов, и сразу приступите к забавным и интересным программным проектам: от разработки 3D-игры до создания серьезного приложения и работы с данными. Четвертое издание книги было полностью обновлено и переработано, чтобы рассказать о возможностях современных C#, Visual Studio и .NET, оно будет интересно всем, кто изучает язык программирования C#.

Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.

КУПИТЬ