





Microsoft Visual C#

ПОДРОБНОЕ РУКОВОДСТВО

8-е издание



Intermediate



 **ПИТЕР**®

Джон ШАРП



Джон ШАРП

Microsoft Visual C#

ПОДРОБНОЕ РУКОВОДСТВО

8-е издание



Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону · Самара · Минск

2017

ББК 32.973.2-018.1
УДК 004.43
Ш26

Шарп Джон

- Ш26 Microsoft Visual C#. Подробное руководство. 8-е изд. — СПб.: Питер, 2017. — 848 с.: ил. — (Серия «Библиотека программиста»).
ISBN 978-5-496-02372-6

Освойте основы программирования и углубите свои познания, используя новейшую версию C# с Visual Studio 2015. Вы научитесь быстро писать код и создавать проекты, работать с переменными, операторами, выражениями и методами, разрабатывать надежные приложения с обработкой ошибок и исключений, использовать коллекции, создавать запросы LINQ, а кроме того, получите навыки объектно-ориентированного программирования. Книга пригодится разработчикам программного обеспечения, которые только начинают работать с Visual C# или хотят перейти на новую версию ПО, а также всем, кто знает хотя бы один язык программирования. Опыт работы с Microsoft .NET или Visual Studio не требуется.

12+ (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 Microsoft Press. 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.

ISBN 978-1509301041 англ.
ISBN 978-5-496-02372-6

© 2015 by CM Group, Ltd.
© Перевод на русский язык ООО Издательство «Питер», 2017
© Издание на русском языке, оформление ООО Издательство «Питер», 2017
© Серия «Библиотека программиста», 2017

Краткое содержание

Введение.....	17
Часть I. Введение в Microsoft Visual C# и Microsoft Visual Studio 2015	31
Глава 1. Добро пожаловать в C#	32
Глава 2. Работа с переменными, операторами и выражениями	65
Глава 3. Создание методов и применение областей видимости.....	93
Глава 4. Использование инструкций принятия решений	125
Глава 5. Использование инструкций составного присваивания и итераций	147
Глава 6. Обработка ошибок и исключений.....	168
Часть II. Основные сведения об объектной модели C#	195
Глава 7. Создание классов и объектов и управление ими	196
Глава 8. Основные сведения о значениях и ссылках.....	222
Глава 9. Создание типов значений с использованием перечислений и структур	250
Глава 10. Использование массивов	272
Глава 11. Основные сведения о массивах параметров.....	297
Глава 12. Работа с наследованием.....	311
Глава 13. Создание интерфейсов и определение абстрактных классов	337
Глава 14. Использование сборщика мусора и управление ресурсами.....	367

Часть III. Определение расширяемых типов в C#	391
Глава 15. Реализация свойств для доступа к полям	392
Глава 16. Использование индексаторов.....	417
Глава 17. Введение в обобщения.....	435
Глава 18. Использование коллекций	469
Глава 19. Перечисляемые коллекции	496
Глава 20. Отделение логики приложения и обработка событий	512
Глава 21. Запрос данных, находящихся в памяти, с помощью выражений в виде запросов	546
Глава 22. Перегрузка операторов	573
Часть IV. Создание приложений универсальной платформы Windows с использованием C#	597
Глава 23. Повышение производительности путем использования задач	598
Глава 24. Сокращение времени отклика путем выполнения асинхронных операций	649
Глава 25. Реализация пользовательского интерфейса для приложений универсальной платформы Windows	698
Глава 26. Отображение и поиск данных в приложении универсальной платформы Windows	755
Глава 27. Доступ к удаленной базе данных из приложения универсальной платформы Windows	789

Оглавление

Введение.....	17
Для кого предназначена эта книга	18
Кому не следует читать эту книгу	18
Структура книги.....	19
С чего лучше начать изучение книги	19
Условные обозначения и особенности оформления	21
Требования к системе.....	21
Примеры кода.....	22
Установка примеров кода	22
Использование примеров кода.....	23
От издательства	28
Благодарности	29
Об авторе.....	30
Часть I. Введение в Microsoft Visual C# и Microsoft Visual Studio 2015	31
Глава 1. Добро пожаловать в C#	32
Начало программирования в среде Visual Studio 2015.....	32
Создание консольного приложения в Visual Studio 2015.....	33
Написание вашей первой программы	38
Написание кода с использованием Microsoft IntelliSense	38
Сборка и запуск консольного приложения.....	41
Использование пространств имен.....	45
Попытка работы с простым написанием имен.....	46
Создание графического приложения	48
Создание графического приложения в Visual Studio 2015.....	49
Создание пользовательского интерфейса	51
Изучение UWP-приложения.....	57
Добавление кода к графическому приложению.....	61
Выводы.....	64
Краткий справочник	64
Глава 2. Работа с переменными, операторами и выражениями	65
Понятие об инструкциях	65

Использование идентификаторов.....	66
Идентификация ключевых слов	67
Использование переменных.....	68
Правила присваивания имен переменным.....	68
Объявление переменных	69
Работа с простыми типами данных.....	70
Локальные переменные с неприсвоенными значениями.....	71
Отображение значений, относящихся к простым типам данных.....	71
Использование арифметических операторов	77
Операторы и типы.....	78
Исследование арифметических операторов	80
Управление очередностью применения операторов	85
Использование ассоциативности для вычисления выражений.....	86
Ассоциативность и оператор присваивания	87
Увеличение и уменьшение значений переменных на единицу	88
Префиксная и постфиксная формы	89
Объявление неявно типизированных локальных переменных.....	90
Выводы.....	91
Краткий справочник	91
Глава 3. Создание методов и применение областей видимости	93
Создание методов.....	94
Объявление метода.....	94
Возвращение данных из метода	95
Использование метода-выражения.....	96
Вызов методов	98
Определение синтаксиса вызова метода.....	98
Применение области видимости переменных	101
Определение локальной области видимости	101
Определение области видимости внутри класса	102
Перегрузка методов	103
Создание методов.....	104
Реорганизация кода.....	109
Использование необязательных параметров и именованных аргументов	114
Определение необязательных параметров.....	115
Передача именованных аргументов.....	116
Устранение неоднозначностей, связанных с необязательными параметрами и именованными аргументами.....	117
Выводы.....	123
Краткий справочник	123
Глава 4. Использование инструкций принятия решений.....	125
Объявление булевых переменных	126
Использование булевых операторов	126
Основные сведения об операторах равенства и отношения	126
Основные сведения об условных логических операторах.....	127
Короткое замыкание	128

Сводная информация о приоритетности и ассоциативности операторов	129
Использование инструкций if для принятия решений.....	130
Основные сведения о синтаксисе инструкции if.....	131
Использование блоков для объединения инструкций в группы.....	132
Создание каскада из if-инструкций	133
Использование инструкций switch.....	139
Основные сведения о синтаксисе инструкции switch.....	140
Выполнение правил использования инструкции switch.....	141
Выводы.....	145
Краткий справочник	145
Глава 5. Использование инструкций составного присваивания и итераций... 147	
Использование операторов составного присваивания.....	148
Запись инструкций while	149
Запись инструкций for.....	154
Основные сведения об области видимости инструкции for.....	156
Запись инструкций do	157
Выводы.....	166
Краткий справочник	167
Глава 6. Обработка ошибок и исключений 168	
Борьба с ошибками.....	169
Попытка выполнения кода и перехват исключений.....	169
Необработанные исключения.....	171
Использование нескольких обработчиков исключений	173
Обработка нескольких исключений	173
Распространение исключений	179
Использование проверяемой и непроверяемой целочисленной арифметики	181
Запись инструкций checked.....	182
Запись проверяемых выражений.....	183
Выдача исключений	186
Использование блока finally	191
Выводы.....	193
Краткий справочник	194
Часть II. Основные сведения об объектной модели C# 195	
Глава 7. Создание классов и объектов и управление ими 196	
Основные сведения о классификации.....	197
Цель инкапсуляции.....	197
Определение и использование класса	198
Управление доступностью	199
Работа с конструкторами.....	201
Перегрузка конструкторов.....	202
Основные сведения о статических методах и данных.....	212
Создание совместно используемых полей	213

Создание статических полей с использованием ключевого слова const	214
Основные сведения о статических классах.....	214
Инструкции использования статических элементов	215
Безымянные классы	218
Выводы.....	220
Краткий справочник	220
Глава 8. Основные сведения о значениях и ссылках	222
Копирование типов значений переменных и классов.....	222
Основные сведения о пустых значениях и о типах, допускающих их использование	229
Использование типов, допускающих пустые значения.....	231
Основные сведения о свойствах типов, допускающих пустые значения	232
Использование параметров ref и out	233
Создание ref-параметров	234
Создание out-параметров	235
Как организована память компьютера	237
Использование стека и кучи.....	239
Класс System.Object	240
Упаковка.....	241
Распаковка.....	242
Безопасное приведение типов данных.....	244
Оператор is.....	245
Оператор as.....	245
Выводы.....	247
Краткий справочник	248
Глава 9. Создание типов значений с использованием перечислений и структур	250
Работа с перечислениями.....	250
Объявление перечисления.....	251
Использование перечисления.....	251
Выбор литеральных значений перечислений	252
Выбор основного типа перечислений	253
Работа со структурами	256
Объявление структуры	257
Основные сведения о том, чем структуры отличаются от классов.....	259
Объявление переменных структуры	260
Представление об инициализации структуры.....	261
Копирование структурных переменных	266
Выводы.....	270
Краткий справочник	271
Глава 10. Использование массивов	272
Объявление и создание массива	272
Объявление переменных массивов.....	273
Создание экземпляра массива	273

Заполнение и использование массива	274
Создание массива с неявно заданным типом элементов	276
Обращение к отдельным элементам массива	277
Последовательный обход элементов массива	277
Передача массивов в качестве параметров и возвращаемых значений метода.....	279
Копирование массивов.....	281
Использование многомерных массивов.....	282
Создание ступенчатых массивов	283
Выводы.....	295
Краткий справочник	295
Глава 11. Основные сведения о массивах параметров.....	297
Давайте вспомним, что такое перегрузка	297
Использование аргументов в виде массивов.....	298
Объявление массива параметров.....	300
Использование конструкции params object[]	302
Использование params-массива	304
Сравнение массивов параметров с необязательными параметрами	307
Выводы.....	309
Краткий справочник	310
Глава 12. Работа с наследованием.....	311
Что такое наследование?	311
Использование наследования.....	312
Повторное обращение к классу System.Object	314
Вызов конструкторов базового класса.....	315
Присваивание классов.....	316
Объявление новых методов	318
Объявление виртуальных методов	319
Объявление методов с помощью ключевого слова override.....	321
Основные сведения о защищенном доступе.....	323
Основные сведения о методах расширения.....	330
Выводы.....	335
Краткий справочник	335
Глава 13. Создание интерфейсов и определение абстрактных классов.....	337
Основные сведения об интерфейсах	338
Определение интерфейса	339
Реализация интерфейса	339
Ссылка на класс через его интерфейс	341
Работа с несколькими интерфейсами	342
Явная реализация интерфейса	342
Ограничения, накладываемые на интерфейсы	345
Определение и использование интерфейсов.....	345
Абстрактные классы.....	355
Абстрактные методы	357

Запечатанные классы.....	357
Запечатанные методы	358
Реализация и использование абстрактного класса	358
Выводы.....	365
Краткий справочник	366
Глава 14. Использование сборщика мусора и управление ресурсами.....	367
Срок существования объекта.....	368
Создание деструкторов	369
Зачем используется сборщик мусора?.....	371
Как работает сборщик мусора?	372
Рекомендации.....	373
Управление ресурсами	374
Методы высвобождения ресурсов.....	374
Высвобождение ресурсов независимо от выдачи исключений.....	375
Инструкция using и интерфейс IDisposable	376
Вызов метода Dispose из деструктора.....	378
Реализация высвобождения ресурсов независимо от выдачи исключений.....	380
Выводы.....	389
Краткий справочник	389
Часть III. Определение расширяемых типов в C#	391
Глава 15. Реализация свойств для доступа к полям	392
Реализация инкапсуляции путем использования методов.....	393
Что такое свойства?.....	395
Использование свойств.....	397
Свойства только для чтения	398
Свойства только для записи	399
Доступность свойств	399
Основные сведения об ограничениях, накладываемых на свойства.....	400
Объявление свойств интерфейса.....	402
Замена методов свойствами	403
Создание свойств в автоматическом режиме	408
Инициализация объектов путем использования свойств	410
Выводы.....	414
Краткий справочник	415
Глава 16. Использование индексаторов	417
Что такое индексатор?.....	417
Пример без использования индексаторов	418
Тот же пример с использованием индексаторов	420
Основные сведения о методах доступа к индексаторам	422
Сравнение индексаторов и массивов	423
Индексаторы в интерфейсах	425
Использование индексаторов в приложении Windows.....	427

Выводы.....	433
Краткий справочник	434
Глава 17. Введение в обобщения.....	435
Проблемы, связанные с типом объект	435
Решение, использующее обобщения.....	439
Сравнение классов-обобщений и обобщенных классов.....	442
Обобщения и ограничения	442
Создание класса-обобщения	443
Теория двоичных деревьев	443
Создание класса двоичного дерева с использованием обобщений.....	446
Создание методов-обобщений.....	457
Определение метода-обобщения для создания двоичного дерева.....	458
Вариантность и интерфейсы-обобщения	460
Ковариантные интерфейсы	462
Контрвариантные интерфейсы	464
Выводы.....	467
Краткий справочник	467
Глава 18. Использование коллекций.....	469
Что такое классы коллекций?.....	469
Класс коллекций List<T>	471
Класс коллекций LinkedList<T>	474
Класс коллекций Queue<T>	476
Класс коллекций Stack<T>	477
Класс коллекций Dictionary< TKey, TValue >	478
Класс коллекций SortedList< TKey, TValue >	479
Класс коллекций HashSet<T>	481
Использование инициализаторов коллекций.....	482
Методы Find, предикаты и лямбда-выражения	483
Формы лямбда-выражений	486
Сравнение массивов и коллекций.....	488
Применение классов коллекций к игральным картам.....	488
Выводы.....	493
Краткий справочник	494
Глава 19. Перечисляемые коллекции.....	496
Перечисление элементов коллекции	496
Самостоятельная реализация нумератора	498
Реализация интерфейса IEnumerable	503
Реализация нумератора с использованием итератора	505
Простой итератор	506
Определение нумератора для класса Tree< TItem > путем использования итератора	508
Выводы.....	510
Краткий справочник	511

Глава 20. Отделение логики приложения и обработка событий	512
Основные сведения о делегатах	513
Примеры делегатов в библиотеке классов .NET Framework	514
Сценарий автоматизированной фабрики	516
Реализация системы управления фабрикой без использования делегатов.....	517
Реализация системы управления фабрикой с использованием делегата.....	518
Объявление и использование делегатов	521
Лямбда-выражения и делегаты	530
Создание метода-адаптера	530
Включение уведомлений путем использования событий	531
Объявление события.....	532
Подписка на событие	533
Отмена подписки на событие.....	533
Инициирование события.....	533
Основные сведения о событиях пользовательского интерфейса.....	534
Использование событий	536
Выводы.....	543
Краткий справочник	543
Глава 21. Запрос данных, находящихся в памяти, с помощью выражений в виде запросов	546
Что такое LINQ?	547
Использование LINQ в приложении на C#	548
Выбор данных.....	550
Фильтрация данных	553
Упорядочение, группировка и статистическая обработка данных	553
Объединение данных	556
Использование операторов запросов	558
Запрос данных в объектах Tree<TItem>.....	560
LINQ и отложенное вычисление	566
Выводы.....	571
Краткий справочник	571
Глава 22. Перегрузка операторов	573
Общие сведения об операторах	573
Ограничения, накладываемые на операторы	574
Перегруженные операторы.....	575
Создание симметричных операторов	576
Основные сведения о вычислении составного присваивания	579
Объявление операторов инкремента и декремента	580
Операторы сравнения в структурах и классах.....	581
Определение пар операторов	582
Реализация операторов	583
Основные сведения об операторах преобразования.....	590

Предоставление встроенных преобразований	590
Реализация операторов преобразований, определяемых пользователем	591
Повторное обращение к теме симметричных операторов	592
Написание операторов преобразований	593
Выводы.....	596
Краткий справочник	596
Часть IV. Создание приложений универсальной платформы Windows с использованием C#	597
Глава 23. Повышение производительности путем использования задач	598
Зачем нужна многозадачность, достигаемая параллельной обработкой данных?	599
Взлет многоядерных процессоров.....	599
Реализация многозадачности с помощью Microsoft .NET Framework.....	601
Задачи, потоки и ThreadPool.....	602
Создание задач, их выполнение и управление ими.....	604
Использование класса Task для реализации выполнения программы в параллельном режиме	607
Абстрагирование задач путем использования класса Parallel.....	621
Когда не нужно использовать класс Parallel.....	627
Отмена задач и обработка исключений.....	629
Механизмы согласованной отмены	629
Использование продолжений с отмененными и давшими сбой задачами.....	645
Выводы.....	646
Краткий справочник	646
Глава 24. Сокращение времени отклика путем выполнения асинхронных операций	649
Реализация асинхронных методов.....	650
Определение асинхронных методов – суть проблемы.....	651
Определение асинхронных методов: Решение	655
Определение асинхронных методов, возвращающих значения	661
Трудности, связанные с асинхронными методами	662
Асинхронные методы и API-интерфейсы Windows Runtime	664
Использование PLINQ для распараллеливания декларативного доступа к данным	668
Использование PLINQ для повышения производительности при переборе элементов коллекции.....	668
Отмена PLINQ-запроса.....	674
Синхронизация одновременного доступа к данным	674
Блокировка данных.....	678
Примитивы синхронизации для координации выполнения задач	679
Отмена синхронизации.....	682
Классы коллекций, к которым осуществляется одновременный доступ	683

Использование коллекции с одновременным доступом и блокировка с целью реализации безопасного доступа к данным в многопоточной среде	684
Выводы.....	695
Краткий справочник	695
Глава 25. Реализация пользовательского интерфейса для приложений универсальной платформы Windows.....	698
Характерные особенности приложения универсальной платформы Windows.....	700
Использование шаблона пустого приложения для создания приложения универсальной платформы Windows.....	703
Реализация масштабируемого пользовательского интерфейса	706
Реализация табличной разметки путем использования элемента управления Grid.....	723
Подстраивание разметки с помощью Диспетчера визуальных состояний.....	733
Применение стилей к пользовательскому интерфейсу	742
Выводы.....	753
Краткий справочник	754
Глава 26. Отображение и поиск данных в приложении универсальной платформы Windows	755
Реализация шаблона Model-View-ViewModel.....	756
Отображение данных путем использования привязки данных.....	757
Изменение данных путем использования привязки данных.....	763
Использование привязки данных к элементам управления типа ComboBox	769
Создание модели представления (ViewModel)	771
Добавление команд к модели представления.....	776
Выводы.....	787
Краткий справочник	787
Глава 27. Доступ к удаленной базе данных из приложения универсальной платформы Windows	789
Извлечение данных из базы данных	790
Создание сервера Azure SQL Database и установка учебной базы данных AdventureWorks	792
Создание entity-модели	797
Создание и использование веб-сервиса REST.....	807
Вставка, обновление и удаление данных через REST веб-сервис	824
Выдача отчета об ошибках и обновление пользовательского интерфейса....	835
Выводы.....	844
Краткий справочник	845

Введение

Microsoft Visual C# представляет собой весьма эффективный и в то же время простой язык, предназначенный преимущественно для разработчиков, создающих сборки приложений в среде Microsoft .NET Framework. Visual C# унаследовал множество лучших свойств от C++ и Microsoft Visual Basic, но при этом его разработчики постарались избавиться от различных несоответствий и анахронизмов, в результате чего появился более понятный и логичный язык. Версия C# 1.0 дебютировала в 2001 году. С появлением C# 2.0 вместе с Visual Studio 2005 в язык были добавлены несколько важных новых свойств, включая обобщения, итераторы и безымянные методы. В версию C# 3.0, выпущенную вместе с Visual Studio 2008, были добавлены методы расширений, лямбда-выражения и, что наиболее важно, встроенное в язык расширение, позволяющее выполнять запросы к данным, — Language-Integrated Query (LINQ). Версия C# 4.0 вышла в 2010 году и обеспечила дальнейшие усовершенствования, улучшающие совместимость с другими языками и технологиями. В число новых свойств входила поддержка именованных и необязательных аргументов и динамических типов, свидетельствующая о том, что среди выполнения, используемой языком, следует реализовывать для объекта позднюю привязку. Важным дополнением к среде .NET Framework, которое вышло параллельно с выпуском версии C# 4.0, стали классы и типы, составляющие библиотеку параллельно выполняемых задач (Task Parallel Library (TPL)). С помощью TPL можно создавать приложения, обладающие высокой степенью масштабируемости и способные по максимуму воспользоваться преимуществами, предоставляемыми многоядерными процессорами. В версию C# 5.0 была добавлена собственная поддержка асинхронной обработки данных на основе применения задач, реализуемая посредством модификатора методов `async` и оператора `await`. Версия C# 6.0 явилась дополняющим обновлением, включающим свойства, предназначенные для облегчения жизни разработчиков. К их числу относятся такие дополнения, как строковая интерполяция (теперь о выражении `String.Format` можно забыть навсегда!), усовершенствованные способы реализации свойств, методы, чье тело представлено в виде выражения, и многие другие улучшения. Все они рассматриваются в данной книге.

Еще одним важным событием для компании Microsoft стал выпуск Windows 10. Эта новая версия Windows сочетает в себе лучшие и наиболее полюбившиеся пользователям аспекты предыдущих версий операционной системы

и поддержку приложений с высокой степенью интерактивности, способных совместно использовать данные и взаимодействовать друг с другом, а также подключаться к сервисам, развернутым в среде облачных вычислений. Ключевым понятием в Windows 10 являются приложения универсальной платформы Windows (Universal Windows Platform (UWP)), разработанные для выполнения на любом устройстве с Windows 10, имеющем ограниченные ресурсы, будь то полноценная настольная система, ноутбук, планшетный компьютер, смартфон или даже устройство так называемого Интернета вещей (Internet of Things (IoT)). После освоения основных функциональных возможностей C# важная роль будет отводиться приобретению навыков создания приложений, которые могут работать на всех этих платформах.

Облачные вычисления превратились в такой важный элемент архитектуры многих систем, от широко масштабируемых приложений для промышленных предприятий до мобильных приложений, работающих на пользовательских смартфонах, что я считал необходимым в заключительной главе данной книги сконцентрироваться именно на этом аспекте разработки.

Среда разработки, предоставляемая Visual Studio 2015, существенно упрощает использование функциональных возможностей языка, а многочисленные новые мастера и усовершенствованные механизмы, включенные в последнюю версию, способны значительно повысить производительность труда разработчика. Надеюсь, что работа с книгой принесет вам столько же удовольствия, сколько я получил при ее написании!

Для кого предназначена эта книга

Предполагается, что читателем книги будет разработчик, желающий изучить основы программирования на C# с использованием среды Visual Studio 2015 и .NET Framework версии 4.6. Прочитав книгу, вы получите полное представление о языке C# и сможете воспользоваться им для создания адаптивных и широко масштабируемых приложений, способных работать под управлением операционной системы Windows 10.

Кому не следует читать эту книгу

Эта книга предназначена для разработчиков, ранее пользовавшихся языком C#, и не рассчитана на абсолютных новичков в вопросах программирования. Поэтому ее материал основан преимущественно на применении языка C#. Эта книга не задумывалась в качестве подробного описания множества технологий, доступных для создания приложений корпоративного уровня, работающих под

управлением Windows, таких как ADO.NET, ASP.NET, Windows Communication Foundation или Windows Workflow Foundation. Если вам нужны дополнительные сведения обо всех этих технологиях, обратитесь к другим изданиям, выпущенным в Microsoft Press.

Структура книги

Книга разбита на четыре части.

- ❑ Часть I «Введение в Microsoft Visual C# и Microsoft Visual Studio 2015» представляет собой введение в основной синтаксис языка C# и приемы работы со средой программирования Visual Studio.
- ❑ Часть II «Основные сведения об объектной модели C#» рассматривает подробности создания новых типов в C# и управления ими, а также способы управления ресурсами, на которые ссылаются эти типы.
- ❑ Часть III «Определение расширяемых типов в C#» включает более подробное описание элементов, предоставляемых языком C# для создания типов, пригодных для многократного использования несколькими приложениями.
- ❑ Часть IV «Создание приложений универсальной платформы Windows с использованием C#» включает описание универсальной модели программирования Windows 10 и порядка использования C# с целью создания интерактивных приложений, предназначенных для этой новой модели.

С чего лучше начать изучение книги

Данная книга написана с целью помочь вам приобрести навыки в ряде важных областей. Она может стать полезной как для начинающих программистов, так и для тех, кто собирается перейти на C# с других языков программирования, например C, C++, Java или Visual Basic. Чтобы узнать, с чего лучше начать изучение материалов книги, обратитесь к следующей таблице.

Если вы	Выполните следующие действия
Новичок в области объектно-ориентированного программирования	<ol style="list-style-type: none">1. Установите на свой компьютер файлы с примерами приложений согласно описанию в следующем далее разделе «Примеры кода».2. Проработайте последовательно главы первых трех частей книги.3. Проработайте часть IV сообразно уровню вашего опыта и интересов

Если вы	Выполните следующие действия
Знакомы с таким языком процедурного программирования, как C, но новичок в программировании на C#	<ol style="list-style-type: none">Установите на свой компьютер файлы с примерами приложений согласно описанию в следующем далее разделе «Примеры кода».Бегло просмотрите первые пять глав, чтобы получить представление о C# и Visual Studio 2015, а затем сконцентрируйтесь на изучении глав с 6-й по 22-ю.Проработайте часть IV сообразно уровню вашего опыта и интересов
Перешли с такого объектно-ориентированного языка программирования, как C++ или Java	<ol style="list-style-type: none">Установите на свой компьютер файлы с примерами приложений согласно описанию в следующем далее разделе «Примеры кода».Бегло просмотрите первые семь глав, чтобы получить представление о C# и Visual Studio 2015, а затем сконцентрируйтесь на изучении глав с 8-й по 22-ю.За сведениями по созданию приложений универсальной платформы Windows обратитесь к части IV данной книги
Переключаетесь с Visual Basic на C#	<ol style="list-style-type: none">Установите на свой компьютер файлы с примерами приложений согласно описанию в следующем далее разделе «Примеры кода».Проработайте последовательно главы первых трех частей книги.За сведениями по созданию приложений универсальной платформы Windows обратитесь к части IV данной книги.Прочитайте разделы «Краткий справочник» в конце глав, чтобы получить сведения о специфике программных конструкций C# и Visual Studio 2015
Обращаетесь к книге уже после проработки всех упражнений	<ol style="list-style-type: none">Для поиска информации по конкретной теме воспользуйтесь предметным указателем или оглавлением.Прочитайте раздел «Краткий справочник» в конце каждой главы с целью поиска обобщающего обзора синтаксиса и технологий, представленных в главе

Многие главы книги включают практические примеры, позволяющие проверить на деле только что изученные концепции. Но на каком бы разделе вы ни сосредоточились, следует загрузить и установить на свою систему примеры приложений.

Условные обозначения и особенности оформления

Эта книга предоставляет информацию в соответствии с соглашениями, которые призваны сделать информацию удобной и простой для усвоения.

- ❑ Каждое упражнение состоит из серий задач, зачастую разбитых на отдельные абзацы, с описанием тех действий, которые вам следует предпринять для его выполнения.
- ❑ Во врезках с пометкой ПРИМЕЧАНИЕ предоставляется дополнительная информация или альтернативные методы для успешного завершения того или иного этапа упражнения.
- ❑ Текст, который вам следует набрать (помимо блоков кода), дан жирным шрифтом.
- ❑ Знак «плюс» (+) между двумя названиями клавиш означает, что клавиши следует нажимать одновременно. Например, «Нажмите Alt+Tab» означает, что, удерживая в нажатом состоянии клавишу Alt, следует нажать клавишу Tab.

Требования к системе

Для выполнения практических упражнений, предлагаемых в данной книге, вам понадобятся следующее оборудование и программные средства:

- ❑ Windows 10 профессиональная редакция (или выше);
- ❑ редакция Visual Studio Community 2015, Visual Studio Professional 2015 или Visual Studio Enterprise 2015.



ВНИМАНИЕ Вам нужно с помощью Visual Studio 2015 установить инструментарий разработчика приложений под Windows 10.

- ❑ компьютер, имеющий процессор с тактовой частотой не ниже 1,6 ГГц (рекомендуется 2 ГГц);
- ❑ 1 Гбайт (на 32-разрядной системе) или 2 Гбайт (на 64-разрядной системе) оперативной памяти (дополнительно 512 Мбайт памяти при запуске на виртуальной машине);
- ❑ 10 Гбайт доступного дискового пространства;
- ❑ жесткий диск с частотой вращения шпинделя 5400 об./мин;

- ❑ видеокарта, совместимая с технологией DirectX 9 при разрешении дисплея не ниже 1024 × 768;
- ❑ DVD-привод (если установка Visual Studio будет выполняться с DVD-диска);
- ❑ интернет-подключение для загрузки программных средств или примеров к главам.

В зависимости от используемой конфигурации Windows для установки и настройки Visual Studio 2015 вам могут понадобиться права администратора.

Для создания и запуска UWP-приложений вам также потребуется включить на своем компьютере режим разработчика. Подробные сведения о необходимых для этого действиях можно найти в статье «Включение устройства для разработки» по адресу <https://msdn.microsoft.com/library/windows/apps/dn706236.aspx>.

Примеры кода

В большинстве глав данной книги имеются упражнения, с помощью которых можно в интерактивном режиме получить практические навыки использования только что изученного материала основного текста. Все учебные проекты можно загрузить в обоих форматах, стартовом и финишном, с веб-страницы <http://aka.ms/sharp8e/companioncontent>.



ПРИМЕЧАНИЕ В дополнение к примерам кода на вашей системе должна быть установлена среда Visual Studio 2015. По мере доступности следует также установить самые последние обновления для Windows и Visual Studio.

Установка примеров кода

Для установки примеров кода на свой компьютер, чтобы их можно было использовать с упражнениями, рассматриваемыми в данной книге, выполните следующие действия.

1. Разархивируйте файл CSharpSBS.zip, загруженный с веб-сайта книги, извлекая из него файлы в свою папку документов.
2. Если будет предложено, просмотрите соглашение о лицензии конечного пользователя. Если вас все устроит, выберите пункт Accept (Принимаю), а затем щелкните на кнопке Next (Далее).



ПРИМЕЧАНИЕ Если лицензионное соглашение не появится, вы можете обратиться за ним с той же веб-страницы, которая использовалась для загрузки файла CSharpSBS.zip.

Использование примеров кода

Когда и как используется тот или иной пример кода, объясняется в каждой главе. Когда приходит время использовать пример кода, в книге дается перечень инструкций, объясняющих порядок открытия файлов.



ВНИМАНИЕ Многие из примеров кода зависят от пакетов NuGet, не включенных в код. Эти пакеты загружаются автоматически при первой сборке проекта. В результате этой особенности, если открыть проект и изучить код перед выполнением сборки, Visual Studio может отрапортовать о большом количестве ошибок, возникших из-за ссылок, не нашедших разрешения. Сборка проекта заставит среду найти нужное разрешение этих ссылок, и ошибки должны исчезнуть.

Для любителей подробностей приводится перечень учебных проектов и решений Visual Studio 2015, сгруппированных в папки, в которых их можно найти. Во многих случаях упражнения предоставляют начальные файлы и завершенные версии одних и тех же проектов, которые могут использоваться в качестве эталона. Завершенные версии проектов для каждой главы сохранены в папках с названием «Complete».

Проект или решение	Описание
Chapter 1	
TextHello	Этот проект позволяет приступить к работе с языком. В нем представлено поэтапное создание простой программы, выводящей на экран текстовое приветствие
Hello	Этот проект открывает окно, приглашающее пользователя ввести свое имя, после чего выводит на экран приветствие
Chapter 2	
PrimitiveDataTypes	Этот проект показывает, как объявляются переменные путем использования каждого из элементарных типов, как этим переменным присваиваются значения и как их значения отображаются в окне
MathsOperators	Эта программа показывает применение арифметических операторов (+, -, *, /, %)
Chapter 3	
Methods	В этом проекте пересматривается код проекта MathsOperators и изучается порядок использования методов для структурирования кода
DailyRate	Этот проект проводит вас через написание собственных методов, запуск методов и пошаговое выполнение кода при вызове метода с использованием отладчика Visual Studio 2015

Проект или решение	Описание
DailyRate Using Optional Parameters	Этот проект показывает, как определить метод, получающий необязательные параметры, и вызвать этот метод с использованием именованных аргументов
Chapter 4	
Selection	Этот проект показывает, как для реализации сложной логики, такой как сравнение равенства двух дат, используется цепочка инструкций if
SwitchStatement	Эта простая программа использует инструкцию switch для преобразования символов в их XML-представления
Chapter 5	
WhileStatement	Этот проект демонстрирует работу инструкции while , построчночитывающей содержимое исходного файла и выводящей каждую строку на экран в текстовом поле формы
DoStatement	Этот проект используется для перевода десятичного числа в его восьмеричное представление
Chapter 6	
MathsOperators	Этот проект повторно обращается к проекту MathsOperators из главы 2 и показывает, как различные необрабатываемые исключения могут привести к аварийному завершению программы. Затем надежность приложения повышается с помощью использования конструкций с ключевыми словами try и catch , позволяя исключить последующие аварийные завершения работы этого приложения
Chapter 7	
Classes	Этот проект охватывает основы определения ваших собственных классов, комплектуемых открытыми конструкторами, методами и закрытыми полями. Он также показывает, как с помощью ключевого слова new создаются экземпляры класса и как определяются статические методы и поля
Chapter 8	
Parameters	Эта программа исследует разницу между параметрами значений и параметрами ссылок. Она показывает порядок использования ключевых слов ref и out
Chapter 9	
StructsAndEnums	Этот проект определяет с помощью ключевого слова struct тип в виде структуры, предназначенный для представления календарной даты

Проект или решение	Описание
Chapter 10	
Cards	Этот проект показывает использование массива для моделирования сдачи карт на руки при карточной игре
Chapter 11	
ParamsArray	Этот проект демонстрирует порядок использования ключевого слова params для создания простого метода, способного принимать любое количество int -аргументов
Chapter 12	
Vehicles	Этот проект создает простую иерархию классов транспортных средств путем использования наследования. Он также показывает порядок определения виртуального метода
ExtensionMethod	Этот проект показывает, как создается метод расширения для int -типа, предоставляющий метод, который преобразует целое число по основанию 10 в число с другим основанием системы счисления
Chapter 13	
Drawing	Этот проект реализует на практике использование части графического пакета рисования. Для определения методов, предоставляемых и реализуемых классами рисуемых фигур, в проекте используются интерфейсы
Drawing Using Interfaces	Этот проект служит отправным пунктом для расширения проекта Drawing с целью помещения общих функций объектов, представляющих рисуемые фигуры, в абстрактные классы
Chapter 14	
GarbageCollectionDemo	Этот проект показывает порядок реализации высвобождения ресурсов независимо от выдачи исключений путем использования метода Dispose
Chapter 15	
Drawing Using Properties	Этот проект расширяет приложение в проекте Drawing , разработанное в главе 13 с целью инкапсуляции данных в классе путем использования свойств
AutomaticProperties	Этот проект показывает порядок создания свойств для класса в автоматическом режиме и порядок их использования для инициализации экземпляров класса

Проект или решение	Описание
Chapter 16	
Indexers	Этот проект использует два индексатора: один для поиска телефонного номера сотрудника, когда задано его имя, а второй — для поиска имени сотрудника, когда задан его номер телефона
Chapter 17	
BinaryTree	Это решение показывает, как обобщения используются для создания структур, не зависящих от используемых типов и способных содержать элементы любого типа
IteratorBinaryTree	Это решение использует итератор для генерирования нумератора для класса-обобщения <code>Tree</code>
Chapter 18	
Cards	Этот проект обновляет код из главы 10, чтобы показать использование коллекций для моделирования сдачи карт на руки при карточной игре
Chapter 19	
BinaryTree	Этот проект показывает порядок реализации интерфейса-обобщения <code>IEnumerator<T></code> с целью создания нумератора для класса-обобщения <code>Tree</code>
IteratorBinaryTree	Это решение использует итератор с целью создания нумератора для класса-обобщения <code>Tree</code>
Chapter 20	
Delegates	Этот проект показывает, как отвязать метод от логики приложения, вызывающей его с использованием делегата. Затем проект расширяется, чтобы показать, как использовать событие, чтобы предупредить объект о важном произшествии, и как перехватить событие и выполнить любую требуемую обработку
Chapter 21	
QueryBinaryTree	Этот проект показывает порядок использования LINQ-запросов для извлечения данных из объекта двоичного дерева
Chapter 22	
ComplexNumbers	Этот проект определяет новый тип, моделирующий комплексные числа и реализующий для него самые распространенные операторы

Проект или решение	Описание
Chapter 23	
GraphDemo	Этот проект генерирует сложную графику и выводит ее на экран в форме, принадлежащей UWP-приложению. Для выполнения вычислений в нем используется один поток
Parallel GraphDemo	Эта версия проекта GraphDemo использует класс Parallel для абстрагирования процесса создания задач и управления ими
GraphDemo With Cancellation	Этот проект показывает порядок реализации отмены для остановки задач до их завершения управляемым способом
ParallelLoop	Это приложение представляет пример, показывающий, когда не следует использовать класс Parallel для создания и запуска задач
Chapter 24	
GraphDemo	Это версия проекта GraphDemo из главы 23, которая использует ключевое слово async и оператор await для выполнения вычислений, генерирующих данные графического рисунка в асинхронном режиме
PLINQ	Этот проект показывает ряд примеров использования PLINQ для запроса данных с использованием одновременно выполняемых задач
CalculatePI	Этот проект использует для вычисления приблизительных значений числа π алгоритм статистической выборки. В нем используются одновременно выполняемые задачи
Chapter 25	
Customers	Этот проект реализует масштабируемый пользовательский интерфейс, который адаптируется к различным положениям устройства и его форм-факторам. Применяется XAML-стилизация для замены шрифтов и фонового изображения, отображаемых приложением
Chapter 26	
DataBinding	Это версия проекта Customers , использующая привязку данных для отображения сведений о клиентах, извлеченных из источника данных в пользовательском интерфейсе. В ней также показывается порядок реализации интерфейса INotifyPropertyChanged , позволяющего пользовательскому интерфейсу обновлять сведения о клиентах и отправлять эти изменения обратно в адрес источника данных

Проект или решение	Описание
ViewModel	Эта версия проекта <i>Customers</i> отделяет пользовательский интерфейс от логики, обращающейся к источнику данных, путем реализации схемы Model — View — ViewModel (модель — представление — модель представления)
Chapter 27	
Web Service	Это решение включает веб-приложение, предоставляющее веб-сервис ASP.NET Web API, который используется приложением <i>Customers</i> для извлечения сведений о клиентах из базы данных SQL Server. Для доступа к базе данных веб-сервис использует entity-модель, созданную средой Entity Framework

От издательства

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

Мы будем рады узнать ваше мнение!

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

Раздел, посвященный поиску на компьютере данных с использованием Windows 10 системы голосового ввода Кортана, в русском переводе исключен из главы 26. Эта система в русскоязычной версии Windows 10 не реализована и ее включение в операционной системе заблокировано.

Благодарности

Несмотря на то что на обложке книги стоит мое имя, авторство таких проектов, как этот, невозможно приписывать лишь одному человеку. Я хочу выразить благодарность тем людям, которые всецело поддерживали меня и оказывали помощь на протяжении всей работы над книгой.

В первую очередь спасибо Девону Масгрейву (Devon Musgrave) из Microsoft Press, который вывел меня из оцепенения. (Я был сильно занят написанием материалов для Microsoft Patterns & Practices, но сумел взять творческий отпуск для работы над этим изданием книги.) Он настаивал, уговаривал и, убедившись, что я в курсе скорого выхода Windows 10 и Visual Studio 2015, составил договор, а потом удостоверился, что я его в конце концов подписал с согласованными сроками поставки материала!

Далее хочу поблагодарить Джейсона Ли (Jason Lee), моего бывшего подчиненного, а теперь начальника в Content Master (это весьма запутанная история, но, похоже, он разыскал несколько интересных фотоснимков, на которых я валял дурака в рабочее время). Он взвалил на себя нелегкий начальный труд по созданию новых скриншотов и проверке обновлений и корректировок кода первых двадцати или около того глав этого издания. Если обнаружатся ошибки, то мне хотелось бы указать пальцем на него, но, разумеется, вся ответственность за любые недочеты, пропущенные мною при просмотре материала, ложится только на меня.

Я должен также поблагодарить Марка Янга (Marc Young) за то, что ему пришлось заниматься весьма утомительным изучением моего кода с прицелом на определение его шансов быть откомпилированным и запущенным на выполнение. Его совет был исключительно полезен.

Разумеется, как и многие другие программисты, я мог разобраться в технологии, но выразить ее суть в прозе у меня получалось не настолько свободно и понятно, как требовалось. В связи с этим хочу выразить благодарность Джону Пирсу за правку грамматики, устранение ошибок в правописании и в целом за придание моему материалу более понятного вида.

И, наконец, я должен поблагодарить мою терпеливую жену Диану, которая полагала, что в процессе работы над книгой я медленно схожу с ума (возможно, так оно и было).

Ну и в завершение спасибо моей дочери Франческе, которая бы страшно расстроилась, если бы я ее здесь не упомянул. Она все еще живет в родительском доме, хотя уже выросла, и теперь трудится разработчиком программных средств в одной из компаний в графстве Глостершир (название компании упоминать не буду, потому что они мне ничего «за так» не предлагали).

Об авторе

Джон Шарп является главным специалистом CM Group Ltd, компании, которая занимается разработкой программных средств и оказывает консультационные услуги. Шарп является специалистом по обоим этим направлениям. Он автор многочисленных публикаций и преподаватель с 30-летним стажем. Программировал на Паскале для операционной системы CP/M, разрабатывал приложения на C/Oracle для разнообразных версий операционной системы UNIX, разрабатывал распределенные приложения, написанные на C# и JavaScript, и, наконец, приложения под Windows 10 и Microsoft Azure. Джон — крупный специалист по созданию приложений с использованием среды Microsoft .NET Framework, а также автор книги «Windows Communication Foundation 4 Step By Step» (издательство Microsoft Press).

Часть I

Введение в Microsoft Visual C# и Microsoft Visual Studio 2015

Во вводной части книги рассматриваются основы языка C# и показывается, как приступить к созданию приложений с помощью Visual Studio 2015.

В части I вы узнаете, как создавать в Visual Studio новые проекты и как объявлять переменные, использовать операторы для создания переменных, вызывать методы и записывать множество инструкций, требующихся при реализации программ на C#. Вы также узнаете, как обрабатывать исключения и пользоваться отладчиком Visual Studio для пошагового выполнения кода и выявления проблем, мешающих приложению правильно работать.

1 Добро пожаловать в C#

Прочитав эту главу, вы научитесь:

- пользоваться средой программирования Microsoft Visual Studio 2015;
- создавать консольные приложения на C#;
- объяснять назначение пространств имен;
- создавать простые графические приложения на C#.

В этой главе излагается введение в Visual Studio 2015, рассматриваются среда программирования и набор инструментальных средств, которые были разработаны для создания приложений под Microsoft Windows. Visual Studio 2015 — идеальное средство написания кода на C#, которое предоставляет множество функциональных возможностей, доступных для изучения в процессе чтения книги. В этой главе среда Visual Studio 2015 будет использоваться для создания простых приложений на C#, позволяющих встать на путь создания полноценных решений для Windows.

Начало программирования в среде Visual Studio 2015

Visual Studio 2015 является богатой инструментальными средствами средой программирования, предоставляющей функциональные возможности, необходимые для создания больших или малых C#-проектов, запускаемых под управлением Windows. В этой среде можно даже создавать проекты, которые незаметно для пользователя объединяют модули, написанные на различных языках программирования, таких как C++, Visual Basic и F#. В первом упражнении вы откроете среду программирования Visual Studio 2015 и узнаете, как создавать консольные приложения.



ПРИМЕЧАНИЕ Консольное приложение не предоставляет графический интерфейс пользователя (graphical user interface (GUI)) и запускается в окне командной строки.

Создание консольного приложения в Visual Studio 2015

Щелкните на панели задач Windows на кнопке Пуск (Start), наберите Visual Studio 2015, а затем нажмите Ввод. Будет запущена среда Visual Studio 2015, которая выведет на экран стартовую страницу, похожую на ту, что показана на рис. 1.1. (В зависимости от используемого вами варианта Visual Studio 2015 у вашей стартовой страницы может быть несколько иной внешний вид.)

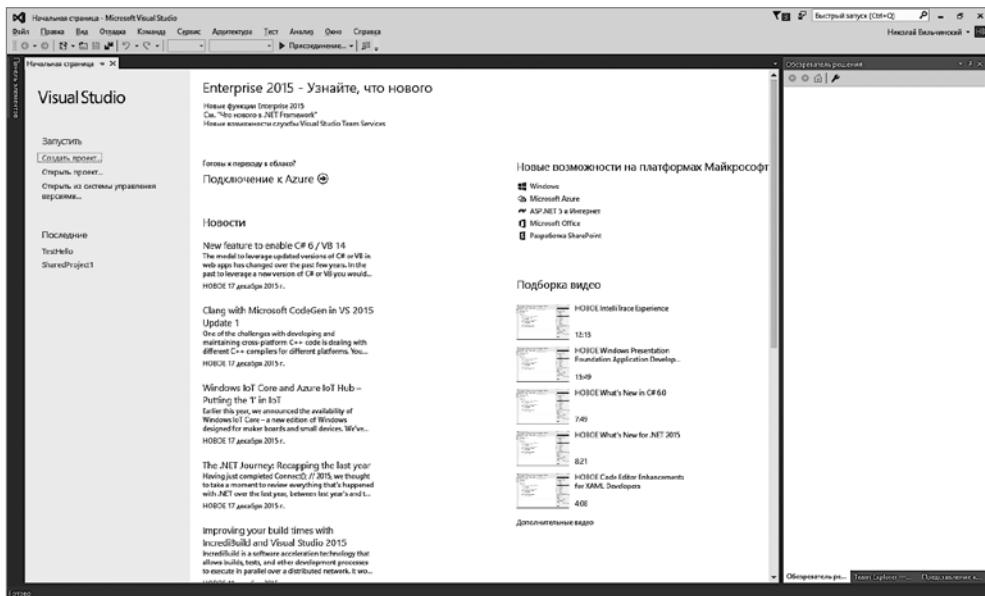


Рис. 1.1



ПРИМЕЧАНИЕ При первом запуске Visual Studio 2015 может выводиться диалоговое окно, приглашающее выбрать исходные установки среды разработки. Visual Studio 2015 способна перенастраиваться в соответствии с предпочтаемым вами языком программирования. Для выбранного языка будут сделаны настройки по умолчанию для различных диалоговых окон и инструментов в интегрированной среде разработки (integrated development environment (IDE)). В первичные настройки разработки (Development Settings) нужно выбрать Visual C#, а затем щелкнуть на пункте Start Visual Studio. После небольшой паузы появится IDE-среда Visual Studio 2015.

Выберите в меню Файл (File) пункт Создать (New), а затем щелкните на пункте Проект (Project). Откроется диалоговое окно Создание проекта (New Project).

В нем показан перечень шаблонов, которыми можно будет воспользоваться, приступая к созданию приложения. В диалоговом окне перечисляются категории шаблонов, соответствующие используемому языку программирования, а также типы приложений.

Раскройте на левой панели узел Установленные (Installed), затем узел Шаблоны (Templates) и щелкните на пункте Visual C#. На средней панели в поле со списком выберите .NET Framework 4.6 и щелкните на пункте Консольное приложение (Console Application) (рис. 1.2).

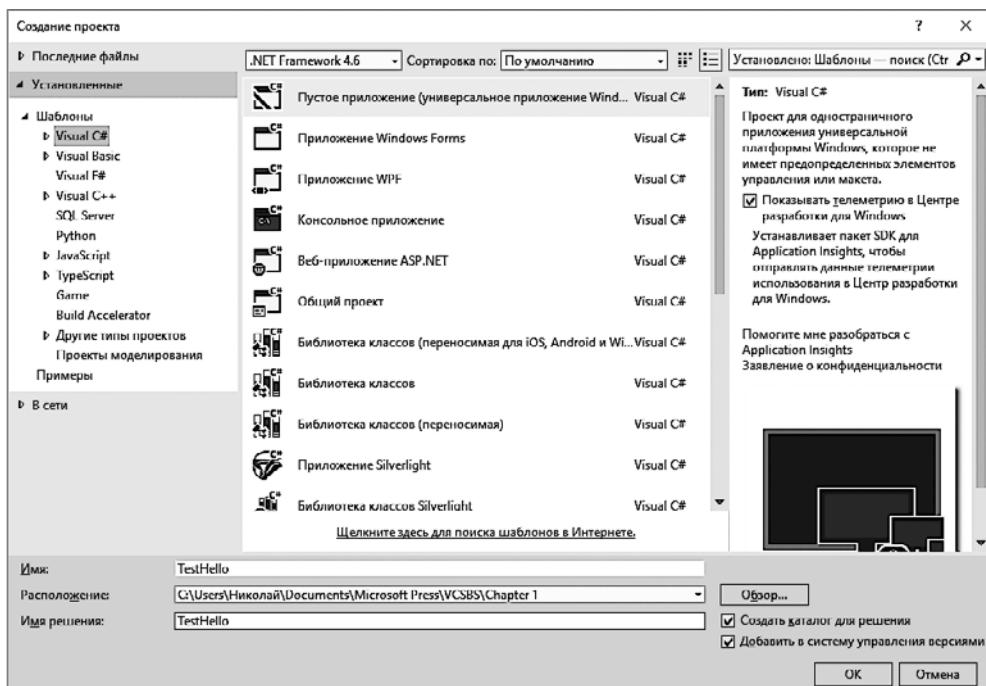


Рис. 1.2

В поле Размещение (Location) наберите строку `C:\Users\YourName\Documents\Microsoft Press\VCSBS\Chapter 1`. Замените текст `YourName` в набранном пути своим именем пользователя, зарегистрированным в Windows.



ПРИМЕЧАНИЕ Чтобы не повторяться и сэкономить место, далее в книге ссылка на путь `C:\Users\YourName\Documents` будет сокращаться до принадлежащей вам папки `Documents`.



СОВЕТ Если указанной вами папки не существует, Visual Studio 2015 создаст ее.

В поле Имя (Name) наберите **TestHello** (вместо предлагаемого имени **ConsoleApplication1**). Установите флажок Создать каталог для решения (Create Directory For Solution) и снимите флажок Добавить в систему управления версиями (Add To Source Control), после чего щелкните на кнопке OK.

Visual Studio создает проект, используя шаблон Консольное приложение. Если будет выведено диалоговое окно (рис. 1.3) с вопросом о том, какую систему управления версиями следует выбрать, значит, вы забыли снять флажок Добавить в систему управления версиями. В этом случае просто щелкните на кнопке Отмена (Cancel), и проект будет создан без использования системы управления версиями.

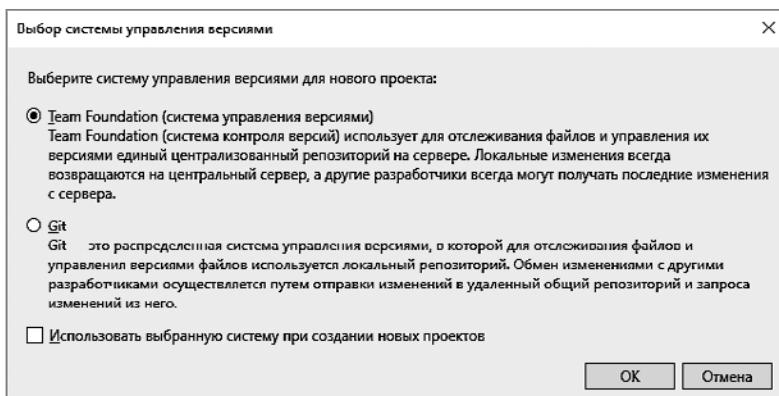


Рис. 1.3

Visual Studio выведет начальный код проекта (рис. 1.4).

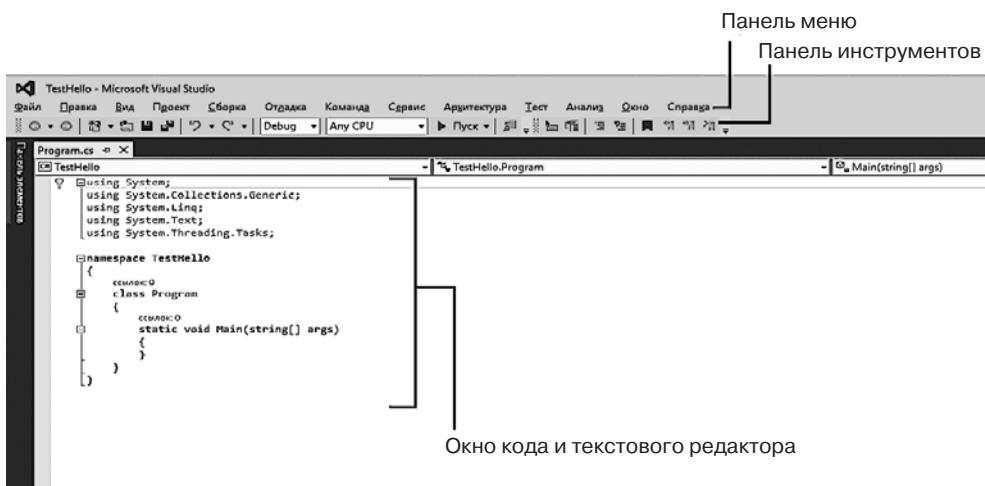


Рис. 1.4

Панель меню в верхней части экрана предоставляет доступ к функциональным возможностям, используемым в среде программирования. Для доступа к пунктам меню и командам можно использовать клавиатуру или мышь точно так же, как это делается во всех программах, работающих под управлением Windows. Сразу под панелью меню располагается панель инструментов. Она обеспечивает кнопочный способ запуска наиболее часто используемых команд.

Окно кода и текстового редактора, занимающее основную часть экрана, отображает содержимое исходных файлов. Если проект состоит из нескольких файлов, то при редактировании более чем одного файла у каждого исходного файла имеется собственная вкладка, подписанная его именем. Чтобы вывести файл на первый план в окне кода и текстового редактора, нужно щелкнуть на вкладке с его именем.

В правой части экрана интегрированной среды разработки находится панель Обозреватель решений (Solution Explorer), примыкающая к окну кода и текстового редактора (рис. 1.5).

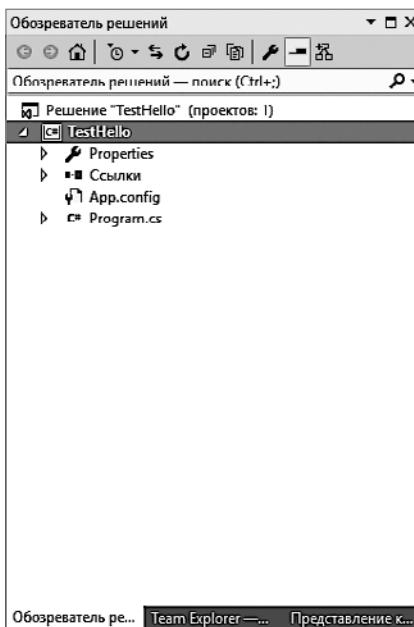


Рис. 1.5

В панели обозревателя решений среди прочего выводятся имена файлов, связанных с проектом. Чтобы исходный файл вышел на первый план в окне кода и текстового редактора, можно также дважды щелкнуть на его имени в обозревателе решений.

Перед написанием какого-либо кода изучите перечисленные в обозревателе решений файлы, созданные средой Visual Studio 2015 в качестве части вашего проекта.

- ❑ Решение TestHello. Это файл решения верхнего уровня. В каждом приложении имеется единственный файл решения. В решении может содержаться один или несколько проектов, и Visual Studio 2015 создает файл решения, чтобы помочь организовать эти проекты. Если в Проводнике просмотреть содержимое папки Documents\Microsoft Press\VCSBS\Chapter 1\TestHello, вы увидите, что этот файл назван TestHello.sln.
- ❑ TestHello. Это файл проекта C#. Каждый файл проекта ссылается на один или несколько файлов, содержащих исходный код, а также на другие артефакты проекта, например графические изображения. Весь исходный код одного и того же проекта должен быть написан на одном и том же языке программирования. В проводнике этот файл носит название TestHello.csproj и хранится в папке \Microsoft Press\VCSBS\Chapter 1\TestHello\TestHello вашей папки документов.
- ❑ Properties. Это папка в проекте TestHello. Если ее открыть (щелкнув на стрелке перед Properties), можно увидеть, что в ней содержится файл AssemblyInfo.cs. Это специальный файл, который можно использовать для добавления к программе таких атрибутов, как имя автора, дата написания программы и т. д. Для изменения способа запуска программы можно указать дополнительные атрибуты. Их использование в данной книге не рассматривается.
- ❑ Ссылки (References). Эта папка содержит ссылки на библиотеки скомпилированного кода, используемого вашим приложением. При компиляции код C# преобразуется в библиотеку и получает уникальное имя. В Microsoft .NET Framework такие библиотеки называются сборками. Разработчики используют сборки для создания написанных ими пакетов полезных функций, чтобы их можно было распространять среди других разработчиков, которые могут выразить желание воспользоваться этими функциями в собственных приложениях. Если открыть папку Ссылки (Referenses), станет виден исходный набор сборок, добавляемый Visual Studio 2015 в ваш проект. Эти сборки обеспечивают доступ ко многим часто используемым функциям .NET Framework и предоставляются компанией Microsoft вместе с Visual Studio 2015. О многих таких сборках вы узнаете по мере выполнения упражнений, приводимых в данной книге.
- ❑ App.config. Это файл конфигурации приложения. В силу своей необязательности он может и отсутствовать. В нем можно задать настройки, используемые вашим приложением во время его выполнения, для изменения его поведения, например версию .NET Framework, применяемую для запуска приложения. Более подробно этот файл будет рассматриваться в следующих главах.

- ❑ Program.cs. Это исходный файл C#, который отображается в окне кода и текстового редактора сразу после создания проекта. В этот файл будет записан ваш код для консольного приложения. В нем также содержится автоматически предоставленный Visual Studio 2015 код, который мы вскоре рассмотрим.

Написание вашей первой программы

В файле Program.cs определяется класс `Program`, содержащий метод `Main`. В C# весь исполняемый код должен быть определен внутри метода, а все методы должны принадлежать классу или структуре. Более подробно классы рассматриваются в главе 7 «Создание классов и объектов и управление ими», а структуры — в главе 9 «Создание типов значений с использованием перечислений и структур».

Метод `Main` определяет точку входа программы. Этот метод должен быть определен в порядке, указанном в классе `Program`, иначе .NET Framework может не распознать его в качестве стартовой точки при запуске вашего приложения. (Более подробно методы будут рассматриваться в главе 3 «Создание методов и применение областей видимости», а в главе 7 содержится больше информации о статических методах.)



ВНИМАНИЕ Язык C# чувствителен к регистру символов. `Main` нужно писать с большой буквы «M».

В следующих упражнениях вы напишете код, выводящий в окно консоли сообщение «Hello World!», создадите и запустите консольное приложение Hello World и узнаете, как для разделения элементов кода используется пространство имен.

Написание кода с использованием Microsoft IntelliSense

Найдите в окне кода и текстового редактора, где отображается содержимое файла Program.cs, метод `Main` и поместите курсор сразу же после принадлежащей ему открывающей фигурной скобки ({), затем для создания новой строки нажмите **Ввод**.

Наберите в новой строке слово `Console`, которое будет именем еще одного класса, предоставляемого сборками, на которые имеются ссылки из вашего приложения. Этот класс предоставляет методы для вывода сообщений в окно консоли и чтения ввода с клавиатуры.

Как только в начале слова `Console` будет набрана буква «C», появится список IntelliSense. В этом списке будут содержаться все ключевые слова C# и типы

данных, подходящие к данному контексту. Можно либо продолжить набор, либо прокрутить список и дважды щелкнуть кнопкой мыши на элементе `Console`. Или же после того, как будет набран фрагмент `Cons`, список IntelliSense автоматически выделит элемент `Console` (рис. 1.6), для выбора которого можно будет нажать клавишу табуляции или клавишу `Ввод`.



Рис. 1.6

Метод `Main` должен приобрести следующий вид:

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



ПРИМЕЧАНИЕ `Console` является встроенным классом.

Наберите сразу же после слова `Console` символ точки. Появится еще один список IntelliSense, показывающий методы, свойства и поля класса `Console`.

Прокрутите список вниз, выберите элемент `WriteLine` и нажмите `Ввод`. Можно также продолжить набор и вводить символы `W`, `r`, `i`, `t`, `e`, `L`, пока не будет выбран элемент `WriteLine`, а затем нажать `Ввод`.

Список IntelliSense закроется, и к исходному файлу будет добавлено слово `WriteLine`. Теперь `Main` должен приобрести следующий вид:

```
static void Main(string[] args)
{
    Console.WriteLine
}
```

Наберите открывающую скобку, (. Появится еще одна подсказка IntelliSense. В этой подсказке будут выведены параметры, воспринимаемые методом `WriteLine`, который относится к так называемым перегружаемым методам, а это означает, что в классе `Console` имеется более одного метода по имени `WriteLine`, фактически же им предствляются 19 различных версий этого метода. Каждую версию метода `WriteLine` можно использовать для вывода различных типов данных. (Более подробно перегружаемые методы рассматриваются в главе 3.) Теперь `Main` должен иметь следующий вид:

```
static void Main(string[] args)
{
    Console.WriteLine(
}
```



СОВЕТ Чтобы прокручивать различные варианты перегружаемого метода `WriteLine`, можно воспользоваться клавишами вертикальных стрелок.

Наберите закрывающую круглую скобку,), а после нее поставьте точку с запятой, ;. Теперь `Main` должен выглядеть следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine();
}
```

Переместите курсор и наберите строку "Hello World!", включая кавычки, между левой и правой круглыми скобками, которые следуют за именем метода `WriteLine`. Теперь `Main` должен приобрести следующий вид:

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



СОВЕТ Возьмите в привычку набирать соответствующие друг другу пары символов, например круглые скобки, (и), и фигурные скобки, { и }, до заполнения пространства между ними нужным содержимым. Если откладывать набор закрывающего символа на потом, то после ввода содержимого о нем легко забыть.

Значки IntelliSense

При наборе точки после имени класса IntelliSense показывает имя каждого элемента класса. Слева от каждого имени элемента располагается значок с изображением типа элемента. К числу стандартных значков и их типов относятся следующие.

Значок	Значение
	Метод (рассматривается в главе 3)
	Свойство (рассматривается в главе 15 «Реализация свойств для доступа к полям»)
	Класс (рассматривается в главе 7)
	Структура (рассматривается в главе 9)
	Перечисление (рассматривается в главе 9)
	Метод расширения (рассматривается в главе 12 «Работа с наследованием»)
	Интерфейс (рассматривается в главе 13 «Создание интерфейсов и определение абстрактных классов»)
	Делегирование (рассматривается в главе 17 «Введение в обобщения»)
	Событие (рассматривается в главе 17)
	Пространство имен (рассматривается в следующем разделе данной главы)

При наборе кода в различных контекстах будут попадаться и другие значки IntelliSense.

Вам будут часто встречаться строки кода, содержащие два прямых слеша (//), за которыми следует обычный текст. Это комментарии, игнорируемые компилятором, но приносящие большую пользу разработчикам, поскольку они помогают документировать текущие действия программы. Возьмем, к примеру, следующий фрагмент кода:

```
Console.ReadLine(); // Ожидание, пока пользователь не нажмет Ввод
```

Компилятор пропускает весь текст, начиная с двух слешей и заканчивая концом строки. Можно также добавлять многострочные комментарии, начинающиеся с прямого слеша со звездочкой (*). Компилятор пропустит все, пока не встретит звездочку с прямым слешем (*), которые могут находиться несколькими строчками ниже. Настоятельно рекомендуется документировать код в необходимых объемах предметными комментариями.

Сборка и запуск консольного приложения

Щелкните в меню Сборка (Build) на пункте Собрать решение (Build Solution). Код C# будет откомпилирован, в результате чего появится программа, которую можно будет запустить. Под окном редактора появится окно Вывод (Output).



СОВЕТ Если окно вывода не появится, щелкните для его отображения на пункте Вывод меню Вид (View).

В окне вывода должно появиться следующее сообщение, показывающее результаты компилирования программы:

```
1>----- Сборка начата: проект: TestHello, Конфигурация: Debug Any CPU -----
1> TestHello -> C:\Users\Николай\Documents\Microsoft Press\VCSBS\Chapter 1\
TestHello\TestHello\bin\Debug\TestHello.exe
===== Сборка: успешно: 1, с ошибками: 0, без изменений: 0, пропущен
```

Если допущены какие-либо ошибки, отчет о них будет помещен в окне Список ошибок (Error List). На рис. 1.7 показано, что получится, если забыть набрать закрывающие кавычки после текста Hello World в инструкции `WriteLine`. Обратите внимание на то, что одна допущенная ошибка может стать причиной сразу нескольких ошибок компиляции.

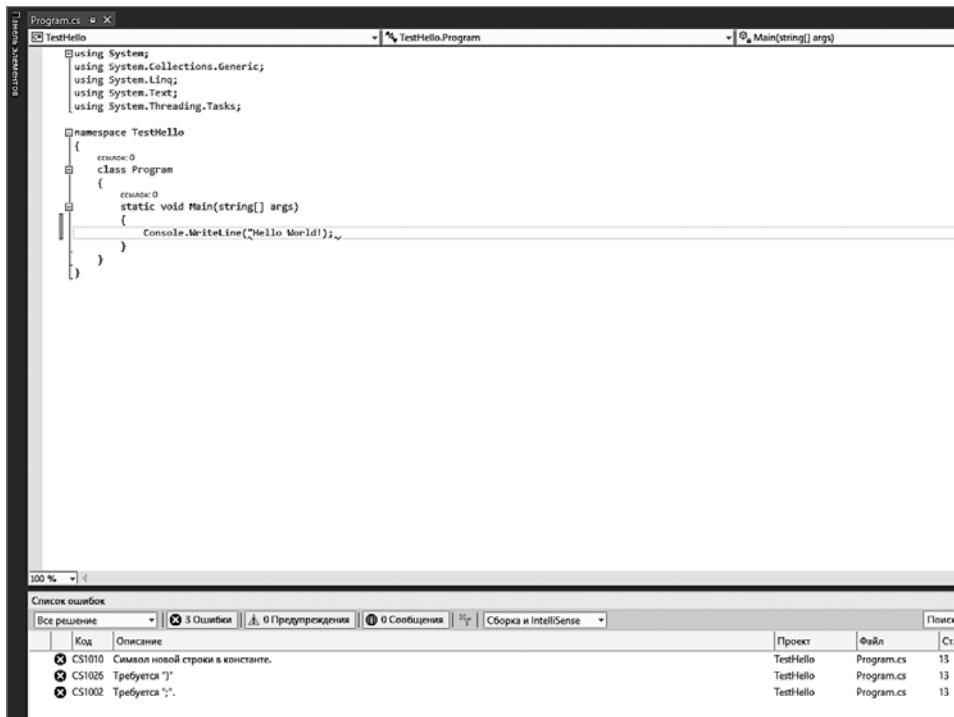


Рис. 1.7



СОВЕТ Чтобы попасть точно на ту строку, где находится ошибка, можно дважды щелкнуть на записи в окне списка ошибок. Обратите внимание на то, что Visual Studio выводит красную волнистую линию под всеми строками кода, не прошедшими компиляцию после их ввода.

Если вы точно следовали предыдущим инструкциям, никаких ошибок и предупреждений быть не должно и сборка программы должна завершиться успешно.



СОВЕТ Сохранять файл непосредственно перед сборкой нет никакого смысла, поскольку команда Собрать решение автоматически выполнит его сохранение.

Символ звездочки, стоящий после имени файла во вкладке над окном редактора, показывает, что файл со времени последнего сохранения был изменен.

Щелкните в меню Отладка (Debug) на пункте Запуск без отладки (Start Without Debugging). Откроется окно командной строки, и программа запустится. Появится сообщение «Hello World!». Как показано на рис. 1.8, теперь программа ждет от вас нажатия клавиши.

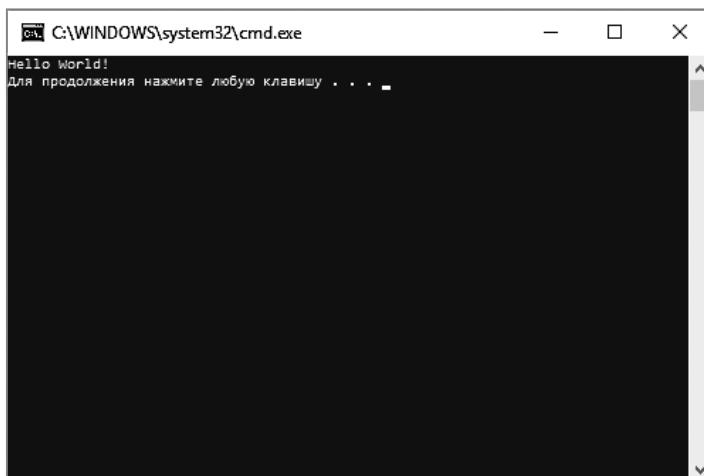


Рис. 1.8



ПРИМЕЧАНИЕ Приглашение «Для продолжения нажмите любую клавишу...» («Press any key to continue») генерируется Visual Studio, вы для этого не написали ни строчки кода. Если программа запущена с использованием команды Начать отладку (Start Debugging) в меню Отладка, приложение запускается, но окно командной строки закрывается сразу же, не ожидая нажатия клавиши.

Убедитесь в том, что окно командной строки, показывающее вывод программы, имеет фокус (то есть находится в активном состоянии), а затем нажмите Ввод. Окно закроется, и вы вернетесь в среду программирования Visual Studio 2015.

Щелкните на панели Обозреватель решений (Solution Explorer) на проекте (но не на решении) TestHello, а затем на панели задач этого обозревателя щелкните на

кнопке Показать все файлы (Show All Files). Чтобы эта кнопка появилась, может понадобиться щелкнуть на кнопке с двумя стрелками, находящейся на правом краю той же панели инструментов (рис. 1.9).

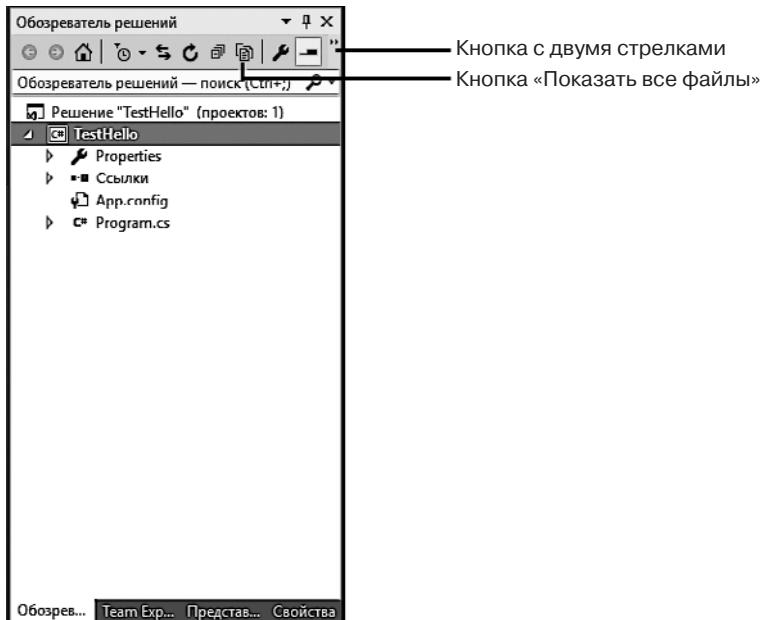


Рис. 1.9

Над именем файла Program.cs появятся записи bin и obj. Они напрямую связаны с одноименными папками, находящимися в папке проекта (Microsoft Press\VCBS\Chapter 1\ TestHello\TestHello). Visual Studio создает эти папки при сборке приложения. В них содержится исполняемая версия программы, а также некоторые другие файлы, используемые для сборки и отладки приложения.

Откройте в обозревателе решений папку bin. Появится еще одна папка с именем Debug (отладка).



ПРИМЕЧАНИЕ Можно также увидеть папку Release.

Откройте папку Debug. Появятся еще несколько элементов, включая файл TestHello.exe. Это скомпилированная программа в виде файла, запускаемого при щелчке на пункте Запуск без отладки в меню Отладка. В остальных файлах содержится информация, используемая Visual Studio 2015 при запуске программы в режиме отладки при щелчке на пункте Начать отладку (Start Debugging).

Использование пространств имен

Рассмотренный пример представляет собой очень маленькую программу. Но небольшие программы могут вскоре разрастись и приобрести весьма солидный объем. При увеличении размеров программы возникают две проблемы. Во-первых, по сравнению с небольшими программами, в больших труднее разобраться и их сложнее сопровождать. Во-вторых, обычно чем больше кода, тем больше классов, в которых больше методов, что требует отслеживания все большего количества имен. По мере роста количества имен растет и вероятность сбоя при сборке проекта из-за конфликта двух и более имен. Как, например, при попытке создания двух классов с одинаковыми именами. Ситуация еще больше усложняется, когда программа ссылается на сборки, написанные другими разработчиками, которые также использовали разнообразные имена.

В прошлом программисты пытались решить проблему конфликта имен, ставя перед именами префиксы с каким-либо классификатором (или набором классификаторов). Но это решение нельзя назвать удачным, поскольку оно не позволяет проводить масштабирование. Имена становятся длиннее, и написанию программы уделяется меньше времени, а на набор текста (там, где нужна разница в именах) и чтение, а также перечитывание слишком длинных имен его уходит гораздо больше.

Данная проблема решается с помощью пространств имен (*namespaces*) путем создания контейнеров для таких элементов, как классы. Два класса с одинаковыми именами не приведут к возникновению путаницы, если они находятся в разных пространствах имен. Используя ключевое слово `namespace`, можно внутри пространства имен под названием `TestHello` создать класс по имени `Greeting`:

```
namespace TestHello
{
    class Greeting
    {
        ...
    }
}
```

После этого в вашей программе можно обращаться к классу `Greeting` с помощью ссылки `TestHello.Greeting`. Если другой разработчик также создаст класс `Greeting`, но уже в другом пространстве имен, например `NewNamespace`, и вы установите сборку, содержащую этот класс, на свой компьютер, ваши программы не изменят своего поведения, поскольку в них используется ваш класс `TestHello.Greeting`. Если понадобится обратиться к классу `Greeting` другого разработчика, нужно будет указать на него как на `NewNamespace.Greeting`.

Определение всех классов в пространствах имен является рекомендуемой нормой, и среда Visual Studio 2015 следует этой рекомендации, используя в качестве пространства имен верхнего уровня название вашего проекта. В библиотеке классов .NET Framework также соблюдается эта норма, поэтому каждый класс в этой библиотеке находится в своем пространстве имен. Например, класс `Console` находится в пространстве имен `System`. Следовательно, его полным именем будет `System.Console`.

Конечно, если вам придется каждый раз писать полное имя класса, то это будет ничем не лучше использования префиксных классификаторов или глобальных имен вроде `System.Console`. К счастью, эту проблему можно решить с помощью директивы использования `using`. Если вернуться к программе `TestHello` в Visual Studio 2015 и посмотреть на содержимое файла `Program.cs`, то в самом начале этого файла можно заметить следующие строки:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

Это директивы `using`. С их помощью пространство имен вводится в область видимости. В последующем коде этого же файла уже не нужно конкретно указывать объектам то пространство имен, которому они принадлежат. Показанные пять пространств имен содержат классы, используемые настолько часто, что Visual Studio 2015 автоматически добавляет эти директивы `using` при каждом создании нового проекта. Если нужны ссылки на другие пространства имен, то в начале файла исходного кода можно добавлять дополнительные директивы `using`.

Более подробно концепция пространства имен иллюстрируется на следующих примерах.

Попытка работы с простым написанием имен

Закомментируйте в окне с содержимым файла `Program.cs` первую директиву `using`:

```
//using System;
```

Щелкните в меню Сборка на пункте Собрать решение. Сборка даст сбой, а в окно списка ошибок будет выведено сообщение

Имя "Console" не существует в текущем контексте.

Щелкните дважды на сообщении об ошибке. Идентификатор, вызвавший ошибку, будет подчеркнут в тексте исходного кода в Program.cs красной волнистой линией.

Отредактируйте метод `Main` под использование полностью классифицированного имени `System.Console`. Метод `Main` должен приобрести следующий вид:

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



ПРИМЕЧАНИЕ При наборе точки после `System` модуль IntelliSense выводит имена всех элементов, имеющихся в пространстве имен `System`.

Щелкните в меню Сборка на пункте Собрать решение. На этот раз сборка должна пройти успешно. Если этого не произошло, убедитесь, что у метода `Main` точно такой же код, как показан ранее, и проведите сборку повторно.

Запустите приложение, чтобы убедиться в его работоспособности, щелкнув на пункте Запуск без отладки (Start Without Debugging) меню Отладка. После запуска программы и вывода в окне консоли приветствия «Hello World!» нажмите Ввод, чтобы вернуться в Visual Studio 2015.

Пространства имен и сборки

Директива `using` просто помещает элементы пространства имен в область видимости и освобождает вас от необходимости использования в именах кода полной классификации. Классы компилируются в сборки.

Сборка представляет собой файл, для имени которого обычно используется расширение `.dll`, хотя, строго говоря, исполняемые программы в файлах с расширением `.exe` также являются сборками.

В сборке может содержаться множество классов. Классы, включенные в библиотеку классов .NET Framework, например `System.Console`, предоставляются в сборках, устанавливаемых на ваш компьютер вместе с Visual Studio. Вы увидите, что библиотека классов .NET Framework содержит тысячи классов. Если бы все они содержались в одной сборке, она была бы слишком большой и сложной в сопровождении. (Если бы компании Microsoft понадобилось обновить всего лишь один метод в одном из классов, то ей пришлось бы распространить всю библиотеку классов среди всех разработчиков!)

Поэтому библиотека классов .NET Framework разбита на целый ряд сборок, разделенных функциональными областями, к которым относятся содержащиеся в них классы. Например, основная сборка (`mscorlib.dll`) содержит все

классы общего назначения, такие как `System.Console`, а другие сборки содержат классы для работы с базами данных, обращения к веб-службам, создания GUI-интерфейсов и т. д. Если нужно воспользоваться классом в сборке, в проект следует добавить ссылку на эту сборку. Затем к коду можно добавить директиву `using`, помещающую элементы в пространство имен этой сборки в область видимости.

Следует заметить, что сборка и пространство имен необязательно должны относиться друг с другом по принципу один к одному: в одной сборке могут содержаться классы, определенные во многих пространствах имен, а одно пространство имен может распространяться на несколько сборок. Например, классы и элементы в пространстве имен `System` реализуются несколькими сборками, включая, кроме всего прочего, `mscorlib.dll`, `System.dll` и `System.Core.dll`. Поначалу все это кажется немного странным, но вскоре вы к этому привыкнете.

Когда Visual Studio используется для создания приложения, выбранный вами шаблон автоматически включает ссылки на соответствующие сборки. К примеру, откройте в обозревателе решений папку Ссылки проекта `TestHello`. Там вы увидите, что в консольное приложение автоматически включены ссылки на сборки с названиями `Microsoft.CSharp`, `System`, `System.Core`, `System.Data`, `System.Data.DataSetExtensions`, `System.Net.Http`, `System.Xml` и `System.Xml.Linq`. Возможно, вы удивитесь отсутствию в этом перечне библиотеки `mscorlib.dll`. Дело в том, что все приложения .NET Framework должны использовать эту сборку, поскольку в ней содержатся основные функции времени выполнения. В папке Ссылки перечислены только дополнительные сборки, и при необходимости их перечень можно расширить или сократить.

Чтобы добавить к проекту ссылки на дополнительные сборки, нужно щелкнуть правой кнопкой мыши на папке Ссылки, после чего щелкнуть на пункте Добавить ссылку (Add Reference). Этим вам придется заниматься при выполнении следующих упражнений. Удалить сборку можно из того же контекстного меню, щелкнув правой кнопкой на сборке и выбрав пункт Удалить (Remove).

Создание графического приложения

Итак, вы уже воспользовались средой Visual Studio 2015 для создания и запуска простого консольного приложения. Но в этой среде программирования есть все необходимое и для создания графических приложений для Windows 10. Ссылки на соответствующие шаблоны выглядят как универсальное приложение Windows (приложение Universal Windows Platform, или UWP-приложение), поскольку они позволяют создавать приложения, работающие на любых устройствах под управлением Windows: настольных компьютерах, планшетах и телефонах. Разработать пользовательский интерфейс (UI) Windows-приложения можно

в интерактивном режиме. После этого Visual Studio 2015 создаст программные инструкции для реализации разработанного вами пользовательского интерфейса.

Visual Studio 2015 предоставляет графическое приложение в двух видах: в виде конструктора и в виде кода. Окно редактора используется для изменения и обслуживания кода и программной логики графического приложения, а окно конструктора (Design View) — для разметки вашего пользовательского интерфейса (UI). Переключаться между этими двумя видами можно будет по мере надобности.

В следующем наборе упражнений вы научитесь создавать графическое приложение с использованием Visual Studio 2015. Создаваемая программа показывает простую форму, содержащую текстовое поле, в которое можно ввести ваше имя, и кнопку, после щелчка на которой в окне сообщений выводится персональное приветствие.

Если вам нужна более полная информация о специфике написания UWP-приложений, то соответствующее руководство и подробности вы найдете в последних нескольких главах части IV этой книги.

Создание графического приложения в Visual Studio 2015

Откройте диалоговое окно *Создание проекта*, щелкнув в меню **Файл** на пункте **Создать**, а затем на пункте **Проект**. Раскройте в левой панели узел **Установленные** и последовательно раскройте пункты **Шаблоны**, **Visual C#**, **Windows**, а затем щелкните на пункте **Универсальные**. Щелкните в средней панели на значке **Пустое приложение...** (Blank App...).



ПРИМЕЧАНИЕ XAML означает расширяемый язык разметки для приложений — Extensible Application Markup Language. Этот язык используется приложениями универсальной платформы Windows — Universal Windows Platform для определения разметки графического пользовательского интерфейса (GUI) приложения. Более полные сведения о XAML вы получите по мере выполнения представленных в книге упражнений.

В поле **Расположение (Location)** должна быть ссылка на папку **\Microsoft Press\VCSBS\Chapter 1**, которая находится в вашей папке **Документы**.

Наберите в поле **Имя** название проекта **Hello**. Установите флажок **Создать каталог для решения** и щелкните на кнопке **OK**.

Когда UWP-приложение создается в первый раз, может появиться предложение включить режим разработчика для Windows 10 (рис. 1.10). В зависимости от устройства, на котором вы работаете, и от версии Windows 10 у вас может быть возможность включить режим разработчика через интерфейс пользователя, но ее может и не быть. Дополнительное руководство по включению режима

разработчика можно найти в статье «Enable your device for development» по адресу <https://msdn.microsoft.com/library/windows/apps/xaml/dn706236.aspx>.

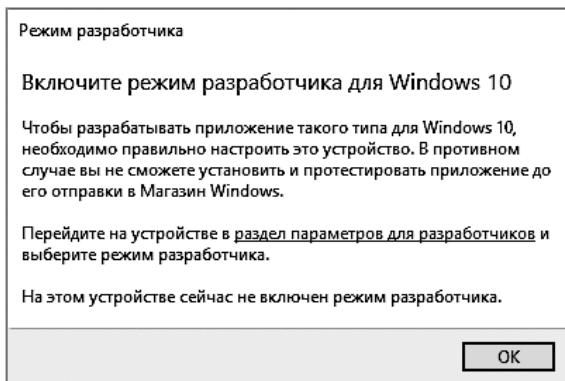


Рис. 1.10



ПРИМЕЧАНИЕ Это диалоговое окно может появиться при создании нового UWP-приложения или при первой попытке запуска UWP-приложения из Visual Studio.

После создания приложения обратите внимание на панель обозревателя решений. Пусть вас не смущает слово «пустое» в названии шаблона приложения — этот шаблон предоставляет целый ряд файлов и содержит программный код. К примеру, если открыть папку MainPage.xaml, то в ней окажется файл с кодом C# по имени MainPage.xaml.cs. Потом в этот файл нужно будет добавить код, запускаемый при отображении на экране пользовательского интерфейса, определенного в файле MainPage.xaml.

Дважды щелкните в обозревателе решений на имени файла MainPage.xaml. В этом файле содержится разметка пользовательского интерфейса. В окне конструктора будут показаны два представления данного файла (рис. 1.11).

В верхней части будет находиться графическое представление, по умолчанию изображающее 5-дюймовый экран телефона, а в нижней панели — описание содержимого этого экрана на XML-подобном языке XAML, который используется UWP-приложениями для определения разметки формы и ее содержимого. Тем, кто имеет опыт работы с XML, язык XAML должен показаться знакомым.

В следующем упражнении окно конструктора будет использоваться для разметки пользовательского интерфейса приложения и изучения XAML-кода, создаваемого на основе этой разметки.



СОВЕТ Чтобы получить больше места для отображения окна конструктора, закройте окна вывода и списка ошибок.

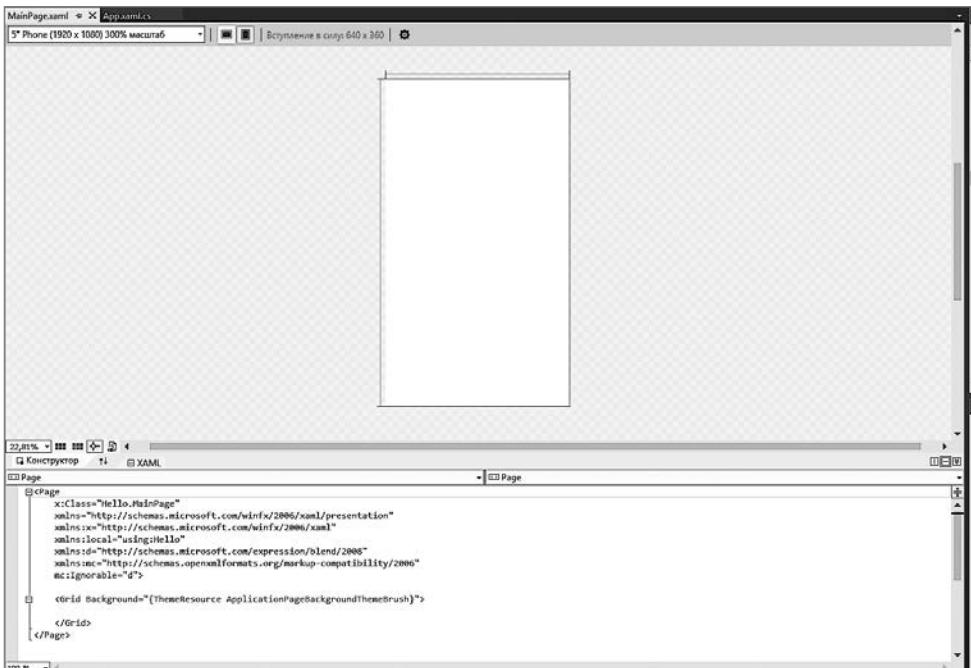


Рис. 1.11



ПРИМЕЧАНИЕ Перед тем как продолжить выполнение упражнения, нужно разобраться в некоторых терминологических тонкостях. В традиционных Windows-приложениях пользовательский интерфейс состоит из одного или нескольких окон, но в приложении, относящемся к Universal Windows Platform, соответствующие понятия фигурируют как страницы. Для большей ясности я просто буду ссылаться на оба понятия, используя общий термин «форма», но при этом продолжу использовать слово «окно» для ссылок на такие элементы интегрированной среды разработки Visual Studio 2015, как, к примеру, окно конструктора.

В следующих примерах окно конструктора будет использоваться для добавления к форме, выводимой вашим приложением, трех элементов управления. Кроме того, будет проанализирован код C#, автоматически создаваемый Visual Studio 2015 для реализации этих элементов управления.

Создание пользовательского интерфейса

Раскройте вкладку Панель элементов (Toolbox), появившуюся слева от формы в окне конструктора, где отобразятся различные компоненты и элементы управления, которые можно помещать в форму. Раскройте раздел Типовые элементы управления XAML (Common XAML Controls), где вы увидите перечень элементов управления, используемых большинством графических приложений.



СОВЕТ Расширенный перечень элементов управления приведен в разделе Все элементы управления XAML (All XAML Controls).

Щелкните в этом разделе на пункте **TextBlock** и перетащите представляемый им элемент управления в форму, отображаемую в окне конструктора.



СОВЕТ Не перепутайте **TextBlock** и **TextBox**. Если случайно в форму будет помещен не тот элемент управления, его можно будет удалить, щелкнув на его изображении в форме и нажав клавишу **Del**.

К форме будет добавлен элемент управления **TextBlock**, обозначающий надпись (который вскоре будет перемещен на предназначеннное ему место), и панель элементов скроется из виду.



СОВЕТ Если нужно, чтобы панель элементов оставалась видимой, но не заслоняла собой часть формы, найдите на правой границе заголовка этой панели кнопку Автоматически скрывать (Auto Hide), на которой изображен значок закрепления. Панель элементов зафиксируется в левой части окна Visual Studio 2015, и окно конструктора сожмется, приспособливаясь к новым условиям. (Если экран имеет низкое разрешение, то будет потеряна существенная часть рабочего пространства.) Повторный щелчок на кнопке автоматического скрытия приведет к тому, что панель элементов будет удалена с экрана.

Скорее всего, **TextBlock** не будет сразу помещен в форме именно туда, куда вам нужно. Добавленные к форме элементы можно перетаскивать, изменяя их местоположение. Переместите **TextBlock** в верхний левый угол формы. (Его точное местоположение для этого приложения не важно.) Учтите, что перед перемещением элемента в окне конструктора может сначала понадобиться щелкнуть где-нибудь за его пределами, а затем еще раз щелкнуть, но уже на самом элементе.

Теперь XAML-описание формы в нижней панели будет включать элемент управления **TextBlock** вместе с такими свойствами, как его местоположение в форме, регулируемое свойством **Margin**, исходный текст, отображаемый этим элементом управления и задаваемый значением свойства **Text**, настройка выравнивания текста, который выводится этим элементом управления, регулируемая свойствами **HorizontalAlignment** и **VerticalAlignment**, и настройка необходимости переноса текста на новые строки, если он шире элемента.

XAML-код для **TextBlock** будет иметь приблизительно следующий вид (в зависимости от конкретного размещения **TextBlock** в форме значения свойства **Margin** могут быть несколько иными):

```
<TextBlock x:Name="textBlock" HorizontalAlignment="Left" Margin="10,10,0,0"
TextWrapping="Wrap"
Text="TextBlock" VerticalAlignment="Top"/>
```

Панель XAML и окно конструктора имеют двунаправленные взаимоотношения друг с другом. Можно отредактировать значения в панели XAML, и изменения будут отображены в окне конструктора. Например, можно изменить местоположение `TextBlock`, отредактировав значения в свойстве `Margin`.

Щелкните в меню Вид (View) на пункте Окно свойств (Properties Window). Если этого окна еще не было на экране, то оно появится в его правом нижнем углу под панелью обозревателя решений. Свойства элементов управления можно указать с помощью панели XAML под окном конструктора, но окно свойств предоставляет более удобный способ изменения свойств элементов формы и других элементов проекта.

Содержимое окна свойств зависит от контекста и показывает свойства текущего выбранного элемента. Если щелкнуть на форме, отображаемой в окне конструктора (за пределами элемента `TextBlock`), в окне свойств будут выведены свойства элемента `Grid`. Если посмотреть на панель XAML, станет понятно, что элемент `TextBlock` находится внутри элемента `Grid`. Этот элемент содержится во всех формах и управляет разметкой отображаемых элементов. Можно, к примеру, определить табличные макеты, добавив к `Grid` строки и столбцы.

Щелкните в окне конструктора на элементе `TextBlock`, и в окне свойств опять появятся свойства этого элемента.

Раскройте в окне свойств свойство `Text`. Измените свойство `FontSize`, установив для него значение `20 pt`, и нажмите Ввод. Это свойство следует сразу же за раскрывающимся списком с названиями шрифтов, в котором показан шрифт `Segoe UI` (рис. 1.12).

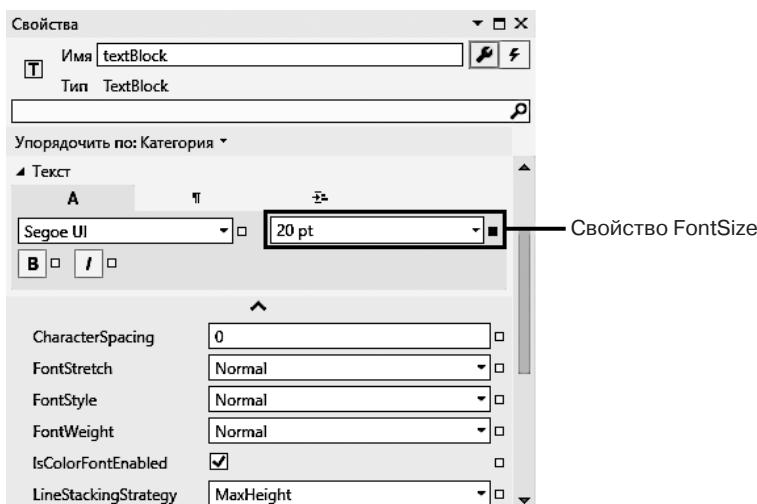


Рис. 1.12



ПРИМЕЧАНИЕ Суффикс `pt` показывает, что размер шрифта измеряется в пунктах, где 1 пункт равен 1/72 дюйма.

Посмотрите на текст с определением элемента управления `TextBlock` в панели XAML, расположенной ниже окна конструктора. Если добраться до конца строки, то станет виден текст `FontSize= "26.667"`. Это значение, полученное путем приблизительного преобразования размера шрифта из пунктов в пиксели (считается, что 3 пункта приблизительно равны 4 пикселям, хотя точное преобразование будет зависеть от размера и разрешения вашего экрана). Любые изменения, внесенные в окне свойств, автоматически отображаются в XAML-определении, и наоборот.

Поменяйте значение атрибута `FontSize` в панели XAML на `24`. Размеры шрифта текста для элемента `TextBlock` в окне конструктора и в окне свойств тут же изменятся.

Изучите в окне свойств другие свойства элемента `TextBlock`. Поэкспериментируйте с ними, изменяя их значения и наблюдая за производимыми эффектами.

Обратите внимание на то, что в ходе изменения значений свойств эти свойства добавляются в определение элемента `TextBlock` на панели XAML. Каждый добавляемый к форме элемент имеет исходный набор значений свойств, и эти значения не отображаются в панели XAML, пока в них не вносятся изменения.

Измените значение свойства `Text` элемента `TextBlock` с `TextBlock` на **Пожалуйста, введите свое имя**. Это можно сделать либо редактированием элемента `Text` в панели XAML, либо изменением значения в окне свойств, где это свойство находится в разделе **Общие** (`Common`). Обратите внимание на изменение текста, показанного в элементе `TextBlock` в окне конструктора.

Щелкните еще раз на форме в окне конструктора, а затем снова выведите на экран панель элементов. Перетащите из этой панели на форму элемент управления `TextBox`, обозначающий текстовое поле. Разместите этот элемент непосредственно под элементом управления `TextBlock`.



СОВЕТ Когда при перетаскивании элемента в форму он выходит на уровень вертикальных или горизонтальных границ других элементов, автоматически появляются индикаторы выравнивания. Это позволяет быстро оценить точность выравнивания элементов управления друг относительно друга.

Поместите указатель мыши в окне конструктора над правым краем элемента `TextBox`. Указатель должен принять вид двунаправленной стрелки, указывающей на возможность изменения размера элемента. Перетащите правый край элемента `TextBox` на уровень правого края расположенного выше элемента `TextBlock`, и когда края элементов будут на одном уровне, появится индикатор выравнивания.

Выберите элемент TextBox и в верхней части окна свойств измените значение свойства Name с textBox на userName (рис. 1.13).

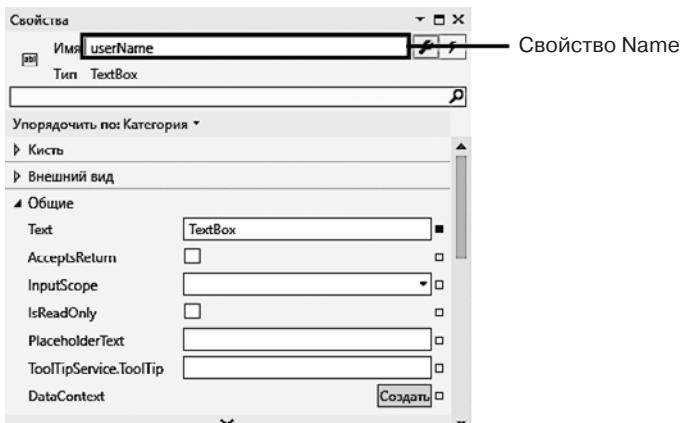


Рис. 1.13



ПРИМЕЧАНИЕ Более подробно соглашение об именах элементов управления и переменных будет рассматриваться в главе 2 «Работа с переменными, операторами и выражениями».

Еще раз выведите на экран панель элементов и перетащите с нее на форму элемент управления Button, обозначающий кнопку. Поместите кнопку справа от элемента TextBox, выровняв ее по горизонтали с нижним краем текстового поля.

Используя окно свойств, измените свойство Name элемента Button на ok и измените значение свойства Content в разделе Общие с Button на OK, после чего нажмите Ввод. Убедитесь, что надпись в элементе Button на форме изменилась и показывает текст OK.

Теперь форма должна приобрести приблизительно следующий вид (рис. 1.14).



ПРИМЕЧАНИЕ Раскрывающееся меню в левом верхнем углу окна конструктора позволяет посмотреть, как форма будет выводиться на экраны разных размеров и разрешений. В данном примере исходным выбором является просмотр на экране пятидюймового телефона с разрешением 1920×1080. Справа от раскрывающегося меню расположены две кнопки, позволяющие переключаться между книжной и альбомной ориентацией экрана. Проекты следующих глав будут использовать в качестве рабочей области конструирования экран настольного компьютера с диагональю 13,3 дюйма, но для этого упражнения можно оставить форм-фактор пятидюймового телефона.

Щелкните в меню Сборка на пункте Собрать решение, после чего убедитесь в успешной сборке проекта.

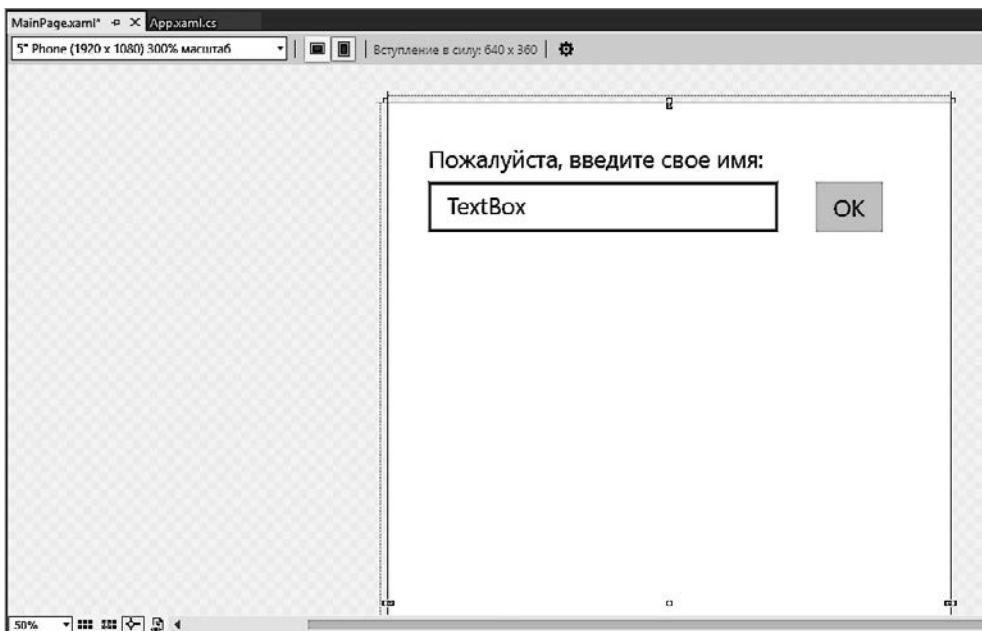


Рис. 1.14

Убедитесь, что в показанном на рис. 1.15 раскрывающемся списке цели отладчика выбран пункт **Локальный компьютер** (Local Machine). (Изначально целью отладчика может быть Device, и среда попытается подключиться к устройству Windows phone, в результате чего сборка пройдет неудачно.) Затем щелкните в меню Отладка на пункте Начать отладку.

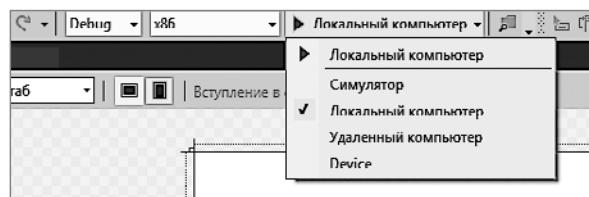
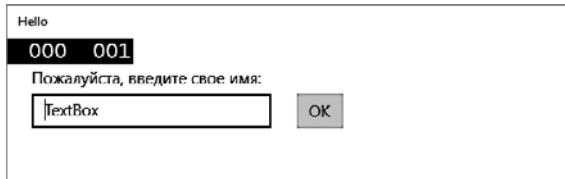


Рис. 1.15

Приложение должно запуститься и вывести на экран вашу форму, которая будет иметь следующий вид (рис. 1.16).

В текстовом поле поверх имеющегося текста можно набрать свое имя, а затем щелкнуть на кнопке OK, но пока что это ни к чему не приведет. Чтобы показать, что должно произойти при щелчке пользователя на кнопке OK, нужно добавить программный код, чем мы далее и займемся.

**Рис. 1.16**

ПРИМЕЧАНИЕ При запуске UWP-приложения в режиме отладки в левом верхнем углу формы появляются два числа. Эти числа являются результатом трассировки частоты кадров, и разработчики могут воспользоваться этой информацией для определения момента, когда приложение начинает откликаться медленнее положенного (возможно, выявляя при этом проблемы производительности). Эти числа появляются только тогда, когда приложение запускается в режиме отладки. Их полное описание не входит в круг вопросов, рассматриваемых в данной книге, поэтому сейчас вы можете их проигнорировать.

Вернитесь в Visual Studio 2015. Щелкните в меню Отладка на пункте Остановить отладку (Stop Debugging).



ПРИМЕЧАНИЕ Чтобы закрыть форму, остановить отладку и вернуться в Visual Studio, можно также щелкнуть на кнопке закрытия (которая имеет значок X и находится в правом верхнем углу формы).

Вам удалось справиться с созданием графического приложения, не написав ни одной строчки кода C#. Это приложение пока что ни к чему не пригодно (поэтому вскоре вам придется заняться написанием дополнительного кода), но среда Visual Studio 2015 фактически уже создала для вас немалый объем кода, справляющийся с рутинными задачами, выполняемыми всеми графическими приложениями, в числе которых запуск и отображение окна. Перед добавлением к приложению собственного кода полезно будет разобраться в том, что именно среда Visual Studio создала для вас. Автоматически созданные ею артефакты рассматриваются в следующем разделе.

Изучение UWP-приложения

Раскройте в обозревателе решений узел MainPage.xaml, после чего дважды щелкните на появившемся имени файла MainPage.xaml.cs. В окне кода и текстового редактора появится следующий предназначенный для формы код:

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

```
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// Документацию по шаблону элемента "Пустая страница" см. по адресу
// http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

namespace Hello
{
    /// <summary>
    /// Пустая страница, которую можно использовать саму по себе или для перехода
    /// внутри фрейма.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

Кроме солидного количества директив `using`, вводящих в область видимости ряд пространств имен, используемых большинством UWP-приложений, в файле содержится лишь определение класса `MainPage`. В этом определении имеется небольшой фрагмент кода для класса `MainPage`, известный как конструктор, вызывающий метод `InitializeComponent`. Конструктором называется специальный метод, имя которого совпадает с именем класса. Он запускается при создании экземпляра класса и может содержать код для инициализации экземпляра. Конструкторы будут рассматриваться в главе 7.

Обычно в классе содержится намного больше кода, чем те несколько строк, что показаны в файле `MainPage.xaml.cs`, но существенная часть этого кода генерируется автоматически на основе XAML-описания формы и скрыта от просмотра. Этот скрытый код выполняет такие операции, как создание и вывод формы и создание и позиционирование в форме различных элементов управления.



СОВЕТ Файл кода C# для страницы в UWP-приложении можно также вывести на экран при отображении окна конструктора, щелкнув в меню Вид на пункте Код.

Вы можете спросить: а где же метод `Main` и как форма отображается при запуске приложения? Вспомним, что в консольном приложении метод `Main` определяет точку старта программы. Графическое приложение немного отличается от консольного.

В обозревателе решений нужно обратить внимание на другой исходный файл по имени App.xaml. Если раскрыть узел, показанный для этого файла, то станет виден еще один файл по имени App.xaml.cs. В UWP-приложении файл App.xaml предоставляет точку входа, с которой начинается запуск приложения. Если в обозревателе решений дважды щелкнуть на имени файла App.xaml.cs, станет виден код, имеющий следующий вид:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

namespace Hello
{
    /// <summary>
    /// Обеспечивает зависящее от конкретного приложения поведение, дополняющее
    /// класс Application по умолчанию.
    /// </summary>
    sealed partial class App : Application
    {
        /// <summary>
        /// Инициализирует одноэлементный объект приложения. Это первая
        /// выполняемая строка разрабатываемого
        /// кода, поэтому она является логическим эквивалентом main() или
        /// WinMain().
        /// </summary>
        public App()
        {
            Microsoft.ApplicationInsights.WindowsAppInitializer.InitializeAsync(
                Microsoft.ApplicationInsights.WindowsCollectors.Metadata |
                Microsoft.ApplicationInsights.WindowsCollectors.Session);
            this.InitializeComponent();
            this.Suspending += OnSuspending;
        }

        /// <summary>
        /// Вызывается при обычном запуске приложения пользователем. Будут
        /// использоваться другие точки входа,
        /// например, если приложение запускается для открытия конкретного файла.
        /// </summary>
        /// <param name="e">Сведения о запросе и обработке запуска.</param>
```

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
#if DEBUG
    if (System.Diagnostics.Debugger.IsAttached)
    {
        this.DebugSettings.EnableFrameRateCounter = true;
    }
#endif

    Frame rootFrame = Window.Current.Content as Frame;

    // Не повторяйте инициализацию приложения, если в окне уже имеется
    // содержимое, только обеспечьте активность окна
    if (rootFrame == null)
    {
        // Создание фрейма, который станет контекстом навигации, и переход
        // к первой странице
        rootFrame = new Frame();

        rootFrame.NavigationFailed += OnNavigationFailed;

        if (e.PreviousExecutionState ==
                        ApplicationExecutionState.Terminated)
        {
            //TODO: Загрузить состояние из ранее приостановленного
            // приложения
        }

        // Размещение фрейма в текущем окне
        Window.Current.Content = rootFrame;
    }

    if (rootFrame.Content == null)
    {
        // Если стек навигации не восстанавливается для перехода к первой
        // странице, настройка новой страницы осуществляется путем передачи
        // необходимой информации в качестве параметра навигации
        rootFrame.Navigate(typeof(MainPage), e.Arguments);
    }
    // Обеспечение активности текущего окна
    Window.Current.Activate();
}

/// <summary>
/// Вызывается в случае сбоя навигации на определенную страницу
/// </summary>
/// <param name="sender">Фрейм, для которого произошел сбой
/// навигации</param>
/// <param name="e">Сведения о сбое навигации</param>
void OnNavigationFailed(object sender, NavigationFailedEventArgs e)
{
    throw new Exception("Failed to load Page " +
        e.SourcePageType.FullName);
}
```

```
/// <summary>
/// Вызывается при приостановке выполнения приложения. Состояние
/// приложения сохраняется без учета информации о том, будет оно
/// завершено или возобновлено с неизменным содержимым памяти.
/// </summary>
/// <param name="sender">Источник запроса приостановки.</param>
/// <param name="e">Сведения о запросе приостановки.</param>
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Сохранить состояние приложения и остановить все фоновые
    // операции
    deferral.Complete();
}
}
```

Значительная часть этого кода состоит из комментариев (строк, начинающихся с `///`) и других инструкций, в которых вам пока не нужно разбираться, а ключевые элементы находятся в методе `OnLaunched`, код которого выделен полужирным шрифтом. Этот метод запускается в начале работы приложения, а его код заставляет приложение создать новый объект фрейма — `Frame`, вывести в этом фрейме форму `MainPage`, а затем активировать эту форму. На данном этапе нам не нужно полностью разбираться в принципе работы этого кода или понимать синтаксис любой из этих инструкций, достаточно просто понять, что этот код предоставляет способ отображения формы при запуске приложения.

Добавление кода к графическому приложению

Немного разобравшись в структуре графического приложения, следует перейти к написанию кода, позволяющего приложению выполнять какую-либо реальную работу.

Давайте напишем код для кнопки ОК. Откройте в окне конструктора файл `MainPage.xaml`, дважды щелкнув на его имени в обозревателе решений. Выберите в форме окна конструктора кнопку ОК, дважды щелкнув на ее изображении. Щелкните в окне свойств на кнопке Обработчик событий для выбранного элемента (Event Handlers button for the selected element). Значок на этой кнопке похож на вспышку молнии (рис. 1.17).

В окне свойств появится перечень событий для кнопки. Событие указывает на значимое действие, требующее, как правило, ответной реакции, для которой можно написать собственный код.

Наберите в поле, относящемся к событию `Click`, текст `okClick` и нажмите Ввод. В окне редактора появится содержимое файла `MainPage.xaml.cs` с новым,

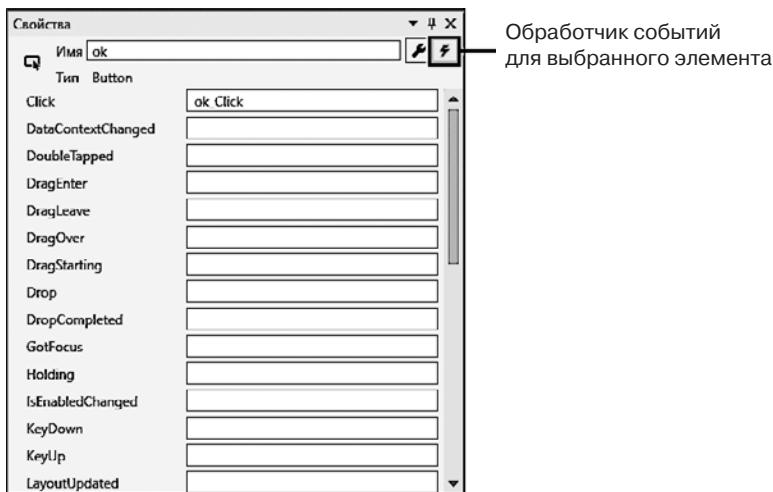


Рис. 1.17

добавленным к классу `MainPage` методом под названием `okClick`, у которого будет следующий внешний вид:

```
private void okClick(object sender, RoutedEventArgs e)
{
}
```

Синтаксис этого кода нас пока не интересует, поскольку подробнее методы будут рассмотрены в главе 3. Добавьте к перечню в начале файла директиву `using` (выделена жирным шрифтом, символ многоточия ... заменяет пропущенные для краткости инструкции):

```
using System;
...
using Windows.UI.Xaml.Navigation;
using Windows.UI.Popups;
```

Добавьте к методу `okClick` код, выделенный жирным шрифтом:

```
private void okClick(object sender, RoutedEventArgs e)
{
    MessageDialog msg = new MessageDialog("Привет, " + userName.Text);
    msg.ShowAsync();
}
```

Это код запустится, когда пользователь щелкнет на кнопке **OK**. Здесь мы также пока оставим синтаксис без внимания, от вас требуется лишь точно скопировать показанный код, а смысл инструкций станет понятен при изучении нескольких следующих глав. Главное — уяснить, что первая инструкция создает объект

MessageDialog с сообщением «Привет, <ВашеИмя>», где вместо вставки <ВашеИмя> появится то имя, которое вы ввели в текстовое поле TextBox формы. Вторая инструкция отобразит диалоговое окно сообщения MessageDialog, что приведет к его появлению на экране. Класс MessageDialog определен в пространстве имен Windows.UI.Popups, поэтому данное пространство и было ранее добавлено в программу.



ПРИМЕЧАНИЕ Под последней строкой введенного кода можно будет заметить зеленую волнистую линию, добавленную Visual Studio 2015. Если навести на нее указатель мыши, Visual Studio выведет следующее предупреждение: «Поскольку этот вызов не ожидается, выполнение текущего метода продолжается до завершения вызова. Попробуйте применить оператор await к результату вызова». По существу, это предупреждение означает, что вы не воспользовались всеми преимуществами асинхронных функциональных возможностей, предоставляемых библиотекой .NET Framework. Это предупреждение можно проигнорировать.

Чтобы снова отобразить форму в окне конструктора, щелкните на вкладке MainPage.xaml, которая находится над окном редактора.

Посмотрите на XAML-описание формы, находящееся на нижней панели, и найдите в нем элемент Button, но ничего в нем не меняйте. Обратите внимание на то, что теперь он включает атрибут по имени Click, который ссылается на метод okClick:

```
<Button x:Name="ok" ... Click="okClick" />
```

Щелкните в меню Отладка на пункте Начать отладку. При появлении формы введите в текстовое поле поверх имеющегося текста свое имя и щелкните на кнопке OK. Появится диалоговое окно сообщения, отображающее следующее приветствие (рис. 1.18).

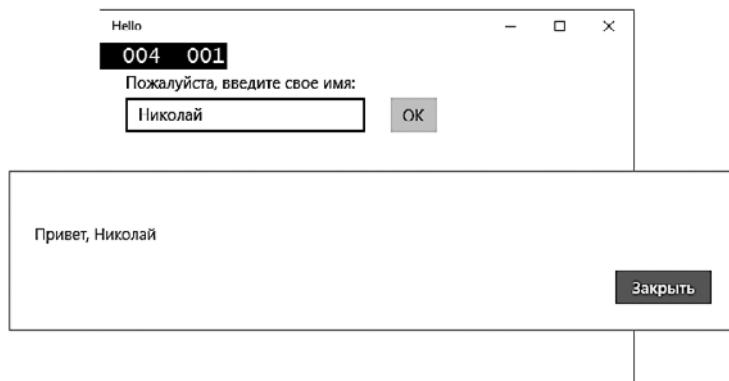


Рис. 1.18

В окне сообщения щелкните на кнопке Закрыть. Вернитесь в Visual Studio 2015 и щелкните в меню Отладка на пункте Остановить отладку.

Выходы

В этой главе было показано, как с помощью Visual Studio 2015 создавать, собирать и запускать приложения. Было создано консольное приложение, отображающее свой вывод в окне консоли, и UWP-приложение с простым графическим интерфейсом (GUI).

Если хотите перейти к работе с главой 2, оставьте среду Visual Studio 2015 в рабочем состоянии и приступайте.

Если сейчас вам нужно выйти из Visual Studio 2015, щелкните в меню Файл на пункте Выход. Если увидите диалоговое окно с предложением сохранить проект, ответьте согласием.

Краткий справочник

Чтобы	Сделайте следующее
Создать с помощью Visual Studio 2015 новое консольное приложение	Откройте диалоговое окно нового проекта: Файл ▶ Создать ▶ Проект. Раскройте в левой панели Установленные ▶ Шаблоны ▶ Visual C#. Щелкните в средней панели на пункте Консольное приложение. Укажите в поле Расположение каталог для файлов проекта. Введите имя проекта и щелкните на кнопке OK
Создать с помощью Visual Studio 2015 новое пустое UWP-приложение	Откройте диалоговое окно нового проекта: Файл ▶ Создать ▶ Проект. Раскройте в левой панели Установленные ▶ Шаблоны ▶ Visual C# ▶ Windows ▶ Универсальные. Щелкните в средней панели на пункте Пустое приложение (универсальное приложение Windows). Укажите в поле Расположение каталог для файлов проекта. Введите имя проекта и щелкните на кнопке OK
Собрать приложение	Щелкните в меню Сборка на пункте Собрать решение
Запустить приложение в режиме отладки	Щелкните в меню Отладка на пункте Начать отладку
Запустить приложение без отладки	Щелкните в меню Отладка на пункте Запуск без отладки

2 Работа с переменными, операторами и выражениями

Прочитав эту главу, вы научитесь:

- разбираться в инструкциях, идентификаторах и ключевых словах;
- использовать переменные для хранения информации;
- работать с простыми типами данных;
- использовать арифметические операторы, такие как знаки «плюс» (+) и «минус» (-);
- увеличивать и уменьшать значения переменных.

В главе 1 «Добро пожаловать в C#» были рассмотрены вопросы использования среды программирования Microsoft Visual Studio 2015 для создания и запуска консольных программ и графических приложений. В данной главе будут представлены элементы синтаксиса и семантики Microsoft Visual C#, включая инструкции, ключевые слова и идентификаторы. Вы научитесь использовать простые типы данных, встроенные в язык C#, и разбираться в характеристиках значений, содержащихся в каждом типе. Вы также увидите, как объявляются и используются локальные переменные (которые существуют только внутри метода или другого небольшого раздела кода), узнаете о предоставляемых C# арифметических операторах, научитесь пользоваться операторами для работы со значениями и узнаете, как работать с выражениями, содержащими два и более оператора.

Понятие об инструкциях

Инструкция представляет собой команду, выполняющую какое-либо действие, например вычисление значения и сохранение результата или вывод сообщения на экран пользователя. Инструкции объединяются для создания методов. Более подробно методы рассматриваются в главе 3 «Создание методов и применение

областей видимости», а пока представим себе, что методы являются поименованными последовательностями инструкций. Примером метода может служить `main`, представленный в предыдущей главе.

Инструкции в C# придерживаются четко определенного набора правил, описывающих их формат и конструкцию. Совокупность этих правил называется **синтаксисом** (в отличие от *семантики*, являющейся совокупностью описаний *действий* этих инструкций). Одно из самых простых и наиболее важных правил синтаксиса C# гласит, что все инструкции должны заканчиваться точкой с запятой. Например, в главе 1 было показано, что без закрывающей точки с запятой следующая инструкция компиляцию не пройдет:

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



СОВЕТ C# считается языком свободного формата, а это значит, что пробельные символы, такие как символы пробела или новой строки, не имеют значения, если только не применяются в качестве разделителей. Иными словами, вы вправе размечать свои инструкции в любом удобном стиле. Но чтобы ваши программы легче читались и в них было проще разобраться, лучше придерживаться простого постоянного стиля разметки.

Успех программирования на любом языке обусловливается изучением его синтаксиса и семантики с последующим использованием языка естественным, характерным для него образом. Такой подход упростит сопровождение ваших программ. По мере изучения материала данной книги вы увидите примеры использования наиболее важных инструкций C#.

Использование идентификаторов

Идентификаторами называются имена, используемые вами для идентификации элементов в своих программах, например пространств имен, классов, методов и переменных. (Что такое переменные, вы скоро узнаете.) При выборе идентификаторов в C# нужно придерживаться следующих синтаксических правил.

- ❑ Разрешается использовать только буквы (в верхнем или в нижнем регистре), цифры и символы подчеркивания.
- ❑ Идентификаторы должны начинаться с буквы или символа подчеркивания.

Например, `result`, `_score`, `footballTeam` и `plan9` считаются допустимыми идентификаторами, а `result%`, `footballTeam$` и `9plan` не допускаются.



ВНИМАНИЕ C# относится к языкам, чувствительным к регистру символов: `footballTeam` и `FootballTeam` — это два разных идентификатора.

Идентификация ключевых слов

В C# для внутреннего потребления зарезервировано 77 идентификаторов, которые вам нельзя использовать для своих нужд. Эти идентификаторы называются ключевыми словами, и у каждого из них имеется конкретное назначение. К примерам ключевых слов относятся `class`, `namespace` и `using`. По мере чтения книги вы изучите большинство ключевых слов C#. Перечень ключевых слов выглядит следующим образом.

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



СОВЕТ В окне редактора Visual Studio 2015 набранные ключевые слова выделяются синим цветом.

В C# используются также идентификаторы, показанные далее. Они не считаются зарезервированными в C#, следовательно, их можно использовать в качестве идентификаторов своих методов, переменных и классов, но по возможности этого следует избегать.

add	Get	remove
alias	Global	select
ascending	Group	set
async	Into	value
await	Join	var
descending	Let	where
dynamic	orderby	yield
from	partial	

Использование переменных

Переменной называется место в памяти, где хранится значение. Переменную можно представить в виде блока в компьютерной памяти, в котором хранится временная информация. Каждой переменной в программе должно быть дано вполне определенное имя, которое уникальным образом ее идентифицирует в используемом контексте. Имя переменной используется для ссылки на содержащееся в этой переменной значение. Например, если нужно сохранить значение стоимости товара в магазине, можно создать переменную с простым именем `cost` и сохранить в ней стоимость товара.

После этого при ссылке на переменную `cost` извлекаемым значением будет сохраненная ранее стоимость товара.

Правила присваивания имен переменным

Во избежание путаницы с определяемыми вами переменными нужно придерживаться соглашения о порядке присваивания имен переменным. Особую важность это приобретает, если вы являетесь частью команды, состоящей из нескольких разработчиков, работающих над различными компонентами приложения, где строгое соглашение о присваивании имен помогает избежать путаницы и сократить поле допущения ошибок. Основные рекомендации изложены в следующем перечне.

- ❑ Не начинайте идентификатор со знака подчеркивания. Хотя это и допускается в C#, у вас могут возникнуть ограничения во взаимодействии вашего кода с приложениями, созданными на других языках, например на Microsoft Visual Basic.
- ❑ Не создавайте идентификаторы, отличающиеся друг от друга только регистром символов. Например, не создавайте для одновременного использования одну переменную с именем `myVariable`, а другую — с именем `MyVariable`,

потому что их легко перепутать. Кроме того, определение переменных, отличающихся друг от друга только регистром символов, ограничивает возможность повторного использования классов в приложениях, разработанных с помощью других языков, нечувствительных к регистру символов, таких, к примеру, как Visual Basic.

- ❑ Начинайте имя с буквы в нижнем регистре.
- ❑ В идентификаторе, состоящем из нескольких слов, начинайте второе и каждое последующее слово с буквы в верхнем регистре. (Такая форма записи называется смешанным регистром, или, по-английски, – camelCase.)
- ❑ Не используйте венгерскую нотацию. (Если у вас есть опыт разработки на Microsoft Visual C++, то, скорее всего, венгерская нотация вам знакома. Если же вы не знаете, что это такое, то и нечего забивать голову ненужной информацией!)

Например, `score`, `footballTeam`, `_score` и `FootballTeam` являются допустимыми именами переменных, но к числу рекомендованных относятся только два первых.

Объявление переменных

В переменных содержатся значения. В C# имеется множество различных типов значений, которые могут храниться и обрабатываться. Назовем только три из них: целые числа, числа с плавающей точкой и строки символов. При объявлении переменной нужно указать тип содержащихся в ней данных.

Объявление типа и имени переменной производится в инструкции объявления. Например, в следующей инструкции объявляется, что в переменной по имени `age` содержатся значения типа `int` (integer). Как всегда, инструкцию нужно завершить точкой с запятой:

```
int age;
```

Тип переменной, `int`, является названием одного из простых типов языка C#, `integer`, представляющего собой целое число. (Чуть позже в этой главе нам станут известны некоторые другие простые типы данных.)



ПРИМЕЧАНИЕ Если вы программируали на Visual Basic, то, наверное, заметили, что в C# запрещено неявное объявление переменных. Перед использованием все переменные сначала должны быть объявлены.

После объявления переменной ей можно присвоить значение. Следующая инструкция присваивает переменной `age` значение 42. Еще раз обратите внимание на обязательность использования точки с запятой:

```
age = 42;
```

Знак равенства (=) является оператором присваивания, который присваивает значение, находящееся справа от него, той переменной, которая находится слева от него. После этого присваивания переменную `age` можно использовать в вашем коде для ссылки на хранящееся в ней значение. Следующая инструкция записывает значение переменной `age` (42) на консоль:

```
Console.WriteLine(age);
```



СОВЕТ Если в окне редактора Visual Studio 2015 указатель мыши установить над переменной, то тип этой переменной будет показан в экранной подсказке (ScreenTip).

Работа с простыми типами данных

В C# имеется ряд встроенных типов, которые называются простыми типами данных. Наиболее распространенные простые типы данных C# и диапазоны их значений перечислены в табл. 2.1.

Таблица 2.1

Тип данных	Описание	Размерность, бит	Диапазон	Пример использования
int	Целые числа (integers)	32	От -2^{31} до $2^{31} - 1$	int count; count = 42;
long	Целые числа (с расширенным диапазоном)	64	От -2^{63} до $2^{63} - 1$	long wait; wait = 42L;
float	Числа с плавающей точкой	32	От $-3,4 \cdot 10^{-38}$ до $3,4 \cdot 10^{38}$	float away; away = 0.42F;
double	Числа с плавающей точкой двойной точности	64	От $\pm 5,0 \cdot 10^{-324}$ до $\pm 1,7 \cdot 10^{308}$	double trouble; trouble = 0.42;
decimal	Денежные значения	128	28 значащих цифр	decimal coin; coin = 0.42M;
string	Последовательность символов	По 16 бит на символ	Неприменимо	string vest; vest = "forty two";
char	Отдельно взятый символ	16	От 0 до $2^{16} - 1$	char grill; grill = 'x';
bool	Булево значение	8	true или false	bool teeth; teeth = false;

Локальные переменные с неприсвоенными значениями

При объявлении переменной в ней, пока ей не будет присвоено значение, содержится произвольное значение. Это обстоятельство часто становится источником ошибок в программах, написанных на С и С++, где переменная создавалась и случайно использовалась в качестве источника информации еще до того, как ей давалось значение. В C# использование переменных с неприсвоенными значениями недопустимо. Перед тем как воспользоваться переменной, ей нужно присвоить значение, в противном случае ваша программа не будет скомпилирована. Это требование называется правилом определенности переменной. Например, причиной выдачи сообщения об ошибке во время компиляции служат следующие инструкции: "Use of unassigned local variable 'age'" (Использование переменной 'age' с неприсвоенным значением), поскольку инструкция `Console.WriteLine` пытается вывести на экран значение неинициализированной переменной:

```
int age;
Console.WriteLine(age); // ошибка в ходе компиляции
```

Отображение значений, относящихся к простым типам данных

В следующем примере для демонстрации порядка работы нескольких простых типов данных используется программа на C# под названием `PrimitiveDataTypes`.

Отображение значений, относящихся к простым типам данных

Выберите в меню Файл среды Visual Studio 2015 пункт Открыть, а затем щелкните на пункте Решение или проект. Появится диалоговое окно Открыть проект. Перейдите в папку `\Microsoft Press\VCSBS\Chapter 2\PrimitiveDataTypes`, которая находится в вашей папке документов. Выберите файл решения `PrimitiveDataTypes`, а затем щелкните на кнопке Открыть. Решение загрузится, и в обозревателе решений будет показан проект `PrimitiveDataTypes`.



ПРИМЕЧАНИЕ Имена файлов решений имеют суффикс `.sln`, например `PrimitiveDataTypes.sln`. В решении может содержаться один или несколько проектов. У файлов проектов Visual C# имеется суффикс `.csproj`. Если открыть проект, а не решение, среда Visual Studio 2015 автоматически создаст для него новый файл решения. Если вы не в курсе данной особенности, это может привести к путанице, поскольку в результате может быть случайно создано сразу несколько решений для одного и того же проекта.

Щелкните в меню Отладка на пункте Начать отладку.

В Visual Studio могут появиться несколько предупреждений, которые можно совершенно свободно проигнорировать. (Соответствующие корректировки будут внесены в следующем упражнении.)

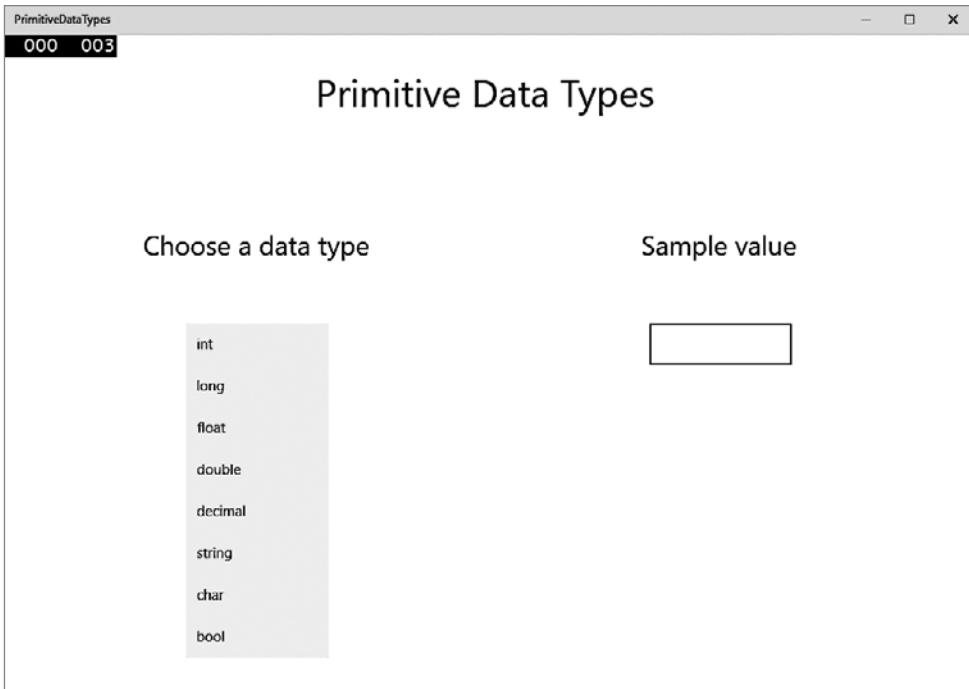


Рис. 2.1

Щелкните в списке Choose A Data Type (Выберите тип данных) (рис. 2.1) на пункте string (Строковое значение). В поле Sample Value (Образец значения) появится текст `forty two`. Щелкните еще раз в списке Choose A Data Type на пункте int (Целочисленное значение). В поле Sample Value появится значение `to do`, показывающее, что инструкции для отображения значения типа `int` еще должны быть написаны. Щелкните на каждом пункте списка типа данных. Убедитесь, что код для типов `double` и `bool` еще не создан. Вернитесь в среду Visual Studio 2015, а затем щелкните в меню Отладка на пункте Остановить отладку. Для остановки отладки можно также закрыть окно.

Использование в коде простых типов данных

Раскройте в обозревателе решений проект PrimitiveDataTypes, а затем дважды щелкните на имени файла `MainPage.xaml`. В окне конструктора появится форма для приложения.



СОВЕТ Если ваш экран недостаточно велик для отображения всей формы, можно воспользоваться функцией увеличения и уменьшения, оперируя комбинациями клавиш **Ctrl+Alt+=** и **Ctrl+Alt+-** или выбрав размер из раскрывающегося списка Масштаб (Zoom) в левом нижнем углу окна конструктора.

Прокрутите в панели XAML код до того места, где расположена разметка для элемента управления **ListBox**. В этом элементе отображается перечень типов данных из левой части формы, он имеет следующий вид (ряд свойств из этого текста был изъят):

```
<ListBox x:Name="type" ... SelectionChanged="typeSelectionChanged">
    <ListBoxItem>int</ListBoxItem>
    <ListBoxItem>long</ListBoxItem>
    <ListBoxItem>float</ListBoxItem>
    <ListBoxItem>double</ListBoxItem>
    <ListBoxItem>decimal</ListBoxItem>
    <ListBoxItem>string</ListBoxItem>
    <ListBoxItem>char</ListBoxItem>
    <ListBoxItem>bool</ListBoxItem>
</ListBox>
```

Элемент управления **ListBox** отображает каждый тип данных в виде отдельного элемента **ListBoxItem**. Если при работающем приложении пользователь щелкает на элементе списка, происходит событие **SelectionChanged** (оно немного похоже на показанное в главе 1 событие **Click**, происходящее, когда пользователь щелкает на кнопке). Можно увидеть, что в таком случае **ListBox** вызывает метод **typeSelectionChanged**. Этот метод определен в файле **MainPage.xaml.cs**.

Щелкните в меню **Вид** на пункте **Код**. Откроется окно редактора, в котором будет показан файл **MainPage.xaml.cs**.



ПРИМЕЧАНИЕ Не забудьте, что для доступа к коду можно также воспользоваться обозревателем решений. Щелкните на стрелке слева от имени файла **MainPage.xaml**, чтобы раскрылся узел, а затем дважды щелкните на имени файла **MainPage.xaml.cs**.

Найдите в окне редактора метод **typeSelectionChanged**.



ПРИМЕЧАНИЕ Чтобы найти нужное место в вашем проекте, следует в меню **Правка** (**Edit**) щелкнуть на пункте **Найти** и заменить (**Find And Replace**), а затем щелкнуть на пункте **Быстрый поиск** (**Quick Find**). В правом верхнем углу окна редактора открывается меню. В текстовом поле этого контекстного меню наберите имя искомого объекта, а затем щелкните на кнопке **Найти далее** (она имеет вид направленной вправо стрелки, расположенной сразу за текстовым полем) (рис. 2.2).

Изначально поиск ведется без учета регистра символов. Если нужно провести поиск с учетом регистра, щелкните на кнопке **Учитывать регистр** (с изображением букв **Aa**), расположенной ниже текста, задающего поиск.

Для отображения диалогового окна быстрого поиска вместо использования меню **Правка** можно также воспользоваться комбинацией клавиш **Ctrl+F**. А для отображения окна быстрой замены можно воспользоваться комбинацией клавиш **Ctrl+H**.



Рис. 2.2

В качестве альтернативы использованию функции быстрого поиска установить местонахождение методов в классе можно также с помощью раскрывающегося списка элементов класса, расположенного чуть выше окна редактора справа (рис. 2.3).

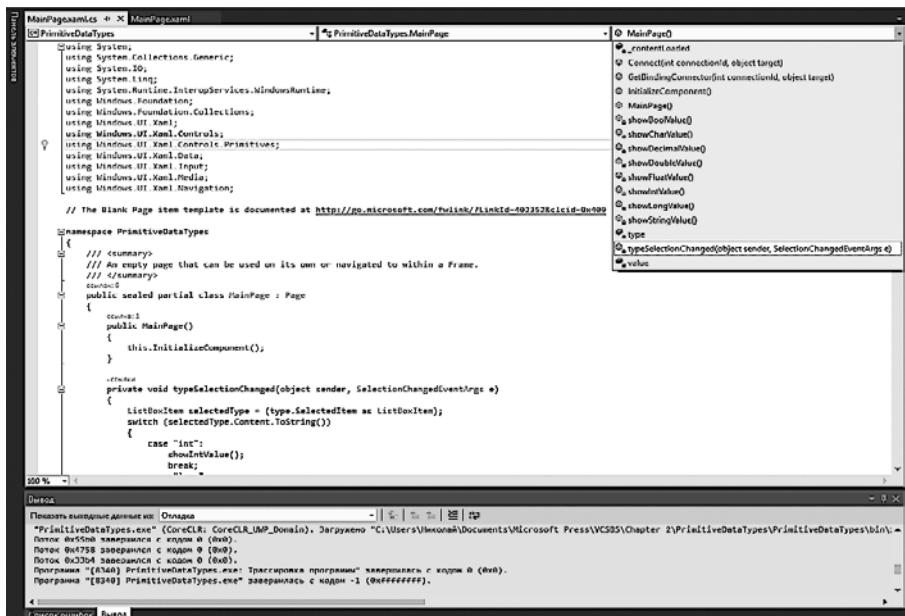


Рис. 2.3

Все методы класса вместе с переменными и другими составляющими класса отображаются в поле раскрывающегося списка элементов класса. (Подробнее эти элементы будут рассматриваться в следующих главах.) Щелкните в этом списке на методе `typeSelectionChanged`, и курсор переместится непосредственно на имеющийся в классе метод `typeSelectionChanged`.

При наличии опыта программирования на других языках вы могли бы уже догадаться, как работает метод `typeSelectionChanged`, если же такого опыта у вас нет, разобраться в коде поможет материал главы 4 «Использование инструкций принятия решений». А сейчас вам нужно лишь усвоить, что при щелчке пользователя на составляющей элемента управления `ListBox` подробности этой составляющей передаются данному методу, который затем использует эту информацию для определения того, что произойдет далее. Например, если пользователь щелкнет на составляющей `float`, этот метод вызовет другой метод по имени `showFloatValue`.

Прокрутите код, чтобы найти метод `showFloatValue`, имеющий следующий вид:

```
private void showFloatValue()
{
    float floatVar;
    floatVar = 0.42F;
    value.Text = floatVar.ToString();
}
```

Основу метода составляют три инструкции. Первая инструкция объявляет переменную `floatVar`, относящуюся к типу `float`. Вторая инструкция присваивает `floatVar` значение `0.42F`.



ВНИМАНИЕ Буква F представляет собой суффикс типа, указывающий, что 0,42 должно рассматриваться как значение числа с плавающей точкой. Если забыть поставить F, значение 0,42 будет рассматриваться как число с двойной точностью и ваша программа не пройдет компиляцию, поскольку вы не можете присвоить значение одного типа переменной другого типа без написания дополнительного кода, — в этом C# проявляет особую строгость.

Третья инструкция выводит значение этой переменной в поле значений формы. Обратите внимание на эту инструкцию. Как уже было показано в главе 1, способ вывода информации в текстовое поле заключается в установке значения его свойства `Text` (в главе 1 это было проделано с помощью XAML). Также эту задачу можно решить чисто программными средствами, чем мы здесь и занимаемся. Заметьте, что обращение к свойству объекта происходит с использованием той же системы записи с точкой, которую вы видели при запуске метода. (Помните `Console.WriteLine` из главы 1?) Кроме этого, данные, помещаемые в свойство `Text`, должны быть строкой, а не числом. При попытке присвоить свойству `Text` число ваша программа не пройдет компиляцию. К счастью, .NET Framework предоставляет помощь в виде метода `ToString`.

У каждого типа данных в .NET Framework имеется метод `ToString`. Задача метода `ToString` заключается в преобразовании объекта в его строковое представление. Метод `showFloatValue` использует метод `ToString` переменной `float` объекта `floatVar` для создания строковой версии значения этой переменной. Затем можно будет без всякого опасения присвоить эту строку свойству `Text` значения текстового поля. При создании собственных типов данных и классов можно определять свои реализации метода `ToString` для указания того, как ваш класс должен быть представлен в виде строки. Как создаются собственные классы, вы узнаете в главе 7 «Создание классов и объектов и управление ими».

Найдите в окне редактора метод `showIntValue`:

```
private void showIntValue()
{
    value.Text = "to do";
}
```

Этот метод вызывается, когда в списке выбора типа данных делается щелчок на поле `int`. В начале кода метода `showIntValue` в новой строке после открывающей фигурной скобки наберите следующие две инструкции, выделенные жирным шрифтом:

```
private void showIntValue()
{
    int intVar;
    intVar = 42;
    value.Text = "to do";
}
```

Первая инструкция создает переменную по имени `intVar`, в которой может содержаться значение типа `int`. Вторая инструкция присваивает этой переменной значение 42. В исходной инструкции данного метода измените строку "to do" на `intVar.ToString()`.

Теперь этот метод должен приобрести следующий вид:

```
private void showIntValue()
{
    int intVar;
    intVar = 42;
    value.Text = intVar.ToString();
}
```

Щелкните в меню Отладка на пункте Начать отладку. Форма появится еще раз. В списке выбора типа данных выберите тип `int`. Убедитесь в том, что в текстовом поле Sample Value выводится значение 42. Вернитесь в Visual Studio, а затем щелкните в меню Отладка на пункте Остановить отладку.

Найдите в окне редактора метод `showDoubleValue`. Отредактируйте этот метод, чтобы он приобрел точно такой же вид, как в следующем коде, и содержал инструкции, выделенные жирным шрифтом:

```
private void showDoubleValue()
{
    double doubleVar;
    doubleVar = 0.42;
    value.Text = doubleVar.ToString();
}
```

Этот код похож на код метода `showIntValue`, за исключением того, что он создает переменную `doubleVar`, содержащую значение с двойной точностью, и ей присваивается значение 0,42.

Найдите в окне редактора метод `showBoolValue`. Отредактируйте этот метод, чтобы он приобрел следующий вид:

```
private void showBoolValue()
{
    bool boolVar;
    boolVar = false;
    value.Text = boolVar.ToString();
}
```

Этот код также похож на код предыдущего примера, за исключением того, что переменная `boolVar` может содержать только булево значение, `true` или `false`. В данном случае присваиваемым значением является `false`.

Щелкните в меню Отладка на пункте Начать отладку. В списке выбора типов данных выберите типы `float`, `double` и `bool`. В каждом случае проверьте, что в текстовом поле Sample Value выводится правильное значение.

Вернитесь в Visual Studio, а затем щелкните в меню Отладка на пункте Остановить отладку.

Использование арифметических операторов

C# поддерживает обычные арифметические операции, известные вам с детства: знак «плюс» (+) для сложения, знак «минус» (-) для вычитания, звездочка (*) для умножения и прямой слеш (/) для деления. Символы +, -, * и / называются операторами, потому что они выполняют операции над значениями для создания новых значений. В следующем примере переменная `moneyPaidToConsultant` содержит в конечном итоге произведение 750 (дневной ставки) и 20 (количество рабочих дней консультанта):

```
long moneyPaidToConsultant;
moneyPaidToConsultant = 750 * 20;
```



ПРИМЕЧАНИЕ Значения, над которыми оператор производит свое действие, называются операндами. В выражении `750 * 20`, символ `*` является оператором, а `750` и `20` — операндами.

Операторы и типы

Не все операторы применимы ко всем типам данных. Какие именно операторы могут быть применены к значению, зависит от типа значения. Например, все математические операторы можно применять к значениям типа `char`, `int`, `long`, `float`, `double` или `decimal`. Но за исключением оператора, обозначаемого знаком «плюс» (`+`), арифметические операторы нельзя применять к значениям типа `string` и никакие из этих операторов нельзя использовать со значениями типа `bool`. Следовательно, показанная далее инструкция недопустима, поскольку для типа данных `string` оператор «минус» не поддерживается (вычитание одной строки из другой не имеет никакого смысла):

```
// ошибка в ходе компиляции  
Console.WriteLine("Gillingham" - "Forest Green Rovers");
```

Но для объединения строковых значений можно использовать оператор `+`. При этом нужно проявлять осмотрительность, поскольку могут быть получены неожиданные результаты. Например, следующая инструкция выведет на консоль `431` (а не `44`):

```
Console.WriteLine("43" + "1");
```



СОВЕТ Если нужно выполнить арифметическое вычисление над значениями, содержащимися в виде строк, среда .NET Framework предоставляет метод под названием `Int32.Parse`, которым можно воспользоваться для преобразования строкового значения в целое число.

Строковая интерполяция

В самой последней версии C# появилось новое функциональное свойство, называемое строковой интерполяцией, переводящее многие варианты использования оператора `+` для объединения строк в разряд устаревших.

Довольно часто объединение строк применяется для создания строковых значений, включающих значения переменных. Пример такого применения мы уже видели при создании графического приложения в главе 1. В метод `okClick` добавлялась следующая строка кода:

```
MessageDialog msg = new MessageDialog("Hello " + userName.Text);
```

Строковая интерполяция позволяет воспользоваться вместо этого следующим синтаксисом:

```
MessageDialog msg = new MessageDialog($"Hello {userName.Text}");
```

Символ \$ в начале строки показывает, что это интерполируемая строка и что любое выражение между символами { и } должно быть вычислено, а результат должен быть вставлен вместо него. Без лидирующего символа \$ строка {username.Text} будет рассматриваться буквально.

Строковая интерполяция более эффективна, чем использование оператора +. (Объединение строк с помощью оператора + с учетом того, каким образом сре-да .NET Framework обрабатывает строки, может потреблять много памяти.) Строковая интерполяция претендует и на то, что в ней легче разобраться и при ее применении труднее допустить ошибку.

Следует также иметь в виду, что тип результата арифметической операции зависит от типа используемых операндов. Например, значением выражения 5.0/2.0 является 2.5, оба операнда относятся к типу double, следовательно, результат также будет типа double. (В C# числовые литералы с десятичной точкой всегда относятся к типу double, а не к типу float, чтобы их обработка велась с максимально возможной точностью.) Но значение выражения 5/2 равно 2. В этом случае оба операнда принадлежат к типу int, следовательно, результат также принадлежит к типу int.

В подобных ситуациях C# всегда производит округление в сторону нуля. При смешивании типов операндов ситуация немного усложняется. Например, выражение 5/2.0 состоит из типов данных int и double. Компилятор C# обнаруживает несоответствие и создает код, преобразующий int в double до выполнения операции. Поэтому результатом операции является значение типа double (2.5). Но несмотря на то что этот код работает, подобное смешивание типов считается порочной практикой.

C# также поддерживает менее известный арифметический оператор — извлечение остатка целочисленного деления, который представлен знаком процента (%). Результатом x % y является остаток после деления значения x на значение y. Следовательно, 9 % 2, к примеру, дает результат 1, поскольку, если 9 разделить на 2, то получится 4, а остаток будет равен 1.



ПРИМЕЧАНИЕ Если вам знаком язык C или C++, то вы знаете, что использовать в этих языках оператор извлечения остатка от деления в отношении значения типов float или double невозможно. Но в C# это правило смягчено. Оператор извлечения остатка от деления можно применять с числовыми значениями любых типов, и результат не обязательно должен быть целым числом. Например, результат выражения 7.0 % 2.4 равен 2.2.

Числовые типы и бесконечные значения

В C# имеется еще ряд относящихся к числам особенностей, о которых вам следует знать. Например, результатом деления любого числа на нуль является бесконечность, что выходит за рамки определения типов `int`, `long` и `decimal`, следовательно, вычисление такого выражения, как `5/0`, приводит к возникновению ошибки. Но у типов `double` и `float` имеется специальное значение, которое может представлять бесконечность, и значением выражения `5.0/0.0` будет `Infinity`. В этом правиле есть одно исключение — значение выражения `0.0/0.0`. Обычно при делении нуля на какое-нибудь число получается нулевой результат, но при делении любого числа на нуль получается бесконечность. Выражение `0.0/0.0` приводит к парадоксу — значение должно быть нулем и бесконечностью одновременно. Для данной ситуации в C# имеется еще одно специальное значение, которое называется `NaN` и означает not a number (не число). Следовательно, при вычислении выражения `0.0/0.0` результат будет иметь значение `NaN`.

Значения `NaN` и `Infinity` распространяются и на выражения. Если вычисляется выражение `10 + NaN`, результатом будет `NaN`, а если вычисляется выражение `10 + Infinity`, результатом будет `Infinity`. Результатом вычисления выражения `Infinity * 0` будет `NaN`.

Исследование арифметических операторов

В следующем примере показан способ использования арифметических операторов в отношении значений типа `int`.

Работа с проектом MathsOperators

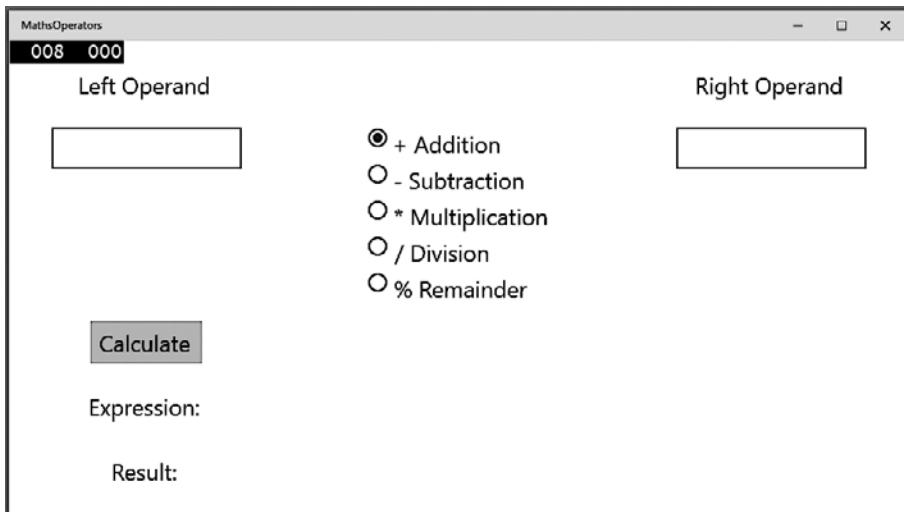
Откройте в Visual Studio 2015 проект `MathsOperators`, который находится в папке `\Microsoft Press\VCBS\Chapter 2\MathsOperators` вашей папки документов.

Щелкните в меню Отладка на пункте Начать отладку. Появится следующая форма (рис. 2.4).

Наберите в поле `Left Operand` (Левый operand) число `54`, а в поле `Right Operand` (Правый operand) — число `13`. Теперь к значениям в текстовых полях можно применить любой оператор. Щелкните на варианте `- Subtraction` (Вычитание), а затем на кнопке `Calculate` (Вычислить). Текст в поле `Expression` (Выражение) изменится на `54 - 13`, но в поле `Result` (Результат) появится `0`, что, несомненно, неправильно.

Щелкните на варианте `/ Division` (Деление), а затем на кнопке `Calculate`. Текст в поле `Expression` изменится на `54/13`, и в поле `Result` опять появится `0`.

Щелкните на варианте `/ Remainder` (Получение остатка от деления), а затем на кнопке `Calculate`. Текст в поле `Expression` изменится на `54%13`, а в поле `Result` опять

**Рис. 2.4**

появится 0. Проверьте другие комбинации чисел и операторов, и вы увидите, что все они на данный момент при вычислении приводят к появлению значения 0.



ПРИМЕЧАНИЕ Если в любое из полей operandов ввести нецелочисленное значение, приложение обнаружит ошибку и выведет на экран сообщение о том, что введенная строка имеет неверный формат (Input string was not in a correct format). Дополнительные сведения о перехвате ошибок и их обработке, а также об исключениях будут даны в главе 6 «Обработка ошибок и исключений».

Завершив перечисленные действия, вернитесь в Visual Studio, а затем щелкните в меню Отладка на пункте Остановить отладку.

Как вы уже могли догадаться, пока никакие вычисления в приложении MathsOperators не реализованы. Этот недостаток будет исправлен в следующем упражнении.

Выполнение вычислений в приложении MathsOperators

Выведите в окно конструктора форму MainPage.xaml (дважды щелкните на имени файла MainPage.xaml, указанного в обозревателе решений в проекте MathsOperators). В меню Вид выберите пункт Другие окна (Other Windows), после чего щелкните на пункте Структура документа (Document Outline).

Появится окно Структура документа, в котором будут показаны имена и типы имеющихся в форме элементов управления. Это окно обеспечивает простой способ поиска и выбора элементов управления в сложной форме. Эти элементы

управления выстроены в определенной иерархии, начиная с элемента `Page`, составляющего форму. Как упоминалось в главе 1, страница приложения универсальной платформы Windows (Universal Windows Platform (UWP)) содержит элемент управления `Grid` и другие элементы, помещенные внутрь этого `Grid`-элемента. Если в окне структуры документа раскрыть узел `Grid`, появятся другие элементы управления, которые начнутся с еще одного элемента `Grid` (внешний `Grid` выступает в роли структуры, а внутренний `Grid` содержит элементы управления, которые вы видите в форме). Если раскрыть внутренний `Grid`-элемент, вы увидите каждый элемент управления формы (рис. 2.5).

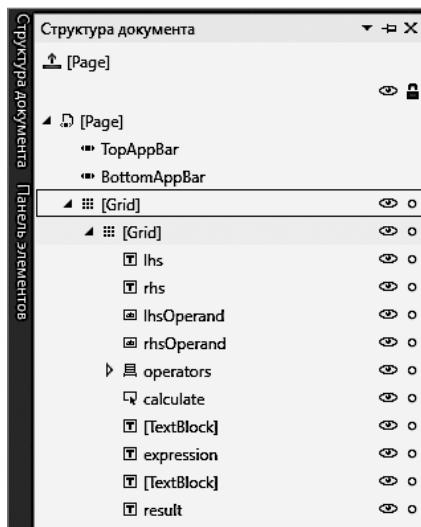


Рис. 2.5

Если щелкнуть на любом из этих элементов управления, в окне конструктора будет выделен соответствующий ему элемент. Сообразно этому, если выбрать элемент управления в окне конструктора, соответствующий элемент будет выбран в окне структуры документа. (Чтобы посмотреть на это в действии, закрепите окно структуры документа на месте, отменив выбор режима скрытия окна, щелкнув для этого на кнопке *Автоматически скрывать* (Auto Hide) в правом верхнем углу окна структуры документа.)

Щелкните в форме на двух элементах управления типа `TextBox`, в которых пользователь набирает числа. Проверьте в окне структуры документа, что они носят названия `lhsOperand` и `rhsOperand`. При запуске формы в свойстве `Text` каждого из этих элементов управления содержатся значения, введенные пользователем.

В нижней части формы проверьте, что элемент управления типа `TextBlock`, используемый для вывода вычисляемого значения, называется `expression` и что

элемент управления типа `TextBlock`, используемый для отображения результата вычисления, называется `result`.

Закройте окно структуры документа. Щелкните в меню Вид на пункте Код, чтобы в окне редактора отобразился код файла `MainPage.xaml.cs`.

Найдите в редакторе метод `addValues`. Он имеет следующий вид:

```
private void addValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Add rhs to lhs and store the result in outcome
    expression.Text = $"{lhsOperand.Text} + {rhsOperand.Text}";
    result.Text = outcome.ToString();
}
```

Первая инструкция в этом методе объявляет переменную типа `int` по имени `lhs` и инициализирует ее с использованием целого числа, соответствующего значению, набранному пользователем в поле `lhsOperand`. Вспомним, что свойство `Text` элемента управления типа `TextBox` содержит строку, но переменная `lhs` относится к типу `int`, поэтому, прежде чем присваивать значение переменной `lhs`, вы должны преобразовать эту строку в целое число. Тип данных `int` предоставляет метод `int.Parse`, который именно этим и занимается.

Вторая инструкция объявляет переменную типа `int` по имени `rhs` и инициализирует ее значением, находящимся в поле `rhsOperand`, после преобразования этого значения в `int`.

Третья инструкция объявляет переменную типа `int` по имени `outcome`.

Далее следует строка комментария, в которой сообщается о том, что вам нужно сложить `rhs` и `lhs` и сохранить результат в `outcome`. Этот фрагмент кода здесь не представлен, и вам нужно его создать, чем мы вскоре и займемся.

Пятая инструкция использует строковую интерполяцию, показывающую, что вычисление было выполнено, и присваивает результат свойству `expression.Text`, что приводит к появлению строки в поле `Expression` этой формы.

Последняя инструкция выводит результат вычисления, присваивая его свойству `Text` поля `Result`. Следует помнить, что свойство `Text` содержит значение типа `string`, а результат вычисления содержит значение типа `int`, поэтому, прежде чем присваивать результат свойству `Text`, значение тип `int` нужно преобразовать в `string`. Именно этим и занимается метод `ToString`, принадлежащий типу `int`.

Добавьте ниже комментария в середине метода `addValues` следующую инструкцию, выделенную жирным шрифтом:

```
private void addValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Add rhs to lhs and store the result in outcome
    outcome = lhs + rhs;
    expression.Text = $"{lhsOperand.Text} + {rhsOperand.Text}";
    result.Text = outcome.ToString();
}
```

Эта инструкция вычисляет выражение `lhs + rhs` и сохраняет результат в переменной `outcome`.

Исследуйте содержимое метода `subtractValues`. Вы увидите, что он соответствует тому же самому шаблону. В него нужно добавить инструкцию для вычисления результата вычитания `rhs` из `lhs` и его сохранения в `outcome`. Добавьте к методу следующую инструкцию, выделенную жирным шрифтом:

```
private void subtractValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Subtract rhs from lhs and store the result in outcome
    outcome = lhs - rhs;
    expression.Text = $"{lhsOperand.Text} - {rhsOperand.Text}";
    result.Text = outcome.ToString();
}
```

Исследуйте содержимое методов `multiplyValues`, `divideValues` и `remainderValues`. В них точно так же не хватает важной инструкции, выполняющей вычисление. Добавьте соответствующие инструкции, выделенные жирным шрифтом:

```
private void multiplyValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Multiply lhs by rhs and store the result in outcome
    outcome = lhs * rhs;
    expression.Text = $"{lhsOperand.Text} * {rhsOperand.Text}";
    result.Text = outcome.ToString();
}

private void divideValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Divide lhs by rhs and store the result in outcome
    outcome = lhs / rhs;
    expression.Text = $"{lhsOperand.Text} / {rhsOperand.Text}";
    result.Text = outcome.ToString();
}
```

```

}

private void remainderValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Work out the remainder after dividing lhs by rhs and store the
    // result in outcome
    outcome = lhs % rhs;
    expression.Text = $"{lhsOperand.Text} % {rhsOperand.Text}";
    result.Text = outcome.ToString();
}

```

Тестирование приложения MathsOperators

Щелкните в меню Отладка на пункте Начать отладку, чтобы произвести сборку и запуск приложения.

Наберите 54 в поле Left Operand и 13 — в поле Right Operand, щелкните на варианте + Addition, а затем на кнопке Calculate. В поле Result должно появиться значение 67.

Щелкните на варианте – Subtraction, а затем на кнопке Calculate. В поле Result должно появиться значение 41.

Щелкните на варианте * Multiplication, а затем на кнопке Calculate. Убедитесь в том, что теперь в поле Result появилось значение 702.

Щелкните на варианте / Division, а затем на кнопке Calculate. Убедитесь в том, что теперь в поле Result появилось значение 4. В реальности $54/13$ равно 4,153846 в периоде, но это не реальность, это язык C#, выполняющий целочисленное деление. Как уже говорилось, при делении одного целого числа на другое ответ должен быть возвращен в виде целого числа.

Щелкните на варианте % Remainder, а затем на кнопке Calculate. Убедитесь в том, что теперь показано значение 2. При работе с целыми числами остаток от деления 54 на 13 равен 2: $(54 - ((54/13) \cdot 13)) = 2$. Дело в том, что вычисление на каждой стадии округляется вниз до целого числа. (Мой учитель математики в средней школе схватился бы за голову, если бы ему сказали, что $(54/13) \cdot 13$ не равно 54!)

Вернитесь в Visual Studio и остановите отладку.

Управление очередностью применения операторов

Очередность определяет порядок, в котором в выражении происходит вычисление с участием тех или иных операторов. Рассмотрим следующее выражение, использующее операторы + и *:

2 + 3 * 4

Это выражение содержит потенциальную неопределенность: что выполняется в первую очередь, сложение или умножение? Порядок выполнения операций играет важную роль, поскольку он изменяет результат.

- Если сначала выполняется сложение, а за ним умножение, результат сложения $(2 + 3)$ формирует левый operand оператора $*$ и результатом всего выражения становится $5 \cdot 4$, что равно 20.
- Если сначала выполняется умножение, а за ним сложение, результат умножения $(3 \cdot 4)$ формирует правый operand оператора $+$ и результатом всего выражения становится $2 + 12$, что равно 14.

В C# мультипликативные операторы $(*, / \text{ и } %)$ имеют приоритет над аддитивными операторами $(+ \text{ и } -)$, следовательно, в таком выражении, как $2 + 3 \times 4$, сначала выполняется умножение, а затем сложение. Поэтому ответ вычисления $2 + 3 \times 4$ равен 14.

Для изменения очередности и принуждения operandов к иной привязке к операторам можно воспользоваться круглыми скобками. Например, в следующем выражении круглые скобки заставляют 2 и 3 быть привязанными к оператору $+$ (выдавая результат 5) и результат сложения формирует левый operand для оператора $*$, чтобы получилось значение 20:

$(2 + 3) * 4$



ПРИМЕЧАНИЕ Под круглыми скобками понимается пара символов $()$. Под фигурными скобками понимается пара символов $\{ \}$. А под квадратными скобками понимается пара символов $[]$.

Использование ассоциативности для вычисления выражений

Приоритетность операторов — всего лишь половина истории. Давайте подумаем: что происходит, когда выражение содержит различные операторы, имеющие одинаковый уровень приоритета? Тут на первый план выступает ассоциативность, выражаящаяся в направлении (левом или правом), в котором вычисляются operandы операторов. Рассмотрим следующее выражение, в котором используются операторы $/$ и $*$:

$4 / 2 * 6$

На первый взгляд в этом выражении присутствует потенциальная неоднозначность. Что выполняется первым, деление или умножение? Уровень приоритета у обоих операторов одинаковый (они оба относятся к мультипликативным

операторам), но порядок их применения в выражении играет важную роль, поскольку вы можете получить два разных результата.

- ❑ Если сначала выполняется деление, результат этого деления ($4/2$) формирует operand оператора $*$, а результатом всего выражения будет $(4/2) \times 6$, или 12.
- ❑ Если сначала выполняется умножение, результат этого умножения (2×6) формирует operand оператора $/$, а результатом всего выражения будет $4/(2 \times 6)$, или 4/12.

В данном случае порядок вычисления определяется ассоциативностью операторов. Оба оператора, $*$ и $/$, обладают левой ассоциативностью, что означает, что эти operandы вычисляются слева направо. В данном случае $4/2$ будет вычислено до умножения на 6, что даст результат 12.

Ассоциативность и оператор присваивания

В C# знак равенства ($=$) является оператором. Все операторы возвращают значение на основе своих operandов. Оператор присваивания $=$ исключением не является. Он получает два operandов: operand справа от него вычисляется, а затем сохраняется в operandе, расположенному слева от него. Значением оператора присваивания является значение, присвоенное левому operandу. Например, в следующей инструкции присваивания значение, возвращаемое оператором присваивания, равно 10, и оно же является значением, присваиваемым переменной `myInt`:

```
int myInt;  
myInt = 10; // значением выражения присваивания является 10
```

Сейчас вы можете подумать: все это, конечно, красиво и заумно, но какая от этого польза? Дело в том, что оператор присваивания возвращает значение и это же самое значение можно использовать для другого экземпляра инструкции присваивания:

```
int myInt;  
int myInt2;  
myInt2 = myInt = 10;
```

Значение, присвоенное переменной `myInt2`, является значением, которое было присвоено переменной `myInt`. Инструкция присваивания дает одно и то же значение обеим переменным. Этот прием пригодится, если нужно инициализировать сразу несколько переменных одним и тем же значением. Всем, кто читает ваш код, будет абсолютно понятно, что у всех переменных должно быть одно и то же значение:

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

Рассматривая данный вопрос, можно прийти к заключению, что оператор присваивания имеет ассоциативность справа налево. Первым происходит самое правое присваивание, и присвоенное значение распространяется по переменным справа налево. Если какая-нибудь из переменных ранее уже имела значение, оно переписывается присваиваемым значением.

Но к данной конструкции нужно относиться весьма осмотрительно. Одной из часто совершаемых ошибок является то, что новички в программировании на C# пытаются объединить такое применение оператора присваивания с объявлением переменных. Например, можно предположить, что следующий код создаст и инициализирует три переменные одним и тем же значением (10):

```
int myInt, myInt2, myInt3 = 10;
```

Это вполне приемлемый код C# (поскольку он проходит компиляцию). Он объявляет переменные `myInt`, `myInt2` и `myInt3` и инициализирует переменную `myInt3` значением 10. Но он не инициализирует `myInt` или `myInt2`. Если попытаться использовать `myInt` или `myInt2` в подобном выражении:

```
myInt3 = myInt / myInt2;
```

компилятор выдаст ошибки, связанные с использованием локальных переменных, которым не присвоены значения:

```
Use of unassigned local variable 'myInt'  
Use of unassigned local variable 'myInt2'
```

Увеличение и уменьшение значений переменных на единицу

Если к значению переменной нужно прибавить 1, можно, как показано далее, воспользоваться оператором `+`:

```
count = count + 1;
```

Но добавление 1 к переменной (операция инкремента) встречается настолько часто, что в C# для этой цели предоставляется собственный оператор `++`. Чтобы увеличить значение переменной `count` на 1, можно написать следующую инструкцию:

```
count++;
```

Аналогично этому в C# для операции декремента предоставляется оператор `--`, которым можно воспользоваться для вычитания 1 из значения переменной:

```
count--;
```

Операторы `++` и `--` являются унарными, а это означает, что они получают только один операнд. Они имеют одинаковый приоритет и обладают левой ассоциативностью (вычисляются слева направо).

Префиксная и постфиксная формы

Операторы инкремента (`++`) и декремента (`--`) необычны тем, что их можно ставить либо перед, либо после переменной. Помещение символов оператора перед переменной называется префиксной формой оператора, а их помещение после переменной — постфиксной формой. Рассмотрим примеры:

```
count++; // постфиксный инкремент
++count; // префиксный инкремент
count--; // постфиксный декремент
--count; // префиксный декремент
```

Какая бы форма, префиксная или постфиксная, оператора `++` или `--` ни использовалась, для переменной нет никакой разницы, она все равно подвергнется инкрементированию или декрементированию. Например, если вы напишете `count++`, значение `count` увеличится на единицу, и если вы напишете `++count`, оно также увеличится на единицу. Зная об этом, вы, наверное, спросите: к чему эти два способа написания одной и той же операции? Чтобы понять смысл ответа, нужно вспомнить, что `++` и `--` являются операторами и что все операторы используются для вычисления выражения, имеющего значение. Значение, возвращаемое `count++`, является значением `count` до добавления к нему единицы, а значение, возвращаемое `++count`, является значением `count` после добавления к нему единицы. Рассмотрим пример:

```
int x;
x = 42;
Console.WriteLine(x++); // x теперь равен 43, а на консоль выводится 42
x = 42;
Console.WriteLine(++x); // x теперь равен 43, и на консоль выводится 43
```

Чтобы запомнить, что происходит с operandами, нужно посмотреть на порядок следования элементов (операнда и оператора) в префиксном или постфиксном выражении. В выражении `x++` сначала стоит переменная, следовательно, ее значение используется в качестве значения выражения до увеличения `x` на единицу. В выражении `++x` сначала стоит оператор, следовательно, его операция выполняется до того, как `x` вычисляется как результат.

Чаще всего эти операторы используются в инструкциях `while` и `do`, которые представлены в главе 5 «Использование инструкций составного присваивания и итераций». Если операторы инкремента и декремента используются изолированно, придерживайтесь постфиксной формы — и будьте в этом последовательны.

Объявление неявно типизированных локальных переменных

Ранее в этой главе было показано, что переменная объявляется путем указания типа данных и идентификатора:

```
int myInt;
```

Также упоминалось о том, что перед попыткой применения переменной ей нужно присвоить значение. Объявить и инициализировать переменную можно в одной и той же инструкции, как показано в следующей строке кода:

```
int myInt = 99;
```

Или же можно сделать это как в следующей строке кода, при условии, что `myOtherInt` является инициализированной целочисленной переменной:

```
int myInt = myOtherInt * 99;
```

Теперь вспомните, что значение, присваиваемое переменной, должно быть того же типа, что и переменная. Например, значение типа `int` может быть присвоено переменной с указанным типом данных `int`. Компилятор C# способен быстро определять тип выражения, используемого для инициализации переменной, и указывать на то, что оно не соответствует типу переменной. Как показано в следующих примерах, путем использования ключевого слова `var` на месте типа компилятор C# можно также попросить определить тип переменной из выражения и воспользоваться этим типом при объявлении переменной:

```
var myVariable = 99;
var myOtherVariable = "Hello";
```

Переменные `myVariable` и `myOtherVariable` называются неявно типизированными переменными. Ключевое слово `var` заставляет компилятор установить тип переменных из типов выражений, используемых для их инициализации. В этих примерах `myVariable` имеет тип `int`, а `myOtherVariable` относится к типу `string`. Но вам важно понять, что это пригодится только для объявления переменных и что после того, как переменная была объявлена, ей можно присваивать только значения предполагаемого типа. К примеру, далее в программе вы не сможете присваивать переменной `myVariable` значения типов `float`, `double` или `string`. Нужно также усвоить, что вы можете использовать ключевое слово `var` только тогда, когда для инициализации переменной указывается выражение. Следующее выражение считается недопустимым и вызывает ошибку компиляции:

```
var yetAnotherVariable; // Ошибка – компилятор не может установить тип
```



ВНИМАНИЕ Тем, кому в прошлом приходилось заниматься программированием на Visual Basic, должен быть знаком тип Variant, который можно использовать для хранения в переменной значения любого типа. Я акцентирую ваше внимание здесь и сейчас на том, что вы должны забыть все, что когда-либо изучали о переменных типа Variant при программировании на Visual Basic. Хотя ключевые слова var и Variant выглядят похожими, они означают совершенно разные вещи. При объявлении переменной в C# с использованием ключевого слова var тип значений, присваиваемых переменной, не может изменяться и должен быть таким же, как при ее инициализации.

Если вы педант, то, наверное, уже поскрипываете зубами, недоумевая, почему разработчики такого чистого языка, каким является C#, позволили вкрасться в него такой особенности, как использование ключевого слова var. Ведь это похоже на оправдание чрезмерной лени у определенной части программистов и может усложнить понимание того, что делает программа, или отслеживание допущенной ошибки (и даже с легкостью может способствовать появлению новых ошибок в коде вашей программы). Но поверьте мне, что var находится в C# на своем законном месте, в чем вы сможете убедиться, изучая следующие главы. И тем не менее на данный момент мы будем придерживаться использования явно типизированных переменных, прибегая к неявной типизации в случае крайней необходимости.

Выходы

В данной главе вы увидели, как создавать и использовать переменные, и узнали о некоторых наиболее распространенных типах данных, доступных для переменных в C#. Вы также узнали об идентификаторах. Кроме того, вы воспользовались несколькими операторами для создания выражений и узнали о том, как в зависимости от уровней приоритета и ассоциативности операторов определяется способ вычисления выражений.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 3.

Если хотите выйти из среды Visual Studio 2015, то в меню Файл щелкните на пункте Выход (Exit). Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке Да (Yes) и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Объявить переменную	Напишите название типа данных, затем имя переменной, после чего поставьте точку с запятой, например: <code>int outcome;</code>

Чтобы	Сделайте следующее
Объявить переменную и присвоить ей начальное значение	Напишите название типа данных, затем имя переменной, после чего поставьте оператор присваивания и начальное значение. Завершите инструкцию точкой с запятой, например: <code>int outcome = 99;</code>
Изменить значение переменной	Напишите имя переменной слева, за ним поставьте оператор присваивания, далее выражение, вычисляющее новое значение, и завершите инструкцию точкой с запятой, например: <code>outcome = 42;</code>
Создать строку, представляющую значение, хранящееся в переменной	Вызовите в отношении переменной метод <code>ToString</code> , например: <code>int intVar = 42; string stringVar = intVar.ToString();</code>
Преобразовать строку в целое число	Вызовите метод <code>System.Int32.Parse</code> , например: <code>string stringVar = "42"; int intVar = System.Int32.Parse(stringVar);</code>
Изменить очередность применения операторов	Воспользуйтесь круглыми скобками для принудительного задания порядка вычисления, например: <code>(3 + 4) * 5</code>
Присвоить одно и то же значение нескольким переменным	Воспользуйтесь инструкцией присваивания, перечисляющей все переменные, например: <code>myInt4 = myInt3 = myInt2 = myInt = 10;</code>
Увеличить или уменьшить значение переменной на единицу	Воспользуйтесь оператором <code>++</code> или <code>--</code> , например: <code>count++;</code>

3

Создание методов и применение областей видимости

Прочитав эту главу, вы научитесь:

- объявлять и вызывать методы;
- передавать методу информацию;
- возвращать информацию из метода;
- определять локальную область видимости и область видимости класса;
- пользоваться встроенным отладчиком для пошагового выполнения кода внутри выполняемых методов и за их пределами.

В главе 2 «Работа с переменными, операторами и выражениями» вы узнали, как объявляются переменные, как с использованием операторов создаются выражения и как на способ вычисления выражений, содержащих несколько операторов, влияют приоритетность и ассоциативность. В этой главе будут изучаться методы. Будет показано, как объявляются и вызываются методы, как для передачи методу информации используются аргументы и параметры и как с помощью инструкции `return` из метода возвращается информация. Вы также увидите, как с помощью встроенного отладчика Microsoft Visual Studio 2015 осуществляется пошаговое выполнение кода внутри и за пределами методов. Эта информация вам пригодится, если методы станут работать неожиданным образом и возникнет необходимость проведения трассировки их выполнения. И наконец, вы научитесь объявлять методы, получающие дополнительные параметры, и вызывать методы путем использования поименованных аргументов.

Создание методов

Метод представляет собой именованную последовательность инструкций. Если вам уже приходилось заниматься программированием на таких языках, как C, C++ или Microsoft Visual Basic, то вы поймете, что метод похож на функцию или подпрограмму. У метода есть имя и тело. Имя метода должно представлять собой идентификатор со смысловым значением, указывающим на общую цель метода (например, для метода, вычисляющего налог на прибыль, можно выбрать имя `calculateIncomeTax`). Тело метода содержит инструкции, выполняемые при его запуске. Кроме этого, методам могут передаваться данные для обработки, а также методы могут возвращать информацию, которая обычно является результатом их работы. Методы являются основным и весьма эффективным механизмом языка.

Объявление метода

Для объявления метода в C# используется следующий синтаксис:

```
returnType methodName ( parameterList )
{
    // тело метода, куда помещаются инструкции
}
```

Объявление состоит из следующих элементов:

- ❑ `returnType` — это название типа, указывающее на разновидность информации, возвращаемой методом в качестве результата его работы. Указываться может любой тип, например `int` или `string`. Если создается метод, не возвращающий информацию, то вместо типа возвращаемого значения нужно использовать ключевое слово `void` (пустой);
- ❑ `methodName` — это имя, используемое для вызова метода. Имена методов следуют тем же правилам идентификации, что и имена переменных. Например, `addValues` является приемлемым именем метода, а `add$Values` — нет. Впредь для назначения имен методам вам нужно будет следовать соглашению о смешанном регистре (`camelCase`), например `displayCustomer`;
- ❑ `parameterList` — это необязательный список, описывающий типы и имена элементов информации, которые могут передаваться в метод для их обработки. Параметры записываются между открывающей и закрывающей круглыми скобками, `()`, в таком же виде, как и при объявлении переменных, с названием типа, за которым следует имя параметра. Если у создаваемого метода имеется два и более параметра, их нужно отделять друг от друга запятыми;

- инструкций тела метода, которые представляют собой строки кода, запускаемые при вызове метода. Они заключены между открывающей и закрывающей фигурными скобками, { }.



ВНИМАНИЕ Если вам уже приходилось программировать на C, C++ и Visual Basic, то вы могли заметить, что глобальные методы в C# не поддерживаются. Все методы нужно создавать внутри класса, в противном случае ваш код не пройдет компиляцию.

Рассмотрим определение метода по имени `addValues`, возвращающего результат типа `int` и имеющего два `int`-параметра, `leftHandSide` и `rightHandSide`:

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
    // тело метода, куда помещаются инструкции
    // ...
}
```



ПРИМЕЧАНИЕ Тип любого параметра, а также тип возвращаемого методом значения нужно указывать явным образом. Ключевое слово `var` использовать нельзя.

Рассмотрим определение метода по имени `showResult`, не возвращающего значение и имеющего всего один параметр типа `int` под названием `answer`:

```
void showResult(int answer)
{
    // ...
}
```

Обратите внимание на использование ключевого слова `void`, указывающего на то, что метод ничего не возвращает.



ВНИМАНИЕ Если вы знакомы с Visual Basic, то должны заметить, что в C# не используются разные ключевые слова, отличающие метод, возвращающий значение (функцию), от метода, не возвращающего значение (от процедуры или подпрограммы). В C# следует указывать либо тип возвращаемого значения, либо ключевое слово `void`.

Возвращение данных из метода

Если нужно, чтобы метод возвращал информацию (то есть чтобы тип его возвращаемого значения не обозначался как `void`), вы должны включить в тело метода в конце обработки данных инструкцию `return`. Эта инструкция состоит из ключевого слова `return`, за которым следуют выражение, определяющее возвращаемое значение, и ставится точка с запятой. Тип выражения должен совпадать с типом, указанным при объявлении метода. Например, если метод

возвращает значение типа `int`, значит, инструкция `return` должна возвращать `int`, в противном случае ваша программа не откомпилируется. Рассмотрим пример метода с инструкцией `return`:

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
    return leftHandSide + rightHandSide;
}
```

Обычно инструкция `return` размещается в конце метода, поскольку она заставляет метод завершить работу и вернуть управление инструкции, вызвавшей метод, о чем будет рассказано далее. Любые инструкции, оказавшиеся после инструкции `return`, не выполняются (если инструкции помещены после инструкции `return`, компилятор выдает соответствующее предупреждение).

Если методу не нужно возвращать информацию (то есть его тип возвращаемого значения обозначен как `void`), можно воспользоваться вариантом инструкции `return`,зывающим немедленный выход из метода. При этом сразу же после ключевого слова `return` следует поставить точку с запятой, например:

```
void showResult(int answer)
{
    // вывод на экран ответа
    Console.WriteLine($"The answer is {answer}");
    return;
}
```

Если метод ничего не возвращает, можно также вообще опустить инструкцию `return`, поскольку он завершает свою работу автоматически, как только встретится закрывающая фигурная скобка. Несмотря на распространенность подобного приема, применять его не рекомендуется.

Использование метода-выражения

Некоторые методы могут быть очень простыми, выполняющими только одну задачу или возвращающими результаты вычисления без применения какой-либо дополнительной логики. Для методов, содержащих только одно выражение, в C# поддерживается упрощенная форма. Также эти методы могут принимать параметры и возвращать значения и работают точно так же, как методы, с которыми вы уже знакомы. В следующих примерах кода показаны упрощенные версии методов `addValues` и `showResult`, записанные в виде методов-выражений:

```
int addValues(int leftHandSide, int rightHandSide) => leftHandSide + rightHandSide;
void showResult(int answer) => Console.WriteLine($"The answer is {answer}");
```

Основное отличие от обычных методов состоит в использовании для ссылки на выражение, формирующее тело метода, оператора `=>` и в отсутствии инструкции `return`. В качестве возвращаемого значения используется значение выражения, а если выражение не возвращает значение, значит, метод объявляется с использованием ключевого слова `void`.

Функционально между использованием обычного метода и метода-выражения нет никакой разницы, последний из них — это не более чем синтаксическое соглашение. Но далее в книге будут приводиться примеры, в которых методы-выражения позволят четче выразить суть программы, освобождая ее от множества ненужных символов `{` и `}` и упрощая чтение кода.

В следующем упражнении будет исследована еще одна версия проекта `MathsOperators` из главы 2. Эта версия была усовершенствована за счет разумного использования ряда небольших методов. Такой способ разбиения кода упрощает его понимание и дальнейшее сопровождение.

Исследование определений методов

Откройте в среде Visual Studio 2015 проект `Methods`, который находится в папке `\Microsoft Press\VCSBS\Chapter 3\Methods` вашей папки документов.

Щелкните в меню Отладка на пункте Начать отладку. Среда Visual Studio 2015 проведет сборку и запуск приложения. Его внешний вид должен быть таким же, как и у приложения из главы 2. Ознакомьтесь еще раз с приложением и порядком его работы, а затем вернитесь в Visual Studio. Щелкните в меню Отладка на пункте Остановить отладку.

Вызовите в окно редактора код файла `MainPage.xaml.cs` (для этого в обозревателе решений раскройте узел `MainPage.xaml`, а затем дважды щелкните на имени файла `MainPage.xaml.cs`). Найдите метод `addValues`, который должен иметь следующий вид:

```
private int addValues(int leftHandSide, int rightHandSide)\n{\n    expression.Text = $"{leftHandSide} + {rightHandSide}";\n    return leftHandSide + rightHandSide;\n}
```



ПРИМЕЧАНИЕ Пусть присутствие ключевого слова `private` в начале определения этого метода вас пока не смущает; о том, что оно означает, вы узнаете при изучении главы 7 «Создание классов и объектов и управление ими».

Метод `addValues` содержит две инструкции. Первая из них показывает производимое вычисление в имеющемся в форме поле выражения, вторая использует `int`-версию оператора `+` для сложения `int`-переменных `leftHandSide`

и `rightHandSide`, а затем возвращает результат этой операции. Вспомним, что сложение двух `int`-значений создает еще одно `int`-значение, следовательно, типом возвращаемого методом `addValues` значения будет `int`.

Если изучить методы `subtractValues`, `multiplyValues`, `divideValues` и `remainderValues`, можно увидеть, что в них используется похожая схема.

Найдите в окне редактора метод `showResult`, имеющий следующий вид:

```
private void showResult(int answer) => result.Text = answer.ToString();
```

Это метод-выражение, выводящий на экран, в имеющееся в форме поле результата, строку, представляющую собой параметр `answer`. Он не возвращает значение, поэтому для него указан тип `void`.



СОВЕТ Минимальной длины метода не существует. Если метод позволяет избавиться от повторений и облегчает понимание программы, его следует признать полезным, каким бы он ни был коротким.

Нет и максимальной длины метода, но обычно применительно к выполняемой задаче код метода стараются сделать как можно короче. Если код вашего метода не помещается на экран монитора, подумайте о возможности его разбиения на более мелкие методы, чтобы их было легче читать.

Вызов методов

Методы существуют для того, чтобы их вызывали. Для того чтобы метод выполнил свою задачу, его вызывают по имени. Если методу нужна информация (определенная его параметрами), его нужно снабдить всем необходимым. Если метод возвращает информацию, определенную типом его возвращаемого значения, то вы должны каким-то образом ее забрать.

Определение синтаксиса вызова метода

Синтаксис вызова метода выглядит в C# следующим образом:

```
result = methodName ( argumentList )
```

Рассмотрим описание элементов, составляющих вызов метода.

- ❑ Элемент `methodName` должен в точности соответствовать имени вызываемого метода. При этом нужно помнить, что язык C# чувствителен к регистру символов.
- ❑ Элемент `result` = необязательный. Если он указан, переменная с идентификатором `result` будет содержать значение, возвращенное методом. Если это `void`-метод (то есть метод, не возвращающий значение), элемент инструкции

`result` = следует опустить. Если для метода, возвращающего значение, не указать `result` =, метод запустится, но возвращаемое значение останется неиспользованным.

- Элемент `argumentList` предоставляет информацию, принимаемую методом. Каждому параметру нужно предоставить аргумент, а значение каждого аргумента должно быть совместимо с типом соответствующего ему параметра. Если у вызываемого метода несколько параметров, аргументы должны отделяться друг от друга запятыми.



ВНИМАНИЕ Круглые скобки нужно включать в каждый вызов метода, даже если у метода нет аргументов.

Чтобы разъяснить эти положения, рассмотрим метод `addValues` еще раз:

```
int addValues(int leftHandSide, int rightHandSide)
{
    // ...
}
```

У метода `addValues` имеется два параметра типа `int`, следовательно, его нужно вызвать с указанием двух отделенных друг от друга запятой аргументов, имеющих тип `int`:

```
addValues(39, 3); // все правильно
```

Можно также заменить буквальные значения 39 и 3 именами `int`-переменных. Затем значения этих переменных передаются методу в качестве его аргументов:

```
int arg1 = 99;
int arg2 = 1;
addValues(arg1, arg2);
```

Если попытаться вызвать `addValues` каким-нибудь другим способом, то, скорее всего, попытка будет безуспешной по причинам, раскрытым в следующих примерах:

```
addValues;           // ошибка в ходе компиляции, отсутствуют круглые скобки
addValues();         // ошибка в ходе компиляции, недостаточно аргументов
addValues(39);       // ошибка в ходе компиляции, недостаточно аргументов
addValues("39", "3"); // ошибка в ходе компиляции, неверные типы аргументов
```

Метод `addValues` возвращает значение, имеющее тип `int`. Это `int`-значение можно применить там, где такое значение используется. Рассмотрим следующие примеры:

```
int result = addValues(39, 3); // в правой части присваивания
showResult(addValues(39, 3)); // в качестве аргумента вызова другого метода
```

В следующем упражнении будет продолжена работа с приложением Methods. На этот раз будут исследованы вызовы метода.

Исследование вызовов метода

Вернитесь в проект Methods. Выведите в окно редактора код файла MainPage.xaml.cs. Найдите метод calculateClick и посмотрите на первые две инструкции этого метода после инструкции try и открывающей фигурной скобки. (Инструкции try будут рассматриваться в главе 6 «Обработка ошибок и исключений».) Они имеют следующий вид:

```
int leftHandSide = System.Int32.Parse(lhsOperand.Text);  
int rightHandSide = System.Int32.Parse(rhsOperand.Text);
```

Эти две инструкции объявляют две int-переменные с именами `leftHandSide` и `rightHandSide`. Обратите внимание на способ их инициализации. В обоих случаях вызывается метод `Parse` структуры `System.Int32`. (`System` является пространством имен, а `Int32` — именем структуры в этом пространстве имен. О структурах вы узнаете в главе 9 «Создание типов значений с использованием перечислений и структур».) Ранее вам уже встречался этот метод, он получает один строковый параметр и преобразует его в `int`-значение. Эти две строки кода получают то, что пользователь набрал в имеющихся в форме текстовых полях `lhsOperand` и `rhsOperand`, и преобразуют эти значения в тип `int`.

Посмотрите на четвертую инструкцию метода `calculateClick` (после инструкции `if` и еще одной открывающей фигурной скобки):

```
calculatedValue = addValues(leftHandSide, rightHandSide);
```

Эта инструкция вызывает метод `addValues`, передавая ему в качестве аргументов значения переменных `leftHandSide` и `rightHandSide`. Значение, возвращаемое методом `addValues`, сохраняется в переменной `calculatedValue`.

Посмотрите на следующую инструкцию:

```
showResult(calculatedValue);
```

Она вызывает метод `showResult`, передавая ему в качестве аргумента значение переменной `calculatedValue`. Метод `showResult` никакого значения не возвращает.

Найдите в окне редактора рассмотренный ранее метод `showResult`. В нем содержится всего одна инструкция:

```
result.Text = answer.ToString();
```

Обратите внимание на то, что при вызове метода `ToString` используются круглые скобки даже при том, что у него нет аргументов.



СОВЕТ Методы, принадлежащие другим объектам, можно вызывать, указывая в качестве префикса имя объекта. В предыдущем примере выражение `answer.ToString()` вызывает метод по имени `ToString`, принадлежащий объекту по имени `answer`.

Применение области видимости переменных

Переменные создаются для хранения значений. Их можно создавать в различных местах вашего приложения. Например, в методе `calculateClick` проекта `Methods` создается `int`-переменная по имени `calculatedValue` и ей присваивается начальное значение 0:

```
private void calculateClick(object sender, RoutedEventArgs e)
{
    int calculatedValue = 0;
...
}
```

Эта переменная начинает свое существование в том месте, где она определена, после чего следующие инструкции в методе `calculateClick` могут ее использовать. Важный момент: переменная может использоваться только после ее создания. Когда метод завершает свою работу, эта переменная исчезает и уже не может где-либо использоваться.

Когда к переменной имеется доступ в конкретном месте программы, говорится, что переменная в этом месте находится в области видимости. Область видимости переменной `calculatedValue` ограничивается методом, к ней можно получить доступ в пространстве метода `calculateClick`, но не за его пределами. Можно также определить переменные с другой областью видимости, например за пределами метода, но внутри класса, и доступ к этим переменным можно получить из любого метода внутри этого класса. Про такие переменные говорят, что их областью видимости является пространство класса.

Иначе говоря, область видимости переменной — это просто участок программы, в пределах которого переменной можно воспользоваться. Область видимости имеется не только у переменных, но и у методов. Далее вы узнаете, что область видимости идентификатора (переменной или метода) привязана к месту объявления, вводящего идентификатор в программу.

Определение локальной области видимости

Область видимости внутри метода определяют открывающая и закрывающая фигурные скобки, формирующие его тело. Любые переменные, объявленные внутри тела метода, обладают областью видимости в его пределах и исчезают, как только метод завершает свою работу. Они могут быть доступны только коду,

выполняемому в этом методе. Такие переменные называются локальными, поскольку они имеют локальную привязку к методу, в котором объявлены, и не находятся в области видимости любого другого метода.

Область видимости локальных переменных означает, что вы не можете использовать их для обмена информацией между методами. Рассмотрим следующий пример:

```
class Example
{
    void firstMethod()
    {
        int myVar;
        ...
    }
    void anotherMethod()
    {
        myVar = 42; // ошибка – такая переменная в области видимости отсутствует
        ...
    }
}
```

Этот код не будет откомпилирован, поскольку метод `anotherMethod` пытается использовать переменную `myVar`, не находящуюся в его области видимости. Переменная `myVar` доступна только инструкциям в методе `firstMethod`, следующим в нем после строки, объявляющей `myVar`.

Определение области видимости внутри класса

Область видимости внутри класса определяют открывающая и закрывающая фигурные скобки, формирующие тело этого класса. Любые переменные, объявленные внутри тела класса (но не внутри метода), имеют область видимости в пределах этого класса. В C# для переменной, определенной классом, применяется термин *поле*. Как уже упоминалось, в отличие от локальных переменных, поля можно использовать для обмена информацией между методами. Рассмотрим следующий пример:

```
class Example
{
    void firstMethod()
    {
        myField = 42; // ok
        ...
    }
    void anotherMethod()
    {
        myField++; // ok
        ...
    }
    int myField = 0;
}
```

Переменная `myField` определена в классе, но за пределами методов `firstMethod` и `anotherMethod`. Поэтому областью видимости `myField` является класс, и эту переменную можно использовать в любых методах данного класса.

Применительно к примеру следует отметить еще одно обстоятельство. В методе переменная должна быть объявлена до ее применения. Поля в этом плане несколько отличаются. Метод может использовать поле до той инструкции, в которой это поле определяется, — все тонкости этого обстоятельства улаживаются компилятором.

Перегрузка методов

Если у двух идентификаторов одинаковые имена и они объявлены в одной и той же области видимости, то к ним применяется понятие *перегрузка*. Зачастую перегрузка идентификатора является ошибкой, отлавливаемой в ходе компиляции. Например, если в одном и том же методе объявляются две локальные переменные с одинаковыми именами, компилятор выдает ошибку. Аналогично, если в одном и том же классе объявляются два поля с одинаковыми именами или два одинаковых метода, будет получена ошибка в ходе компиляции. Казалось бы, зачем об этом говорить, учитывая, что все пока что превращалось в ошибку в ходе компиляции. Но существует весьма полезный и важный способ, позволяющий перегрузить идентификатор для метода.

Рассмотрим метод `WriteLine` класса `Console`. Вы уже им пользовались для вывода строки на экран. Но в процессе написания кода C# при наборе `WriteLine` в окне редактора кода и текста можно заметить, что механизм Microsoft IntelliSense дает вам в виде подсказки 19 различных вариантов! Каждая версия метода `WriteLine` получает разный набор параметров: одна вообще не получает параметров и просто выводит пустую строку, другая получает булев параметр и выводит строку, представляющую его значение (`True` или `False`), а еще одна реализация получает параметр в виде десятичного числа и выводит его в виде строки и т. д. В процессе работы компилятор смотрит на типы передаваемых вами аргументов, а затем подстраивается под ваше приложение для вызова той версии метода, которая соответствует имеющемуся набору параметров. Рассмотрим пример:

```
static void Main()
{
    Console.WriteLine("The answer is ");
    Console.WriteLine(42);
}
```

В первую очередь перегрузка пригодится тогда, когда нужно будет выполнить одну и ту же операцию над данными различных типов или при различных вариантах группировки информации. Вы можете перегрузить метод, когда

различные реализации имеют разные наборы параметров, то есть когда имеется одно и то же имя, но разное количество параметров или когда различаются типы этих параметров. При вызове метода ему предоставляется список аргументов, отделенных друг от друга запятыми, а количество и тип аргументов используются компилятором для выбора одного из вариантов перегружаемых методов. Но следует иметь в виду, что, несмотря на возможность перегрузки, связанный с параметрами метода, вы не можете выполнить перегрузку, исходя из типа возвращаемого методом значения. Иными словами, нельзя объявить два метода с одним и тем же именем, которые отличались бы друг от друга только типом возвращаемого ими значения. (Компьютер, конечно, устройство умное, но не до такой же степени.)

Создание методов

В следующих упражнениях будет создан метод, вычисляющий сумму, которую получит консультант за заданное количество консультационных дней при фиксированной посуточной ставке. Сначала будет разработана логика приложения, а затем задействован мастер создания заглушек для методов — Generate Method Stub Wizard, помогающий создавать методы, используемые этой логикой. Далее, чтобы разобраться в работе программы, вы запустите эти методы в консольном приложении. И наконец, воспользуетесь отладчиком Visual Studio 2015, чтобы выполнить код в пошаговом режиме внутри вызываемого метода и за его пределами.

Разработка логики для приложения

Откройте в Visual Studio 2015 проект DailyRate, который находится в папке `\Microsoft Press\VCSBS\Chapter 3\DailyRate` вашей папки документов. Зайдите в обозреватель решений и в проекте DailyRate дважды щелкните на файле `Program.cs`, чтобы код для программы отобразился в окне редактора. Эта программа является простым стендом для проверки работоспособности кода. Когда приложение приступает к работе, оно вызывает метод `run`. Вы добавляете к методу `run` тот код, чью работу хотите опробовать. (Способ вызова метода требует знаний о том, что такое классы, а этот вопрос будет рассматриваться в главе 7.)

Добавьте к телу метода `run` между открывающей и закрывающей фигурными скобками следующие инструкции, выделенные жирным шрифтом:

```
void run()
{
    double dailyRate = readDouble("Enter your daily rate: ");
    int noOfDays = readInt("Enter the number of days: ");
    writeFee(calculateFee(dailyRate, noOfDays));
}
```

Блок кода, только что добавленный к методу `run`, вызывает метод `readDouble` (который вскоре будет создан), чтобы спросить у пользователя, какова дневная ставка консультанта. Следующая инструкция вызывает метод `readInt` (который также вскоре будет создан), чтобы получить количество дней. И наконец, вызывается метод `writeFee` (который также еще не создан) с целью вывода результата на экран. Обратите внимание на то, что значение, передаваемое методу `writeFee`, является значением, возвращенным методом `calculateFee` (это последний намеченный к созданию метод), который получает дневную ставку и количество дней и вычисляет сумму подлежащего к выплате заработка.



ПРИМЕЧАНИЕ Методы `readDouble`, `readInt`, `writeFee` и `calculateFee` пока еще не созданы, поэтому IntelliSense при наборе данного кода их не показывает. Не пытайтесь выполнять сборку приложения на этом этапе — у вас ничего не получится.

Создание методов с помощью мастера заглушек Generate Method Stub Wizard

Найдите в окне редактора, в методе `run`, вызов метода `readDouble` и щелкните на нем правой кнопкой мыши. Появится контекстное меню, содержащее команды, необходимые для создания и редактирования кода (рис. 3.1).

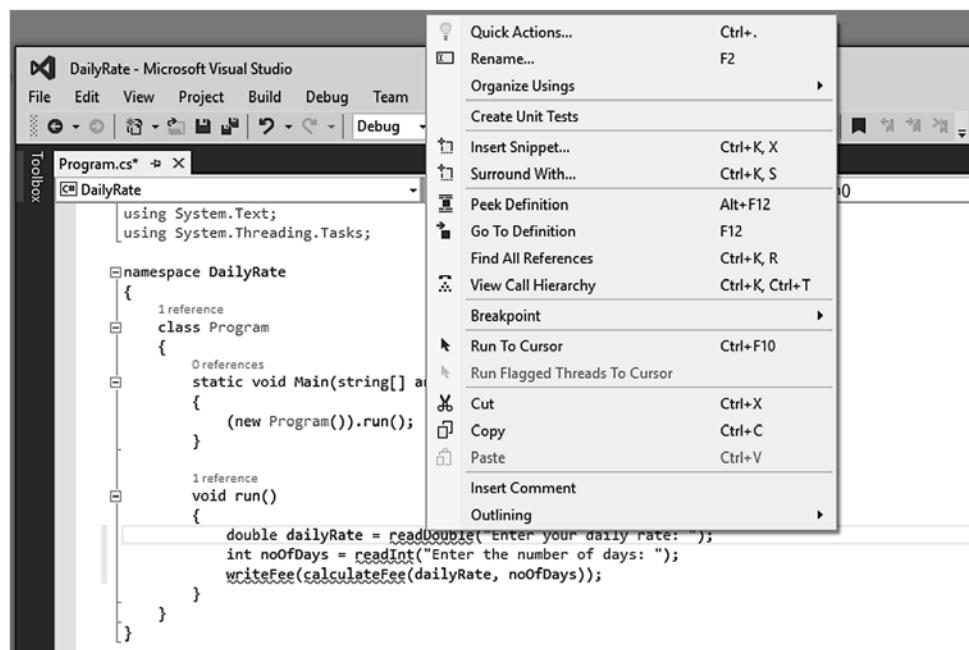


Рис. 3.1

Щелкните в этом меню на пункте Быстрые действия (Quick Actions).

Visual Studio удостоверится в том, что метод `readDouble` не существует, и покажет мастер, позволяющий создать заглушку для этого метода. Visual Studio исследует вызов метода `readDouble`, установит тип его параметров и тип возвращаемого значения и предложит исходную реализацию этого метода (рис. 3.2).

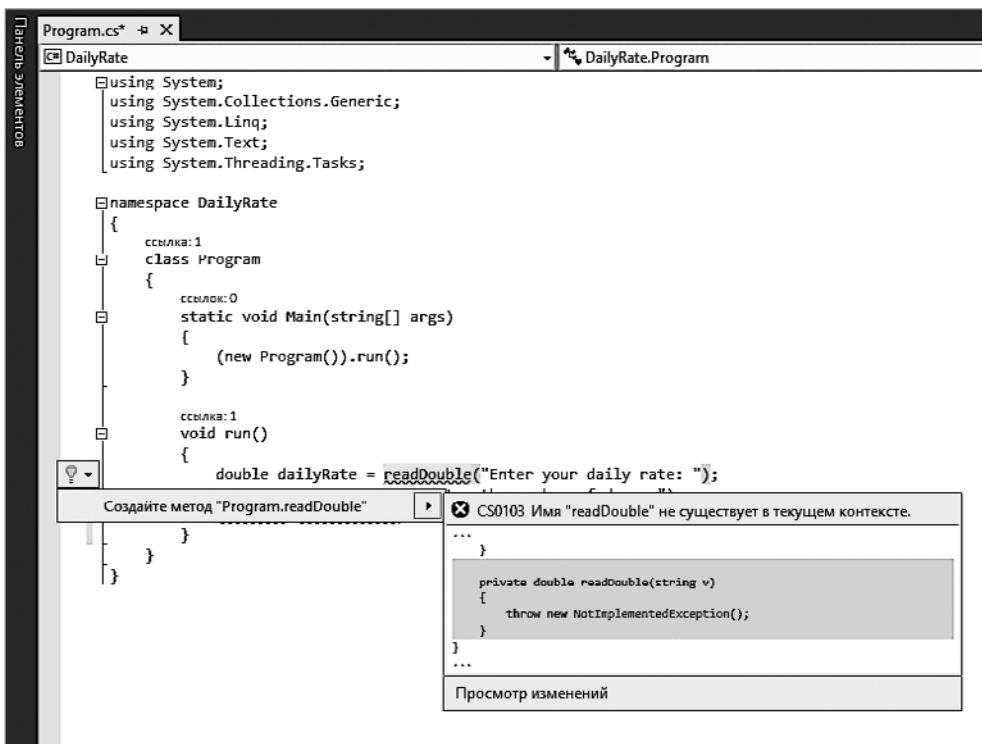


Рис. 3.2

Щелкните на пункте Создайте метод "Program.readDouble" (Generate Method 'Program.readDouble'). Среда Visual Studio добавит к вашему коду следующий метод:

```
private double readDouble(string v)
{
    throw new NotImplementedException();
}
```

Новый метод создан с классификатором `private`, который будет рассмотрен в главе 7. Пока в теле метода просто выдается исключение `NotImplementedException`. (Исключения рассматриваются в главе 6.) Далее вы замените это тело своим собственным кодом.

Удалите инструкцию `throw new NotImplementedException();` из метода `readDouble`, заменив ее следующими строками кода, выделенными жирным шрифтом:

```
private double readDouble(string v)
{
    Console.Write(v);
    string line = Console.ReadLine();
    return double.Parse(line);
}
```

Этот блок кода выводит на экран строку, находящуюся в переменной `v`. Эта переменная является строковым параметром, передаваемым при вызове метода, и предлагает пользователю ввести посுточную ставку.



ПРИМЕЧАНИЕ Метод `Console.WriteLine` похож на инструкцию `Console.WriteLine`, которая использовалась в предыдущих упражнениях, и отличается только тем, что не выводит после сообщения символ новой строки.

Пользователь набирает значение, которое считывается в строку с помощью метода `ReadLine` и преобразуется в значение числа с двойной точностью, для чего применяется метод `double.Parse`. Результат передается назад в качестве возвращаемого значения вызываемого метода.



ПРИМЕЧАНИЕ Метод `ReadLine` является методом-компаньоном для `WriteLine`, он считывает пользовательский ввод с клавиатуры, завершающийся нажатием клавиши Ввод. Текст, набранный пользователем, передается назад в виде возвращаемого значения. Текст возвращается в виде строкового значения.

Щелкните в методе `run` правой кнопкой мыши на вызове метода `readInt`, после этого щелкните на пункте **Быстрые действия**, а затем на пункте **Создайте метод "Program.readInt"**.

Будет создан метод `readInt`:

```
private int readInt(string v)
{
    throw new NotImplementedException();
}
```

Замените инструкцию `throw new NotImplementedException();` в теле метода `readInt` следующим кодом, выделенным жирным шрифтом:

```
private int readInt(string v)
{
    Console.Write(v);
    string line = Console.ReadLine();
    return int.Parse(line);
}
```

Этот блок кода похож на код для метода `readDouble`. Единственная разница заключается в том, что этот метод возвращает значение типа `int`, следовательно, строка, набранная пользователем, преобразуется в число, для чего используется метод `int.Parse`.

Щелкните правой кнопкой мыши на вызове метода `calculateFee`, после этого щелкните на пункте **Быстрые действия**, а затем на пункте **Создайте метод "Program.calculateFee"**.

Будет создан метод `calculateFee`:

```
private object calculateFee(double dailyRate, int noOfDays)
{
    throw new NotImplementedException();
}
```

Обратите внимание на то, что в данном случае Visual Studio для создания имен параметров использует имена переданных методу аргументов. (Разумеется, вы можете изменить имена параметров, если они вам не подходят.) Еще более интересным является выбор типа, возвращаемого методом, который указывается как `object`. Среда Visual Studio не может точно определить тип значения, которое должно быть возвращено методом из того контекста, в котором он вызван. Тип `object` просто означает нечто, и вы при добавлении в метод кода должны изменить его на требуемый тип. Более подробно тип `object` рассматривается в главе 7.

Измените определение метода `calculateFee` таким образом, чтобы он, как показано в этом коде жирным шрифтом, возвращал значение в виде числа с двойной точностью:

```
private double calculateFee(double dailyRate, int noOfDays)
{
    throw new NotImplementedException();
}
```

Замените тело метода `calculateFee` и переделайте его в метод-выражение со следующим выражением, выделенным жирным шрифтом. Эта инструкция вычисляет подлежащий выплате заработка путем перемножения двух параметров:

```
private double calculateFee(double dailyRate, int noOfDays) => dailyRate *
noOfDays;
```

Щелкните правой кнопкой мыши на вызове метода `writeFee`, находящемся в методе `run`, щелкните на пункте **Быстрые действия**, а затем на пункте **Создайте метод "Program.writeFee"**.

Обратите внимание: среда Visual Studio, чтобы выяснить, что его параметр должен принадлежать типу `double`, использует определение метода `writeFee`.

Кроме того, в вызове метода не используется возвращаемое значение, поэтому для него объявляется тип `void`:

```
private void writeFee(double v)
{
    ...
}
```



СОВЕТ Если вы хорошо разбираетесь в синтаксисе, то можете создавать методы, набирая их в окне редактора. Постоянно использовать пункт меню Создайте... вас никто не заставляет.

Замените код в теле метода `writeFee` следующей инструкцией, которая выводом результата на экран вычисляет заработка и добавляет 10 % комиссии. Еще раз обратите внимание на то, что теперь это не просто метод, а метод-выражение:

```
private void writeFee(double v) => Console.WriteLine($"The consultant's fee is:
{v * 1.1}");
```

Щелкните в меню Сборка на пункте Собрать решение.

Реорганизация кода

Весьма полезной особенностью Visual Studio 2015 является возможность реорганизации кода.

Временами вы станете замечать, что создаете один и тот же или очень похожий код более чем в одном месте приложения. Когда это произойдет, выделите только что набранный блок кода, щелкните на нем правой кнопкой мыши, после чего щелкните на пункте **Быстрые действия**, а затем на пункте **Извлечение метода** (*Extract Method*). Выбранный код будет перемещен в новый метод по имени `NewMethod`. Мастер извлечения метода способен также определить, должен ли метод принимать какие-либо параметры и иметь возвращаемое значение. После создания метода нужно изменить его имя на что-либо более понятное, вписав его поверх прежнего, а также изменить инструкцию, созданную для вызова этого метода, вписав в нее новое имя.

Тестирование программы

Щелкните в меню Отладка на пункте Запуск без отладки (*Start Without Debugging*). Среда Visual Studio 2015 проведет сборку программы, а затем запустит ее на выполнение. Появится окно консоли. В строке приглашения ввести посуточную ставку `Enter Your Daily Rate` наберите `525` и нажмите **Ввод**. В строке приглашения ввести количество дней `Enter The Number of Days` наберите `17` и нажмите **Ввод**.

Программа выведет в окно консоли следующее сообщение о подлежащем выплате заработке:

```
The consultant's fee is: 9817,5
```

Нажмите Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

В следующем упражнении для прогона программы в медленном темпе будет использован отладчик Visual Studio 2015. Вы увидите, что происходит, когда вызывается каждый метод (это называется шагом с заходом в метод), а затем увидите, как каждая инструкция `return` передает управление назад вызывавшей инструкции (это называется шагом с выходом из метода). При шаге с заходом в метод и при шаге с выходом из метода вы сможете воспользоваться инструментами, имеющимися на панели отладки. Когда приложение запущено в режиме отладки, те же самые команды доступны и в меню Отладка.

Пошаговое выполнение методов с использованием отладчика Visual Studio 2015

Найдите в окне редактора метод `run`. Поставьте курсор на первую инструкцию этого метода:

```
double dailyRate = readDouble("Enter your daily rate: ");
```

Щелкните правой кнопкой мыши в любом месте этой строки, а затем щелкните на пункте Выполнить до текущей позиции (Run To Cursor).

Программа запустится и будет выполнятья, пока не дойдет до первой инструкции в методе `run`, после чего встанет на паузу. Текущая инструкция будет помечена желтой стрелкой в левом поле окна редактора, а сама инструкция будет выделена желтым фоном (рис. 3.3).

Выберите в меню Вид пункт Панели инструментов (Toolbars) и убедитесь, что слева от пункта Отладка (Debug toolbar) установлен флажок. Когда он установлен, панель инструментов отладки будет открыта. Эта панель может быть пристыкована к другим панелям инструментов. Если вы не сможете ее найти, попробуйте воспользоваться командой Панели инструментов в меню Вид, чтобы скрыть эту панель (снять флажок), и посмотрите, какие кнопки исчезнут. Затем снова выведите панель на экран. Панель инструментов отладки имеет следующий вид (рис. 3.4).

Щелкните на панели инструментов отладки на кнопке Шаг с заходом (Step Into) (это шестая кнопка слева). Отладчик сделает шаг с заходом в вызываемый метод. Желтый курсор переместится на открывающую фигурную скобку в начале метода `readDouble`.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DailyRate
{
    class Program
    {
        static void Main(string[] args)
        {
            (new Program()).run();
        }

        void run()
        {
            double dailyRate = readDouble("Enter your daily rate: ");
            int noOfDays = readInt("Enter the number of days: ");
            writeFee(calculateFee(dailyRate, noOfDays));
        }

        private void writeFee(double v) => Console.WriteLine($"The consultant's fee is: {v * 1.1}");

        private double calculateFee(double dailyRate, int noOfDays) => dailyRate * noOfDays;

        private int readInt(string v)
        {
            Console.Write(v);
            string line = Console.ReadLine();
            return int.Parse(line);
        }
    }
}

```

Рис. 3.3

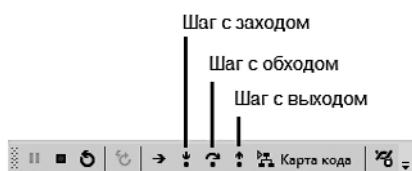


Рис. 3.4

Щелкните еще раз на кнопке Шаг с заходом, чтобы курсор переместился на первую инструкцию:

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



СОВЕТ Вместо повторных щелчков на кнопке Шаг с заходом на панели отладки можно нажимать клавишу F11.

Щелкните на панели отладки на кнопке Шаг с обходом (Step Over) (это седьмая кнопка слева). Метод выполнит следующую инструкцию без ее отладки (без шага с заходом в нее). Такое действие может пригодиться в том случае, если инструкция вызывает метод, но вы не хотите вызывать пошаговое выполнение каждой его инструкции. Желтый курсор переместится на вторую инструкцию метода, и программа перед возвращением в Visual Studio 2015 выведет в окно консоли приглашение на ввод посutoчной ставки. (Окно консоли может скрываться за окном Visual Studio.)



СОВЕТ Вместо того чтобы щелкать на кнопке Шаг с обходом на панели отладки, можно нажимать клавишу F10.

Еще раз щелкните на панели отладки на кнопке Шаг с обходом. На этот раз желтый курсор исчезнет и активируется окно консоли, поскольку программа выполняет метод `Console.ReadLine` и ожидает, что вы что-нибудь наберете в строке приглашения. Наберите в окне консоли 525 и нажмите Ввод.

Управление вернется в Visual Studio 2015. Желтый курсор появится на третьей строчке метода.

Наведите указатель мыши на имя переменной `line` во второй или в третьей строке метода (неважно, в какой именно). Появится подсказка, выводящая текущее значение переменной `line` ("525"). Это позволяет убедиться в том, что при пошаговом выполнении кода метода переменной было присвоено ожидаемое вами значение (рис. 3.5).

```
ссылка:1
private double readDouble(string v)
{
    Console.WriteLine(v);
    string line = Console.ReadLine();
    return double.Parse(line); ⏴ 11 410 мс прошло
}
}
```

Рис. 3.5

Щелкните на панели отладки на кнопке Шаг с выходом (Step Out) (это восьмая кнопка слева). Запустится непрерывное выполнение кода метода до самого конца. Работа метода `readDouble` завершится, и желтый курсор вернется к первой инструкции метода `run`. Теперь эта инструкция завершит свою работу.



СОВЕТ Вместо того чтобы щелкать на кнопке Шаг с выходом на панели отладки, можно нажимать комбинацию клавиш Shift+F11.

Щелкните на панели отладки на кнопке Шаг с заходом. Желтый курсор переместится на вторую инструкцию метода run:

```
int noOfDays = readInt("Enter the number of days: ");
```

Щелкните на панели отладки на кнопке Шаг с обходом. На этот раз выбран запуск метода без пошагового выполнения кода внутри него. Снова появится окно консоли с приглашением на ввод количества дней. Наберите в окне консоли 17 и нажмите Ввод. Управление будет возвращено Visual Studio 2015 (возможно, придется вывести эту среду на первый план). Желтый курсор переместится на третью инструкцию метода run:

```
writeFee(calculateFee(dailyRate, noOfDays));
```

Щелкните на панели отладки на кнопке Шаг с заходом. Желтый курсор перейдет на выражение, определяющее тело метода calculateFee. Этот метод будет вызван раньше метода writeFee, поскольку значение, которое им возвращается, метод writeFee использует в качестве параметра.

Щелкните на панели отладки на кнопке Шаг с выходом. Вызов метода calculateFee завершится, и желтый курсор вернется обратно на третью инструкцию метода run.

Щелкните на панели отладки на кнопке Шаг с заходом. На этот раз желтый курсор перейдет на инструкцию, определяющую тело метода writeFee. Поставьте в определении метода указатель мыши над параметром v. В экранной подсказке будет показано значение v, равное 8925.

Щелкните на панели отладки на кнопке Шаг с выходом. В окне консоли появится сообщение The consultant's fee is: 9817,5. (Если окно консоли скрыто за окном Visual Studio 2015, его нужно вывести на первый план.) Желтый курсор вернется на третью инструкцию в методе run.

Щелкните на панели инструментов на кнопке Продолжить (Continue), чтобы заставить программу продолжить выполнение без остановок на каждой инструкции.



СОВЕТ Если кнопка Продолжить не видна, щелкните на раскрывающемся меню Добавить или удалить кнопки (Add Or Remove Buttons), которое появляется в конце панели отладки, а затем выберите пункт Продолжить. После этого должна появиться кнопка Продолжить. Вместо этого для продолжения выполнения приложения без отладки можно нажать клавишу F5.

Приложение выполнится до конца и завершит работу. Обратите внимание на то, что с окончанием работы приложения исчезнет и инструментальная панель отладчика, поскольку при исходных настройках она отображается только при запуске приложения в режиме отладки.

Использование необязательных параметров и именованных аргументов

Вы уже видели, как путем определения перегружаемых методов можно реализовать разные версии метода, получающего различные параметры. При создании приложения, использующего перегружаемые методы, компилятор определяет, какие конкретные экземпляры каждого метода нужно использовать для удовлетворения каждого вызова. Это общая черта многих объектно-ориентированных языков, присущая не только C#. Но для создания Windows-приложений и компонентов разработчики могут использовать другие языки, которые не придерживаются этих правил. Основной особенностью C# и других языков, разработанных для среды .NET Framework, является возможность взаимодействия с приложениями и компонентами, написанными с применением других технологий. Одной из важнейших технологий, положенных в основу многих Windows-приложений и служб, работающих вне среды .NET Framework, является объектная модель компонентов (Component Object Model (COM)). По сути, общеязыковая среда выполнения (common language runtime (CLR)), используемая .NET Framework, так же сильно зависит от применения COM, как и среда выполнения Windows Runtime операционной системы Windows 10. В COM не поддерживаются перегружаемые методы, вместо этого в ней используются методы, способные принимать необязательные параметры. Чтобы упростить использование COM-библиотек и компонентов в решении на C#, язык C# также поддерживает необязательные параметры.

Необязательные параметры могут пригодиться и в других ситуациях. Они предоставляют компактные и простые решения при невозможности использования перегружаемых методов по причине несущественных различий типов параметров, не позволяющих компилятору заметить разницу между реализациями. Рассмотрим, к примеру, следующий код:

```
public void DoWorkWithData(int intData, float floatData, int moreIntData)
{
    ...
}
```

Метод `DoWorkWithData` принимает три параметра: два типа `int` и один типа `float`. Теперь предположим, что вам нужно предоставить реализацию метода `DoWorkWithData`, принимающего только два параметра: `intData` и `floatData`. Можно перегрузить метод:

```
public void DoWorkWithData(int intData, float floatData)
{
    ...
}
```

Если написать инструкцию, вызывающую метод `DoWorkWithData`, вы можете предоставить либо два, либо три параметра соответствующих типов, и компилятор использует информацию о типах для определения того, какой из перегружаемых методов следует вызывать:

```
int arg1 = 99;
float arg2 = 100.0F;
int arg3 = 101;

DoWorkWithData(arg1, arg2, arg3); // Вызов перегруженного метода с тремя
                                 // параметрами
DoWorkWithData(arg1, arg2);      // Вызов перегруженного метода с двумя
                                 // параметрами
```

Но предположим, что вам понадобилось реализовать две дополнительные версии `DoWorkWithData`, принимающие только первый и третий параметры. Можно попытаться сделать следующее:

```
public void DoWorkWithData(int intData)
{
    ...
}
public void DoWorkWithData(int moreIntData)
{
    ...
}
```

Проблема здесь в том, что компилятору эти два перегружаемых метода представляются совершенно идентичными. Ваш код не пройдет компиляцию, и будет выдана ошибка, говорящая о том, что тип с таким-то названием уже определяет экземпляр по имени `DoWorkWithData` с такими же типами параметров. Чтобы разобраться в случившемся, подумайте, что бы произошло, будь этот код допустимым. Рассмотрим следующие инструкции:

```
int arg1 = 99;
int arg3 = 101;

DoWorkWithData(arg1);
DoWorkWithData(arg3);
```

Какая перегрузка или перегрузки должны вызываться, чтобы задействовать `DoWorkWithData`? Решить эту проблему помогут необязательные параметры и именованные аргументы.

Определение необязательных параметров

Отсылка на необязательность параметра делается при определении метода предоставлением параметру исходного значения. Это значение указывается путем

использования оператора присваивания. В показанном далее методе `optMethod` первый параметр является обязательным, поскольку для него не указано исходное значение, а вот второй и третий параметры являются необязательными:

```
void optMethod(int first, double second = 0.0, string third = "Hello")
{
    ...
}
```

Сначала должны быть указаны все обязательные и только потом — любые необязательные параметры.

Вызов метода, принимающего необязательные параметры, можно осуществлять тем же способом, что и вызов любого другого метода: нужно указать имя метода и предоставить любые необходимые аргументы. Отличие методов, принимающих необязательные параметры, состоит в том, что соответствующие этим параметрам аргументы можно опустить, и тогда при запуске метод воспользуется исходным значением. В следующем примере при первом вызове метода `optMethod` ему предоставляются значения для всех трех параметров. При втором вызове указываются только два аргумента, и эти значения применяются для первого и второго параметров. Третий параметр получает при запуске метода исходное значение `"Hello"`:

```
optMethod(99, 123.45, "World"); // Аргументы предоставлены всем трем параметрам
optMethod(100, 54.321);          // Аргументы предоставлены только первым двум
                                // параметрам
```

Передача именованных аргументов

Изначально C# использует позицию каждого аргумента в вызове метода для определения того, к какому параметру какой аргумент применить. Поэтому второй пример метода `OptMethod`, показанный в предыдущем разделе, передает два аргумента параметрам метода `first` и `second`, поскольку таков порядок их появления в объявлении метода. C# позволяет также указывать параметры по именам. Это свойство позволяет передавать аргументы в другой последовательности. Для передачи аргумента в качестве именованного параметра указывается имя параметра, затем ставится двоеточие и указывается используемое значение. Следующие примеры выполняют ту же самую функцию, что и показанные в предыдущем разделе, за исключением того, что параметры указаны по имени:

```
optMethod(first : 99, second : 123.45, third : "World");
optMethod(first : 100, second : 54.321);
```

Именованные аргументы дают возможность передавать аргументы в любом порядке. Код вызова метода `optMethod` можно переписать следующим образом:

```
optMethod(third : "World", second : 123.45, first : 99);
optMethod(second : 54.321, first : 100);
```

Это свойство позволяет также опускать аргументы. Например, при вызове метода `optMethod` можно указать значения только для первого и третьего параметров, а для второго параметра воспользоваться его исходным значением:

```
optMethod(first : 99, third : "World");
```

Кроме того, позиционные и именованные аргументы можно смешивать. Но если использовать этот технический прием, то перед первым именованным аргументом нужно указать все позиционные аргументы:

```
optMethod(99, third : "World"); // Сначала указан позиционный аргумент
```

Устранение неоднозначностей, связанных с необязательными параметрами и именованными аргументами

Применение необязательных параметров и именованных аргументов может привести к некоторой неоднозначности вашего кода. Стоит разобраться с тем, как компилятор разрешает эти неоднозначности, иначе в приложениях может обнаружиться неожиданное поведение. Предположим, что вы определили метод `optMethod` в качестве перегружаемого, как показано в следующем примере:

```
void optMethod(int first, double second = 0.0, string third = "Hello")
{
    ...
}
void optMethod(int first, double second = 1.0, string third = "Goodbye",
               int fourth = 100 )
{
    ...
}
```

Это абсолютно допустимый код C#, который следует правилам для перегружаемых методов. Компилятор может отличить один метод от другого, поскольку у них разные списки параметров. Но как показано в следующем примере, при вызове метода `optMethod` и пропуске некоторых аргументов, соответствующих одному или нескольким необязательным параметрам, может возникнуть проблема:

```
optMethod(1, 2.5, "World");
```

Это опять же вполне допустимый код, но какой из версий метода `optMethod` он будет выполняться? Ответ таков: той, которая больше соответствует вызову

метода. Следовательно, код вызовет тот метод, который получает три, а не четыре параметра. Имеет смысл рассмотреть следующий пример:

```
optMethod(1, fourth : 101);
```

В этом примере в вызове `optMethod` не указаны аргументы для второго и третьего параметров, а аргумент для четвертого параметра указан по имени. Этому вызову соответствует только одна версия `optMethod`, следовательно, проблемы не возникнет. А вот следующий пример заставит вас призадуматься:

```
optMethod(1, 2.5);
```

На этот раз точного соответствия предоставленному списку аргументов нет ни у одного из вариантов метода `optMethod`. Обе его версии имеют необязательные параметры для второго, третьего и четвертого аргументов. Тогда какую из версий вызовет эта инструкция: ту, которая примет три параметра и воспользуется при этом исходным значением для третьего из них, или ту, которая примет четыре параметра и воспользуется при этом исходным значением для третьего и четвертого? Ответ будет следующим: ни ту ни другую. Это неразрешимая неоднозначность, и компилятор не позволит скомпилировать приложение. Сходная ситуация с тем же результатом возникает при попытках вызова метода `optMethod` в любой из показанных далее инструкций:

```
optMethod(1, third : "World");
optMethod(1);
optMethod(second : 2.5, first : 1);
```

В заключительном упражнении этой главы вы попрактикуетесь в реализации методов, принимающих необязательные параметры, и их вызове с использованием именованных аргументов. А также протестируете типовые примеры разрешения компилятором C# вызовов, использующих необязательные параметры и именованные аргументы.

Определение и вызов метода, принимающего необязательные параметры

Откройте в среде Visual Studio 2015 проект `DailyRate`, который находится в папке `\Microsoft Press\VCSBS\Chapter 3\DailyRate Using Optional Parameters` вашей папки документов. Раскройте в обозревателе решений проект `DailyRate` и дважды щелкните на имени файла `Program.cs` для отображения кода программы в окне редактора. Эта версия приложения не содержит ничего, кроме метода `Main` и структурной версии метода `run`.

В классе `Program` после метода `run` добавьте метод `calculateFee`. Это та же самая версия метода, которая была создана в предыдущем наборе упражнений,

за исключением того, что она принимает два необязательных параметра с исходными значениями. Метод также выводит сообщение, показывающее версию вызванного метода `calculateFee`. (Перегруженные реализации этого метода будут добавлены в дальнейшем.)

```
private double calculateFee(double dailyRate = 500.0, int noOfDays = 1)
{
    Console.WriteLine("calculateFee using two optional parameters");
    return dailyRate * noOfDays;
}
```

Добавьте к классу `Program` еще одну реализацию метода `calculateFee`, имеющую код, показанный далее. Эта версия принимает один необязательный параметр по имени `dailyRate` типа `double`. Тело метода вычисляет и возвращает сведения о заработке, причитающемся только за один день:

```
private double calculateFee(double dailyRate = 500.0)
{
    Console.WriteLine("calculateFee using one optional parameter");

    int defaultNoOfDays = 1;
    return dailyRate * defaultNoOfDays;
}
```

Добавьте к классу `Program` третью реализацию метода `calculateFee`. Эта версия не принимает параметры и использует зафиксированные в ней значения для посуточной ставки и количества дней:

```
private double calculateFee()
{
    Console.WriteLine("calculateFee using hardcoded values");
    double defaultDailyRate = 400.0;
    int defaultNoOfDays = 1;
    return defaultDailyRate * defaultNoOfDays;
}
```

Добавьте в метод `run` следующие инструкции, выделенные жирным шрифтом, которые вызывают `calculateFee` и показывают результат:

```
public void run()
{
    double fee = calculateFee();
    Console.WriteLine($"Fee is {fee}");
}
```



СОВЕТ Определение метода можно быстро просмотреть из вызывающей его инструкции. Для этого щелкните правой кнопкой мыши на вызове метода, а затем на пункте Показать определение (Peek Definition). Появляющееся после этого окно с определением метода `calculateFee` показано на рис. 3.6.

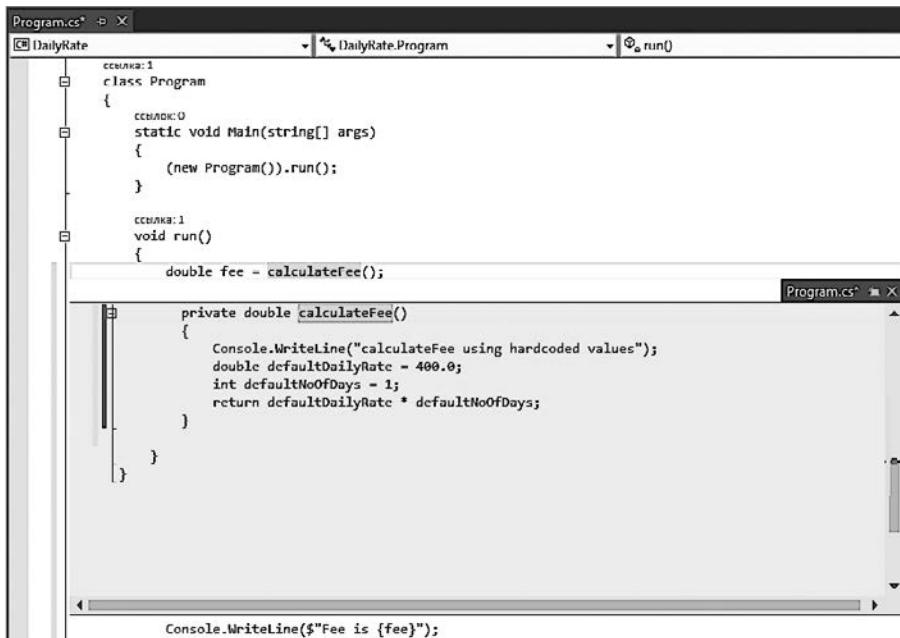


Рис. 3.6

Это свойство принесет особую пользу, если ваш код разбит на несколько файлов или находится в одном, но слишком длинном файле.

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запустить программы. Она запустится в окне консоли и выведет следующее сообщение:

```
calculateFee using hardcoded values
Fee is 400
```

Метод `run` вызвал версию `calculateFee`, не принимающую параметры, а не та реализации, которые принимают необязательные параметры, поскольку именно эта версия больше всех подходит вызову метода. Нажмите любую клавишу, чтобы закрыть окно консоли и вернуться в среду Visual Studio.

Измените в методе `run` инструкцию, вызывающую `calculateFee`, чтобы она приняла вид, выделенный ниже жирным шрифтом:

```
public void run()
{
    double fee = calculateFee(650.0);
    Console.WriteLine($"Fee is {fee}");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запуск программы. Будет выведено следующее сообщение:

```
calculateFee using one optional parameter
Fee is 650
```

На этот раз метод `run` вызвал версию `calculateFee`, принимающую один необязательный параметр. Как и прежде, эта версия больше остальных подходила вызову метода. Нажмите любую клавишу, чтобы закрыть окно консоли и вернуться в среду Visual Studio.

Еще раз измените в методе `run` инструкцию, вызывающую `calculateFee`:

```
public void run()
{
    double fee = calculateFee(500.0, 3);
    Console.WriteLine($"Fee is {fee}");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запуск программы. Будет выведено следующее сообщение:

```
calculateFee using two optional parameters
Fee is 1500
```

Как и при двух предыдущих запусках, метод `run` ожидаемо вызвал версию `calculateFee`, принимающую два необязательных параметра. Нажмите любую клавишу, чтобы закрыть окно консоли и вернуться в среду Visual Studio.

Измените в методе `run` инструкцию, вызывающую `calculateFee`, указав параметр `dailyRate` по имени:

```
public void run()
{
    double fee = calculateFee(dailyRate : 375.0);
    Console.WriteLine($"Fee is {fee}");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запуск программы. Будет выведено следующее сообщение:

```
calculateFee using one optional parameter
Fee is 375
```

Метод `run` опять вызвал версию `calculateFee`, принимающую один необязательный параметр. Изменение кода для использования именованного аргумента не привело к изменению способа принятия компилятором решения о том, какую версию метода вызывать в этом примере. Нажмите любую клавишу, чтобы закрыть окно консоли и вернуться в среду Visual Studio.

Измените в методе `run` инструкцию, вызывающую `calculateFee`, указав по имени параметр `noOfDays`:

```
public void run()
{
    double fee = calculateFee(noOfDays : 4);
    Console.WriteLine($"Fee is {fee}");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запустить программы. Будет выведено следующее сообщение:

```
calculateFee using two optional parameters
Fee is 2000
```

На этот раз метод `run` вызвал версию `calculateFee`, принимающую два необязательных параметра. В вызове метода был опущен первый параметр (`dailyRate`) и указан по имени второй. Этому вызову соответствовала только одна версия метода `calculateFee`, принимающая два необязательных параметра. Нажмите любую клавишу, чтобы закрыть окно консоли и вернуться в среду Visual Studio.

Измените реализацию метода `calculateFee`, принимающего два необязательных параметра. Измените имя первого параметра на `theDailyRate` и обновите инструкцию `return`, чтобы она соответствовала тому, что в следующем коде выделено жирным шрифтом:

```
private double calculateFee(double theDailyRate = 500.0, int noOfDays = 1)
{
    Console.WriteLine("calculateFee using two optional parameters");
    return theDailyRate * noOfDays;
}
```

Измените в методе `run` инструкцию, вызывающую `calculateFee`, указав параметр `theDailyRate` по имени:

```
public void run()
{
    double fee = calculateFee(theDailyRate : 375.0);
    Console.WriteLine("Fee is {fee}");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запустить программы. Будет выведено следующее сообщение:

```
calculateFee using two optional parameters
Fee is 375
```

В прошлый раз, когда указывалась посуточная ставка, но не указывалось количество дней, метод `run` вызывал версию `calculateFee`, принимающую один необязательный параметр. В этот раз метод `run` вызвал версию `calculateFee`,

принимающую два необязательных параметра. В данном случае использование именованного аргумента изменило способ принятия компилятором решения о том, какую из версий метода нужно вызывать. Если указывается именованный аргумент, компилятор сравнивает имя аргумента с именами параметров, указанными при объявлении метода, и выбирает метод, имеющий параметр с соответствующим именем. Если бы в вызове метода `calculateFee` аргумент был указан как `aDailyRate: 375.0`, программа не была бы откомпилирована, потому что ни у одной из версий метода нет параметра, соответствующего этому имени. Нажмите любую клавишу, чтобы закрыть окно консоли и вернуться в среду Visual Studio.

Выходы

В данной главе вы узнали, как определяются методы для реализации именованного блока кода. Вам было показано, как в методы передаются параметры и как из методов возвращаются данные. Вы также увидели, как вызывается метод, передаются аргументы и получается возвращаемое значение. Вы узнали, как определяются перегружаемые методы с различными списками параметров, и увидели, как область видимости переменных определяет, где к ним можно обращаться. Затем вы воспользовались отладчиком Visual Studio 2015 для пошагового выполнения кода. И наконец, узнали, как создаются методы, принимающие необязательные параметры, и как вызываются методы с использованием именованных аргументов.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 4 «Использование инструкций принятия решений».

Если хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Объявить метод	<p>Напишите метод внутри класса. Укажите имя метода, список параметров и тип возвращаемого значения, затем создайте тело метода, заключив его в фигурные скобки, например:</p> <pre>int addValues(int leftHandSide, int rightHandSide) { ... }</pre>

Чтобы	Сделайте следующее
Возвратить значение из метода	Напишите внутри метода инструкцию return, например: <code>return leftHandSide + rightHandSide;</code>
Вернуться из метода до его окончания	Напишите инструкцию return внутри метода, например: <code>return;</code>
Определить метод-выражение	Воспользуйтесь последовательностью символов =>, указав после нее выражение, определяющее тело метода, и поставив закрывающую точку с запятой, например: <code>double calculateFee(double dailyRate, int noOfDays)=> dailyRate * noOfDays;</code>
Вызвать метод	Напишите имя метода, за которым укажите в круглых скобках любые аргументы, например: <code>addValues(39, 3);</code>
Воспользоваться мастером создания методов-заглушек (Generate Method Stub Wizard)	Щелкните правой кнопкой мыши на вызове метода, а затем щелкните на пункте Создайте метод
Вывести панель инструментов отладчика	Выберите в меню Вид пункт Панели инструментов и установите флажок слева от пункта Отладка
Сделать шаг с заходом в метод	Щелкните на панели отладки на кнопке Шаг с заходом или щелкните в меню Отладка на пункте Шаг с заходом
Сделать шаг с выходом из метода	Щелкните на панели отладки на кнопке Шаг с выходом или щелкните в меню Отладка на пункте Шаг с выходом
Указать методу необязательный параметр	Предоставьте исходное значение для параметра в объявлении метода, например: <code>void optMethod(int first, double second = 0.0, string third = "Hell") { ... }</code>
Передать методу именованный аргумент	Укажите имя параметра в вызове метода, например: <code>optMethod(first : 100, third : "World");</code>

4

Использование инструкций принятия решений

Прочитав эту главу, вы научитесь:

- объявлять булевые переменные;
- использовать булевые операторы для создания выражений, вычисляемых в `true` ИЛИ в `false`;
- записывать инструкции `if` для принятия решений на основе результата вычисления булева выражения;
- записывать инструкции `switch` для принятия более сложных решений.

Из главы 3 «Создание методов и применение областей видимости» вы узнали, как группировать взаимосвязанные инструкции в методы. Кроме этого, вы научились использовать параметры для передачи информации методам, а инструкции `return` — для передачи информации из методов. Разделение программы на набор отдельных методов, каждый из которых разработан для выполнения конкретных задач или вычислений, является необходимой стратегией разработки. Многие программы нуждаются в решении больших и сложных задач. Разбиение программ на методы помогает разобраться с этими задачами и сконцентрироваться на их поэтапном решении.

Методы, рассмотренные в главе 3, имели весьма простую структуру, в которой каждая инструкция выполнялась последовательно после завершения выполнения предшествующей инструкции. Но для решения многих реальных задач нужна также возможность написания кода для выборочного выполнения различных действий и направления хода выполнения программы, в зависимости от обстоятельств, через тот или иной метод. Узнать о том, как выполнить эту задачу, вам позволит данная глава.

Объявление булевых переменных

В мире C#-программирования, в отличие от реального мира, все либо белое, либо черное, либо правильное, либо неправильное, либо истина (`true`), либо ложь (`false`). Например, если создать целочисленную переменную по имени `x`, присвоить ей значение 99, а затем спросить, содержит ли `x` значение 99, ответ определенно будет положительным (`true`). Если же спросить, правда ли, что `x` меньше 10, ответ вполне очевидно будет отрицательным (`false`). Все это примеры булевых выражений, которые всегда вычисляются в `true` или `false`.



ПРИМЕЧАНИЕ Ответы на эти вопросы не обязательно точно такие же для всех других языков программирования. У неинициализированной переменной значение не определено, и вы не можете, к примеру, со всей определенностью сказать, что она меньше 10. В программах, написанных на С и С++, подобные проблемы зачастую являются источником ошибок. Компилятор Microsoft Visual C# решает эту проблему за счет гарантированного присвоения вами значения переменной перед тем, как проверять ее на имеющееся значение. При попытке проверки содержимого неинициализированной переменной программа просто не пройдет компиляцию.

В Visual C# имеется тип данных под названием `bool`. В переменной типа `bool` может храниться одно из двух значений, `true` или `false`. Например, следующие три инструкции объявляют переменную типа `bool` по имени `areYouReady`, присваивают этой переменной значение `true`, а затем выводят ее значение на консоль:

```
bool areYouReady;
areYouReady = true;
Console.WriteLine(areYouReady); // запись True на консоль
```

Использование булевых операторов

Булевым называется оператор, выполняющий вычисление, результатом которого будет либо `true`, либо `false`. В C# есть несколько очень полезных булевых операторов, простейшим из которых является оператор `NOT`, представленный восклицательным знаком (!). Оператор `!` инвертирует булево значение, выдавая его противоположность. В предыдущем примере, если значение переменной `areYouReady` равно `true`, значение выражения `!areYouReady` будет `false`.

Основные сведения об операторах равенства и отношения

Двумя булевыми операторами, которыми вам придется пользоваться довольно часто, являются равенство (`==`) и неравенство (`!=`). Это бинарные операторы, с помощью которых можно определить, является ли значение таким же, как

и другое значение того же типа, получая при этом булев результат. Обобщенное представление о работе этих операторов при использовании в качестве примера `int`-переменной по имени `age` дано в табл. 4.1.

Таблица 4.1

Оператор	Значение	Пример	Результат, если значение age равно 42
<code>==</code>	Равно	<code>age == 100</code>	false
<code>!=</code>	Не равно	<code>age != 0</code>	true

Оператор равенства `==` не следует путать с оператором присваивания `=`. Выражение `x==y` сравнивает `x` с `y` и имеет значение `true`, если значения одинаковы. А выражение `x=y` присваивает значение переменной `y` переменной `x` и возвращает в качестве результата значение переменной `y`.

Ближайшими родственниками операторов `==` и `!=` являются операторы отношений. Эти операторы применяются для выяснения того, верно ли, что значение меньше или больше другого значения того же типа. Порядок использования этих операторов показан в табл. 4.2.

Таблица 4.2

Оператор	Значение	Пример	Результат, если значение age равно 42
<code><</code>	Меньше чем	<code>age < 21</code>	false
<code><=</code>	Меньше или равно	<code>age <= 18</code>	false
<code>></code>	Больше чем	<code>age > 16</code>	true
<code>>=</code>	Больше или равно	<code>age >= 30</code>	true

Основные сведения об условных логических операторах

C# предоставляет также два других бинарных булевых оператора: оператор логического И, представленный группой символов `&&`, и оператор логического ИЛИ, представленный группой символов `||`. Обобщенно они известны как *условные логические операторы*. Их целью является объединение двух булевых выражений или значений в один булев результат. Эти операторы похожи на операторы равенства и отношения тем, что значения выражений, в которых они появляются, бывают либо истинными, либо ложными, но отличаются тем, что значения, к которым они применяются, также должны быть либо истинными, либо ложными.

Результат использования оператора `&&` равен `true` только в том случае, если оба булева выражения вычисляются в `true`. Например, следующая инструкция присваивает значение `true` переменной `validPercentage` только в том случае, если значение переменной `percent` больше или равно 0 или меньше или равно 100:

```
bool validPercentage;
validPercentage = (percent >= 0) && (percent <= 100);
```



СОВЕТ Типичной ошибкой новичков является попытка объединить две проверки, указывая переменную `percent` лишь единожды:

```
percent >= 0 && <= 100 // эта инструкция не пройдет компиляцию
```

Избежать подобной ошибки, а также конкретизировать цель использования выражения помогают круглые скобки. Например, сравнение

```
validPercentage = percent >= 0 && percent <= 100
```

и сравнение

```
validPercentage = (percent >= 0) && (percent <= 100)
```

возвращают одно и то же значение, поскольку приоритет оператора `&&` ниже, чем операторов `>=` и `<=`. Но второе выражение сообщает о своей цели в более понятной форме.

Результат использования оператора `||` равен `true`, если любое из булевых выражений вычисляется в `true`. Оператор `||` используется для выяснения того, вычисляется ли любое из объединяемых им булевых выражений в `true`. Например, следующая инструкция присваивает значение `true` переменной `invalidPercentage` в том случае, если значение переменной `percent` меньше 0 или больше 100:

```
bool invalidPercentage;
invalidPercentage = (percent < 0) || (percent > 100);
```

Короткое замыкание

Оба оператора, как `&&`, так и `||`, проявляют свойство под названием *короткое замыкание*. Иногда при выяснении результата условного логического выражения вычислять оба операнда нет необходимости. Например, если левый operand оператора `&&` вычисляется в `false`, результат всего выражения должен быть `false` независимо от значения правого операнда. Аналогично этому, если значение левого операнда оператора `||` вычисляется в `true`, результат всего выражения должен быть `true` независимо от значения правого операнда. В этих случаях операторы `&&` и `||` пренебрегают вычислением правого операнда. Рассмотрим несколько примеров.

Если в выражении

```
(percent >= 0) && (percent <= 100)
```

значение переменной `percent` меньше нуля, булево выражение слева от оператора `&&` вычисляется в `false`. Это значение служит признаком того, что результат всего выражения должен быть `false`, и булево выражение справа от оператора `&&` не вычисляется.

Если в выражении

```
(percent < 0) || (percent > 100)
```

значение переменной `percent` меньше нуля, булево выражение слева от оператора `||` вычисляется в `true`. Это значение служит признаком того, что результат всего выражения должен быть `true`, и булево выражение справа от оператора `||` не вычисляется.

При осмотрительном составлении выражений, использующих условные логические операторы, можно ускорить выполнение кода, избегая ненужной работы. Помещать простые, легко вычисляемые булевые выражения следует слева от условного логического оператора, а более сложные — справа от него. Во многих случаях окажется, что вычислять более сложные выражения программе уже не нужно.

Сводная информация о приоритетности и ассоциативности операторов

В табл. 4.3 помещена сводная информация о приоритетности и ассоциативности всех уже знакомых вам операторов. Операторы в одной и той же категории имеют одинаковый уровень приоритета. Операторы тех категорий, которые расположены в таблице выше остальных, имеют приоритет над операторами тех категорий, которые расположены ниже.

Обратите внимание на то, что у операторов `&&` и `||` разные приоритеты: у `&&` приоритет выше, чем у `||`.

Использование инструкций if для принятия решений

Когда в методе нужно выбрать между выполнением двух различных инструкций в зависимости от результата вычисления булева выражения, можно воспользоваться инструкцией `if`.

Таблица 4.3

Категория	Оператор	Описание	Ассоциативность
Основные	()	Переопределение приоритета	Левая
	++	Постинкремент	
	--	Постдекремент	
Унарные	!	Логическое НЕ	Левая
	+	Возвращение значения операнда в неизменном виде	
	-	Возвращение значения операнда с отрицательным знаком	
Мультипликативные	++	Преинкремент	Левая
	--	Предекремент	
	*	Умножение	
Аддитивные	/	Деление	Левая
	%	Остаток целочисленного деления	
	+	Сложение	
Отношений	-	Вычитание	Левая
	<	Меньше чем	
	<=	Меньше или равно	
Равенства	>	Больше чем	Левая
	>=	Больше или равно	
	==	Равно	
Логического И	!=	Не равно	Левая
	&&	Условное И	
Логического ИЛИ		Условное ИЛИ	Левая
Присваивания	=	Присваивание значения правостороннего операнда левостороннему и возвращение присвоенного значения	Правая

Основные сведения о синтаксисе инструкции if

Для инструкции `if` используется следующий синтаксис (`if` и `else` являются ключевыми словами C#):

```
if ( booleanExpression )
    statement-1;
else
    statement-2;
```

Если булево выражение `booleanExpression` вычисляется в `true`, выполняется инструкция `statement-1`, в противном случае выполняется инструкция `statement-2`. Ключевое слово `else` и следующая за ним инструкция `statement-2` являются необязательными. Если условие `else` отсутствует и выражение `booleanExpression` вычисляется в `false`, выполнение продолжается, начиная с кода, следующего за инструкцией `if`. Обратите также внимание на то, что булево выражение должно быть заключено в круглые скобки, иначе код не пройдет компиляцию.

Рассмотрим, к примеру, инструкцию `if`, повышающую значение переменной, представляющей секундную стрелку секундомера, на единицу (минуты пока игнорируются). Если значение переменной `seconds` равно 59, оно переустанавливается на 0, в противном случае увеличивается на единицу путем использования оператора `++`:

```
int seconds;
...
if (seconds == 59)
    seconds = 0;
else
    seconds++;
```

ПОЖАЛУЙСТА, ПОЛЬЗУЙТЕСЬ ТОЛЬКО БУЛЕВЫМИ ВЫРАЖЕНИЯМИ!

Выражение в инструкции `if` должно быть заключено в круглые скобки. Кроме того, выражение должно быть булевым. В некоторых других языках, в частности С и С++, можно написать целочисленное выражение, и компилятор молча конвертирует целочисленное значение в `true` (если оно будет ненулевым) или в `false` (если оно будет нулевым). В C# данное поведение не поддерживается и при записи такого выражения компилятор выдает ошибку.

Если в инструкции `if` случайно указать вместо оператора проверки равенства (`==`) оператор присваивания (`=`), компилятор C# обнаружит вашу ошибку и, как показано в следующем примере, откажется от компиляции кода:

```
int seconds;
...
if (seconds = 59) // ошибка в ходе компиляции
...
if (seconds == 59) // все в порядке
```

Случайные присваивания были еще одним весьма распространенным источником ошибок в программах на С и С++, в которых присваиваемое значение (59) молча конвертировалось в булево выражение (при этом любое неотрицательное значение рассматривалось как `true`), в результате чего код, непосредственно следовавший за инструкцией `if`, неизменно выполнялся.

Кстати, для инструкции `if` вы можете в качестве выражения воспользоваться булевой переменной, но она, как показано в следующем примере, все равно должна быть заключена в круглые скобки:

```
bool inWord;
...
if (inWord == true) // допустимо, но используется крайне редко
...
if (inWord)          // используется чаще и считается более подходящей
                     разновидностью
```

Использование блоков для объединения инструкций в группы

Обратите внимание на то, что в синтаксисе ранее показанной инструкции `if` после `if (booleanExpression)` и после ключевого слова `else` указано только по одной инструкции. Временами, когда булево выражение вычисляется в `true`, требуется выполнить более одной инструкции. Можно, конечно, сгруппировать инструкции внутри нового метода, а затем вызвать этот метод, но проще все же сгруппировать инструкции внутри блока, представляющего собой простую последовательность инструкций, указанных между открывающей и закрывающей фигурными скобками.

В следующем примере две инструкции, одна из которых переустанавливает переменную `seconds` в нуль, а вторая увеличивает значение переменной `minutes` на единицу, сгруппированы внутри блока, и если значение `seconds` равно 59, выполняется весь блок:

```
int seconds = 0;
int minutes = 0;
...
if (seconds == 59)
{
    seconds = 0;
    minutes++;
}
else
{
    seconds++;
}
```



ВНИМАНИЕ Если не поставить фигурные скобки, компилятор C# свяжет с инструкцией `if` только первую инструкцию (`seconds = 0;`). Следующая за ней инструкция (`minutes++;`) при компиляции программы не будет распознана компилятором в качестве части инструкции `if`. Более того, когда компилятор дойдет до ключевого слова `else`, он не свяжет его с предыдущей инструкцией `if`, а выдаст вместо этого синтаксическую ошибку. Поэтому лучше всего всегда определять инструкции для каждой исполняемой ветви инструкции `if` внутри блока, даже если блок состоит только из одной инструкции. Если в дальнейшем потребуется добавить дополнительный код, это убережет вас от неприятностей.

В блоке также запускается новая область видимости. Внутри блока можно определить переменные, исчезающие сразу же после того, как будут выполнены содержащиеся в нем инструкции. Это положение иллюстрируется следующим фрагментом кода:

```
if (...)  
{  
    int myVar = 0;  
    ... // здесь myVar может использоваться  
} // а здесь myVar исчезает  
else  
{  
    // здесь myVar уже не может использоваться  
    ...  
}  
// и здесь myVar не может использоваться
```

Создание каскада из if-инструкций

Инструкции `if` могут находиться внутри других инструкций `if`. Таким образом можно собирать в цепочку последовательность булевых выражений, проверяемых одно за другим до тех пор, пока одно из них не будет вычислено в `true`. В следующем примере, если значение переменной `day` равно нулю, то первая проверка вычисляется в `true` и переменной `dayName` присваивается строковое значение "Sunday". Если значение `day` не равно нулю, первая проверка дает отрицательный результат и управление передается условию `else`, которое запускает вторую инструкцию `if` и сравнивает значение переменной `day` с единицей. Вторая инструкция `if` выполняется только в том случае, если результат первой проверки не является положительным. Аналогично этому третья инструкция `if` выполняется только в том случае, если с положительным результатом не проходит вторая проверка:

```
if (day == 0)  
{  
    dayName = "Sunday";  
}  
else if (day == 1)  
{
```

```
        dayName = "Monday";
    }
else if (day == 2)
{
    dayName = "Tuesday";
}
else if (day == 3)
{
    dayName = "Wednesday";
}
else if (day == 4)
{
    dayName = "Thursday";
}
else if (day == 5)
{
    dayName = "Friday";
}
else if (day == 6)
{
    dayName = "Saturday";
}
else
{
    dayName = "unknown";
}
```

В следующем упражнении вам предстоит создать метод, использующий каскад, созданный из инструкций `if`, для сравнения двух дат.

Запись инструкций `if`

Откройте в Microsoft Visual Studio 2015 проект `Selection`, который находится в папке `\Microsoft Press\VCSBS\Chapter 4\Selection` вашей папки документов.

Щелкните в меню Отладка на пункте **Начать отладку**. Среда Visual Studio 2015 выполнит сборку и запуск приложения. В форме будут выведены два элемента управления выбором даты под названиями `firstDate` и `secondDate`. В обоих элементах будет показана текущая дата. Щелкните на кнопке **Compare** (Сравнить).

В текстовом поле, находящемся в нижней части окна, появится следующий текст:

```
firstDate == secondDate : False
firstDate != secondDate : True
firstDate < secondDate : False
firstDate <= secondDate : False
firstDate > secondDate : True
firstDate >= secondDate : True
```

Булево выражение `firstDate == secondDate` будет вычислено в `true`, поскольку и в `firstDate`, и в `secondDate` установлена текущая дата. Похоже, что на самом

деле правильно сработали только оператор «меньше чем» и оператор «больше или равно». Работающее приложение показано на рис. 4.1.

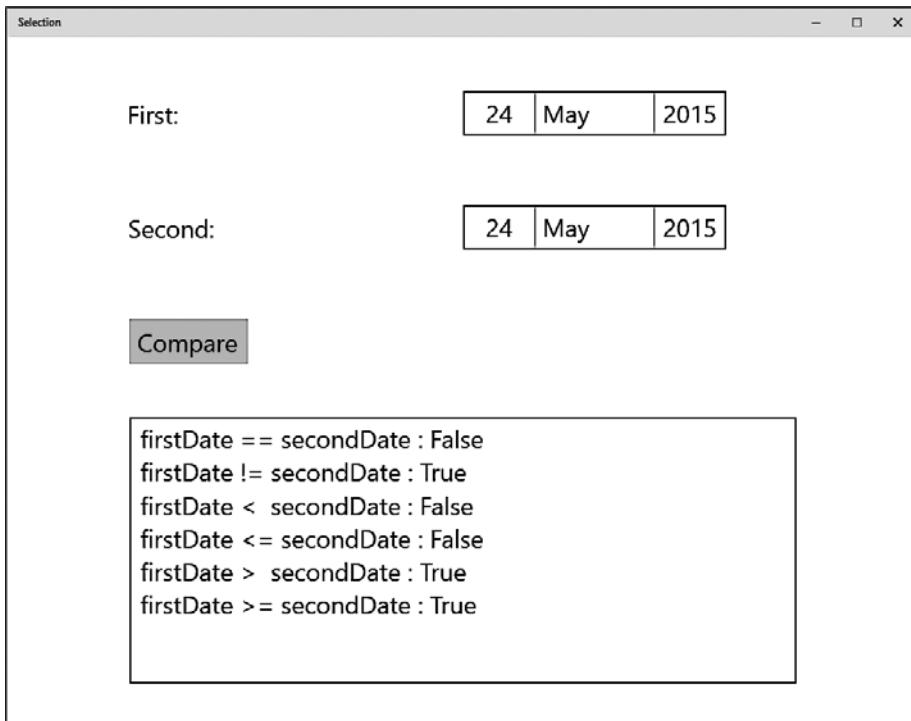


Рис. 4.1

Вернитесь в Visual Studio 2015. Щелкните в меню Отладка на пункте Остановить отладку. Выведите в окно редактора код файла MainPage.xaml.cs. Найдите метод compareClick, который должен выглядеть следующим образом:

```

private void compareClick(object sender, RoutedEventArgs e)
{
    int diff = dateCompare(firstDate.Date.LocalDateTime,
                           secondDate.Date.LocalDateTime);
    info.Text = "";
    show("firstDate == secondDate", diff == 0);
    show("firstDate != secondDate", diff != 0);
    show("firstDate < secondDate", diff < 0);
    show("firstDate <= secondDate", diff <= 0);
    show("firstDate > secondDate", diff > 0);
    show("firstDate >= secondDate", diff >= 0);
}

```

Этот метод запускается, когда пользователь щелкает в форме на кнопке Compare. В выражениях `firstDate.Date.LocalDateTime` и `secondDate.Date.LocalDateTime`

содержатся значения типа `DateTime`, представляющие даты, отображаемые в форме в элементах управления `firstDate` и `secondDate` в другом месте приложения. Тип данных `DateTime` относится к еще одному типу, как и `int` или `float`, кроме того что он содержит субэлементы, позволяющие получить доступ к отдельным частям даты, таким как год, месяц или день.

Метод `compareClick` передает методу `dateCompare` два значения типа `DateTime`. Целями этого метода являются сравнение дат и возвращение `int`-значения 0, если они одинаковые, `-1` — если первая дата меньше второй и `+1` — если первая дата больше второй. Дата считается больше другой даты, если хронологически она наступает позже. Метод `dateCompare` мы изучим на следующем этапе.

Метод `show` выводит результаты сравнения в текстовую область `info`, которая находится в нижней половине формы.

Найдите метод `dateCompare`, который должен выглядеть следующим образом:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    // TO DO
    return 42;
}
```

На данный момент этот метод при вызове, независимо от значения своих параметров, вместо 0, `-1` или `+1` возвращает одно и то же значение. Этим и объясняется, почему приложение работает не так, как от него ожидалось. Чтобы правильно сравнить две даты, необходимо реализовать в методе соответствующую логику.

Удалите из метода `dateCompare` комментарий `// TO DO` и инструкцию `return`. Добавьте к телу метода `dateCompare` следующие инструкции, выделенные жирным шрифтом:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    int result = 0;

    if (leftHandSide.Year < rightHandSide.Year)
    {
        result = -1;
    }
    else if (leftHandSide.Year > rightHandSide.Year)
    {
        result = 1;
    }
}
```



ПРИМЕЧАНИЕ Не пытайтесь именно сейчас выполнять сборку приложения. Метод `dateCompare` еще не имеет законченного вида, и сборка не состоится.

Если выражение `leftHandSide.Year < rightHandSide.Year` вычисляется в `true`, дата в переменной `leftHandSide` должна по хронологии наступить раньше даты в переменной `rightHandSide` и программа устанавливает значение переменной `result` в `-1`. В противном случае, если выражение `leftHandSide.Year > rightHandSide.Year` вычисляется в `true`, дата в переменной `leftHandSide` должна наступить позже даты в переменной `rightHandSide` и программа устанавливает значение переменной `result` в `1`.

Если выражение `leftHandSide.Year < rightHandSide.Year` вычисляется в `false` и выражение `leftHandSide.Year > rightHandSide.Year` также вычисляется в `false`, то свойство `Year` обеих дат должно иметь одинаковое значение и программе следует сравнить месяцы каждой из дат.

Добавьте к телу метода `dateCompare` следующие инструкции, выделенные жирным шрифтом. Наберите их после кода, введенного на предыдущем этапе:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    ...
    else if (leftHandSide.Month < rightHandSide.Month)
    {
        result = -1;
    }
    else if (leftHandSide.Month > rightHandSide.Month)
    {
        result = 1;
    }
}
```

Эти инструкции сравнивают месяцы, следуя той же логике, которая использовалась при сравнении годов на предыдущем этапе.

Если выражение `leftHandSide.Month < rightHandSide.Month` вычисляется в `false` и выражение `leftHandSide.Month > rightHandSide.Month` также вычисляется в `false`, то свойства `Month` обеих дат должны иметь одинаковые значения, следовательно, в завершение программы нужно сравнить дни в каждой из дат.

После кода, введенного на двух предыдущих этапах, добавьте к телу метода `dateCompare` следующие инструкции, выделенные жирным шрифтом, :

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    ...
    else if (leftHandSide.Day < rightHandSide.Day)
    {
        result = -1;
    }
    else if (leftHandSide.Day > rightHandSide.Day)
    {
        result = 1;
```

```
    }
    else
    {
        result = 0;
    }

    return result;
}
```

Теперь вы уже должны разбираться в построении этой логики. Если оба выражения, `leftHandSide.Day < rightHandSide.Day` и `leftHandSide.Day > rightHandSide.Day`, вычисляются в `false`, значения свойств `Day` обеих переменных должны быть одинаковыми. Чтобы логика программы дошла до этого места, значения `Month` и значения `Year` также должны быть идентичны, следовательно, обе даты должны быть одинаковыми и программа устанавливает значение переменной `result` в нуль.

Завершающая инструкция возвращает значение, сохраненное в переменой `result`.

Щелкните в меню Отладка на пункте Начать отладку. Приложение будет собрано заново и запущено на выполнение. Щелкните на кнопке Compare. В текстовой области появится следующий текст:

```
firstDate == secondDate : True
firstDate != secondDate : False
firstDate < secondDate: False
firstDate <= secondDate: True
firstDate > secondDate: False
firstDate >= secondDate: True
```

Для одинаковых дат будут выведены абсолютно правильные результаты.

Воспользуйтесь элементом управления типа `DatePicker` и выберите для второй даты большее значение, после чего щелкните на кнопке Compare. В текстовой области появится следующий текст:

```
firstDate == secondDate: False
firstDate != secondDate: True
firstDate < secondDate: True
firstDate <= secondDate: True
firstDate > secondDate: False
firstDate >= secondDate: False
```

И снова первая дата по хронологии наступает раньше второй — это абсолютно правильные результаты.

Протестируйте работу приложения для каких-нибудь других дат и убедитесь в том, что результаты соответствуют вашим ожиданиям. Когда закончите, вернитесь в Visual Studio 2015 и остановите отладку.

СРАВНЕНИЕ ДАТ В РЕАЛЬНЫХ ПРИЛОЖЕНИЯХ

Теперь, когда вы увидели, как используются довольно длинные и сложные серии инструкций if и else, должен заметить, что в реальных приложениях для сравнения дат такой прием использоваться не будет. Если взглянуть на метод dateCompare из предыдущего упражнения, то в нем можно увидеть два параметра, leftHandSide и rightHandSide, значения которых относятся к типу данных DateTime. Логика, которую вы записывали, сравнивает только ту часть параметров, которая относится к дате, но в них содержится и элемент времени, который вами не рассматривался (и не показывался). Чтобы два значения типа DateTime считались одинаковыми, у них должны быть не только одинаковые даты, но и одинаковое время. Сравнение дат и времени является настолько распространенной операцией, что в типе DateTime имеется встроенный метод под названием Compare, который занимается следующим: берет два аргумента типа DateTime и сравнивает их, возвращая значение, показывающее, меньше ли первый аргумент, чем второй, в случае чего результатом будет отрицательное значение; больше ли первый аргумент, чем второй, в случае чего результатом будет положительное значение; или оба аргумента представляют одни и те же дату и время, в случае чего результатом будет нуль.

Использование инструкций switch

Иногда при написании каскада из if-инструкций выясняется, что все эти инструкции похожи друг на друга, поскольку вычисляют сходные выражения. Разница между ними лишь в том, что каждая из if-инструкций сравнивает результат выражения с другим значением. Рассмотрим, к примеру, следующий блок кода, использующий инструкцию if для проверки значения переменной day и определения того, к какому дню недели оно относится:

```
if (day == 0)
{
    dayName = "Sunday";
}
else if (day == 1)
{
    dayName = "Monday";
}
else if (day == 2)
{
    dayName = "Tuesday";
}
else if (day == 3)
{
    ...
}
else
{
    dayName = "Unknown";
}
```

Зачастую в подобных ситуациях можно вместо каскада `if`-инструкций воспользоваться инструкцией `switch`, повысив эффективность и удобочитаемость программы.

Основные сведения о синтаксисе инструкции `switch`

Для инструкции `switch` используется следующий синтаксис (`switch`, `case` и `default` являются ключевыми словами):

```
switch ( controllingExpression )
{
    case constantExpression :
        statements
        break;
    case constantExpression :
        statements
        break;
    ...
    default :
        statements
        break;
}
```

Выражение `controllingExpression`, которое должно быть заключено в круглые скобки, вычисляется только один раз. Затем управление переходит к блоку кода, определяемому выражением `constantExpression`, чье значение равно результату вычисления выражения `controllingExpression`. (Идентификатор `constantExpression` также называют меткой альтернативы.) Код выполняется вплоть до инструкции `break`, на которой инструкция `switch` завершает свою работу, и программа продолжает работу, выполняя инструкцию, которая стоит сразу же за закрывающей фигурной скобкой инструкции `switch`. Если ни одно из значений `constantExpression` не равно значению `controllingExpression`, выполняются инструкции, находящиеся ниже необязательной метки `default`.



ПРИМЕЧАНИЕ Каждое значение `constantExpression` должно быть уникальным, чтобы `controllingExpression` могло соответствовать только одному из них. Если значение `controllingExpression` не соответствует какому-либо значению `constantExpression`, а метка `default` отсутствует, выполнение программы продолжается с первой инструкции, следующей за закрывающей фигурной скобкой инструкции `switch`.

Предыдущий каскад из `if`-инструкций можно переписать, воспользовавшись следующей инструкцией `switch`:

```
switch (day)
{
    case 0 :
        dayName = "Sunday";
        break;
    case 1 :
```

```

dayName = "Monday";
break;
case 2 :
    dayName = "Tuesday";
    break;
...
default :
    dayName = "Unknown";
    break;
}

```

Выполнение правил использования инструкции switch

При всем несомненном удобстве инструкции `switch`, к сожалению, ее нельзя применять везде, где заблагорассудится. Любая применяемая `switch`-инструкция должна отвечать следующим правилам.

- ❑ Инструкцию `switch` можно применять только к конкретным типам данных, а именно к `int`, `char` или `string`. С любыми другими типами (включая `float` и `double`) следует использовать инструкцию `if`.
- ❑ Метки альтернатив должны быть выражениями, представленными константами, например, `42`, если типом данных инструкции `switch` является `int`, '`'4'`', если типом данных является `char`, или "`42`", если типом данных является `string`. Если метку альтернативы нужно вычислить во время выполнения программы, следует воспользоваться инструкцией `if`.
- ❑ Метки альтернатив должны быть уникальными выражениями. Иными словами, две метки альтернатив не могут иметь одно и то же значение.
- ❑ Вы можете указать, что хотите выполнить одинаковые инструкции для более чем одного значения, предоставив список меток альтернатив, не прерываемый инструкциями, в случае чего код для завершающей метки списка выполняется для всех указанных в списке меток. Но если у метки имеется одна и больше связанных с ней инструкций, выполнение не может быть передано дальше вниз следующим меткам, и в таком случае компилятор выдает ошибку. Все эти положения проиллюстрированы в следующем фрагменте кода:

```

switch (trumps)
{
    case Hearts :
    case Diamonds :      // Передача управления вниз разрешена, поскольку
                          // код между метками отсутствует
        color = "Red"; // код выполняется для меток Hearts и Diamonds
        break;
    case Clubs :
        color = "Black";
    case Spades : // Ошибка – код между метками
        color = "Black";
        break;
}

```



ПРИМЕЧАНИЕ Наиболее распространенным способом остановки передачи управления следующим меткам альтернатив является применение инструкции `break`, но для выхода из метода, содержащего инструкцию `switch`, можно также воспользоваться инструкцией `return` или `throw`, выдающей исключение и прекращающей выполнение инструкции `switch`. Инструкция `throw` рассматривается в главе 6 «Обработка ошибок и исключений».

ПРАВИЛА ПЕРЕДАЧИ УПРАВЛЕНИЯ ВНУТРИ ИНСТРУКЦИИ SWITCH

Поскольку случайная передача управления от одной метки альтернатив следующей такой метке при наличии какого-либо кода в промежутке между ними невозможна, вы можете вполне свободно переставлять секции инструкции `switch`, что не оказывает на ее работу никакого влияния (включая и метку `default`, которая по соглашению обычно ставится в качестве последней метки, но именно на этом месте стоять не обязана).

Программисты, работающие на С и С++, должны заметить, что инструкция `break` является обязательной для каждой метки в инструкции `switch` (даже для метки `default`). Это требование имеет свою положительную сторону, так как в программах на С или С++ программисты часто забывают ставить инструкцию `break`, позволяя программе продолжать выполнение с передачей управления на следующую метку, что приводит к трудно обнаруживаемым ошибкам.

При желании вы можете имитировать такую же передачу управления другой метке, как в С и С++, и в программе на С#, для чего нужно воспользоваться инструкцией `goto` для перехода к следующей метке альтернатив или к метке `default`. Но вообще-то использовать `goto` не рекомендуется, и в этой книге такой прием не демонстрируется.

В следующем упражнении вам предстоит завершить программу, считывающую символы из строки и отображающую каждый символ на его XML-представление. Например, символ левой угловой скобки (`<`) имеет в XML специальное значение (используется для формирования элементов). Если в данных содержится этот символ, он должен быть переведен в свое текстовое представление `<`, чтобы XML-процессор знал, что это данные, а не часть XML-инструкции. Похожие правила применяются к символам правой угловой скобки (`>`), амперсанда (`&`), одинарной кавычки (`'`) и двойной кавычки (`"`). Вам будет создана инструкция `switch`, проверяющая значение символа и перехватывающая специальные символы XML, фигурирующие в качестве меток альтернатив.

Запись инструкций switch

Откройте в Microsoft Visual Studio 2015 проект `SwitchStatement`, который находится в папке `\Microsoft Press\VCSSBS\Chapter 4\SwitchStatement` вашей папки документов.

Щелкните в меню Отладка на пункте Начать отладку. Среда Visual Studio 2015 выполнит сборку и запуск приложения, которое выведет на экран форму, содержащую две текстовые области, разделенные кнопкой Copy (Копировать).

Наберите в верхней текстовой области следующий образец текста:

```
inRange = (lo <= number) && (hi >= number);
```

Щелкните на кнопке Copy.

Инструкция, как показано на рис. 4.2, будет скопирована в нижнюю текстовую область в неизменном виде, без перевода символов <, & и > в их представления.

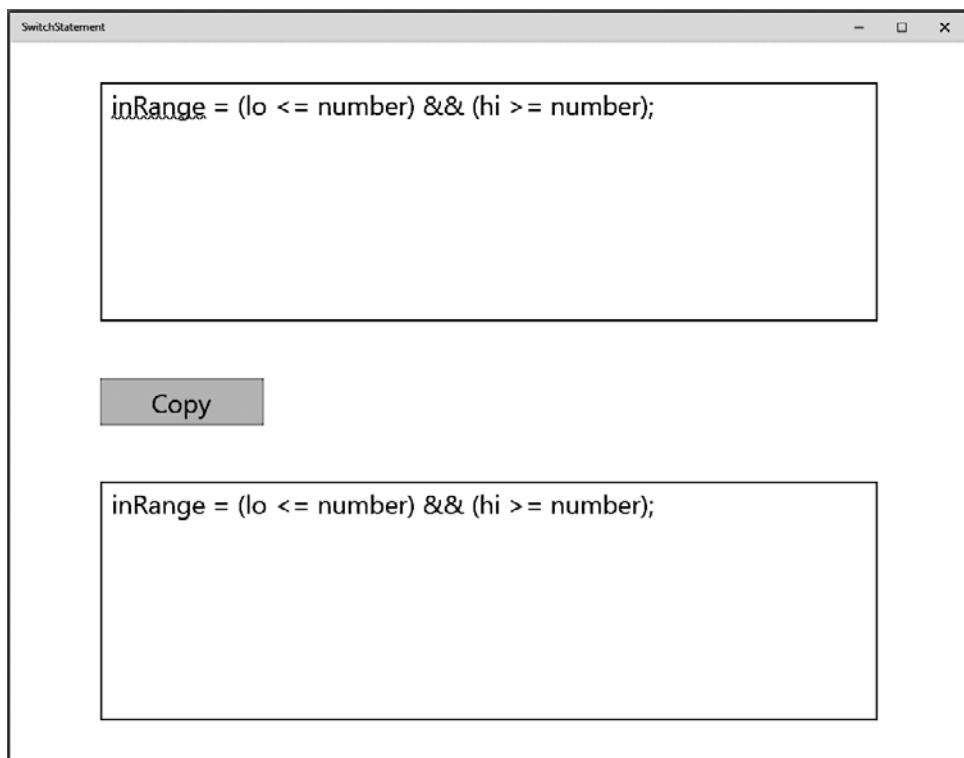


Рис. 4.2

Вернитесь в Visual Studio 2015 и остановите отладку.

Выполните в окне редактора код файла MainPage.xaml.cs и найдите в нем метод copyOne, который копирует символ, указанный в качестве его входного параметра, в конец текста, выводимого в нижнюю текстовую область. На данный момент в copyOne содержится инструкция switch с единственной меткой альтернативы

`default`. На следующих нескольких этапах вы внесете в эту инструкцию `switch` изменения, позволяющие переводить символы, имеющие для XML особое значение, в их отображение в языке XML. Например, символ `<` будет преобразован в строку `<`.

Добавьте к инструкции `switch` после ее открывающей фигурной скобки и непосредственно перед меткой `default` следующие инструкции, выделенные жирным шрифтом:

```
switch (current)
{
    case '<' :
        target.Text += "&lt;";
        break;
    default:
        target.Text += current;
        break;
}
```

Если текущим копируемым символом будет левая угловая скобка (`<`), предыдущий код добавит к тексту, выводимому на свое место, строку `"<"`.

Добавьте после только что добавленной инструкции `break` и выше метки `default` следующие инструкции:

```
case '>' :
    target.Text += "&gt;";
    break;
case '&' :
    target.Text += "&";
    break;
case '\'' :
    target.Text += "&#34;";
    break;
case '\"' :
    target.Text += "&#39;";
    break;
```



ПРИМЕЧАНИЕ Одинарная кавычка (`'`) и двойная кавычка (`"`) имеют в C# специальное значение — они используются в качестве разделителей символов и строковых констант. Обратный слеш (`\`) в последних двух метках альтернатив является символом отмены специального действия, заставляющим компилятор C# рассматривать эти символы в качестве литералов, а не разделителей.

Щелкните в меню Отладка на пункте Начать отладку. Наберите в верхней текстовой области:

```
inRange = (lo <= number) && (hi >= number);
```

Щелкните на кнопке Copy.

Инструкция будет скопирована в нижнюю текстовую область. На этот раз в XML-отображение, реализованное в инструкции `switch`, переводится каждый символ. В целевом текстовом поле отобразится следующее:

```
inRange = (lo <= number) && (hi >= number);
```

Поэкспериментируйте с другими строками и убедитесь в том, что все специальные символы (<, >, &, " и ') обрабатываются должным образом.

Вернитесь в Visual Studio и остановите отладку.

Выходы

В данной главе вы узнали, что такое булевые выражения и переменные, увидели, как используются булевые выражения с инструкциями `if` и `switch` для принятия решений в ваших программах, и объединили булевые выражения с помощью булевых операторов.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и перейдите к главе 5 «Использование инструкций составного присваивания и итераций».

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее	Пример
Определить равенство двух значений	Воспользуйтесь оператором <code>==</code> или оператором <code>!=</code>	<code>answer == 42</code>
Сравнить значения двух выражений	Воспользуйтесь оператором <code><</code> , <code><=</code> , <code>></code> или <code>>=</code>	<code>age >= 21</code>
Объявить булеву переменную	Воспользуйтесь в качестве типа переменной ключевым словом <code>bool</code>	<code>bool inRange;</code>
Создать булево выражение, вычисляемое в <code>true</code> только в случае, когда оба условия вычисляются в <code>true</code>	Воспользуйтесь оператором <code>&&</code>	<code>inRange = (lo <= number) && (number <= hi);</code>

Чтобы	Сделайте следующее	Пример
Создать булево выражение, вычисляемое в true, если любое из двух условий вычисляется в true	Воспользуйтесь оператором	<code>outOfRange = (number < lo) (hi < number);</code>
Выполнить инструкцию, если условие вычисляется в true	Воспользуйтесь инструкцией if	<code>if (inRange) process();</code>
Выполнить более одной инструкции, если условие вычисляется в true	Воспользуйтесь инструкцией if и блоком	<code>if (seconds == 59) { seconds = 0; minutes++; }</code>
Связать различные инструкции с различными значениями управляющего выражения	Воспользуйтесь инструкцией switch	<code>switch (current) { case 0: ... break; case 1: ... break; default : ... break; }</code>

5

Использование инструкций составного присваивания и итераций

Прочитав эту главу, вы научитесь:

- обновлять значение переменной с использованием операторов составного присваивания;
- записывать инструкции `while`, `for` и `do`;
- останавливать программу в ходе выполнения инструкции `do` и наблюдать за изменением значений переменных.

Из главы 4 «Использование инструкций принятия решений» вы узнали, как для выборочного запуска инструкций используются конструкции `if` и `switch`. В этой главе вы узнаете, как использовать различные инструкции итераций (или циклов) для повторных запусков одной или нескольких инструкций.

При запуске инструкций итераций требуется, как правило, средство управления количеством выполненных итераций. Этого можно достичь с помощью переменных, обновления при выполнении каждой итерации и остановки процесса по достижении переменной конкретного значения. Чтобы упростить этот процесс, начнем с изучения специальных операторов присваивания, которые будут использоваться для обновления значения переменной в подобных обстоятельствах.

Использование операторов составного присваивания

Порядок использования арифметических операторов для создания новых значений вам уже известен. К примеру, в следующей инструкции для вывода на консоль значения на 42 единицы больше значения переменной `answer` используется оператор «плюс» (+):

```
Console.WriteLine(answer + 42);
```

Вы также знаете, как для изменения значения переменной используются операторы присваивания. В следующей инструкции для изменения значения `answer` на 42 используется оператор присваивания (=):

```
answer = 42;
```

Если нужно прибавить 42 к значению переменной, можно использовать сочетание оператора присваивания и оператора +. Например, в следующей инструкции к `answer` добавляются 42 единицы. После запуска этой инструкции значение `answer` становится на 42 единицы больше, чем было до того:

```
answer = answer + 42;
```

Эта инструкция вполне справляется со своей задачей, но вам, наверное, не встречались опытные программисты, записывающие код таким образом. Добавление значения к переменной происходит настолько часто, что в C# вам предоставляется способ выполнения этой задачи в более лаконичной записи с использованием оператора +=. Для добавления 42 к `answer` можно воспользоваться следующей инструкцией:

```
answer += 42;
```

В табл. 5.1 показано, что эта форма записи может использоваться для объединения с оператором присваивания любых арифметических операторов. Обобщенно такие операторы называются операторами составного присваивания.

Таблица 5.1

Вместо этой формы записи	Используйте эту форму записи
переменная = переменная * число;	переменная *= число;
переменная = переменная / число;	переменная /= число;
переменная = переменная % число;	переменная %= число;
переменная = переменная + число;	переменная += число;
переменная = переменная - число;	переменная -= число;



СОВЕТ Операторы составного присваивания пользуются таким же уровнем приоритета и имеют такую же ассоциативность, что и простой оператор присваивания (=).

Оператор += применяется также в отношении строк, он добавляет одну строку к концу другой строки. Например, при выполнении следующего кода на консоль выводится приветствие «Hello John»:

```
string name = "John";
string greeting = "Hello ";
greeting += name;
Console.WriteLine(greeting);
```

Никакие другие операторы составного присваивания со строками использоваться не могут.



СОВЕТ Вместо операторов составного присваивания, увеличивающих или уменьшающих значение переменной на единицу, следует использовать операторы инкремента (++) и декремента (--). Например, вместо

```
count += 1;
следует использовать
count++;
```

Запись инструкций while

Инструкция `while` используется для повторных запусков инструкции до тех пор, пока заданное условие вычисляется в `true`. Для нее используется следующий синтаксис:

```
while ( булево выражение )
    инструкция
```

Вычисляется булево выражение, которое нужно заключить в круглые скобки, и если его значение равно `true`, инструкция выполняется, а затем булево выражение вычисляется еще раз. Если выражение по-прежнему вычисляется в `true`, выполнение инструкции повторяется, а затем булево выражение вычисляется еще раз. Этот процесс продолжается, пока булево выражение не будет вычислено в `false`, и тогда происходит выход из инструкции `while`. Затем выполнение продолжается с первой инструкции, следующей за инструкцией `while`. Для инструкции `while` используется тот же синтаксис, что и для инструкции `if` (фактически синтаксис идентичен, за исключением ключевого слова):

- ❑ Выражение должно быть булевым.
- ❑ Булево выражение должно быть заключено в круглые скобки.

- ❑ Если булево выражение с первого же раза вычисляется в `false`, инструкция не выполняется.
- ❑ Если под управлением инструкции `while` нужно выполнить две и более инструкции, их следует сгруппировать в блок.

Посмотрите на инструкцию `while`, записывающую в консоль значения от 0 до 9. Обратите внимание на то, что как только переменная `i` достигнет значения 10, выполнение инструкции `while` завершается и код блока не выполняется:

```
int i = 0;
while (i < 10)
{
    Console.WriteLine(i);
    i++;
}
```

Когда-либо выполнение всех инструкций `while` должно завершаться. Новички довольно часто забывают включать инструкцию, вызывающую со временем вычисление булева выражения в `false` и прекращение цикла, из-за чего программа входит в режим бесконечного выполнения. В примере роль такой инструкции выполняет `i++;`.



ПРИМЕЧАНИЕ Переменная `i` в цикле `while` управляет количеством выполняемых в нем итераций. Это общая идиома, и переменную, играющую эту роль, иногда называют переменной-ограничителем. Можно также создавать вложенные циклы (один цикл внутри другого), и в этих случаях обычно эту схему имен расширяют, используя в качестве имен переменных-ограничителей, применяемых для управления итерациями в этих циклах, буквы `j`, `k` и даже `l`.



СОВЕТ Точно так же, как и при использовании инструкций `if`, с инструкцией `while` рекомендуется всегда использовать блок, даже если в нем содержится всего одна инструкция. Тогда, если позже будет принято решение о добавлении к телу конструктора `while` еще одной инструкции, будет совершенно ясно, что ее нужно добавлять к блоку. Если этого не сделать, то в качестве части цикла в конструкции `while` будет выполнена только та инструкция, которая следует непосредственно за булевым выражением, что приведет к возникновению трудно обнаруживаемых ошибок, похожих на те, которые возникают при использовании следующего кода:

```
int i = 0;
while (i < 10)
    Console.WriteLine(i);
    i++;
```

Этот код будет выполняться в цикле до бесконечности, выводя на консоль бесчисленное количество нулей, поскольку в качестве части конструкции `while` будет выполняться инструкция `Console.WriteLine`, но не будет выполняться инструкция `i++`.

В следующем упражнении вам предстоит написать цикл `while` для построчного перебора содержимого текстового файла и записи каждой его строки в текстовое поле формы.

Запись инструкции while

Откройте в Microsoft Visual Studio 2015 проект WhileStatement, который находится в папке \Microsoft Press\VCSBS\Chapter 5 \WhileStatement вашей папки документов.

Щелкните в меню Отладка на пункте Начать отладку. Среда Visual Studio 2015 выполнит сборку и запуск приложения, представляющего собой простой просмотрщик текстового файла, которым можно воспользоваться для выбора файла и вывода на экран его содержимого.

Щелкните на кнопке Open File (Открыть файл). Появится окно выбора файла, в котором, как показано на рис. 5.1, будут отображены файлы, находящиеся в папке документов (перечень файлов и папок может отличаться от того, который выводится на вашем компьютере).

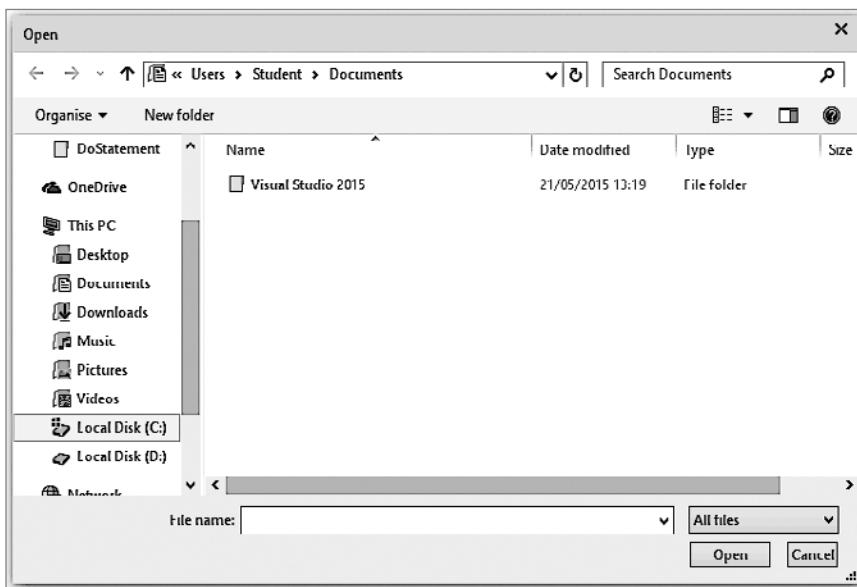


Рис. 5.1

Этим диалоговым окном можно воспользоваться для перехода в папку и выбора файла для вывода на экран. Перейдите в папке документов в папку \Microsoft Press\VCSBS\Chapter 5\WhileStatement\WhileStatement. Выберите файл MainPage.xaml.cs и щелкните на кнопке Открыть.

В текстовом поле в верхней части формы появится имя файла, MainPage.xaml.cs, но содержимое этого файла в более крупном поле не появится. Дело в том, что вами еще не создан код, считывающий содержимое файла и отображающий его на экране. Эти функции будут добавлены на следующих этапах.

Вернитесь в Visual Studio 2015 и остановите отладку. Выведите код для файла `MainPage.xaml.cs` в окно редактора и найдите метод `openFileClick`.

Этот метод запускается, когда пользователь щелкает на кнопке Открыть в диалоговом окне открытия файла. Разбираться в подробностях работы этого метода вам пока не нужно, просто уясните тот факт, что этот метод выводит предложение пользователю выбрать файл (в панели выбора файла `FileOpenPicker` или в окне `OpenFileDialog`) и открывает выбранный файл для чтения.

Для нас важны две последние инструкции метода `openFileClick`, имеющие следующий вид:

```
TextReader reader = new StreamReader(inputStreamAsStreamForRead());
displayData(reader);
```

В первой инструкции объявляется `TextReader`-переменная по имени `reader`. `TextReader` — это класс, предоставляемый средой Microsoft.NET Framework, которым можно воспользоваться для чтения потоков символов из таких источников, как файлы. Он находится в пространстве имен `System.IO`. Эта инструкция открывает объекту `TextReader`, который затем может использоваться для чтения данных из файла, доступ к данным файла, указанного пользователем в панели `FileOpenPicker`. Последняя инструкция вызывает метод по имени `displayData`, передавая ему в качестве параметра переменную `reader`. Метод `displayData`, используя объект `reader`, считывает данные и выводит их на экран (или же он станет это делать, как только вы запишете код для выполнения этих действий).

Изучите метод `displayData`. Пока он выглядит следующим образом:

```
private void displayData(TextReader reader)
{
    // TODO: add while loop here
}
```

Как видите, если не считать комментария, этот метод пуст. В него нужно добавить код для извлечения данных и вывода их на экран.

Замените комментарий `// TODO: add while loop here` следующей инструкцией:

```
source.Text = "";
```

Переменная `source` ссылается на большое текстовое поле формы. Установка для ее свойства `Text` значения пустой строки ("") удаляет любой текст, отображавшийся перед этим в данном текстовом поле.

Добавьте после предыдущей добавленной к методу `displayData` строки следующую инструкцию:

```
string line = reader.ReadLine();
```

Она объявляет строковую переменную по имени `line` и, чтобы считать в эту переменную первую строку из файла, вызывает метод `reader.ReadLine`. Этот метод возвращает либо следующую строку текста из файла, либо, когда больше нет строк для считывания, специальное значение, которое называется `null`.

Добавьте к методу `displayData` после только что введенного кода следующие инструкции:

```
while (line != null)
{
    source.Text += line + '\n';
    line = reader.ReadLine();
}
```

Это цикл `while`, производящий последовательный построчный перебор содержимого файла, пока строки не закончатся.

Булево выражение в начале цикла `while` проверяет значение переменной `line`. Если оно не равно `null`, тело цикла выводит текущую строку текста, добавляя ее, а вместе с ней и символ новой строки ('`\n`' — метод `ReadLine` объекта `TextReader` при считывании каждой новой строки удаляет символы новой строки, поэтому код нуждается в том, чтобы они были возвращены в конец строки), к свойству `Text` исходного текстового поля. Затем, прежде чем выполнить следующую итерацию, цикл `while` считывает новую строку текста. Выполнение цикла `while` завершается, когда в файле заканчивается текст для считывания, и метод `ReadLine` возвращает значение `null`.

Наберите после закрывающей фигурной скобки в конце цикла `while` следующую инструкцию:

```
reader.Dispose();
```

Эта инструкция закрывает файл и высвобождает связанные с ним ресурсы. Польза от ее применения в том, что теперь файлом смогут воспользоваться другие приложения, кроме того, будут высвобождены память и другие ресурсы, задействованные для доступа к файлу.

Щелкните в меню Отладка на пункте Начать отладку. Как только появится форма, щелкните на кнопке Open File. В панели выбора файла или в диалоговом окне его открытия перейдите к папке `\Microsoft Press\VCBS\Chapter 5\WhileStatement\WhileStatement` в вашей папке документов, выберите файл `MainPage.xaml.cs`, а затем щелкните на кнопке Открыть.



ПРИМЕЧАНИЕ Не пытайтесь открыть файл, не содержащий текст. К примеру, при попытке открыть исполняемую программу или графический файл приложение просто покажет текстовое представление содержащейся в этом файле двоичной информации. Если размер файла довольно большой, он может «подвесить» приложение, что потребует от вас принудительного завершения его работы.

На этот раз содержимое выбранного файла появится в текстовом поле, и вы должны узнать отредактированный вами код. При запуске приложение должно вывести следующее изображение (рис. 5.2).

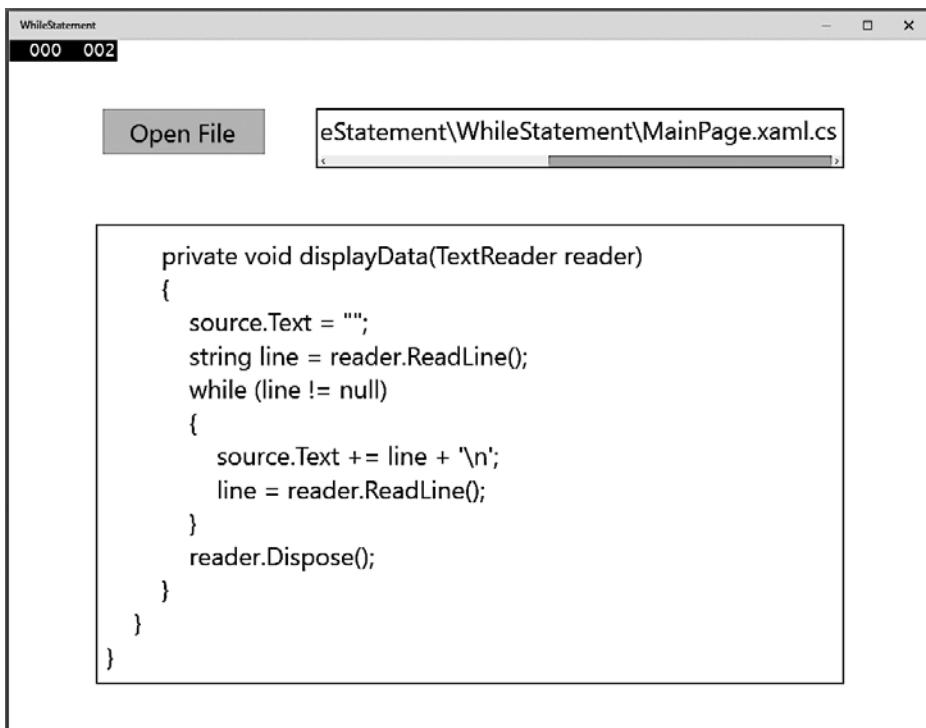


Рис. 5.2

Прокрутите текст в поле и найдите метод `displayData`. Убедитесь, что в нем содержится только что добавленный вами код.

Вернитесь в Visual Studio и остановите отладку.

Запись инструкций `for`

В C# большинство записей инструкций `while` имеют следующую общепринятую структуру:

```
инициализация
while (булево выражение)
{
    инструкция
    обновляемая управляемая переменная
}
```

Инструкция `for` в C# предоставляет более формализованную версию конструкции данного типа, в которой объединены инициализация, булево выражение и код, обновляющий значение управляющей переменной. Несомненная польза от применения инструкции `for` заключается в том, что при этом практически невозможно случайно не указать код, инициализирующий или обновляющий управляющую переменную, поэтому вам вряд ли удастся написать код бесконечного цикла. Синтаксис инструкции `for` имеет следующий вид:

```
for (инициализация; булево выражение; обновляемая управляющая переменная)
    инструкция
```

Инструкция, играющая роль тела конструкции `for`, может быть одной строкой кода или кодовым блоком, заключенным в фигурные скобки.

Показанный ранее цикл `while`, с помощью которого на экран выводились целые числа от 0 до 9, можно переделать в следующий цикл `for`:

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

Инициализация проводится только один раз, в самом начале цикла. Затем, если булево выражение вычисляется в `true`, запускается инструкция. Происходит обновление управляющей переменной, а затем булево выражение вычисляется еще раз. Если условие по-прежнему вычисляется в `true`, инструкция снова выполняется, управляющая переменная обновляется, булево выражение снова вычисляется и т. д.

Обратите внимание на то, что инициализация происходит только один раз, инструкция в теле цикла всегда выполняется перед обновлением переменной, а это обновление происходит перед новым вычислением булева выражения.



СОВЕТ Считается, что лучше, как и в инструкции `while`, всегда использовать кодовый блок, даже если в теле цикла `for` содержится всего одна инструкция. Тогда позже при добавлении к телу дополнительных инструкций вы обеспечите непременное выполнение своего кода в качестве части каждой из итераций.

Любая из трех частей инструкции `for` может быть опущена. Если опустить булево выражение, то условие всегда будет вычисляться в `true` и следующий цикл `for` станет выполняться бесконечно:

```
for (int i = 0; ;i++)
{
    Console.WriteLine("somebody stop me!");
}
```

Если опустить те части, где осуществляются инициализация и обновление, получится что-то вроде необычной записи цикла `while`:

```
int i = 0;
for (; i < 10; )
{
    Console.WriteLine(i);
i++;
}
```



ПРИМЕЧАНИЕ Части инструкции `for`, относящиеся к инициализации, булеву выражению и обновлению, всегда должны отделяться друг от друга точкой с запятой, даже если они опущены.

Циклу `for` можно также предоставить сразу несколько выражений инициализации и нескольких выражений обновления. (Но булево выражение должно быть только одно.) Для этого различные инициализации и обновления нужно отделить друг от друга запятыми, как показано в следующем примере:

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
    ...
}
```

В качестве заключительного примера посмотрим, как цикл `while` из предыдущего упражнения можно было бы переделать в цикл `for`:

```
for (string line = reader.ReadLine(); line != null; line = reader.ReadLine())
{
    source.Text += line + '\n';
}
```

Основные сведения об области видимости инструкции `for`

Возможно, вы уже заметили, что в инициализационной части инструкции `for` можно объявить переменную. Областью видимости этой переменной станет тело инструкции `for`, и она исчезнет, как только эта инструкция завершит выполнение. Это правило имеет два важных следствия. Во-первых, эту переменную невозможно использовать после завершения выполнения инструкции `for`, поскольку в области видимости ее больше нет. Посмотрите на следующий пример:

```
for (int i = 0; i < 10; i++)
{
    ...
}
Console.WriteLine(i); // ошибка в ходе компиляции
```

Во-вторых, можно записывать две и более инструкции `for`, заново использующие переменную с одним и тем же именем, поскольку, как показано в следующем примере кода, каждая переменная будет находиться в другой области видимости:

```
for (int i = 0; i < 10; i++)
{
    ...
}
for (int i = 0; i < 20; i += 2) // все в порядке
{
    ...
}
```

Запись инструкций `do`

Инструкции `while` и `for` проводят проверку своих булевых выражений в самом начале цикла. Следовательно, если выражение при первой же проверке вычисляется в `false`, тело цикла ни разу не выполняется. Инструкция `do` действует по-другому: ее булево выражение вычисляется после каждой итерации, поэтому тело всегда выполняется как минимум один раз.

Синтаксис инструкции `do` выглядит следующим образом (не забудьте ставить завершающую точку с запятой):

```
do
    инструкция
while (булево выражение);
```

Если в теле цикла содержится более одной инструкции, нужно использовать кодовый блок (если этого не сделать, компилятор выдаст синтаксическую ошибку). Вот как выглядит версия примера, записывающего на консоль значения от 0 до 9, которая теперь построена с применением инструкции `do`:

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
}
while (i < 10);
```

ИНСТРУКЦИИ `BREAK` И `CONTINUE`

В главе 4 было показано применение инструкции `break` для выхода из инструкции `switch`. Инструкцию `break` можно использовать также для выхода из тела инструкции итерации. При выходе из цикла его работа тут же прекращается, выполнение программы продолжается с первой инструкции, следующей за циклом. Но обновление и вычисление условия продолжения цикла больше не производятся. В отличие

от этого, инструкция `continue` заставляет программу перейти к немедленному выполнению следующей итерации цикла (после перевычисления булева выражения). Посмотрите на еще одну версию примера, записывающего на консоль значения от 0 до 9, на этот раз с применением инструкций `break` и `continue`:

```
int i = 0;
while (true)
{
    Console.WriteLine(i);
    i++;
    if (i < 10)
        continue;
    else
        break;
}
```

Это абсолютно несуразный код. Многие руководства по программированию рекомендуют использовать инструкцию `continue` крайне осмотрительно или вообще отказаться от нее, поскольку зачастую это приводит к созданию кода, в котором очень трудно разобраться. Также порой инструкция `continue` обуславливает весьма странный ход выполнения программы. Например, если инструкция `continue` выполняется внутри инструкции `for`, то перед выполнением следующей итерации запускается та ее часть, которая обновляет значение управляющей переменной.

В следующем упражнении вы напишете инструкцию `do`, преобразующую положительное десятичное целое число в его строковое представление в восьмеричной форме записи. Программа основана на следующем алгоритме, в котором применяется широко известная математическая процедура:

сохраните десятичное число в переменной `dec`
сделайте следующее
разделите `dec` на 8 и сохраните остаток
установите для `dec` значение частного из предыдущего шага,
пока `dec` не равен нулю
объединяйте значения, сохраненные в остатке для каждого вычисления, в обратном порядке

Предположим, к примеру, что вам нужно преобразовать десятичное число 999 в восьмеричное. Для этого нужно выполнить следующие шаги.

1. Разделить 999 на 8. Частное будет равно 124, а остаток — 7.
2. Разделить 124 на 8. Частное будет равно 15, а остаток — 4.
3. Разделить 15 на 8. Частное будет равно 1, а остаток — 7.
4. Разделить 1 на 8. Частное будет равно 0, а остаток — 1.
5. Объединить значения, вычисленные для остатка на каждом шаге, в обратном порядке. Получится результат 1747. Это и будет восьмеричным представлением десятичного значения 999.

Запись инструкции do

Откройте в Visual Studio 2015 проект DoStatement, который находится в папке `\Microsoft Press\VCSBS\Chapter 5\DoStatement` вашей папки документов.

Выведите в окно конструктора форму `MainPage.xaml`. Эта форма содержит текстовое поле по имени `number`, в которое пользователь может ввести десятичное число.

Когда пользователь щелкает на кнопке `Show Steps` (Показать шаги), создается восьмеричное представление введенного числа. Результаты каждой стадии вычисления показываются в расположенному справа текстовом поле по имени `steps`.

Выведите в окно редактора кода и текста код файла `MainPage.xaml.cs` и найдите метод `showStepsClick`. Этот метод запускается, когда пользователь щелкает в форме на кнопке `Show Steps`. Пока что он пуст.

Добавьте в метод `showStepsClick` следующие инструкции, выделенные жирным шрифтом:

```
private void showStepsClick(object sender, RoutedEventArgs e)
{
    int amount = int.Parse(number.Text);
    steps.Text = "";
    string current = "";
}
```

Первая инструкция превращает строковое значение в свойстве `Text` текстового поля `number` в целое число путем использования метода `Parse`, принадлежащего типу `int`, и сохраняет результат в локальной переменной по имени `amount`.

Вторая инструкция удаляет текст, отображаемый в текстовом поле `steps`, устанавливая в качестве значения его свойства `Text` пустую строку.

Третья инструкция объявляет строковую переменную по имени `current` и инициализирует ее значением пустой строки. Эта переменная будет применяться для хранения цифр, создаваемых при каждой итерации цикла, используемого для преобразования десятичного числа в его восьмеричное представление.

Добавьте к методу `showStepsClick` следующую инструкцию `do` (выделенную жирным шрифтом):

```
private void showStepsClick(object sender, RoutedEventArgs e)
{
    int amount = int.Parse(number.Text);
    steps.Text = "";
    string current = "";

    do
    {
```

```

int nextDigit = amount % 8;
amount /= 8;
int digitCode = '0' + nextDigit;
char digit = Convert.ToChar(digitCode);
current = digit + current;
steps.Text += current + "\n";
}
while (amount != 0);
}

```

Используемый здесь алгоритм многократно выполняет целочисленную арифметическую операцию, деля значения переменной `amount` на 8 и определяя остаток. Этот остаток после каждого успешного выполненного деления представляет собой следующую цифру в создаваемой строке. Со временем, когда значение переменной `amount` сократится до нуля, цикл завершится. Заметьте, что код тела должен быть выполнен хотя бы один раз. Именно такое поведение нам и требуется, поскольку даже у числа 0 имеется одна восьмеричная цифра.

Если присмотреться к коду, можно заметить, что первой в цикле `do` выполняется следующая инструкция:

```
int nextDigit = amount % 8;
```

Она объявляет `int`-переменную по имени `nextDigit` и инициализирует ее остатком от деления значения переменной `amount` на 8. Этот остаток будет числом из диапазона от 0 до 7.

Следующей инструкцией в цикле `do` является

```
amount /= 8;
```

Это составное присваивание, эквивалентное записи `amount = amount / 8;`. Если значение `amount` равно 999, то после выполнения этой инструкции оно станет равно 124.

Следующей инструкцией является

```
int digitCode = '0' + nextDigit;
```

Тут необходимо кое-что пояснить. У символов имеется уникальный код, соответствующий набору символов, используемому операционной системой. В наборе символов операционной системы Windows у кода символа «0» имеется целочисленное значение 48. Код символа «1» имеет значение 49, символа «2» — 50 и так далее вплоть до кода символа «9», имеющего целочисленное значение 57. В C# можно рассматривать символ как целое число и выполнять над ним арифметические действия, но при этом C# использует в качестве значения код символа. Следовательно, выражение `'0' + nextDigit` фактически вычисляется в значение в диапазоне между 48 и 55 (вспомним, что значение `nextDigit` всегда будет между 0 и 7), что соответствует коду для эквивалентной восьмеричной цифры.

Четвертой инструкцией в цикле do является

```
char digit = Convert.ToChar(digitCode);
```

Эта инструкция объявляет символьную переменную по имени `digit` и инициализирует ее результатом вызова метода `Convert.ToChar(digitCode)`. Метод `Convert.ToChar` получает целое число, содержащее код символа, и возвращает соответствующий символ. К примеру, если у `digitCode` значение 54, `Convert.ToChar(digitCode)` возвращает символ '6'.

В целом, первые четыре инструкции цикла do определяют символ, представляющий самую младшую (крайнюю справа) восьмеричную цифру, соответствующую числу, введенному пользователем. Следующей задачей будет добавление этой цифры к началу выводимой строки:

```
current = digit + current;
```

Следующей инструкцией в цикле do является

```
steps.Text += current + "\n";
```

Эта инструкция добавляет к текстовому полю `steps` строку, содержащую цифры, созданные до этого момента для восьмеричного представления числа. Кроме этого, инструкция добавляет символ новой строки, чтобы каждая стадия преобразования появлялась в текстовом поле на отдельной строке.

И наконец, в компоненте `while` в конце цикла do вычисляется условие:

```
while (amount != 0);
```

Поскольку значение переменной `amount` пока не равно нулю, цикл выполняет следующую итерацию.

В заключительном упражнении данной главы будет задействован отладчик Visual Studio 2015, позволяющий запустить пошаговое выполнение предыдущей инструкции do, чтобы помочь вам разобраться в том, как она работает.

Пошаговое выполнение инструкции do

В окне редактора, показывающего содержимое файла `MainPage.xaml.cs`, переместите курсор на первую инструкцию метода `showStepsClick`:

```
int amount = int.Parse(number.Text);
```

Щелкните правой кнопкой мыши на любом месте первой инструкции и выберите пункт Выполнить до текущей позиции (Run To Cursor). Когда появится

форма, наберите в расположеннем слева текстовом поле число 999 и щелкните на кнопке Show Steps (Показать шаги).

Программа остановится, и вы попадете в режим отладки Visual Studio 2015. Желтая стрелка в левом поле окна редактора и желтая фоновая подсветка выделят код текущей инструкции. Выведите панель инструментов отладки, если ее еще нет на экране (выберите в меню Вид пункт Панели инструментов и установите флашок Отладка).



ПРИМЕЧАНИЕ Команды панели отладки доступны также в меню Отладка, отображаемом на панели меню.

Щелкните в панели инструментов отладки на кнопке со стрелкой вниз, указывающей на пункт Добавить или удалить кнопки (Add Or Remove Buttons), а затем выберите пункт Окна (рис. 5.3).

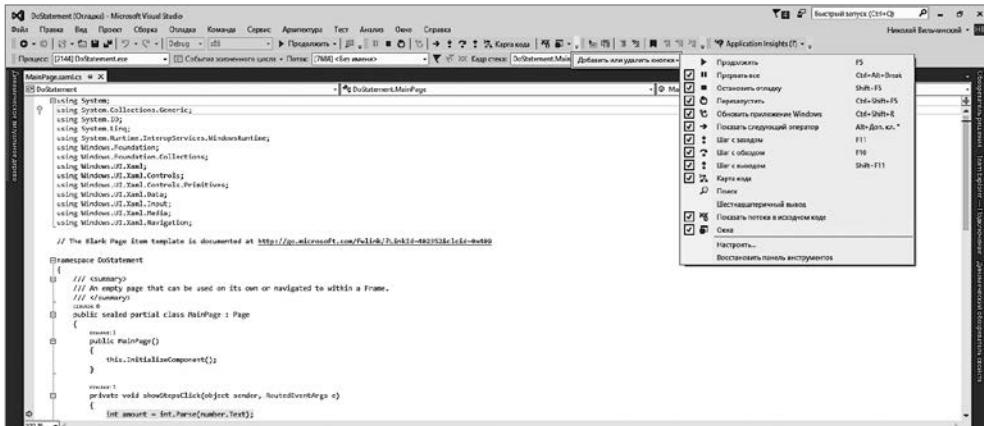


Рис. 5.3

В результате этого к инструментам добавится кнопка Точки останова (Breakpoints Window).

Щелкните в панели отладчика на стрелке, которая находится сразу же за кнопкой Точки останова, а затем щелкните на пункте Локальные (Locals) (рис. 5.4).

Появится окно Локальные (если оно еще не было открыто). В нем отобразятся имена, значения и типы локальных переменных, имеющихся в текущем методе, включая локальную переменную `amount`. Обратите внимание на то, что текущим значением `amount` указан нуль (рис. 5.5).

Щелкните на панели отладчика на кнопке Шаг с заходом. Отладчик выполнит следующую инструкцию:

```
int amount = int.Parse(number.Text);
```

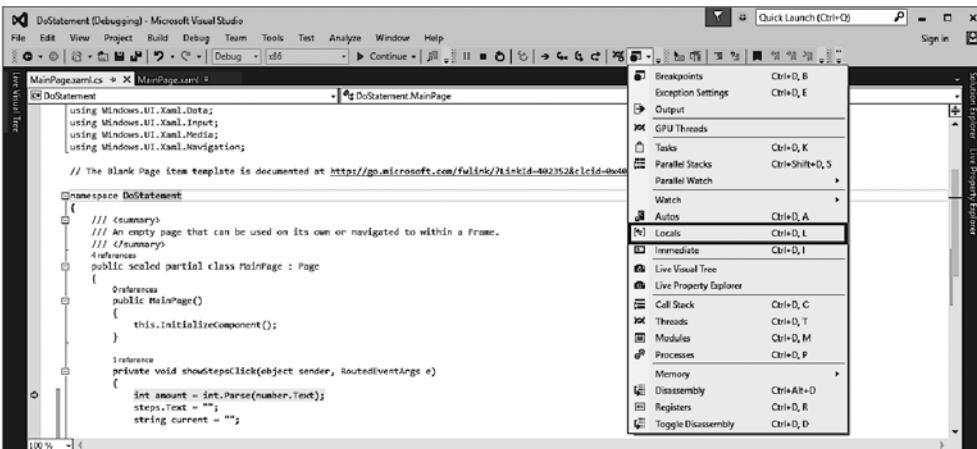


Рис. 5.4

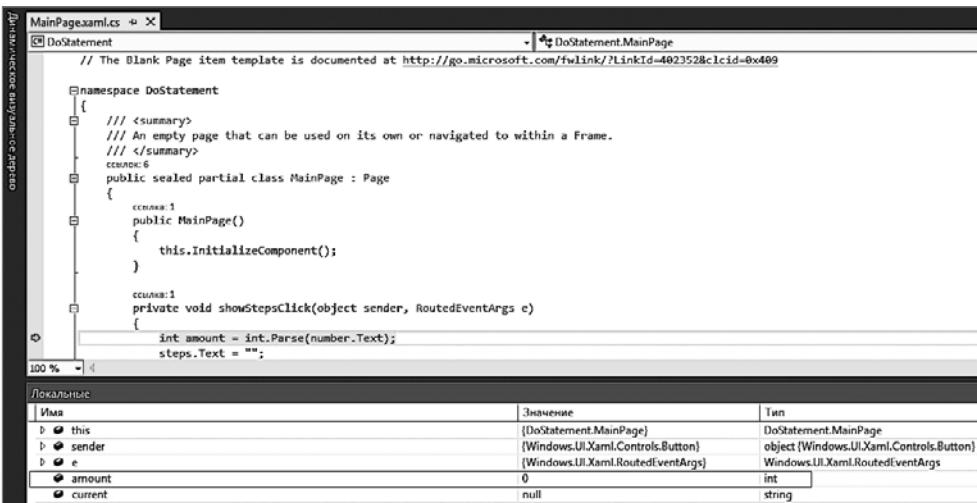


Рис. 5.5

Значение переменной `amount` в окне **Локальные** изменится на 999, и желтая стрелка переместится на следующую инструкцию. Еще раз щелкните на кнопке **Шаг с заходом**. Отладчик выполнит следующую инструкцию:

```
steps.Text = "";
```

Это не повлияет на состояние содержимого окна **Локальные**, поскольку `steps` является элементом управления формы, а не локальной переменной. Желтая

стрелка переместится на следующую инструкцию. Щелкните на кнопке Шаг с заходом. Отладчик выполнит следующую инструкцию:

```
string current = "";
```

Желтая стрелка переместится на фигурную скобку, открывающую цикл do. В этом цикле содержатся три собственные локальные переменные: `nextDigit`, `digitCode` и `digit`. Обратите внимание на их появление в окне Локальные и на то, что все три переменные изначально имеют нулевые значения.

Щелкните на кнопке Шаг с заходом. Желтая стрелка переместится на первую инструкцию внутри цикла do. Щелкните на кнопке Шаг с заходом. Отладчик выполнит следующую инструкцию:

```
int nextDigit = amount % 8;
```

Значение переменной `nextDigit` в окне Локальные изменится на 7. Это остаток от деления 999 на 8. Щелкните на кнопке Шаг с заходом. Отладчик выполнит такую инструкцию:

```
amount /= 8;
```

Значение переменной `amount` в окне Локальные изменится на 124. Щелкните на кнопке Шаг с заходом. Отладчик выполнит очередную инструкцию:

```
int digitCode = '0' + nextDigit;
```

Значение переменной `digitCode` в окне Локальные изменится на 55. Это код символа «7» (48 + 7). Щелкните на кнопке Шаг с заходом. Отладчик выполнит следующую инструкцию:

```
char digit = Convert.ToChar(digitCode);
```

Значение переменной `digit` в окне Локальные изменится на '7'. Символьные значения в окне Локальные отображаются как в исходном числовом значении (в данном случае 55), так и в символьном представлении ('7').

Обратите внимание на то, что значение переменной `current` в окне Локальные по-прежнему является пустой строкой "". Щелкните на кнопке Шаг с заходом. Отладчик выполнит следующую инструкцию:

```
current = current + digit;
```

Значение переменной `current` в окне Локальные изменится на "7". Щелкните на кнопке Шаг с заходом. Отладчик выполнит очередную инструкцию:

```
steps.Text += current + "\n";
```

Эта инструкция выведет в текстовое поле `steps` текст "7", за которым следует символ новой строки, чтобы следующий вывод в текстовое поле отображался на новой строке. (Форма в данный момент скрыта за окном Visual Studio, поэтому вы ее увидеть не сможете.) Курсор переместится на закрывающую фигурную скобку в конце цикла `do`.

Щелкните на кнопке Шаг с заходом. Желтая стрелка переместится на инструкцию `while` для вычисления, позволяющего определить, завершился цикл `do` или же его следует продолжить для следующей итерации.

Щелкните на кнопке Шаг с заходом. Отладчик выполнит следующую инструкцию:

```
while (amount != 0);
```

Значение переменной `amount` равно 124, и выражение `124 != 0` вычисляется в `true`, следовательно, цикл `do` выполняет еще одну итерацию. Желтая стрелка переходит обратно к открывающей фигурной скобке в начале цикла `do`. Щелкните на кнопке Шаг с заходом. Желтая стрелка опять переместится на первую инструкцию внутри цикла `do`.

Продолжайте щелкать на кнопке Шаг с заходом, чтобы были пройдены следующие три итерации цикла `do`, и наблюдайте за тем, как в окне Локальные изменяются значения переменных.

В конце четвертой итерации цикла значение переменной `amount` станет равно нулю, а значение переменной `current` станет равно "1747". Желтая стрелка будет находиться на условии `while` в конце цикла `do`:

```
while (amount != 0);
```

Поскольку теперь значение переменной `amount` равно нулю, выражение `amount != 0` будет вычислено в `false` и цикл `do` должен завершиться.

Щелкните на кнопке Шаг с заходом. Отладчик выполнит следующую инструкцию:

```
while (amount != 0);
```

Как и предсказывалось, цикл `do` завершается и желтая стрелка перемещается на закрывающую фигурную скобку в конце метода `showStepsClick`.

Щелкните в меню Отладка на пункте Продолжить. Появится форма, отображающая четыре этапа создания восьмеричного представления числа 999: 7, 47, 747 и 1747 (рис. 5.6).

Вернитесь в Visual Studio 2015. Щелкните в меню Отладка на пункте Остановить отладку.

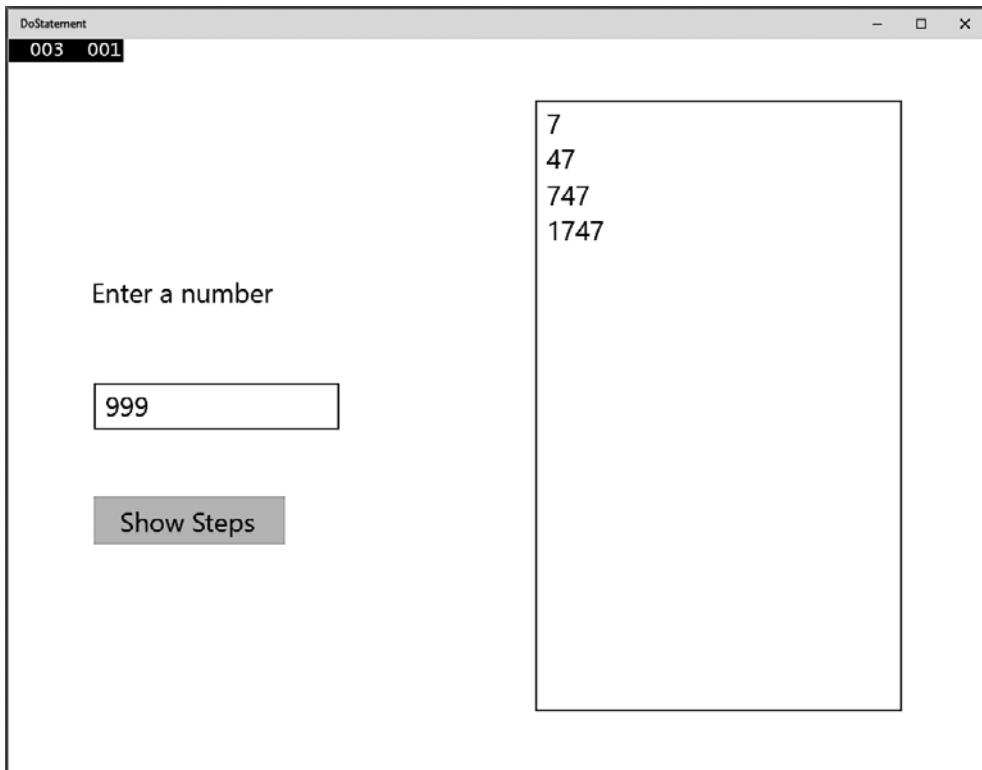


Рис. 5.6

Выводы

В данной главе вы узнали, как использовать операторы составного присваивания для обновления значения числовых переменных и для добавления одной строки к другой. Увидели, как использовать инструкции `while`, `for` и `do` для многократного выполнения кода до тех пор, пока некоторое булево выражение вычисляется в `true`.

Если вы хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 6 «Обработка ошибок и исключений».

Если вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Добавить значение к переменной	Воспользуйтесь оператором составного присваивания со сложением, например: <code>переменная += значение;</code>
Вычесть значение из переменной	Воспользуйтесь оператором составного присваивания с вычитанием, например: <code>переменная -= значение;</code>
Выполнить одну или несколько инструкций нуль и более раз, пока условие вычисляется в true	Воспользуйтесь инструкцией while, например: <code>int i = 0; while (i < 10) { Console.WriteLine(i); i++; }</code> Или же воспользуйтесь инструкцией for, например: <code>for (int i = 0; i < 10; i++) { Console.WriteLine(i); }</code>
Выполнить инструкции один или более раз	Воспользуйтесь инструкцией do, например: <code>int i = 0; do { Console.WriteLine(i); i++; } while (i < 10);</code>

6 Обработка ошибок и исключений

Прочитав эту главу, вы научитесь:

- обрабатывать исключения с использованием инструкций `try`, `catch` и `finally`;
- управлять целочисленным переполнением с использованием ключевых слов `checked` и `unchecked`;
- выдавать исключения из своих собственных методов с использованием ключевого слова `throw`;
- гарантировать безусловное выполнение кода даже после выдачи исключения с использованием блока `finally`.

Основные инструкции C#, знание которых позволит выполнить наиболее распространенные задачи, в том числе написание методов, объявление переменных, использование операторов для создания значений, написание инструкций `if` и `switch` для выборочного выполнения кода и написание инструкций `while`, `for` и `do` для многократного выполнения кода, вы уже видели. Но в предыдущих главах не рассматривалась возможность (или вероятность) того, что что-то может пойти не по сценарию.

Гарантировать, что какая-либо часть кода всегда будет работать абсолютно ожидаемым образом, очень трудно. Сбои может вызвать широкий спектр причин, со многими из которых вы, как программист, в состоянии справиться. Любые создаваемые вами приложения должны быть в состоянии обнаруживать сбои и обрабатывать их подобающим образом: либо выполняя корректирующие действия, либо, если исправить ситуацию невозможно, доводя до пользователя в максимально понятном виде причины сбоя. В заключительной главе первой части книги вы узнаете, как в C# в целях оповещения о возникновении ошибки используются исключения и как для обработки ошибок, представленных этими исключениями, применяются инструкции `try`, `catch` и `finally`.

К концу этой главы у вас будет сформирован прочный фундамент из знаний обо всех основных элементах C#, на котором вы сможете построить все, что будет изложено во второй части книги.

Борьба с ошибками

Неприятности порой случаются, такова суровая реальность. Шины прокалываются, батареи садятся, шуруповерты никогда не находятся там, где их оставили, а пользователи ваших приложений ведут себя непредсказуемым образом. В компьютерном мире отказывают жесткие диски, на том же компьютере запускаются другие приложения, делая неуправляемой вашу программу и забирая всю доступную память, в самый неподходящий момент пропадает беспроводное соединение, влияние могут оказывать даже такие природные явления, как удар молнии поблизости, вызвавший отключение электричества или сетевой сбой. Ошибки могут случаться практически на любой стадии выполнения программы, и причина многих из них может быть не связана со сбоями вашего собственного приложения. Поэтому возникает вопрос: как их обнаружить и попытаться исправить?

На протяжении многих лет совершенствовался целый ряд механизмов. В обычных подходах, принятых такими весьма почтенными по возрасту системами, как UNIX, предполагалось при сбое метода проводить подстройку операционной системы под установку специальной глобальной переменной. Затем после каждого вызова метода значение этой глобальной переменной проверялось, с тем чтобы убедиться в успешном завершении его работы. В C#, как и в большинстве современных объектно-ориентированных языков программирования, ошибки этим способом не обрабатываются, поскольку это обходится слишком дорого. Вместо этого используются исключения. Если нужно добиться от программ, создаваемых на C#, надежной работы, следует узнать о том, что такое исключения.

Попытка выполнения кода и перехват исключений

Ошибки могут случаться в любое время, и использование традиционных технологий, предусматривающих добавление вручную кода обнаружения ошибок в отношении каждой инструкции, — занятие слишком обременительное, затратное по времени и не гарантирующее отсутствия собственных ошибок в этом коде. К счастью, путем использования исключений и их обработчиков C# облегчает отделение кода обработки ошибок от кода, реализующего основную

логику программы. Чтобы создавать программы, поддерживающие исключения, нужно сделать две вещи:

- ❑ поместить код внутри блока `try` (это слово в C# является ключевым). Когда код запускается, предпринимается попытка выполнения всех инструкций, имеющихся в блоке `try`, и если ни одна из инструкций не выдает исключения, они выполняются одна за другой до завершения блока. Но если создаются условия для возникновения ошибки, выдается исключение, которое приводит к передаче управления от блока `try` другому фрагменту кода, разработанному для перехвата и обработки исключений, — обработчику исключения;
- ❑ создать один или несколько обработчиков исключений с применением еще одного ключевого слова C#, `catch`, помещаемого сразу же после блока `try`, для обработки любых возможных условий возникновения ошибки. Обработчик исключений предназначен для перехвата и обработки исключения указанного типа, и после блока `try` у вас может быть указано сразу несколько обработчиков исключений, каждый из которых разработан для перехвата и обработки конкретного исключения. Это позволяет вам предоставлять различные обработчики для различных ошибок, которые могут возникать в блоке `try`. Если любая из инструкций внутри блока `try` вызывает ошибку, среда выполнения выдает исключение. Затем эта же среда исследует обработчики исключений, которые находятся после блока `try`, и передает управление непосредственно первому подходящему обработчику.

Рассмотрим пример блока `try`, содержащего код, предпринимающий попытку преобразования строк, введенных пользователем в текстовые поля формы, в целочисленные значения. Затем код вызывает метод для вычисления значения и записи результата в другое текстовое поле. Преобразование строки в целое число требует, чтобы в строке содержался подходящий набор цифр, а не какая-нибудь произвольная последовательность символов. Если в строке содержатся неподходящие символы, метод `int.Parse` выдает исключение `FormatException` и управление передается соответствующему обработчику исключения. Когда обработчик исключения завершит свою работу, выполнение программы продолжится передачей управления первой же следующей за обработчиком инструкции. Учтите, что в отсутствие обработчика, соответствующего исключению, это исключение считается необработанным (данная ситуация вскоре будет рассмотрена):

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (FormatException fEx)
```

```
{  
    // Обработка исключения  
    ...  
}
```

Для указания перехватываемого исключения в обработчике исключения используется синтаксис, похожий на тот, который использовался параметром метода. В предыдущем примере при выдаче исключения `FormatException` переменная `fEx` заполняется объектом, содержащим подробности исключения.

У типа `FormatException` имеется ряд свойств, которые можно исследовать для определения конкретной причины выдачи исключения. Многие из этих свойств являются общими для всех исключений. Например, в свойстве `Message` содержится текстовое описание ошибки, вызвавшей исключение. Этой информацией можно воспользоваться при обработке исключения, возможно, для записи подробностей в регистрационный журнал или для вывода пользователю информативного сообщения с последующей просьбой сделать еще одну попытку.

Необработанные исключения

А что произойдет, если блок `try` выдаст исключение, но соответствующего обработчика не окажется? В предыдущем примере существует возможность того, что в текстовом поле `lhsOperand` будет содержаться строковое представление допустимого целого числа, но такого, которое выходит за пределы диапазона поддерживаемых в C# целых чисел (например, 2 147 483 648). В таком случае инструкция `int.Parse` выдаст исключение переполнения `OverflowException`, которое не будет перехвачено обработчиком `FormatException`. Если такое случится и блок `try` окажется частью метода, тут же произойдет выход из метода и управление будет возвращено тому методу, который его вызывал. Если вызвавший метод использует блок `try`, среда выполнения предпримет попытку найти соответствующий обработчик исключения для этого блока `try` и выполнит код обработчика. Если вызвавший метод не использует блок `try` или в нем не имеется соответствующего обработчика исключения, тут же произойдет выход из метода и управление будет возвращено тому методу, который его вызывал, где процесс повторится. Если соответствующий обработчик исключения в конце концов найдется, он отработает и выполнение продолжится с первой инструкции, следующей за обработчиком исключения того метода, в котором оно было перехвачено.



ВНИМАНИЕ Учтите, что после перехвата исключения выполнение продолжается в том методе, который содержит блок `catch`, перехвативший исключение. Если исключение произошло в методе, отличном от того, который содержит обработчик исключения, управление не будет возвращено тому методу, выполнение которого стало причиной выдачи исключения.

Если после каскадного возвращения назад через список вызывавших методов среда выполнения не сможет найти соответствующий обработчик исключений, выполнение программы прервется с необработанным исключением.

Изучить исключения, выданные вашим приложением, нетрудно. Если при запуске приложения в Microsoft Visual Studio 2015 в режиме отладки (для чего нужно в меню Отладка выбрать пункт Начать отладку) будет выдано исключение, появится диалоговое окно, похожее на показанное на рис. 6.1, и приложение встанет на паузу, помогая вам определить причину выдачи исключения.

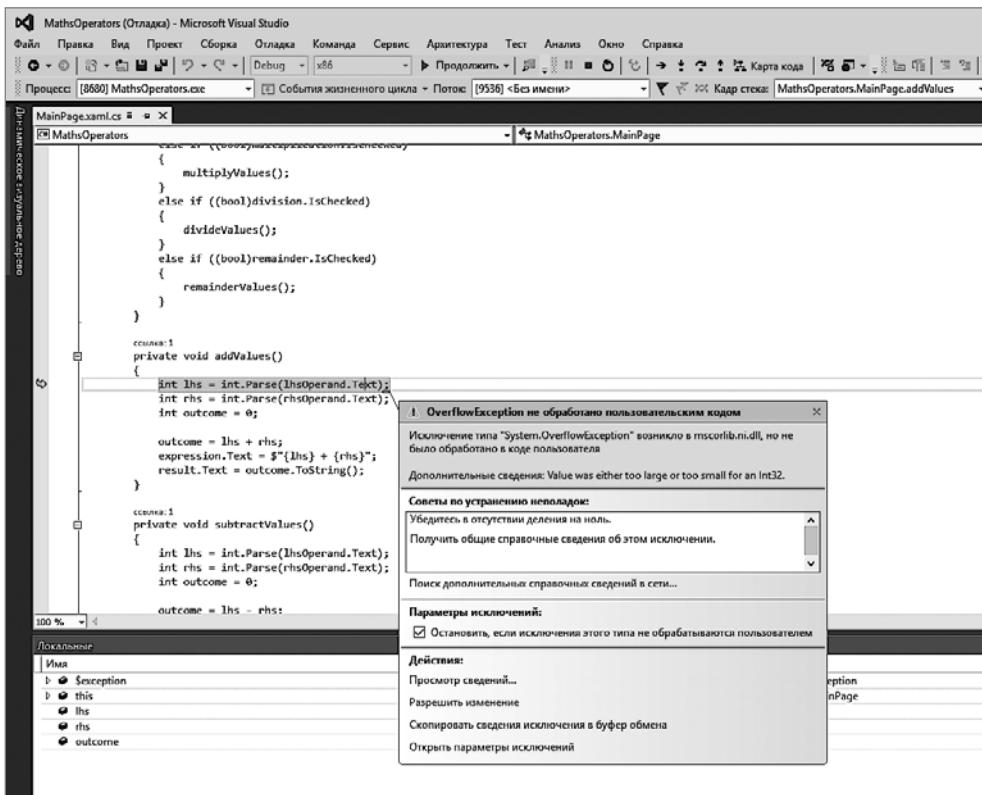


Рис. 6.1

Выполнение приложения остановится на инструкции, вызвавшей выдачу исключения, и вы попадете в отладчик. При этом появится возможность изучить значения переменных, изменить эти значения и приступить к пошаговому выполнению кода с того места, в котором произошла выдача исключения, для чего следует воспользоваться панелью инструментов отладки и различными окнами отладчика.

Использование нескольких обработчиков исключений

Ранее мы уже выяснили, как из-за разных ошибок выдаются различные типы исключений для представления различных видов отказов. Чтобы справиться с такими ситуациями, можно предоставить несколько обработчиков исключений, расположив их друг за другом:

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (FormatException fEx)
{
    //...
}
catch (OverflowException oEx)
{
    //...
}
```

Если код в блоке `try` выдает исключение `FormatException`, будут запущены инструкции в блоке `catch`, предназначенном для исключения `FormatException`. Если код выдаст исключение `OverflowException`, будет запущен блок `catch`, предназначенный для исключения `OverflowException`.



ПРИМЕЧАНИЕ Если код в блоке `catch` `FormatException` выдает исключение `OverflowException`, это не становится причиной запуска примыкающего блока `catch` для `OverflowException`. Вместо этого, как было рассмотрено ранее в данном разделе, исключение распространяется на метод, вызвавший этот код.

Обработка нескольких исключений

Механизм перехвата исключений, предоставляемый C# и средой Microsoft .NET Framework, весьма совершенен. Среда .NET Framework определяет многие типы исключений, и большинство из них может быть выдано любой написанной вами программой. Вряд ли вам захочется создавать обработчики исключений для каждого возможного исключения, которое в состоянии выдать ваш код, — следует помнить, что приложение должно быть способно обрабатывать исключения, о которых вы при написании кода можете даже не подозревать! Так как же гарантировать перехват и обработку вашей программой всех возможных исключений?

Ответ на этот вопрос кроется в способах взаимосвязанности различных исключений. Все исключения сводятся в семейства, называемые иерархиями наследования. (О наследовании вы узнаете в главе 12 «Работа с наследованием».) Исключения `FormatException` и `OverflowException`, как и ряд других, принадлежат

семейству по имени `SystemException`. А само семейство `SystemException` входит в более обширное семейство, которое называется просто `Exception` и является прародителем всех исключений. Если перехватывается `Exception`, то в обработчик попадают все выдаваемые исключения.



ПРИМЕЧАНИЕ Семейство `Exception` включает широкое разнообразие исключений, многие из которых предназначены для использования различными частями среды .NET Framework. Некоторые из этих исключений случаются крайне редко, но разобраться в том, как они перехватываются, все равно не помешает.

Следующий пример показывает, как можно перехватить все возможные исключения:

```
try
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int answer = doCalculation(leftHandSide, rightHandSide);
    result.Text = answer.ToString();
}
catch (Exception ex) // это общий обработчик исключений
{
    //...
}
```



ПРИМЕЧАНИЕ Если требуется перехватить `Exception`, его имя в перехватчике исключения можно не указывать, поскольку этот тип исключения используется по умолчанию:

```
catch
{
    // ...
}
```

И тем не менее делать это не рекомендуется. Объект исключения, который передается в обработчик исключения, может содержать относящуюся к исключению полезную информацию, получить доступ к которой при использовании данной версии конструкции блока `catch` будет совсем не просто.

Остается еще один, последний вопрос, который должен быть задан к этому моменту: что произойдет, если одно и то же исключение соответствует нескольким обработчикам исключений, расположенным за блоком `try`? Если `FormatException` и `Exception` перехватываются в двух различных обработчиках, то какой из них будет запущен (или же будут выполнены оба обработчика)?

Когда выдается исключение, среда выполнения использует первый же найденный ею обработчик, соответствующий исключению, игнорируя при этом все остальные. Это означает, что при помещении обработчика для `Exception` перед обработчиком для `FormatException` обработчик `FormatException` не будет запущен никогда. Поэтому после блока `try` более конкретизированные обработчики

следует помещать выше обработчиков более общего плана. Если ни один из конкретных обработчиков не будет соответствовать исключению, ему будет соответствовать общий обработчик исключений.

ФИЛЬТРЫ ИСКЛЮЧЕНИЙ

Фильтры исключений являются новым свойством C#, оказывающим влияние на способ соответствия исключений их обработчикам. Фильтр исключения позволяет указать дополнительные условия, при которых используется обработчик исключения. Эти условия принимают форму булева выражения, перед которым ставится ключевое слово `when`. Синтаксис показан в следующем примере:

```
catch (Exception ex) when (ex.GetType() != typeof(System.OutOfMemoryException))
{
    // Обработка всех ранее не перехваченных исключений, кроме того,
    // которое называется OutOfMemoryException
}
```

В этом примере перехватываются все исключения (типа `Exception`), но фильтр указывает, что если по типу исключение будет относиться к нехватке памяти (`out-of-memory`), этот обработчик будет проигнорирован. (Метод `GetType` возвращает тип переменной, указанной в качестве аргумента.) Тем самым предоставляется конкретный способ обработки всех исключений, кроме исключения, выдаваемого при нехватке памяти. Если возникают условия для выдачи этого исключения, среда выполнения продолжит поиск, чтобы использовать другой обработчик исключения, и если не сможет его найти, исключение будет считаться необработанным.

В следующих упражнениях вы увидите, что происходит, когда приложение выдает необработанное исключение, а затем вы напишете блоки `try` и `catch` и обработаете исключение.

Наблюдение за тем, как приложение оповещает о необработанных исключениях

Откройте в среде Visual Studio 2015 решение `MathsOperators`, которое находится в папке `\Microsoft Press\VCSBS\Chapter 6\MathsOperators` вашей папки документов.

Это версия программы из главы 2 «Работа с переменными, операторами и выражениями», демонстрирующей работу различных арифметических операторов.

Щелкните в меню Отладка на пункте Запуск без отладки.



ПРИМЕЧАНИЕ Для этого упражнения очень важно запустить приложение без отладки.

Появится форма. Теперь вы намереваетесь ввести в поле `Left Operand` текст, который вызовет выдачу исключения. Эта операция продемонстрирует ненадежность текущей версии программы.

Наберите в поле Left Operand строку John, а в поле Right Operand — цифру 2. Щелкните на пункте + Addition, а затем на кнопке Calculate.

Этот ввод запустит в Windows обработку исключения по умолчанию: приложение просто прекратит работу, а вы вернетесь на Рабочий стол!

Следующим этапом после того, как вы увидели, как ведет себя приложение при выдаче необработанного исключения, станет создание более надежного приложения, способного обработать неверный ввод и предотвратить выдачу необработанного исключения.

Создание блоков инструкций try и catch

Вернитесь в Visual Studio 2015. Щелкните в меню Отладка на пункте Начать отладку. Когда появится форма, наберите в поле Left Operand строку John, а в поле Right Operand — цифру 2. Щелкните на пункте + Addition, а затем на кнопке Calculate.

Этот ввод должен стать причиной выдачи того же исключения, что и в предыдущем упражнении, не считая того, что теперь вы работаете в режиме отладки и среда Visual Studio перехватит исключение и оповестит вас о его выдаче.

Среда Visual Studio выведет ваш код и выделит инструкцию, вызвавшую выдачу исключения. Она также выведет на экран диалоговое окно с описанием исключения, которое в данном случае будет таким: «Input string was not in a correct format». («Введенная строка не соответствовала допустимому формату») (рис. 6.2).

Вы можете увидеть, что исключение было выдано внутри метода addValues из-за вызова метода int.Parse. Проблема в том, что данный метод не в состоянии преобразовать текст «John» в допустимое число.

Щелкните в диалоговом окне исключения на пункте Просмотр сведений (View Detail). Откроется еще одно диалоговое окно, в котором будет выведена дополнительная информация об исключении. Если раскрыть пункт System.FormatException, станет видна следующая информация (рис. 6.3).



СОВЕТ Некоторые исключения становятся результатом других, ранее выданных исключений. То исключение, о котором оповещает среда Visual Studio, — это всего лишь последнее исключение в цепочке, но обычно имеется более раннее исключение, высвечивающее реальную причину проблемы. Вы можете добраться до этих ранее выданных исключений, если раскроете свойство InnerException в диалоговом окне Просмотр сведений. У внутреннего исключения могут быть свои внутренние исключения, и вы можете продолжать углубляться, пока не увидите исключение, у которого свойство InnerException установлено в null (см. рис. 6.3). Это будет означать, что вы достигли исходного исключения, и, как правило, это и есть исключение, для устранения выдачи которого нужно внести изменения.

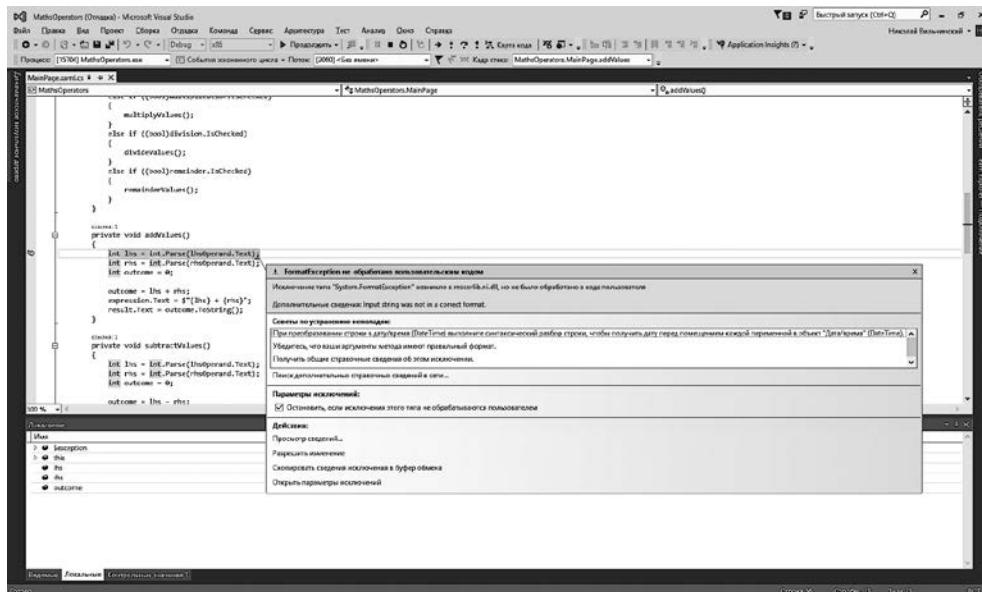


Рис. 6.2

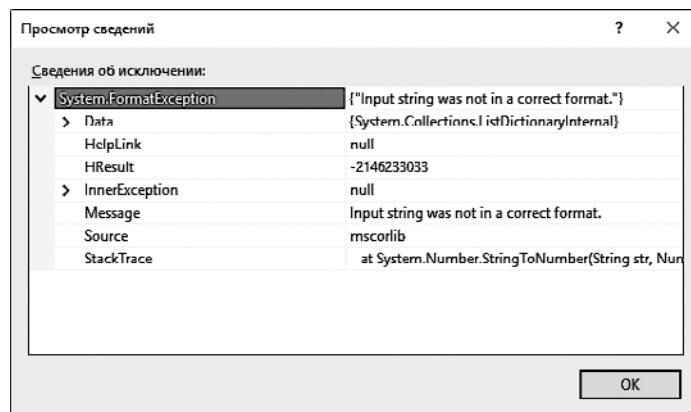


Рис. 6.3

Щелкните в диалоговом окне Просмотр сведений на кнопке OK, а затем в меню Отладка в среде Visual Studio щелкните на пункте Остановить отладку.

Выведите в окно редактора код файла MainPage.xaml.cs и найдите метод addValues.

Добавьте выделенный ниже жирным шрифтом блок `try` (включая фигурные скобки), охватив им инструкции внутри этого метода, и добавьте обработчик исключения `FormatException`:

```
try
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;

    outcome = lhs + rhs;
    expression.Text = $"{lhs} + {rhs}";
    result.Text = outcome.ToString();
}
catch (FormatException fEx)
{
    result.Text = fEx.Message;
}
```

Если будет выдано исключение `FormatException`, обработчик исключения выведет текст, содержащийся в свойстве исключения `Message` в текстовом поле результата, расположенному в нижней части формы.

Щелкните в меню Отладка на кнопке Начать отладку.

При появлении формы наберите в поле `Left Operand` строку `John`, а в поле `Right Operand` — цифру `2`. Щелкните на пункте `+ Addition`, а затем на кнопке `Calculate`.

Обработчик исключения успешно перехватит исключение `FormatException`, и в текстовое поле `Result` будет записано сообщение «`Input string was not in a correct format`». Теперь приложение стало немного надежнее.

Замените строку `John` числом `10`. Наберите в поле `Right Operand` строку `Sharp`, а затем щелкните на кнопке `Calculate`.

Блок `try` охватывает инструкции, выполняющие преобразование текста, введенного в оба текстовых поля, поэтому ошибки пользовательского ввода в обоих текстовых полях обрабатываются при выдаче исключения одним и тем же обработчиком.

Замените в поле `Right Operand` строку `Sharp` числом `20`, щелкните на пункте `+ Addition`, а затем на кнопке `Calculate`. Теперь приложение работает, как и ожидалось, и показывает в поле `Result` значение `30`.

Замените в поле `Left Operand` число `10` строкой `John`, щелкните на пункте `- Subtraction`, а затем на кнопке `Calculate`.

Среда Visual Studio передаст управление отладчику и снова выведет оповещение об исключении `FormatException`. На этот раз ошибка произошла в методе `subtractValues`, который не включает в себя необходимую структуру выдачи и обработки исключения `try-catch`.

Щелкните в меню Отладка на кнопке Остановить отладку.

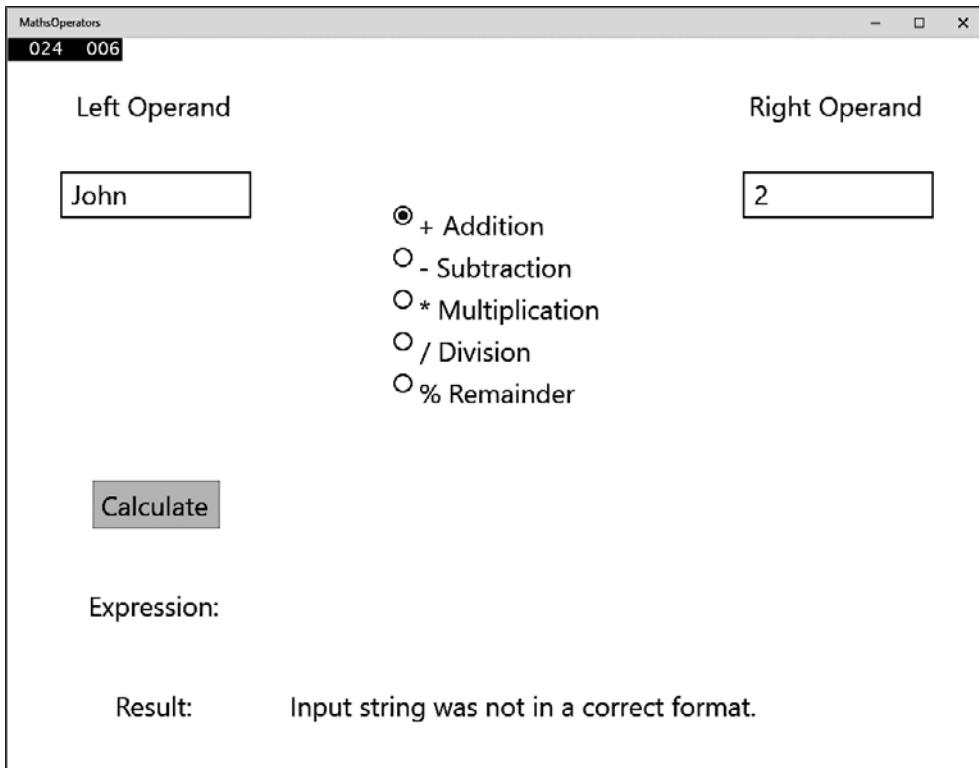


Рис. 6.4

Распространение исключений

Добавление блоков `try` и `catch` к методу `addValues` позволило повысить его надежность, но вам нужно применить такую же обработку исключений и к другим методам: `subtractValues`, `multiplyValues`, `divideValues` и `remainderValues`. Код для каждого из обработчиков исключений будет практически одинаковым, и получится, что вы будете вписывать в каждый метод один и тот же код. Каждый из этих методов вызывается методом `calculateClick` после того, как пользователь щелкает кнопкой мыши на кнопке Calculate. Стало быть, чтобы избежать дублирования кода обработки исключений, имеет смысл переместить его в метод `calculateClick`. Если в методе `subtractValues`, `multiplyValues`, `divideValues` или `remainderValues` будет выдано исключение `FormatException`, оно распространится в обратном направлении на метод `calculateClick` для обработки в том порядке, который был рассмотрен ранее в этой главе, в разделе «Необработанные исключения».

Распространение исключения с возвратом управления в вызывающий метод

Выполните в окне редактора код файла MainPage.xaml.cs и найдите метод addValues. Удалите из этого метода блок try и обработчик исключения, вернув его в исходное состояние:

```
private void addValues()
{
    int leftHandSide = int.Parse(lhsOperand.Text);
    int rightHandSide = int.Parse(rhsOperand.Text);
    int outcome = 0;

    outcome = lhs + rhs;
    expression.Text = lhsOperand.Text + " + " + rhsOperand.Text
    result.Text = outcome.ToString();
}
```

Найдите метод calculateClick. Добавьте к нему код блока try и обработчика исключения, выделенный в следующем примере жирным шрифтом:

```
private void calculateClick(object sender, RoutedEventArgs e)
{
    try
    {
        if ((bool)addition.IsChecked)
        {
            addValues();
        }
        else if ((bool)subtraction.IsChecked)
        {
            subtractValues();
        }
        else if ((bool)multiplication.IsChecked)
        {
            multiplyValues();
        }
        else if ((bool)division.IsChecked)
        {
            divideValues();
        }
        else if ((bool)remainder.IsChecked)
        {
            remainderValues();
        }
    }
    catch (FormatException fEx)
    {
        result.Text = fEx.Message;
    }
}
```

Щелкните в меню Отладка на кнопке Начать отладку.

После появления формы наберите в поле Left Operand строку John, а в поле Right Operand — число 2. Щелкните на пункте + Addition, а затем на кнопке Calculate.

Как и раньше, обработчик исключения успешно справится с перехватом исключения `FormatException` и в текстовое поле Result будет записано сообщение «Input string was not in a correct format». Но не забудьте, что исключение было фактически выдано в методе `addValues`, а перехвачено обработчиком в методе `calculateClick`.

Щелкните на пункте - Subtraction, а затем на кнопке Calculate. На этот раз исключение будет выдано методом `subtractValues`, но распространится в обратном направлении на метод `calculateClick` и обрабатывается так же, как и раньше.

Проверьте работу программы, выбирая пункты * Multiplication, / Division и % Remainder, и убедитесь в том, что исключение `FormatException` перехватывается и обрабатывается правильно.

Вернитесь в среду Visual Studio и остановите отладку.



ПРИМЕЧАНИЕ Решение о том, в каком именно методе следует перехватывать небоработанные исключения, зависит от характера создаваемого приложения. Иногда имеет смысл перехватывать исключения как можно ближе к тому месту, из которого они были выданы. А иногда сподручнее дать исключению возможность распространяться в обратном направлении на метод, вызвавший код, выдавший исключение, и обработать ошибку именно там.

Использование проверяемой и непроверяемой целочисленной арифметики

В главе 2 рассматривались вопросы использования бинарных арифметических операторов, таких как + и *, в отношении простых типов данных, таких как `int` и `double`. Там же говорилось, что эти простые типы данных имеют фиксированный размер. Например, в C# размер типа `int` составляет 32 бита. Поскольку `int` имеет фиксированный размер, вам точно известен диапазон значений, которые могут в нем содержаться: от -2 147 483 648 до 2 147 483 647.



СОВЕТ Если в коде нужно сослаться на минимальное или максимальное значение `int`, можно воспользоваться свойством `int.MinValue` или `int.MaxValue`.

Фиксированный размер `int`-типа является источником проблем. К примеру, что произойдет, если прибавить единицу к `int`-переменной, значение которой уже 2 147 483 647? Ответ зависит от способа компиляции приложения. По умолчанию компилятор C# создает код, позволяющий создать переполнение, в результате чего вы получите неверный ответ. (Фактически вычисление пройдет по

кругу, остановившись на самом большом отрицательном числе и выдав результат $-2\ 147\ 483\ 648$.) Причиной такого поведения является необходимость обеспечения высокой производительности: целочисленная арифметика встречается чуть ли не в каждой программе, и издержки на проверке переполнения каждого целочисленного выражения могут привести к очень низкому уровню производительности. Во многих случаях риск является вполне оправданным, поскольку вы знаете (или надеетесь!), что ваши `int`-значения не достигнут своих лимитов. Если такой подход вам не нравится, можно включить проверку переполнения.



СОВЕТ В среде Visual Studio 2015 включение и отключение проверки переполнения осуществляется установкой свойств проекта. Щелкните в обозревателе решений на имени YourProject, где в качестве YourProject должно фигурировать фактическое имя проекта. Щелкните в меню Проект на пункте Свойства: YourProject. В диалоговом окне свойств проекта щелкните на вкладке Сборка. Теперь в правом нижнем углу страницы щелкните на кнопке Дополнительно. В диалоговом окне Дополнительные параметры сборки установите или снимите флажок Проверять арифметические переполнения и потери точности.

Независимо от режима компиляции приложения для включения или выключения проверки арифметического переполнения в тех частях программы, которые, по вашему мнению, в этом нуждаются, вы можете использовать ключевые слова `checked` и `unchecked`. Эти ключевые слова отменяют тот вариант работы компилятора, который был указан для проекта.

Запись инструкций `checked`

Инструкция `checked` представляет собой блок, предваряемый ключевым словом `checked`. В том случае, когда целочисленное вычисление в блоке приводит к переполнению, вся целочисленная арифметика, находящаяся в инструкции `checked`, всегда выдает исключение `OverflowException`:

```
int number = int.MaxValue;
checked
{
    int willThrow = number++;           // эта инструкция вызовет переполнение
    Console.WriteLine("this won't be reached"); // а до этой программа не дойдет
}
```



ВНИМАНИЕ Проверка на переполнение будет подвергаться только та целочисленная арифметика, которая находится непосредственно в блоке `checked`. Например, если одной из инструкций в блоках `checked` будет вызов метода, проверка не будет применяться к коду, выполняемому в вызываемом методе.

Для создания блока инструкций, не подвергаемых проверке, можно также воспользоваться ключевым словом `unchecked`. Вся целочисленная арифметика

в блоке `unchecked` не проверяется и никогда не выдает исключение `OverflowException`, например:

```
int number = int.MaxValue;
unchecked
{
    int wontThrow = number++;
    Console.WriteLine("this will be reached");
}
```

Запись проверяемых выражений

Ключевые слова `checked` и `unchecked` могут применяться также для проверки переполнения в отношении целочисленных выражений, заключенных в круглые скобки, которым, как показано в следующем примере, предшествует ключевое слово `checked` или `unchecked`:

```
int wontThrow = unchecked(int.MaxValue + 1);
int willThrow = checked(int.MaxValue + 1);
```

Операторы составного присваивания (`+=` и `-=`), а также операторы инкремента (`++`) и декремента (`--`) также являются арифметическим операторами, ими можно управлять с помощью ключевых слов `checked` и `unchecked`. Вспомним, что `x += y` — это то же самое, что `x = x + y`.



ВНИМАНИЕ Использовать ключевые слова `checked` и `unchecked` для управления арифметикой с плавающей точкой (нечелочисленной арифметикой) нельзя. Они применимы только к целочисленной арифметике, использующей данные, имеющие типы `int` и `long`. Арифметика с плавающей точкой никогда не выдает исключение `OverflowException`, даже если разделить число на 0,0. (Вспомним, что в главе 2 уже говорилось о том, что в среде .NET Framework для операций с плавающей точкой имеется специальное представление для бесконечности.)

В следующем упражнении вы увидите, как проверяемая арифметика выполняется при использовании среды Visual Studio 2015.

Использование проверяемых выражений

Откройте в Visual Studio 2015 меню Отладка и щелкните на кнопке Начать отладку. Теперь попробуйте перемножить два больших значения.

В поле `Left Operand` наберите число `9876543`, а в поле `Right Operand` — `9876543`. Щелкните на пункте `* Multiplication`, затем на кнопке `Calculate`. В поле `Result` формы появится значение `-1195595903`. Оно отрицательное и не может быть правильным. Это значение является результатом операции умножения, которая допустила молчаливое переполнение 32-битного лимита, характерного для типа `int`.

Вернитесь в среду Visual Studio и остановите отладку.

Вызовите в окно редактора код файла MainPage.xaml.cs и найдите метод multiplyValues, который должен иметь следующий вид:

```
private void multiplyValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;

    outcome = lhs * rhs;
    expression.Text = $"{lhs} * {rhs}";
    result.Text = outcome.ToString();
}
```

В инструкции `outcome = lhs * rhs;` содержится операция умножения, которая допустила молчаливое переполнение. Отредактируйте эту инструкцию, чтобы вычисляемое значение проходило проверку:

```
outcome = checked(lhs * rhs);
```

Теперь умножение проверяется и будет выдавать исключение `OverflowException`, а не тихо возвращать неверный ответ.

Щелкните в меню Отладка на кнопке Начать отладку. Наберите в поле Left Operand число 9876543, а в поле Right Operand — 9876543. Щелкните на пункте * Multiplication, а затем на кнопке Calculate. Среда Visual Studio перейдет в режим отладки и сообщит, что умножение выдало исключение `OverflowException`. Теперь для перехвата этого исключения нужно добавить обработчик исключения и вместо простого выхода с ошибкой получить его более изящную обработку.

Щелкните в меню Отладка на кнопке Остановить отладку. В окне редактора, показывающего код файла MainPage.xaml.cs, найдите метод calculateClick.

Добавьте следующий обработчик исключения (выделен жирным шрифтом) сразу же после уже существующего в этом методе обработчика исключения FormatException:

```
private void calculateClick(object sender, RoutedEventArgs e)
{
    try
    {
        ...
    }
    catch (FormatException fEx)
    {
        result.Text = fEx.Message;
    }
    catch (OverflowException oEx)
    {
```

```

        result.Text = oEx.Message;
    }
}

```

Логика у этого обработчика исключения точно такая же, как и у обработчика исключения `FormatException`. Но пока вместо записи общего обработчика `Exception` есть смысл отделять эти обработчики друг от друга, поскольку в будущем может быть принято решение обрабатывать их исключения по-разному.

Чтобы выполнить сборку и запуск приложения, щелкните в меню Отладка на кнопке Начать отладку. Наберите в поле Left Operand число 9876543, а в поле Right Operand — 9876543. Щелкните на пункте * Multiplication, а затем на кнопке Calculate.

Второй обработчик исключения успешно перехватит `OverflowException` и выведет в поле Result сообщение «*Arithmetic operation resulted in an overflow*» («Арифметическая операция привела к переполнению»).

Вернитесь в Visual Studio и остановите отладку.

ОБРАБОТКА ИСКЛЮЧЕНИЙ И ОТЛАДЧИК VISUAL STUDIO

По умолчанию отладчик Visual Studio лишь останавливает выполнение приложения, запущенного в режиме отладки, и оповещает об исключениях, не прошедших обработку. Иногда будет полезно иметь возможность отладить сами обработчики исключений, и в таком случае вам нужно получить возможность трассировки исключений от момента их выдачи приложением до перехвата. Сделать это совсем нетрудно. Щелкните в меню Отладка на пункте Окна, а затем на пункте Параметры исключений. Ниже окна редактора появится панель Параметры исключений (рис. 6.5).

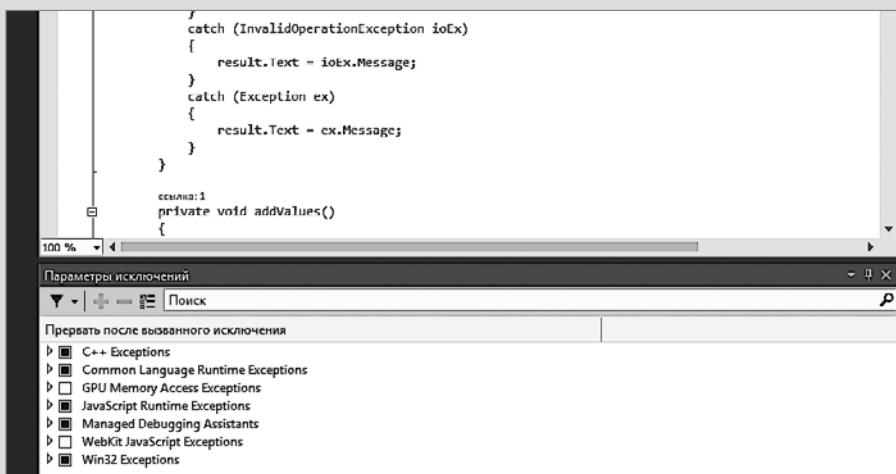


Рис. 6.5

Раскройте на панели Параметры исключений строку Common Language Runtime Exceptions, прокрутите список и установите флажок System.OverflowException (рис. 6.6).

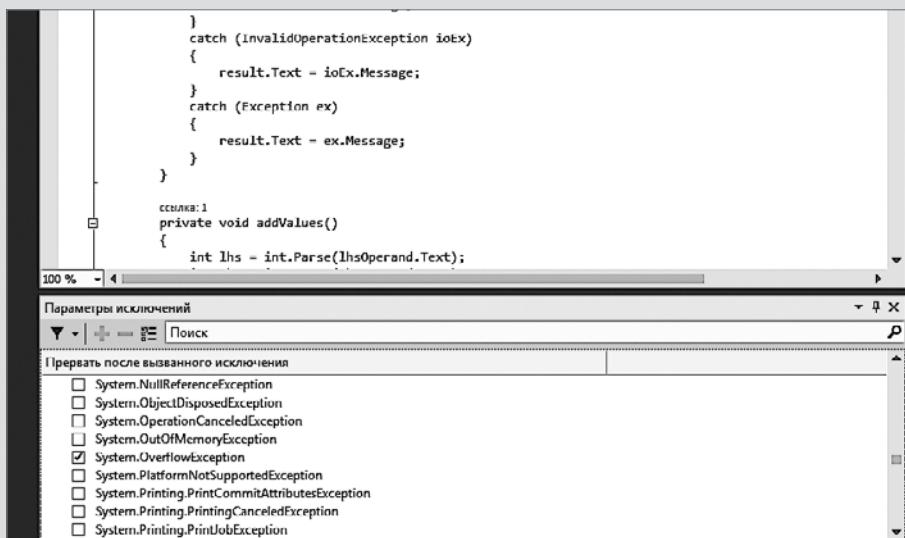


Рис. 6.6

Теперь при выдаче таких исключений, как OverflowException, Visual Studio будет переходить в отладчик, и для шага с заходом в отладчик исключения можно воспользоваться кнопкой Шаг с заходом на панели инструментов отладчика.

Выдача исключений

Предположим, вы создаете метод по имени `monthName`, который получает один `int`-аргумент и возвращает название соответствующего месяца. Например, `monthName(1)` возвращает January, `monthName(2)` возвращает February и т. д. Вопрос в том, что должен возвратить метод, если целочисленный аргумент меньше 1 или больше 12? Лучшим ответом будет, что метод вообще не должен ничего возвращать, он должен выдать исключение. Библиотеки классов .NET Framework содержат множество классов исключений, специально разработанных для подобных ситуаций. В большинстве случаев окажется, что один из таких классов описывает условия именно вашей исключительной ситуации. (Если нет, вы легко сможете создать собственный класс исключения, но прежде чем сделать это, вам нужно будет расширить свои познания в языке C#.) В данном случае вам как раз подойдет имеющийся в .NET Framework класс

`ArgumentOutOfRangeException`. Как показано в следующем примере, выдать исключение можно, воспользовавшись инструкцией `throw`:

```
public static string monthName(int month)
{
    switch (month)
    {
        case 1 :
            return "January";
        case 2 :
            return "February";
        ...
        case 12 :
            return "December";
        default :
            throw new ArgumentOutOfRangeException("Bad month");
    }
}
```

Инструкции `throw` для выдачи исключения нужен объект исключения. Этот объект содержит подробности исключения, в том числе любые сообщения об ошибках. В данном примере используется выражение, создающее новый объект `ArgumentOutOfRangeException`. Этот объект инициализируется строкой, которая заполняет его свойство `Message` путем использования конструктора. Более подробно конструкторы рассматриваются в главе 7 «Создание классов и объектов и управление ими».

В следующих упражнениях вы приступите к модификации проекта `MathsOperators`, которая позволит выдавать исключение, если пользователь попробует выполнить вычисление без указания оператора.



ПРИМЕЧАНИЕ Это упражнение носит несколько надуманный характер, поскольку при любой более или менее качественной конструкции приложению будет предоставленся оператор по умолчанию, но в данном случае оно играет сугубо иллюстративную роль.

Выдача исключения

Щелкните в меню Отладка среди Visual Studio 2015 на пункте Начать отладку. Наберите в поле Left Operand число 24, а в поле Right Operand — число 36, после чего щелкните на кнопке Calculate.

В полях Expression и Result ничего не появится. Тот факт, что вы не выбрали вариант оператора, может обнаружиться не сразу. Неплохо бы было написать в поле Result диагностическое сообщение.

Вернитесь в Visual Studio и остановите отладку. В окне редактора, отображающего код файла `MainPage.xaml.cs`, найдите и изучите метод `calculateClick`, который должен иметь следующий вид:

```
private int calculateClick(object sender, RoutedEventArgs e)
{
    try
    {
        if ((bool)addition.IsChecked)
        {
            addValues();
        }
        else if ((bool)subtraction.IsChecked)
        {
            subtractValues();
        }
        else if ((bool)multiplication.IsChecked)
        {
            multiplyValues();
        }
        else if ((bool)division.IsChecked)
        {
            divideValues();
        }
        else if ((bool)remainder.IsChecked)
        {
            remainderValues();
        }
    }
    catch (FormatException fEx)
    {
        result.Text = fEx.Message;
    }
    catch (OverflowException oEx)
    {
        result.Text = oEx.Message;
    }
}
```

Поля сложения, вычитания, умножения, деления и извлечения остатка от деления представляют собой появляющиеся в форме переключатели. У каждого переключателя есть свойство по имени `IsChecked`, в котором указывается, что пользователь выбрал именно его. Свойство `IsChecked` является обнуляемым булевым значением, устанавливаемым в `true`, если переключатель выбран, или `false`, если не выбран. (Обнуляемые значения будут более подробно рассмотрены в главе 8 «Основные сведения о значениях и ссылках».) Каскад из инструкций `if` поочередно проверяет каждый переключатель, отыскивая тот из них, который был выбран. (Переключатели обладают свойством взаимоисключения, поэтому пользователь может выбрать только одно положение.) Если не выбрано ни одно из положений, ни одно из условий `if`-инструкций не будет вычисляться в `true` и не будет вызван ни один из методов вычислений.

Можно попытаться решить проблему путем добавления к каскаду `if-else` еще одной инструкции `else` для записи сообщения в имеющееся в форме текстовое

поле `result`, но лучше будет отделить обнаружение ошибки и оповещение о ее возникновении от ее перехвата и обработки.

Добавьте к концу списка инструкций `if-else` еще одну инструкцию `else` и выдайте исключение `InvalidOperationException`. Соответствующий код выделен в следующем примере жирным шрифтом:

```
if ((bool)addition.IsChecked)
{
    addValues();
}
...
else if ((bool)remainder.IsChecked)
{
    remainderValues();
}
else
{
    throw new InvalidOperationException("No operator selected");
}
```

Для сборки и запуска приложения щелкните в меню Отладка на кнопке Начать отладку.

Наберите в поле Left Operand число 24, в поле Right Operand — число 36, а затем щелкните на кнопке Calculate.

Среда Visual Studio обнаружит, что ваше приложение выдает исключение `InvalidOperationException`, и откроет диалоговое окно исключения. Приложение выдало исключение, но код его пока не перехватил.

Щелкните в меню Отладка на кнопке Остановить отладку.

Теперь, после того как записана инструкция `throw` и проверено, что она выдает исключение, вы напишете обработчик этого исключения.

Перехват исключений

В окне редактора, отображающем код файла `MainPage.xaml.cs`, добавьте в метод `calculateClick`, сразу же за двумя уже имеющимися обработчиками исключений, следующий обработчик, выделенный жирным шрифтом:

```
...
catch (FormatException fEx)
{
    result.Text = fEx.Message;
}
catch (OverflowException oEx)
{
    result.Text = oEx.Message;
}
```

```
catch (InvalidOperationException ioEx)
{
    result.Text = ioEx.Message;
}
```

Этот код будет перехватывать исключение `InvalidOperationException`, выдаваемое в том случае, когда не выбран ни один из переключателей математических операторов.

Щелкните в меню Отладка на кнопке Начать отладку. Наберите в поле Left Operand число 24, в поле Right Operand — число 36, а затем щелкните на кнопке Calculate. В поле Result появится сообщение «No operator selected» («Не выбран ни один из операторов»).



ПРИМЕЧАНИЕ Если вы попадете в отладчик Visual Studio, то, вероятнее всего, среда Visual Studio у вас настроена на перехват всех исключений общеязыковой среды выполнения (common language runtime) сразу же, как только они будут выданы. В таком случае щелкните в меню Отладка на пункте Продолжить. Как только завершите выполнение данного упражнения, не забудьте отключить перехват в Visual Studio CLR-исключений по мере их выдачи!

Вернитесь в Visual Studio и остановите отладку.

Теперь приложение стало гораздо надежнее. Но некоторые неперехватываемые исключения все же могут выдаваться, что приведет к аварийному завершению работы приложения. Например, если будет предпринята попытка деления на нуль, это приведет к выдаче необрабатываемого исключения `DivideByZeroException`. (В отличие от деления на нуль числа с плавающей точкой деление на нуль целого числа приводит к выдаче исключения.) Один из способов решения этой проблемы заключается в написании в методе `calculateClick` еще большего количества обработчиков исключений. Другое решение предусматривает добавление в конец списка обработчиков исключений общего обработчика исключений, перехватывающего исключения `Exception`. Он станет отлавливать все неожиданные исключения, о которых вы можете забыть или которые могут быть вызваны в результате каких-либо действительно необычных обстоятельств.



ПРИМЕЧАНИЕ Использование общего обработчика для перехвата исключения `Exception` не может служить оправданием для игнорирования перехвата конкретных исключений. Чем определенное подходит к обработке исключений, тем проще будет поддерживать ваш код и выявлять причины любых присущих ему или часто повторяющихся проблем. Исключение `Exception` должно использоваться только для действительно исключительных ситуаций. Чтобы сделать следующее упражнение нагляднее, в эту категорию зачислено и исключение «деление на нуль». Но убедившись в том, что данное исключение с большой долей вероятности может выдаваться в реальном приложении, лучше будет добавить в него для исключения `DivideByZeroException` отдельный обработчик.

Перехват необработанных исключений

В окне редактора, отображающего код файла MainPage.xaml.cs, добавьте к методу calculateClick в конец перечня уже имеющихся обработчиков исключений следующий обработчик исключений:

```
catch (Exception ex)
{
    result.Text = ex.Message;
}
```

Он будет перехватывать все необработанные до этого исключения независимо от их конкретного типа.

Щелкните в меню Отладка на кнопке Начать отладку. Теперь вы будете предпринимать попытки выполнения ряда вычислений, о которых заведомо известно, что они станут причиной выдачи исключений, и убеждаться в том, что все эти исключения обрабатываются должным образом.

Наберите в поле Left Operand число 24, в поле Right Operand — число 36, а затем щелкните на кнопке Calculate. Убедитесь в том, что в поле Result по-прежнему появляется диагностическое сообщение «No operator selected». Это сообщение было создано обработчиком исключения InvalidOperationException.

Наберите в поле Left Operand строку John, щелкните на пункте + Addition, а затем на кнопке Calculate. Убедитесь в том, что в поле Result появилось диагностическое сообщение «Input string was not in a correct format». Это сообщение было создано обработчиком исключения FormatException.

Наберите в поле Left Operand число 24, а в поле Right Operand — число 0. Щелкните на пункте / Division, а затем на кнопке Calculate. Убедитесь в том, что в поле Result появилось диагностическое сообщение «Attempted to divide by zero». Это сообщение было создано общим обработчиком исключений Exception.

Поэкспериментируйте с другими комбинациями значений и убедитесь в том, что условия, вызывающие выдачу исключений, обрабатываются без выхода из приложения по аварийному сбою. Когда закончите, вернитесь в среду Visual Studio и остановите отладку.

Использование блока finally

Важно уяснить, что при выдаче исключения изменяется ход выполнения программы. Это означает, что вы не можете гарантировать, что какая-то инструкция будет безусловно выполняться по завершении выполнения предыдущей инструкции, поскольку эта предыдущая инструкция может вызвать выдачу исключения. Следует напомнить, что в таком случае после выполнения кода

обработчика исключения управление будет передано следующей инструкции в блоке, содержащем этот обработчик, а не той инструкции, которая следует непосредственно за кодом, вызвавшим выдачу исключения.

Посмотрите на следующий пример, позаимствованный из кода главы 5 «Использование инструкций составного присваивания и итераций». Нетрудно предположить, что вызов `reader.Dispose` всегда будет происходить по завершении цикла `while`, ведь он находится в коде именно после цикла:

```
TextReader reader = ...;
...
string line = reader.ReadLine();
while (line != null)
{
    ...
    line = reader.ReadLine();
}
reader.Dispose();
```

Иногда невыполнение какой-то одной конкретной инструкции не вызывает никаких проблем, но во многих случаях оно может превратиться в весьма серьезную проблему. Если инструкция высвобождает ресурс, полученный в предыдущей инструкции, невозможность ее выполнения приводит к удержанию ресурса. В данном примере мы имеем дело как раз с таким случаем: когда вы открываете файл для чтения, эта операция получает ресурс (описатель файла), и вы должны гарантировать, что для высвобождения ресурса будет вызван метод `reader.Dispose`. Если этого не сделать, то рано или поздно у вас закончится лимит описателей файлов и вы уже не сможете открыть файл. Если проблема с описателями файлов вам покажется несущественной, подумайте о подключении к базе данных вместо этого.

Способ обеспечения безусловного запуска инструкции независимо от того, было или не было выдано исключение, состоит в записи инструкции внутри блока `finally`. Этот блок должен находиться сразу же после блока `try` или сразу же после последнего обработчика, расположенного после блока `try`. Как только программа входит в блок `try`, связанный с блоком `finally`, код, находящийся в блоке `finally`, будет безусловно выполнен, даже если будет выдано исключение. Если исключение выдается и обрабатывается локально, то сначала выполняется код обработчика исключения, а затем уже блок `finally`. Если же исключение не перехватывается локально (то есть среда выполнения, чтобы найти обработчик, вынуждена вести поиск по перечню вызывающих методов), то сначала выполняется блок `finally`. Как бы то ни было, блок `finally` выполняется всегда.

В качестве решения проблемы с выполнением метода `reader.Dispose` может послужить следующий код:

```
TextReader reader = ...;
...
try
{
    string line = reader.ReadLine();
    while (line != null)
    {
        ...
        line = reader.ReadLine();
    }
}
finally
{
    if (reader != null)
    {
        reader.Dispose();
    }
}
```

Даже если исключение выдается в момент чтения файла, блок `finally` гарантирует безусловное выполнение инструкции `reader.Dispose`. Еще один способ выхода из этой ситуации будет показан в главе 14 «Использование сборщика мусора и управление ресурсами».

Выводы

В данной главе вы узнали, как с помощью конструкций `try` и `catch` перехватываются и обрабатываются исключения. Увидели, как с помощью ключевых слов `checked` и `unchecked` можно включать и выключать проверку целочисленного переполнения. Научились выдавать исключения в случае обнаружения вашим кодом исключительной ситуации и увидели, как для обеспечения безусловного выполнения важного кода даже при условии выдачи исключения используется блок `finally`.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 7.

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Перехватить конкретное исключение	<p>Напишите обработчик исключения, перехватывающий конкретный класс исключений, например:</p> <pre>try { ... } catch (FormatException fEx) { ... }</pre>
Обеспечить постоянную проверку целочисленной арифметики на переполнение	<p>Воспользуйтесь ключевым словом checked, например:</p> <pre>int number = Int32.MaxValue; checked { number++; }</pre>
Выдать исключение	<p>Воспользуйтесь инструкцией throw, например:</p> <pre>throw new FormatException(source);</pre>
Перехватить все исключения в одном обработчике исключений	<p>Напишите обработчик исключений, перехватывающий Exception, например:</p> <pre>try { ... } catch (Exception ex) { ... }</pre>
Гарантировать безусловное выполнение кодового фрагмента, даже если выдано исключение	<p>Запишите код внутри блока finally, например:</p> <pre>try { ... } finally { // выполняется всегда }</pre>

Часть II

Основные сведения об объектной модели C#

В части I вы узнали о том, как объявляются переменные, как для создания переменных используются операторы, как вызываются методы и пишутся многие инструкции, необходимые для реализации метода. Теперь у вас достаточно знаний, чтобы перейти к следующей стадии — объединению методов и данных в свои собственные функциональные структуры данных. Как это делается, показано в главах части II. Здесь вы узнаете о том, что такое классы и структуры — два фундаментальных типа, используемых для моделирования сущностей, и о других элементах, из которых образуется обычное приложение на языке C#. В частности, вы увидите, как в C# на основе определений классов и структур создаются объекты и типы значений и как общеязыковая среда выполнения (common language runtime (CLR)) управляет жизненным циклом этих элементов. Вы узнаете, как с помощью наследования создаются семейства классов, и научитесь объединять элементы путем создания массивов.

7

Создание классов и объектов и управление ими

Прочитав эту главу, вы научитесь:

- определять класс, содержащий родственный набор методов и элементов данных;
- управлять доступностью компонентов класса с помощью ключевых слов `public` и `private`;
- создавать объекты с помощью ключевого слова `new` для привлечения к их созданию конструктора;
- записывать и вызывать собственные конструкторы;
- создавать с помощью ключевого слова `static` методы и данные, которые могут совместно использоваться экземплярами одного и того же класса;
- разбираться в способах создания безымянных классов.

Среда выполнения Windows, а также среда Microsoft .NET Framework содержат тысячи классов. Некоторыми из них, включая `Console` и `Exception`, вы уже пользовались. Классы обеспечивают удобный механизм для моделирования тех сущностей, с которыми работают приложения. Сущность может представлять собой конкретный элемент, например клиента, или что-либо более абстрактное, например транзакцию. Часть процесса проектирования любой системы осозначают определение сущностей, играющих важную роль для реализуемых системой процессов, а затем анализ с целью определения информации, которая должна содержаться в той или иной сущности, и операций, которые должны выполняться этими сущностями. Информация, содержащаяся в классе, хранится в виде полей, а для реализации операций, которые могут выполняться классом, используются методы.

Основные сведения о классификации

В слове «классификация» корнем является «класс». При проектировании класса вы систематизируете информацию и поведение, создавая их осмысленное единство. Это упорядочение является актом классификации, которым занимаются не только программисты. Например, у всех автомобилей одинаковое поведение (ими можно управлять, их можно останавливать, ускорять и т. д.) и одинаковые свойства (у них есть руль, двигатель и пр.). Люди используют слово «машина» для обозначения объекта, обладающего этими общими характеристиками поведения и свойствами. Пока все соглашаются со значением данного слова, система работает хорошо и сложные, но точные мысли вы можете выражать в краткой форме. Трудно представить, как без классификации люди вообще смогли бы размышлять или общаться друг с другом.

Учитывая, что классификация так глубоко укоренилась в нашем способе мышления и общения, есть смысл попробовать создавать программы, классифицируя различные понятия, присущие задаче и ее решению, и моделируя затем эти классы в языке программирования. Именно это позволяют сделать объектно-ориентированные языки, к которым относится и Microsoft Visual C#.

Цель инкапсуляции

При определении классов применяется такое важное понятие, как инкапсуляция. Идея инкапсуляции заключается в том, что программа, использующая класс, не должна брать в расчет то, что делается внутри класса, — она просто создает экземпляр класса и вызывает методы, содержащиеся в этом классе. Пока эти методы работают в строгом соответствии со своим предназначением, программе не нужно знать подробности их реализации. Например, когда вызывается метод `Console.WriteLine`, вам не хочется вникать во все сложности того, как класс `Console` физически организован для данных, выводимых на экран. Классу, возможно, потребуется поддерживать все виды внутренней информации о состоянии, позволяющем выполнить имеющиеся в нем разнообразные методы. Эта дополнительная информация о состоянии и внутренняя работа скрыты от программы, использующей класс. Поэтому инкапсуляцию иногда называют скрытием информации. Инкапсуляция преследует две цели:

- ❑ объединить методы и данные внутри класса, то есть поддержать классификацию;
- ❑ управлять доступностью методов и данных, то есть контролировать использование класса.

Определение и использование класса

Для определения нового класса в C# используется ключевое слово `class`. Данные и методы класса помещаются в его тело между двумя фигурными скобками. В следующем примере показан класс по имени `Circle`, в котором содержится один метод (для вычисления площади круга) и один элемент данных (радиус окружности):

```
class Circle
{
    int radius;
    double Area()
    {
        return Math.PI * radius * radius;
    }
}
```



ПРИМЕЧАНИЕ В классе `Math` содержатся методы для выполнения математических вычислений и поля с математическими константами. В поле `Math.PI` содержится значение 3,14159265358979, приблизительно равное значению числа π .

В теле класса содержатся обычные методы (такие, как `Area`) и поля (такие, как `radius`). Ранее уже упоминалось, что переменные в классе называются полями. Как объявляются переменные, было показано в главе 2 «Работа с переменными, операторами и выражениями», а как создаются методы, вы видели в главе 3 «Создание методов и применение областей видимости», поэтому незнакомый синтаксис здесь практически отсутствует.

Класс `Circle` можно использовать так же, как вы использовали другие типы, уже встречавшиеся при чтении книги. Создается переменная, в качестве типа которой указывается `Circle`, а затем эта переменная инициализируется какими-нибудь подходящими данными, например:

```
Circle c;           // Создание Circle-переменной
c = new Circle(); // Ее инициализация
```

В этом коде заслуживает внимания использование ключевого слова `new`. Ранее при инициализации переменной, имеющей тип `int` или `float`, вы просто присваивали ей значение:

```
int i;
i = 42;
```

С переменной типа класса сделать то же самое невозможно. Одной из причин является то, что в C# не предоставляется синтаксис для присвоения переменным значений литерала класса. Вы не можете написать следующую инструкцию:

```
Circle c;  
c = 42;
```

Кроме всего прочего, каким тогда будет `Circle`-эквивалент числа 42? Есть еще одна причина, имеющая отношение к способу, который используется средой выполнения для выделения памяти переменным типов классов и управления этой памятью. Она будет рассмотрена в главе 8 «Основные сведения о значениях и ссылках». А пока просто примите как данность то, что ключевое слово `new` создает новый экземпляр класса, который чаще всего называют объектом.

Но присвоить экземпляр класса другой переменной того же типа можно напрямую:

```
Circle c;  
c = new Circle();  
Circle d;  
d = c;
```

И тем не менее по причинам, рассматриваемым в главе 8, все не так просто, как может показаться на первый взгляд.



ВНИМАНИЕ Не следует путать понятия класса и объекта. Класс является определением типа. А объект является экземпляром этого типа, создаваемым при выполнении программы. Экземплярами одного и того же класса могут быть несколько различных объектов.

Управление доступностью

Как ни удивительно, но в данном виде класс `Circle` практически бесполезен. Изначально, когда методы и данные инкапсулируются внутри класса, этот класс отгораживается от внешнего мира. Определенные в классе поля (такие, как `radius`) и методы (такие, как `Area`) могут использоваться другими методами внутри класса, но только не во внешнем мире, то есть они являются закрытыми собственностью класса. Получается, что, несмотря на то что вы можете создать в программе объект `Circle`, вы не можете обратиться к его полю `radius` или вызвать его метод `Area`, и поэтому класс для вас пока что бесполезен! Но используя ключевые слова `public` и `private`, можно изменить определение поля или метода и управлять их доступностью за пределами экземпляра класса.

- ❑ Методы или поля являются закрытыми (`private`), если они доступны только внутри класса. Чтобы объявить метод или поле закрытым, перед его объявлением нужно указать ключевое слово `private`. Как уже упоминалось, этот статус приобретается по умолчанию, но во избежание путаницы лучше все же определять закрытый статус полей и методов явным образом.

- Методы или поля являются открытыми (`public`), если они доступны как внутри класса, так и за его пределами. Чтобы объявить метод или поле открытыми, перед их определением нужно указать ключевое слово `public`.

Вернемся к классу `Circle`. На этот раз `Area` объявлен открытым методом, а `radius` объявлен закрытым полем:

```
class Circle
{
    private int radius;

    public double Area()
    {
        return Math.PI * radius * radius;
}
```



ПРИМЕЧАНИЕ Если у вас есть опыт программирования на C++, имейте в виду, что после ключевых слов `public` и `private` двоеточие не ставится. Ключевое слово должно повторяться для каждого объявления поля и метода.

Хотя `radius` объявлен закрытым полем, недоступным за пределами класса, доступ к нему можно получить из класса `Circle`. Метод `Area` находится внутри класса `Circle`, следовательно, из тела метода `Area` имеется доступ к полю `radius`.

Но практическая ценность класса все еще невелика, поскольку в нем отсутствует способ инициализации поля `radius`. Чтобы исправить положение, можно воспользоваться конструктором.



СОВЕТ Запомните, что переменные, объявленные в методе, изначально пребывают в неинициализированном состоянии. А поля в классе в зависимости от их типа автоматически инициализируются в нуль, `false` или `null`. Но все же лучше предоставлять инициализируемым полям явные значения.

ЗАДАНИЕ ИМЕН И ДОСТУПНОСТЬ

У многих организаций есть свой фирменный стиль, и они требуют от разработчиков придерживаться его при написании кода. Частью этого стиля обычно являются правила подбора имен идентификаторов. Чаще всего такие правила призваны облегчить сопровождение кода. Следующие рекомендации носят общий характер и связаны с соглашением о задании имен для полей и методов на основе доступности компонентов класса, хотя в C# подобные требования не выдвигаются.

Идентификаторы для открытых компонентов класса должны начинаться с прописной буквы. Например, `Area` начинается с «A» (а не с «а»), потому что это открытый метод. Эта система называется схемой задания имен PascalCase, потому что впервые она была применена в языке Pascal.

Идентификаторы для закрытых компонентов (включая локальные переменные) должны начинаться с буквы в нижнем регистре. Например, `radius` начинается с «`r`» (а не с «`R`»), потому что это закрытое поле. Эта система называется схемой задания имен в смешанном регистре (`camelCase`).



ПРИМЕЧАНИЕ В некоторых организациях схема `camelCase` используется только для методов, а в отношении имен закрытых полей придерживаются схемы задания имен, начинающихся с символа подчеркивания, например, `_radius`. Но в примерах данной книги для задания имен закрытых методов и полей мы будем придерживаться схемы, использующей смешанный регистр.

Из этого правила есть только одно исключение: имена классов должны начинаться с прописной буквы и конструкторы должны в точности соответствовать имени своего класса, поэтому имя закрытого конструктора должно начинаться с заглавной буквы.



ВНИМАНИЕ Не следует объявлять два открытых компонента класса, чьи имена различаются только регистром символов. Если поступить таким образом, то разработчики, использующие другие языки, нечувствительные к регистру символов (например, Microsoft Visual Basic), могут лишиться возможности интегрировать ваш класс в свои решения.

Работа с конструкторами

Когда ключевое слово `new` используется для создания объекта, среди выполнения нужно сконструировать этот объект, используя определение класса. Она должна забрать часть памяти у операционной системы, заполнить ее полями, определенными в классе, а затем для выполнения любой требуемой инициализации воспользоваться конструктором.

Конструктор — это специальный метод, автоматически запускаемый при создании экземпляра класса. У него такое же имя, что и у класса, и он может принимать параметры, но не может возвращать значение (даже типа `void`). Конструктор должен быть у каждого класса. Если он не будет написан, компилятор автоматически создаст для вас пассивный конструктор. (Но созданный компилятором пассивный конструктор практически ничего не делает.) Вы можете, не прилагая особых усилий, создать собственный пассивный конструктор. Просто добавьте открытый метод, не возвращающий значение, и дайте ему такое же имя, как у класса. В следующем примере показан класс `Circle` с пассивным конструктором, инициализирующим поле `radius` значением 0:

```
class Circle
{
    private int radius;
```

```
public Circle() // пассивный конструктор
{
    radius = 0;
}
public double Area()
{
    return Math.PI * radius * radius;
}
```



ПРИМЕЧАНИЕ В манере выражений, свойственных C#, пассивным считается конструктор, не принимающий никаких параметров. Неважно, чем или кем он был создан, компилятором или вами, это все равно пассивный конструктор. Вы можете также написать активные конструкторы (принимающие параметры), что и будет показано в разделе «Перегрузка конструкторов».

В данном примере конструктор помечен как открытый (`public`). Если это ключевое слово не указано, конструктор будет закрытым, как и любые другие метод или поле. Если конструктор закрытый (`private`), он не может быть использован за пределами класса, что не дает вам возможности создавать объекты `Circle` из методов, не являющихся частью класса `Circle`. Поэтому может сложиться мнение, что закрытые конструкторы не имеют особой ценности. Им находится применение, но этот вопрос выходит за рамки рассматриваемой темы.

После добавления открытого конструктора вы можете воспользоваться классом `Circle` и на деле испытать имеющийся у него метод `Area`. Обратите внимание на форму записи с точкой, применяемую для вызова метода `Area` в отношении `Circle`-объекта:

```
Circle c;
c = new Circle();
double areaOfCircle = c.Area();
```

Перегрузка конструкторов

Но это еще не все. Теперь вы можете объявить переменную, имеющую тип `Circle`, воспользоваться ею для ссылки на только что созданный `Circle`-объект, а затем вызвать метод `Area`. Осталась еще одна, последняя проблема. Площадь всех `Circle`-объектов всегда будет равна нулю, поскольку пассивный конструктор устанавливает значение поля `radius` в нуль и оно так и остается с нулевым значением. Поле `radius` является закрытым, поэтому у нас нет простого способа изменить его значение после инициализации. Конструктор является всего лишь специальной разновидностью метода, и он, как и все другие методы, может быть перегружен. Точно так же, как существует несколько версий метода `Console.WriteLine`, каждая из которых принимает разные параметры, могут существовать и различные созданные вами версии конструктора. То есть вы можете добавить

к классу `Circle` еще один конструктор с параметрами, указывающими на используемый радиус окружности:

```
class Circle
{
    private int radius;

    public Circle() // пассивный конструктор
    {
        radius = 0;
    }

    public Circle(int initialRadius) // перегруженный конструктор
    {
        radius = initialRadius;
    }

    public double Area()
    {
        return Math.PI * radius * radius;
    }
}
```



ПРИМЕЧАНИЕ Порядок следования конструкторов в классе не имеет никакого значения, конструкторы можно определять в наиболее удобном для вас порядке.

Затем этим конструктором можно воспользоваться при создании нового объекта `Circle`:

```
Circle c;
c = new Circle(45);
```

При сборке приложения компилятор определяет, какой конструктор должен быть вызван, основываясь на параметрах, указанных для оператора `new`. В данном примере было передано целое число, поэтому компилятор создаст код, вызывающий конструктор, принимающий `int`-параметр.

Вам нужно знать о важной особенности языка C#: если вы создаете для класса свой собственный конструктор, компилятор не создает пассивный конструктор. Поэтому если вы написали собственный конструктор, принимающий один или несколько параметров, и вам нужен еще и пассивный конструктор, то его придется создавать отдельно.

В следующем упражнении вами будет объявлен класс, моделирующий точку в двумерном пространстве. Класс будет состоять из двух закрытых полей для хранения *x*- и *y*-координат точки и предоставлять конструкторы для инициализации этих полей. Для создания экземпляров класса вы воспользуетесь ключевым словом `new` и вызовом конструкторов.

РАЗДЕЛЯЕМЫЕ КЛАССЫ

В классе могут содержаться несколько методов, полей и конструкторов, а также другие элементы, рассматриваемые в следующих главах. Класс, имеющий высокоразвитые функциональные возможности, может стать слишком большим. Программируя на C#, вы можете разбить исходный код класса на несколько отдельных файлов, получив возможность организовать определение большого класса в небольших частях, с которыми проще работать. Это свойство используется средой Visual Studio 2015 для приложений, разрабатываемых под универсальную платформу Windows (Universal Windows Platform (UWP)), где исходный код, который разработчик может редактировать, сохраняется в отдельном файле, обособляясь таким образом от кода, создаваемого средой Visual Studio при каждом изменении разметки формы.

При разбиении класса на несколько файлов части класса определяются с использованием в каждом файле ключевого слова `partial`. Например, если класс `Circle` разбит на два файла с именами `circ1.cs` (в этом файле содержатся конструкторы) и `circ2.cs` (в этом файле содержатся методы и поля), то содержимое `circ1.cs` выглядит следующим образом:

```
partial class Circle
{
    public Circle() // пассивный конструктор
    {
        this.radius = 0;
    }

    public Circle(int initialRadius) // перегруженный конструктор
    {
        this.radius = initialRadius;
    }
}
```

А содержимое `circ2.cs` имеет следующий вид:

```
partial class Circle
{
    private int radius;

    public double Area()
    {
        return Math.PI * this.radius * this.radius;
    }
}
```

При компиляции класса, разбитого на два отдельных файла, компилятору следует предоставить все файлы.

Написание конструкторов и создание объектов

Откройте в среде Visual Studio 2015 проект *Classes*, который находится в папке *\Microsoft Press\VCSBS\Chapter 7\Classes* вашей папки документов.

Дважды щелкните в обозревателе решений на файле *Program.cs*, чтобы его код был выведен в окно редактора. Найдите в классе *Program* метод *Main*. Этот метод вызывает метод *doWork*, заключенный в блок *try*, за которым следует обработчик исключения. Используя блок *try-catch*, вы можете написать код, который обычно помещается в *Main*, не в этом методе, а в методе *doWork*, и быть в полной уверенности, что этот блок перехватит и обработает любые исключения. Метод *doWork* на данный момент не содержит ничего, кроме комментария *// TODO:*.



СОВЕТ Комментарии *TODO* часто используются разработчиками в качестве напоминания о том, что они отложили работу над фрагментом кода и к ней нужно будет вернуться. В этих комментариях часто содержится описание той работы, которую нужно сделать, например, *// TODO: Реализовать метод doWork*. Visual Studio распознает эту форму комментария, и вы можете быстро найти такие комментарии в любом месте приложения, воспользовавшись окном Список задач. Чтобы вывести это окно на экран, нужно щелкнуть в меню Вид на пункте Список задач. Изначально окно с этим названием открывается ниже окна редактора (рис. 7.1). В нем будут выведены списком все комментарии *TODO*. Затем, чтобы перейти непосредственно к соответствующему коду, который будет отображен в окне редактора, можно будет дважды щелкнуть на любом из этих комментариев.

Описание	Проект	Файл	Строка
TODO:	Classes	Point.cs	13
TODO:	Classes	Program.cs	13

Рис. 7.1

Выполните в окне редактора файл Point.cs. В этом файле определяется класс по имени **Point**, который будет вами использоваться для представления местонахождения точки в двумерном пространстве, определяемого парой координат *x* и *y*. Класс **Point** на данный момент не содержит ничего, кроме еще одного комментария `// TODO:`.

Вернитесь к файлу Program.cs. Отредактируйте в классе **Program** тело метода **dowork**, заменив комментарий `// TODO:` следующей инструкцией:

```
Point origin = new Point();
```

Эта инструкция создает новый экземпляр класса **Point** и запускает его пассивный конструктор.

Щелкните в меню Сборка (Build) на пункте Собрать решение (Build Solution).

Код пройдет сборку без ошибок, потому что компилятор автоматически создает код для пассивного конструктора класса **Point**. Но вы не можете видеть код C# для этого конструктора, потому что компилятор не создает на этом языке никаких исходных инструкций.

Вернитесь в класс **Point**, который находится в файле Point.cs. Замените комментарий `// TODO:` открытым конструктором, принимающим два **int**-аргумента с именами *x* и *y*, а затем вызовите метод **Console.WriteLine** для отображения в консоли значений этих аргументов. Все изменения в следующем примере кода выделены жирным шрифтом:

```
class Point
{
    public Point(int x, int y)
    {
        Console.WriteLine($"x:{x}, y:{y}");
    }
}
```

Щелкните в меню Сборка на пункте Собрать решение. Теперь компилятор выдаст сообщение об ошибке: Отсутствует аргумент, соответствующий требуемому формальному параметру "x" из "Point.Point(int, int)".

Смысл этого весьма многословного сообщения состоит в том, что вызов пассивного конструктора в методе **dowork** теперь неприемлем, поскольку такого конструктора больше нет. Вы создали собственный конструктор для класса **Point**, поэтому компилятор не создает пассивный конструктор. Теперь вам нужно устранить эту проблему, написав собственный пассивный конструктор.

Отредактируйте класс **Point**, добавив открытый пассивный конструктор, вызывающий инструкцию **Console.WriteLine** для записи в консоль строки

«Default constructor called» («Вызван пассивный конструктор»). Добавляемый код выделен жирным шрифтом, а класс `Point` должен теперь приобрести следующий вид:

```
class Point
{
    public Point()
    {
        Console.WriteLine("Default constructor called");
    }

    public Point(int x, int y)
    {
        Console.WriteLine($"x:{x}, y:{y}");
    }
}
```

Щелкните в меню Сборка на пункте Собрать решение. Теперь сборка программы пройдет успешно.

Отредактируйте в файле `Program.cs` тело метода `doWork`. Объявите переменную по имени `bottomRight` с типом `Point` и инициализируйте ее новым объектом `Point`, воспользовавшись конструктором с двумя аргументами. Соответствующий код показан ниже и выделен жирным шрифтом. Предоставьте конструктору значения 1366 и 768 — координаты правой нижней точки экрана, имеющего разрешение 1366×768, которое встречается на многих планшетных устройствах. Теперь метод `doWork` должен приобрести следующий вид:

```
static void doWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1366, 768);
}
```

Щелкните в меню Отладка на пункте Запуск без отладки.

Программа пройдет сборку и будет запущена, в результате чего на консоль будет выведено следующее сообщение (рис. 7.2).

Нажмите клавишу Ввод, чтобы завершить работу программы, и вернитесь в среду Visual Studio 2015.

Теперь вы добавите к классу `Point` два `int`-поля для представления принадлежащих точке координат `x` и `y` и внесете изменения в конструктор для инициализации этих полей. Отредактируйте в файле `Point.cs` класс `Point`, добавив к нему два закрытых поля `int`-типа с именами `x` и `y`. Соответствующий код показан ниже и выделен жирным шрифтом. Класс `Point` должен приобрести следующий вид:

```
class Point
{
    private int x, y;

    public Point()
    {
        Console.WriteLine("default constructor called");
    }

    public Point(int x, int y)
    {
        Console.WriteLine($"x:{x}, y:{y}");
    }
}
```

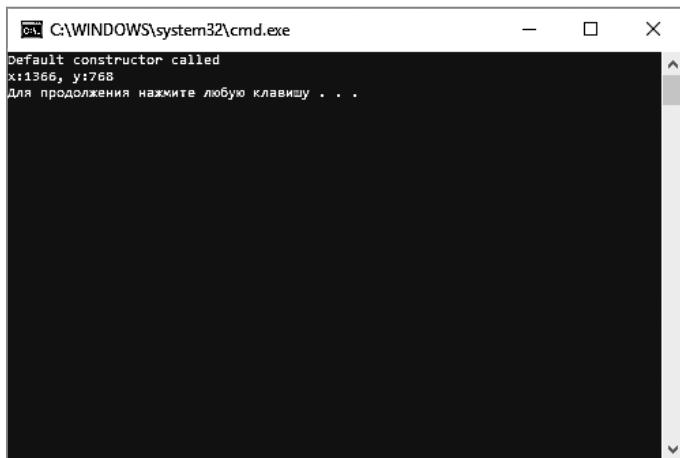


Рис. 7.2

Теперь вы отредактируете второй конструктор `Point` для инициализации полей `x` и `y` значениями параметров `x` и `y`. Но здесь вас поджидает потенциальная ловушка. Если проявить невнимательность, код конструктора может стать похожим на следующий:

```
public Point(int x, int y) // Не набирайте этот код!
{
    x = x;
    y = y;
}
```

Несмотря на то что этот код пройдет компиляцию, используемые в нем инструкции представляются неоднозначными. Как из инструкции `x = x`; компилятор узнает, что первый идентификатор `x` представляет собой поле, а второй является параметром? Ответ такой: он просто не сможет этого сделать! Параметр метода

с таким же именем, как и у поля, скрывает поле для всех инструкций в методе. Все, что фактически делает этот код, — это присваивает значения параметров им же самим, с полями он вообще ничего не делает. Разумеется, это совсем не то, что вы хотели получить.

Решение проблемы заключается в использовании ключевого слова `this`, чтобы уточнить, что здесь являются параметрами, а что полями. Указание перед именем переменной ключевого слова `this` означает «поле этого объекта».

Отредактируйте конструктор `Point`, принимающий два параметра, заменив инструкцию `Console.WriteLine` следующим кодом, выделенным жирным шрифтом:

```
public Point(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

Отредактируйте пассивный конструктор `Point` для инициализации полей `x` и `y` значениями `-1`. Показанные далее изменения выделены жирным шрифтом. Хотя здесь и нет параметров, которые могли бы вызвать путаницу, лучше все же с помощью ключевого слова `this` указать ссылки на поля:

```
public Point()
{
    this.x = -1;
    this.y = -1;
}
```

Щелкните в меню Сборка на пункте Собрать решение. Убедитесь в том, что код компилируется без ошибок или предупреждений. (Его можно запустить, но никакого вывода на экран он не выполнит.)

Методы, принадлежащие классу и совершающие операции с данными, принадлежащими конкретному экземпляру класса, называются методами экземпляра. (Сведения о других типах методов будут даны в этой главе чуть позже.) В следующем упражнении вы создадите метод экземпляра для класса `Point`, называющийся `DistanceTo` и вычисляющий расстояние между двумя точками.

Создание и вызов методов экземпляра

В среде Visual Studio 2015 добавьте к классу `Point` в проекте `Classes` сразу после конструкторов следующий открытый метод экземпляра под названием `DistanceTo`. Этот метод принимает один `Point`-аргумент по имени `other` и возвращает значение, имеющее тип данных `double`.

Метод `DistanceTo` должен иметь следующий вид:

```
class Point
{
    ...
    public double DistanceTo(Point other)
    {
    }
}
```

При выполнении следующих этапов к телу метода экземпляра `DistanceTo` будет добавлен код для вычисления и возвращения расстояния между объектом типа `Point`, задействованным для совершения вызова, и объектом типа `Point`, переданным в качестве параметра. Для этого нужно вычислить разницу между координатами *x* и *y*.

Объявите в методе `DistanceTo` локальную `int`-переменную по имени `xDiff` и проинициализируйте ее разницей значений `this.x` и `other.x`. Соответствующий код, показанный далее, выделен жирным шрифтом:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
}
```

Объявите еще одну локальную `int`-переменную по имени `yDiff` и проинициализируйте ее разницей значений `this.y` и `other.y`. Новый код выделен жирным шрифтом:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
    int yDiff = this.y - other.y;
}
```



ПРИМЕЧАНИЕ Хотя поля *x* и *y* являются закрытыми, доступ к ним все же может быть получен из других экземпляров того же класса. Важно понимать, что понятие закрытости (`private`) действует на уровне класса, а не на уровне объекта: два объекта, являющиеся экземплярами одного и того же класса, могут иметь доступ к закрытым данным друг друга, а объекты, являющиеся экземплярами другого класса, такого доступа иметь не могут.

Для вычисления расстояния можно воспользоваться теоремой Пифагора и найти квадратный корень из суммы квадратов `xDiff` и `yDiff`. Метод `Sqrt`, который можно использовать для вычисления квадратных корней, предоставляется классом `System.Math`.

Объявите переменную по имени `distance`, имеющую тип данных `double`, и воспользуйтесь ею для сохранения результата только что рассмотренного вычисления:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
    int yDiff = this.y - other.y;
    double distance = Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
}
```

Добавьте к концу метода `DistanceTo` инструкцию `return` и возвратите значение переменной `distance`:

```
public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
    int yDiff = this.y - other.y;
    double distance = Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
    return distance;
}
```

А теперь займемся тестированием метода `DistanceTo`.

Вернитесь к методу `doWork` в классе `Program`. После инструкций, объявляющих `Point`-переменные `origin` и `bottomRight`, объявите переменную по имени `distance` с типом данных `double`. Инициализируйте эту `double`-переменную результатом, получаемым при вызове метода `DistanceTo`, в отношении объекта `origin` после передачи этому методу в качестве аргумента значения переменной `bottomRight`.

Теперь метод `doWork` должен приобрести следующий вид:

```
static void doWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1366, 768);
    double distance = origin.DistanceTo(bottomRight);
}
```



ПРИМЕЧАНИЕ Как только после слова `origin` будет набран символ точки, система Microsoft IntelliSense должна вывести название метода `DistanceTo`.

Добавьте к методу `doWork` еще одну инструкцию, записывающую значение переменной `distance` в консоль с помощью метода `Console.WriteLine`.

В окончательном виде метод `doWork` должен выглядеть следующим образом:

```
static void doWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1366, 768);
    double distance = origin.DistanceTo(bottomRight);
    Console.WriteLine($"Distance is: {distance}");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки.

Убедитесь в том, что в окно консоли записано значение 1568,45465347265, а затем нажмите Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

Основные сведения о статических методах и данных

В предыдущем упражнении был использован метод `Sqrt`, принадлежащий классу `Math`. Также, когда рассматривался класс `Circle`, из класса `Math` производилось чтение поля `PI`. Если приглядеться, то способ вызова метода `Sqrt` или чтения поля `PI` был немного странным. Вы вызывали метод или читали поле самого класса, а не объекта типа `Math`. Это все равно что в коде, который был добавлен в предыдущем упражнении, попытаться написать `Point.DistanceTo` вместо `origin.DistanceTo`. Так что же все-таки произошло и как это работает?

Вам часто придется сталкиваться с тем, что не все методы естественным образом принадлежат экземпляру класса, есть еще и методы общего пользования, считающиеся таковыми по причине реализации полезной функции, не зависящей от какого-либо конкретного экземпляра класса. Типичным примером может послужить метод `WriteLine` класса `Console`, повсеместно используемый в данной книге. В качестве еще одного примера можно привести метод `Sqrt`. Если бы `Sqrt` был методом экземпляра класса `Math`, вам пришлось бы создавать объект `Math` и вызывать `Sqrt` в отношении этого объекта:

```
Math m = new Math();
double d = m.Sqrt(42.24);
```

Это было бы слишком громоздко. Объект `Math` не играл бы никакой роли в вычислении квадратного корня. Все вводимые данные, необходимые `Sqrt`, представлены в списке параметров, а результат передан назад вызывающему методу с помощью имеющегося в методе возвращаемого значения. Объекты здесь совершенно ни к чему, поэтому пихать `Sqrt` в экземпляр не имеет никакого смысла.



ПРИМЕЧАНИЕ Кроме того что в классе `Math` содержатся метод `Sqrt` и поле `PI`, там имеется множество других полезных математических методов, например `Sin`, `Cos`, `Tan` и `Log`.

В C# все методы должны быть объявлены внутри класса. Но если объявить метод или поле в качестве статического, то вызывать метод или обращаться к полю можно с использованием имени класса. Вот как выглядит объявление метода `Sqrt` класса `Math`:

```
class Math
{
    public static double Sqrt(double d)
    {
        ...
    }
    ...
}
```

А метод `Sqrt` можно вызывать следующим образом:

```
double d = Math.Sqrt(42.24);
```

Статический метод не зависит от экземпляра класса и не может обращаться к каким-либо определенным в классе методам экземпляра или полям экземпляра, он может использовать только поля и другие методы, которые обозначены ключевым словом `static`.

Создание совместно используемых полей

Определение поля в качестве статического позволяет создавать единственный экземпляр поля, совместно используемый всеми объектами, созданными из одного и того же класса. (Нестатические поля являются для каждого экземпляра объекта локальными.) В следующем примере при каждом создании нового объекта `Circle` значение статического поля `NumCircles` в классе `Circle` увеличивается конструктором `Circle` на единицу:

```
class Circle
{
    private int radius;
    public static int NumCircles = 0;

    public Circle() // пассивный конструктор
    {
        radius = 0;
        NumCircles++;
    }

    public Circle(int initialRadius) // перегруженный конструктор
    {
        radius = initialRadius;
        NumCircles++;
    }
}
```

Все `Circle`-объекты совместно используют один и тот же экземпляр поля `NumCircles`, поэтому инструкция `NumCircles++`; при создании каждого нового экземпляра увеличивает на единицу одни и те же данные. Следует заметить,

что вы не можете использовать для `NumCircles` префикс из ключевого слова `this`, поскольку `NumCircles` не принадлежит конкретному объекту.

Как показано в следующем примере, доступ к полю `NumCircles` можно получить за пределами класса, указав не `Circle`-объект, а класс `Circle`:

```
Console.WriteLine($"Number of Circle objects: {Circle.NumCircles}");
```



ПРИМЕЧАНИЕ Следует иметь в виду, что статические методы называются также методами класса. А вот статические поля полями класса обычно не называют — они просто называются статическими полями (иногда статическими переменными).

Создание статических полей с использованием ключевого слова `const`

Используя для поля префикс, состоящий из ключевого слова `const`, можно объявить поле статическим, но при этом его значение уже никогда не может быть изменено. Ключевое слово `const` является сокращением от слова `constant` — константа. Ключевое слово `static` не используется при определении `const`-поля, но оно все равно является статическим. По причинам, которые в данной книге не рассматриваются, поле может быть объявлено как `const` только в том случае, если оно относится к числовому типу (например, `int` или `double`), является строкой или перечислением. (Перечисления будут рассматриваться в главе 9 «Создание типов значений с использованием перечислений и структур».) Вот как, к примеру, в качестве поля `const` в классе `Math` объявляется константа `PI`:

```
class Math
{
    ...
    public const double PI = 3.14159265358979;
}
```

Основные сведения о статических классах

Еще одной особенностью языка C# является возможность объявлять класс статическим. Статический класс может содержать только статические элементы. (Все объекты, создаваемые с использованием класса, совместно используют одну и ту же копию этих элементов.) Статические классы предназначены для применения исключительно в качестве хранилищ совместно используемых методов и полей. Статический класс не может содержать какие-либо данные или методы экземпляров, и нет никакого смысла пытаться с помощью оператора `new` создавать из статического класса объект. Фактически вам не удастся использовать `new` для создания экземпляра объекта, даже если вы этого захотите.

(При такой попытке компилятор выдаст отчет об ошибке.) Если вам нужно выполнить любую инициализацию, у статического класса должен быть пассивный конструктор, если, конечно, он тоже объявлен с использованием ключевого слова `static`. Любые другие типы конструктора недопустимы, о чем и будет объявлено компилятором.

Если вами определена собственная версия класса `Math`, содержащая только статические элементы, она должна иметь следующий вид:

```
public static class Math
{
    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Sqrt(double x) {...}
    ...
}
```



ПРИМЕЧАНИЕ Настоящий класс `Math` таким образом не определен, поскольку в нем все-таки имеются методы экземпляров.

Инструкции использования статических элементов

При вызове статического метода или при ссылке на статическое поле нужно указывать класс, которому принадлежат эти метод или поле, например `Math.Sqrt` или `Console.WriteLine`. Инструкции использования статических элементов (`using`) позволяют вводить класс в область видимости и не указывать имя класса при доступе к статическим элементам. Эти инструкции работают практически так же, как и обычные инструкции `using`, помещающие пространства имен в область видимости. Порядок их использования показан в следующем примере:

```
using static System.Math;
using static System.Console;
...
var root = Sqrt(99.9);
WriteLine($"The square root of 99.9 is {root}");
```

Обратите внимание на использование с инструкцией `using` ключевого слова `static`. В этом примере статические методы классов `System.Math` и `System.Console` помещаются в область видимости (вы должны указать классы с их пространствами имен в развернутой форме). Затем можно будет просто вызывать методы `Sqrt` и `WriteLine`. Компилятор сам определит, какой метод какому классу принадлежит. Но за всем этим кроется потенциальная проблема сопровождения кода. Стремясь уменьшить объем набираемого кода, вам нужно выдерживать разумный баланс, учитывая те дополнительные усилия, которые потребуется приложить кому-то другому, кто будет сопровождать этот код, поскольку теперь уже непонятно, к какому классу какой метод принадлежит. В какой-то мере

помогает система IntelliSense, имеющаяся в среде Visual Studio, но для разработчика, вычитывающего код при попытке выяснить причину ошибки, это обстоятельство может все запутать. Инструкции использования статических элементов следует применять с оглядкой, — сам автор предпочитает с ними не работать, но выбор всегда остается за вами!

В заключительном упражнении этой главы вам предстоит добавить к классу `Point` закрытое статическое поле и инициализировать его нулевым значением. Инкремент этого поля счетчика будет осуществляться в обоих конструкторах. В завершение вы напишете открытый статический метод для возвращения значения закрытого статического поля, с помощью которого сможете определить количество созданных `Point`-объектов.

Создание статических элементов и вызов статических методов

В среде Visual Studio 2015 выведите в окно редактора класс `Point`. Добавьте к нему непосредственно перед конструкторами закрытое статическое `int`-поле по имени `objectCount` и инициализируйте его во время объявления нулевым значением:

```
class Point
{
    ...
    private static int objectCount = 0;
    ...
}
```



ПРИМЕЧАНИЕ При объявлении такого поля, как `objectCount`, ключевые слова `private` и `static` можно размещать в любом порядке. Но предпочтительнее сначала указать `private`, а затем `static`.

Добавьте к обоим `Point`-конструкторам инструкции для увеличения значения поля `objectCount` на единицу (в следующем примере они выделены жирным шрифтом). Теперь класс `Point` должен приобрести следующий вид:

```
class Point
{
    private int x, y;
    private static int objectCount = 0;

    public Point()
    {
        this.x = -1;
        this.y = -1;
        objectCount++;
    }
}
```

```
public Point(int x, int y)
{
    this.x = x;
    this.y = y;
    objectCount++;
}

public double DistanceTo(Point other)
{
    int xDiff = this.x - other.x;
    int yDiff = this.y - other.y;
    double distance = Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
    return distance;
}
}
```

При каждом создании объекта вызывается его конструктор. Поскольку значение `objectCount` увеличивается на единицу в каждом конструкторе, включая пассивный, в `objectCount` будет содержаться определенное количество созданных на данный момент объектов. Эта стратегия работает благодаря тому, что `objectCount` является совместно используемым статическим полем. Если бы `objectCount` был полем экземпляра, то у каждого объекта было бы свое персональное поле `objectCount` с установленным для него значением 1.

А теперь возникает вопрос: как пользователи класса `Point` смогут определить количество созданных `Point`-объектов? На данный момент поле `objectCount` является закрытым, к нему нет доступа за пределами класса. Решение открыть публичный доступ к полю `objectCount` было бы неразумным, поскольку тогда была бы нарушена инкапсуляция класса и у вас после этого не было бы никаких гарантий относительно корректности значения поля `objectCount`, поскольку его мог бы изменить кто угодно. Более удачным решением будет предоставить открытый статический метод, возвращающий значение поля `objectCount`. Именно этим вы сейчас и займитесь.

Добавьте к классу `Point` открытый статический метод по имени `ObjectCount`, возвращающий `int`-значение, но не принимающий никаких параметров. Этот метод, выделенный в следующем примере жирным шрифтом, должен возвращать значение поля `objectCount`:

```
class Point
{
    ...
    public static int ObjectCount() => objectCount;
}
```

Выполните в окне редактора код класса `Program`. Добавьте к методу `doWork` инструкцию для вывода на экран значения, возвращаемого методом `ObjectCount` класса `Point` (выделена жирным шрифтом):

```
static void doWork()
{
    Point origin = new Point();
    Point bottomRight = new Point(1366, 768);
    double distance = origin.distanceTo(bottomRight);
    Console.WriteLine($"Distance is: {distance}");
    Console.WriteLine($"Number of Point objects: {Point.ObjectCount()}");
}
```

Метод `ObjectCount` вызывается за счет ссылки на `Point`, то есть на имя класса, а не на имя `Point`-переменной, такой как `origin` или `bottomRight`. Поскольку на момент вызова `ObjectCount` уже были созданы два `Point`-объекта, метод должен возвратить значение 2.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что в окно консоли выводится сообщение «Number of Point objects: 2» (после сообщения, отображающего значение переменной `distance`). Нажмите Ввод, чтобы закрыть программу и вернуться в среду Visual Studio 2015.

Безымянные классы

Безымянным называется класс, не имеющий имени. Звучит, конечно, странно, но как будет показано далее, в некоторых ситуациях использовать такие классы весьма удобно, особенно когда используются выражения запросов. (Эти выражения будут рассматриваться в главе 20 «Отделение логики приложения и обработка событий».) А сейчас в пользу их применения нужно просто поверить.

Безымянный класс создается путем использования ключевого слова `new` и пары фигурных скобок, внутри которых определяются содержащиеся в классе поля и значения:

```
myAnonymousObject = new { Name = "John", Age = 47 };
```

Этот класс содержит открытое поле `Name`, которое инициализируется строкой «`John`», и открытое поле `Age`, которое инициализируется целым числом `47`. Компилятор определяет типы полей по типам данных, указанных при их инициализации.

Когда определяется безымянный класс, компилятор создает для него свое собственное имя, но какое именно, вам не сообщает. В связи с этим возникает, казалось бы, парадоксальная ситуация: если не известно имя класса, то как можно создать объект соответствующего типа и присвоить ему экземпляр класса? Тогда каким в показанном ранее примере кода должен быть тип переменной

myAnonymousObject? Ответ такой: вы об этом не знаете, в чем и заключается характерная особенность безымянных классов!

Но если вы воспользуетесь ключевым словом `var` и объявите `myAnonymousObject` как переменную с неявно указанным типом, как показано в следующем примере, то проблемы не будет:

```
var myAnonymousObject = new { Name = "John", Age = 47 };
```

Вспомним, что ключевое слово `var` заставляет компилятор создавать переменную того же типа, к которому относится инициализирующее ее выражение. В данном случае типом выражения станет то самое имя, которое компилятор создаст для безымянного класса.

Доступ к полям объекта можно получить, используя уже известную форму записи с точкой:

```
Console.WriteLine($"Name: {myAnonymousObject.Name} Age: {myAnonymousObject.Age}");
```

Можно даже создавать другие экземпляры одного и того же безымянного класса, но с разными значениями, например:

```
var anotherAnonymousObject = new { Name = "Diana", Age = 46 };
```

Для определения того, относятся ли два экземпляра безымянного класса к одному и тому же типу, компилятор C# использует имена, типы и порядок следования полей. В данном случае переменные `myAnonymousObject` и `anotherAnonymousObject` имеют одинаковое количество полей с одинаковыми именами и типами, которые стоят в одном и том же порядке, значит, обе переменные являются экземплярами одного и того же безымянного класса. Это означает, что вы можете использовать следующие инструкции присваивания:

```
anotherAnonymousObject = myAnonymousObject;
```



ПРИМЕЧАНИЕ Имейте в виду, что эта инструкция присваивания может не выполнить то, что вы от нее ожидаете. Более подробно вопрос о присваивании объектов-переменных будет рассмотрен в главе 8.

На содержимое безымянных классов накладывается довольно много ограничений. Например, безымянные классы могут содержать только открытые поля, все поля должны быть инициализированы, они не могут быть статическими, и вы не можете определять для этих классов какие-либо методы. Безымянные классы будут время от времени использоваться в данной книге, по мере чтения вы сможете с ними поближе познакомиться.

Выводы

В данной главе вы увидели, как определяются новые классы. Узнали, что по умолчанию поля и методы класса являются закрытыми и недоступными для кода, находящегося за пределами класса, но чтобы открыть поля и методы для доступа к ним из внешнего мира, можно воспользоваться ключевым словом `public`. Вы увидели, как для создания нового экземпляра класса используется ключевое слово `new` и как определяются конструкторы, которые могут инициализировать экземпляры класса. И наконец, вы увидели, как создаются статические поля и методы для предоставления данных и операций, которые не зависят от конкретных экземпляров класса.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 8.

Если вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Объявить класс	Напишите ключевое слово <code>class</code> , за которым укажите имя класса, а затем поставьте открывающую и закрывающую фигурные скобки. Методы и поля класса объявляются между этими фигурными скобками, например:
	<pre>class Point { ... }</pre>
Объявить конструктор	Напишите метод, имя которого совпадает с именем класса, а возвращаемый тип (даже <code>void</code>) отсутствует, например:
	<pre>class Point { public Point(int x, int y) { ... } }</pre>
Вызвать конструктор	Воспользуйтесь ключевым словом <code>new</code> и укажите конструктор с соответствующим набором параметров, например:
	<pre>Point origin = new Point(0, 0);</pre>

Чтобы	Сделайте следующее
Объявить статический метод	Напишите перед объявлением метода ключевое слово static, например: <pre>class Point { public static int ObjectCount() { ... } }</pre>
Вызвать статический метод	Напишите имя класса, после него поставьте точку, а далее укажите имя метода, например: <pre>int pointsCreatedSoFar = Point.ObjectCount();</pre>
Объявить статическое поле	Поставьте ключевое слово static перед типом поля, например: <pre>class Point { ... private static int objectCount; }</pre>
Объявить поле const	Напишите перед объявлением поля ключевое слово const и не указывайте ключевое слово static, например: <pre>class Math { ... public const double PI = ...; }</pre>
Получить доступ к статическому полю	Напишите имя класса, поставьте после него точку, а далее укажите имя статического поля, например: <pre>double area = Math.PI * radius * radius;</pre>

8 Основные сведения о значениях и ссылках

Прочитав эту главу, вы научитесь:

- объяснять разницу между типом значений и типом ссылок;
- изменять способ передачи аргументов в качестве параметров методов с помощью ключевых слов `ref` и `out`;
- превращать значение в ссылку, используя упаковку (`boxing`);
- превращать ссылку обратно в значение, используя распаковку (`unboxing`) и приведение типов (`casting`).

В главе 7 «Создание классов и объектов и управление ими» было показано, как объявляются свои собственные классы и как создаются объекты с помощью использования ключевого слова `new`. Там же было показано, как с помощью конструктора инициализируется объект. В данной главе вы узнаете, чем характеристики простых типов, таких как `int`, `double` и `char`, отличаются от характеристик типов классов.

Копирование типов значений переменных и классов

Большинство элементарных типов, встроенных в C#, например `int`, `float`, `double` и `char` (но не `string` — по причинам, которые вскоре будут рассмотрены), обобщенно называются типами значений. У этих типов фиксированный размер, и когда вы объявляете переменную как тип значения, компилятор создает код, занимающий блок памяти, достаточный по размеру для хранения соответствующего значения. Например, объявление `int`-переменной заставляет компилятор выделить для хранения целочисленного значения 4 байта памяти (32 бита).

Инструкция, присваивающая значение (например, 42) `int`-переменной, приводит к тому, что значение копируется в этот блок памяти.

Такие типы классов, как `Circle`, рассмотренный в главе 7, обрабатываются по-другому. Когда объявляется `Circle`-переменная, компилятор не создает код, распределяющий блок памяти, размер которого достаточен для хранения значения типа `Circle`, а просто отводит небольшой участок памяти, где потенциально может содержаться адрес другого блока памяти (или ссылка на него), в котором содержится `Circle`-значение (адрес, указывающий место элемента в памяти). Память под реальный `Circle`-объект выделяется, только когда для создания объекта используется ключевое слово `new`. Класс является примером ссылочного типа. В ссылочных типах содержатся ссылки на блоки памяти. Для создания эффективных программ на C#, которые в полной мере используют среду Microsoft .NET Framework, нужно разобраться в том, чем типы значений отличаются от ссылочных типов.



ПРИМЕЧАНИЕ Тип `string` в C# фактически является классом. Дело в том, что для строки не существует стандартного размера (различные строки могут содержать разное количество символов) и динамическое выделение памяти под строку в ходе выполнения программы работает гораздо эффективнее статического выделения в ходе компиляции. Описание ссылочных типов, таких как классы, приведенное в этой главе, применимо также к типу `string`. Фактически ключевое слово `string` в C# является псевдонимом класса `System.String`.

Рассмотрим ситуацию объявления переменной по имени `i` с типом значения `int` и присваивания ей значения 42. Если объявить еще одну переменную по имени `copyi` с типом значения `int`, а затем присвоить переменную `i` переменной `copyi`, то `copyi` будет содержать точно такое же значение, что и переменная `i` (42). Но даже при том что `copyi` и `i` содержат одно и то же значение, это значение 42 содержит два блока памяти: один для `i`, другой для `copyi`. Если вы измените значение `i`, значение `copyi` не изменится. Давайте посмотрим, как выглядит соответствующий код:

```
int i = 42;      // объявление и инициализация i
int copyi = i; /* copyi содержит копию данных, имеющихся в i:
                 как i, так и copyi содержит значение 42 */
i++;           /* увеличение i на единицу не влияет на copyi;
                 i теперь содержит 43, а copyi – по-прежнему 42 */
```

Эффект, получаемый от объявления переменной `c` в качестве типа класса, такого как `Circle`, совершенно иной. При объявлении `c` в качестве `Circle`-переменной `c` может ссылаться на `Circle`-объект; фактическим значением, содержащимся в `c`, является адрес `Circle`-объекта в памяти. Если объявить еще одну переменную по имени `refc` (также в качестве `Circle`-переменной) и присвоить ей значение переменной `c`, в `refc` будет содержаться копия точно такого же адреса, что и в `c`.

Иными словами, будет существовать только один `Circle`-объект и теперь на него будут ссылаться обе переменные, как `refc`, так и `c`. Соответствующий пример кода выглядит следующим образом:

```
Circle c = new Circle(42);
Circle refc = c;
```

Оба примера показаны на рис. 8.1. Знак «эт» (@) в `Circle`-объектах обозначает ссылку, в которой содержится адрес в памяти.

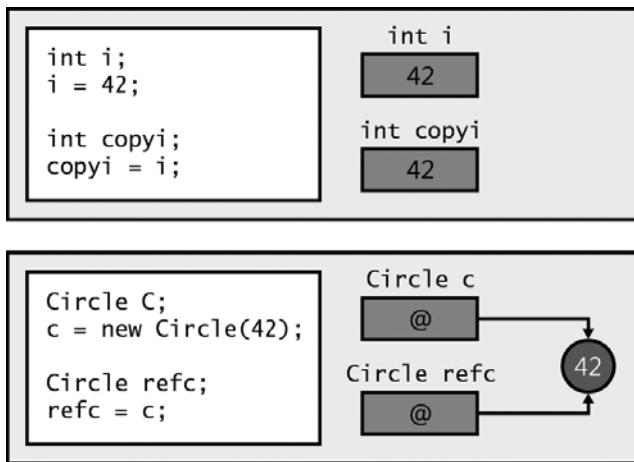


Рис. 8.1

Это различие играет весьма важную роль. В частности, оно означает, что поведение параметров метода зависит от того, к каким типам они относятся: типам значений или ссылочным типам. Это различие будет изучено при выполнении следующего упражнения.

КОПИРОВАНИЕ ССЫЛОЧНЫХ ТИПОВ И ЗАКРЫТОСТЬ ДАННЫХ

Если нужно скопировать содержимое `Circle`-объекта по имени `c` в другой `Circle`-объект по имени `refc`, то вместо простого копирования ссылки нужно сделать так, чтобы объект `refc` ссылался на новый экземпляр класса `Circle`, а затем скопировать данные из `c` в `refc`, причем отдельно скопировать каждое поле:

```
Circle refc = new Circle();
refc.radius = c.radius; // Не пытайтесь это делать
```

Но если какие-либо элементы класса `Circle` являются закрытыми (такими, как поле `radius`), то скопировать их будет невозможно. В качестве альтернативы можно сделать данные в закрытых полях доступными, выставив их в качестве

свойств, а затем воспользоваться этими свойствами для чтения данных из объекта по имени с в объект по имени `refc`. Как это делается, будет показано в главе 15 «Реализация свойств для доступа к полям».

Кроме этого, класс может предоставить метод `Clone`, возвращающий еще один экземпляр того же класса, но наполненный точно такими же данными. Метод `Clone` будет иметь доступ к закрытым данным внутри объекта и может копировать эти данные непосредственно в другой экземпляр того же класса. Например, метод `Clone` для класса `Circle` может быть определен следующим образом:

```
class Circle
{
    private int radius;
    // Конструкторы и другие методы опущены
    ...
    public Circle Clone()
    {
        // Создание нового Circle-объекта
        Circle clone = new Circle();
        // копирование закрытых данных из этого объекта в клон
        clone.radius = this.radius;
        // Возвращение нового Circle-объекта, содержащего скопированные
        // данные
        return clone;
    }
}
```

Этот подход не вызывает затруднений, если все закрытые данные состоят из значений, но если одно или несколько полей сами по себе являются ссылочными типами (например, класс `Circle` может быть расширен, чтобы в нем мог содержаться `Point`-объект из главы 7, указывающий позицию окружности `Circle` на графическом изображении), этим ссылочным объектам также необходимо предоставить метод `Clone`, в противном случае метод `Clone` класса `Circle` просто скопирует ссылку на эти поля. Этот процесс известен как создание углубленной копии. Иной подход, при котором метод `Clone` просто копирует ссылки, известен как создание поверхностной копии.

Предыдущий пример кода вызывает также весьма интересный вопрос: насколько же закрыты закрытые данные? Ранее вы уже видели, что ключевое слово `private` делает поле или метод недоступными за пределами класса. Но это не значит, что к нему можно получить доступ из одного-единственного объекта. Если создать два объекта одного и того же класса, каждый из них может обращаться к данным, закрытым внутри кода для этого класса. Как бы странно это ни звучало, но факт остается фактом: работа таких методов, как `Clone`, зависит от этого свойства. Инструкция `clone.radius = this.radius;` работает только потому, что закрытое поле `radius` в объекте `clone` доступно из текущего экземпляра класса `Circle`. Следовательно, закрытость (`private`) означает «собственность класса», а не «собственность объекта». Но не нужно путать `private` со `static`. Если просто объявить поле закрытым (`private`), то каждый экземпляр класса получит свои собственные данные. Если поле объявлено статическим (`static`), то каждый экземпляр класса использует одни и те же данные совместно с другими экземплярами этого же класса.

Использование параметров-значений и параметров-ссылок

Откройте в Microsoft Visual Studio 2015 проект Parameters, который находится в папке \Microsoft Press\VCBS\Chapter 8\Parameters вашей папки документов. Проект содержит три файла с кодом на C#: Pass.cs, Program.cs и WrappedInt.cs.

Выведите в окно редактора файл Pass.cs. В нем определен класс по имени Pass, который пока что не содержит ничего, кроме комментария // TODO:.



СОВЕТ Не забудьте, что для обнаружения всех имеющихся в решении комментариев TODO можно воспользоваться окном Список задач.

Вместо комментария // TODO: добавьте к классу Pass открытый статический метод по имени Value. Этот метод должен получить один int-параметр (с типом значения) по имени param и иметь возвращаемый тип void. Тело метода Value, выделенное в следующем примере кода жирным шрифтом, будет просто присваивать значение 42 переменной param:

```
namespace Parameters
{
    class Pass
    {
        public static void Value(int param)
        {
            param = 42;
        }
    }
}
```



ПРИМЕЧАНИЕ Чтобы упражнение не усложнилось, этот метод определен с использованием ключевого слова static. Метод Value можно вызвать непосредственно в отношении класса Pass без предварительного создания нового Pass-объекта. Принципы, проиллюстрированные в этом упражнении, применяются точно так же и к методам экземпляров.

Выведите в окно редактора файл Program.cs, а затем найдите метод doWork класса Program. Метод doWork вызывается методом Main, когда программа начинает работу. Как говорилось в главе 7, вызов метода заключен в блок try, за которым следует обработчик исключения.

Добавьте к методу doWork четыре инструкции, выполняющие следующие задачи.

- Обявление локальной int-переменной по имени i и ее инициализация нулевым значением.
- Запись значения переменной i в консоль с помощью метода Console.WriteLine.

- Вызов `Pass.Value` с передачей `i` в качестве аргумента.
- Повторная запись значения `i` в консоль.

Благодаря вызовам `Console.WriteLine` до и после вызова `Pass.Value` вы сможете увидеть, действительно ли вызов `Pass.Value` изменяет значение переменной `i`. Окончательно метод `doWork` должен приобрести следующий вид:

```
static void doWork()
{
    int i = 0;
    Console.WriteLine(i);
    Pass.Value(i);
    Console.WriteLine(i);
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запуск программы. Убедитесь, что значение 0 записано в консоль дважды. Инструкция присваивания внутри метода `Pass.Value`, обновляющая параметр и устанавливающая для него значение 42, использует копию переданного аргумента, а на исходный аргумент `i` это абсолютно не влияет.

Нажмите Ввод и закройте приложение.

А теперь посмотрите, что произойдет, когда передается `int`-параметр, заключенный в классе.

Выполните в окне редактора файл `WrappedInt.cs`. В нем содержится класс `WrappedInt`, в котором нет ничего, кроме комментария `// TODO:`.

Добавьте к классу `WrappedInt` выделенное в следующем примере кода жирным шрифтом открытое поле экземпляра с именем `Number`, имеющее тип `int`:

```
namespace Parameters
{
    class WrappedInt
    {
        public int Number;
    }
}
```

Выполните в окне редактора файл `Pass.cs`. Добавьте к классу `Pass` открытый статический метод по имени `Reference`. Этот метод должен принимать единственный `WrappedInt`-параметр по имени `param` и иметь возвращаемый тип `void`. В теле метода `Reference` значение 42 должно присваиваться `param.Number`:

```
public static void Reference(WrappedInt param)
{
    param.Number = 42;
}
```

Выполните в окно редактора файл Program.cs. Закомментируйте существующий код в методе `dowork` и добавьте четыре инструкции, выполняющие следующие задачи.

- ❑ Объявление локальной `WrappedInt`-переменной по имени `wi` и инициализация ее новым `WrappedInt`-объектом путем вызова пассивного конструктора.
- ❑ Запись значения `wi.Number` в консоль.
- ❑ Вызов метода `Pass.Reference` с передачей `wi` в качестве аргумента.
- ❑ Повторная запись значения `wi.Number` в консоль.

Как и прежде, при вызове метода `Console.WriteLine` вы сможете увидеть, изменяется ли значение `wi.Number` вызов `Pass.Reference`. Теперь метод `dowork` должен приобрести следующий вид (новые инструкции выделены жирным шрифтом):

```
static void dowork()
{
    // int i = 0;
    // Console.WriteLine(i);
    // Pass.Value(i);
    // Console.WriteLine(i);

    WrappedInt wi = new WrappedInt();
    Console.WriteLine(wi.Number);
    Pass.Reference(wi);
    Console.WriteLine(wi.Number);
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запустить приложения. На этот раз два значения, отображаемые в окне консоли, соответствуют значению `wi.Number` до и после вызова метода `Pass.Reference`. Вы должны увидеть, что на экране отображаются значения 0 и 42.

Нажмите Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

Давайте разберемся с тем, что показывает предыдущее упражнение. Значением `wi.Number` при инициализации, осуществляющей создаваемым компилятором пассивным конструктором, становится нуль. В переменной `wi` содержится ссылка на только что созданный `WrappedInt`-объект, содержащий `int`-значение. Затем переменная `wi` копируется в качестве аргумента в метод `Pass.Reference`. Поскольку `WrappedInt` является классом (ссылочный тип), и `wi` и `param` ссылаются на один и тот же `WrappedInt`-объект. Любые изменения, вносимые в содержимое объекта посредством переменной `param` в методе `Pass.Reference`, видны вследствие использования переменной `wi`, когда метод завершает работу. На следующей схеме показано, что происходит, когда `WrappedInt`-объект передается методу `Pass.Reference` в качестве аргумента (рис. 8.2).

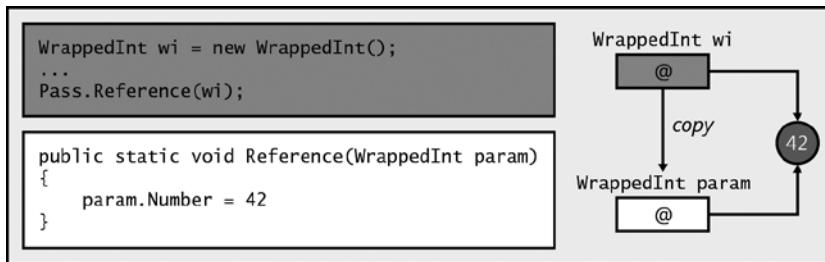


Рис. 8.2

Основные сведения о пустых значениях и о типах, допускающих их использование

Когда объявляется переменная, ее лучше всего сразу же инициализировать. При использовании типов значений наиболее часто встречается следующий код:

```
int i = 0;
double d = 0.0;
```

Вспомним, что для инициализации такой ссылочной переменной, как класс, можно создать новый экземпляр класса и присвоить ссылочной переменной новый объект:

```
Circle c = new Circle(42);
```

Все вроде бы хорошо, но что делать, если вам не нужно создавать новый объект? Возможно, переменная предназначается для простого сохранения ссылки на существующий объект в каком-нибудь другом месте программы. В следующем примере кода инициализируется копия `Circle`-переменной, но затем ей присваивается ссылка на другой экземпляр класса `Circle`:

```
Circle c = new Circle(42);
Circle copy = new Circle(99); // произвольное значение для инициализации копии
...
copy = c; // переменные copy и c ссылаются на один и тот же объект
```

А что же случилось с исходным `Circle`-объектом со значением поля `radius`, равным 99, которое использовалось для инициализации `copy` после того, как переменной `copy` была присвоена переменная `c`? Теперь на него больше ничего не ссылается. В такой ситуации среда выполнения может освободить используемую им память, выполнив операцию, известную как сборка мусора, которая более подробно будет рассмотрена в главе 14 «Использование сборщика мусора»

и управление ресурсами». А пока нам важно уяснить, что сборка мусора является потенциально затратной по времени операцией, и во избежание пустой траты времени и ресурсов вы не должны создавать объекты, которые никогда не будут использоваться.

Можно согласиться с тем, что если переменной в каком-то месте программы собираются присвоить ссылку на другой объект, то нет никакого смысла ее инициализировать. Но применять подобные приемы программирования не стоит, поскольку это может привести к проблемам в коде. Например, вы неизбежно окажетесь в такой ситуации, когда нужно будет, чтобы переменная ссылалась на объект только в том случае, если, как показано в следующем примере кода, эта переменная еще не содержит ссылку:

```
Circle c = new Circle(42);
Circle copy; // Неинициализированная переменная !!!
...
if (copy ==    // выполнить для переменной copy операцию присваивания только в том
    // случае, если эта переменная не инициализирована, но что тогда
    // должно быть на этом месте?
{
    copy = c; // переменные copy и c ссылаются на один и тот же объект
    ...
}
```

Инструкция `if` предназначена для проверки переменной `copy` на предмет того, инициализирована она или нет, но с каким значением следует сравнивать эту переменную? Ответ будет следующим: со специальным значением, которое называется пустым (`null`-значением).

В C# присваивать пустое значение `null` можно любой ссылочной переменной. Значение `null` просто означает, что переменная не ссылается на объект в памяти. Его можно использовать следующим образом:

```
Circle c = new Circle(42);
Circle copy = null; // Инициализирована
...
if (copy == null)
{
    copy = c;        // переменные copy и c ссылаются на один и тот же объект
    ...
}
```

ОПЕРАТОРЫ ПРОВЕРКИ НА NULL-ЗНАЧЕНИЕ

В самую последнюю версию C# включен новый оператор проверки на `null`-значение, позволяющий использовать более краткую запись. Чтобы воспользоваться этим оператором, к имени переменной следует добавить вопросительный знак (?).

Предположим, к примеру, что вы пытаетесь вызвать метод Area в отношении Circle-объекта, когда этот объект имеет значение null:

```
Circle c = null;  
Console.WriteLine($"The area of circle c is {c.Area()}");
```

В этом случае метод Circle.Area выдаст исключение NullReferenceException, что вполне резонно, поскольку вычислить площадь несуществующего круга нельзя.

Чтобы избежать такого исключения, перед попыткой вызова метода Circle.Area можно протестировать наличие у Circle-объекта значения null:

```
if (c != null)  
{  
    Console.WriteLine($"The area of circle c is {c.Area()}");  
}
```

В данном случае, если переменная c имеет значение null, в окно командной строки ничего выводиться не будет. Перед попыткой вызова метода Circle.Area в отношении Circle-объекта можно также воспользоваться оператором проверки на null-значение:

```
Console.WriteLine($"The area of circle c is {c?.Area()}");
```

Этот оператор заставляет среду выполнения проигнорировать текущую инструкцию, если переменная, к которой применяется оператор, имеет значение null. В данном случае в окне командной строки будет выведен следующий текст:

```
The area of circle c is
```

Применимы оба этих подхода, и они могут отвечать вашим потребностям при различных сценариях. Оператор проверки на null-значение может помочь сохранить краткость кода при работе со сложными свойствами, имеющими вложенные ссылочные типы, у которых могут быть null-значения.

Использование типов, допускающих пустые значения

Пустое значение хорошо подходит для инициализации ссылочных типов. Для типов значений требуется эквивалентное значение, а значение null само по себе является ссылкой, поэтому присваивать его типу значения нельзя. Из-за этого недопустимо применять в C# следующую инструкцию:

```
int i = null; // недопустимо
```

И тем не менее в C# определен модификатор, которым можно воспользоваться для объявления принадлежности переменной к типу значений, допускающих пустое значение. Тип значений, допускающий пустое значение, ведет себя точно так же, как и исходный тип значений, но переменной этого типа можно присвоить значение null. Для обозначения того, что тип значений допускает использование пустых значений, применяется вопросительный знак (?):

```
int? i = null; // допустимо
```

Выяснить, содержит ли переменная, относящаяся к этому типу, значение `null`, можно путем тестирования этой переменной аналогично тестированию переменной ссылочного типа:

```
if (i == null)  
    ...
```

Вы можете присвоить выражение соответствующего типа значений переменной, допускающей пустое значение. Все следующие примеры кода являются вполне допустимыми:

```
int? i = null;  
int j = 99;  
i = 100; // Копирование константы в переменную типа, допускающего пустые значения  
i = j; // Копирование переменной в переменную типа, допускающего пустые значения
```

Следует заметить, что обратное утверждение будет неверно. Переменную, допускающую пустое значение, нельзя присвоить переменной обычного типа значений. То есть если взять определения переменных `i` и `j` из предыдущего примера, то следующая инструкция будет недопустимой:

```
j = i; // недопустимо
```

И это вполне резонно, если учесть, что переменная `i` может содержать `null`-значение, а переменная `j` относится к типу значений, которые не могут быть пустыми. Это также означает, что переменную, допускающую пустое значение, нельзя использовать в качестве параметра метода, ожидающего значение, относящееся к обычному типу значений. Если вспомнить, что метод `Pass.Value` из предыдущего упражнения ожидал обычный `int`-параметр, то следующий вызов метода откомпилирован не будет:

```
int? i = 99;  
Pass.Value(i); // Ошибка в ходе компиляции
```



ПРИМЕЧАНИЕ Не следует путать обозначение типов значений, допускающих пустые значения, с оператором проверки на `null`-значение. Типы, допускающие пустые значения, обозначаются добавлением вопросительного знака к имени типа, а оператор проверки на `null`-значение добавляется к имени переменной.

Основные сведения о свойствах типов, допускающих пустые значения

Типы значений, допускающие пустые значения, предоставляют два свойства, которыми можно воспользоваться для определения наличия непустого значения и того, каким это значение является. Свойство `HasValue` показывает, содержит

ли переменная, допускающая пустое значение, именно его, или у нее есть непустое значение. А непустое значение из переменной, относящейся к типу, допускающему пустые значения, можно извлечь путем считывания свойства `Value`:

```
int? i = null;
...
if (!i.HasValue)
{
    // Если переменная i содержит пустое значение, присвоить ей значение 99
    i = 99;
}
else
{
    // Если она не содержит пустое значение, вывести ее значение на экран
    Console.WriteLine(i.Value);
}
```

В главе 4 «Использование инструкций принятия решений» утверждалось, что оператор НЕ (!) инвертирует булево значение. В данном фрагменте кода производится тестирование переменной `i`, относящейся к типу значений, допускающему пустые значения, и если в ней нет значения (если она пустая), ей присваивается значение 99, в противном случае на экран выводится значение этой переменной. В этом примере использование свойства `HasValue` не дает каких-либо преимуществ по сравнению с непосредственным тестированием на пустое значение. Кроме того, считывание значения свойства `Value` — слишком многословный способ чтения содержимого переменной. Но эти очевидные недостатки обусловлены тем, что `int?` является очень простым типом, допускающим пустые значения. Преимущества использования свойств `HasValue` и `Value` проявятся, скорее всего, при создании более сложных типов значений, которые можно будет использовать для объявления переменных, допускающих пустые значения. Соответствующие примеры можно будет увидеть в главе 9 «Создание типов значений с использованием перечислений и структур».



ПРИМЕЧАНИЕ Свойство `Value` типа значений, допускающего пустые значения, предназначено только для чтения. Им можно воспользоваться для считывания значения переменной, но не для его изменения. Для изменения значения переменной, допускающей пустые значения, используется обычная инструкция присваивания.

Использование параметров `ref` и `out`

Обычно, когда методу передается аргумент, его копией инициализируется соответствующий параметр. Это утверждение справедливо независимо от того, к какому типу значений относится параметр — обычному (например, `int`), или допускающему пустые значения (например, `int?`), или же ссылочному (например, `WrappedInt`). Это обстоятельство означает, что никакие изменения параметра

не повлияют на значение переданного ему аргумента. Например, в следующем коде значением, выводимым на консоль, является 42, а не 43. Метод `doIncrement` выполняет операцию инкремента копии аргумента (`arg`), а не его оригинала:

```
static void doIncrement(int param)
{
    param++;
}
static void Main()
{
    int arg = 42;
    doIncrement(arg);
    Console.WriteLine(arg); // записывается 42, а не 43
}
```

В предыдущем упражнении было показано, что если параметр метода относится к ссылочному типу, то любые изменения с использованием этого параметра приводят к изменениям данных, на которые ссылался переданный методу аргумент. Ключевым моментом здесь является следующее: хотя данные, на которые делалась ссылка, изменились, аргумент, переданный в качестве параметра, изменений не претерпел — он по-прежнему ссылается на тот же самый объект. Иными словами, хотя через параметр можно изменить объект, на который ссылается аргумент, сам аргумент изменить нельзя (например, перенастроить его таким образом, чтобы он ссылался на совершенно другой объект). В большинстве случаев такая гарантия весьма полезна и способна помочь в уменьшении количества ошибок в программе. Но временами может понадобиться создать метод, которому действительно нужно изменить аргумент. Для этих целей в C# предоставляются ключевые слова `ref` и `out`.

Создание `ref`-параметров

Если перед параметром поставить ключевое слово `ref`, компилятор C# создаст код, передающий ссылку на сам аргумент, а не на его копию. При использовании `ref`-параметра все, что делается с параметром, делается и с исходным аргументом, поскольку и параметр и аргумент ссылаются на одни и те же данные. При передаче аргумента в качестве `ref`-параметра впереди него также нужно поставить ключевое слово `ref`. Этот синтаксис является для программиста полезным визуальным признаком того, что аргумент может измениться. Вот еще одна версия предыдущего примера, в которой на этот раз используется ключевое слово `ref`:

```
static void doIncrement(ref int param) // используется ref
{
    param++;
}
static void Main()
```

```
{
    int arg = 42;
    doIncrement(ref arg); // используется ref
    Console.WriteLine(arg); // записывается 43
}
```

Теперь метод `doIncrement` получает ссылку на оригинальный аргумент, а не на его копию, поэтому любые изменения, вносимые этим методом, при использовании данной ссылки фактически изменяют оригинальное значение. Потому-то на консоль и выводится значение 43.

Не забывайте, что в C# действует правило, согласно которому значение переменной должно быть присвоено до того, как вы сможете его прочитать. Это правило применяется и к аргументам метода: методу нельзя передать в качестве аргумента переменную, не прошедшую инициализацию, даже если аргумент определен в качестве `ref`-аргумента. Например, в следующем примере переменная `arg` не прошла инициализацию, поэтому код не пройдет компиляцию. Этот сбой произойдет по причине того, что инструкция `param++`; внутри метода `doIncrement` на самом деле является псевдонимом инструкции `arg++`;, а эта операция допустима, только если у `arg` имеется вполне определенное значение:

```
static void doIncrement(ref int param)
{
    param++;
}

static void Main()
{
    int arg; // переменная не инициализирована
    doIncrement(ref arg);
    Console.WriteLine(arg);
}
```

Создание out-параметров

Компилятор перед вызовом метода проверяет, присвоено ли `ref`-параметру значение. Но может возникнуть ситуация, при которой инициализацию параметра нужно будет возложить на сам метод. Это можно сделать с использованием ключевого слова `out`.

Синтаксически это ключевое слово похоже на `ref`. Ключевое слово `out` можно поставить перед параметром, превратив его в псевдоним для аргумента. Как и при использовании `ref`, все, что делается с параметром, делается и с оригинальным аргументом. Когда аргумент передается `out`-параметру, перед этим аргументом также следует поставить `out`.

Ключевое слово `out` является сокращением от слова `output` (вывод). Когда методу передается `out`-параметр, то, как показано в следующем примере, метод должен присвоить ему значение до завершения своей работы или до возвращения управления коду, вызвавшему этот метод:

```
static void doInitialize(out int param)
{
    param = 42; // Инициализация param до завершения работы
}
```

Следующий пример кода не проходит компиляцию, потому что `doInitialize` не присваивает `param` никакого значения:

```
static void doInitialize(out int param)
{
    // Ничего не делается
}
```

Поскольку метод должен присвоить `out`-параметру какое-либо значение, этот метод может быть вызван без инициализации его аргумента. Например, в следующем коде `doInitialize` вызывается для инициализации переменной `arg`, значение которой затем выводится на консоль:

```
static void doInitialize(out int param)
{
    param = 42;
}
static void Main()
{
    int arg;           // переменная не инициализирована
    doInitialize(out arg); // вполне допустимо
    Console.WriteLine(arg); // записывается 42
}
```

Работа `ref`-параметров будет изучена в следующем упражнении.

Использование `ref`-параметров

Вернитесь в среду Visual Studio 2015 к проекту `Parameters`. Выберите в окне редактора файл `Pass.cs` и отредактируйте метод `Value`, чтобы он принимал `ref`-параметр.

Метод `Value` должен выглядеть следующим образом:

```
class Pass
{
    public static void Value(ref int param)
    {
        param = 42;
    }
    ...
}
```

Выполните в окно редактора файл Program.cs. Уберите символы комментария из первых четырех инструкций. Обратите внимание на то, что в третьей инструкции метода doWork — Pass.Value(i); — выявляется ошибка. Дело в том, что теперь метод Value ожидает ref-параметр. Отредактируйте эту инструкцию, чтобы при вызове метода Pass.Value аргумент ему передавался в виде ref-параметра.



ПРИМЕЧАНИЕ Четыре инструкции, создающие и тестирующие объект WrappedInt, оставьте в неизменном виде.

Теперь метод doWork должен выглядеть так:

```
class Program
{
    static void doWork()
    {
        int i = 0;
        Console.WriteLine(i);
        Pass.Value(ref i);
        Console.WriteLine(i);
        ...
    }
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запустить программы. На этот раз первыми двумя значениями, записанными в окно консоли, будут 0 и 42. Этот результат показывает, что вызов метода Pass.Value успешно изменил аргумент i.

Нажмите Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.



ПРИМЕЧАНИЕ Модификаторы ref и out можно использовать и для параметров ссылочного типа. Эффект будет точно таким же — параметр превратится в псевдоним для аргумента.

Как организована память компьютера

Компьютеры используют память для хранения выполняемых программ и используемых ими данных. Чтобы понять, чем отличаются друг от друга типы значений и ссылочные типы, полезно будет разобраться с организацией данных в памяти.

Операционные системы и среда выполнения, которая используется в языке C#, зачастую делят память, применяемую для хранения данных, на две отдельные области, управляемые по-разному. Эти две области памяти традиционно

называются стеком и динамической памятью (кучей). Они служат разным целям.

- ❑ При вызове метода память, требующаяся для хранения его параметров и локальных переменных, всегда берется из стека. Когда работа метода завершается (либо по причине возвращения из него, либо из-за выдачи исключения), память, занятая в стеке параметрами и локальными переменными, автоматически высвобождается и снова становится доступной при вызове другого метода. Параметры метода и локальные переменные в стеке имеют вполне определенный период существования: они появляются при запуске метода и исчезают, как только метод завершает свою работу.



ПРИМЕЧАНИЕ Фактически такой же период существования и у переменных, определяемых в любом блоке кода, находящемся внутри открывающей и закрывающей фигурных скобок. В следующем примере кода переменная *i* создается при запуске на выполнение тела цикла `while` и исчезает, когда цикл `while` завершается, и выполнение кода продолжается уже после закрывающей фигурной скобки:

```
while (...)  
{  
    int i = ...; // здесь i создается в стеке  
    ...  
}  
// а здесь i исчезает из стека
```

- ❑ Когда с помощью ключевого слова `new` создается объект (экземпляр класса), память, необходимая для его создания, всегда берется из динамической области (кучи). Вы уже видели, что с помощью ссылочных переменных ссылаться на один и тот же объект можно из разных мест. Когда исчезает последняя ссылка на объект, память, используемая под этот объект, снова становится доступной (хотя, возможно, и не сразу). Более подробное описание процесса высвобождения динамической памяти дается в главе 14. Поэтому у объектов, созданных в динамической памяти, менее определенный период существования: объект создается с помощью ключевого слова `new`, а исчезает только после удаления на него последней ссылки.



ПРИМЕЧАНИЕ Все переменные типов значений создаются в стеке. Все переменные ссылочных типов (объекты) создаются в динамической памяти (хотя сами ссылки находятся в стеке). Типы, допускающие пустые значения, фактически являются ссылочными типами, и переменные этих типов создаются в динамической памяти.

Названия «стек» и «куча» произошли от способа управления памятью со стороны среды выполнения.

- ❑ Стековая память организована наподобие стопки коробок, стоящих друг на друге. При вызове метода каждый параметр помещается в коробку, которая ставится на вершину стопки. Каждой локальной переменной как бы

назначается своя собственная коробка, помещаемая поверх тех коробок, которые уже находятся в стопке. Когда работа метода завершается, можно представить все дело так, что коробки удаляются из стопки.

- ❑ Динамическая память похожа на кучу коробок, разбросанных по всей комнате, а не стоящих друг на друге. У каждой коробки имеется надпись, указывающая на то, используется она или нет. При создании нового объекта среда выполнения ищет пустую коробку и выделяет ее объекту. Ссылка на объект сохраняется в локальной переменной в стеке. Среда выполнения отслеживает количество ссылок на каждую коробку. (Вспомним, что две переменные могут ссылаться на один и тот же объект.) Когда исчезает последняя ссылка, среда выполнения помечает коробку как неиспользуемую, в некий момент в будущем опустошает ее и открывает к ней доступ.

Использование стека и кучи

Давайте посмотрим, что произойдет при вызове метода по имени **Method**:

```
void Method(int param)
{
    Circle c;
    c = new Circle(param);
    ...
}
```

Предположим, что аргумент, переданный в **param**, имеет значение 42. При вызове метода из стека выделяется блок памяти, достаточный для хранения **int**-значения, который инициализируется значением 42. Как только выполнение кода перемещается в метод, из стека выделяется еще один блок памяти, достаточно большой для хранения ссылки (адреса памяти), но он остается без инициализации. Этот блок памяти предназначен для **Circle**-переменной по имени **c**. Затем из кучи выделяется еще один фрагмент памяти, размер которого достаточен для хранения объекта **Circle**. Это делается при использовании ключевого слова **new**. Для превращения этого ничем не заполненного участка динамической памяти в объект **Circle** вызывается **Circle**-конструктор. Ссылка на этот объект **Circle** сохраняется в переменной **c**. Данную ситуацию может проиллюстрировать рис. 8.3.

На данный момент нужно отметить два обстоятельства.

- ❑ Хотя объект сохранен в куче, ссылка на объект (переменная **c**) хранится в стеке.
- ❑ Динамическая память (куча) небезгранична. Если она будет исчерпана, оператор **new** выдаст исключение недостатка памяти — **OutOfMemoryException** и объект создан не будет.

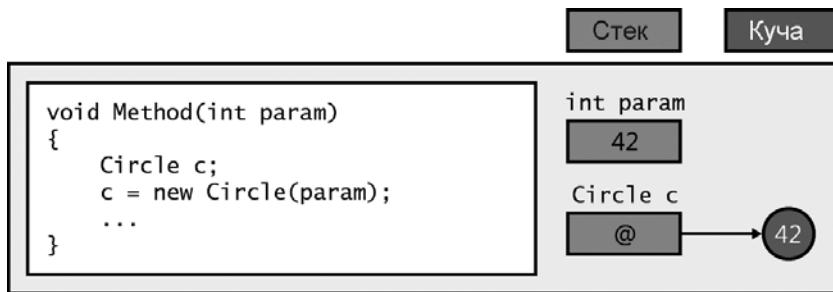


Рис. 8.3



ПРИМЕЧАНИЕ Circle-конструктор также может выдать исключение. Если это произойдет, память, выделенная объекту `Circle`, будет возвращена в оборот и конструктор вернет пустое значение (`null`).

Когда метод завершит свою работу, параметры и локальные переменные выйдут из области видимости. Память, полученная для `c` и `param`, автоматически высвободится для дальнейшего использования в стеке. Среда выполнения замечает, что на объект `Circle` больше нет ссылок, и в какой-то момент в будущем помечает выделенную ему память как предназначенную для возвращения в оборот в куче (см. главу 14).

Класс System.Object

Одним из наиболее важных ссылочных типов в среде .NET Framework является класс `Object`, относящийся к пространству имен `System`. Чтобы в полной мере оценить важность класса `System.Object`, нужно разобраться с таким понятием, как наследование, которое рассматривается в главе 12 «Работа с наследованием». А пока просто примите на веру, что все классы являются специализированными типами `System.Object` и что `System.Object` можно использовать для создания переменной, ссылающейся на любой ссылочный тип. Важность класса `System.Object` настолько высока, что в C# в качестве его псевдонима предоставляется ключевое слово `object`. В вашем коде можно использовать как `object`, так и `System.Object` — они имеют одинаковое значение.



СОВЕТ Использование ключевого слова `object` более предпочтительно, чем `System.Object`. Оно имеет более конкретный смысл и хорошо сочетается с другими ключевыми словами, являющимися синонимами для классов (например, со `string` для `System.String` и с рядом других ключевых слов, рассматриваемых в главе 9).

В следующем примере переменные `c` и `o` ссылаются на один и тот же объект `Circle`. Тот факт, что типом `c` является `Circle`, а типом `o` является `object`

(псевдоним для `System.Object`), в действительности дает две разные точки зрения на один и тот же элемент в памяти:

```
Circle c;
c = new Circle(42);
object o;
o = c;
```

На следующей схеме (рис. 8.4) показано, как переменные `c` и `o` ссылаются на один и тот же элемент в динамической памяти (куче).

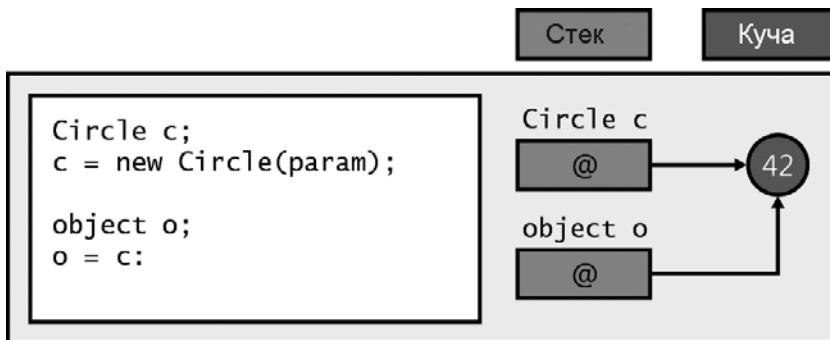


Рис. 8.4

Упаковка

Как вы только что видели, переменные типа `object` могут ссылаться на любой элемент любого ссылочного типа. Но переменные типа `object` могут ссылаться и на типы значений. Например, следующие две инструкции инициализируют переменную `i` (типа `int`, который относится к типам значений) значением 42, а затем инициализируют переменную `o` (типа `object`, который относится к ссылочным типам) значением `i`:

```
int i = 42;
object o = i;
```

Чтобы разобраться в том, что происходит на самом деле, вторая инструкция требует небольшого пояснения. Вспомним, что `i` относится к типу значений, в силу чего находится в стеке. Если ссылка внутри переменной `o` ссылается непосредственно на `i`, то она будет ссылаться на стек. Но все ссылки должны ссылаться на объекты в динамической памяти: создание ссылок на элементы в стеке может создать серьезную угрозу устойчивости среды выполнения и открыть потенциальную брешь в системе безопасности, следовательно, это недопустимо. Поэтому среда выполнения выделяет участок динамической памяти, копирует в него значение

целочисленной переменной `i`, а затем делает `object` о ссылкой на эту копию. Такое автоматическое копирование элемента из стека в кучу называется упаковкой (boxing). Результат показан на следующей схеме (рис. 8.5).

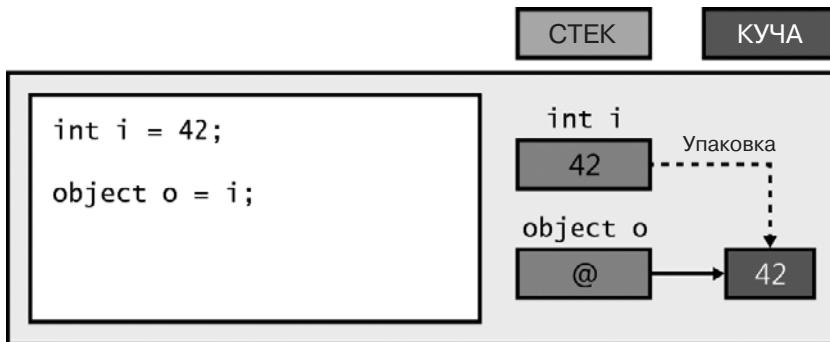


Рис. 8.5



ВНИМАНИЕ Если изменить оригинальное значение переменной `i`, значение в динамической памяти, на которое делается ссылка посредством переменной `o`, не изменится. Точно так же, если изменить значение в динамической памяти, оригинальное значение переменной не изменится.

Распаковка

Поскольку переменная типа `object` может ссылаться на упакованную копию значения, то смысл этого состоит всего лишь в том, чтобы позволить вам получить это упакованное значение через переменную. Вполне ожидаемо иметь возможность обращаться к упакованному `int`-значению, на которое ссылается переменная `o`, используя обычную инструкцию присваивания:

```
int i = o;
```

Но при попытке использования такого синтаксиса в ходе компиляции будет выдана ошибка. Если вдуматься, то невозможность использования синтаксиса `int i = o;` поддается разумному объяснению. В конечном счете `o` может ссылаться на что угодно, а не только на `int`-значение. Посмотрим, что бы получилось в следующем коде, будь такая инструкция разрешена:

```
Circle c = new Circle();
int i = 42;
object o;

o = c; // o ссылается на circle-переменную
i = o; // так что тогда сохраняется в i?
```

Чтобы получить значение из упакованной копии, нужно воспользоваться приведением типов (cast). Это операция, проверяющая перед фактическим созданием копии безопасность превращения элемента одного типа в элемент другого типа. При этом перед переменной типа `object` нужно поставить название типа в круглых скобках:

```
int i = 42;
object o = i; // упаковка
i = (int)o;   // компиляция проходит успешно
```

При таком приведении типов проделывается довольно тонкая работа. Компилятор видит, что в приведении указан тип `int`. Затем он создает код, проверяющий, на что фактически ссылается `o` в процессе выполнения программы. Это может быть что угодно. То, что приведение типов предписывает `o` ссылаться на `int`, еще не означает, что эта переменная ссылается на значение именно этого типа. Если `o` действительно ссылается на упакованное `int`-значение и все совпадает, приведение выполняется успешно и созданный компилятором код извлекает значение из `int`-упаковки и копирует его в `i`. (В данном примере упакованное значение затем хранится в `i`.) Эта операция называется распаковкой. Все происходящее проиллюстрировано на следующей схеме (рис. 8.6).

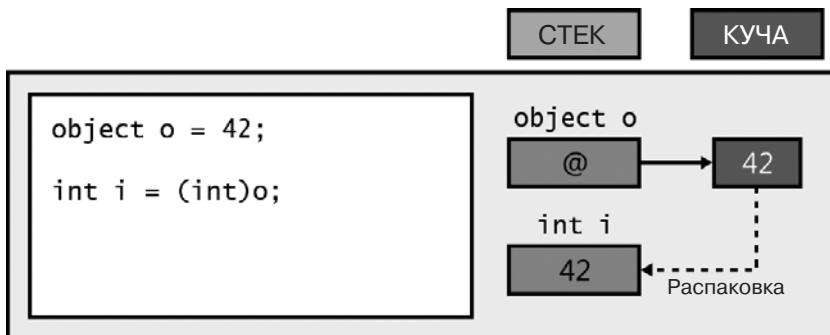
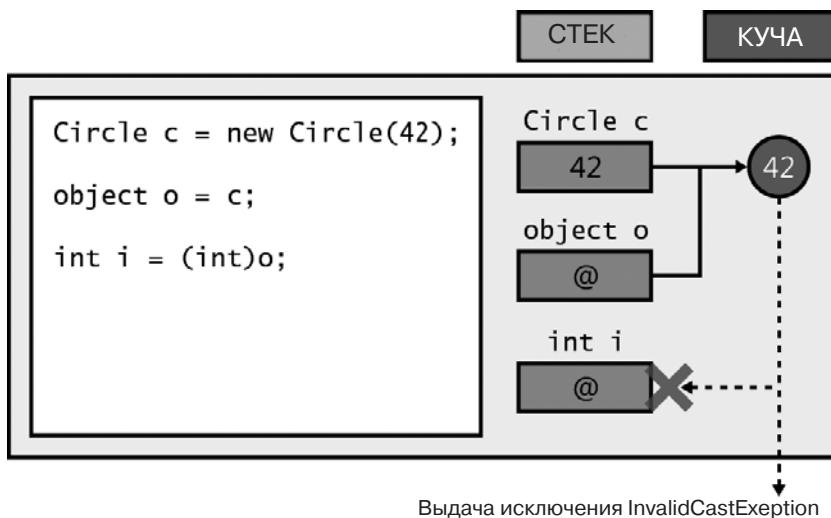


Рис. 8.6

В то же время, если `o` не ссылается на упакованное `int`-значение, возникает несоответствие типов, не позволяющее успешно выполнить приведение типа. Созданный компилятором код в ходе этого выдает исключение неверного приведения — `InvalidCastException`. Пример неудачного приведения типа показан в следующем фрагменте кода:

```
Circle c = new Circle(42);
object o = c;    // упаковка не происходит, поскольку Circle – ссылочная переменная
int i = (int)o;  // компиляция проходит, но во время выполнения кода выдается
                 // исключение
```

Происходящее проиллюстрировано на рис. 8.7.

**Рис. 8.7**

Упаковка и распаковка будут применяться в следующих упражнениях. Нужно иметь в виду, что упаковка и распаковка относятся к ресурсоемким операциям, поскольку требуют определенного объема проверок и нуждаются в выделении дополнительных объемов динамической памяти. Упаковка, конечно, находит применение, но ее необдуманное использование может существенно снизить производительность программы. Альтернатива упаковке будет показана в главе 17 «Введение в обобщения».

Безопасное приведение типов данных

Используя приведение типов, можно указать, что, по вашему мнению, данные, на которые ссылается объект, относятся к определенному типу и ссылаться на объект с использованием данного типа вполне безопасно. Ключевым здесь является уточнение «по вашему мнению». Компилятор C# не станет проверять, так ли это на самом деле, а вот среда выполнения обязательно проверит. Если тип объекта в памяти не соответствует приведению типа, среда выполнения, как говорилось в предыдущем разделе, выдаст исключение `InvalidCastException`. Вы должны быть готовы перехватить это исключение и соответствующим образом его обработать.

Но перехват исключений и попытка исправления ситуации в случае, когда тип объекта не соответствует ожиданиям, являются весьма неуклюжим подходом. C# предоставляет два очень полезных оператора, `is` и `as`, помогающих выполнить приведение типов более элегантно.

Оператор `is`

Оператор `is` можно использовать для проверки факта принадлежности объекта к тому или иному ожидаемому вами типу:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
if (o is WrappedInt)
{
    WrappedInt temp = (WrappedInt)o; // Операция безопасна; o имеет тип WrappedInt
    ...
}
```

Этот оператор использует два операнда: ссылку на объект, указываемую по левую сторону от него, и название типа, указываемое по правую сторону. Если тип объекта в динамической памяти, на который осуществляется ссылка, соответствует указанному, выражение вычисляется в `true`, а если нет — в `false`. Предыдущий код пытается осуществить приведение типа в отношении ссылки на объект, содержащейся в переменной `o`, только в том случае, если заранее станет известно, что эта операция пройдет успешно.

Оператор `as`

Оператор `as` выполняет такую же роль, что и `is`, но в несколько усеченном виде. Этот оператор используется следующим образом:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
WrappedInt temp = o as WrappedInt;
if (temp != null)
{
    ... // приведение типа прошло успешно
}
```

Подобно оператору `is`, оператор `as` использует операнды, представляющие собой объект и тип. Исполняющая среда пытается привести объект к указанному типу. Если приведение проходит успешно, возвращается его результат, в данном примере он присваивается `WrappedInt`-переменной по имени `temp`. Если выполнить приведение не удается, оператор вычисляется в `null` и переменной `temp` присваивается пустое значение.

Что касается операторов `is` и `as`, то можно привести их развернутое описание, что и будет сделано в главе 12.

УКАЗАТЕЛИ И НЕБЕЗОПАСНЫЙ КОД

Эта врезка является просто информацией к размышлению и предназначена для разработчиков, знакомых с С или С++. Новички в программировании могут ее пропустить.

Если вам уже приходилось создавать программы на языках С и С++, то с темой ссылок на объекты вы, скорее всего, уже знакомы, поскольку в обоих языках имеется конструкция, предоставляющая такую же функциональную возможность. Речь идет об указателе.

Указатель представляет собой переменную, в которой содержится адрес элемента в памяти или ссылка на этот элемент (неважно где, в куче или в стеке). Для обозначения переменной в качестве указателя используется специальный синтаксис. Например, следующая инструкция объявляет переменную *pi* в качестве указателя на целочисленное значение:

```
int *pi;
```

Хотя переменная *pi* объявлена в качестве указателя, фактически до своей инициализации она ни на что не указывает. Например, чтобы использовать *pi* в качестве указателя на целочисленную переменную *i*, можно воспользоваться следующими инструкциями и оператором получения адреса, *address-of*, (&), который возвращает адрес переменной:

```
int *pi;
int i = 99;
...
pi = &i;
```

У вас есть возможность через указатель в переменной *pi* обратиться к значению переменной *i* и изменить его:

```
*pi = 100;
```

Этот код изменяет значение переменной *i* на 100, поскольку *pi* указывает на то же самое место в памяти, что и переменная *i*.

Одной из основных проблем, с которыми сталкиваются разработчики, изучающие С и С++, является понимание синтаксиса, используемого указателями. Оператор *** имеет как минимум два значения (кроме того что он является оператором арифметического умножения), и зачастую возникает путаница при попытке определить, что именно нужно использовать, оператор *&* или оператор ***. Еще одна проблема указателей состоит в том, что они легко могут указать на что-то недопустимое или можно вообще забыть о конкретном нацеливании указателя, а затем пытаться сослаться на данные, на которые он якобы указывает. В результате произойдет либо засорение памяти, либо программный сбой с выдачей ошибки, поскольку операционная система обнаружит попытку доступа к запрещенному адресу памяти. Кроме того, во многих существующих системах неправильное управление указателями становится причиной возникновения целого ряда изъянов безопасности: некоторые среды (но не Windows) не в состоянии принуждать к проверке, относящейся к ссылке указателя на память, принадлежащую другому процессу, открывая тем самым возможность несанкционированного доступа к конфиденциальным данным.

Во избежание всех этих проблем в C# и были введены ссылочные переменные. Если есть желание, то можно продолжать использование указателей и в C#, но код при этом следует помечать как небезопасный. Для того чтобы поставить на блок кода или на весь метод метку, указывающую на его небезопасность, следует воспользоваться ключевым словом unsafe:

```
public static void Main(string [] args)
{
int x = 99, y = 100;
unsafe
{
swap (&x, &y);
}
Console.WriteLine($"x is now {x}, y is now {y}");
}
public static unsafe void swap(int *a, int *b)
{
int temp;
temp = *a;
*a = *b;
*b = temp;
}
```

При компиляции программ, содержащих небезопасный код, следует при сборке проекта установить флагок Разрешить небезопасный код. Для этого нужно щелкнуть правой кнопкой мыши на проекте в окне обозревателя решений, а затем щелкнуть на пункте Свойства. В окне свойств нужно щелкнуть на вкладке Сборка, установить флагок Разрешить небезопасный код, а затем в меню Файл щелкнуть на пункте Сохранить все.

Небезопасный код имеет отношение также к управлению памятью. Объекты, созданные в небезопасном коде, называются неуправляемыми. Хотя ситуации, требующие получения доступа к памяти подобным образом, возникают нечасто, с некоторыми из них вам, возможно, придется столкнуться, особенно если вы создаете код, требующий выполнения некоторых низкоуровневых операций Windows.

Более подробно о последствиях использования кода, обращающегося к неуправляемой памяти, будет рассказано в главе 14.

Выводы

В этой главе вы узнали о ряде важных отличий типов значений, содержащих свои значения непосредственно в стеке, от ссылочных типов, опосредованно ссылающихся на свои объекты в динамической памяти. Вы также научились с целью доступа к аргументам использовать в отношении параметров методов ключевые слова `ref` и `out`. Вы увидели, как присваивание значения (например, `int`-значения 42) переменной класса `System.Object` создает упакованную копию

значения в динамической памяти, а затем заставляет переменную `System.Object` ссылаться на эту упакованную копию. Вы также увидели, как присваивание переменной типа значений (например, `int`-переменной) из переменной класса `System.Object` приводит к копированию (или распаковке) значения в классе `System.Object` в память, используемую `int`-переменной.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 9.

Если хотите выйти из среды Visual Studio 2015, то в меню Файл щелкните на пункте Выход. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке Да и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Скопировать переменную типа значений	Просто создайте копию. Поскольку переменная относится к типу значений, у вас будут две копии одного и того же значения, например: <code>int i = 42;</code> <code>int copyi = i;</code>
Скопировать переменную ссылочного типа	Просто создайте копию. Поскольку переменная относится к ссылочному типу, у вас будут две ссылки на один и тот же объект, например: <code>Circle c = new Circle(42);</code> <code>Circle refc = c;</code>
Объявить переменную, способную содержать тип значений или пустое значение	Объявите переменную с использованием модификатора <code>? </code> с указанием типа, например: <code>int? i = null;</code>
Передать аргумент <code>ref</code> -параметру	Поставьте перед параметром ключевое слово <code>ref</code> . Оно превратит параметр в псевдоним для аргумента, а не в копию этого аргумента. Метод может изменить значение параметра, и это изменение будет применено к самому аргументу, а не к его локальной копии, например: <code>static void Main()</code> <code>{</code> <code> int arg = 42;</code> <code> doWork(ref arg);</code> <code> Console.WriteLine(arg);</code> <code>}</code>

Чтобы	Сделайте следующее
Передать аргумент out-параметру	<p>Поставьте перед параметром ключевое слово <code>out</code>. Оно превратит параметр в псевдоним для аргумента, а не в копию этого аргумента. Метод должен присвоить значение параметру, и это значение станет значением самого аргумента, например:</p> <pre>static void Main() { int arg; doWork(out arg); Console.WriteLine(arg); }</pre>
Упаковать значение	<p>Инициализируйте переменную типа <code>object</code> этим значением или присвойте ей это значение, например:</p> <pre>object o = 42;</pre>
Распаковать значение	<p>Выполните операцию приведения типа для ссылки на объект, которая ссылается на упакованное значение, к типу значения переменной, например:</p> <pre>int i = (int)o;</pre>
Провести безопасное приведение типов	<p>Воспользуйтесь оператором <code>is</code> для тестирования допустимости приведения типа, например:</p> <pre>WrappedInt wi = new WrappedInt(); ... object o = wi; if (o is WrappedInt) { WrappedInt temp = (WrappedInt)o; ... }</pre> <p>Или же воспользуйтесь оператором <code>as</code> для приведения типа и тестирования результата на null-значение, например:</p> <pre>WrappedInt wi = new WrappedInt(); ... object o = wi; WrappedInt temp = o as WrappedInt; if (temp != null) ...</pre>

9

Создание типов значений с использованием перечислений и структур

Прочитав эту главу, вы научитесь:

- объявлять перечисляемый тип;
- создавать и использовать перечисляемый тип;
- объявлять структурный тип;
- создавать и использовать структурный тип;
- объяснять разницу в поведении между структурой и классом.

В главе 8 «Основные сведения о значениях и ссылках» было дано описание двух базовых типов, имеющихся в Microsoft Visual C#: типов значений и ссылочных типов. Вспомним, что переменные типа значения содержат свои значения непосредственно в стеке, а переменные ссылочного типа содержат ссылки на объект, который находится в динамической памяти (куче). В главе 7 «Создание классов и объектов и управление ими» было показано, как путем определения классов создаются собственные ссылочные типы. В этой главе вы изучите способы создания собственных типов значений.

В C# поддерживаются два вида типов значений: перечисления и структуры. Рассмотрим их по очереди.

Работа с перечислениями

Предположим, вам нужно представить в программе времена года. Для представления весны, лета, осени и зимы можно воспользоваться целыми числами 0, 1, 2 и 3 соответственно. Такая система будет работать, но вряд ли можно назвать

ее интуитивно понятной. Если в коде используется целочисленное значение 0, то совсем не очевидно, что какое-то конкретное нулевое значение представляет именно весну. Надежность такого решения вызывает большие сомнения. Например, если объявляется переменная по имени `season`, то ничто не помешает назначить ей любое допустимое целочисленное значение за пределами набора 0, 1, 2 или 3. Язык C# предлагает более подходящее решение. Вы можете создать перечисление (иногда называемое `enum`-типов), чьи значения ограничиваются набором символьных имен.

Объявление перечисления

Перечисление определяется с помощью ключевого слова `enum`, за которым следует набор заключенных в фигурные скобки символов, идентифицирующих допустимые для типа значения. Вот как объявляется перечисление по имени `Season`, чьи литеральные значения ограничиваются символьными именами `Spring`, `Summer`, `Fall` и `Winter`:

```
enum Season { Spring, Summer, Fall, Winter }
```

Использование перечисления

После объявления перечисления им можно воспользоваться точно так же, как и любым другим типом. Если перечисление носит имя `Season`, можно, как показано в следующем примере, создать переменные типа `Season`, поля типа `Season` и параметры типа `Season`:

```
enum Season { Spring, Summer, Fall, Winter }

class Example
{
    public void Method(Season parameter) // пример параметра метода
    {
        Season localVariable;           // пример локальной переменной
        ...
    }
    private Season currentSeason;      // пример поля
}
```

Перед тем как получить возможность считывания значения переменной перечисления, ей это значение должно быть присвоено. Значение, определенное как перечисление, можно присвоить только переменной перечисления:

```
Season colorful = Season.Fall;
Console.WriteLine(colorful); // записывается 'Fall'
```



ПРИМЕЧАНИЕ Как и в случае со всеми другими типами значений, воспользовавшись модификатором в виде вопросительного знака (?), вы можете создать версию переменной перечисления, допускающую пустое значение. Затем переменной можно будет присвоить значение null — точно так же, как и значения, определяемые перечислением:

```
Season? colorful = null;
```

Обратите внимание на необходимость записи `Season.Fall`, а не просто `Fall`. Все лiteralные имена перечисления находятся в области видимости своего перечисляемого типа, что позволяет различным перечислениям содержать лiteralы с одинаковыми именами.

Также обратите внимание на то, что при выводе переменной перечисления на экран с помощью метода `Console.WriteLine` компилятор создает код, записывающий имя лiteralа, чье значение совпадает со значением переменной. При необходимости можно непосредственно преобразовать переменную перечисления в строку, представляющую ее текущее значение, путем использования встроенного метода `ToString`, автоматически содержащегося во всех перечислениях:

```
string name = colorful.ToString();
Console.WriteLine(name); // также записывает 'Fall'
```

В отношении переменных перечислений (кроме побитовых операторов и операторов сдвига, которые будут рассматриваться в главе 16 «Использование индексаторов») могут использоваться и многие стандартные операторы, пригодные для применения в отношении целочисленных переменных. Например, определить равенство двух перечислений одного и того же типа можно с помощью оператора (`==`), а в отношении переменных перечислений можно даже выполнять арифметические операции, хотя определенный практический смысл у результата бывает не всегда!

Выбор лiteralных значений перечислений

В своем внутреннем механизме перечисляемый тип ассоциирует с каждым элементом перечисления целочисленное значение. По умолчанию нумерация начинается с нуля для первого элемента и возрастает на единицу с каждым следующим элементом. Это исходное целочисленное значение переменной перечисления можно извлечь. Для этого нужно привести это значение к его основному типу. При рассмотрении механизма распаковки в главе 8 говорилось, что при приведении типа данные преобразуются из одного типа в другой при условии, что это преобразование допустимо и имеет какой-либо смысл. В следующем примере кода записывается не слово `Fall`, а значение 2 (вспомним, что в `Season`-перечислении `Spring` — это 0, `Summer` — 1, `Fall` — 2 и `Winter` — 3):

```
enum Season { Spring, Summer, Fall, Winter }
...
Season colorful = Season.Fall;
Console.WriteLine((int)colorful); // записывается '2'
```

При необходимости с литералом перечисления (например, со `Spring`) можно ассоциировать конкретную целочисленную константу (например, 1):

```
enum Season { Spring = 1, Summer, Fall, Winter }
```



ВНИМАНИЕ Целочисленное значение, которым инициализируется литерал перечисления, должно быть на момент компиляции значением константы (таким, как 1).

Если литералу перечислений постоянное целочисленное значение явным образом не задается, компилятор дает ему значение на единицу больше значения предыдущего литерала перечисления, за исключением самого первого литерала перечисления, которому компилятор дает исходное значение 0. Применительно к предыдущему примеру основными целочисленными значениями для `Spring`, `Summer`, `Fall` и `Winter` теперь являются 1, 2, 3 и 4.

Разрешается одному и тому же основному значению задавать более одного литерала перечисления. Например, в Великобритании время года `fall` (осень) называют `autumn`. Угодить обеим вариантам английского можно следующим образом:

```
enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }
```

Выбор основного типа перечислений

При объявлении перечисления литералам перечисления задаются значения типа `int`. Можно также выбрать за основу перечисления другой базовый целочисленный тип. Например, чтобы объявить основной тип для `Season` не `int`, а `short`, можно использовать следующий код:

```
enum Season : short { Spring, Summer, Fall, Winter }
```

Основной смысл использования `short` заключается в экономии памяти: `int` занимает больше памяти, чем `short`, и если вы не нуждаетесь в полном диапазоне значений, доступных в `int`, практичеснее будет воспользоваться типом данных с более скромными запросами.

Заложить основу перечисления можно с помощью любого из восьми целочисленных типов: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`. Значения всех литералов перечисления должны входить в диапазон избранного основного типа. Например, если за основу перечисления выбрать тип данных `byte`, у вас может быть максимум 256 литералов (начиная с нуля).

Теперь, после того как вы узнали о способах объявления перечислений, настала пора перейти к их использованию. В следующем упражнении вам предлагается работа с консольным приложением, объявляющим и использующим перечисление, представляющее месяцы года.

Создание и использование перечисления

Откройте в среде Microsoft Visual Studio 2015 проект **StructsAndEnums**, который находится в папке **\Microsoft Press\VCSBS\Chapter 9\StructsAndEnums** вашей папки документов.

Выполните в окне редактора файл Month.cs. В исходном файле нет ничего, кроме объявления пространства имен **StructsAndEnums** и комментария **// TODO:**.

Замените комментарий **// TODO:** выделенным в следующем примере кода жирным шрифтом перечислением по имени Month, находящемся в пространстве имен **StructsAndEnums**. Это перечисление моделирует месяцы года. В качестве 12 литералов перечисления Month фигурируют названия месяцев от January до December.

```
namespace StructsAndEnums
{
    enum Month
    {
        January, February, March, April,
        May, June, July, August,
        September, October, November, December
    }
}
```

Выполните в окне редактора файл Program.cs. Метод Main, как и в упражнениях из предыдущих глав, вызывает метод doWork и перехватывает любые возникающие исключения.

Добавьте в окне редактора к методу doWork еще одну инструкцию, объявляющую Month-переменную по имени first, и инициализируйте ее значением Month. January. Добавьте еще одну инструкцию для записи значения переменной first в консоль. Метод doWork должен приобрести следующий вид:

```
static void doWork()
{
    Month first = Month.January;
    Console.WriteLine(first);
}
```



ПРИМЕЧАНИЕ Как только после Month будет набрана точка, среда Microsoft IntelliSense автоматически выведет все значения, имеющиеся в перечислении Month.

Щелкните в меню Отладка на пункте Запуск без отладки. Среда Visual Studio 2015 выполнит сборку и запуск программы. Убедитесь в том, что в консоль будет записано слово *January*.

Нажмите Ввод, чтобы закрыть программу и вернуться в среду программирования Visual Studio 2015.

Добавьте к методу *doWork* еще две инструкции (в следующем примере кода выделены жирным шрифтом), предназначенные для выполнения операции инкремента переменной *first* и вывода ее нового значения на консоль:

```
static void doWork()
{
    Month first = Month.January;
    Console.WriteLine(first);
    first++;
    Console.WriteLine(first);
}
```

Щелкните в меню Отладка на пункте Запуск без отладки. Среда Visual Studio 2015 произведет сборку и запуск программы. Убедитесь в том, что в консоль записаны слова *January* и *February*.

Заметьте, что выполнение математической операции, такой как инкремент, в отношении переменной перечисления изменяет внутреннее целочисленное значение переменной. Когда переменная записывается в консоль, на экран выводится соответствующее значение перечисления. Нажмите Ввод, чтобы закрыть программу и вернуться в среду программирования Visual Studio 2015.

Измените, как показано жирным шрифтом в следующем примере, первую инструкцию в методе *doWork*, чтобы инициализировать переменную *first* значением *Month.December*:

```
static void doWork()
{
    Month first = Month.December;
    Console.WriteLine(first);
    first++;
    Console.WriteLine(first);
}
```

Щелкните в меню Отладка на пункте Запуск без отладки. Среда Visual Studio 2015 произведет сборку и запуск программы. На этот раз в консоль будет записано слово *December*, за которым будет следовать число 12 (рис. 9.1).

Выполнять арифметические операции в отношении перечисления вполне допустимо, но если



Рис. 9.1

результат операции находится вне диапазона значений, определенных для перечисления, то все, что может сделать среда выполнения, — это интерпретировать значение переменной как соответствующее целочисленное значение. Нажмите Ввод, чтобы закрыть программу и вернуться в среду программирования Visual Studio 2015.

Работа со структурами

В главе 8 было показано, что классы определяют ссылочные типы, которые всегда создаются в динамической памяти. В некоторых случаях класс может содержать настолько малые по объему данные, что издержки на управление динамической памятью становятся неоправданно высокими. В таких случаях лучше определить тип как структуру. Структура относится к типу значений. Поскольку структура хранится в стеке, то, пока она невелика, издержки на управление памятью зачастую снижаются.

Подобно классу, у структуры могут быть свои собственные поля, методы и (за важным исключением, рассматриваемым далее) конструкторы.

ОБЩИЕ ТИПЫ СТРУКТУР

Хотя вы этого и не осознавали, в предыдущих примерах книги структуры уже использовались. В C# такие простые числовые типы, как int, long и float, являются псевдонимами структур System.Int32, System.Int64 и System.Single соответственно. У этих структур имеются поля и методы, и эти методы могут вызываться в отношении переменных и литералов этих типов. Например, все эти структуры предоставляют метод ToString, проводящий преобразование числового значения в его строковое представление. В C# вполне допустимы все следующие инструкции:

```
int i = 55;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());
float f = 98.765F;
Console.WriteLine(f.ToString());
Console.WriteLine(98.765F.ToString());
```

Слишком часто видеть такое использование метода ToString не приходилось, поскольку метод Console.WriteLine вызывает его автоматически по мере необходимости. Чаще используются предоставляемые этими структурами статические методы. Например, в предыдущих главах использовался метод int.Parse, занимающийся преобразованием строки в соответствующее ей целочисленное значение. Фактически при этом вызывался метод Parse структуры Int32:

```
string s = "42";
int i = int.Parse(s); // абсолютно то же самое, что и Int32.Parse
```

Эти структуры включают также ряд полезных статических полей. Например, Int32.MaxValue является максимальным значением, которое может содержаться в int-переменной, а Int32.MinValue является соответственно минимально возможным значением такой переменной.

В следующей таблице показаны простые типы, имеющиеся в C#, и эквивалентные им типы в среде Microsoft .NET Framework. Обратите внимание на то, что типы string и object являются классами (ссыльочными типами), а не структурами.

Ключевое слово	Эквивалент типа	Класс или структура
bool	System.Boolean	Структура
byte	System.Byte	Структура
decimal	System.Decimal	Структура
double	System.Double	Структура
float	System.Single	Структура
int	System.Int32	Структура
long	System.Int64	Структура
object	System.Object	Класс
sbyte	System.SByte	Структура
short	System.Int16	Структура
string	System.String	Класс
uint	System.UInt32	Структура
ulong	System.UInt64	Структура
ushort	System.UInt16	Структура

Объявление структуры

Для объявления своей собственной структуры используется ключевое слово **struct**, за которым следуют имя типа и заключенное в фигурные скобки тело структуры. Синтаксически процесс похож на объявление класса. Вот как, к примеру, выглядит структура по имени **Time**, содержащая три открытых **int**-поля с именами **hours**, **minutes** и **seconds**:

```
struct Time
{
    public int hours, minutes, seconds;
}
```

Как и при объявлении классов, в большинстве случаев указывать поля открытymi нежелательно, поскольку управлять значениями, содержащимися в открытых полях, невозможно. Например, установить для `minutes` или `seconds` значение, превышающее 60, может кто угодно. Лучше сделать поля закрытыми и обеспечить структуру конструкторами и методами для инициализации этих полей и управления ими:

```
struct Time
{
    private int hours, minutes, seconds;
    ...
    public Time(int hh, int mm, int ss)
    {
        this.hours = hh % 24;
        this.minutes = mm % 60;
        this.seconds = ss % 60;
    }

    public int Hours()
    {
        return this.hours;
    }
}
```



ПРИМЕЧАНИЕ Автоматически использовать многие типовые операторы в ваших собственных структурах не получится. Например, к переменным, относящимся к вашей собственной структуре, невозможно применять операторы равенства (`==`) и неравенства (`!=`). Но для сравнения переменных типа структуры можно использовать встроенный метод `Equals()`, предоставляемый всеми структурами, также можно для типов своей собственной структуры объявить явным образом и реализовать операторы. Используемый при этом синтаксис рассматривается в главе 21 «Запрос данных, находящихся в памяти, с помощью выражений в виде запросов».

При копировании переменной типа значения вы получаете две копии значения. В отличие от этого при копировании переменной ссылочного типа вы получаете две ссылки на один и тот же объект. Таким образом, структуры следует использовать для значений, имеющих небольшой объем данных, для которых копирование значения имеет практически такой же эффект, как и копирование адреса. А классы следует использовать для более сложных данных, слишком больших для эффективного копирования.



СОВЕТ Используйте структуры для реализации простых понятий, чьей основной характеристикой является их значение, а не предоставляемые ими функциональные возможности.

Основные сведения о том, чем структуры отличаются от классов

Структуры и классы синтаксически похожи друг на друга, но у них есть ряд важных различий. Давайте рассмотрим некоторые из них.

Для структуры невозможно объявить пассивный конструктор (конструктор без параметров). Следующий пример был бы скомпилирован, если бы `Time` был классом, но он не пройдет компиляцию, потому что `Time` — это структура:

```
struct Time
{
    public Time() { ... } // ошибка в ходе компиляции
    ...
}
```

Причина, по которой вы не можете объявить для структуры свой собственный пассивный конструктор, заключается в том, что компилятор *всегда* создает такой конструктор самостоятельно. В классе компилятор создает пассивный конструктор только в том случае, если вы не создали его сами. Созданный компилятором пассивный конструктор для структуры всегда устанавливает для полей значения `0`, `false` или `null`. Он делает это точно так же, как и для класса. Поэтому нужно убедиться в том, что значение структуры, созданное пассивным конструктором, ведет себя логично и имеет с этими исходными значениями вполне определенный смысл. Имеющиеся разновидности будут исследованы в следующем упражнении.

Предоставляя активный конструктор, поля можно будет инициализировать различными значениями. Но при этом ваш активный конструктор должен инициализировать все имеющиеся в структуре поля явным образом — инициализации по умолчанию больше происходить не будет. Если этого не сделать, то в ходе компиляции будет выдана ошибка. К примеру, если бы следующее объявление относилось к классу, то оно было бы откомпилировано, а поля были бы по умолчанию проинициализированы, но поскольку `Time` является структурой, компиляция даст сбой:

```
struct Time
{
    private int hours, minutes, seconds;
    ...
    public Time(int hh, int mm)
    {
        this.hours = hh;
        this.minutes = mm;
    }      // ошибка в ходе компиляции: поле seconds не проинициализировано
}
```

В классе можно инициализировать поля экземпляра в тех же местах, где они объявляются. В структуре этого делать нельзя. Следующий пример был бы откомпилирован, если бы объявление `Time` относилось к классу, но в ходе компиляции возникнет ошибка, поскольку `Time` объявляется структурой:

```
struct Time
{
    private int hours = 0; // ошибка в ходе компиляции
    private int minutes;
    private int seconds;
    ...
}
```

Основные отличия структур от классов сведены в табл. 9.1.

Таблица 9.1

Вопрос	Структура	Класс
Это тип значения или ссылочный тип?	Структура относится к типу значения	Класс относится к ссылочному типу
Где размещается экземпляр, в стеке или в динамической памяти?	Экземпляры структуры называются значениями и размещаются в стеке	Экземпляры классов называются объектами и размещаются в динамической памяти
Можно ли объявить пассивный конструктор?	Нет	Да
Если вы объявили собственный конструктор, будет ли компилятор непременно создавать пассивный конструктор?	Да	Нет
Если вы в собственном конструкторе не проинициализируете поле, проинициализирует ли компилятор это поле автоматически?	Нет	Да
Разрешено ли выполнять инициализацию полей экземпляра в месте их объявления?	Нет	Да

Есть и другие отличия классов от структур, касающиеся наследования. Они будут рассмотрены в главе 12 «Работа с наследованием».

Объявление переменных структуры

После определения типа структуры им можно воспользоваться абсолютно так же, как и любым другим типом. Например, если была определена структура `Time`, можно создать переменные, поля и параметры типа `Time`:

```

struct Time
{
    private int hours, minutes, seconds;
    ...
}
class Example
{
    private Time currentTime;

    public void Method(Time parameter)
    {
        Time localVariable;
        ...
    }
}

```



ПРИМЕЧАНИЕ Как и в случае с перечислениями, можно создать версию структурной переменной, допускающую пустые значения, воспользовавшись для этого модификатором в виде вопросительного знака (?). После чего переменной можно будет присвоить пустое значение.

```
Time? currentTime = null;
```

Представление об инициализации структуры

Ранее в данной главе было показано, что поля в структуре можно проинициализировать с помощью конструктора. При вызове конструктора ранее описанные правила гарантируют, что все поля в структуре будут проинициализированы:

```
Time now = new Time();
```

Состояние полей в этой структуре показано на рис. 9.2.

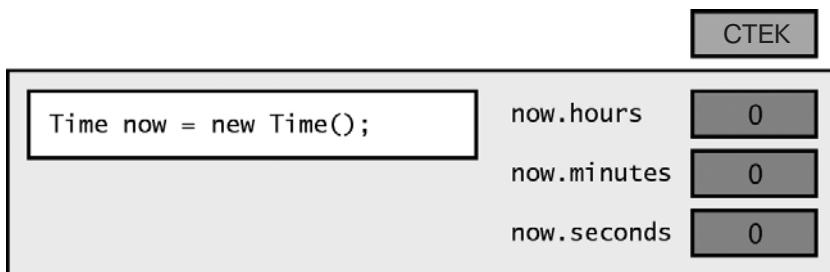


Рис. 9.2

Но поскольку структуры относятся к типам значений, можно также создавать структурные переменные, не вызывая конструктор:

```
Time now;
```

На этот раз переменная создана, но поля остаются в своем не прошедшем инициализацию состоянии. На рис. 9.3 показано состояние полей переменной `now`. Любая попытка обращения к значениям этих полей приведет к ошибке в ходе компиляции.



Рис. 9.3

Заметьте, что в обоих случаях переменная `now` создается в стеке.

Написав свой собственный конструктор структуры, можете использовать его также для инициализации структурной переменной. В соответствии с теми объяснениями, которые уже давались в этой главе, конструктор структуры всегда должен инициализировать все свои поля, например:

```

struct Time
{
    private int hours, minutes, seconds;
    ...
    public Time(int hh, int mm)
    {
        hours = hh;
        minutes = mm;
        seconds = 0;
    }
}
    
```

В следующем примере `now` инициализируется путем вызова конструктора, определенного пользователем:

```
Time now = new Time(12, 30);
```

Результат выполнения этого примера показан на рис. 9.4.

Пора применить эти знания на практике. В следующем упражнении будет создана и использована структура для представления даты.

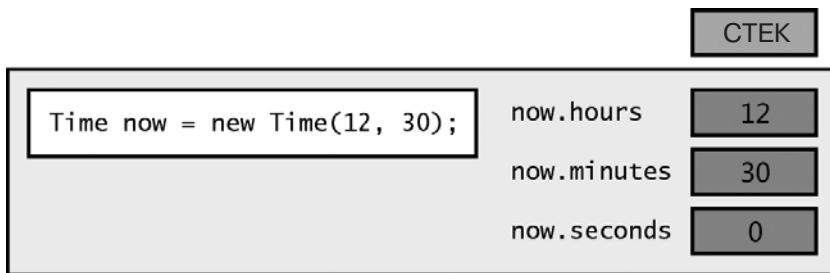


Рис. 9.4

Создание и использование типа структуры

Выполните в окно редактора файл Date.cs, принадлежащий проекту StructsAndEnums. Замените комментарий TODO структурой по имени `Date`, находящейся в пространстве имен `StructsAndEnums`.

В этой структуре должны содержаться три закрытых поля: одно типа `int` с именем `year`, другое типа `Month` с именем `month` (с использованием созданного в предыдущем упражнении перечисления) и третье типа `int` с именем `day`. Структура `Date` должна иметь следующий вид:

```
struct Date
{
    private int year;
    private Month month;
    private int day;
}
```

Рассмотрим пассивный конструктор, который компилятор должен создать для `Date`. Этот конструктор установит для `year` значение `0`, для `month` — значение `0` (это значение соответствует литералу `January`) и для `day` — значение `0`. Значение `0` для `year` не подойдет, поскольку такого года не существует, не подойдет оно и для `day`, так как каждый месяц начинается с первого числа. Одним из способов устранения этой проблемы является перевод значений `year` и `day` путем такой реализации структуры `Date`, при которой поле `year` содержит значение `Y`, которое представляет собой год `Y + 1900` (или, если нужно, можно выбрать другое столетие), а поле `day` содержит значение `D`, представляющее день `D + 1`. Тогда пассивный конструктор назначит трем полям значения, представляющие дату 1 января 1900 года.

Если бы можно было переопределить пассивный конструктор и написать свой собственный, этой проблемы не возникло бы, поскольку затем можно было

бы проинициализировать поля `year` и `day` допустимыми значениями. Но вы не можете сделать этого, поэтому нужно реализовать в вашей структуре логику перевода созданных компилятором исходных значений в значения, имеющие смысл для вашей предметной области.

И хотя вы не можете переопределить пассивный конструктор, все же было бы неплохо определить активный конструктор, позволяющий пользователю инициализировать поля в структуре явным образом, присвоив им значения, отличные от исходных и имеющие вполне определенный смысл.

Добавьте к структуре `Date` открытый конструктор. Этот конструктор должен принимать три параметра: `int`-параметр по имени `ccyy` для поля `year`, `Month`-параметр по имени `mm` для месяца и `int`-параметр по имени `dd` для дня. Используйте эти три параметра для инициализации соответствующих полей. Поле `year` со значением `Y` представляет год `Y + 1900`, поэтому вам нужно инициализировать поле `year` значением `ccyy - 1900`. Поле `day` со значением `D` представляет день `D + 1`, следовательно, нужно инициализировать поле `day` значением `dd - 1`.

Теперь структура `Date` должна выглядеть следующим образом (конструктор выделен жирным шрифтом):

```
struct Date
{
    private int year;
    private Month month;
    private int day;

    public Date(int ccyy, Month mm, int dd)
    {
        this.year = ccyy - 1900;
        this.month = mm;
        this.day = dd - 1;
    }
}
```

Добавьте к структуре `Date` после конструктора открытый метод `ToString`. Этот метод не получает аргументов и возвращает строку, представляющую дату. Вспомним, что значение поля `year` представляет `year + 1900`, а значение поля `day` представляет `day + 1`.

Метод `ToString` должен выглядеть следующим образом:

```
struct Date
{
    ...
    public override string ToString()
    {
        string data = $"{this.month} {this.day + 1} {this.year + 1900}";
        return data;
    }
}
```



ПРИМЕЧАНИЕ Метод `ToString` немного отличается от тех методов, которые вы видели до сих пор. Любой тип, включая определяемые вами структуры и классы, хотите вы того или нет, автоматически уже имеет метод `ToString`. Изначально он преобразует данные, хранящиеся в переменной, в строку, являющуюся представлением этих данных. Иногда это исходное поведение имеет вполне определенный смысл, а иногда смысл особо не прослеживается. Например, исходное поведение метода `ToString`, созданного для структуры `Date`, просто заключается в создании строки «`StructsAndEnums.Date`». По выражению Зафода Библрокса, персонажа книги «Автостопом по галактике», автором которой является Дуглас Адамс, это «проникновенно, но скучновато». Вам нужно с помощью ключевого слова `override` определить новую версию этого метода, переопределяющую исходное поведение. Более подробно перегружаемые методы рассматриваются в главе 12.

В этом методе создается отформатированная строка, использующая текстовое представление значения поля `month`, а также выражения `this.day + 1` и `this.year + 1900`. В качестве результата метод `ToString` возвращает отформатированную строку.

Выполните в окне редактора файл `Program.cs`. Закомментируйте в методе `doWork` четыре имеющиеся в нем инструкции. Добавьте к методу `doWork` инструкции, объявляющие локальную переменную по имени `defaultDate`, и проинициализируйте ее `Date`-значением, сконструированным путем использования пассивного `Date`-конструктора. Добавьте к методу `doWork` еще одну инструкцию, выводящую значение переменной `defaultDate` на консоль с помощью вызова метода `Console.WriteLine`.



ПРИМЕЧАНИЕ Метод `Console.WriteLine` для форматирования своего аргумента в виде строки автоматически вызывает для него метод `ToString`.

Теперь метод `doWork` должен приобрести следующий вид:

```
static void doWork()
{
    ...
    Date defaultDate = new Date();
    Console.WriteLine(defaultDate);
}
```



ПРИМЕЧАНИЕ После набора `new Date()` система IntelliSense автоматически определяет, что типу `Date` доступны два конструктора.

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запустить программы. Убедитесь в том, что на консоль будет записана дата `January 1 1900`.

Нажмите Ввод для возвращения в среду программирования Visual Studio 2015.

Вернитесь в окне редактора к методу `doWork` и добавьте еще две инструкции. В первой инструкции объявите локальную переменную по имени `weddingAnniversary` и инициализируйте ее значением `July 4 2015`. (Я действительно женился в День независимости, хотя это было много лет назад.) Во второй инструкции запишите в консоль значение `weddingAnniversary`.

Теперь метод `doWork` должен выглядеть так:

```
static void doWork()
{
    ...
    Date weddingAnniversary = new Date(2015, Month.July, 4);
    Console.WriteLine(weddingAnniversary);
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, а затем убедитесь в том, что в консоль ниже предыдущей информации записана дата `July 4 2015`. Нажмите Ввод и вернитесь в среду Visual Studio 2015.

Копирование структурных переменных

Одной структурной переменной можно инициализировать другую структурную переменную или же одной из них присвоить значение другой, но только если структурная переменная, указанная справа, полностью инициализирована, то есть если все ее поля заполнены допустимыми данными, а не неопределенными значениями. Следующий пример проходит компиляцию, потому что на данный момент полная инициализация уже проведена:

```
Date now = new Date(2012, Month.March, 19);
Date copy = now;
```

Результат выполнения такого присваивания показан на рис. 9.5.

Следующий пример не пройдет компиляцию, потому что переменная `now` не инициализирована:

```
Date now;
Date copy = now; // ошибка в ходе компиляции: now не присвоено значение
```

При копировании структурной переменной каждое поле той переменной, которая находится слева от оператора присваивания, получает значение непосредственно из соответствующего ему поля переменной, расположенной справа от него. Копирование осуществляется в виде быстрой единой операции, копирующей содержимое всей структуры, и при этом никогда не выдается исключение. Сравните такое поведение с тем, которое наблюдалось бы, будь `Time` классом, при котором обе переменные, `now` и `copy`, в конечном итоге указывали бы на один и тот же объект в динамической памяти.

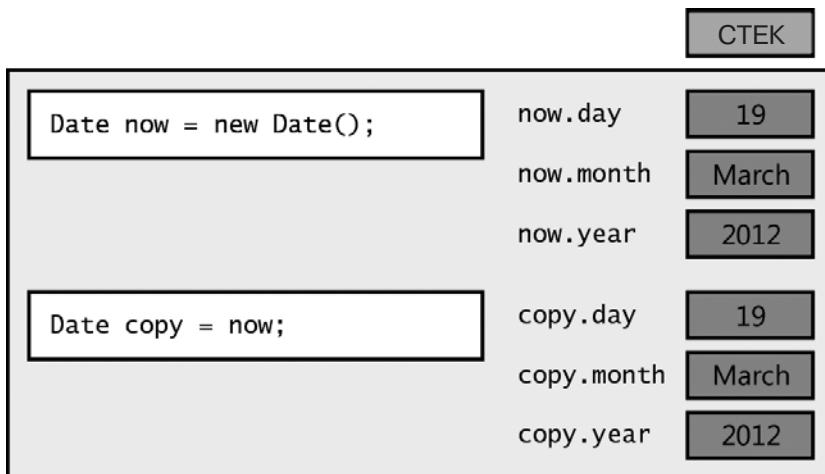


Рис. 9.5



ПРИМЕЧАНИЕ При наличии опыта программирования на C++ вы должны заметить, что поведение при копировании настройке не поддается.

В заключительном упражнении данной главы вы сопоставите поведение при копировании структуры с поведением при копировании класса.

Сравнение поведения структуры и класса

Выполните в окне редактора файл Date.cs, принадлежащий проекту StructsAndEnums. Добавьте к структуре Date следующий метод, увеличивающий дату структуры на один месяц. Если после этого значение поля month выйдет за рамки December, код переключит month на January и увеличит значение поля year на единицу:

```
struct Date
{
    ...
    public void AdvanceMonth()
    {
        this.month++;
        if (this.month == Month.December + 1)
        {
            this.month = Month.January;
            this.year++;
        }
    }
}
```

Выполните в окне редактора файл Program.cs. Закомментируйте в методе doWork первые две незакомментированные инструкции, создающие и выводящие на экран значение переменной defaultDate.

Добавьте следующий код, выделенный далее жирным шрифтом, в конец метода `doWork`. Этот код создает копию переменной `weddingAnniversary` под названием `weddingAnniversaryCopy` и выводит значение новой переменной на экран:

```
static void doWork()
{
    ...
    Date weddingAnniversaryCopy = weddingAnniversary;
    Console.WriteLine($"Value of copy is {weddingAnniversaryCopy}");
}
```

Добавьте следующие выделенные далее жирным шрифтом инструкции в конец метода `doWork`. Эти инструкции вызывают метод `AdvanceMonth` переменной `weddingAnniversary`, а затем выводят значения переменных `weddingAnniversary` и `weddingAnniversaryCopy` на экран:

```
static void doWork()
{
    ...
    weddingAnniversary.AdvanceMonth();
    Console.WriteLine($"New value of weddingAnniversary is
                      {weddingAnniversary}");
    Console.WriteLine($"Value of copy is still {weddingAnniversaryCopy}");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запустить приложения. Убедитесь в том, что в окно консоли выводятся следующие сообщения:

```
July 4 2015
Value of copy is July 4 2015
New value of weddingAnniversary is August 4 2015
Value of copy is still July 4 2015
```

В первом сообщении выводится исходное значение переменной `weddingAnniversary` (`July 4 2015`). Во втором сообщении выводится значение переменной `weddingAnniversaryCopy`. Вы можете увидеть, что оно содержит те же данные, которые хранятся в переменной `weddingAnniversary` (`July 4 2015`). В третьем сообщении выводится значение переменной `weddingAnniversary` после изменения месяца на август (`August 4 2015`). И в последнем сообщении выводится значение переменной `weddingAnniversaryCopy`. Обратите внимание на то, что по сравнению с исходным значением `July 4 2015` оно не изменилось.

Если `Date` объявить классом, то создание копии стало бы ссылкой на тот же объект в памяти, который принадлежал исходному экземпляру. Из-за этого изменение месяца в исходном экземпляре повлечет за собой изменение даты, на которую идет ссылка из копии. Это утверждение вы проверите при выполнении следующих действий.

Нажмите **Ввод** и вернитесь в среду Visual Studio 2015.

Выполните в окно редактора файл Date.cs. Измените структуру **Date**, превратив ее, как показано жирным шрифтом, в класс:

```
class Date
{
    ...
}
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы снова выполнить сборку и запустить приложения. Убедитесь, что в окно консоли выведены следующие сообщения:

```
July 4 2015
Value of copy is July 4 2015
New value of weddingAnniversary is August 4 2015
Value of copy is still August 4 2015
```

Первые три сообщения совпадают с предыдущими. А вот четвертое сообщение показывает, что значение переменной **weddingAnniversaryCopy** изменилось на **August 4 2015**.

Нажмите **Ввод**, чтобы вернуться в среду Visual Studio 2015.

СТРУКТУРЫ И СОВМЕСТИМОСТЬ СО СРЕДОЙ ВЫПОЛНЕНИЯ WINDOWS

Все приложения C# выполняются с использованием общеязыковой среды выполнения (common language runtime (CLR)), относящейся к среде .NET Framework. CLR отвечает за предоставление для кода вашего приложения безопасной во всех отношениях среды в виде виртуальной машины (если у вас есть опыт работы с Java, то эта концепция должна быть вам знакома). При компиляции приложения на C# компилятор преобразует ваш код C# в набор инструкций, используя псевдомашинный код, который называется общим промежуточным языком (Common Intermediate Language (CIL)). Эти инструкции хранятся в сборке. При запуске приложения на C# CLR отвечает за преобразование CIL-инструкций в реальные инструкции машины, понимаемые и выполняемые центральным процессором вашего компьютера. Все это окружение известно под именем управляемой среды выполнения, и программы на C# зачастую называют управляемым кодом. Управляемый код можно создавать и на других языках, поддерживаемых .NET Framework, например Visual Basic и F#.

На Windows 7 и более ранних версий можно было кроме этого создавать неуправляемые приложения, известные также как внутренний код, основанный на Win32 API-интерфейсах, взаимодействующих непосредственно с операционной системой Windows. (Если вы запускаете управляемое приложение, CLR также преобразует многие функции в .NET Framework в вызовы Win32 API, хотя этот процесс совершенно открыт для вашего кода.) Для этого можно воспользоваться

языком C++. Среда .NET Framework позволяет интегрировать управляемый код в неуправляемые приложения посредством набора технологий обеспечения взаимодействия. Подробности работы этих технологий и порядок их использования в данной книге не рассматриваются, и нам вполне достаточно знать, что они не всегда носят простой и понятный характер.

Последние версии Windows предоставляют альтернативную стратегию в виде среды выполнения Windows Runtime, или WinRT. Эта среда является надстройкой над Win32 API и другими избранными исходными API-интерфейсами Windows, предоставляющей однородную функциональность на различных типах оборудования, от серверов до смартфонов. При создании приложения универсальной платформы (Universal Windows Platform (UWP)) используются API-интерфейсы, предоставляемые WinRT, а не Win32. Аналогично этому CLR на Windows 10 также использует WinRT: весь управляемый код, написанный с использованием C# или любого другого управляемого языка, по-прежнему выполняется с помощью CLR, но в ходе выполнения CLR преобразует ваш код в вызовы WinRT API, а не в вызовы Win32. CLR и WinRT совместно отвечают за безопасное управление вашим кодом и его выполнение.

Основным назначением WinRT является упрощение языкового взаимодействия, направленное на облегчение интегрирования компонентов, разработанных с использованием различных языков программирования, в единое цельное приложение. Но за эту простоту приходится платить, и вы должны быть готовы идти на ряд компромиссов, основанных на доступности различных функциональных наборов разных языков. В частности, в силу исторических причин язык C++ хоть и поддерживает структуры, но не распознает принадлежащие им функции. В понятиях C# входящая в состав структуры функция является методом экземпляра. Следовательно, если выполняется сборка структур C#, которые требуется упаковать в библиотеку, сделав их доступными для разработчиков, программирующих на C++ (или на других неуправляемых языках), эти структуры не должны содержать никаких методов экземпляров. Аналогичные ограничения действуют в структурах и по отношению к статическим методам. Если нужно включить методы экземпляров или статические методы, следует преобразовать вашу структуру в класс. Кроме того, структуры не могут содержать закрытые поля, а все открытые поля должны относиться к простым типам C#, соответствующим типам значений или строкам.

WinRT налагает и некоторые другие ограничения на те классы и структуры C#, которые требуется сделать доступными для обычных приложений. Дополнительные сведения по этому вопросу можно будет найти в главе 12.

Выводы

В этой главе вы увидели, как создаются и используются перечисления и структуры. Узнали, в чем сходство и различия структуры и класса, и увидели, как определяются конструкторы для инициализации полей структуры. Вы также увидели, как можно представить структуру в виде строки путем переопределения метода `ToString`.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 10 «Использование массивов».

Если хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Объявить перечисление	Напишите ключевое слово <code>enum</code> , за ним укажите имя типа, а затем в фигурных скобках напишите через запятые список литеральных имен перечисления, например: <code>enum Season { Spring, Summer, Fall, Winter }</code>
Объявить переменную перечисления	Напишите слева название перечисления, а затем имя переменной, поставив после него точку с запятой, например: <code>Season currentSeason;</code>
Присвоить значение переменной перечисления	Напишите имя литерала перечисления, указав также имя того перечисления, которому оно принадлежит, например: <code>currentSeason = Spring; // ошибка</code> <code>currentSeason = Season.Spring; // правильно</code>
Объявить структурный тип	Напишите ключевое слово <code>struct</code> , после которого укажите имя структурного типа, а затем укажите тело структуры (конструкторы, методы и поля), например: <code>struct Time</code> <code>{</code> <code> public Time(int hh, int mm, int ss)</code> <code> { ... }</code> <code> ...</code> <code> private int hours, minutes, seconds;</code> <code>}</code>
Объявить структурную переменную	Напишите имя структурного типа, за ним имя переменной, поставив после него точку с запятой, например: <code>Time now;</code>
Инициализировать структурную переменную значением	Инициализируйте переменную структурным значением путем вызова конструктора структуры, например: <code>Time lunch = new Time(12, 30, 0);</code>

10 Использование массивов

Прочитав эту главу, вы научитесь:

- объявлять переменные массивов;
- заполнять массив набором элементов данных;
- обращаться к элементам данных, хранящимся в массиве;
- осуществлять последовательный обход элементов данных в массиве.

Вы уже видели, как создаются и используются переменные множества различных типов. Но все встречавшиеся до сих пор примеры переменных имели одно общее свойство — они хранили информацию об одном элементе (типа `int`, `float`, `Circle`, `Date` и т. д.). А что делать, если нужно работать с набором элементов? Одним из решений будет создание переменной для каждого элемента набора, но это вызовет ряд новых вопросов: сколько переменных понадобится? Как их следует назвать? Если нужно с каждым элементом набора проделать одну и ту же операцию (например, увеличить значение каждой целочисленной переменной набора на единицу), то как тогда избежать применения повторяющегося кода? Использование переменной для отдельных элементов предполагает, что при написании программы вы знаете, сколько именно элементов потребуется. Но часто ли такое бывает? Например, если создается приложение,читывающее и обрабатывающее записи из базы данных, то сколько записей в этой базе и насколько высока вероятность изменения этого количества?

Механизм, помогающий решить эти проблемы, предоставляют массивы.

Объявление и создание массива

Массив представляет собой неупорядоченную последовательность элементов. В отличие от полей в структуре или классе, у которых могут быть разные типы, у всех элементов массива один и тот же тип. Элементы массива находятся в сплошном блоке памяти, и в отличие от полей в структуре или классе, доступ к которым осуществляется по именам, доступ к элементам массива получают с помощью индекса.

Объявление переменных массивов

Переменная массива объявляется путем указания названия типа элемента, за которым стоит пара квадратных скобок, а затем имя переменной. Квадратные скобки обозначают, что переменная является массивом. Например, для объявления массива `int`-переменных по имени `pins` (для набора личных идентификаторов) можно набрать следующий код:

```
int[] pins; // Личные идентификаторы
```



ПРИМЕЧАНИЕ Если вам приходилось программировать на Microsoft Visual Basic, обратите внимание на то, что в объявлении используются не круглые, а квадратные скобки. Если вы знакомы с языками С и С++, также заметьте, что размер массива не является частью объявления. Кроме того, квадратные скобки должны стоять перед именем переменной.

В качестве элементов массива можно выбирать не только простые типы. Допускается также создание массивов структур, перечислений и классов. Например, можно создать массив `Date`-структур:

```
Date[] dates;
```



СОВЕТ Зачастую массивам больше подходят имена во множественном числе, такие как `places` (где элемент относится к типу `Place`), `people` (где элемент относится к типу `Person`) или `times` (где каждый элемент относится к типу `Time`).

Создание экземпляра массива

Независимо от типов своих элементов массивы являются ссылочными типами. Это означает, что переменная массива ссылается на сплошной блок памяти, содержащий элементы массива в динамической памяти (куче). (Описание значений и ссылок, а также отличий стека от кучи можно найти в главе 8 «Основные сведения о значениях и ссылках».) Это правило действует независимо от типа данных, к которому относятся элементы массива. Даже если массив содержит значение такого типа, как `int`, память все равно будет выделяться в динамической области — этот как раз один из тех случаев, когда типам значений память в стеке не выделяется.

Вспомним, что при объявлении переменной класса память объекту не выделяется до тех пор, пока с помощью ключевого слова `new` не будет создан экземпляр этого класса. Массивы действуют по такой же схеме: когда объявляется переменная массива, вы не объявляете ее размер и ей не выделяется память, лишь в стек помещается ссылка. Память массиву выделяется только при создании экземпляра, и именно тогда указывается размер массива.

Для создания экземпляра массива используется ключевое слово `new`, за которым стоит тип элемента, затем указывается размер создаваемого массива,

заключенный в квадратные скобки. При создании массива происходит также инициализация его элементов с использованием уже известных исходных значений (`0`, `null` или `false` в зависимости от того, к какому типу относятся его элементы — числовому, ссылочному или булеву соответственно). Например, чтобы создать и проинициализировать новый массив из четырех целых чисел для объявленной ранее переменной `pins`, нужно набрать следующий код:

```
pins = new int[4];
```

На рис. 10.1 показано, что произойдет при объявлении массива, а затем при создании его экземпляра.

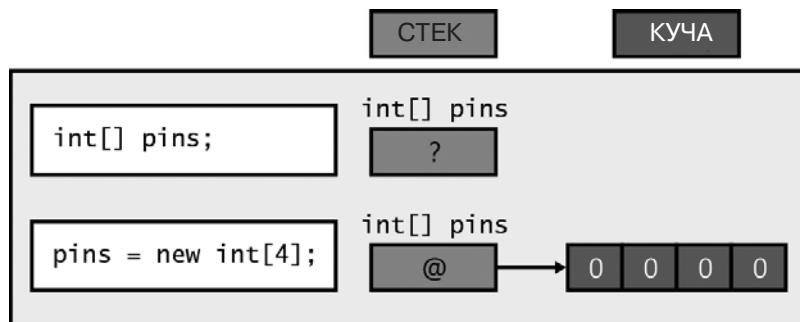


Рис. 10.1

Поскольку память для экземпляра массива выделяется динамически, размер массива не обязательно должен быть постоянным — он, как показано в следующем примере, может вычисляться в ходе выполнения программы:

```
int size = int.Parse(Console.ReadLine());
int[] pins = new int[size];
```

Можно также создать массив размером 0. Как ни странно, но такой массив пригодится в тех случаях, когда размер массива определяется динамически и может быть даже равен нулю. Массив нулевого размера не является массивом пустых элементов — это массив, содержащий нуль элементов.

Заполнение и использование массива

При создании экземпляра массива все элементы этого массива инициализируются исходным значением, зависящим от их типа. Например, для всех числовых значений устанавливается исходное значение `0`, объекты и строки инициализируются в `null`, а для значений типа `DateTime` устанавливаются дата

и время "01/01/0001 00:00:00". Можно изменить этот стиль поведения и инициализировать элементы массива конкретными нужными вам значениями. Это обеспечивается предоставлением списка перечисленных через запятую значений, заключенных в фигурные скобки. Например, чтобы инициализировать переменную `pins` массивом из четырех `int`-переменных со значениями 9, 3, 7 и 2, нужно набрать следующий код:

```
int[] pins = new int[4]{ 9, 3, 7, 2 };
```

Значения, указываемые в фигурных скобках, не обязательно должны быть константами, они, как показано в следующем примере, могут быть значениями, вычисляемыми в ходе выполнения программы. Код этого примера заполняет массив `pins` четырьмя случайными числами:

```
Random r = new Random();
int[] pins = new int[4]{ r.Next() % 10, r.Next() % 10,
    r.Next() % 10, r.Next() % 10 };
```



ПРИМЕЧАНИЕ Класс `System.Random` представляет собой генератор псевдослучайных чисел. Метод `Next` по умолчанию возвращает положительное случайное целое число в диапазоне от 0 до `Int32.MaxValue`. Метод `Next` является перегружаемым, и его другие версии позволяют указывать минимальное и максимальное значения диапазона. Пассивный конструктор класса `Random` в качестве инициализирующего значения, используемого генератором случайного числа, применяет значение, зависящее от текущего времени, что снижает возможности класса по дублированию последовательности случайных чисел. Используя перегружаемую версию конструктора, можно предоставить собственное инициализирующее значение. Таким образом, в целях тестирования приложения можно будет генерировать повторяющуюся последовательность случайных чисел.

Количество значений в фигурных скобках должно в точности соответствовать размеру создаваемого экземпляра массива:

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // ошибка в ходе компиляции
int[] pins = new int[4]{ 9, 3, 7 };     // ошибка в ходе компиляции
int[] pins = new int[4]{ 9, 3, 7, 2 }; // компиляция завершается успешно
```

Когда переменная массива инициализируется именно таким образом, `new`-выражение и размер массива можно опустить. В таком случае компилятор вычислит размер из количества инициализирующих значений и сгенерирует код для создания массива:

```
int[] pins = { 9, 3, 7, 2 };
```

Если создается массив структур или объектов, каждую структуру в массиве можно инициализировать путем вызова конструктора структуры или класса:

```
Time[] schedule = { new Time(12,30), new Time(5,30) };
```

Создание массива с неявно заданным типом элементов

Тип элемента при объявлении массива должен соответствовать типу тех элементов, которые вы пытаетесь поместить в массив. Например, если, как показано в предыдущих примерах, объявляется, что `pins` будет массивом `int`-элементов, вы не можете помещать в этот массив элементы типов `double`, `string`, `struct` или любых других, отличных от `int`. Если при объявлении массива указывается список значений для инициализации, можно позволить компилятору C# вывести фактический тип элементов массива за вас:

```
var names = new[]{"John", "Diana", "James", "Francesca"};
```

В этом примере компилятор C# определяет, что переменная `names` является строковым массивом. В этом объявлении стоит обратить внимание на две синтаксические особенности. Во-первых, из указания типа убраны квадратные скобки — переменная `names` в данном случае объявлена просто как `var`, а не как `var[]`. Во-вторых, перед списком значений инициализации потребовалось указать оператор `new` и поставить квадратные скобки.

Если используется этот синтаксис, нужно обеспечить принадлежность всех инициализирующих значений одному и тому же типу. Следующий пример вызовет ошибку во время компиляции «*No best type found for implicitly-typed array*», которая означает, что при объявлении массива без явного указания типа компилятор не смог определить доминирующий тип данных:

```
var bad = new[]{"John", "Diana", 99, 100};
```

Но в некоторых случаях компилятор приводит элементы к другому типу, если в этом есть какой-то смысл. В следующем примере кода числовой массив является массивом `double`-элементов, поскольку константы `3.5` и `99.999` относятся к данным типа `double` и компилятор C# может привести целочисленные значения `1` и `2` к `double`-значениям:

```
var numbers = new[]{1, 2, 3.5, 99.999};
```

Все же лучше избегать смешанных типов, и не стоит тешить себя надеждой, что компилятор выполнит за вас их приведение к нужному типу.

Объявление массивов с неявным заданием типа лучше всего подходит для работы с безымянными типами, рассмотренными в главе 7 «Создание классов и объектов и управление ими». Следующий код предназначен для создания массива безымянных объектов, каждый из которых содержит два поля, определяющих имя и возраст членов моей семьи:

```
var names = new[] { new { Name = "John", Age = 50 },
                    new { Name = "Diana", Age = 50 },
                    new { Name = "James", Age = 23 },
                    new { Name = "Francesca", Age = 21 } };
```

Поля в безымянных типах должны быть одинаковыми для каждого элемента массива.

Обращение к отдельным элементам массива

Для получения доступа к отдельному элементу массива следует задать индекс, показывающий, какой именно элемент вам нужен. Индексация элементов массива начинается с нуля, поэтому начальный элемент массива имеет индекс 0, а не 1. Значение индекса 1 позволяет обратиться ко второму элементу. Например, считать содержимое элемента 2 (третьего элемента) массива `pins` в `int`-переменную можно с помощью следующего кода:

```
int myPin;  
myPin = pins[2];
```

Аналогично этому, назначая значение проиндексированному элементу, можно изменить содержимое массива:

```
myPin = 1645;  
pins[2] = myPin;
```

Все обращения к элементам массива проходят проверку на выход за пределы его границ. Если указать индекс меньше нуля или больше длины массива или равный ей, компилятор выдаст исключение, связанное с выходом индекса массива за пределы допустимого диапазона значений, — `IndexOutOfRangeException`:

```
try  
{  
    int[] pins = { 9, 3, 7, 2 };  
    Console.WriteLine(pins[4]); // ошибка, 4-й, он же последний, элемент  
                           // имеет индекс 3  
}  
catch (IndexOutOfRangeException ex)  
{  
    ...  
}
```

Последовательный обход элементов массива

Массивы в Microsoft .NET Framework фактически являются экземплярами класса `System.Array`, и в этом классе определен ряд полезных свойств и методов. Например, чтобы узнать количество содержащихся в массиве элементов, можно запросить значение свойства `Length`, а для последовательного обхода всех элементов массива — воспользоваться инструкцией `for`. В следующем простом примере кода значения элементов массива `pins` записываются в консоль:

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
    int pin = pins[index];
    Console.WriteLine(pin);
}
```



ПРИМЕЧАНИЕ `Length` является свойством, а не методом, поэтому при его вызове круглые скобки не используются. Свойства будут рассматриваться в главе 15 «Реализация свойств для доступа к полям».

Начинающие программисты склонны забывать, что массивы начинаются с элемента 0, а номер последнего элемента вычисляется с помощью выражения `Length - 1`. В C# предоставляется инструкция `foreach`, с помощью которой можно выполнять последовательный обход всех элементов массива, не задумываясь об этих обстоятельствах. Вот как, к примеру, выглядит предыдущая инструкция `for`, переписанная в виде ее эквивалента, использующего инструкцию `foreach`:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
    Console.WriteLine(pin);
}
```

Инструкция `foreach` объявляет переменную итерации (в данном примере это `int pin`), которая автоматически получает значение каждого элемента массива. Тип этой переменной должен соответствовать типу элементов массива. Использование инструкции `foreach` является предпочтительным способом последовательного обхода всех элементов массива: с ее помощью предназначение кода выражается напрямую, что позволяет отказаться от всего оснащения, присущего циклу `for`. Но в некоторых случаях потребность возвратиться к использованию инструкции `for` все же будет возникать.

- ❑ Инструкция `foreach` всегда выполняет итерацию в отношении всего массива. Если нужно обойти только известную часть массива (например, первую половину) или пропустить конкретные элементы (например, каждый третий), проще воспользоваться инструкцией `for`.
- ❑ Инструкция `foreach` всегда выполняет итерацию от индекса 0 до индекса `Length - 1`. Если нужно обойти элементы массива в обратном порядке или в какой-то иной последовательности, проще воспользоваться инструкцией `for`.
- ❑ Если телу цикла необходимо знать значение индекса элемента, а не просто его значение, следует воспользоваться инструкцией `for`.
- ❑ Если нужно изменить элементы массива, придется воспользоваться инструкцией `for`. Дело в том, что переменная итерации инструкции `foreach` является копией каждого элемента массива, предназначенней только для чтения.



СОВЕТ Пытаться выполнять итерацию в отношении массива нулевой длины с помощью инструкции `foreach` абсолютно безопасно.

Можно объявить переменную итерации с помощью ключевого слова `var` и позволить компилятору C# самому определить тип переменной на основе типа элементов массива. Особую пользу от этого можно извлечь, если вам неизвестен тип элементов массива, например, в том случае, когда массив содержит безымянные объекты. В следующем примере показано, как можно выполнить последовательный обход элементов массива членов семьи, который был показан ранее:

```
var names = new[] { new { Name = "John", Age = 50 },
                    new { Name = "Diana", Age = 50 },
                    new { Name = "James", Age = 23 },
                    new { Name = "Francesca", Age = 21 } };
foreach (var familyMember in names)
{
    Console.WriteLine($"Name: {familyMember.Name}, Age: {familyMember.Age}");
}
```

Передача массивов в качестве параметров и возвращаемых значений метода

В C# вполне допустимо определять методы, принимающие массивы в качестве параметров или передающие их обратно в качестве возвращаемых значений.

Синтаксис для передачи массива в качестве параметра во многом похож на тот, который используется для объявления массива. Например, в следующем фрагменте кода определяется метод по имени `ProcessData`, получающий в качестве параметра массив целочисленных значений. Тело метода последовательно обходит элементы массива и выполняет некую неуказанную обработку каждого элемента:

```
public void ProcessData(int[] data)
{
    foreach (int i in data)
    {
        ...
    }
}
```

Важно помнить, что массивы являются ссылками на объекты, поэтому при изменении внутри такого метода, как `ProcessData`, содержимого массива, переданного ему в качестве параметра, изменение можно проследить через все ссылки на массив, включая исходный аргумент, принимаемый методом в качестве параметра.

Для возвращения массива из метода нужно в качестве типа возвращаемого значения указать тип массива. Массив создается и заполняется в методе. В следующем примере пользователю предлагается ввести размер массива, а затем данные для каждого элемента. Массив, созданный методом, передается назад в качестве возвращаемого значения:

```
public int[] ReadData()
{
    Console.WriteLine("How many elements?");
    string reply = Console.ReadLine();
    int numElements = int.Parse(reply);

    int[] data = new int[numElements];
    for (int i = 0; i < numElements; i++)
    {
        Console.WriteLine($"Enter data for element {i}");
        reply = Console.ReadLine();
        int elementData = int.Parse(reply);
        data[i] = elementData;
    }
    return data;
}
```

Метод `ReadData` можно вызвать с помощью следующего кода:

```
int[] data = ReadData();
```

ПАРАМЕТРЫ В ВИДЕ МАССИВА И МЕТОД MAIN

Вероятно, вы заметили, что имеющийся в приложении метод `Main` получает в качестве параметра массив строк:

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

Вспомним, что метод `Main` вызывается в самом начале выполнения программы и является точкой входа вашего приложения. Если запускать приложение из окна командной строки, ему можно указать дополнительные аргументы командной строки. Операционная система Windows передает эти аргументы общеязыковой среде выполнения (common language runtime (CLR)), которая в свою очередь передает их в качестве аргументов методу `Main`. Этот механизм дает вам простой способ, позволяющий пользователю предоставлять информацию при запуске приложения, вместо создания интерактивного приглашения для пользователя на ввод данных. Польза от применения этого подхода проявляется при создании утилит, которые могут запускаться из автоматизированных сценариев.

Следующий пример взят из служебного приложения по имени `MyFileUtil`, занимающегося обработкой файлов. Оно ожидает получения в командной строке набора имен файлов и вызывает метод `ProcessFile` (который здесь не показан), обрабатывающий каждый указанный файл:

```
static void Main(string[] args)
{
    foreach (string filename in args)
    {
        ProcessFile(filename);
    }
}
```

Пользователь может запустить приложение MyFileUtil из командной строки следующим образом:

```
MyFileUtil C:\Temp\TestData.dat C:\Users\John\Documents\MyDoc.txt
```

Каждый аргумент командной строки отделен от другого аргумента пробелом. Проверка на допустимость этих аргументов возлагается на приложение MyFileUtil.

Копирование массивов

Массивы относятся к ссылочным типам (вспомним, что массив является экземпляром класса `System.Array`). Переменная массива содержит ссылку на экземпляр массива. Это означает, что при копировании переменной массива вы, как показано в следующем примере, получаете две ссылки на один и тот же экземпляр массива:

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias и pins ссылаются на один и тот же экземпляр массива
```

В данном примере изменения, вносимые в значение `pins[1]`, будут видны также при чтении `alias[1]`.

Если нужно создать копию экземпляра массива (данных в динамической памяти), на который ссылается переменная массива, следует выполнить два действия. Во-первых, нужно создать новый экземпляр массива того же типа, что и у копируемого массива, имеющий такую же длину, как и у него. Во-вторых, нужно выполнить поэлементное копирование данных из исходного массива в новый массив:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
    copy[i] = pins[i];
}
```

Заметьте, что для задания размера нового массива в этом коде используется свойство `Length` исходного массива.

Копирование массива является весьма частым требованием многих приложений, настолько частым, что класс `System.Array` предоставляет ряд полезных методов,

которыми можно воспользоваться при проведении этой операции. Например, метод `CopyTo` копирует содержимое одного массива в другой массив с заданного стартового индекса. В следующем примере все элементы из массива `pins`, начиная с нулевого элемента, копируются в массив `copy`:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

Еще один способ копирования значений предполагает использование имеющегося в `System.Array` статического метода по имени `Copy`. Как и в случае использования `CopyTo`, перед вызовом `Copy` вам следует проинициализировать целевой массив:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```



ПРИМЕЧАНИЕ Методу `Array.Copy` обязательно нужно указать правильное значение параметра `length`. Если указать отрицательное значение, метод выдаст исключение `ArgumentOutOfRangeException`. Если указать значение, превышающее количество элементов в исходном массиве, метод выдаст исключение `ArgumentException`.

Еще одной альтернативой является использование имеющегося в `System.Array` метода экземпляра по имени `Clone`. Этот метод можно вызвать для создания всего массива и копирования в него данных одним действием:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone();
```



ПРИМЕЧАНИЕ Описание методов `Clone` давалось в главе 8. Метод `Clone` класса `Array` возвращает тип `object`, а не тип `Array`, поэтому при использовании этого метода вам следует выполнить приведение типа возвращаемого им значения к типу соответствующего массива. Кроме того, методы `Clone`, `CopyTo` и `Copy` создают поверхностную копию массива (поверхностное и углубленное копирование также описываются в главе 8). Если копируемые элементы массива содержат ссылки, метод `Clone` просто копирует ссылки, а не объекты, на которые они указывают. После копирования оба массива ссылются на один и тот же набор объектов. Если нужно создать углубленную копию такого массива, следует воспользоваться соответствующим кодом в цикле `for`.

Использование многомерных массивов

Массивы, показанные до сих пор, были одномерными, и их можно было бы представить в виде простого списка значений. Но допускается создание

и многомерных массивов. Например, чтобы создать двумерный массив, указывается массив, требующий двух целочисленных индексов. Следующий код позволяет создать двумерный массив по имени `items`, состоящий из 24 целых чисел. Если это вам поможет, можете представлять себе этот массив в виде таблицы, в которой первое значение указывает номера строк, а второе — номера столбцов:

```
int[,] items = new int[4, 6];
```

Чтобы обратиться к элементу массива, предоставляются два индексных значения, указывающих на ячейку (пересечение строки и столбца), содержащую элемент. В следующем фрагменте кода показан ряд примеров использования массива `items`:

```
items[2, 3] = 99; // установка для элемента в ячейке (2, 3) значения 99  
items[2, 4] = items[2, 3]; // копирование элемента из ячейки (2, 3) в ячейку (2, 4)  
items[2, 4]++; // увеличение целочисленного значения в ячейке (2, 4) на единицу
```

Для массива можно указать любое количество размерностей без каких-либо ограничений. В следующем примере кода создается и используется трехмерный массив по имени `cube`. Обратите внимание на то, что для обращения к каждому элементу этого массива следует указывать три индекса:

```
int[, ,] cube = new int[5, 5, 5];  
cube[1, 2, 1] = 101;  
cube[1, 2, 2] = cube[1, 2, 1] * 3;
```

Здесь вполне уместно предостеречь вас от создания массивов более чем с тремя размерностями. Массивы могут занимать довольно большой объем памяти. Массив `cube` содержит 125 элементов ($5 \cdot 5 \cdot 5$). Четырехмерный массив, в котором каждая размерность состоит из пяти индексов, содержит 625 элементов. Если вы начнете создавать массивы с тремя и более размерностями, то вскоре можете исчерпать память. Поэтому при использовании многомерных массивов вы всегда должны быть готовы к перехвату и обработке исключений, свидетельствующих об отсутствии нужного объема памяти, — `OutOfMemoryException`.

Создание ступенчатых массивов

В C# обычные многомерные массивы иногда называют массивами в прямоугольных координатах. У каждой размерности имеется постоянная форма. Например, в следующем табличном представлении, двумерном массиве `items`, в каждой строке содержится 40 элементов, то есть всего 160 элементов:

```
int[,] items = new int[4, 40];
```

Как говорилось в предыдущем разделе, многомерные массивы могут расходовать слишком много памяти. Если приложение использует только некоторые данные в каждом столбце, выделение памяти под неиспользуемые элементы приводит к ее неоправданному расходованию. В данном сценарии можно воспользоваться ступенчатым массивом, в котором все столбцы имеют разную длину:

```
int[][] items = new int[4][];
int[] columnForRow0 = new int[3];
int[] columnForRow1 = new int[10];
int[] columnForRow2 = new int[40];
int[] columnForRow3 = new int[25];
items[0] = columnForRow0;
items[1] = columnForRow1;
items[2] = columnForRow2;
items[3] = columnForRow3;
...
```

В этом примере приложению требуется только 3 элемента в первом столбце, 10 элементов во втором, 40 элементов в третьем и 25 элементов в четвертом. В данном коде показан массив массивов по имени `items`, который, вместо того чтобы быть двумерным массивом, имеет только одну размерность, но элементы в этой размерности сами по себе являются массивами. Более того, общий размер массива `items` составляет 78, а не 160 элементов, и пространство под неиспользуемые приложением элементы не выделяется.

В этом примере стоит выделить ряд синтаксических особенностей. Следующее объявление указывает, что `items` является массивом массивов, состоящих из `int`-элементов:

```
int[][] items;
```

Следующая инструкция инициализирует `items` на содержание четырех элементов, каждый из которых является массивом неопределенной длины:

```
items = new int[4][];
```

Все массивы с `columnForRow0` до `columnForRow3` являются одномерными `int`-массивами, инициализированными на содержание объема данных, требующегося для каждого столбца. И наконец, каждый массив столбцов присвоен соответствующим элементам в массиве `items`, например:

```
items[0] = columnForRow0;
```

Вспомним, что массивы являются ссылочными объектами, поэтому данная инструкция просто добавляет ссылку на `columnForRow0` к первому элементу массива `items`, не копируя при этом никаких данных. Заполнить этот столбец данными можно, либо присваивая значение индексированному элементу в `columnForRow0`,

либо ссылаясь на него посредством массива `items`. Следующие инструкции выполняют одну и ту же задачу:

```
columnForRow0[1] = 99;  
items[0][1] = 99;
```

Этот замысел можно развивать и дальше, если вам нужно будет создать массив массивов из массивов, а не прямоугольный трехмерный массив.



ПРИМЕЧАНИЕ Если вам уже приходилось создавать код на языке программирования Java, то вам должно быть знакомо это понятие. В Java нет многомерных массивов, вместо них можно создавать массивы массивов точно так же, как только что было описано.

В следующем упражнении массивы будут использоваться для создания приложения, сдающего игральные карты и являющегося частью карточной игры. Приложение выводит форму с четырьмя карточными раздачами, произведенными случайным образом из обычной колоды игральных карт (в 52 карты). Вам предстоит завершить создание кода, раздающего карты четырем игрокам.

Использование массивов для реализации карточной игры

Откройте в Microsoft Visual Studio 2015 проект `Cards`, который находится в папке `\Microsoft Press\VCSBS\Chapter 10\Cards` вашей папки документов.

Щелкните в меню Отладка на пункте Начать отладку, чтобы выполнить сборку и запуск приложения. Появится форма с надписью `Card Game` и четырьмя текстовыми полями с надписями `North`, `South`, `East` и `West`. В нижней части будет находиться панель команд, обозначенная многоточием (...). Щелкните на нем, чтобы раскрыть командную панель. Появится кнопка с надписью `Deal` (Раздать) (рис. 10.2).



ПРИМЕЧАНИЕ Используемая здесь технология считается предпочтительным механизмом размещения командных кнопок в приложениях универсальной платформы Windows (Universal Windows Platform (UWP)), и с этого момента все UWP-приложения, представленные в этой книге, будут выполнены в таком стиле.

Щелкните на кнопке `Deal`. Ничего не произойдет. Код, раздающий карты, пока не создан, и в данном упражнении вы как раз и займитесь его написанием.

Вернитесь в среду Visual Studio 2015 и в меню Отладка щелкните на пункте Остановить отладку.

Найдите в окне обозревателя решений файл `Value.cs` и откройте его в окне редактора. В этом файле содержится перечисление по имени `Value`, представляющее по нарастающей карты различного достоинства:

```
enum Value { Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack,  
Queen, King, Ace }
```

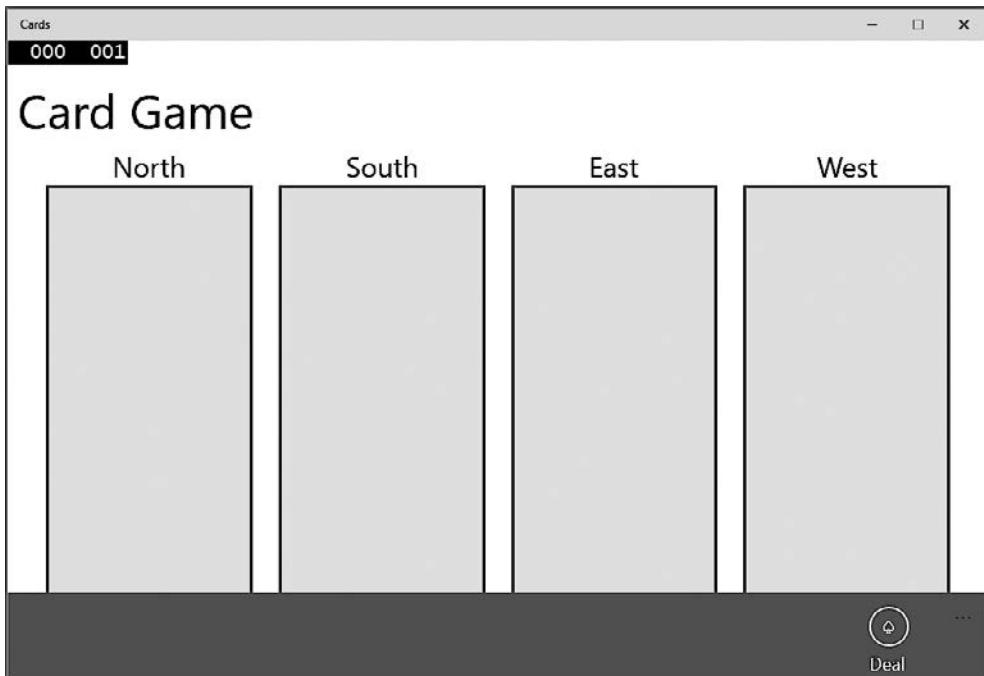


Рис. 10.2

Откройте в окне редактора файл `Suit.cs`. В нем содержится перечисление, представляющее карточные масти в обычной колоде:

```
enum Suit { Clubs, Diamonds, Hearts, Spades }
```

Выведите в окно редактора файл `PlayingCard.cs`. В нем содержится класс `PlayingCard`, моделирующий отдельно взятую игральную карту.

```
class PlayingCard
{
    private readonly Suit suit;
    private readonly Value value;

    public PlayingCard(Suit s, Value v)
    {
        this.suit = s;
        this.value = v;
    }

    public override string ToString()
    {
        string result = $"{this.value} of {this.suit}";
        return result;
    }
}
```

```
public Suit CardSuit()
{
    return this.suit;
}

public Value CardValue()
{
    return this.value;
}

}
```

У этого класса имеются два поля только для чтения, представляющие достоинство и масть карты. Инициализацией этих полей занимается конструктор.



ПРИМЕЧАНИЕ Поля только для чтения хорошо подходят для моделирования данных, которые не должны изменяться после своей инициализации. Присвоить значение полю только для чтения можно с помощью инициализатора при его объявлении или в конструкторе, но после этого значение изменить уже невозможно.

Класс содержит два метода с именами `CardValue` и `CardSuit`, которые возвращают эту информацию, в нем также переопределяется метод `ToString` для возвращения текстового представления карты.



ПРИМЕЧАНИЕ Вообще-то методы `CardValue` и `CardSuit` лучше реализовать в виде свойств, чему вы научитесь, изучая главу 15.

Откройте в окне редактора файл `Pack.cs`. В нем содержится класс `Pack`, моделирующий колоду игральных карт. В начале определения класса `Pack` находятся два открытых поля с именами `NumSuits` и `CardsPerSuit`, содержащие целочисленные константы. В этих двух полях указывается количество мастей в карточной колоде и количество карт каждой масти. Закрытая переменная `cardPack` является двумерным массивом из объектов `PlayingCard`. Первая размерность будет использоваться для указания масти, а вторая — для указания достоинства карты в масти. Переменная `randomCardSelector` является случайным числом, сгенерированным на основе использования класса `Random`. Переменная `randomCardSelector` поможет перетасовать карты перед раздачей игрокам:

```
class Pack
{
    public const int NumSuits = 4;
    public const int CardsPerSuit = 13;
    private PlayingCard[,] cardPack;
    private Random randomCardSelector = new Random();
    ...
}
```

Найдите пассивный конструктор для класса `Pack`. В данный момент этот конструктор не содержит ничего, кроме комментария `// TODO:`. Удалите этот

комментарий и добавьте следующую инструкцию, выделенную жирным шрифтом, чтобы создать экземпляр массива `cardPack` с соответствующими значениями для каждой размерности:

```
public Pack()
{
    this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
}
```

Добавьте к конструктору `Pack` следующий код, выделенный жирным шрифтом. Имеющиеся в нем инструкции заполнят массив `cardPack` отсортированной колодой карт:

```
public Pack()
{
    this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
    for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
    {
        for (Value value = Value.Two; value <= Value.Ace; value++)
        {
            this.cardPack[(int)suit, (int)value] = new PlayingCard(suit, value);
        }
    }
}
```

Внешний цикл `for` осуществляет последовательный обход списка значений перечисления `Suit`, а внутренний цикл обходит все значения, которые каждая карта может иметь в каждой масти. Внутренний цикл создает новый объект `PlayingCard` указанных масти и значения и добавляет его к соответствующему элементу в массиве `cardPack`.



ПРИМЕЧАНИЕ В качестве индексов внутри массива следует использовать один из целочисленных типов. Переменные `suit` и `value` являются переменными перечислений. Но перечисления основаны на целочисленных типах, поэтому их, как показано в коде, вполне свободно можно привести к `int`-типу.

Найдите в классе `Pack` метод `DealCardFromPack`. Он предназначен для выбора случайной карты из колоды, удаления этой карты из колоды, чтобы она не могла быть выбрана повторно, и последующей передачи ее назад в качестве возвращаемого из метода значения.

Первой задачей метода является случайный выбор масти. Удалите из метода комментарий и инструкцию, выдающую исключение `NotImplementedException`, заменив их следующей инструкцией, выделенной жирным шрифтом:

```
public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
}
```

Эта инструкция использует метод `Next`, имеющийся в объекте генератора случайных чисел `randomCardSelector` и предназначенный для возвращения случайного числа, соответствующего масти. Параметр для метода `Next` указывает используемую исключительную верхнюю границу диапазона, выбираемое значение находится между нулем и этим значением за вычетом единицы. Учтите, что возвращаемое значение относится к типу `int`, поэтому, прежде чем присваивать его `Suit`-переменной, следует привести его к соответствующему типу.

Всегда есть вероятность того, что карты выбранной масти уже не останутся. Вам нужно справиться с этой ситуацией и при необходимости выбрать другую масть.

После кода для выбора случайной масти добавьте цикл `while`, выделенный далее жирным шрифтом. Этот цикл вызывает метод `IsSuitEmpty`, чтобы определить, остались ли еще в колоде карты указанной масти (вскоре вам предстоит реализовать логику работы этого метода). Если таких карт не осталось, происходит случайный выбор другой масти (может быть опять выбрана та же самая масть) и выполняется новая проверка. Цикл повторяет процесс, пока не будет выбрана масть, которой принадлежит хотя бы одна оставшаяся в колоде карта:

```
public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
    while (this.IsSuitEmpty(suit))
    {
        suit = (Suit)randomCardSelector.Next(NumSuits);
    }
}
```

Теперь вами уже выбрана случайная масть, представленная в колоде хотя бы одной картой. Следующая задача будет заключаться в выборе случайной карты этой масти. Чтобы выбрать значение карты, можно воспользоваться генератором случайных чисел, но, как и прежде, нет никакой гарантии, что такая карта уже не была сдана. Но можно воспользоваться тем же приемом, что и прежде: вызвать метод `IsCardAlreadyDealt` (который будет изучен и дополнен чуть позже), чтобы определить, не была ли карта уже сдана, а если была, то выбрать другую случайную карту и снова проверить, не была ли она сдана, и повторять процесс до тех пор, пока не будет найдена свободная карта. Чтобы выполнить эти действия, добавьте к методу `DealCardFromPack` после уже имеющегося в нем кода следующие инструкции, выделенные жирным шрифтом:

```
public PlayingCard DealCardFromPack()
{
    ...
    Value value = (Value)randomCardSelector.Next(CardsPerSuit);
```

```
        while (this.IsCardAlreadyDealt(suit, value))
        {
            value = (Value)randomCardSelector.Next(CardsPerSuit);
        }
    }
```

Теперь вы выбрали случайную, не покинувшую колоду ранее игральную карту. Добавьте в конец метода `DealCardFromPack` следующий код для возвращения этой карты и присвоения соответствующему элементу массива `cardPack` значения `null`:

```
public PlayingCard DealCardFromPack()
{
    ...
    PlayingCard card = this.cardPack[(int)suit, (int)value];
    this.cardPack[(int)suit, (int)value] = null;
    return card;
}
```

Найдите метод `IsSuitEmpty`. Не забудьте, что целью этого метода является получение параметра `Suit` и возвращение булева значения, показывающего, остались ли еще карты данной масти в колоде. Удалите из этого метода комментарий и инструкцию, выдающую исключение `NotImplementedException`, а затем добавьте к нему следующий код, выделенный жирным шрифтом:

```
private bool IsSuitEmpty(Suit suit)
{
    bool result = true;
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        if (!IsCardAlreadyDealt(suit, value))
        {
            result = false;
            break;
        }
    }
    return result;
}
```

Этот код обеспечивает последовательный доступ к возможным значениям карт и использует метод `IsCardAlreadyDealt` (создание кода которого будет завершено на следующем этапе) для определения того, осталась ли еще в массиве `cardPack` карта с указанными мастью и достоинством. Если при прохождении цикла карта будет найдена, значение переменной `result` будет установлено в `false` и инструкция `break` прервет выполнение цикла. Если цикл завершится, а карта не будет найдена, у переменной `result` так и останется исходное значение `true`. Значение переменной `result` будет передано обратно в качестве возвращаемого методом значения.

Найдите метод `IsCardAlreadyDealt`. Целью этого метода является определение того, сдана ли карта с указанными мастью и достоинством и удалена ли она из колоды. Чуть позже вы увидите, что в процессе сдачи карты метод `DealCardFromPack` удаляет карту из массива `cardPack` и устанавливает значение соответствующего элемента в `null`. Замените тело этого метода следующим кодом, выделенным жирным шрифтом:

```
private bool IsCardAlreadyDealt(Suit suit, Value value)
=> (this.cardPack[(int)suit, (int)value] == null);
```

Этот метод возвращает `true`, если элемент в массиве `cardPack` соответствующих масти и достоинства имеет значение `null`, и возвращает `false` в противном случае.

Следующий этап будет заключаться в добавлении выбранной игральной карты к картам игрока. Откройте в окне редактора файл `Hand.cs`. В нем содержится класс `Hand`, реализующий карты, находящиеся на руках у игрока (то есть все карты, сданные одному игроку).

В этом файле содержится открытое поле с неизменным значением `int`-типа по имени `HandSize`, имеющее значение размера того набора карт, который один игрок получает на руки (13). В нем также содержится массив объектов `PlayingCard`, инициализированный с использованием константы `HandSize`. Поле `playingCardCount` будет использоваться вашим кодом, чтобы отслеживать количество карт, имеющихся в данный момент у игрока в результате сдачи карт:

```
class Hand
{
    public const int HandSize = 13;
    private PlayingCard[] cards = new PlayingCard[HandSize];
    private int playingCardCount = 0;
    ...
}
```

Метод `ToString` создает строку, представляющую карты, находящиеся на руках у игрока. В нем используется цикл `foreach` для последовательного обращения к элементам массива `cards`, и в отношении каждого найденного им объекта `PlayingCard` вызывается метод `ToString`. Для получения нужного формата эти строки объединяются с установкой между ними символа новой строки (`\n`):

```
public override string ToString()
{
    string result = "";
    foreach (PlayingCard card in this.cards)
    {
        result += $"{card.ToString()}\\n";
    }
    return result;
}
```

Найдите в классе `Hand` метод `AddCardToHand`. Целью этого метода является выдача игральной карты, указанной в качестве параметра, на руки игроку. Добавьте к этому методу следующий код, выделенный жирным шрифтом:

```
public void AddCardToHand(PlayingCard cardDealt)
{
    if (this.playingCardCount >= HandSize)
    {
        throw new ArgumentException("Too many cards");
    }
    this.cards[this.playingCardCount] = cardDealt;
    this.playingCardCount++;
}
```

Сначала этот код проверяет, не сданы ли на руки все полагающиеся игроку карты. Если набор полон, выдается исключение `ArgumentException` (это не должно случиться, но все же лучше подстраховаться). В противном случае карта добавляется к массиву `cards` с индексом, указанным переменной `playingCardCount`, и значение этой переменной увеличивается на единицу.

Раскройте в обозревателе решений узел `MainPage.xaml` и откройте в окне редактора файл `MainPage.xaml.cs`.

Код этого файла предназначен для окна `Card Game`. Найдите метод `dealClick`. Этот метод запускается, когда пользователь щелкает на кнопке `Deal`. На данный момент в нем содержится пустой `try`-блок и обработчик исключения, выводящий на экран сообщение о том, что произошло исключение.

Добавьте к `try`-блоку следующую инструкцию, выделенную жирным шрифтом:

```
private void dealClick(object sender, RoutedEventArgs e)
{
    try
    {
        pack = new Pack();
    }
    catch (Exception ex)
    {
        ...
    }
}
```

Эта инструкция просто создает новую колоду карт. Ранее уже было показано, что соответствующий класс содержит двумерный массив из находящихся в колоде карт и конструктор заполняет этот массив подробными данными о каждой карте. Из этой колоды следует создать четыре набора карт, сдаваемых на руки каждому игроку.

Добавьте к `try`-блоку следующую инструкцию, выделенную жирным шрифтом:

```
try
{
    pack = new Pack();
    for (int handNum = 0; handNum < NumHands; handNum++)
    {
        hands[handNum] = new Hand();
    }
}
catch (Exception ex)
{
    ...
}
```

Этот цикл **for** создает из колоды четыре набора карт, сдаваемых на руки каждому из игроков, и сохраняет их в массиве по имени **hands**. Каждый набор изначально пуст, поэтому вам нужно сдать карты из колоды каждому игроку.

Добавьте к циклу **for** следующий код, выделенный жирным шрифтом:

```
try
{
    ...
    for (int handNum = 0; handNum < NumHands; handNum++)
    {
        hands[handNum] = new Hand();
        for (int numCards = 0; numCards < Hand.HandSize; numCards++)
        {
            PlayingCard cardDealt = pack.DealCardFromPack();
            hands[handNum].AddCardToHand(cardDealt);
        }
    }
}
catch (Exception ex)
{
    ...
}
```

Внутренний цикл **for** заполняет каждый набор сдаваемых игрокам карт с помощью метода **DealCardFromPack**, позволяющего извлечь случайную карту из колоды, и метода **AddCardToHand**, позволяющего добавить эту карту к набору карт, сдаваемых игроку.

Добавьте после внешнего цикла **for** следующий код, выделенный жирным шрифтом:

```
try
{
    ...
    for (int handNum = 0; handNum < NumHands; handNum++)
    {
        ...
    }

    north.Text = hands[0].ToString();
```

```

    south.Text = hands[1].ToString();
    east.Text = hands[2].ToString();
    west.Text = hands[3].ToString();
}
catch (Exception ex)
{
    ...
}

```

Когда будут сданы все карты, этот код покажет в текстовых полях формы, что именно находится на руках у каждого игрока. Эти текстовые поля называются `north`, `south`, `east` и `west`. Для форматирования вывода карт, находящихся на руках у каждого игрока, в коде используется метод `ToString`.

Если на каком-либо из этапов работы будет выдано исключение, обработчик исключения выведет окно сообщения со сведениями об ошибке, характерной для данного исключения.

Щелкните в меню Отладка на пункте Начать отладку. Как только появится окно Card Game, раскройте панель команд и щелкните на кнопке Deal. Карты из колоды должны быть сданы на руки в случайном порядке и показаны в форме для каждого игрока, что создаст на экране примерно следующую картину (рис. 10.3).

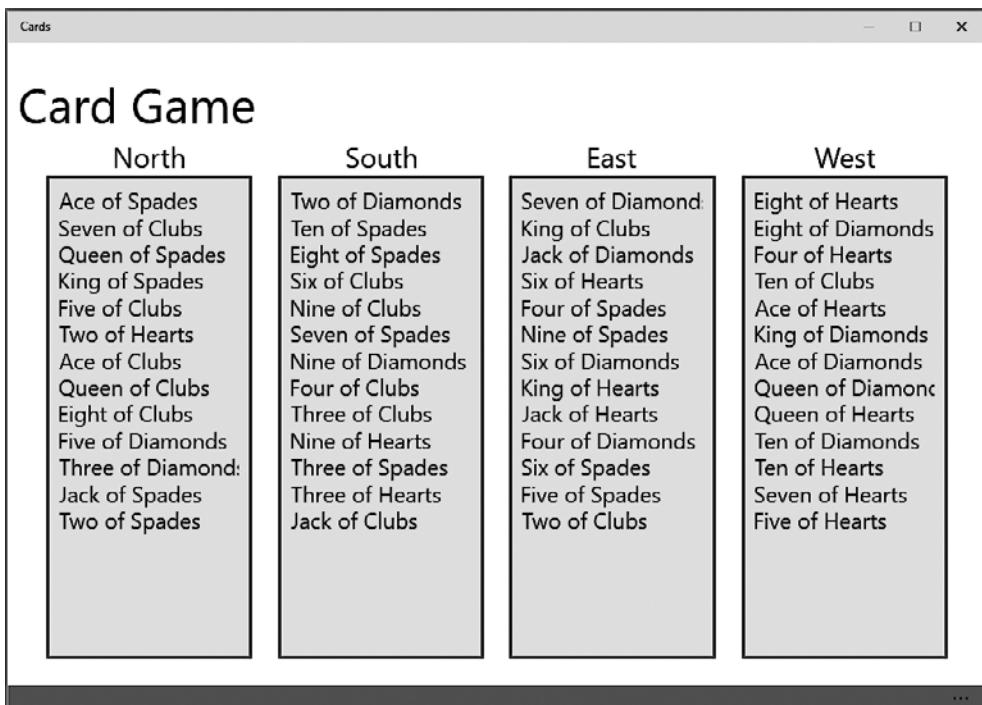


Рис. 10.3

Щелкните на кнопке Deal еще раз. Убедитесь в том, что будет создан новый набор карт и на этот раз карты у каждого игрока будут другими.

Вернитесь в среду Visual Studio и остановите отладку.

Выводы

В этой главе вы научились создавать и использовать массивы для работы с наборами данных. Вы увидели, как объявляются и инициализируются массивы, как осуществляется доступ к данным, содержащимся в массивах, как массивы передаются в качестве параметров методам и как массивы возвращаются из методов. Вы также научились создавать многомерные массивы и узнали, как используются массивы массивов.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 11.

Если хотите выйти из среды Visual Studio 2015, то в меню Файл щелкните на пункте Выход. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке Да и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Объявить переменную массива	Наберите название типа элемента, затем поставьте квадратные скобки, а за ними укажите имя переменной и поставьте точку с запятой, например: <code>bool[] flags;</code>
Создать экземпляр массива	Наберите ключевое слово new, затем укажите название типа элемента, после чего укажите в квадратных скобках размер массива, например: <code>bool[] flags = new bool[10];</code>
Инициализировать элементы массива конкретными значениями	Запишите для массива через запятую список значений, заключенный в фигурные скобки, например: <code>bool[] flags = { true, false, true, false };</code>
Определить количество элементов в массиве	Используйте свойство Length, например: <code>bool[] flags = ...;</code> <code>...</code> <code>int noOfElements = flags.Length;</code>

Чтобы	Сделайте следующее
Обратиться к отдельно взятому элементу массива	<p>Наберите имя переменной массива, после чего поставьте целочисленный индекс элемента, заключенный в квадратные скобки. Не забудьте, что индексация массива начинается с нуля, а не с единицы, например:</p> <pre>bool initialElement = flags[0];</pre>
Обратиться последовательно ко всем элементам массива	<p>Воспользуйтесь инструкцией for или foreach, например:</p> <pre>bool[] flags = { true, false, true, false }; for (int i = 0; i < flags.Length; i++) { Console.WriteLine(flags[i]); } foreach (bool flag in flags) { Console.WriteLine(flag); }</pre>
Объявить переменную многомерного массива	<p>Наберите название типа элемента, за ним поставьте набор, состоящий из квадратных скобок и разделителя в виде запятой, указывающий на количество размерностей, после него укажите имя переменной, а за ним поставьте точку с запятой. Например, для создания двумерного массива по имени table и его инициализации для хранения четырех строк по шесть столбцов воспользуйтесь следующими инструкциями:</p> <pre>int[,] table; table = new int[4,6];</pre>
Объявить переменную ступенчатого массива	<p>Объявите переменную в виде массива дочерних массивов. Каждый дочерний массив можно инициализировать на различную длину. Например, чтобы создать ступенчатый массив по имени items и инициализировать каждый дочерний массив, воспользуйтесь следующими инструкциями:</p> <pre>int[][] items; items = new int[4][]; items[0] = new int[3]; items[1] = new int[10]; items[2] = new int[40]; items[3] = new int[25];</pre>

11 Основные сведения о массивах параметров

Прочитав эту главу, вы научитесь:

- создавать метод, принимающий любое количество аргументов с использованием ключевого слова `params`;
- создавать метод, принимающий любое количество аргументов любого типа с использованием ключевого слова `params` в сочетании с типом объекта;
- объяснять, чем отличаются методы, принимающие массивы параметров, от методов, принимающих необязательные параметры.

Массивы параметров применяются при необходимости создания методов, принимающих в качестве параметров любое количество аргументов, возможно, относящихся к разным типам. Если вы знакомы с понятиями объектно-ориентированного программирования, то последнее предложение может вам сильно не понравиться, ведь при объектно-ориентированном подходе подобная задача решается путем определения перегружаемых методов. Но перегрузка не всегда оказывается самым рациональным подходом, особенно если нужно создать метод, действительно способный принимать разное количество параметров, каждый из которых при вызове метода может иметь свой, отличный от других тип. В этой главе описываются способы использования массивов параметров, предназначенные для разрешения подобных ситуаций.

Давайте вспомним, что такое перегрузка

Перегрузка — это техническое понятие для объявления двух и более методов с одним и тем же именем в одном и том же пространстве имен. Перегрузка метода хорошо подходит для тех случаев, когда нужно выполнить одно и то

же действие в отношении аргументов различных типов. Классическим примером перегрузки в Microsoft Visual C# является метод `Console.WriteLine`. Этот метод может многократно перегружаться, позволяя вам передавать ему любой аргумент, относящийся к простым типам данных. В следующем примере кода показано несколько способов определения метода `WriteLine` в классе `Console`:

```
class Console
{
    public static void WriteLine(Int32 value)
    public static void WriteLine(Double value)
    public static void WriteLine(Decimal value)
    public static void WriteLine(Boolean value)
    public static void WriteLine(String value)
    ...
}
```



ПРИМЕЧАНИЕ В документации по методу `WriteLine` в качестве его параметров используются структурные типы, определенные в пространстве имен `System`, а не псевдонимы, используемые в C# для этих типов. Например, перегружаемый метод, выводящий значение типа `int`, на самом деле получает в качестве параметра значение типа `Int32`. Список структурных типов и их отображений на C#-псевдонимы приведен в главе 9 «Создание типов значений с использованием перечислений и структур».

Как бы ни была полезна перегрузка, она не может подойти абсолютно для всех случаев. В частности, перегрузка не позволяет легко справиться с ситуацией, при которой тип параметров не изменяется, а вот их количество может изменяться. Что, к примеру, делать, если нужно вывести на консоль множество значений? Неужели придется предоставлять версию `Console.WriteLine`, способную принять два параметра в различных комбинациях, а также другие версии, способные принять три параметра и т. д.? Так ведь можно слишком быстро утомиться. И неужели вы так легко согласитесь с таким большим количеством дублей перегружаемых методов? Вряд ли. К счастью, есть способ написания метода, принимающего переменное число аргументов: можно воспользоваться массивом параметров, объявляемым с использованием ключевого слова `params`.

Чтобы понять, как `params`-массивы решают эту проблему, будет полезно разобраться с использованием обычных массивов и со слабыми сторонами этого подхода.

Использование аргументов в виде массивов

Предположим, нужно создать метод для определения минимального значения в наборе значений, переданных в качестве параметров. Один из способов предполагает использование массива. Например, чтобы определить наименьшее

из нескольких `int`-значений, можно написать статический метод по имени `Min` с единственным параметром, представляющим собой массив из `int`-значений:

```
class Util
{
    public static int Min(int[] paramList)
    {
        // Проверка предоставления вызывающим кодом хотя бы одного параметра.
        // Если нет, выдача исключения ArgumentException, свидетельствующего о том,
        // что невозможно найти наименьшее значение в пустом списке.
        if (paramList == null || paramList.Length == 0)
        {
            throw new ArgumentException("Util.Min: not enough arguments");
        }

        // Установка текущего минимального значения, найденного в списке
        // параметров, на первый элемент

        int currentMin = paramList[0];

        // Последовательное обращение к списку параметров с целью поиска любого
        // значения, меньшего, чем то, что содержится в currentMin
        foreach (int i in paramList)
        {
            // Если в цикле обнаруживается элемент, меньший, чем значение,
            // содержащееся в currentMin, установка для currentMin этого значения
            if (i < currentMin)
            {
                currentMin = i;
            }
        }

        // В конце цикла в currentMin содержится значение наименьшего элемента
        // из списка параметров, поэтому возвращается именно это значение.
        return currentMin;
    }
}
```



ПРИМЕЧАНИЕ Класс `ArgumentException` разработан специально для того, чтобы метод выдавал исключения, если предоставленные аргументы не отвечают требованиям метода.

Чтобы воспользоваться методом `Min` для поиска минимума среди двух `int`-переменных с именами `first` и `second`, можно создать следующий код:

```
int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Util.Min(array);
```

А для использования метода `Min` с целью поиска минимума среди трех `int`-переменных (с именами `first`, `second` и `third`) можно создать следующий код:

```
int[] array = new int[3];
array[0] = first;
array[1] = second;
array[2] = third;
int min = Util.Min(array);
```

Как видите, такое решение позволяет не создавать большое количество перегрузок, но за это приходится платить: вам следует написать дополнительный код для заполнения массива тем, что вы передаете методу. Разумеется, при желании можно воспользоваться безымянным массивом:

```
int min = Util.Min(new int[] {first, second, third});
```

Но главное здесь то, что вам все равно нужно создавать и заполнять массив, и синтаксис при этом может выглядеть не вполне понятным. Решение проблемы заключается в том, чтобы заставить компилятор написать часть этого кода за вас за счет использования в качестве параметра для метода `Min` массива параметров.

Объявление массива параметров

Используя `params`-массив, вы сможете передавать методу переменное количество аргументов. Этот массив обозначается при определении параметров метода с помощью ключевого слова `params`, которое служит модификатором массива параметров. Вот как, к примеру, выглядит еще один вариант метода `Min`, на этот раз с массивом параметров, объявленным в качестве `params`-массива:

```
class Util
{
    public static int Min(params int[] paramList)
    {
        // код точно такой же, как и ранее использованный
    }
}
```

Эффект от применения ключевого слова `params` в отношении метода `Min` заключается в том, что вы получаете возможность вызывать метод, используя любое количество целочисленных аргументов, не заботясь о создании массива. Например, для нахождения минимума из двух целочисленных значений можно просто написать следующий код:

```
int min = Util.Min(first, second);
```

компилятор преобразует этот вызов в код, похожий на следующий:

```
int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Util.Min(array);
```

Для нахождения минимума из трех целочисленных значений следует написать код, показанный далее, который также преобразуется компилятором в соответствующий код, использующий массив:

```
int min = Util.Min(first, second, third);
```

Оба вызова метода `Min` (один с двумя, а один с тремя аргументами) обращаются к одному и тому же методу `Min`, объявленному с ключевым словом `params`. И как вы уже могли догадаться, этот метод `Min` можно вызывать с любым количеством `int`-аргументов. Компилятор просто подсчитывает количество `int`-аргументов, создает `int`-массив такого же размера, заполняет массив аргументами, а затем вызывает метод, передавая ему единственный массив параметров.



ПРИМЕЧАНИЕ При наличии опыта программирования на С или на С++ вы можете распознать в `params` безопасный по отношению к типам эквивалент макроса `varargs` из заголовочного файла `stdarg.h`. В языке Java есть также средство `varargs`, работающее примерно так же, как ключевое слово `params` в C#.

По поводу `params`-массивов нужно отметить следующий ряд обстоятельств.

- ❑ Ключевое слово `params` нельзя использовать с многомерными массивами. Код следующего примера компиляцию не пройдет:

```
// ошибка в ходе компиляции  
public static int Min(params int[,] table)  
...
```

- ❑ Метод, основанный исключительно на применении ключевого слова `params`, перегрузить невозможно. Как показано в следующем примере, ключевое слово `params` не является частью сигнатуры метода. В данном случае в контексте вызывающего кода компилятор не сможет отличить эти методы друг от друга:

```
// ошибка в ходе компиляции: продублированное объявление  
public static int Min(int[] paramList)  
...  
public static int Min(params int[] paramList)  
...
```

- ❑ При использовании `params`-массивов нельзя указывать модификатор `ref` или `out`:

```
// ошибки в ходе компиляции  
public static int Min(ref params int[] paramList)  
...  
public static int Min(out params int[] paramList)  
...
```

- ❑ Массив `params` должен быть последним параметром. (Это означает, что у вас на один метод может быть только один `params`-массив.) Изучите данный пример:

```
// ошибка в ходе компиляции
public static int Min(params int[] paramList, int i)
...
```

- Метод, в объявлении которого не используется ключевое слово `params`, всегда имеет приоритет над `params`-методом. Это означает, что создать для наиболее распространенных случаев перегружаемую версию метода все же можно:

```
public static int Min(int leftHandSide, int rightHandSide)
...
public static int Min(params int[] paramList)
...
```

Первая версия метода `Min` используется, когда он вызывается с использованием двух `int`-аргументов. Вторая версия используется при предоставлении другого количества `int`-аргументов. Сюда же включается и случай вызова метода без аргументов. Добавление метода, не использующего `params`-массив, может стать полезным приемом оптимизации, поскольку компилятору не придется создавать и заполнять слишком большое количество массивов.

Использование конструкции `params object[]`

Массив параметров `int`-типа весьма удобен. С его помощью в вызове метода можно передать любое количество `int`-аргументов. А что делать, если изменяется не только количество аргументов, но и их тип? В C# имеется способ решения и этой задачи. Технология основана на том, что `object` является основой всех классов и компилятор может генерировать код, преобразующий типы значений (не являющиеся классами) в объекты с использованием упаковки (`boxing`), согласно описанию, приведенному в главе 8 «Основные сведения о значениях и ссылках». Массив параметров типа `object` можно использовать при объявлении метода, принимающего любое количество `object`-аргументов, что позволяет передаваемым аргументам иметь отношение к любому типу. Рассмотрим следующий пример:

```
class Black
{
    public static void Hole(params object[] paramList)
    ...
}
```

Я назвал этот метод `Black.Hole` (черная дыра), потому что им поглощаются любые аргументы.

- Методу можно вообще не передавать никакие аргументы, в случае чего компилятор передаст `object`-массив, имеющий длину 0:

```
Black.Hole();
// преобразуется в Black.Hole(new object[0]);
```

- Метод `Black.Hole` можно вызвать, передав ему в качестве аргумента `null`. Массив относится к ссылочному типу, поэтому инициализировать его значением `null` не запрещается:

```
Black.Hole(null);
```

- Методу `Black.Hole` можно передать уже существующий массив. Иными словами, массив, который обычно создается компилятором, можно создать вручную:

```
object[] array = new object[2];
array[0] = "forty two";
array[1] = 42;
Black.Hole(array);
```

- Методу `Black.Hole` можно передать аргументы разных типов, и эти аргументы будут автоматически помещены в `object`-массив:

```
Black.Hole("forty two", 42);
//преобразуется в Black.Hole(new object[]{"forty two", 42});
```

МЕТОД CONSOLE.WRITELINE

Класс `Console` содержит множество перегружаемых версий метода `WriteLine`. Одна из таких версий имеет следующий вид:

```
public static void WriteLine(string format, params object[] args);
```

Хотя строковая интерполяция практически сделала эту версию метода `WriteLine` излишней, в предыдущих редакциях языка C# она использовалась довольно часто. Эта перегрузка позволяла методу `WriteLine` поддерживать аргумент форматирования строки, содержащий числовые поля для заполнения, каждое из которых могло заменяться в ходе выполнения программы переменной любого типа, указанной в списке параметров (поле для замены `{i}` заменялось `i`-той переменной, указываемой в последующем списке). Рассмотрим пример вызова этого метода (переменные `fname` и `lname` являются строками, `mi` относится к типу `char`, а `age` – к `int`):

```
Console.WriteLine("Forename:{0}, Middle Initial:{1}, Last name:{2},
Age:{3}", fname, mi, lname, age);
```

Компилятор превращает этот вызов в следующий:

```
Console.WriteLine("Forename:{0}, Middle Initial:{1}, Last name:{2},
Age:{3}", new object[4]{fname, mi, lname, age});
```

Использование params-массива

В следующем упражнении будет создан и протестирован статический метод по имени `Sum`. Он предназначен для вычисления суммы переданного ему переменного количества `int`-аргументов и возвращения результата в виде `int`-значения. Делаться это будет путем создания `Sum`, принимающего параметр `params int[]`. Будут также выполнены две проверки параметра `params`, чтобы убедиться, что метод `Sum` абсолютно надежен. Затем с целью тестирования метод `Sum` будет вызываться с разным количеством различных аргументов.

Создание метода, использующего params-массив

Откройте в среде Microsoft Visual Studio 2015 проект `ParamsArray`, который находится в папке `\Microsoft Press\VCSBS\Chapter 11\ParamsArray` вашей папки документов.

В проекте `ParamsArray` в файле `Program.cs` содержится класс `Program`, включающий в себя метод среди `doWork`, уже встречавшийся в предыдущих главах. Метод `Sum` будет создан как статический метод другого класса по имени `Util` (сокращение от слова `utility`), который вы добавите к проекту.

Щелкните правой кнопкой мыши в окне обозревателя решений на проекте `ParamsArray`, который находится в решении `ParamsArray`, найдите пункт **Добавить** (Add), а затем щелкните на пункте **Класс** (Class).

В средней панели диалогового окна **Добавить новый элемент** – `ParamsArray` щелкните на пункте Класс. В поле Имя (Name) наберите строку `Util.cs`, а затем щелкните на кнопке **Добавить** (Add). В результате этого будет создан и добавлен к проекту файл `Util.cs`. В нем содержится пустой класс по имени `Util`, находящийся в пространстве имен `ParamsArray`.

Добавьте к классу `Util` открытый статический метод по имени `Sum`. Этот метод должен возвращать `int`-значение и принимать `params`-массив из `int`-значений по имени `paramList`. Он должен иметь следующий вид:

```
public static int Sum(params int[] paramList)
{
}
```

Первым шагом на пути создания метода `Sum` станет проверка параметра `paramList`. Кроме того что он должен содержать допустимый набор целых чисел, он также может содержать значение `null` или быть массивом нулевой длины. В обоих этих случаях суммы вычислить сложно, поэтому лучше будет выдать исключение `ArgumentException`. (Конечно, можно согласиться с тем, что сумма целых чисел в массиве нулевой длины равна нулю, но в данном примере такая ситуация будет расцениваться как исключительная.)

Добавьте к методу `Sum` код, выделенный жирным шрифтом. Этот код выдает исключение `ArgumentException`, если `paramList` имеет значение `null`. Теперь метод `Sum` должен приобрести следующий вид:

```
public static int Sum(params int[] paramList)
{
    if (paramList == null)
    {
        throw new ArgumentException("Util.Sum: null parameter list");
    }
}
```

Добавьте к методу `Sum` код, выделенный жирным шрифтом и предназначенный для выдачи исключения `ArgumentException`, если длина списка параметров равна нулю:

```
public static int Sum(params int[] paramList)
{
    if (paramList == null)
    {
        throw new ArgumentException("Util.Sum: null parameter list");
    }
    if (paramList.Length == 0)
    {
        throw new ArgumentException("Util.Sum: empty parameter list");
    }
}
```

Если массив пройдет эти два теста, то нужно будет сложить все элементы массива. Для этого можно воспользоваться инструкцией `foreach`; кроме этого, понадобится локальная переменная для хранения возрастающего общего значения.

Сразу же после кода, созданного на предыдущих этапах, объявите целочисленную переменную по имени `sumTotal` и инициализируйте ее значением 0:

```
public static int Sum(params int[] paramList)
{
    ...
    if (paramList.Length == 0)
    {
        throw new ArgumentException("Util.Sum: empty parameter list");
    }

    int sumTotal = 0;
}
```

Добавьте к методу `Sum` инструкцию `foreach`, позволяющую получить последовательный доступ к элементам массива `paramList`. В теле этого цикла `foreach` нужно прибавить значение каждого элемента массива к переменной `sumTotal`. В конце метода нужно путем использования инструкции `return` возвратить значение переменной `sumTotal`. Все новые добавления выделены жирным шрифтом:

```
public static int Sum(params int[] paramList)
{
    ...
    int sumTotal = 0;
    foreach (int i in paramList)
    {
        sumTotal += i;
    }
    return sumTotal;
}
```

Щелкните в меню Сборка на пункте Собрать решение, после чего убедитесь, что сборка прошла без ошибок.

Тестирование метода Util.Sum

Выведите в окно редактора файл Program.cs и замените в методе doWork комментарий // TODO: следующей инструкцией:

```
Console.WriteLine(Util.Sum(null));
```

Щелкните в меню Отладка на пункте Начать отладку. Произойдет сборка и запуск программы, после чего на консоль будет выведено следующее сообщение:

```
Exception: Util.Sum: null parameter list
```

Тем самым вы получите подтверждение работоспособности первой имеющейся в методе проверки. Нажмите Ввод, чтобы закрыть программу и вернуться в среду Visual Studio 2015.

Замените в окне редактора вызов Console.WriteLine, находящийся в методе doWork, следующим вызовом:

```
Console.WriteLine(Util.Sum());
```

На этот раз метод вызывается без аргументов. Компилятор транслирует пустой список аргументов в пустой массив. Щелкните в меню Отладка на пункте Запуск без отладки. Произойдет сборка и запуск программы, после чего на консоль будет выведено следующее сообщение:

```
Exception: Util.Sum: empty parameter list
```

Тем самым вы получите подтверждение работоспособности второй имеющейся в методе проверки. Нажмите Ввод, чтобы закрыть программу и вернуться в среду Visual Studio 2015.

Замените в окне редактора вызов Console.WriteLine, находящийся в методе doWork, следующим вызовом:

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

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что программа прошла сборку, запуск и вывела на экран консоли значение 55.

Нажмите Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

Сравнение массивов параметров с необязательными параметрами

В главе 3 «Создание методов и применение областей видимости» было показано, как определяются методы, получающие необязательные параметры. На первый взгляд может показаться, что функционально методы, использующие массивы параметров, и методы, получающие необязательные параметры, чем-то перекрывают друг друга. Но между ними есть принципиальные различия.

- ❑ Метод, получающий дополнительные параметры, все же располагает фиксированным списком параметров, и передать ему произвольный список аргументов невозможно. Компилятор создает код, который перед запуском метода вставляет в стек значения по умолчанию для любого пропущенного аргумента, и метод остается в неведении о том, какие аргументы предоставлены вызывающим его кодом, а какие создаются компилятором из значений, предоставляемых по умолчанию.
- ❑ Метод, эффективно использующий массив параметров, имеет совершенно произвольный список параметров, и ни один из них не имеет значения по умолчанию. Более того, метод может с абсолютной точностью определить, сколько именно аргументов предоставил вызывающий его код.

Обычно массивы параметров используются для методов, способных принять любое количество параметров (включая и их полное отсутствие), а необязательные параметры используются, только если вызывающий код неудобно заставлять заниматься предоставлением аргумента для каждого параметра.

Стоит задуматься и еще об одной ситуации. Если определяется метод, получающий список параметров, и предоставляется его перегруженная версия, получающая необязательные параметры, то не всегда очевидно, какая из версий будет вызвана, если список аргументов в вызывающей инструкции подходит для обеих сигнатур метода. Этот сценарий будет исследован в заключительном упражнении данной главы.

Сравнение массива параметров и необязательных параметров

Вернитесь в среду Visual Studio 2015 к решению ParamsArray и выведите в окно редактора файл Util.cs. Добавьте к началу метода Sum в классе Util показанную жирным шрифтом инструкцию Console.WriteLine:

```
public static int Sum(params int[] paramList)
{
    Console.WriteLine("Using parameter list");
    ...
}
```

Добавьте к классу `Util` еще одну реализацию метода `Sum`. Эта версия будет получать четыре необязательных `int`-параметра, для каждого из которых по умолчанию устанавливается нулевое значение. В теле метода присутствует инструкция вывода сообщения «Using optional parameters» («Использование необязательных параметров»), после чего вычисляется и возвращается сумма четырех параметров. Окончательный вариант этого метода выделен в следующем примере жирным шрифтом:

```
class Util
{
    ...
    public static int Sum(int param1 = 0, int param2 = 0, int param3 = 0,
    int param4 = 0)
    {
        Console.WriteLine("Using optional parameters");
        int sumTotal = param1 + param2 + param3 + param4;
        return sumTotal;
    }
}
```

Выполните в окне редактора файл `Program.cs`. Закомментируйте в методе `doWork` имеющийся код и добавьте к методу следующую инструкцию:

```
Console.WriteLine(Util.Sum(2, 4, 6, 8));
```

Эта инструкция вызывает метод `Sum`, передавая ему четыре `int`-параметра. Этот вызов подходит обоим перегружаемым вариантам метода `Sum`.

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запустить программы, после чего на консоль будут выведены следующие сообщения:

```
Using optional parameters
20
```

В данном случае компилятор создал код, который вызывает метод, принимающий четыре необязательных параметра. Эта версия метода наилучшим образом подходит к вызову метода. Нажмите Ввод и вернитесь в среду Visual Studio.

Измените в методе `doWork` инструкцию, вызывающую метод `Sum`, удалив последний аргумент (8):

```
Console.WriteLine(Util.Sum(2, 4, 6));
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запуск приложения, после чего на консоль будут выведены следующие сообщения:

Using optional parameters
12

Компилятор по-прежнему создает код, который вызывает метод, принимающий четыре необязательных параметра, даже при том что его сигнатура не в полной мере подходит к вызову. Когда компилятор C# стоит перед выбором между методом, принимающим необязательные параметры, и методом, принимающим список параметров, он отдает предпочтение первому варианту. Нажмите Ввод и вернитесь в среду Visual Studio.

Еще раз измените в методе `doWork` ту инструкцию, которая вызывает метод `Sum`, добавив к ней еще два дополнительных аргумента:

```
Console.WriteLine(Util.Sum(2, 4, 6, 8, 10));
```

Щелкните в меню Отладка на пункте Запуск без отладки, чтобы выполнить сборку и запуск приложения, после чего на консоль будут выведены следующие сообщения:

Using parameter list
30

На этот раз предоставлено больше аргументов, чем указано в методе, принимающем необязательные параметры, поэтому компилятор создает код, который вызывает метод, принимающий массив параметров. Нажмите Ввод и вернитесь в среду Visual Studio.

Выводы

В этой главе вы узнали, как используются массивы параметров для определения метода, который может принимать любое количество аргументов. Вы также увидели, как используется `params`-массив элементов типа `object` для создания метода, принимающего любое количество аргументов любого типа. Кроме того, вы увидели, как компилятор разбирается с вызовами метода, когда у него есть выбор между вызовом варианта метода, принимающего массив параметров, и вызовом варианта метода, принимающего необязательные параметры.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 12 «Работа с наследованием».

Если вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Создать метод, принимающий любое количество аргументов заданного типа	Создайте метод, чьим параметром будет params-массив заданного типа. Например, метод, принимающий любое количество bool-аргументов, объявляется следующим образом: <pre>someType Method(params bool[] flags) { ... }</pre>
Создать метод, принимающий любое количество аргументов любого типа	Создайте метод, чьим параметром будет params-массив, с элементами типа object, например: <pre>someType Method(params object[] paramList) { ... }</pre>

12 Работа с наследованием

Прочитав эту главу, вы научитесь:

- создавать производный класс, наследующий свойства базового класса;
- управлять скрытием и перегрузкой методов с помощью ключевых слов `new`, `virtual` и `override`;
- ограничивать доступность в иерархии наследования путем использования ключевого слова `protected`;
- определять методы расширения в качестве механизма, альтернативного использованию наследования.

В мире объектно-ориентированного программирования наследование является ключевым приемом. Его можно использовать в качестве инструмента, позволяющего избегать дублирования кода при определении различных классов, имеющих ряд общих свойств и вполне конкретно связанных друг с другом. Возможно, это разные классы одного и того же типа, у каждого из которых имеются собственные отличительные признаки, например классы менеджеров, разнорабочих и всех заводских работников. Если создается приложение для моделирования предприятия, то как указать на то, что у этих менеджеров и разнорабочих имеется ряд одинаковых свойств, но имеются и свойства, отличающие их друг от друга? Например, у них у всех есть регистрационный номер работника, но обязанности и выполняемые задачи менеджеров отличаются от таковых у разнорабочих.

Именно здесь и пригодится наследование.

Что такое наследование?

Если спросить у нескольких опытных программистов, что такое наследование, они дадут весьма противоречивые ответы. Частично неразбериха возникает из-за того, что само слово «наследование» имеет несколько слегка отличающихся друг от друга значений. Если кто-нибудь что-нибудь вам завещает, то

говорится, что вы это наследуете. Наряду с этим говорится, что вы наследуете половину своих генов от матери, а вторую половину — от отца. Оба значения этого слова не имеют практически ничего общего с тем, что подразумевается под наследованием в программировании.

Наследование в мире программирования, по сути, является классификацией, то есть имеет отношение к связи между классами. Например, в школе вы, возможно, изучали млекопитающих и вас учили тому, что к ним относятся как лошади, так и киты. У тех и у других есть все признаки млекопитающих (они дышат атмосферным воздухом, в младенчестве питаются молоком матери, они теплокровные и т. д.), но у каждого из них есть также свои собственные особые черты (у лошади — копыта, а у кита — плавники).

Как создать в программе модель лошади и кита? Один из вариантов предполагает создание различных классов с именами `Horse` (лошадь) и `Whale` (кит). В каждом классе можно реализовать своего рода уникальное поведение, свойственное данному типу млекопитающего, например способность передвигаться рысью — `Trot` (для лошади) или плавать — `Swim` (для кита). Но как справиться с поведением, общим для лошади и кита, например дыханием — `Breathe` или питанием материнским молоком — `SuckleYoung`? К обоим классам можно добавить продублированные методы с этими именами, но это существенно затруднит сопровождение программы, особенно если вы решите создать модели и для других типов млекопитающих, например человека — `Human` и трубкузуба — `Aardvark`.

Чтобы справиться с этими задачами, в C# можно воспользоваться наследованием классов. Лошадь, кит, человек и трубкузуб являются разновидностями млекопитающих, поэтому можно создать класс по имени `Mammal` (млекопитающее), предоставляющий функциональные особенности, характерные для всех этих типов. Затем можно объявить, что классы `Horse`, `Whale`, `Human` и `Aardvark` наследуют свои свойства от класса `Mammal`. Эти классы автоматически включают в себя функциональные возможности класса `Mammal` (`Breathe`, `SuckleYoung` и т. д.), но можно дополнить каждый класс функциональными возможностями, уникальными для конкретного типа млекопитающего: методом `Trot` для класса `Horse` и методом `Swim` для класса `Whale`. Если потребуется изменить способ работы такого общего метода, как `Breathe` (дыхание), это нужно будет сделать только в одном месте — в классе `Mammal`.

Использование наследования

Объявление факта наследования, получаемого одним классом от другого класса, осуществляется с помощью следующего синтаксиса:

```
class ПроизводныйКласс : БазовыйКласс
{
    ...
}
```

Производный класс получает наследуемое от базового класса, и методы базового класса становятся частью производного класса. В C# классом, из которого разрешается создавать производный класс, является главным образом один базовый класс. Класс не разрешается создавать в качестве производного от двух и более классов. Но исключая те случаи, когда *ПроизводныйКласс* объявлен запечатанным, точно такой же синтаксис можно применять для создания других производных классов, наследующих свойства *ПроизводногоКласса* (запечатанные классы будут рассматриваться в главе 13 «Создание интерфейсов и определение абстрактных классов»):

```
class ПроизводныйПодКласс : ПроизводноКласс
{
    ...
}
```

Продолжая работу с ранее рассмотренным примером, класс *Mammal*, в котором содержатся общие для всех млекопитающих методы *Breathe* и *SuckleYoung*, можно объявить следующим образом:

```
class Mammal
{
    public void Breathe()
    {
        ...
    }
    public void SuckleYoung()
    {
        ...
    }
    ...
}
```

Затем, как показано в следующем примере, можно определить классы для каждого отдельного вида млекопитающего, добавляя по мере необходимости дополнительные методы:

```
class Horse : Mammal
{
    ...
    public void Trot()
    {
        ...
    }
}
class Whale : Mammal
{
    ...
    public void Swim()
    {
        ...
    }
}
```



ПРИМЕЧАНИЕ При наличии опыта программирования на C++ можно было заметить, что наследование тут не указывается явным образом, да и не может указываться открытым, закрытым или защищенным. В C# наследование всегда подразумевается открытым. Если вы знакомы с Java, обратите внимание на использование двоеточия и отсутствие ключевого слова `extends`.

При создании в приложении объекта `Horse` можно вызывать методы `Trot`, `Breathe` и `SuckleYoung`:

```
 Horse myHorse = new Horse();
myHorse.Trot();
myHorse.Breathe();
myHorse.SuckleYoung();
```

Можно также создать объект `Whale`, но на этот раз можно будет вызвать методы `Swim`, `Breathe` и `SuckleYoung`. Метод `Trot` будет недоступен, потому что он определен только для класса `Horse`.



ВНИМАНИЕ Наследование применяется только к классам, но не к структурам. Вы не можете определить свою собственную иерархию наследования, работая со структурами, и не можете определить структуру, являющуюся производной класса или другой структуры.

Все структуры фактически являются производными от абстрактного класса по имени `System.ValueType`. (Абстрактные классы рассматриваются в главе 13.) Это просто особенности реализации способа, с помощью которого среда Microsoft .NET Framework определяет общее поведение для типов значений, основанных на использовании стека. Непосредственное использование `ValueType` в ваших собственных приложениях маловероятно.

Повторное обращение к классу `System.Object`

Класс `System.Object` является корневым классом всех остальных классов. Подразумевается, что все классы являются производными от `System.Object`. Поэтому компилятор C# по умолчанию переписывает класс `Mammal` в следующий код (который, если в этом есть реальная потребность, можно записать явным образом):

```
 class Mammal : System.Object
{
    ...
}
```

Любые методы в классе `System.Object` автоматически передаются вниз по цепочке наследования тем классам, которые являются производными от `Mammal`, например классам `Horse` и `Whale`. На практике это означает, что все определяемые вами классы автоматически наследуют все свойства класса

`System.Object`. К ним относятся и такие методы, как `ToString` (эти методы рассматривались в главе 2 «Работа с переменными, операторами и выражениями»), используемые для преобразования объекта в строку, обычно с целью вывода информации на экран.

Вызов конструкторов базового класса

В дополнение к наследуемым методам производный класс автоматически содержит все поля из базового класса. Обычно при создании объекта эти поля требуют инициализации. Как правило, инициализация такого рода выполняется в конструкторе. Вспомним, что все классы имеют как минимум один конструктор. (Если вы не предоставляете конструктор сами, компилятор создает для вас пассивный конструктор.)

Полезно, чтобы в качестве составной части инициализации конструктор в производном классе вызывал конструктор своего базового класса, что позволит конструктору базового класса выполнить требующуюся ему дополнительную инициализацию. Для вызова конструктора базового класса можно при определении конструктора наследующего класса указать ключевое слово `base`:

```
class Mammal // базовый класс
{
    public Mammal(string name) // конструктор для базового класса
    {
        ...
    }
    ...
}

class Horse : Mammal // производный класс
{
    public Horse(string name)
        : base(name) // вызывает Mammal(name)
    {
        ...
    }
    ...
}
```

Если в конструкторе производного класса нет явного указания на вызов конструктора базового класса, компилятор перед выполнением кода в конструкторе производного класса пытается по умолчанию вставить вызов пассивного конструктора базового класса. Если взять ранее показанный пример, то компилятор переписывает код

```
class Horse : Mammal
{
    public Horse(string name)
```

```
{  
    ...  
}  
...  
}
```

в следующий код:

```
class Horse : Mammal  
{  
    public Horse(string name)  
        : base()  
    {  
        ...  
    }  
    ...  
}
```

Этот код работает в том случае, если у класса `Mammal` имеется открытый пассивный конструктор. Но такой конструктор есть не у всех классов (к примеру, стоит вспомнить, что компилятор создает пассивный конструктор, только если вами не созданы какие-либо активные конструкторы), и если в таком случае забыть вызвать правильный конструктор базового класса, это приведет к ошибке в ходе компиляции.

Присваивание классов

В предыдущих примерах этой книги было показано, как объявляется переменная путем использования типа класса и как для создания объекта используется ключевое слово `new`. Приводились также примеры того, как правила проверки типов C# не позволяли вам присваивать объект одного типа переменной, объявленной с указанием другого типа. Например, при показанных здесь определениях классов `Mammal`, `Horse` и `Whale` код, следующий за этими определениями, будет некорректным:

```
class Mammal  
{  
    ...  
}  
class Horse : Mammal  
{  
    ...  
}  
class Whale : Mammal  
{  
    ...  
}  
...  
Horse myHorse = new Horse(...);  
Whale myWhale = myHorse; // ошибка – разные типы
```

Но можно сослаться на объект из переменной другого типа при условии, что используемый тип является классом, находящимся выше по иерархии наследования. Поэтому следующие инструкции вполне корректны:

```
Horse myHorse = new Horse(...);
Mammal myMammal = myHorse; // допустимо, Mammal – базовый класс для Horse
```

Если мыслить логически, все лошади (`Horse`) являются млекопитающими (`Mammal`), поэтому можно вполне свободно присвоить объект типа `Horse` переменной типа `Mammal`. Иерархия наследования означает, что о `Horse` можно думать просто как о специализированном типе `Mammal`: у этого объекта есть все, что есть у `Mammal`, плюс к этому имеются и дополнения, определяемые любыми методами и полями, добавленными вами к классу `Horse`. Можно также сделать так, чтобы `Mammal`-переменная ссылалась на `Whale`-объект. Но есть одно существенное ограничение: при ссылке на `Horse`- или `Whale`-объект с использованием `Mammal`-переменной доступ можно получить только к методам и полям, определенным в классе `Mammal`. Любые дополнительные методы, определенные в классе `Horse` или `Whale`, увидеть из класса `Mammal` невозможно:

```
Horse myHorse = new Horse(...);
Mammal myMammal = myHorse;
myMammal.Breathe(); // все в порядке - Breathe является частью класса Mammal
myMammal.Trot(); // ошибка - Trot не является частью класса Mammal
```



ПРИМЕЧАНИЕ Ранее уже объяснялось, почему `object`-переменной можно присвоить практически всё. Вспомним, что `object` является псевдонимом для `System.Object` и все классы наследуются из `System.Object` либо напрямую, либо опосредованно.

Следует понимать, что обратного действия это правило не имеет. Вы не можете просто так присвоить `Mammal`-объект `Horse`-переменной:

```
Mammal myMammal = newMammal(...);
Horse myHorse = myMammal; // ошибка
```

Это ограничение может показаться несколько странным, но вспомним, что не все `Mammal`-объекты, моделирующие млекопитающих, относятся к лошадям (`Horse`), некоторые могут относиться и к китам (`Whale`). `Mammal`-объект можно присвоить `Horse`-переменной только в том случае, если сначала с помощью оператора `as` или `or` удастся проверить, что млекопитающее действительно относится к лошади, или же после приведения типа с использованием ключевого слова `cast` (операторы `as` и `or`, а также приведение типов рассматриваются в главе 7 «Создание классов и объектов и управление ими»). В следующем примере оператор `as` используется для проверки того, что переменная `myMammal` ссылается на `Horse`-объект, и если проверка пройдет успешно, присваивание значения переменной `myHorseAgain` приведет к тому, что она будет ссылаться на

тот же самый `Horse`-объект. Если же переменная `myMammal` ссылается на какой-либо другой тип млекопитающего (`Mammal`), оператор `as` вместо этого приведет к тому, что будет возвращено значение `null`:

```
Horse myHorse = new Horse(...);
Mammal myMammal = myHorse; // myMammal ссылается на Horse
...
Horse myHorseAgain = myMammal as Horse; // порядок: myMammal ссылалась на Horse
...
Whale myWhale = new Whale(...);
myMammal = myWhale;
...
myHorseAgain = myMammal as Horse; // возвращает null - myMammal ссылалась на Whale
```

Объявление новых методов

Одной из наиболее трудных задач в компьютерном программировании является придумывание уникальных и в то же время имеющих определенный смысл имен для идентификаторов. Если определяется метод для класса и этот класс является частью иерархии наследования, то рано или поздно вы вознамеритесь снова применить имя, которое уже используется одним из классов, стоящих выше в структуре иерархии. Если в базовом классе и производном классе объявляются два метода, имеющих одинаковую сигнатуру, то при компиляции приложения вам будет выдано предупреждение.



ПРИМЕЧАНИЕ К сигнатуре метода относятся имя метода, а также количество и типы его параметров, но не тип его возвращаемого значения. Два метода с одинаковыми именами, получающие один и тот же список параметров, имеют одинаковую сигнатуру, даже если возвращают разные типы.

Метод в производном классе маскирует (или скрывает) метод в базовом классе, имеющий такую же сигнатуру. Например, если откомпилировать следующий код, компилятор выдаст предупреждающее сообщение о том, что `Horse.Talk` загораживает унаследованный метод `Mammal.Talk`:

```
class Mammal
{
    ...
    public void Talk() // предполагается, что все млекопитающие могут издавать
                      // звуки
}

class Horse : Mammal
{
    ...
}
```

```
public void Talk() // лошади издают не такие звуки, как другие млекопитающие!
{
    ...
}
```

Хотя ваш код будет откомпилирован и запущен, к данному предупреждению нужно отнестись всерьез. Если другой класс станет производным от `Horse` и вызовет метод `Talk`, ожидания могут быть связаны с вызовом того метода, который реализован в классе `Mammal`. Но метод `Talk` в классе `Horse` загораживает метод `Talk` в классе `Mammal`, и вместо него будет вызван метод `Horse.Talk`. Такое совпадение в лучшем случае создает путаницу, и во избежание накладок лучше рассмотреть возможность переименования методов.

Но если есть уверенность в необходимости наличия двух методов с одинаковой сигнатурой, позволяющей загородить метод `Mammal.Talk`, предупреждение можно заглушить, воспользовавшись для этого ключевым словом `new`:

```
class Mammal
{
    ...
    public void Talk()
    {
        ...
    }
}

class Horse : Mammal
{
    ...
    new public void Talk()
    {
        ...
    }
}
```

Такое использование ключевого слова `new` не разобщает эти два метода, и загораживание по-прежнему происходит. Отключается только предупреждение. По сути, ключевое слово `new` как бы говорит следующее: «Я знаю, что делаю, поэтому прекращай показывать мне эти предупреждения».

Объявление виртуальных методов

Иногда способ реализации метода в базовом классе нужно скрыть. Рассмотрим в качестве примера метод `ToString` в `System.Object`. Назначение метода `ToString` заключается в преобразовании объекта в его строковое представление. Наличие метода в классе `System.Object` объясняется высокой востребованностью этого метода, поэтому класс автоматически предоставляет его всем классам. Но

откуда версия метода `ToString`, реализованная в классе `System.Object`, знает, как преобразовать в строку экземпляр производного класса? В производном классе может быть любое количество полей с интересующими вас значениями, которые могут стать частью строки. Ответ заключается в простоте реализации `ToString` в `System.Object`. Все, на что он способен, — это преобразовать объект в строку, содержащую имя его типа, например «`Mammal`» или «`Horse`». Но пользы от этого мало. Тогда зачем предоставлять почти бесполезный метод? Ответ на этот вопрос требует некоторых размышлений.

Очевидно, что, в принципе, идея наличия метода `ToString` неплоха и все классы должны предоставлять метод, который можно было бы использовать для преобразования объектов в строки для их отображения на экране или выполнения отладки. Внимания требует лишь сама реализация. Фактически вы не рассчитываете на вызов метода `ToString`, определенного классом `System.Object`, и он является всего лишь прототипом. Скорее всего, вам придется предоставить собственную версию метода `ToString` в каждом определяемом вами классе, перегружая его исходную реализацию, имеющуюся в `System.Object`. Версия в `System.Object` является всего лишь своеобразной страховкой на случай, если в классе не реализована своя собственная, особая версия метода `ToString` или если она ему и вовсе не требуется.

Метод, предназначенный для перегрузки, называется *виртуальным* методом. Разницу между перегрузкой метода и его скрытием нужно прояснить. Перегрузка метода является механизмом для предоставления различных реализаций одного и того же метода, все методы являются родственными, поскольку предназначены для решения одной и той же задачи, но способом, специфичным для того или иного класса, а скрытие метода является средством замены одного метода другим, при этом методы обычно не являются родственными и могут выполнять абсолютно разные задачи. Перегрузка метода является полезным понятием, относящимся к программированию, а скрытие метода зачастую является результатом ошибки.

Метод можно пометить как виртуальный, воспользовавшись для этого ключевым словом `virtual`. Например, метод `ToString` в классе `System.Object` определен следующим образом:

```
namespace System
{
    class Object
    {
        public virtual string ToString()
        {
            ...
        }
        ...
    }
    ...
}
```



ПРИМЕЧАНИЕ Если вам приходилось программировать на Java, то вы могли заметить, что изначально методы в C# не являются виртуальными.

Объявление методов с помощью ключевого слова `override`

Если в базовом классе метод объявлен виртуальным, то в производном классе для объявления другой реализации метода можно воспользоваться ключевым словом `override`:

```
class Horse : Mammal
{
    ...
    public override string ToString()
    {
        ...
    }
}
```

Новая реализация метода в производном классе может вызвать исходную реализацию метода в базовом классе, для чего нужно воспользоваться ключевым словом `base`:

```
public override string ToString()
{
    string temp = base.ToString();
    ...
}
```

При объявлении полиморфных методов, рассматриваемых во врезке «Виртуальные методы и полиморфизм», нужно следовать некоторым важным правилам использования ключевых слов `virtual` и `override`.

- ❑ Виртуальный метод не может быть закрытым, поскольку он предназначен для доступа к нему посредством механизма наследования со стороны других классов. Не могут быть закрытыми и переопределенные методы, потому что изменить уровень защиты наследуемого метода класс не может. Но в следующем разделе будет показано, что у переопределенных методов может быть специальная форма закрытости, известная как *зашщщенный доступ*.
- ❑ Сигнатуры виртуальных и переопределенных методов должны быть идентичными: у них должны быть одинаковые имена, а также количество и типы параметров.
- ❑ Перегрузить можно только виртуальный метод. Если метод базового класса не является виртуальным и вы попытаетесь его перегрузить, в ходе компиляции будет выдана ошибка. Это вполне резонно, поскольку решать, какие из методов могут быть перегружены, — прерогатива проектировщика базового класса.

- ❑ Если в производном классе метод объявлен без использования ключевого слова `override`, метод базового класса не перегружается, а скрывается. Иными словами, получается реализация совершенно другого метода с точно таким же именем. Как и ранее, в результате этого во время компиляции будет выдано предупреждение, от которого можно избавиться, воспользовавшись, как уже было показано, ключевым словом `new`.
- ❑ Переопределенный метод является потенциально виртуальным и сам может быть перегружен в последующих производных классах. Но объявить явным образом переопределенный метод виртуальным, воспользовавшись для этого ключевым словом `virtual`, нельзя.

ВИРТУАЛЬНЫЕ МЕТОДЫ И ПОЛИМОРФИЗМ

Используя виртуальные методы, можно вызывать различные версии одного и того же метода, основываясь на типе объекта, который определяется динамически в ходе выполнения программы. Рассмотрим следующие примеры классов, определяющих один из вариантов ранее рассмотренной иерархии `Mammal`:

```
class Mammal
{
    ...
    public virtual string GetTypeName()
    {
        return "This is a mammal" ;
    }
}

class Horse : Mammal
{
    ...
    public override string GetTypeName()
    {
        return "This is a horse";
    }
}
class Whale : Mammal
{
    ...
    public override string GetTypeName()
    {
        return "This is a whale";
    }
}
class Aardvark : Mammal
{
    ...
}
```

Здесь следует отметить две особенности: во-первых, ключевое слово `override`, используемое при определении метода `GetTypeName` в классах `Horse` и `Whale`, и во-вторых, тот факт, что в классе `Aardvark` нет метода `GetTypeName`.

А теперь изучим следующий блок кода:

```
Mammal myMammal;
Horse myHorse = new Horse(...);
Whale myWhale = new Whale(...);
Aardvark myAardvark = new Aardvark(...);
myMammal = myHorse;
Console.WriteLine(myMammal.GetTypeName()); // Horse
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName()); // Whale
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName()); // Aardvark
```

Что будут выводить на экран консоли три различные инструкции `Console.WriteLine`? На первый взгляд от всех них следует ожидать вывода на экран строки «`This is a mammal`», поскольку в каждой инструкции метод `GetTypeName` вызывается в отношении переменной `myMammal`, имеющей тип `Mammal`. Но в первом случае можно увидеть, что `myMammal` фактически ссылается на `Horse`. (Вспомним, что `Horse`-переменную можно присвоить `Mammal`-переменной, потому что класс `Horse` наследуется из класса `Mammal`.) Поскольку метод `GetTypeName` определен в качестве виртуального, среда выполнения определяет, что нужно вызвать метод `Horse.GetTypeName`, поэтому инструкция выведет на экран сообщение «`This is a horse`». Точно такая же логика применяется ко второй инструкции `Console.WriteLine`, которая выводит сообщение «`This is a whale`». Третья инструкция вызывает метод `Console.WriteLine` в отношении объекта `Aardvark`. Но в классе `Aardvark` нет метода `GetTypeName`, поэтому вызывается исходный метод, находящийся в классе `Mammal` и возвращающий строку «`This is a mammal`».

Этот механизм вызова с помощью одной и той же инструкции различных методов в зависимости от имеющегося контекста называется полиморфизмом, что буквально означает «множество форм».

Основные сведения о защищенном доступе

С помощью ключевых слов `public` и `private` создаются две крайние степени доступности: открытые поля и методы класса доступны отовсюду, а закрытые поля и методы доступны только внутри самого класса.

С точки зрения изолированности классов этих двух крайностей вполне достаточно. Но как известно всем опытным программистам, пользующимся объектно-ориентированными технологиями, решать сложные проблемы с помощью изолированных классов невозможно. Эффективным способом связи классов является наследование, позволяющее понятным образом получить специализированные и тесные взаимоотношения между производным классом и его базовым классом. Зачастую базовому классу полезно будет разрешить

производным классам получать доступ к некоторым своим компонентам, одновременно закрывая к ним доступ со стороны классов, не являющихся частью иерархии наследования. В такой ситуации эти компоненты можно пометить ключевым словом `protected`. Данный механизм работает следующим образом.

- ❑ Если класс А является производным другого класса Б, он может получать доступ к защищенным составляющим класса Б. Иными словами, внутри производного класса А защищенная составляющая класса Б фактически является открытой.
- ❑ Если класс А не является производным класса Б, он не может получать доступ к защищенным составляющим класса Б. Таким образом, внутри класса А защищенная составляющая класса Б фактически является закрытой.

Язык C# дает программистам полную свободу в объявлении методов и полей защищенными. Но в большинстве руководств по объектно-ориентированному программированию рекомендуется содержать поля в строго закрытом состоянии везде, где это возможно, и ослаблять это ограничение, только когда в этом возникнет абсолютная необходимость. Открытые поля нарушают инкапсуляцию, поскольку все пользователи класса получают непосредственный неограниченный доступ к полям. Защищенные поля сохраняют инкапсуляцию для тех пользователей класса, которым эти поля недоступны. Но защищенные поля все же позволяют нарушать инкапсуляцию другими классами, наследуемыми из базового класса.



ПРИМЕЧАНИЕ Доступ к защищенной составляющей базового класса можно получить не только из производного класса, но и из классов, являющихся производными по отношению к данному производному классу.

В следующем упражнении будет определена простая иерархия классов для моделирования различных типов транспортных средств. Будут определены базовый класс по имени `Vehicle` и производные классы с именами `Airplane` и `Car`. В классе `Vehicle` будут также определены общие методы с именами `StartEngine` и `StopEngine`, означающими запуск и остановку двигателя соответственно, а в оба производных класса будут добавлены методы, специфичные для этих классов. И наконец, к классу `Vehicle` будет добавлен виртуальный метод по имени `Drive`, а в оба производных класса будут добавлены методы, переопределяющие исходную реализацию этого метода.

Создание иерархии классов

Откройте в среде Microsoft Visual Studio 2015 проект `Vehicles`, который находится в папке `\Microsoft Press\VCBS\Chapter 12\Vehicles` вашей папки документов. В проекте `Vehicles` содержится файл `Program.cs`, в котором определяется класс `Program` с методами `Main` и `doWork`, уже встречавшимися в предыдущих упражнениях.

Щелкните правой кнопкой мыши в обозревателе решений на проекте Vehicles, укажите на пункт Добавить, а затем щелкните на пункте Класс. Откроется диалоговое окно Добавить новый элемент, в котором нужно убедиться в выделении шаблона Класс. Наберите в поле Имя строку Vehicle.cs и щелкните на кнопке Добавить. Будет создан и добавлен к проекту файл Vehicle.cs, код которого появится в окне редактора. В файле содержится определение пустого класса по имени Vehicle.

Добавьте к классу Vehicle методы StartEngine и StopEngine, выделенные далее жирным шрифтом:

```
class Vehicle
{
    public void StartEngine(string noiseToMakeWhenStarting)
    {
        Console.WriteLine($"Starting engine: {noiseToMakeWhenStarting}");
    }

    public void StopEngine(string noiseToMakeWhenStopping)
    {
        Console.WriteLine($"Stopping engine: {noiseToMakeWhenStopping}");
    }
}
```

Эти методы будут унаследованы всеми классами, являющимися производными от класса Vehicle. Значения для параметров noiseToMakeWhenStarting (шум при запуске) и noiseToMakeWhenStopping (шум при глушении) для каждого типа транспортного средства будут разными и помогут вам позже идентифицировать, какое транспортное средство запускается или глушится.

Щелкните в меню Проект на пункте Добавить класс. На экране снова откроется диалоговое окно Добавить новый элемент. Наберите в поле Имя строку Airplane.cs и щелкните на кнопке Добавить. К проекту будет добавлен новый файл, содержащий класс по имени Airplane, код которого появится в окне редактора. Внесите в определение класса Airplane изменение, выделенное здесь жирным шрифтом и показывающее, что он наследуется из класса Vehicle:

```
class Airplane : Vehicle
{
}
```

Добавьте к классу Airplane методы TakeOff (взлет) и Land (приземление), выделенные здесь жирным шрифтом:

```
class Airplane : Vehicle
{
    public void TakeOff()
    {
        Console.WriteLine("Taking off");
    }
}
```

```
}

public void Land()
{
    Console.WriteLine("Landing");
}
}
```

Щелкните в меню Project на пункте Добавить класс. На экране снова появится диалоговое окно Добавить новый элемент. Наберите в поле Имя строку Car.cs и щелкните на кнопке Добавить. К проекту будет добавлен новый файл, содержащий класс по имени Car, код которого появится в окне редактора. Внесите в определение класса Car изменение, выделенное здесь жирным шрифтом и показывающее, что он наследуется из класса Vehicle:

```
class Car : Vehicle
{
}
```

Добавьте к классу Car методы Accelerate (газ) и Brake (тормоз), выделенные жирным шрифтом:

```
class Car : Vehicle
{
    public void Accelerate()
    {
        Console.WriteLine("Accelerating");
    }

    public void Brake()
    {
        Console.WriteLine("Braking");
    }
}
```

Выведите в окно редактора файл Vehicle.cs. Добавьте к классу Vehicle виртуальный метод Drive, выделенный здесь жирным шрифтом:

```
class Vehicle
{
    ...
    public virtual void Drive()
    {
        Console.WriteLine("Default implementation of the Drive method");
    }
}
```

Выведите в окно редактора файл Program.cs. Удалите в методе doWork комментарий // TODO: и добавьте код, который создает экземпляр класса Airplane и тестирует работу его методов, имитируя быстрое путешествие на самолете:

```
static void doWork()
{
    Console.WriteLine("Journey by airplane:");
    Airplane myPlane = new Airplane();
    myPlane.StartEngine("Contact");
    myPlane.TakeOff();
    myPlane.Drive();
    myPlane.Land();
    myPlane.StopEngine("Whirr");
}
```

Добавьте сразу же после только что добавленного к методу `doWork` кода следующие инструкции, выделенные жирным шрифтом. Они создадут экземпляр класса `Car` и протестируют работу его методов.

```
static void doWork()
{
    ...
    Console.WriteLine("\nJourney by car:");
    Car myCar = new Car();
    myCar.StartEngine("Brm brm");
    myCar.Accelerate();
    myCar.Drive();
    myCar.Brake();
    myCar.StopEngine("Phut phut");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь, что в окне консоли эта программа выводит сообщения, имитирующие различные стадии путешествия на самолете и на машине (рис. 12.1).

Обратите внимание на то, что оба транспортных средства вызывают исходную реализацию виртуального метода `Drive`, поскольку ни один из классов пока этот метод не перегружает.

```
C:\WINDOWS\system32\cmd.exe
Journey by airplane:
Starting engine: Contact
Taking off
Default implementation of the Drive method
Landing
Stopping engine: Whirr

Journey by car:
Starting engine: brm brm
Accelerating
Default implementation of the Drive method
Braking
Stopping engine: phut phut
Для продолжения нажмите любую клавишу . . .
```

Рис. 12.1

Нажмите Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

Выведите в окно редактора класс `Airplane`. Создайте в классе `Airplane` переопределенный метод `Drive`, выделенный здесь жирным шрифтом:

```
class Airplane : Vehicle
{
    ...
    public override void Drive()
    {
        Console.WriteLine("Flying");
    }
}
```



ПРИМЕЧАНИЕ Система IntelliSense выведет список доступных виртуальных методов. Если выбрать из списка IntelliSense метод `Drive`, Visual Studio автоматически вставит в ваш код инструкцию, вызывающую метод `base.Drive`. В таком случае удалите инструкцию, поскольку в данном упражнении она не требуется.

Выведите в окно редактора класс `Car`. Создайте в классе `Car` переопределенный метод `Drive`, выделенный здесь жирным шрифтом:

```
class Car : Vehicle
{
    ...
    public override void Drive()
    {
        Console.WriteLine("Motoring");
    }
}
```

Щелкните в меню Отладка на пункте Запуск без отладки. Обратите внимание на то, что теперь при вызове приложения объект `Airplane` выводит в окне консоли сообщение `Flying`, а объект `Car` выводит сообщение `Motoring` (рис. 12.2).

```
C:\WINDOWS\system32\cmd.exe
Journey by airplane:
Starting engine: Contact
Taking off
Flying
Landing
Stopping engine: Whirr

Journey by car:
Starting engine: Brm brm
Accelerating
Motoring
Braking
Stopping engine: Phut phut
Для продолжения нажмите любую клавишу . . .
```

Рис. 12.2

Нажмите Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

Выполните в окно редактора файл Program.cs. Добавьте к методу doWork инструкции, выделенные здесь жирным шрифтом:

```
static void doWork()
{
    ...
    Console.WriteLine("\nTesting polymorphism");
    Vehicle v = myCar;
    v.Drive();
    v = myPlane;
    v.Drive();
}
```

Этот код тестирует полиморфность, возникающую при использовании виртуального метода Drive. В нем путем использования Vehicle-переменной создается ссылка на объект Car (что вполне допустимо, поскольку все Car-объекты являются Vehicle-объектами), а затем с использованием Vehicle-переменной вызывается метод Drive. Последние две инструкции создают ссылку Vehicle-переменной на объект Airplane и совершают еще один вызов, похожий на очередной вызов того же самого метода Drive.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что в окне консоли появились те же самые сообщения, что и прежде, а за ними был выведен следующий текст (рис. 12.3):

```
Testing polymorphism
Motoring
Flying
```

```
C:\WINDOWS\system32\cmd.exe
Journey by airplane:
Starting engine: contact
Taking off
Flying
Landing
Stopping engine: whirr

Journey by car:
Starting engine: brm brm
Accelerating
Motoring
Braking
Stopping engine: phut phut

Testing polymorphism
Motoring
Flying
Для продолжения нажмите любую клавишу . . .
```

Рис. 12.3

Метод `Drive` является виртуальным, поэтому среда выполнения (но не компилятор) определяет, какую из версий метода `Drive` нужно вызвать в отношении ссылающейся на него `Vehicle`-переменной, на основе реального типа объекта, на который ссылается эта переменная. В первом случае `Vehicle`-объект ссылается на `Car`, поэтому приложение вызывает метод `Car.Drive`. Во втором случае `Vehicle`-объект ссылается на `Airplane`, поэтому приложение вызывает метод `Airplane.Drive`.

Нажмите `Ввод`, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

Основные сведения о методах расширения

Наследование является весьма эффективным свойством, позволяющим расширять функциональные возможности класса путем создания нового, производного класса. Но иногда использование наследования не является наиболее подходящим механизмом для добавления новых вариантов поведения, особенно если нужно быстро расширить тип, не затрагивая при этом существующий код.

Представим, что вам нужно добавить новую функцию к типу `int` путем создания метода по имени `Negate`, возвращающего эквивалент значения, уже имеющегося у целочисленной переменной, но с отрицательным знаком. (Понятно, что для выполнения подобной задачи можно просто воспользоваться оператором унарного минуса (`-`), но потерпите меня еще немного.) Один из способов достижения поставленной цели заключается в определении нового типа по имени `NegInt32`, наследуемого из `System.Int32` (`int` является псевдонимом для `System.Int32`), и добавлении метода `Negate`:

```
class NegInt32 : System.Int32 // не пытайтесь сделать это!
{
    public int Negate()
    {
        ...
    }
}
```

Теоретически в дополнение к методу `Negate` класс `NegInt32` унаследует все функциональные возможности, связанные с типом `System.Int32`. Но отказаться от применения этого подхода можно по двум причинам.

- ❑ Этот метод применяется только к типу `NegInt32`, и если потребуется воспользоваться им с существующими в вашем коде `int`-переменными, придется изменить определение каждой `int`-переменной на тип `NegInt32`.
- ❑ Тип `System.Int32` фактически является структурой, а не классом, а применить наследование в отношении структуры невозможно.

Именно здесь и пригодятся методы расширения.

Использование метода расширения позволяет расширить существующий тип (класс или структуру) дополнительными статическими методами. Эти статические методы тут же становятся доступными вашему коду в любых инструкциях, ссылающихся на данные расширяемого типа.

Методы расширения определяются в статическом классе и в качестве первого параметра метода вместе с ключевым словом `this` определяют тип, к которому применяется метод. Следующий пример показывает, как можно реализовать метод расширения `Negate` для типа `int`:

```
static class Util
{
    public static int Negate(this int i)
    {
        return -i;
    }
}
```

Синтаксис выглядит немного странно, но на его принадлежность к методу расширения указывает ключевое слово `this`, стоящее перед параметром метода `Negate`, а принадлежность параметра, перед которым стоит `this`, к типу `int` означает, что расширению подвергается тип `int`.

Чтобы воспользоваться методом расширения, нужно ввести класс `Util` в область видимости. (Если необходимо, добавьте инструкцию `using`, указывающую на то пространство имен, которому принадлежит класс `Util`, или инструкцию `using static`, указывающую непосредственно на класс `Util`.) Затем можно будет для ссылки на метод воспользоваться обычной формой записи с использованием символа точки (`.`):

```
int x = 591;
Console.WriteLine($"x.Negate {x.Negate()}");
```

Обратите внимание на то, что ссылаться на класс `Util` где-либо в инструкции, вызывающей метод `Negate`, не нужно. Компилятор C# автоматически обнаружит все методы расширения для заданного типа из всех статических классов, находящихся в области видимости. Можно также вызвать метод `Util.Negate`, передав ему в качестве параметра `int`-значение и воспользовавшись уже встречавшимся ранее обычным синтаксисом, но тогда было бы незачем определять этот метод в качестве метода расширения:

```
int x = 591;
Console.WriteLine($"x.Negate {Util.Negate(x)}");
```

В следующем упражнении метод расширения будет добавлен к типу `int`. С помощью этого метода расширения можно будет преобразовать значение

`int`-переменной с основанием 10 в его представление в системе счисления с другим основанием.

Создание метода расширения

Откройте в среде Visual Studio 2015 проект `ExtensionMethod`, который находится в папке `\Microsoft Press\VCSBS\Chapter 12\ExtensionMethod` вашей папки документов.

Выполните в окне редактора файл `Util.cs`. В этом файле содержится статический класс по имени `Util`, находящийся в пространстве имен `Extensions`. Не забудьте, что методы расширения нужно определять в статическом классе. В этом классе не содержится ничего, кроме комментария `// TODO:`.

Удалите комментарий и объягите в классе `Util` открытый статический метод по имени `ConvertToBase`. Этот метод должен принимать два параметра: `int`-параметр по имени `i`, перед которым будет указано ключевое слово `this`, свидетельствующее о том, что это метод расширения для типа `int`, и еще один обычный `int`-параметр по имени `baseToConvertTo` (основание счисления для преобразования).

Этот метод будет заниматься преобразованием значения, содержащегося в `i`, в число по основанию, которое указано в параметре `baseToConvertTo`. Метод должен возвратить `int`-значение с конвертированной величиной.

Метод `ConvertToBase` должен иметь следующий вид:

```
static class Util
{
    public static int ConvertToBase(this int i, int baseToConvertTo)
    {
    }
}
```

Добавьте к методу `ConvertToBase` инструкцию `if`, проверяющую, что значение параметра `baseToConvertTo` находится между 2 и 10.

Алгоритм, используемый для данного упражнения, за пределами этого диапазона значений не работает. Если значение параметра `baseToConvertTo` выходит за пределы этого диапазона, выдайте исключение с соответствующим сообщением.

Метод `ConvertToBase` должен приобрести следующий вид:

```
public static int ConvertToBase(this int i, int baseToConvertTo)
{
    if (baseToConvertTo < 2 || baseToConvertTo > 10)
    {
        throw new ArgumentException("Value cannot be converted to base " +
            baseToConvertTo.ToString());
    }
}
```

Добавьте к методу `ConvertToBase` после инструкции, выдающей исключение `ArgumentException`, инструкции, выделенные далее жирным шрифтом.

В этом коде реализуется широко известный алгоритм, преобразующий число по основанию 10 в число с другим основанием системы счисления. (Версия этого алгоритма для преобразования десятичного числа в восьмеричное уже была представлена в главе 5 «Использование инструкций составного присваивания и итераций».)

```
public static int ConvertToBase(this int i, int baseToConvertTo)
{
    ...
    int result = 0;
    int iterations = 0;
    do
    {
        int nextDigit = i % baseToConvertTo;
        i /= baseToConvertTo;
        result += nextDigit * (int)Math.Pow(10, iterations);
        iterations++;
    }
    while (i != 0);

    return result;
}
```

Выполните в окне редактора файл `Program.cs`. После инструкции `using System;`, находящейся в начале файла, добавьте инструкцию `using`:

```
using Extensions;
```

Эта инструкция вводит пространство имен, содержащее класс `UTIL`, в область видимости. Если этого не сделать, то в файле `Program.cs` метод расширения `ConvertToBase` виден не будет.

Добавьте к методу `doWork` класса `Program` инструкции, выделенные здесь жирным шрифтом, заменив ими комментарий `// TODO:`:

```
static void doWork()
{
    int x = 591;
    for (int i = 2; i <= 10; i++)
    {
        Console.WriteLine($"{x} in base {i} is {x.ConvertToBase(i)}");
    }
}
```

Этот код создает `int`-переменную по имени `x` и присваивает ей значение `591`. (Можете выбрать любое другое понравившееся вам целочисленное значение.) Затем в коде используется цикл для вывода на экран значения `591` во всех видах

с основанием систем счисления между 2 и 10 (рис. 12.4). Обратите внимание на то, что `ConvertToBase` появляется в качестве метода расширения в системе IntelliSense, как только вы в инструкции `Console.WriteLine` наберете после `x` символ точки (.)).

```

Program.cs* + X Util.cs*
ExtensionMethod
ExtensionMethod
using System;
using Extensions;
namespace ExtensionMethod
{
    class Program
    {
        static void doWork()
        {
            int x = 591;
            for (int i = 2; i <= 10; i++)
            {
                Console.WriteLine($"{x} in base {i} is {x.ConvertToBase(i)}");
            }
        }

        static void Main()
        {
            try
            {
                doWork();
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception: {0}", ex.Message);
            }
        }
    }
}

```

Рис. 12.4

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь, что программа выводит на консоль сообщения, показывающие значение 591 в различных системах счисления (рис. 12.5).

```

cmd C:\WINDOWS\system32\cmd.exe
591 in base 2 is 1001001111
591 in base 3 is 210220
591 in base 4 is 21033
591 in base 5 is 4331
591 in base 6 is 2423
591 in base 7 is 1503
591 in base 8 is 1117
591 in base 9 is 726
591 in base 10 is 591
Для продолжения нажмите любую клавишу . . .

```

Рис. 12.5

Нажмите Ввод, чтобы закрыть программу и вернуться в среду Visual Studio 2015.

Выводы

В этой главе вы узнали, как наследование используется для определения иерархии классов, и теперь должны уже разбираться в том, как перегружаются унаследованные методы и реализуются виртуальные методы. Вы также научились добавлять к существующему типу методы расширения.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 13.

Если вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Создать производный класс из базового класса	Объявите новое имя класса, после него поставьте двоеточие и укажите имя базового класса, например: <pre>class DerivedClass : BaseClass { ... }</pre>
Вызвать конструктор базового класса в качестве части конструктора для производного класса	Укажите в определении конструктора перед телом конструктора производного класса суффикс в виде вызова <code>base</code> и предоставьте базовому конструктору любые необходимые ему параметры, например: <pre>class DerivedClass : BaseClass { ... public DerivedClass(int x) : base(x) { ... } ... }</pre>

Чтобы	Сделайте следующее
Объявить виртуальный метод	<p>Воспользуйтесь при объявлении метода ключевым словом <code>virtual</code>, например:</p> <pre data-bbox="450 318 833 521"><code>class Mammal { public virtual void Breathe() { ... } ... }</code></pre>
Реализовать в производном классе метод, переопределяющий унаследованный виртуальный метод	<p>Воспользуйтесь при объявлении метода ключевым словом <code>override</code>, например:</p> <pre data-bbox="450 618 833 821"><code>class Whale : Mammal { public override void Breathe() { ... } ... }</code></pre>
Определить для типа метод расширения	<p>Добавьте к статическому классу статический открытый метод. Первым параметром должен быть расширяемый тип, перед которым ставится ключевое слово <code>this</code>, например:</p> <pre data-bbox="450 935 912 1120"><code>static class Util { public static int Negate(this int i) { return -i; } }</code></pre>

13

Создание интерфейсов и определение абстрактных классов

Прочитав эту главу, вы научитесь:

- определять интерфейс, указывая сигнатуры и возвращаемые методами типы;
- реализовывать интерфейс в структуре или классе;
- ссылаться на класс через интерфейс;
- фиксировать общие детали реализации в абстрактном классе;
- реализовывать запечатанные классы, которые не могут использоваться для создания на их основе производных классов.

Наследование, осуществляемое из класса, является весьма эффективным механизмом, но реальная эффективность наследования проявляется при наследовании из интерфейса. Интерфейс не содержит никакого кода и никаких данных, в нем только определяются методы и свойства, которые должен предоставить класс, получающий наследование из интерфейса. Используя интерфейс, можно полностью отделить имена и сигнатуры методов класса от реализации методов.

Абстрактные классы во многом похожи на интерфейсы, за исключением того, что они могут содержать код и данные. Но конкретные методы абстрактного класса можно определить как виртуальные, чтобы класс, получающий наследование от абстрактного класса, мог дополнительно предоставить свою собственную реализацию этих методов. Абстрактные классы часто используются с интерфейсами, и вместе они предоставляют ключевую технологию, с помощью которой, как вы увидите в этой главе, можно создавать расширяемые среды программирования.

Основные сведения об интерфейсах

Предположим, что вам нужно определить новый класс, в котором можно хранить коллекцию объектов примерно так же, как и при использовании массива. Но, в отличие от массива, вам нужно предоставить метод по имени `RetrieveInOrder`, чтобы приложение могло извлекать объекты в последовательности, зависящей от типа объекта, содержащегося в коллекции. (При использовании обычного массива можно обойти все его содержимое, а по умолчанию элементы извлекаются по их индексам.) Если в коллекции содержатся буквенно-цифровые объекты, например строки, коллекция должна позволить приложению извлекать эти строки в последовательности, соответствующей последовательности упорядочения их компьютером, а если коллекция содержит числовые объекты, например целые числа, она должна позволить приложению извлекать эти объекты в цифровой последовательности.

При определении класса коллекции вам не хочется накладывать ограничение на типы объектов, которые могут в ней содержаться (объекты могут даже иметь типы классов или структур), следовательно, вы не будете знать, как упорядочить эти объекты. Как же тогда можно будет снабдить класс коллекции методом, сортирующим объекты тех типов, о которых вам ничего не известно при написании этого класса коллекции? На первый взгляд эта задача похожа на задачу с методом `ToString`, рассмотренную в главе 12 «Работа с наследованием», которая может быть решена путем объявления виртуального метода, который в свою очередь может быть переопределен подклассами вашего класса коллекции. Но любое сходство вводит в заблуждение. Отношений наследования между классом коллекции и содержащимися в нем объектами нет, поэтому пользы от виртуального метода не будет. Если вдуматься, то задача относится не к самой коллекции, а к способу упорядочения объектов в коллекции в зависимости от типа имеющегося в ней объекта. Решение заключается в требовании, чтобы всеми объектами предоставлялся метод, подобный методу `CompareTo`, показанному в следующем примере кода, который может вызываться методом коллекции `RetrieveInOrder`, открывая для коллекции возможность сравнивать объекты друг с другом:

```
int CompareTo(object obj)
{
    // возвращает нулевое значение, если данный экземпляр равен obj
    // возвращает значение меньше нуля, если данный экземпляр меньше, чем obj
    // возвращает значение больше нуля, если данный экземпляр больше, чем obj
    ...
}
```

Для собираемых в коллекцию объектов, включающих метод `CompareTo`, вы можете определить интерфейс и указать, что класс коллекции может содержать только классы, которые реализуют этот интерфейс. Таким образом, интерфейс аналогичен договору. Если класс реализует интерфейс, этот интерфейс

гарантирует, что в классе содержатся все методы, указанные в интерфейсе. Этот механизм гарантирует возможность вызова метода `CompareTo` в отношении всех объектов в коллекции и их сортировки.

Используя интерфейсы, можно реально отделить понятие «что» от понятия «как». Интерфейс дает вам только имя, тип возвращаемого значения и параметры метода. А конкретная реализация метода в задачу интерфейса не входит. В интерфейсе дается описание той функциональной возможности, которую должен предоставить класс, но не способ ее реализации.

Определение интерфейса

Определение интерфейса синтаксически похоже на определение класса, за исключением того, что вместо ключевого слова `class` используется ключевое слово `interface`. Внутри интерфейса, точно так же, как внутри класса или структуры, объявляются методы, за исключением того, что для них никогда не указывается спецификатор доступа (`public`, `private` или `protected`). Кроме того, методы в интерфейсе не имеют реализаций — они представляют собой простое объявление, и все типы, реализующие интерфейс, должны предоставлять для методов свои собственные реализации. Соответственно тело метода заменяется точкой с запятой, например:

```
interface IComparable
{
    int CompareTo(object obj);
}
```



СОВЕТ В документации по среде Microsoft .NET рекомендуется давать интерфейсам имена, начинающиеся с прописной буквы I. Это соглашение является пережитком венгерской нотации в C#. Кстати, только что показанный интерфейс `IComparable` уже определен в пространстве имен `System`.

Интерфейс не может содержать данные, то есть вы не можете добавлять к интерфейсу поля, даже закрытые.

Реализация интерфейса

Чтобы реализовать интерфейс, нужно объявить класс или структуру, наследующую интерфейс, а затем реализовать все указанные интерфейсом методы. Это не наследование как таковое, хотя синтаксис точно такой же, и как вы увидите далее в этой главе, некоторые семантические элементы имеют многие отличительные признаки наследования. Следует заметить, что, в отличие от наследования класса, интерфейс может быть реализован в структуре.

Предположим, к примеру, что вы определяете `Mammal`-иерархию, рассмотренную в главе 12, но при этом вам нужно указать млекопитающих, передвигающихся по суше (`land-bound`), предоставив метод по имени `NumberOfLegs`, который возвращает `int`-значение с количеством имеющихся у млекопитающего ног (или лап). (Этот интерфейс у передвигающихся по суше млекопитающих не реализован.) Интерфейс `ILandBound`, в котором содержится этот метод, можно определить следующим образом:

```
interface ILandBound
{
    int NumberOfLegs();
}
```

Затем интерфейс можно реализовать в классе `Horse`. При наследовании интерфейса каждому определенному в нем методу предоставляется реализация (в данном случае имеется только один метод `NumberOfLegs`):

```
class Horse : ILandBound
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

При реализации интерфейса нужно обеспечить точное совпадение каждого метода с соответствующим методом в интерфейсе, придерживаясь следующих правил:

- ❑ Имена и типы возвращаемых значений должны точно совпадать.
- ❑ Все параметры, включая модификаторы `ref` и `out`, должны точно совпадать.
- ❑ Все методы, реализующие интерфейс, должны находиться в открытом доступе. Но если используется явная реализация интерфейса, метод не должен иметь спецификатор доступа.

Если между интерфейсным определением и объявленной реализацией будут какие-нибудь различия, класс не пройдет компиляцию.



СОВЕТ Интегрированная среда разработки (IDE) Microsoft Visual Studio может способствовать сокращению числа ошибок в программе, вызванных неподобающей реализацией методов, объявленных в интерфейсе. Мастер реализации интерфейсов `Implement Interface` может создавать заглушки для каждого элемента интерфейса, реализуемого классом. Затем эти заглушки могут наполняться соответствующим кодом. Порядок использования этого мастера будет показан в следующих упражнениях.

Класс может быть наследником другого класса и наряду с этим реализовывать интерфейс. В таком случае в C# не делается различий между базовым классом

и интерфейсом путем использования специального ключевого слова `as`, как это, к примеру, делается в Java. Вместо этого в C# используется позиционная система записи. Сначала всегда указывается имя базового класса, затем стоит запятая, после которой указывается интерфейс. В следующем примере определяется класс `Horse`, являющийся производным от класса `Mammal`, но дополнительно еще реализующий интерфейс `ILandBound`:

```
interface ILandBound
{
    ...
}

class Mammal
{
    ...
}

class Horse : Mammal , ILandBound
{
    ...
}
```



ПРИМЕЧАНИЕ Интерфейс `InterfaceA` может быть наследником другого интерфейса, `InterfaceB`. С технической точки зрения это называется не наследованием, а расширением интерфейса. В данном случае любой класс или структура, реализующие `InterfaceA`, должны предоставлять реализацию всех методов, имеющихся как в `InterfaceB`, так и в `InterfaceA`.

Ссылка на класс через его интерфейс

Точно так же, как вы можете сослаться на объект путем использования переменной, определенной как класс, стоящий выше в иерархии, вы можете сослаться на объект путем использования переменной, определенной как интерфейс, реализуемый классом объекта. Если взять предыдущий пример, то на `Horse`-объект можно сослаться с помощью `ILandBound`-переменной:

```
Horse myHorse = new Horse(...);
ILandBound iMyHorse = myHorse; // это вполне допустимо
```

Ссылка работает, поскольку все лошади (`horses`) являются млекопитающими, передвигающимися по суше (`land-bound mammals`), хотя обратное утверждение неверно — вы не можете присвоить `ILandBound`-объект `Horse`-переменной без предварительного приведения к типу, чтобы проверить, что она действительно ссылается на `Horse`-объект, а не на какой-нибудь другой класс, в котором также реализуется интерфейс `ILandBound`.

Польза от ссылки на объект через интерфейс заключается в том, что появляется возможность использования ее для определения методов, способных принимать

в качестве параметров различные типы, при условии, что в типах реализован указанный интерфейс. Например, показанный далее метод `FindLandSpeed` может принять любой аргумент, реализующий интерфейс `ILandBound`:

```
int FindLandSpeed(ILandBound landBoundMammal)
{
    ...
}
```

Вы можете проверить, что объект является экземпляром класса, реализующего конкретный интерфейс, воспользовавшись оператором `is`, показанным в главе 8 «Основные сведения о значениях и ссылках». Оператор `is` используется для определения того, имеет ли объект указанный тип, и работает с интерфейсами точно так же, как с классами и структурами. Например, следующий блок кода перед тем, как присвоить переменную `myHorse` переменной типа `ILandBound`, проверяет факт реализации в переменной `myHorse` интерфейса `ILandBound`:

```
if (myHorse is ILandBound)
{
    ILandBound iLandBoundAnimal = myHorse;
}
```

Учтите, что при ссылке на объект через интерфейс можно вызвать только те методы, которые видны через этот интерфейс.

Работа с несколькими интерфейсами

У класса может быть не более одного базового класса, но в нем допускается реализация неограниченного количества интерфейсов. Класс должен реализовать все методы, объявленные этими интерфейсами.

Если в структуре или классе реализуется более одного интерфейса, то интерфейсы указываются в списке через запятую. Если у класса имеется также базовый класс, интерфейсы перечисляются после базового класса. Предположим, к примеру, что вы определили еще один интерфейс по имени `IGrazable` (травоядные), содержащий метод `ChewGrass` (жевать траву) для всех травоядных животных. Тогда класс `Horse` можно определить следующим образом:

```
class Horse : Mammal, ILandBound, IGrazable
{
    ...
}
```

Явная реализация интерфейса

Приводимые до сих пор примеры показывали классы, реализующие интерфейс неявным образом. Если еще раз посмотреть на интерфейс `ILandBound` и на класс

`Horse`, показанные далее, то можно увидеть: хотя класс `Horse` реализует интерфейс `ILandBound`, ничего в реализации метода `NumberOfLegs` не говорит о том, что он является частью интерфейса `ILandBound`:

```
interface ILandBound
{
    int NumberOfLegs();
}

class Horse : ILandBound
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

В простой ситуации все может обойтись без каких-либо осложнений, но представим себе, что класс `Horse` реализует сразу несколько интерфейсов. Ничто не мешает указать метод точно с таким же именем, хотя у методов может быть различная семантика. Предположим, к примеру, что вам нужно реализовать транспортную систему на основе конных экипажей. Длинный путь может быть разбит на несколько этапов, обозначаемых термином `leg`. Если нужно отслеживать, сколько этапов каждая лошадь тянула за собой экипаж, может быть определен следующий интерфейс:

```
interface IJourney
{
    int NumberOfLegs();
}
```

Теперь, если реализовать этот интерфейс в классе `Horse`, возникнет весьма интересная проблема:

```
class Horse : ILandBound, IJourney
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

Это вполне допустимый код, но что он означает — что у лошади четыре ноги или что она тянула за собой экипаж четыре этапа пути? Ответ таков: и то и это! Изначально C# не различает, какой из интерфейсов реализует данный метод, поэтому один и тот же метод фактически реализует оба интерфейса.

Для решения данной проблемы и устранения неоднозначности при определении того, какой из методов является частью реализации того или иного интерфейса, можно реализовать интерфейс явным образом. Для этого при реализации метода нужно указать, какому интерфейсу он принадлежит:

```
class Horse : ILandBound, IJourney
{
    ...
    int ILandBound.NumberOfLegs()
    {
        return 4;
    }
    int IJourney.NumberOfLegs()
    {
        return 3;
    }
}
```

Теперь можно увидеть, что у лошади четыре ноги и она тянула за собой экипаж на протяжении трех этапов пути.

Кроме указания в имени метода префикса в виде имени интерфейса, в этом синтаксисе присутствует еще одна особенность: методы не помечаются как открытые (public). Для методов, являющихся частью явной реализации интерфейса, можно указать защиту. Это приводит к еще одному весьма интересному явлению. Если создать `Horse`-переменную в коде, то вы фактически не можете вызвать ни один из методов `NumberOfLegs`, поскольку они вне области видимости. Дело в том, что класс `Horse` считает их оба закрытыми. В этом есть вполне определенный смысл. Если бы методы были видны через класс `Horse`, то какой из методов будет вызван следующим кодом: тот, что реализован для интерфейса `ILandBound`, или тот, что предназначен для интерфейса `IJourney`?

```
Horse horse = new Horse();
...
// Следующая инструкция не будет скомпилирована
int legs = horse.NumberOfLegs();
```

Так как же получить доступ к этим методам? Нужно сослаться на `Horse`-объект через соответствующий интерфейс:

```
Horse horse = new Horse();
...
IJourney journeyHorse = horse;
int legsInJourney = journeyHorse.NumberOfLegs();
ILandBound landBoundHorse = horse;
int legsOnHorse = landBoundHorse.NumberOfLegs();
```

Я рекомендую осуществлять по возможности явную реализацию интерфейсов.

Ограничения, накладываемые на интерфейсы

Важно помнить, что интерфейс никогда не содержит никакой реализации. Из этого вполне естественно вытекают следующие ограничения.

- ❑ В интерфейсе не разрешается определять какие-либо поля, даже статические. Поле является особенностью реализации класса или структуры.
- ❑ В интерфейсе не разрешается определять какие-либо конструкторы. Конструктор также рассматривается в качестве особенности реализации класса или структуры.
- ❑ В интерфейсе не разрешается определять деструктор. В деструкторе содержатся инструкции, используемые для уничтожения объекта, являющегося экземпляром класса. (Деструкторы рассматриваются в главе 14 «Использование сборщика мусора и управление ресурсами».)
- ❑ Для метода нельзя указывать модификатор доступа. Все методы в интерфейсе подразумеваются открытыми.
- ❑ В интерфейс нельзя вкладывать какие-либо типы (например, перечисления, структуры, классы или интерфейсы).
- ❑ Интерфейс не может что-либо наследовать от структуры или класса, хотя интерфейс может быть наследником другого интерфейса. В структурах и классах содержится реализация, и если бы интерфейсу разрешалось быть наследником кого-либо из них, то он унаследовал бы какую-нибудь реализацию.

Определение и использование интерфейсов

В следующих упражнениях вы дадите определение и реализуете интерфейсы, являющиеся составной частью простого графического пакета для рисования. Будут определены два интерфейса с именами `IDraw` и `IColor`, а затем вы определите классы, реализующие эти интерфейсы. В каждом классе будет определена фигура, которую можно будет нарисовать на холсте, расположеннем в форме (холст является элементом управления, которым можно воспользоваться для рисования линий, текста и фигур на экране).

В интерфейсе `IDraw` определяются следующие методы:

- ❑ `SetLocation` — с его помощью можно указать для фигуры на холсте позицию в виде координат x и y ;
- ❑ `Draw` — рисует фигуру на холсте в той позиции, которая указана с помощью метода `SetLocation`.

В интерфейсе `IColor` определяется метод `SetColor`. Он используется для указания цвета фигуры. Когда фигура рисуется на холсте, она будет появляться в этом цвете.

Определение интерфейсов `IDraw` и `IColor`

Откройте в среде Microsoft Visual Studio 2015 проект `Drawing`, который находится в папке `\Microsoft Press\VCSBS\Chapter 13\Drawing` вашей папки документов.

Проект `Drawing` является графическим приложением. В нем имеется форма по имени `DrawingPad`. В этой форме содержится элемент управления `canvas` (холст) по имени `drawingCanvas`. Эта форма и холст будут использоваться для тестирования вашего кода.

Щелкните в обозревателе решений на проекте `Drawing`. Щелкните в меню Проект на пункте Добавить новый элемент. Откроется диалоговое окно Добавить новый элемент — `Drawing`, в левой панели которого щелкните на пункте Visual C#, а затем на пункте Код. В средней панели щелкните на названии шаблона Интерфейс. Наберите в поле Имя строку `IDraw.cs`, после чего щелкните на кнопке Добавить. Среда Visual Studio создаст файл `IDraw.cs` и добавит его к вашему проекту. Содержимое `IDraw.cs` появится в окне редактора и будет иметь следующий вид:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Drawing
{
    interface IDraw
    {
    }
}
```

Добавьте к списку в самом начале файла `IDraw.cs` следующую директиву `using`:

```
using Windows.UI.Xaml.Controls;
```

В этом интерфейсе вы будете ссылаться на класс `Canvas`, который находится в пространстве имен `Windows.UI.Xaml.Controls` для приложений универсальной платформы Windows (Universal Windows Platform (UWP)).

Добавьте к определению интерфейса `IDraw` методы, выделенные жирным шрифтом:

```
interface IDraw
{
    void SetLocation(int xCoord, int yCoord);
    void Draw(Canvas canvas);
}
```

Щелкните еще раз в меню Проект на пункте Добавить новый элемент. Щелкните в средней панели появившегося диалогового окна на названии шаблона Интерфейс. Наберите в поле Имя строку `IColor.cs`, после чего щелкните на кнопке Добавить. Среда Visual Studio создаст файл `IColor.cs` и добавит его к вашему проекту. Содержимое `IColor.cs` появится в окне редактора. Добавьте к списку в самом начале файла `IColor.cs` следующую директиву `using`:

```
using Windows.UI;
```

В этом интерфейсе вы будете ссылаться на класс `Color`, который находится в пространстве имен `Windows.UI` для UWP-приложений.

Добавьте к определению интерфейса `IColor` метод, выделенный здесь жирным шрифтом:

```
interface IColor
{
    void SetColor(Color color);
}
```

Теперь вы определили интерфейсы `IDraw` и `IColor`. На следующем этапе для реализации этих интерфейсов будут созданы несколько классов. В следующем упражнении вы создадите два новых класса фигур с именами `Square` и `Circle`. Оба интерфейса будут реализованы именно в этих классах.

Создание классов `Square` и `Circle` и реализация интерфейсов

Щелкните в меню Project на пункте Добавить класс. Убедитесь, что в средней панели в диалоговом окне Добавить новый элемент — Drawing выбран шаблон Класс. В поле Имя наберите строку `Square.cs`, после чего щелкните на кнопке Добавить. Среда Visual Studio создаст файл `Square.cs` и выведет его в окно редактора.

Добавьте к списку в верхней части файла `Square.cs` следующие директивы:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

Измените определение класса `Square` (квадрат), указав выделенную жирным шрифтом настройку на реализацию интерфейсов `IDraw` и `IColor`:

```
class Square : IDraw, IColor
{
}
```

Добавьте в класс `Square` следующие закрытые переменные, выделенные здесь жирным шрифтом:

```
class Square : IDraw, IColor
{
    private int sideLength;
    private int locX = 0, locY = 0;
    private Rectangle rect = null;
}
```

В этих переменных будут храниться позиция и размер `Square`-объекта на холсте. Класс `Rectangle` находится в пространстве имен `Windows.UI.Xaml.Shapes` для UWP-приложений. Этот класс будет использоваться для рисования прямоугольника. Добавьте к классу `Square` конструктор, выделенный жирным шрифтом:

```
class Square : IDraw, IColor
{
    ...
    public Square(int sideLength)
    {
        this.sideLength = sideLength;
    }
}
```

Конструктор инициализирует поле `sideLength` и указывает каждую сторону квадрата.

В определении класса `Square` наведите указатель мыши на интерфейс `IDraw`. В появившемся контекстном меню со значком горящей лампочки щелкните на пункте Реализовать интерфейс явно (рис. 13.1).

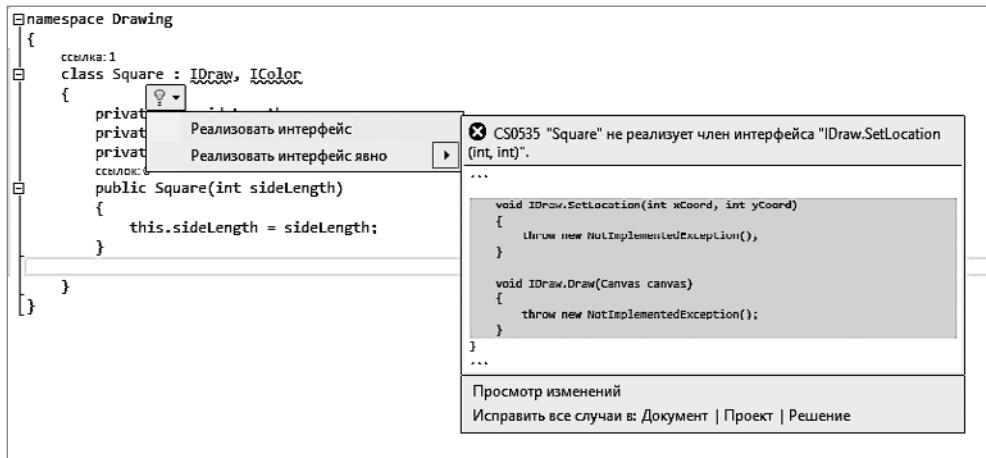


Рис. 13.1

Это заставит Visual Studio создать исходную реализацию методов в интерфейсе `IDraw`. Если есть такое желание, то методы к классу `Square` можно

добавить вручную. Код, созданный средой Visual Studio, показан в следующем примере:

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
    throw new NotImplementedException();
}

void IDraw.Draw(Canvas canvas)
{
    throw new NotImplementedException();
}
```

На данный момент каждый из этих методов выдает исключение `NotImplementedException`. Ожидается, что вы замените тела этих методов собственным кодом.

Замените код, имеющийся в методе `IDraw.SetLocation`, который выдает исключение `NotImplementedException`, следующими инструкциями, выделенными жирным шрифтом:

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
    this.locX = xCoord;
    this.locY = yCoord;
}
```

Этот код сохраняет переданные через параметры значения в полях `locX` и `locY` объекта `Square`.

Замените код, имеющийся в методе `IDraw.Draw`, следующими инструкциями, выделенными жирным шрифтом:

```
void IDraw.Draw(Canvas canvas)
{
    if (this.rect != null)
    {
        canvas.Children.Remove(this.rect);
    }
    else
    {
        this.rect = new Rectangle();
    }
    this.rect.Height = this.sideLength;
    this.rect.Width = this.sideLength;
    Canvas.SetToolTip(this.rect, this.locY);
    Canvas.SetLeft(this.rect, this.locX);
    canvas.Children.Add(this.rect);
}
```

Этот метод выводит `Square`-объект путем рисования на холсте `Rectangle`-фигуры. (Квадрат (`square`) — это просто равносторонний прямоугольник.) Если `Rectangle` уже был ранее нарисован (возможно, в другом месте и другим цветом), он

удаляется с холста. Высота (height) и ширина (width) прямоугольника `Rectangle` устанавливаются путем использования значения поля `sideLength`. Позиция `Rectangle` на холсте устанавливается с помощью статических методов `SetTop` и `SetLeft`, принадлежащих классу `Canvas`, а затем `Rectangle` добавляется к холсту. (Это приводит к его отображению на холсте.)

Добавьте к классу `Square` метод `SetColor` из интерфейса `IColor`:

```
void IColor.SetColor(Color color)
{
    if (this.rect != null)
    {
        SolidColorBrush brush = new SolidColorBrush(color);
        this.rect.Fill = brush;
    }
}
```

Этот метод проверяет, выведен ли объект `Square` на экран. (Поле `rect` будет иметь `null`-значение, если прямоугольник еще не появился на экране.) Код устанавливает для свойства `Fill` поля `rect` конкретный цвет, для чего используется `SolidColorBrush`-объект. (Подробности работы класса `SolidColorBrush` нас в данном случае не интересуют.)

Щелкните в меню Проект на пункте Добавить класс. Наберите в поле Имя появившегося диалогового окна Добавить новый элемент — Drawing строку `Circle.cs`, после чего щелкните на кнопке Добавить. Среда Visual Studio создаст файл `Circle.cs` и выведет его содержимое в окно редактора.

Добавьте к списку в верхней части файла `Circle.cs` следующие директивы `using`:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

Измените определение класса `Circle`, чтобы в нем появилось указание на реализацию интерфейсов `IDraw` и `IColor`, выделенное здесь жирным шрифтом:

```
class Circle : IDraw, IColor
{
}
```

Добавьте к классу `Circle` следующие закрытые переменные, выделенные жирным шрифтом:

```
class Circle : IDraw, IColor
{
    private int diameter;
    private int locX = 0, locY = 0;
    private Ellipse circle = null;
}
```

В этих переменных будут храниться позиция и размер **Circle**-объекта на холсте. Класс **Ellipse** предоставляет функциональные возможности, которые будут использоваться вами для рисования круга.

Добавьте к классу **Circle** конструктор, выделенный здесь жирным шрифтом:

```
class Circle : IDraw, IColor
{
    ...
    public Circle(int diameter)
    {
        this.diameter = diameter;
    }
}
```

Этот конструктор инициализирует поле **diameter**.

Добавьте к классу **Circle** следующий метод **SetLocation**:

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
    this.locX = xCoord;
    this.locY = yCoord;
}
```

Этот метод реализует часть интерфейса **IDraw** и использует такой же код, как и в классе **Square**.

Добавьте к классу **Circle** показанный далее метод **Draw**:

```
void IDraw.Draw(Canvas canvas)
{
    if (this.circle != null)
    {
        canvas.Children.Remove(this.circle);
    }
    else
    {
        this.circle = new Ellipse();
    }

    this.circle.Height = this.diameter;
    this.circle.Width = this.diameter;
    Canvas.SetTop(this.circle, this.locY);
    Canvas.SetLeft(this.circle, this.locX);
    canvas.Children.Add(this.circle);
}
```

Этот метод также является частью интерфейса **IDraw**. Он аналогичен методу **Draw** в классе **Square**, за исключением того, что выводит **Circle**-объект путем рисования на холсте **Ellipse**-фигуры. (Круг является эллипсом с одинаковыми значениями высоты и ширины.)

Добавьте к классу `Circle` следующий метод `SetColor`:

```
void IColor.SetColor(Color color)
{
    if (this.circle != null)
    {
        SolidColorBrush brush = new SolidColorBrush(color);
        this.circle.Fill = brush;
    }
}
```

Этот метод является частью интерфейса `IColor`. Как и прежде, этот метод похож на тот, что принадлежит классу `Square`.

Работа с классами `Square` и `Circle` завершена, и теперь можно воспользоваться формой для их тестирования.

Тестирование классов `Square` и `Circle`

Выполните в окне конструктора файл `DrawingPad.xaml`. Щелкните в форме на большой затененной области, которая является в ней `Canvas`-объектом, в результате чего на нем установится фокус. Щелкните в окне Свойства на кнопке Обработчики событий для выбранного элемента. (Значок на этой кнопке похож на разряд молнии.)

Найдите в списке событий `Tapped` и дважды щелкните на его текстовом поле. Среда Visual Studio создаст метод по имени `drawingCanvas_Tapped` для класса `DrawingPad` и выведет его в окно редактора. Это обработчик события, запускаемый при прикосновении пользователя пальцем к холсту или при щелчке левой кнопкой мыши, когда указатель находится над холстом. Более подробно обработчики событий рассматриваются в главе 20 «Отделение логики приложения и обработка событий».

Добавьте к списку в начале файла `DrawingPad.xaml.cs` следующую директиву `using`:

```
using Windows.UI;
```

В пространстве имен `Windows.UI` содержится определение класса `Colors`, который будет использоваться при установке цвета фигуры в ходе ее рисования.

Добавьте к методу `drawingCanvas_Tapped` следующий код, выделенный жирным шрифтом:

```
private void drawingCanvas_Tapped(object sender, TappedRoutedEventArgs e)
{
    Point mouseLocation = e.GetPosition(this.drawingCanvas);
    Square mySquare = new Square(100);

    if (mySquare is IDraw)
    {
```

```

    IDraw drawSquare = mySquare;
    drawSquare.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
    drawSquare.Draw(drawingCanvas);
}
}

```

Параметр этого метода `TappedRoutedEventArgs` предоставляет полезную информацию о позиции указателя мыши. В частности, метод `GetPosition` возвращает структуру `Point`, содержащую координаты `x` и `y` указателя мыши. Добавленный вами код создает новый `Square`-объект. Затем он проверяет, реализован ли в этом объекте интерфейс `IDraw` (это обычная практика, помогающая убедиться, что код не даст сбой во время выполнения при попытке сослаться на объект через нереализованный интерфейс), и создает ссылку на объект с использованием этого интерфейса. Следует напомнить, что при явной реализации интерфейса определенные в интерфейсе методы доступны только при создании ссылки на этот интерфейс. (Методы `SetLocation` и `Draw` по отношению к классу `Square` являются закрытыми и доступны только через интерфейс `IDraw`.) Затем код устанавливает размещение `Square` в позиции пальца пользователя или указателя мыши. Следует заметить, что координаты `x` и `y` в структуре `Point` фактически являются целочисленными значениями с двойной точностью (`double`), поэтому данный код приводит их к целочисленному типу (`int`). Затем код вызывает метод `Draw` для отображения `Square`-объекта.

Добавьте в конце метода `drawingCanvas_Tapped` следующий код, выделенный жирным шрифтом:

```

private void drawingCanvas_Tapped(object sender, TappedRoutedEventArgs e)
{
    ...
    if (mySquare is IColor)
    {
        IColor colorSquare = mySquare;
        colorSquareSetColor(Colors.BlueViolet);
    }
}

```

Этот код тестирует класс `Square` на реализацию в нем интерфейса `IColor`: если интерфейс реализован, код создает ссылку на класс `Square` через этот интерфейс и вызывает метод `SetColor` для установки цвета `Square`-объекта на `Colors.BlueViolet`.



ВНИМАНИЕ Перед тем как вызвать `SetColor`, нужно вызвать `Draw`, потому что метод `SetColor` устанавливает цвет для `Square`-объекта, только если он уже выведен на холст. Если вызвать `SetColor` до вызова `Draw`, цвет установлен не будет и `Square`-объект не появится.

Вернитесь к файлу `DrawingPad.xaml` в окне конструктора и щелкните на объекте `Canvas`.

Найдите в списке событий `RightTapped`, а затем дважды щелкните на текстовом поле с именем этого события.

Это событие возникает в случае, когда пользователь пальцем прикасается к холсту, некоторое время удерживает на нем палец, а затем убирает его с холста или же когда щелкает на холсте правой кнопкой мыши.

Добавьте к методу `drawingCanvas_RightTapped` следующий код, выделенный жирным шрифтом:

```
private void drawingCanvas_RightTapped(object sender,
                                      RightTappedRoutedEventArgs e)
{
    Point mouseLocation = e.GetPosition(this.drawingCanvas);
    Circle myCircle = new Circle(100);

    if (myCircle is IDraw)
    {
        IDraw drawCircle = myCircle;
        drawCircle.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
        drawCircle.Draw(drawingCanvas);
    }

    if (myCircle is IColor)
    {
        IColor colorCircle = myCircle;
        colorCircle.SetColor(Colors.HotPink);
    }
}
```

Логика, лежащая в основе этого кода, аналогична той логике, которая использовалась в методе `drawingCanvas_Tapped`, за исключением того, что этот код рисует и заполняет цветом круг, а не квадрат.

Щелкните в меню Отладка на пункте Начать отладку, чтобы выполнить сборку и запустить приложения.

Когда откроется окно Drawing Pad, прикоснитесь пальцем к любому месту холста, показанного в окне, или щелкните над этим местом мышью. Должен появиться фиолетовый квадрат.

Прикоснитесь к любому месту холста, некоторое время удерживайте на нем палец, после чего уберите его с холста или же щелкните правой кнопкой мыши над любым местом холста. Должен появиться розовый круг. Можно щелкать левой и правой кнопками мыши любое количество раз, и при каждом щелчке в позиции указателя мыши на холсте будет появляться квадрат или круг. На рис. 13.2 показано приложение, работающее под управлением Windows 10.

Вернитесь в среду Visual Studio и остановите отладку.

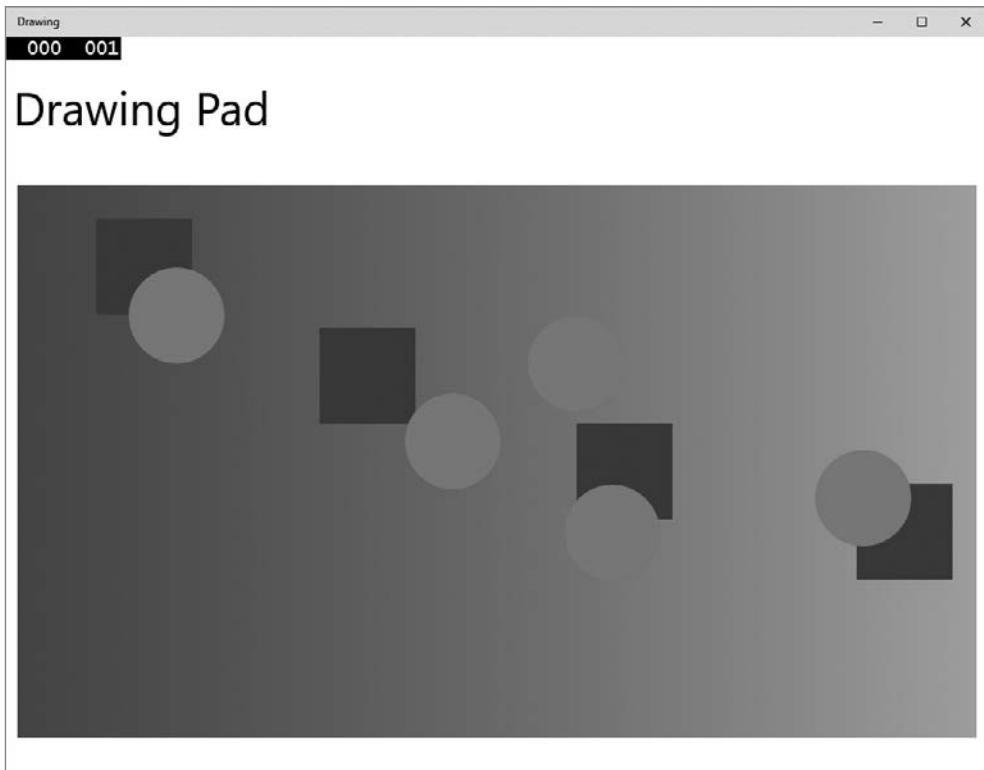


Рис. 13.2

Абстрактные классы

Интерфейсы `ILandBound` и `IGrazable`, рассмотренные в ранее представленном наборе упражнений, могут быть реализованы в множестве различных классов в зависимости от того, сколько различных типов млекопитающих нужно будет смоделировать в C#-приложении. В подобных этим ситуациях фрагменты производных классов довольно часто используют одну и ту же реализацию. Например, в следующих двух классах код явно продублирован:

```
class Horse : Mammal, ILandBound, IGrazable
{
    ...
    void IGrazable.CheatGrass()
    {
        Console.WriteLine("Chewing grass");
        // код для травоядных
    }
}
```

```
class Sheep : Mammal, ILandBound, IGrazable
{
    ...
    void IGrazable.CheatGrass()
    {
        Console.WriteLine("Chewing grass");
        // код для такого же травоядного, что и лошадь
    }
}
```

Повторение кода должно вас настораживать. Нужно по возможности реорганизовать код, избавившись от повторений и сократив затраты на его сопровождение. Один из способов выполнения этой задачи заключается в помещении общей реализации в новый класс, созданный специально для этой цели. В результате, как показано в следующем примере кода, новый класс можно будет поместить в иерархию классов:

```
class GrazingMammal : Mammal, IGrazable
{
    ...
    void IGrazable.CheatGrass()
    {
        // общий код для травоядных
        Console.WriteLine("Chewing grass");
    }
}

class Horse : GrazingMammal, ILandBound
{
    ...
}

class Sheep : GrazingMammal, ILandBound
{
    ...
}
```

Решение неплохое, но есть в нем одна загвоздка: фактически ничто не препятствует созданию экземпляров класса `GrazingMammal` (а заодно и класса `Mammal`). Но на самом деле в этом нет никакого смысла. Существование класса `GrazingMammal` обусловлено предоставлением общей исходной реализации. Он предназначен лишь для того, чтобы быть классом для наследования. Класс `GrazingMammal` является абстракцией общей функциональности, а не полноценным созданием.

Чтобы указать на невозможность создания экземпляров класса, можно объявить класс абстрактным, воспользовавшись для этого ключевым словом `abstract`:

```
abstract class GrazingMammal : Mammal, IGrazable
{
    ...
}
```

Если теперь попытаться создать в качестве его экземпляра `GrazingMammal`-объект, код не пройдет компиляцию:

```
GrazingMammal myGrazingMammal = new GrazingMammal(...); // недопустимый код
```

Абстрактные методы

В абстрактном классе могут содержаться абстрактные методы. В принципе, абстрактный метод похож на виртуальный (рассмотрен в главе 12), за исключением того, что в нем отсутствует тело метода. Этот метод должен быть перегружен производным классом. Абстрактный метод не может быть закрытым. В следующем примере в классе `GrazingMammal` в качестве абстрактного определяется метод `DigestGrass`: у классов травоядных млекопитающих должен использоваться такой же код для жевания травы, но у них должна быть собственная реализация метода `DigestGrass`. Польза от абстрактного метода просматривается в том случае, если его исходная реализации в абстрактном классе не имеет смысла, но при этом нужны гарантии того, что класс-наследник предоставит собственную реализацию этого метода.

```
abstract class GrazingMammal : Mammal, IGrazable
{
    public abstract void DigestGrass();
    ...
}
```

Запечатанные классы

Бывает, что применение наследования дается непросто и требует особой предусмотрительности. При создании интерфейса или абстрактного класса вы заранее пишете код, предназначенный для будущего наследования. Беда в том, что предсказать будущее нелегко. С приобретением опыта и практических навыков можно достичь мастерства в изготовлении гибкой, простой в использовании иерархии интерфейсов, абстрактных и обычных классов, но это требует немалых усилий, а кроме того, нужно иметь весьма четкое представление о моделируемой задаче. Иными словами, пока вы преднамеренно не спроектируете класс с принципом на использование его в качестве базового, крайне маловероятно, что он будет хорошо работать именно в этом качестве. Программируя на C#, можно воспользоваться ключевым словом `sealed` (запечатанный), воспрепятствовав тем самым использованию класса в качестве базового, если будет принято соответствующее решение, например:

```
sealed class Horse : GrazingMammal, ILandBound
{
    ...
}
```

Если какой-либо класс попытается использовать `Horse` в качестве базового класса, то во время компиляции будет выдана ошибка. Следует учесть, что в запечатанном классе невозможно объявлять какие-либо виртуальные методы и что абстрактные классы не могут быть запечатанными.

Запечатанные методы

Ключевое слово `sealed` можно также использовать для объявления того, что отдельный метод в незапечатанном классе является запечатанным. Это означает, что производный класс не может перегрузить данный метод. Запечатать можно только тот метод, который объявлен с ключевым словом `override`, и метод объявляется как `sealed override`. О ключевых словах `interface`, `virtual`, `override` и `sealed` можно составить следующее представление.

- ❑ Интерфейс вводит в обращение имя метода.
- ❑ Виртуальный метод является первой реализацией метода.
- ❑ Переопределенный метод является еще одной реализацией метода.
- ❑ Запечатанный метод является последней реализацией метода.

Реализация и использование абстрактного класса

В следующих упражнениях абстрактный класс используется для рационализации кода, разработанного в предыдущем упражнении. В классах `Square` и `Circle` содержится довольно много повторяющегося кода. Этот код практичеснее убрать в абстрактный класс по имени `DrawingShape`, поскольку это упростит сопровождение классов `Square` и `Circle` в будущем.

Создание абстрактного класса `DrawingShape`

Вернитесь в среду Visual Studio к проекту `Drawing`.



ПРИМЕЧАНИЕ Окончательный рабочий вариант кода предыдущего упражнения можно получить в проекте `Drawing`, который находится в папке `\Microsoft Press\VCSBS\Chapter 13\Drawing Using Interfaces` вашей папки документов.

Щелкните в обозревателе решений на проекте `Drawing`, находящемся в одноименном решении. Щелкните в меню Проект на пункте Добавить класс. Откроется диалоговое окно **Добавить новый элемент — Drawing**. Наберите в поле Имя строку `DrawingShape.cs`, после чего щелкните на кнопке **Добавить**. Среда Visual Studio создаст класс и выведет его в окно редактора.

Добавьте к списку в самом начале файла DrawingShape.cs следующие директивы `using`:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

Цель этого класса заключается в содержании общего кода для классов `Circle` и `Square`. Программа не должна иметь возможности непосредственно создавать экземпляра `DrawingShape`-объекта.

Измените определение класса `DrawingShape`, придав ему статус абстрактного (выделено жирным шрифтом):

```
abstract class DrawingShape
{
}
```

Добавьте к классу `DrawingShape` следующие закрытые переменные, показанные жирным шрифтом:

```
abstract class DrawingShape
{
    protected int size;
    protected int locX = 0, locY = 0;
    protected Shape = null;
}
```

Поля `locX` и `locY` для указания положения объекта на холсте используются как в классе `Square`, так и в классе `Circle`, поэтому их можно переместить в абстрактный класс. Также в обоих этих классах используются поля для указания размера объекта при его выводе на холст. Хотя в каждом классе у них разные имена (`sideLength` и `diameter`), семантически поля в них выполняют одну и ту же задачу. Хорошим обобщением назначения такого поля станет имя `size`.

Внутри класс `Square` использует для вывода фигуры на холст объект типа `Rectangle`, а класс `Circle` использует для той же цели `Ellipse`-объект. Оба этих класса являются частью иерархии, основанной на абстрактном классе `Shape`, определенном в среде .NET Framework. Для представления обоих этих типов в классе `DrawingShape` используется поле `Shape`.

Добавьте к классу `DrawingShape` следующий конструктор:

```
abstract class DrawingShape
{
    ...
    public DrawingShape(int size)
    {
        this.size = size;
    }
}
```

Этот код инициализирует в `DrawingShape`-объекте поле `size`.

Добавьте к классу `DrawingShape` методы `SetLocation` и `SetColor`, выделенные в показанном далее коде жирным шрифтом. Эти методы предоставляют реализацию, наследуемую всеми классами, которые являются производными класса `DrawingShape`. Обратите внимание на то, что они не помечены как виртуальные и в производном классе их перегрузка не ожидается. Кроме того, класс `DrawingShape` не объявляется реализацией интерфейсов `IDraw` или `IColor` (реализацией интерфейсов занимаются `Square` и `Circle`, а не этот абстрактный класс), поэтому методы просто объявляются открытыми:

```
abstract class DrawingShape
{
    ...
    public void SetLocation(int xCoord, int yCoord)
    {
        this.locX = xCoord;
        this.locY = yCoord;
    }

    public void SetColor(Color color)
    {
        if (this.shape != null)
        {
            SolidColorBrush brush = new SolidColorBrush(color);
            this.shape.Fill = brush;
        }
    }
}
```

Добавьте к классу `DrawingShape` метод `Draw`. В отличие от предыдущих методов, он объявляется виртуальным, и ожидается, что в любом производном классе для расширения функциональности он будет перегружен. Код метода проверяет поле `shape` на `null`-значение, а затем рисует его на холсте. Классы, наследующие этот метод, должны предоставить для создания экземпляра класса в виде `shape`-объекта свой собственный код. (Не забудьте, что класс `Square` создает объект `Rectangle`, а класс `Circle` — объект `Ellipse`.)

```
abstract class DrawingShape
{
    ...
    public virtual void Draw(Canvas canvas)
    {
        if (this.shape == null)
        {
            throw new InvalidOperationException("Shape is null");
        }

        this.shape.Height = this.size;
        this.shape.Width = this.size;
        Canvas.SetTop(this.shape, this.locY);
    }
}
```

```
    Canvas.SetLeft(this.shape, this.locX);
    canvas.Children.Add(this.shape);
}
}
```

Теперь создание класса `DrawingShape` завершено. Следующий этап будет заключаться в изменении классов `Square` и `Circle`, превращающем их в наследников этого класса, с последующим удалением из них повторяющегося кода.

Изменение классов `Square` и `Circle` с превращением их в наследников класса `DrawingShape`

Выведите в окно редактора класс `Square`. Измените определение класса `Square`, указав его в качестве наследника класса `DrawingShape` в дополнение к указанию того, что в нем реализуются интерфейсы `IDraw` и `IColor`:

```
class Square : DrawingShape, IDraw, IColor
{
    ...
}
```

Учтите, что класс, наследником которого является класс `Square`, нужно указать перед указанием о необходимости реализации интерфейсов.

Удалите из класса `Square` определения полей `sideLength`, `rect`, `locX` и `locY`. Эти поля не нужны, поскольку теперь они предоставляются классом `DrawingShape`.

Замените существующий конструктор следующим кодом, вызывающим конструктор, находящийся в базовом классе:

```
class Square : DrawingShape, IDraw, IColor
{
    public Square(int sideLength)
        : base(sideLength)
    {
    }
    ...
}
```

Обратите внимание на то, что у этого конструктора пустое тело, поскольку все нужные инициализации выполняются конструктором базового класса.

Удалите из класса `Square` методы `IDraw.SetLocation` и `IColor.SetColor`. Реализацию этих методов предоставляет класс `DrawingShape`.

Измените определение метода `Draw`. Объявите его с ключевыми словами `public override`, а также удалите ссылку на интерфейс `IDraw`. Базовые действия для этого метода также уже предоставлены классом `DrawingShape`, но его функциональность будет расширена специальным кодом, требующимся классу `Square`:

```
public override void Draw(Canvas canvas)
{
    ...
}
```

Замените тело метода `Draw` кодом, выделенным жирным шрифтом:

```
public override void Draw(Canvas canvas)
{
    if (this.shape != null)
    {
        canvas.Children.Remove(this.shape);
    }
    else
    {
        this.shape = new Rectangle();
    }

    base.Draw(canvas);
}
```

Эти инструкции создают экземпляр поля `shape`, унаследованного от класса `DrawingShape`, в качестве нового экземпляра класса `Rectangle` — при условии, что такой экземпляр еще не был создан. Затем они вызывают в классе `DrawingShape` метод `Draw`.

Повторите все предыдущие операции в отношении класса `Circle`, применив при этом для конструктора имя `Circle` и параметр `diameter` и создав в методе `Draw` экземпляр поля `shape` в качестве нового `Ellipse`-объекта. Готовый код для класса `Circle` должен приобрести следующий вид:

```
class Circle : DrawingShape, IDraw, IColor
{
    public Circle(int diameter)
        : base(diameter)
    {

    }

    public override void Draw(Canvas canvas)
    {
        if (this.shape != null)
        {
            canvas.Children.Remove(this.shape);
        }
        else
        {
            this.shape = new Ellipse();
        }

        base.Draw(canvas);
    }
}
```

Щелкните в меню Отладка на пункте Начать отладку. Когда откроется окно Drawing Pad, убедитесь в том, что при щелчке левой кнопкой мыши на холсте появляются **Square**-объекты (квадраты), а при щелчке правой кнопкой мыши — **Circle**-объекты (круги). Приложение должно вести себя точно так же, как и прежде.

Вернитесь в среду Visual Studio и остановите отладку.

ЕЩЕ РАЗ О СОВМЕСТИМОСТИ С WINDOWS RUNTIME

В главе 9 «Создание типов значений с использованием перечислений и структур» приводилось описание того, как платформа Windows из Windows 8 далее реализует модель Windows Runtime (WinRT) в качестве надстройки над исходными API-интерфейсами Windows, предоставляя разработчикам упрощенный интерфейс программирования для создания неуправляемых приложений. (Под неуправляемыми понимаются такие приложения, которые не запускаются с использованием .NET Framework; они создаются с использованием языка C++, а не C#.) Управляемые приложения для запуска приложений .NET Framework используют общезыковую среду выполнения (common language runtime (CLR)). Платформа .NET Framework предоставляет обширный набор библиотек и функций. Под управлением Windows 7 и более ранних версий операционной системы CLR реализует эти функции путем использования исходных API-интерфейсов Windows. При создании приложений для настольных систем или корпоративных приложений и служб, запускаемых под Windows 10, этот же набор функций по-прежнему доступен (хотя сама платформа обновлена до версии 4.6), и любые C#-приложения, работающие под Windows 7, должны без всяких изменений запускаться и под Windows 10.

Под управлением Windows 10 UWP-приложения всегда запускаются с использованием WinRT. Это означает, что при создании UWP-приложений, использующих управляемые языки, подобные C#, CLR фактически вызывает WinRT, а не исходные API-интерфейсы Windows. Microsoft предоставила отображаемый уровень между CLR и WinRT, который способен явным образом переводить запросы на создание объектов и вызывать методы, созданные под .NET Framework, в эквивалентные запросы на создание объектов и вызовы методов в WinRT. Например, когда создается значение Int32 для .NET Framework (int в C#), этот код переводится в код создания значения, использующего эквивалентный тип данных WinRT. Но хотя в CLR и в WinRT имеется большой объем перекрывающихся функциональных возможностей, соответствующие функции в WinRT есть не для всех функций .NET Framework 4.6. Поэтому у UWP-приложений есть доступ лишь к сокращенному набору типов и методов, предоставляемому .NET Framework 4.6. (Механизм IntelliSense в Visual Studio 2015 автоматически показывает ограниченное представление доступных функций при использовании C# для создания UWP-приложений, опуская те типы и методы, которые недоступны при использовании WinRT.)

В то же время WinRT предоставляет внушительный набор функций и типов, не имеющих прямого эквивалента в .NET Framework или работающих совершенно иначе, нежели соответствующие им функции в .NET Framework, и не позволя-

ющих легко выполнять их перевод. WinRT открывает доступ к этим функциям для CLR путем использования уровня отображения, делающего их похожими на типы и методы, имеющиеся в .NET Framework, и их можно вызвать из управляемого кода напрямую.

Интеграция, реализованная в CLR и в WinRT, позволяет CLR свободно пользоваться типами, имеющимися в WinRT, но при этом она также поддерживает функциональную взаимозаменяемость в обратном направлении: вы можете определять типы, используя управляемый код, и открывать к ним доступ со стороны неуправляемых приложений при условии, что эти типы соответствуют ожиданиям WinRT. В главе 9 в этом отношении были обозначены требования структур (методы экземпляров и статические методы в структурах через WinRT недоступны, а закрытые поля не поддерживаются). Если классы создаются с принципом на использование через WinRT в неуправляемых приложениях, то нужно придерживаться следующих правил.

- Все открытые поля, параметры и возвращаемые значения всех открытых методов должны относиться к типам WinRT или типам .NET Framework, свободно переводимым средой WinRT в типы WinRT. Примеры поддерживаемых типов .NET Framework включают согласующиеся типы значений (например, структуры и перечисления) и те типы значений, которые соответствуют простым типам C# (int, long, float, double, string и т. д.). Закрытые поля поддерживаются в классах и могут быть любого типа, доступного в .NET Framework, они не должны согласовываться с WinRT.
- Классы не могут перегружать методы System.Object, за исключением метода ToString, и в них не могут объявляться защищенные конструкторы.
- У пространства имен, в котором определяется класс, должно быть такое же имя, как и у сборки, реализующей класс. Кроме того, имя пространства имен (а следовательно, и имя сборки) не должно начинаться с «Windows».
- Задать через WinRT наследование от управляемых типов в неуправляемых приложениях невозможно. Поэтому все открытые классы должны быть запечатанными. Если нужно реализовать полиморфизм, можно создать открытый интерфейс и реализовать этот интерфейс в классах, в которых должен проявляться полиморфизм.
- Можно выдавать любой тип исключений, включенный в поднабор .NET Framework, доступный UWP-приложениям, — создавать собственные классы исключений вы не сможете. Если при вызове из неуправляемого приложения ваш код выдает необрабатываемое исключение, WinRT выдает эквивалентное исключение в неуправляемом коде.

Далее в книге будут рассматриваться и другие требования, налагаемые WinRT на функциональные возможности кода C#. Эти требования будут выделяться при рассмотрении каждой функции.

Выводы

В этой главе вы увидели, как определяются и реализуются интерфейсы и абстрактные классы. Различные допустимые (Да) и недопустимые (Нет) комбинации ключевых слов при определении методов для интерфейсов, классов и структур сведены в табл. 13.1.

Таблица 13.1

Ключевое слово	Интерфейс	Абстрактный класс	Класс	Запечатанный класс	Структура
abstract	Нет	Да	Нет	Нет	Нет
new	Да*	Да	Да	Да	Нет**
override	Нет	Да	Да	Да	Нет***
private	Нет	Да	Да	Да	Да
protected	Нет	Да	Да	Да	Нет****
public	Нет	Да	Да	Да	Да
sealed	Нет	Да	Да	Да	Нет
virtual	Нет	Да	Да	Нет	Нет

* Интерфейс может расширять другой интерфейс и вводить новый метод с точно такой же сигнатурой.

** Структуры не поддерживают наследование, поэтому они не могут скрывать методы.

*** Структуры не поддерживают наследование, поэтому они не могут перегружать методы.

**** Структуры не поддерживают наследование, они подразумеваются запечатанными и не могут быть производными от других структур.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 14.

Если хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Объявить интерфейс	<p>Воспользуйтесь ключевым словом <code>interface</code>, например:</p> <pre>interface IDemo { string GetName(); string GetDescription(); }</pre>
Реализовать интерфейс	<p>Объявите класс, воспользовавшись таким же синтаксисом, как и при объявлении класса-наследника, а затем реализуйте все функции, входящие в интерфейс, например:</p> <pre>class Test : IDemo { public string IDemo.GetName() { ... } public string IDemo.GetDescription() { ... } }</pre>
Создать абстрактный класс, который можно использовать только в качестве базового класса, содержащего абстрактные методы	<p>Объявите класс, воспользовавшись ключевым словом <code>abstract</code>. Укажите при объявлении каждого абстрактного метода ключевое слово <code>abstract</code>, не указывая при этом тело метода, например:</p> <pre>abstract class GrazingMammal { abstract void DigestGrass(); ... }</pre>
Создать запечатанный класс, который не может использоваться в качестве базового класса	<p>Объявите класс, воспользовавшись ключевым словом <code>sealed</code>, например:</p> <pre>sealed class Horse { ... }</pre>

14

Использование сборщика мусора и управление ресурсами

Прочитав эту главу, вы научитесь:

- управлять системными ресурсами, используя сборщик мусора;
- создавать код, запускаемый при уничтожении объекта;
- высвобождать ресурс в известный момент времени независимо от выдачи исключений путем использования пары инструкций `try-finally`;
- высвобождать ресурс в известный момент времени независимо от выдачи исключений путем использования инструкции `using`;
- реализовывать интерфейс `IDisposable` для поддержки в классе возможности высвобождения ресурсов независимо от выдачи исключений.

В предыдущих главах вы научились создавать переменные и объекты и должны понимать, как при их создании выделяется память. (Напомним, что типы значений создаются в стеке, а ссылочным типам выделяется участок динамической памяти, которую еще называют кучей.) Память компьютеров не безгранична, поэтому как только нужда в переменной или объекте отпадет, они должны быть возвращены. Типы значений уничтожаются, и память, которую они занимали, возвращается, как только они выходят из области видимости. С этим все ясно. А как насчет ссылочных типов? Объект создается с помощью ключевого слова `new`, но как и когда он уничтожается? Ответу на этот вопрос и посвящена данная глава.

Срок существования объекта

Сначала давайте разберемся с тем, что происходит при создании объекта.

Объект создается с помощью оператора `new`. В следующем примере создается новый экземпляр класса `Square`, рассмотренного в главе 13 «Создание интерфейсов и определение абстрактных классов».

```
int sizeOfSquare = 99;
Square mySquare = new Square(sizeOfSquare); // Square является ссылочным типом
```

С вашей точки зрения, операция `new` происходит в один этап, но на самом деле объект создается в два этапа.

1. Операция `new` выделяет участок обычной динамической памяти. Этот этап создания объекта вам не подконтролен.
2. Операция `new` превращает участок обычной памяти в объект, который нужно инициализировать. Этот этап можно контролировать с помощью конструктора.



ПРИМЕЧАНИЕ Программисты, работавшие на C++, должны обратить внимание на то, что в C# перегрузить операцию `new`, чтобы контролировать выделение памяти, невозможно.

После создания объекта доступ к его компонентам можно получить с помощью оператора «точка» (`.`). Например, в классе `Square` имеется метод по имени `Draw`, который можно вызвать следующим образом:

```
mySquare.Draw();
```



ПРИМЕЧАНИЕ Этот код основан на версии класса `Square`, являющегося наследником абстрактного класса `DrawingShape` и не имеющего явной реализации интерфейса `IDraw`. Рассказывания можно найти в главе 13.

Когда переменная `mySquare` исчезает из области видимости, активных ссылок на `Square`-объект не остается. Поэтому объект может быть уничтожен, а занимаемая им память возвращена в оборот. (Но как вы увидите чуть позже, это может произойти не сразу.) Уничтожение объекта происходит в два этапа, являющихся зеркальным отражением двух этапов его создания.

1. Общеязыковой среде выполнения (common language runtime (CLR)) необходимо навести порядок. Контролировать этот этап можно созданием деструктора.
2. CLR должна вернуть память, ранее принадлежавшую объекту, в кучу; память, в которой находился объект, должна быть высвобождена. Контролировать этот этап невозможно.

Процесс уничтожения объекта и возвращения памяти в кучу известен как сборка мусора.



ПРИМЕЧАНИЕ Программисты, работавшие на C++, должны обратить внимание на то, что в C# нет оператора delete. Уничтожением объекта управляет CLR.

Создание деструкторов

Деструктор может использоваться для любой уборки, требующейся, когда объект попадает под сборку мусора. Среда CLR автоматически высвободит любые управляемые ресурсы, используемые объектом, поэтому во многих подобных случаях в написании деструктора нет необходимости. Но если управляемый ресурс велик по объему (например, занят многомерным массивом), возможно, имеет смысл сделать его доступным для немедленного высвобождения путем установки для всех ссылок на этот ресурс, имеющихся в объекте, null-значений. Кроме того, применение деструктора может стать вполне оправданным, если объект непосредственно или опосредованно ссылается на неуправляемый ресурс.



ПРИМЕЧАНИЕ Опосредованные неуправляемые ресурсы встречаются довольно часто. К ним можно отнести файловые потоки, сетевые подключения, подключения к базам данных и другие ресурсы, управляемые Windows. Если, к примеру, в методе открывается файл, может понадобиться добавить деструктор, закрывающий файл при уничтожении объекта. Но, возможно, в зависимости от структуры кода в вашем классе лучше будет закрыть файл в какой-нибудь другой момент времени. (Обратите внимание на инструкцию using, рассматриваемую в этой главе чуть позже.)

Деструктор — это специальный метод, немного похожий на конструктор, но отличающийся от него тем, что он вызывается средой CLR при исчезновении объекта. В синтаксис для написания деструктора входит знак тильды (~), за которым следует имя класса. Например, далее показан простой класс, открывающий в своем конструкторе файл для чтения и закрывающий этот файл в своем деструкторе. (Учитите, что это просто пример, и я не рекомендую применять эту схему для открытия и закрытия файлов.)

```
class FileProcessor
{
    FileStream file = null;

    public FileProcessor(string fileName)
    {
        this.file = File.OpenRead(fileName); // open file for reading
    }

    ~FileProcessor()
    {
        this.file.Close(); // close file
    }
}
```

На деструкторы накладывается ряд весьма важных ограничений.

- ❑ Деструкторы применяются только к ссылочным типам — объявлять деструкторы применительно к типам значений, таким как структура, нельзя:

```
struct MyStruct
{
    ~MyStruct() { ... } // ошибка в ходе компиляции
}
```

- ❑ Для деструкторов нельзя указывать модификатор доступа, например `public`. Деструктор никогда не вызывается из вашего собственного кода, за вас это делает та часть среды CLR, которая называется сборщиком мусора:

```
public ~FileProcessor() { ... } // ошибка в ходе компиляции
```

- ❑ Деструктор не может принимать какие-либо параметры. Причина опять-таки в том, что вы никогда не вызываете деструктор самостоятельно:

```
~FileProcessor(int parameter) { ... } // ошибка в ходе компиляции
```

Компилятор C#, выполняя свою внутреннюю работу, автоматически транслирует деструктор в переопределение метода `Object.Finalize`. Компилятор преобразует этот деструктор

```
class FileProcessor
{
    ~FileProcessor() { // сюда помещается ваш код }
}
```

в следующий код:

```
class FileProcessor
{
    protected override void Finalize()
    {
        try { // сюда помещается ваш код }
        finally { base.Finalize(); }
    }
}
```

Создаваемый компилятором метод `Finalize` содержит внутри `try`-блока тело деструктора, а после этого блока следует блок `finally`, который вызывает в базовом классе метод `Finalize`. (Ключевые слова `try` и `finally` рассматриваются в главе 6 «Обработка ошибок и исключений».) Тем самым гарантируется, что деструктор всегда вызывает деструктор своего базового класса, даже если исключение было выдано при выполнении вашего кода деструктора.

Важно усвоить, что это преобразование делает только компилятор. Вы не сможете написать собственный метод, переопределяющий метод `Finalize`, и не сможете сами вызвать метод `Finalize`.

Зачем используется сборщик мусора?

Самостоятельно уничтожить объект, используя код C#, невозможно. Для этого просто не существует синтаксиса. Среда CLR делает это за вас по своему усмотрению. Кроме того, следует учесть, что для ссылки на один и тот же объект может быть создано более одной ссылочной переменной. В следующем примере кода на один и тот же объект `FileProcessor` указывают переменные `myFp` и `referenceToMyFp`:

```
FileProcessor myFp = new FileProcessor();
FileProcessor referenceToMyFp = myFp;
```

Сколько ссылок на объект можно создать? Да сколько угодно! Но такая неограниченность влияет на срок существования объекта. Среда CLR должна отслеживать все эти ссылки. Если переменная `myFp` исчезнет, выйдя из области видимости, по-прежнему могут существовать другие переменные (вроде `referenceToMyFp`) и ресурсы, используемые объектом `FileProcessor`, не могут быть высвобождены (файл не должен закрываться). Следовательно, срок существования объекта не может быть привязан к конкретной ссылочной переменной. Объект может быть уничтожен, но его память станет доступна для повторного использования, только когда на него исчезнут все ссылки.

Как видите, управлять сроком существования объекта нелегко, поэтому создатели C# решили снять с вашего кода эту ответственность. Если бы обязанность уничтожать объекты возлагалась на вас, то рано или поздно сложилась бы одна из следующих ситуаций.

- ❑ Вы бы забыли уничтожить объект. Следовательно, деструктор объекта (если таковой имелся) запущен не будет, уборка не состоится и память не будет возвращена в кучу. При этом память может быстро исчерпаться.
- ❑ Вы бы попытались уничтожить активный объект, что создало бы вероятность сохранения в одной или нескольких переменных ссылки на уничтоженный объект — так называемой висячей ссылки. Такая ссылка указывает либо на неиспользуемую область памяти, либо, возможно, на совершенно другой объект, который к этому моменту занял тот же участок памяти. Так или иначе, последствия использования висячей ссылки в лучшем случае приведут к неопределенности, а в худшем — к угрозам безопасности. Любой исход будет неприемлем.
- ❑ Вы бы попытались уничтожить один и тот же объект более одного раза. Это могло бы привести — а могло и не привести — к губительным последствиям в зависимости от кода, находящегося в деструкторе.

В таких языках, как C#, где надежность и безопасность ставятся на первые места в списке приоритетных проектных задач, подобные проблемы неприемлемы.

Поэтому уничтожением объектов за вас занимается сборщик мусора. Этот сборщик гарантирует следующее.

- ❑ Каждый объект будет уничтожен, и его деструктор будет запущен. Когда программа завершит свою работу, все использовавшиеся в ней объекты будут уничтожены.
- ❑ Каждый объект будет уничтожен только единожды.
- ❑ Каждый объект будет уничтожен только тогда, когда станет недоступен, то есть когда в процессе выполнения вашего приложения на него не будет ссылок.

Пользу этих гарантий трудно переоценить, поскольку они освобождают программиста от утомительных забот, из-за которых легко ошибиться. Они позволяют вам сосредоточиться на самой логике программы и повысить производительность работы.

Когда же происходит сборка мусора? Этот вопрос может показаться странным. Ведь ясно, что это делается, как только отпадет надобность в объекте. Да, так и есть, но этот процесс не носит обязательный немедленный характер. Сборка мусора может быть весьма затратным делом, поэтому среда CLR занимается этим по мере надобности (когда объем доступной памяти снижается или, к примеру, размер кучи достигает определяемого системой порога), и тогда происходит максимально возможная сборка мусора. Эффективнее проводить несколько крупных операций по очистке памяти, чем множество мелких.



ПРИМЕЧАНИЕ Работу сборщика мусора можно инициировать в программе, вызвав статический метод Collect класса GC, который находится в пространстве имен System. Но за исключением редких случаев делать это не рекомендуется. Метод GC.Collect запускает сборщик мусора, но процесс выполняется в асинхронном режиме — метод GC.Collect перед возвращением управления не дожидается полного завершения работы сборщика, поэтому сведений о том, что ваши объекты уничтожены, вы не получаете. Позвольте выбрать наилучший момент для сборки мусора самой среде CLR.

Одна из особенностей сборщика мусора состоит в том, что вы не знаете порядка уничтожения объектов и не должны от него зависеть. Последняя особенность, которую, наверное, наиболее важно усвоить, заключается в следующем: деструкторы не запускаются до тех пор, пока объекты не попадут под сборку мусора. Если вами создается деструктор, следует понимать, что он будет выполнен, но вы не будете знать, когда именно это произойдет. Следовательно, не нужно создавать код, зависящий от запуска деструкторов в конкретной последовательности или к определенному времени выполнения вашего приложения.

Как работает сборщик мусора?

Сборщик мусора запускается в своем собственном потоке и может выполнятьсь только в определенные моменты времени, обычно когда ваше приложение

завершит выполнение метода. В ходе его работы другие потоки, выполняющие ваше приложение, временно приостановятся, поскольку сборщику мусора может понадобиться переместить объекты и обновить на них ссылки, а он не сможет сделать это, пока объекты используются.



ПРИМЕЧАНИЕ Поток является отдельным путем выполнения приложения. Windows использует потоки, чтобы позволить приложению одновременно выполнять сразу несколько операций.

Сборщик мусора относится к сложным программным средствам, обладающим собственными настроочными возможностями и выполняющим ряд оптимизаций. Они должны сбалансировать потребности в сохранении доступной памяти с требованиями обеспечения высокой производительности приложения. Подробности внутренних алгоритмов и структур, используемых сборщиком мусора, выходят за рамки вопросов, рассматриваемых в данной книге (и компания Microsoft непрерывно совершенствует способы выполнения сборщиком мусора его работы), но если рассматривать вопрос на высоком уровне, шаги, предпринимаемые этим сборщиком, выглядят следующим образом.

1. Он составляет отображение всех доступных объектов, многократно следя по полям ссылок внутри объектов. Это отображение создается сборщиком мусора весьма тщательно и гарантирует, что циклические ссылки не приведут к бесконечной рекурсии. Любой объект, отсутствующий в этом отображении, считается недоступным.
2. Он проверяет, имеется ли в каждом недоступном объекте деструктор, который необходимо выполнить (этот процесс называется финализацией (finalizable)). Любой недоступный объект, требующий финализации, ставится в особую очередь, которая называется очередью объектов, готовых к финализации.
3. Он высвобождает память, ранее выделенную всем остальным недоступным объектам (не требующим финализации), путем перемещения доступных объектов в нижнюю часть кучи, дефрагментируя тем самым кучу и высвобождая память на ее вершине. Когда сборщик мусора перемещает доступный объект, он также обновляет все ссылки на него.
4. В этот момент он позволяет возобновить работу всем остальным потокам.
5. Он проводит финализацию недоступных объектов, требующих финализации (тех, что находятся в finalizable-очереди), запуская в своем собственном потоке методы `Finalize`.

Рекомендации

Создание классов, содержащих деструкторы, усложняет код и сборку мусора и вынуждает программу работать медленнее. Если в программе нет деструкторов,

сборщику мусора не требуется помещать недоступные объекты в `foreachable`-очередь и выполнять их финализацию. Очевидно, что всякая работа требует времени. Поэтому постарайтесь избегать использования деструкторов, за исключением тех случаев, когда в них есть реальная необходимость, используйте их только для возвращения неуправляемых ресурсов. (Чуть позже в этой главе будет рассмотрена альтернатива их использованию в виде инструкции `using`.)

При создании деструкторов нужно проявлять особую осмотрительность. В частности, имейте в виду, что при вызове деструктором других объектов эти другие объекты уже могут иметь собственные деструкторы, вызванные сборщиком мусора. Следует помнить, что при финализации какой-либо порядок выполнения деструкторов не гарантирован. Поэтому следует исключить взаимозависимость деструкторов или их наложение друг на друга. Не следует, к примеру, иметь два деструктора, пытающихся высвобождать одни и те же ресурсы.

Управление ресурсами

Иногда высвобождать ресурс в деструкторе нецелесообразно, поскольку некоторые ресурсы слишком ценные, чтобы выжидать некий произвольный срок, пока сборщик мусора не займется их высвобождением. В высвобождении нуждаются такие дефицитные ресурсы, как память, подключения к базам данных или дескрипторы файлов, и делать это должно как можно скорее. В подобных ситуациях единственным вариантом является самостоятельное высвобождение ресурса. Эту задачу можно выполнить, создав *метод высвобождения ресурсов*, который непосредственно этим и займется. Если в классе имеется метод высвобождения ресурсов, можно вызвать его и управлять моментом высвобождения.



ПРИМЕЧАНИЕ Понятие метода высвобождения ресурсов (*disposal method*) относится к назначению метода, а не к его имени. Этому методу можно дать имя, использующее любой допустимый в C# идентификатор.

Методы высвобождения ресурсов

Примером класса, реализующего метод высвобождения ресурсов, может послужить класс `TextReader` из пространства имен `System.IO`. Он предоставляет механизм для считывания символов из последовательного потока ввода. Класс `TextReader` содержит виртуальный метод по имени `Close`, который закрывает поток. Класс `StreamReader`, занимающийся считыванием символов из потока, например из открытого файла, и класс `StringReader`, считающий символы из строки, являются производными класса `TextReader`, и в обоих этих классах происходит перегрузка метода `Close`. Рассмотрим пример,читывающий строки текста из файла с использованием класса `StreamReader` и выводящий их после этого на экран:

```
TextReader reader = new StreamReader(filename);
string line;
while ((line = reader.ReadLine()) != null)
{
    Console.WriteLine(line);
}
reader.Close();
```

Метод `ReadLine` считывает очередную строку текста из потока в строковую переменную. Если в потоке уже ничего не остается, метод `ReadLine` возвращает значение `null`. Когда считывание завершается, вызов `Close` имеет весьма большое значение для высвобождения дескриптора файла и ресурсов, связанных со считыванием строк. Но в примере есть одна проблема: он не может гарантированно работать при выдаче исключений. Если при вызове `ReadLine` или `WriteLine` выдается исключение, вызова `Close` не происходит — ему просто не будет передано управление. Если это случается довольно часто, ресурс дескрипторов файлов будет исчерпан и программа не сможет открывать дополнительные файлы.

Высвобождение ресурсов независимо от выдачи исключений

Один из способов безусловного вызова метода высвобождения ресурсов, такого как `Close`, независимо от того, произошла или нет выдача исключения, заключается в вызове этого метода в блоке `finally`. Предыдущий пример, за-программированный с применением этой технологии, имеет следующий вид:

```
TextReader reader = new StreamReader(filename);
try
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
finally
{
    reader.Close();
}
```

Такое использование блока `finally` вполне работоспособно, но у него есть ряд недостатков, которые не позволяют признать это решение близким к идеалу.

- ❑ Если приходится высвобождать более одного ресурса, такое решение очень быстро становится громоздким. (В конечном итоге появляется большое количество вложенных блоков `try` и `finally`.)
- ❑ В некоторых случаях, чтобы вписаться в эту идиому, приходится изменять код. (Например, может понадобиться изменить порядок объявления ссылок

на ресурсы, не забыть инициализировать ссылку `null`-значением и не забыть проверить в блоке `finally`, что ссылка не имеет `null`-значения.)

- ❑ Это решение не позволяет создать свою абстракцию. Это означает, что в решении трудно разобраться и приходится повторять код во всех местах, где требуется получение такой функциональной возможности.
- ❑ Ссылки на ресурс после блока `finally` остаются в области видимости. Это означает, что вы можете случайно предпринять попытку воспользоваться ресурсом уже после его высвобождения.

Для решения всех этих проблем и была разработана инструкция `using`.

Инструкция `using` и интерфейс `IDisposable`

Инструкция `using` предоставляет точный механизм управления продолжительностью использования ресурсов. Можно создать объект, который будет уничтожен сразу же, как только будет выполнен код блока `using`.



ВНИМАНИЕ Не нужно путать инструкцию `using`, показанную в этом разделе, с директивой `using`, которая вводит в область видимости пространство имен. Так уж, к сожалению, сложилось, что у одного и того же ключевого слова имеются два совершенно разных назначения.

Для инструкции `using` используется следующий синтаксис:

```
using ( type variable = initialization )
{
    StatementBlock
}
```

А вот наилучший способ гарантировать безусловный вызов `Close` в `TextReader`:

```
using (TextReader reader = new StreamReader(filename))
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
```

Инструкция `using` является точным эквивалентом следующего преобразования:

```
{
    TextReader reader = new StreamReader(filename);
    try
    {
```

```
        string line;
        while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
finally
{
    if (reader != null)
    {
        ((IDisposable)reader).Dispose();
    }
}
}
```



ПРИМЕЧАНИЕ Собственный блок введен в инструкцию `using` с целью обозначения области видимости. Такое решение о структуре кода означает, что переменные, объявленные в инструкции `using`, автоматически выходят из области видимости в конце вложенной в блок инструкции и случайно получить доступ к освобожденному ресурсу будет уже невозможно.

Переменная, объявляемая в инструкции `using`, должна быть того типа, который реализует интерфейс `IDisposable`. Этот интерфейс находится в пространстве имен `System` и содержит только один метод по имени `Dispose`:

```
namespace System
{
    interface IDisposable
    {
        void Dispose();
    }
}
```

Метод `Dispose` предназначен для высвобождения всех используемых объектом ресурсов. Так уж вышло, что класс `StreamReader` реализует интерфейс `IDisposable`, а его метод `Dispose` вызывает метод `Close`, чтобы закрыть поток. Инструкция `using` может использоваться в качестве вполне понятного, не зависящего от выдачи исключений и надежного способа, гарантирующего безусловное высвобождение ресурса. Этот подход решает все проблемы, имеющиеся в создаваемом вручную решении, использующем пару инструкций `try-finally`. У вас появляется решение, способное:

- ❑ отлично масштабироваться, если требуется высвободить сразу несколько ресурсов;
- ❑ не искажать логику программного кода;
- ❑ абстрагироваться от проблем и избегать повторений;
- ❑ работать надежно. Оно исключает возможность случайной ссылки на переменную, объявленную внутри инструкции `using` (в данном случае имеется в виду переменная `reader`), после завершения выполнения инструкции `using`,

поскольку она больше не находится в области видимости, и вы получите ошибку в ходе компиляции приложения.

Вызов метода `Dispose` из деструктора

Нужно ли при написании собственных классов включать в них деструктор или реализацию интерфейса `IDisposable`, чтобы инструкция `using` могла управлять экземплярами класса? Вызов деструктора состоится, но когда именно это произойдет, вам неизвестно. В то же время вы точно знаете, когда состоится вызов метода `Dispose`, но не можете быть уверены, случится ли это, поскольку все зависит от того, не забудет ли программист, использующий ваши классы, написать инструкцию `using`. Но есть возможность гарантировать безусловный запуск метода `Dispose` путем его запуска из деструктора. Получится полезный запасной вариант. Вы можете забыть вызывать метод `Dispose`, но по крайней мере будете уверены в том, что он будет вызван, даже если это произойдет в самом конце работы программы. Подробное изучение этой особенности будет предпринято в конце главы, а пока посмотрим на пример возможной реализации интерфейса `IDisposable`:

```
class Example : IDisposable
{
    private Resource scarce;      // дефицитный ресурс для управления
    и высвобождения
    private bool disposed = false; // флагок, показывающий, был ли ресурс
                                   // уже освобожден
    ...
    ~Example()
    {
        this.Dispose(false);
    }

    public virtual void Dispose()
    {
        this.Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                // здесь происходит высвобождение объемного управляемого ресурса
                ...
            }
            // здесь происходит высвобождение неуправляемых ресурсов
            ...
            this.disposed = true;
        }
    }
}
```

```
public void SomeBehavior() // метод класса Example
{
    checkIfDisposed();
    ...
}

private void checkIfDisposed()
{
    if (this.disposed)
    {
        throw new ObjectDisposedException("Example: object has been
                                         disposed of");
    }
}
```

Обратите внимание на следующие особенности класса `Example`.

- ❑ В классе реализуется интерфейс `IDisposable`.
- ❑ Открытый метод `Dispose` может быть вызван кодом вашего приложения в любое время.
- ❑ Открытый метод `Dispose` вызывает закрытую и перегружаемую версию метода `Dispose`, принимающую параметр в виде булева значения, передавая в качестве аргумента значение `true`. Этот метод фактически выполняет вы-
свобождение ресурса.
- ❑ Деструктор вызывает закрытую и перегружаемую версию метода `Dispose`, принимающую параметр в виде булева значения, передавая в качестве аргу-
ментом значение `false`. Деструктор вызывается только сборщиком мусора
при финализации объекта.
- ❑ Защищенный метод `Dispose` можно безопасно вызывать несколько раз. Пере-
менная `disposed` показывает, запускался ли уже этот метод, и является надеж-
ным средством предотвращения попыток метода многократно высвобождать
ресурсы при его одновременном вызове. (Ваше приложение может вызвать
`Dispose`, но еще до того, как метод завершит работу, ваш объект может попасть
под сборку мусора и метод `Dispose` будет запущен из деструктора средой CLR
еще раз.) Ресурсы высвобождаются только при первом запуске метода.
- ❑ Защищенный метод `Dispose` поддерживает высвобождение управляемых
ресурсов (например, большого массива) и неуправляемых ресурсов (на-
пример, дескриптора файла). Если параметр `disposing` имеет значение
`true`, этот метод должен быть вызван из открытого метода `Dispose`. В этом
случае высвобождаются все управляемые и неуправляемые ресурсы. Если
параметр `disposing` имеет значение `false`, этот метод должен быть вызван
из деструктора и финализацией объекта займется сборщик мусора. В этом
случае нет необходимости в высвобождении управляемых ресурсов (или

в независимости от выдачи исключений), поскольку они будут или уже были обработаны сборщиком мусора и высвобождаются только неуправляемые ресурсы.

- ❑ Открытый метод `Dispose` вызывает статический метод `GC.SuppressFinalize`. Этот метод не позволяет сборщику мусора вызывать в отношении данного объекта деструктор, поскольку объект уже был финализирован.
- ❑ Все обычные методы класса, например `SomeBehavior`, проверяют, не был ли объект уже освобожден, в случае чего выдают исключение.

Реализация высвобождения ресурсов независимо от выдачи исключений

В следующих упражнениях будет изучена возможность применения инструкции `using` для своевременного высвобождения ресурсов даже при выдаче исключений в коде вашего приложения. Сначала будет создан простой класс, реализующий деструктор и проверяющий, когда этот деструктор будет вызван сборщиком мусора.



ПРИМЕЧАНИЕ Класс `Calculator`, создаваемый в этих упражнениях, предназначен исключительно для иллюстрации основных принципов сборки мусора. Класс фактически не потребляет какие-либо существенные объемы управляемых или неуправляемых ресурсов. Для такого простого класса, как этот, вы вряд ли станете создавать деструктор или реализовывать интерфейс `IDisposable`.

Создание простого класса, использующего деструктор

Зайдите в среде Microsoft Visual Studio 2015 в меню Файл, выберите пункт Создать и щелкните на пункте Проект. Откроется диалоговое окно Создание проекта. Найдите в левой панели этого окна под пунктом Шаблоны пункт Visual C# и щелкните на нем. Выберите в средней панели шаблон Консольное приложение. Наберите в поле Имя, находящемся в нижней части окна, строку `GarbageCollectionDemo`. Укажите в поле Расположение папку `Microsoft Press\VCSBS\Chapter 14` вашей папки документов и щелкните на кнопке OK.



СОВЕТ Вместо набора пути к папке вручную можно воспользоваться кнопкой Обзор, примыкающей к полю Расположение, и перейти к папке `Microsoft Press\VCSBS\Chapter 14`.

Visual Studio создаст новое консольное приложение и выведет файл `Program.cs` в окно редактора. Щелкните в меню Проект на пункте Добавить класс. Откроется

диалоговое окно Добавить новый элемент — `GarbageCollectionDemo`. Убедитесь, что выбран шаблон Класс. Наберите в поле Имя строку `Calculator.cs` и щелкните на кнопке Добавить. Будет создан класс `Calculator`, и его код будет выведен в окно редактора.

Добавьте к классу `Calculator` следующий открытый метод `Divide` (выделенный жирным шрифтом):

```
class Calculator
{
    public int Divide(int first, int second)
    {
        return first / second;
    }
}
```

Это очень простой метод, который делит первый параметр на второй и возвращает результат. Он предоставляется исключительно для добавления функциональности, доступной для вызова приложением.

Добавьте выше метода `Divide` в самом начале класса `Calculator` открытый конструктор, выделенный в следующем примере кода жирным шрифтом:

```
class Calculator
{
    public Calculator()
    {
        Console.WriteLine("Calculator being created");
    }
    ...
}
```

Этот конструктор нужен для того, чтобы у вас была возможность проверить факт успешного создания `Calculator`-объекта.

После конструктора добавьте к классу `Calculator` деструктор, выделенный в следующем примере кода жирным шрифтом:

```
class Calculator
{
    ...
    ~Calculator()
    {
        Console.WriteLine("Calculator being finalized");
    }
    ...
}
```

Этот деструктор всего лишь показывает сообщение, чтобы дать понять, что сборщик мусора запущен, и занимается финализацией экземпляра этого класса. При создании классов для настоящих приложений вы вряд ли станете заставлять деструктор выводить какой-либо текст.

Выполните файл Program.cs в окно редактора. Добавьте к методу Main класса Program инструкции, выделенные жирным шрифтом:

```
static void Main(string[] args)
{
    Calculator calculator = new Calculator();
    Console.WriteLine($"120 / 15 = {calculator.Divide(120, 15)}");
    Console.WriteLine("Program finishing");
}
```

Этот код создает Calculator-объект, вызывает метод Divide этого объекта (и выводит результат), а затем выводит сообщение о том, что программа завершила работу.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что программа вывела на экран следующую серию сообщений:

```
Calculator being created
120 / 15 = 8
Program finishing
Calculator being finalized
```

Обратите внимание на то, что финализатор для Calculator-объекта запускается только тогда, когда приложение приближается к завершению своей работы, после того как будет выполнен метод Main.

Нажмите в окне консоли клавишу Ввод и вернитесь в среду Visual Studio 2015.

Среда CLR гарантирует, что под сборку мусора попадут все объекты, созданные вашими приложениями, но узнать, когда именно это произойдет, невозможно. В упражнении программа имела слишком короткий срок выполнения, и Calculator-объект был финализирован сразу же, как только по окончании выполнения программы среда CLR провела уборку. Но вы можете столкнуться и с тем, что такая же ситуация складывается и вокруг более объемных приложений, классы которых потребляют дефицитные ресурсы, и пока вы не предпримете необходимые меры по предоставлению средств высвобождения ресурсов, объекты, созданные вашим приложением, могут удерживать свои ресурсы вплоть до завершения его выполнения. Если ресурсом является файл, это может мешать другим пользователям получать доступ к этому файлу; если ресурсом является подключение к базе данных, то ваше приложение может не дать другим пользователям возможности подключиться к ней. В идеале хотелось бы высвободить ресурсы как можно скорее, то есть сразу же после того, как завершится их использование, не дожидаясь прекращения работы приложения.

В следующем упражнении в классе Calculator будет реализован интерфейс IDisposable, и программа сможет финализировать Calculator-объекты в выбранный ею момент времени.

Реализация интерфейса IDisposable

Выполните в окно редактора файл Calculator.cs. Измените определение класса `Calculator`, чтобы в нем происходила реализация интерфейса `IDisposable` (изменение выделено жирным шрифтом):

```
class Calculator : IDisposable
{
    ...
}
```

Добавьте в конец класса `Calculator` метод по имени `Dispose`. Этот метод определяется интерфейсом `IDisposable`:

```
class Calculator : IDisposable
{
    ...
    public void Dispose()
    {
        Console.WriteLine("Calculator being disposed");
    }
}
```

Обычно к методу `Dispose` добавляется код, высвобождающий ресурсы, удерживаемые объектом. В данном случае такой код отсутствует, и назначение инструкции `Console.WriteLine`, имеющейся в этом методе, состоит в простом оповещении о том, что метод `Dispose` был запущен. В настоящих приложениях можно заметить некоторую повторяемость кода деструктора и метода `Dispose`. Чтобы избавиться от повторения, этот код обычно помещают в одно место, а вызывают из другого. Но поскольку вызвать явным образом деструктор из метода `Dispose` невозможно, вместо этого есть смысл вызвать метод `Dispose` из деструктора и поместить логику, высвобождающую ресурсы, в метод `Dispose`.

Внесите в деструктор следующее изменение, выделенное жирным шрифтом, чтобы он вызывал метод `Dispose`. (Оставьте инструкцию, выводящую на экран сообщение в финализаторе, на месте, чтобы можно было увидеть, когда он будет вызван сборщиком мусора.)

```
~Calculator()
{
    Console.WriteLine("Calculator being finalized");
    this.Dispose();
}
```

Когда в приложении потребуется уничтожить `Calculator`-объект, метод `Dispose` не будет запускаться автоматически — ваш код должен либо вызывать его явным образом (к примеру, с помощью инструкции `calculator.Dispose()`), либо создать `Calculator`-объект внутри инструкции `using`. В вашей программе будет использован последний подход.

Выполните в окне редактора файл Program.cs. Внесите в инструкции метода Main, создающие Calculator-объект и вызывающие метод Divide, изменения, выделенные жирным шрифтом:

```
static void Main(string[] args)
{
    using (Calculator calculator = new Calculator())
    {
        Console.WriteLine($"120 / 15 = {calculator.Divide(120, 15)}");
    }

    Console.WriteLine("Program finishing");
}
```

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что теперь программа выводит на экран следующую серию сообщений:

```
Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
Calculator being finalized
Calculator being disposed
```

Инструкция using заставляет метод Dispose запускаться до инструкции, выдающей сообщение о завершении работы программы «Program finishing». Но можно заметить, что деструктор для Calculator-объекта по-прежнему запускается после завершения работы приложения и еще раз вызывает метод Dispose. Вполне очевидно, что он сработает впустую.

Нажмите в окне консоли клавишу Ввод и вернитесь в среду Visual Studio 2015.

Высвобождение удерживаемых объектом ресурсов более одного раза может пройти без негативных последствий, а может и обернуться крупными неприятностями, но вполне определенно такое развитие событий нельзя признать благоприятным. Рекомендуемый подход к решению данной проблемы заключается в добавлении к классу закрытого поля с булевым значением, показывающего, вызывался или нет метод Dispose, с последующей проверкой значения данного поля в этом методе.

Предотвращение повторных высвобождений ресурсов объекта

Выполните в окне редактора файл Calculator.cs. Добавьте к классу Calculator закрытое булево поле по имени disposed и, как выделено жирным шрифтом, инициализируйте это поле значением false:

```
class Calculator : IDisposable
{
    private bool disposed = false;
    ...
}
```

Это поле предназначено для отслеживания состояния данного объекта и хранения признака вызова метода `Dispose`.

Внесите в код метода `Dispose` изменение, при котором сообщение станет выводиться, только если значением поля `disposed` является `false`. После вывода сообщения установите для поля `disposed` значение `true` (все необходимые изменения выделены жирным шрифтом):

```
public void Dispose()
{
    if (!this.disposed)
    {
        Console.WriteLine("Calculator being disposed");
    }

    this.disposed = true;
}
```

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что программа выводит следующую серию сообщений:

```
Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
Calculator being finalized
```

Теперь `Calculator`-объект уничтожается только один раз, но деструктор все равно работает. Опять же это делается впустую, поскольку нет никакого смысла запускать деструктор для объекта, который уже высвободил свои ресурсы.

Нажмите в окне консоли клавишу Ввод и вернитесь в среду Visual Studio 2015.

Добавьте к концу метода `Dispose` класса `Calculator` инструкцию, выделенную жирным шрифтом:

```
public void Dispose()
{
    if (!this.disposed)
    {
        Console.WriteLine("Calculator being disposed");
    }
    this.disposed = true;
GC.SuppressFinalize(this);
}
```

Класс `GC` предоставляет доступ к сборщику мусора и реализует несколько статических методов, с помощью которых можно контролировать ряд выполняемых им действий. Используя метод `SuppressFinalize`, вы можете показать, что сборщику мусора не нужно выполнять финализацию указанного объекта, предотвратив тем самым запуск деструктора.



ВНИМАНИЕ Класс GC предоставляет целый ряд методов, с помощью которых можно настроить работу сборщика мусора. Но, как правило, лучше позволить заниматься сборкой мусора самой среде CLR, поскольку при неразумном самостоятельном вызове этих методов можно нанести серьезный ущерб производительности своего приложения. К методу SuppressFinalize следует относиться с особой осторожностью, поскольку в случае неудачного высвобождения ресурсов объекта вы рискуете потерять данные (к примеру, если корректно закрыть файл не удастся, то любые данные, находящиеся в буфере памяти, но еще не записанные на диск, могут быть утрачены). Вызывать этот метод следует только в ситуациях, подобных той, что сложилась в этом упражнении, когда вы знаете, что объект уже был уничтожен.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что программа выводит на экран следующую серию сообщений:

```
Calculator being created  
120 / 15 = 8  
Calculator being disposed  
Program finishing
```

Как видите, деструктор больше не запускается, потому что Calculator-объект высвободил ресурсы еще до завершения работы программы.

Нажмите в окне консоли клавишу Ввод и вернитесь в среду Visual Studio 2015.

БЕЗОПАСНОСТЬ ПОТОКА И МЕТОД ВЫСВОБОЖДЕНИЯ РЕСУРСОВ

Пример использования поля disposed для предотвращения многократного уничтожения объекта в большинстве случаев работает неплохо, но следует учесть, что вы не контролируете момент запуска финализатора. В упражнениях, представленных в данной главе, он всегда выполняется сразу же по окончании работы программы, но такое случается не всегда — он может быть запущен в любое время после исчезновения последней ссылки на объект. Поэтому вполне вероятно, что финализатор может быть вызван сборщиком мусора в его собственном потоке в то самое время, когда будет работать метод Dispose, особенно если этому методу приходится выполнять большой объем работы. Вероятность многократных попыток высвобождения ресурсов можно снизить перемещением поля disposed поближе к началу метода Dispose, но в таком случае вы рискуете вообще не высвободить ресурсы, если исключение будет выдано после установки значения для этой переменной, но до высвобождения ресурсов.

Чтобы полностью исключить возможность одновременного высвобождения одних и тех же ресурсов одного и того же объекта из двух параллельно выполняемых потоков, можно путем вставки в код C# инструкции lock написать код в таком стиле, который не зависит от потоковой организации выполнения кода:

```
public void Dispose()  
{  
    lock(this)  
    {  
        if (!disposed)
```

```
{  
    Console.WriteLine("Calculator being disposed");  
}  
this.disposed = true;  
GC.SuppressFinalize(this);  
}  
}
```

Инструкция lock предназначена для предотвращения запуска одного и того же блока кода одновременно в разных потоках. Аргументом инструкции lock (в данном примере это this) должна быть ссылка на объект. Код, заключенный в фигурные скобки, определяет область действия инструкции lock. Если выполнение программы доходит до инструкции lock, а указанный объект уже заблокирован, поток, требующий блокировки, блокируется и выполнение в этом месте программы приостанавливается. Когда поток, удерживающий в данный момент блокировку, дойдет в выполнении кода до закрывающей фигурной скобки инструкции lock, блокировка снимается, позволяя установить ее ранее заблокированному потоку и продолжить выполнение программы. Но к тому моменту, когда это случится, для поля disposed уже будет установлено значение true, поэтому второй поток не станет предпринимать попытку выполнения кода в блоке if (!disposed).

Такое использование блокировки безопасно, но может снизить производительность. Альтернативный подход заключается в использовании стратегии, описание которой давалось ранее, при этом предотвращается только повторное высвобождение управляемых ресурсов. (Высвобождение ресурсов, выполняемое более одного раза, не защищено от выдачи исключений: вы не поставите под угрозу безопасность вашего компьютера, но при попытке высвобождения ресурсов управляемого объекта, который больше не существует, может нарушиться логическая целостность вашего приложения.) В данной стратегии реализуются перегружаемые версии метода Dispose: инструкция using вызывает Dispose(), который в свою очередь запускает инструкцию Dispose(true), в то время как деструктор запускает инструкцию Dispose(false). Управляемые ресурсы высвобождаются, только если параметр перегруженной версии метода Dispose имеет значение true. Дополнительные сведения можно найти в разделе «Вызов метода Dispose из деструктора».

Инструкция using предназначена для выдачи гарантии о безусловном высвобождении ресурсов объекта, даже если в ходе его использования будет выдано исключение. В заключительном упражнении данной главы вам нужно будет убедиться, что именно так и происходит при выдаче исключения в ходе выполнения кода, находящегося в блоке инструкции using.

Проверка высвобождения ресурсов объекта после выдачи исключения

Выполните в окне редактора файл Program.cs. Внесите в инструкцию, вызывающую метод Divide класса Calculator, изменения, выделенные жирным шрифтом:

```
static void Main(string[] args)
{
    using (Calculator calculator = new Calculator())
    {
        Console.WriteLine($"120 / 0 = {calculator.Divide(120, 0)}");
    }
    Console.WriteLine("Program finishing");
}
```

В измененной инструкции предпринимается попытка деления 120 на 0.

Щелкните в меню Отладка на пункте Запуск без отладки. Приложение вполне ожидаемо выдаст необработанное исключение DivideByZeroException.

Щелкните в окне сообщения GarbageCollectionDemo на кнопке Отмена (рис. 14.1). (Нужно успеть сделать это до появления кнопок Отладка и Закрыть программу.)

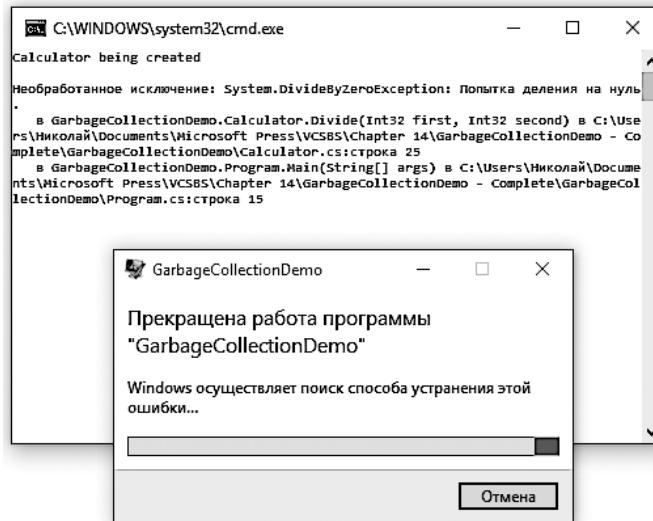


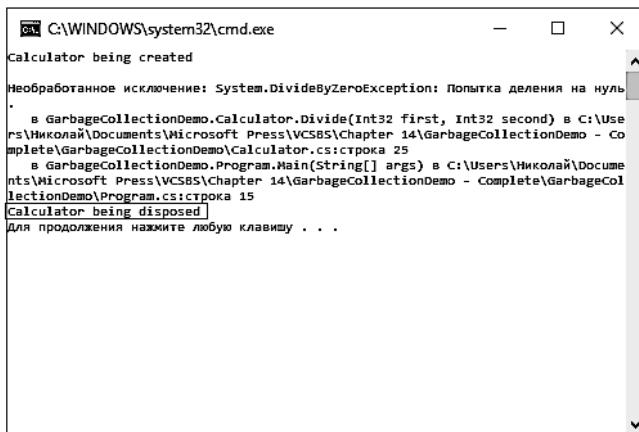
Рис. 14.1

Убедитесь в том, что в окне консоли сообщение «Calculator being disposed» появляется после необработанного исключения (рис. 14.2).



ПРИМЕЧАНИЕ Если вы промедлили и уже появились кнопки Отладка и Закрыть программу, щелкните на кнопке Закрыть программу и еще раз запустите приложение без отладки.

Нажмите в окне консоли клавишу Ввод и вернитесь в среду Visual Studio 2015.

**Рис. 14.2**

Выводы

В этой главе была показана работа сборщика мусора и то, как среда .NET Framework использует его для высвобождения выделенных объектам ресурсов, очищая память. Вы научились создавать деструкторы, предназначенные для очистки используемых объектом ресурсов в ходе возвращения сборщиком мусора памяти для ее повторного использования. Вы также увидели, как используется инструкция `using` для реализации высвобождения ресурсов независимо от выдачи исключений и как реализуется интерфейс `IDisposable` для поддержки этой формы высвобождения ресурсов объекта.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 15 «Реализация свойств для доступа к полям».

Если хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Если увидите диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Создать деструктор	Напишите метод, имя которого совпадает с именем класса и имеет префикс в виде тильды (~). У этого метода не должно быть модификатора доступа (например, <code>public</code>), и у него не может быть никаких параметров и возвращаемого значения, например:

Чтобы	Сделайте следующее
	<pre>class Example { ~Example() { ... } }</pre>
Вызвать деструктор	Вы не можете вызвать деструктор. Это может сделать только сборщик мусора
Принудительно вызывать сборку мусора (не рекомендуется)	Вызовите GC.Collect
Высвободить ресурс в известный момент времени (но с риском утечки ресурса в случае прерывания выполнения программы из-за выдачи исключений)	<p>Напишите метод высвобождения ресурсов (который высвобождает ресурс) и вызовите его из программы явным образом, например:</p> <pre>class TextReader { ... public virtual void Close() { ... } } class Example { void Use() { TextReader reader = ...; // использование переменной reader reader.Close(); } }</pre>
Поддержать в классе высвобождение ресурсов независимо от выдачи исключений	<p>Реализуйте интерфейс IDisposable, например:</p> <pre>class SafeResource : IDisposable { ... public void Dispose() { // здесь высвобождаются ресурсы } }</pre>
Выполнить независимое от выдачи исключений высвобождение ресурсов объекта, реализующего интерфейс IDisposable	<p>Создайте объект в инструкции using, например:</p> <pre>using (SafeResource resource = new SafeResource()) { // здесь используется SafeResource ... }</pre>

Часть III

Определение расширяемых типов в C#

В первых двух частях книги было представлено введение в основной синтаксис языка C#, и в них показывалось, как новые типы создаются с помощью структур, перечислений и классов. Кроме этого вы узнали, как при выполнении программы общязыковая среда выполнения (common language runtime (CLR)) управляет памятью, выделенной под переменные и объекты, и теперь уже должны разбираться в жизненном цикле объектов C#. Материалы глав части III, выстроенные на основе этой информации, дают представление об использовании C# для создания расширяемых компонентов — типов данных с широкими функциональными возможностями, пригодных к повторному использованию во многих приложениях.

В части III книги вы изучите более совершенные средства C#: свойства, индексаторы, универсальные шаблоны и классы коллекций. Вы увидите, как с помощью событий создаются адаптивные системы и как делегаты используются для запуска логики приложения, принадлежащей одному классу, из другого класса без сильной связанности классов, то есть как применяется высокоэффективная технология, позволяющая выстраивать расширяемые системы. Также вами будет изучен полностью интегрированный в C# язык запросов (Language-Integrated Query (LINQ)), позволяющий конструировать сложные запросы к коллекциям объектов понятным и естественным образом. Вы также узнаете, как перегружаются операторы с целью адаптации способа функционирования обычных операторов C# к вашим собственным классам и структурам.

15 Реализация свойств для доступа к полям

Прочитав эту главу, вы научитесь:

- инкапсулировать логические поля с помощью свойств;
- управлять доступом к свойствам для чтения из них информации, объявляя методы доступа `get`;
- управлять доступом к свойствам для записи в них информации, объявляя методы доступа `set`;
- создавать интерфейсы, объявляющие свойства;
- реализовывать интерфейсы, содержащие свойства, путем использования структур и классов;
- создавать свойства в автоматическом режиме на основе определения полей;
- использовать свойства для инициализации объектов.

В этой главе рассматриваются способы определения и использования свойств для инкапсуляции полей и данных в классе. В предыдущих главах привлекалось внимание к тому, что нужно поля в классе делать закрытыми и предоставлять методы для хранения в полях значений, а также извлечения этих значений из полей. Такой подход гарантирует безопасный и контролируемый доступ к полям, и его можно использовать для инкапсуляции дополнительной логики и правил, применяемых к допустимым значениям. Но синтаксис для доступа к полю таким способом выглядит не вполне естественно. Когда нужно считать или записать значение переменной, обычно используется инструкция присваивания, поэтому вызов метода для получения такого же эффекта в отношении поля (которое по своей сути представляет собой переменную) выглядит неуклюжим. Для устранения этой несущности и были разработаны свойства.

Реализация инкапсуляции путем использования методов

Сначала давайте вернемся к исходной мотивации использования методов для сокрытия полей.

Рассмотрим следующую структуру, представляющую позицию на экране компьютера в виде пары координат x и y . Предположим, допустимые значения для координаты x находятся в диапазоне от 0 до 1280, а для координаты y — от 0 до 1024:

```
struct ScreenPosition
{
    public int X;
    public int Y;

    public ScreenPosition(int x, int y)
    {
        this.X = rangeCheckedX(x);
        this.Y = rangeCheckedY(y);
    }

    private static int rangeCheckedX(int x)
    {
        if (x < 0 || x > 1280)
        {
            throw new ArgumentOutOfRangeException("X");
        }
        return x;
    }

    private static int rangeCheckedY(int y)
    {
        if (y < 0 || y > 1024)
        {
            throw new ArgumentOutOfRangeException("Y");
        }
        return y;
    }
}
```

Одной из проблем, связанных с этой структурой, является то, что она не следует золотому правилу инкапсуляции, то есть не хранит свои данные в закрытом состоянии. Открытые данные зачастую далеко не лучший вариант, поскольку класс не в состоянии контролировать значения, указываемые приложением. Например, конструктор `ScreenPosition` проверяет свои параметры, убеждаясь, что они находятся в указанном диапазоне, но такую проверку невозможно выполнить при необработанном доступе к открытому полю. Рано или поздно (вероятнее всего, совсем скоро) ошибка или недопонимание разработчиком

принципов использования этого класса в приложении приведут к тому, что либо X, либо Y выйдет за пределы своего диапазона:

```
ScreenPosition origin = new ScreenPosition(0, 0);
...
int xpos = origin.X;
origin.Y = -100; // вот так!
```

Стандартный способ решения данной проблемы заключается в том, чтобы сделать эти поля закрытыми и добавить к ним метод доступа и метод изменения, чтобы выполнять чтение и запись значения каждого закрытого поля соответственно. После чего методы изменения смогут проверять диапазон значений для нового поля. Например, следующий код содержит для поля X метод доступа (`GetX`) и метод изменения (`SetX`). Обратите внимание на то, как метод `SetX` проверяет переданный ему параметр:

```
struct ScreenPosition
{
    ...
    public int GetX()
    {
        return this.x;
    }

    public void SetX(int newX)
    {
        this.x = rangeCheckedX(newX);
    }
    ...
    private static int rangeCheckedX(int x) { ... }
    private static int rangeCheckedY(int y) { ... }
    private int x, y;
}
```

Теперь в соответствии с нашими требованиями в коде успешно применяются ограничения, соответствующие диапазону допустимых значений. Но за полезные гарантии приходится платить — у `ScreenPosition` теперь нет естественного синтаксиса, присущего полям, а вместо него используется неудобный синтаксис на основе методов. В следующем примере значение X увеличивается на 10 единиц. Чтобы добиться нужного результата, нужно прочитать значение X, воспользовавшись методом доступа `GetX`, а затем записать значение X, воспользовавшись методом изменения `SetX`.

```
int xpos = origin.GetX();
origin.SetX(xpos + 10);
```

Сравним это с эквивалентным кодом, используемым при открытом поле X:

```
origin.X += 10;
```

Несомненно, в данном случае использование открытых полей синтаксически понятнее, короче и легче, но, к сожалению, это нарушает инкапсуляцию. Используя свойства, можно сочетать все лучшие стороны обоих миров (поля и методов), сохраняя инкапсуляцию и предоставляя синтаксис, который похож на тот, что используется с полями.

Что такое свойства?

Свойство — это гибрид поля и метода: оно выглядит как поле, но работает как метод. Доступ к свойству осуществляется с использованием точно такого же синтаксиса, какой используется для доступа к полю. Но компилятор автоматически переводит синтаксис, похожий на тот, который используется для поля, в вызов методов доступа, которые иногда называются получателями (getters) и установщиками (setters) свойств.

Синтаксис для объявления свойства выглядит следующим образом:

```
AccessModifier Type PropertyName
{
    get
    {
        // код метода доступа к чтению свойства
    }

    set
    {
        // код метода доступа к записи свойства
    }
}
```

В свойстве могут содержаться два блока кода, начинающиеся с ключевых слов `get` и `set`. Блок `get` содержит инструкции, выполняемые при чтении свойства, а блок `set` содержит инструкции, выполняемые при записи в свойство. Тип свойства определяет тип данных, считываемых из этого свойства или записываемых в него, методами доступа `get` и `set`.

В следующем примере кода показана структура `ScreenPosition`, переписанная с использованием свойств. Изучая код, обратите внимание на следующее:

- ❑ записанные символами в нижнем регистре `_x` и `_y` являются закрытыми полями;
- ❑ записанные символами в верхнем регистре `X` и `Y` являются открытыми свойствами;
- ❑ все данные, передаваемые установленным методам доступа, должны быть записаны с помощью скрытого встроенного параметра по имени `value`:

```

struct ScreenPosition
{
    private int _x, _y;

    public ScreenPosition(int X, int Y)
    {
        this._x = rangeCheckedX(X);
        this._y = rangeCheckedY(Y);
    }

    public int X
    {
        get { return this._x; }
        set { this._x = rangeCheckedX(value); }
    }

    public int Y
    {
        get { return this._y; }
        set { this._y = rangeCheckedY(value); }
    }

    private static int rangeCheckedX(int x) { ... }
    private static int rangeCheckedY(int y) { ... }
}

```

В данном примере непосредственная реализация каждого свойства осуществляется с использованием закрытого поля, но это лишь один из способов реализации свойства. Требования ограничиваются только тем, что метод доступа `get` должен возвращать значение заданного типа. Вместо того чтобы просто извлекаться из сохранных данных, значение может легко вычисляться в динамическом режиме, и в таком случае надобность в наличии физического поля отпадает.



ПРИМЕЧАНИЕ Хотя примеры в данной главе показывают способы определения свойств для структур, они в равной степени применимы и к классам, для которых используется точно такой же синтаксис.

ПРЕДУПРЕЖДЕНИЕ, КАСАЮЩЕЕСЯ ИМЕН СВОЙСТВ И ПОЛЕЙ

В главе 2 «Работа с переменными, операторами и выражениями» в разделе «Правила присваивания имен переменным» рассматривался ряд рекомендаций по присваиванию имен переменным. В частности, там утверждалось, что использование в начале имени идентификатора символа подчеркивания следует избегать. Но, как видите, в структуре `ScreenPosition` допущено отступление от этой рекомендации, поскольку здесь содержатся поля с именами `_x` и `_y`. Для этого исключения из правил есть довольно веская причина. Во врезке «Задание имен и доступность» в главе 7 «Создание классов и объектов и управление

ими» утверждалось, что идентификаторы, начинающиеся с буквы в верхнем регистре, чаще всего используются для открытых методов и полей, а идентификаторы, начинающиеся с буквы в нижнем регистре, — для закрытых методов и полей. Если придерживаться обоих правил, это может привести к тому, что имена свойств и закрытых полей будут различаться лишь регистром первой буквы, и во многих организациях такое положение вещей считается вполне допустимым.

Если в вашей организации используется такой же подход, следует иметь в виду один его существенный недостаток. Посмотрите на следующий код, в котором реализуется класс по имени Employee. Поле employeeID является закрытым, а свойство EmployeeID предоставляет открытый доступ к нему:

```
class Employee
{
    private int employeeID;
    public int EmployeeID
    {
        get { return this.EmployeeID; }
        set { this.EmployeeID = value; }
    }
}
```

Код без проблем откомпилируется, но в результате при обращении к свойству EmployeeID программа выдаст исключение переполнения стека — StackOverflowException. Дело в том, что методы доступа get и set относятся к свойству (имя которого начинается с большой буквы «E»), а не к закрытому полю (имя которого начинается с маленькой буквы «e»), что вызывает бесконечный рекурсивный цикл, который в итоге приведет к исчерпанию доступной памяти. Подобную ошибку отследить очень трудно! Поэтому в примерах данной книги для закрытых полей, используемых для предоставления свойствам данных, применяются имена с лидирующим символом подчеркивания, из-за чего их становится намного проще отличить от имен свойств. Для всех остальных закрытых полей будут по-прежнему использоваться идентификаторы в смешанном регистре без начального символа подчеркивания.

Использование свойств

В выражении свойство можно использовать в контексте чтения (когда его значение извлекается) и в контексте записи (когда его значение изменяется). В следующем примере показано, как значение читается из свойств X и Y структуры ScreenPosition:

```
ScreenPosition origin = new ScreenPosition(0, 0);
int xpos = origin.X; // вызывается origin.X.get
int ypos = origin.Y; // вызывается origin.Y.get
```

Обратите внимание на то, что для обращения к свойствам и полям используется одинаковый синтаксис. Когда свойство используется в контексте чтения, компилятор автоматически преобразует ваш код, похожий на код, использующий поле, в вызов метода `get` этого свойства. Аналогично этому, если используется контекст записи, компилятор автоматически преобразует ваш код, похожий на код, использующий поле, в вызов метода `set` этого свойства:

```
origin.X = 40; // вызывается origin.X.set со значением value, равным 40
origin.Y = 100; // вызывается origin.Y.set со значением value, равным 100
```

Как упоминалось в предыдущем разделе, присваиваемые значения передаются в метод доступа `set` с использованием переменной `value`. Среда исполнения делает это автоматически.

Свойство можно использовать также в контексте чтения/записи. В таком случае используются оба метода доступа, как `get`, так и `set`. Например, компилятор автоматически переводит такие инструкции, как следующая, в вызовы методов доступа `get` и `set`:

```
origin.X += 10;
```



СОВЕТ Статические свойства можно объявлять точно так же, как и статические поля и методы. Обратиться к статическим свойствам можно с использованием имени класса или структуры, а не их экземпляра.

Свойства только для чтения

Можно объявить свойство, содержащее только метод доступа `get`. В таком случае свойство можно будет использовать только в контексте чтения. Например, здесь свойство `X` структуры `ScreenPosition` объявлено как свойство только для чтения:

```
struct ScreenPosition
{
    private int _x;
    ...
    public int X
    {
        get { return this._x; }
    }
}
```

У свойства `X` нет метода доступа `set`, поэтому, как показано в следующем примере, любая попытка использования `X` в контексте записи будет неудачной:

```
origin.X = 140; // ошибка в ходе компиляции
```

Свойства только для записи

Аналогичным образом можно объявить свойство, содержащее лишь метод доступа `set`. В таком случае свойство можно будет использовать только в контексте записи. Например, в следующем фрагменте кода свойство `X` структуры `ScreenPosition` объявлено как свойство только для записи:

```
struct ScreenPosition
{
    private int _x;
    ...
    public int X
    {
        set { this._x = rangeCheckedX(value); }
    }
}
```

В свойстве `X` не содержится метод доступа `get`, и как показано в следующем примере, любая попытка использования `X` для чтения закончится неудачей:

```
Console.WriteLine(origin.X); // ошибка в ходе компиляции
origin.X = 200;             // компиляция проходит успешно
origin.X += 10;              // ошибка в ходе компиляции
```



ПРИМЕЧАНИЕ Свойства, предназначенные только для чтения, хорошо подходят для конфиденциальных данных, таких как пароли. В идеале приложение, принимающее меры безопасности, должно позволить устанавливать пароль, но никогда не должно позволять его считывать. При попытке войти в систему пользователь должен сообщить пароль. Метод, выполняющий регистрацию, может сравнить этот пароль с сохраненным паролем и должен возвращать лишь свидетельство об их совпадении.

Доступность свойств

Степень доступности свойства можно указать при его объявлении, воспользовавшись ключевыми словами `public`, `private` или `protected`. Но внутри объявления свойства доступность для методов доступа `get` и `set` можно переопределить. Например, показанная в следующем примере кода структура `ScreenPosition` объявляет методы доступа `set` свойств `X` и `Y` закрытыми (`private`). (Методы доступа `get` являются открытыми, потому что открытыми являются сами свойства.)

```
struct ScreenPosition
{
    private int _x, _y;
    ...

    public int X
    {
        get { return this._x; }
    }
}
```

```

        private set { this._x = rangeCheckedX(value); }
    }

    public int Y
    {
        get { return this._y; }
        private set { this._y = rangeCheckedY(value); }
    }
    ...
}

```

При определении методов доступа с различной степенью доступности следует соблюдать ряд правил.

- ❑ При определении методов доступность можно изменить только одному из них. Нет смысла определять свойство открытым только для того, чтобы изменить доступность обоих методов, объявляя их закрытыми.
- ❑ Модификатор доступности не должен определять доступность, накладывающую меньшее ограничение, чем то, что наложено на само свойство. Например, если свойство объявлено закрытым (`private`), метод доступа для чтения не может быть открытым (`public`). (Вместо этого следовало бы объявить свойство открытым, а метод доступа для записи — закрытым.)

Основные сведения об ограничениях, накладываемых на свойства

Когда данныечитываются и записываются с использованием свойств, эти свойства выглядят, работают и воспринимаются как поля. Но на самом деле они не являются полями, и на них накладываются определенные ограничения.

- ❑ Присвоить значение через свойство структуры или класса можно только после инициализации структуры или класса. Следующий пример кода некорректен, потому что переменная `location` не была инициализирована (с использованием ключевого слова `new`):

```

ScreenPosition location;
location.X = 40; // ошибка в ходе компиляции, значение location
                 // не присваивается

```



ПРИМЕЧАНИЕ Может быть, это и покажется несущественным, но если бы идентификатор `X` указывал на поле, а не на свойство, то код был бы вполне работоспособным. Поэтому структуры и классы с самого начала нужно определять с использованием свойств, а не полей, которые позже вы будете превращать в свойства. Код, использующий ваши структуры и классы, после превращения полей в свойства может утратить работоспособность. Данный вопрос будет рассматриваться также в разделе «Создание свойств в автоматическом режиме».

- Свойства нельзя использовать в качестве `ref`- или `out`-аргумента метода (хотя поле с доступом для записи в данном качестве использовать можно). В этом есть определенный смысл, поскольку свойство на самом деле указывает не на место в памяти, а на метод доступа, поэтому код следующего примера некорректен:

```
MyMethod(ref location.X); // ошибка в ходе компиляции
```

- Свойство должно содержать не более одного метода доступа `get` и не более одного метода доступа `set`. Свойство не может содержать другие методы, поля или свойства.
- Методы доступа `get` и `set` не могут принимать параметры. Присваиваемые данные передаются методу доступа `set` автоматически с использованием переменной `value`.
- Свойства нельзя объявлять с использованием ключевого слова `const`:

```
const int X { get { ... } set { ... } } // ошибка в ходе компиляции
```

ПРАВИЛЬНОЕ ИСПОЛЬЗОВАНИЕ СВОЙСТВ

Свойства являются весьма эффективным средством программирования и при правильном использовании позволяют сделать код понятнее и легче в сопровождении. Но они не подменяют собой тщательно выверенную объектно-ориентированную конструкцию, основное внимание в которой уделяется поведению объектов, а не их свойствам. Сам по себе доступ к закрытым полям посредством обычных методов или свойств не превращает ваш код в удачно спроектированную программу. К примеру, на банковском счете хранится баланс, показывающий объем доступных средств. Это может побудить вас к созданию свойства по имени `Balance`, принадлежащего классу `BankAccount`:

```
class BankAccount
{
    private decimal _balance;
    ...
    public decimal Balance
    {
        get { return this._balance; }
        set { this._balance = value; }
    }
}
```

Подобное проектное решение нельзя признать удачным, поскольку в нем не отражается функциональная возможность, необходимая при снятии средств со счета или его пополнении. (Если вам известен банк, позволяющий изменять баланс напрямую, без физического помещения денег на счет, пожалуйста, сообщите мне об этом!) Страйтесь в процессе программирования выразить в решении суть ре-

шаемой задачи, не заблудившись в большом объеме низкоуровневого синтаксиса. Как показано в следующем примере, лучше вместо написания метода доступа по установке значения свойства предусмотреть наличие в классе BankAccount методов Deposit и Withdraw:

```
class BankAccount
{
    private decimal _balance;
    ...
    public decimal Balance { get { return this._balance; } }
    public void Deposit(decimal amount) { ... }
    public bool Withdraw(decimal amount) { ... }
}
```

Объявление свойств интерфейса

Интерфейсы рассматривались в главе 13 «Создание интерфейсов и определение абстрактных классов». В интерфейсах могут объявляться как свойства, так и методы. Для этого нужно указать ключевое слово `get` или `set` или оба этих слова, но вместо тела методов доступа `get` или `set` следует поставить точку с запятой:

```
interface IScreenPosition
{
    int X { get; set; }
    int Y { get; set; }
}
```

Любой класс или структура, реализующие этот интерфейс, должны реализовывать также свойства X и Y с методами доступа `get` и `set`.

```
struct ScreenPosition : IScreenPosition
{
    ...
    public int X
    {
        get { ... }
        set { ... }
    }

    public int Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Если свойства интерфейса реализуются в классе, можно объявить реализации свойств виртуальными, что позволит производным классам переопределять эти реализации:

```
class ScreenPosition : IScreenPosition
{
    ...
    public virtual int X
    {
        get { ... }
        set { ... }
    }

    public virtual int Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```



ПРИМЕЧАНИЕ В данном примере показан класс. Следует напомнить, что использовать ключевое слово `virtual` при создании структур нельзя, потому что структуры не поддерживают наследование.

Можно также выбрать вариант реализации свойства путем использования синтаксиса явного определения интерфейса, рассмотренного в главе 13. Явная реализация свойства не должна иметь модификаторов `public` и `virtual` и не может быть переопределена:

```
struct ScreenPosition : IScreenPosition
{
    ...
    int IScreenPosition.X
    {
        get { ... }
        set { ... }
    }

    int IScreenPosition.Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Замена методов свойствами

Изучая главу 13, вы научились создавать приложение для рисования фигур, с помощью которого пользователь может помещать на холст в окне круги и квадраты. В упражнениях, приводимых в этой главе, функциональность классов `Circle` и `Square` выносится в абстрактный класс по имени `DrawingShape`. Этот класс предоставляет методы `SetLocation` и `SetColor`, которые используются

приложением для указания позиции и цвета фигуры на экране. В следующем упражнении класс `DrawingShape` будет изменен, чтобы он мог представлять местоположение и цвет фигуры в виде свойств.

Использование свойств

Откройте в среде Visual Studio 2015 проект Drawing, который находится в папке `\Microsoft Press\VCSBS\Chapter 15\Drawing Using Properties` вашей папки документов. Выберите в окне редактора файл `DrawingShape.cs`. В нем содержится почти такой же класс `DrawingShape`, какой был показан в главе 13, за исключением того, что в нем учтены рекомендации, рассмотренные ранее в данной главе, и поле `size` переименовано в `_size`, а поля `locX` и `locY` — в `_x` и `_y`:

```
abstract class DrawingShape
{
    protected int _size;
    protected int _x = 0, _y = 0;
    ...
}
```

Откройте в окне редактора файл `IDraw.cs` проекта Drawing. Интерфейс в этом файле дает следующее определение метода `SetLocation`:

```
interface IDraw
{
    void SetLocation(int xCoord, int yCoord);
    ...
}
```

Данный метод предназначен для установки для полей `_x` и `_y` объекта типа `DrawingShape` переданных ему значений. Этот метод может быть заменен двумя свойствами. Удалите метод `SetLocation` и поставьте на его место определение двух свойств с именами `X` и `Y`, выделенное здесь жирным шрифтом:

```
interface IDraw
{
    int X { get; set; }
    int Y { get; set; }
    ...
}
```

Удалите в классе `DrawingShape` метод `SetLocation` и поставьте вместо него следующие реализации свойств `X` и `Y`:

```
public int X
{
    get { return this._x; }
    set { this._x = value; }
}
```

```
public int Y
{
    get { return this._y; }
    set { this._y = value; }
}
```

Выполните в окне редактора файл DrawingPad.xaml.cs и найдите в нем метод `drawingCanvas_Tapped`. Этот метод запускается при касании экрана пальцем или щелчке левой кнопкой мыши. Он рисует на экране квадрат в точке касания или в точке нахождения указателя мыши при щелчке. Найдите инструкцию, вызывающую метод `SetLocation` для установки позиции квадрата на экране. Она находится в блоке инструкции `if` и выделена в следующем фрагменте кода жирным шрифтом:

```
if (mySquare is IDraw)
{
    IDraw drawSquare = mySquare;
    drawSquare.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
    drawSquare.Draw(drawingCanvas);
}
```

Замените инструкцию кодом, устанавливающим свойства `X` и `Y` объекта `Square`, выделенным здесь жирным шрифтом:

```
if (mySquare is IDraw)
{
    IDraw drawSquare = mySquare;
    drawSquare.X = (int)mouseLocation.X;
    drawSquare.Y = (int)mouseLocation.Y;
    drawSquare.Draw(drawingCanvas);
}
```

Найдите метод `drawingCanvas_RightTapped`. Он запускается при касании экрана пальцем и удержании пальца в точке касания или при щелчке правой кнопкой мыши. Он рисует на экране круг в точке касания и удержания пальца или в точке нахождения указателя мыши при щелчке.

Замените в методе инструкцию, вызывающую метод `SetLocation` объекта типа `Circle`, на установку значений для свойств `X` и `Y`, выделенную в следующем примере кода жирным шрифтом:

```
if (myCircle is IDraw)
{
    IDraw drawCircle = myCircle;
    drawCircle.X = (int)mouseLocation.X;
    drawCircle.Y = (int)mouseLocation.Y;
    drawCircle.Draw(drawingCanvas);
}
```

Откройте в окне редактора файл `IColor.cs`. Интерфейс в этом файле дает следующее определение метода `SetColor`:

```
interface IColor
{
    void SetColor(Color color);
}
```

Удалите этот метод и поставьте на его место определение свойства по имени `Color`:

```
interface IColor
{
    Color Color { set; }
}
```

Это предназначено только для записи свойство предоставляет метод доступа `set`, но не предоставляет метод доступа `get`. Свойство определяется именно таким образом, потому что цвет еще не хранится в классе `DrawingShape` и указывается только при прорисовке каждой фигуры, то есть вы не можете сделать запрос к фигуре, чтобы определить, какого она цвета.



ПРИМЕЧАНИЕ По устоявшейся практике свойству дается такое же имя, что и типу (в данном случае это `Color`).

Вернитесь в окне редактора к классу `DrawingShape`. Замените в нем метод `SetColor` показанным здесь свойством `Color`:

```
public Color Color
{
    set
    {
        if (this.shape != null)
        {
            SolidColorBrush brush = new SolidColorBrush(value);
            this.shape.Fill = brush;
        }
    }
}
```



ПРИМЕЧАНИЕ Код для метода доступа `set` практически такой же, как и для исходного метода `SetColor`, за исключением того, что инструкции, создающей объект `SolidColorBrush`, передается параметр `value`.

Вернитесь в окне редактора к файлу `DrawingPad.xaml.cs`. Измените в методе `drawingCanvas_Tapped` код инструкции, устанавливающей цвет `Square`-объекта, чтобы он соответствовал коду, выделенному здесь жирным шрифтом:

```
if (mySquare is IColor)
{
    IColor colorSquare = mySquare;
    colorSquare.Color = Colors.BlueViolet;
}
```

Измените по аналогии с этим инструкцию, устанавливающую цвет Circle-объекта в методе drawingCanvas_RightTapped:

```
if (myCircle is IColor)
{
    IColor colorCircle = myCircle;
    colorCircle.Color = Colors.HotPink;
}
```

Щелкните в меню Отладка на пункте Начать отладку, чтобы инициировать сборку и запуск проекта.

Убедитесь в том, что приложение работает точно так же, как и раньше. При касании экрана или щелчке левой кнопкой мыши на холсте приложение должно рисовать квадрат, а при касании и удержании пальца или щелчке правой кнопкой мыши — окружность. При работе приложения должно получаться следующее изображение (рис. 15.1).

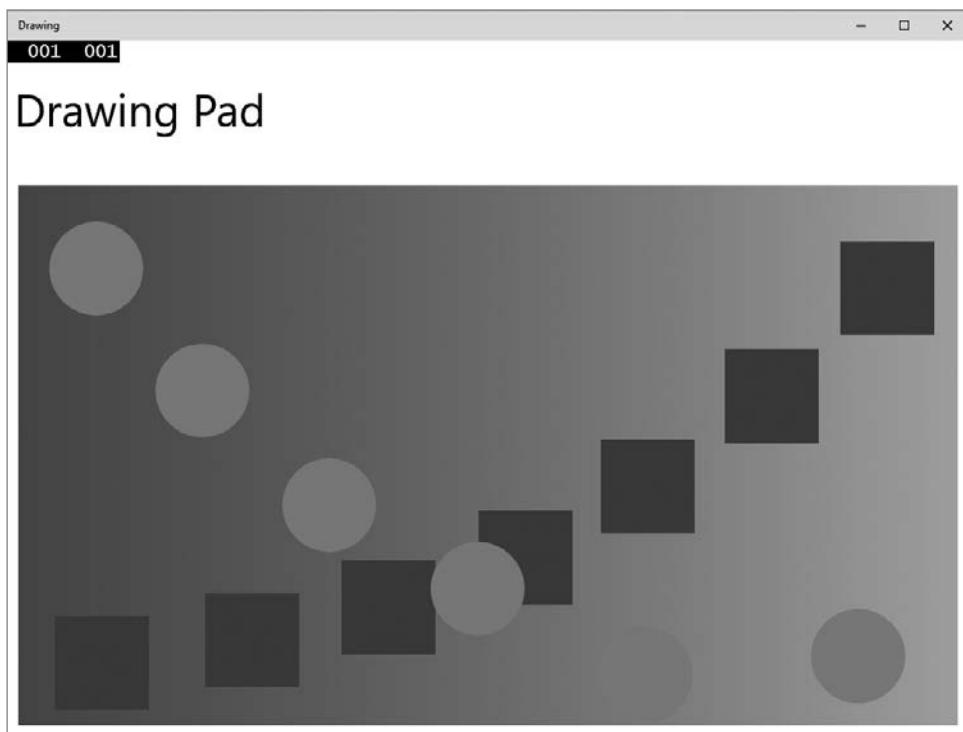


Рис. 15.1

Вернитесь в среду Visual Studio 2015 и остановите отладку.

Создание свойств в автоматическом режиме

Как уже упоминалось, основная задача свойств заключается в том, чтобы скрыть реализацию полей от внешнего мира. Хорошо, когда свойства действительно выполняют некую полезную работу, но если методы доступа `get` и `set` просто заключают в свой блок кода операции, которые всего лишь считывают значение поля или присваивают этому полю значение, ценность такого подхода может оказаться сомнительной. Но даже в таких ситуациях есть как минимум две веские причины, по которым следует определять свойства, а не выставлять данные в качестве открытых полей.

- ❑ **Совместимость с приложениями.** Поля и свойства выставляются путем использования в сборках различных метаданных. Если при разработке класса принимается решение об использовании открытых полей, то любые приложения, использующие этот класс, будут ссылаться на эти элементы как на поля. Хотя для чтения и записи полей в C# используется точно такой же синтаксис, как и для чтения и записи свойств, скомпилированный код различается, просто компилятор C# скрывает от вас эти различия. Если позже прийти к решению, что нужно заменить поля свойствами (возможно, в связи с изменением каких-то бизнес-требований и появлением необходимости в реализации дополнительной логики при присваивании значений), существующее приложение не сможет воспользоваться обновленной версией класса без перекомпиляции. Если приложение развернуто на большом количестве устройств в масштабе всей организации, возникнут затруднения. Существуют, конечно, обходные пути, но лучше все же в первую очередь побеспокоиться о том, чтобы не попасть в подобную ситуацию.
- ❑ **Совместимость с интерфейсами.** При реализации интерфейса, в котором элемент определен как свойство, нужно создать свойство, соответствующее спецификации в интерфейсе, даже если свойство просто считывает данные из закрытого поля и записывает их в него. Вы не можете реализовать свойство просто путем выставления открытого поля с таким же именем.

Разработчики языка C# понимали, что программисты — люди занятые и их не стоит заставлять попусту тратить время на написание не нужного им кода. По этой причине компилятор C# способен создавать код для свойств автоматически:

```
class Circle
{
    public int Radius{ get; set; }
    ...
}
```

В этом примере в классе `Circle` содержится свойство по имени `Radius`. Вы указали лишь тип этого свойства, не указав, как оно работает, поскольку методы `get` и `set` оставлены пустыми. Компилятор C# преобразует это определение в закрытое поле и в исходную реализацию, имеющую следующий вид:

```
class Circle
{
    private int _radius;
    public int Radius{
        get
        {
            return this._radius;
        }
        set
        {
            this._radius = value;
        }
    }
    ...
}
```

Таким образом, приложив минимум усилий, вы можете реализовать простое свойство, создав код автоматически, и если позже понадобится включить дополнительную логику, это можно будет сделать, не ломая существующих приложений.



ПРИМЕЧАНИЕ Синтаксис для определения автоматически создаваемого свойства практически идентичен синтаксису для определения свойства в интерфейсе. Исключение состоит в том, что в автоматически создаваемом свойстве можно указывать такие модификаторы доступа, как `private`, `public` или `protected`.

Автоматическое создание свойства, предназначенного только для чтения, можно инициировать, не указывая в объявлении метода доступа `set`:

```
class Circle
{
    public DateTime CircleCreatedDate { get; }
    ...
}
```

Этот прием пригодится, если понадобится создать неизменяемое свойство, значение которого устанавливается при конструировании объекта и впоследствии не может быть изменено. Например, вам может понадобиться установить дату создания объекта или имя пользователя, создавшего объект, или, возможно, вам потребуется задать для объекта значение уникального идентификатора. Существуют объекты, значения которых желательно устанавливать только один раз, а затем не позволять их изменять. Исходя из этого C# позволяет вам инициализировать автоматически создаваемое свойство, предназначенное только для чтения, одним из двух способов. Свойство можно инициализировать из конструктора:

```
class Circle
{
    public Circle()
```

```

{
    CircleCreatedDate = DateTime.Now;
}

public DateTime CircleCreatedDate { get; }
...
}

```

Или же его можно инициализировать при объявлении:

```

class Circle
{
    public DateTime CircleCreatedDate { get; } = DateTime.Now;
    ...
}

```

Следует иметь в виду, что при инициализации свойства таким образом наряду с установкой его значения в конструкторе значение, предоставленное в конструкторе, перепишет значение, указанное в инициализаторе свойства, поэтому следует использовать один из этих подходов, но не оба сразу!



ПРИМЕЧАНИЕ Автоматически создавать свойства только для записи невозможно. При попытке автоматического создания свойства без метода доступа `get` будет выдана ошибка в ходе компиляции.

Инициализация объектов путем использования свойств

В главе 7 вы научились для инициализации объектов определять конструкторы. У объекта может быть несколько конструкторов, которые можно определять с различными параметрами для инициализации различных элементов объекта. Например, можно определить класс, моделирующий треугольник:

```

public class Triangle
{
    private int side1Length;
    private int side2Length;
    private int side3Length;

    // пассивный конструктор – исходные значения для всех сторон
    public Triangle()
    {
        this.side1Length = this.side2Length = this.side3Length = 10;
    }

    // указание длины для side1Length и исходные значения для других сторон
    public Triangle(int length1)
    {

```

```
this.side1Length = length1;
this.side2Length = this.side3Length = 10;
}

// Указание длины для side1Length и side2Length,
// исходное значение для side3Length
public Triangle(int length1, int length2)
{
    this.side1Length = length1;
    this.side2Length = length2;
    this.side3Length = 10;
}

// Указание длины для всех сторон
public Triangle(int length1, int length2, int length3)
{
    this.side1Length = length1;
    this.side2Length = length2;
    this.side3Length = length3;
}
}
```

В зависимости от количества полей, содержащихся в классе, и различных комбинаций, которые вам хотелось бы разрешить использовать при инициализации полей, вы можете прийти к написанию множества конструкторов. Если у многих полей один и тот же тип, образуется потенциальная проблема: у вас может отсутствовать возможность написания уникального конструктора для всех комбинаций полей. Например, в рассмотренном ранее классе `Triangle` вы не можете просто добавить конструктор, инициализирующий только поля `side1Length` и `side3Length`, поскольку у него не может быть уникальной сигнатуры — он бы получал два `int`-параметра, и точно такую же сигнатуру уже имеет конструктор, инициализирующий `side1Length` и `side2Length`. Одно из возможных решений при создании `Triangle`-объекта заключается в определении конструктора, принимающего необязательные параметры, и в указании значений для параметров в виде поименованных аргументов. Но более удачным и понятным решением будет инициализация закрытых полей для установки исходных значений и выставление их в виде свойств:

```
public class Triangle
{
    private int side1Length = 10;
    private int side2Length = 10;
    private int side3Length = 10;

    public int Side1Length
    {
        set { this.side1Length = value; }
    }

    public int Side2Length
    {
```

```

        set { this.side2Length = value; }
    }

    public int Side3Length
    {
        set { this.side3Length = value; }
    }
}

```

При создании экземпляра класса его можно инициализировать указанием имен и значений для любых открытых свойств, имеющих методы доступа. Например, можно создать `Triangle`-объекты и инициализировать любую комбинацию трех сторон:

```

Triangle tri1 = new Triangle { Side3Length = 15 };
Triangle tri2 = new Triangle { Side1Length = 15, Side3Length = 20 };
Triangle tri3 = new Triangle { Side2Length = 12, Side3Length = 17 };
Triangle tri4 = new Triangle { Side1Length = 9, Side2Length = 12,
                             Side3Length = 15 };

```

Этот синтаксис известен под названием *инициализатора объекта*. При подобном вызове инициализатора объекта компилятор C# создает код, вызывающий пассивный конструктор, а затем вызывает метод доступа `set` каждого названного свойства с целью инициализации этого свойства указанным значением. Инициализаторы объектов можно указывать и в сочетании с активными конструкторами. Например, если класс `Triangle` предоставляет конструктор, получающий один строковый параметр, который описывает тип треугольника, то можно вызвать этот конструктор и наряду с этим инициализировать другие свойства:

```

Triangle tri5 = new Triangle("Equilateral triangle") { Side1Length = 3,
                                                       Side2Length = 3,
                                                       Side3Length = 3 };

```

Важно помнить, что первым запускается конструктор, а после этого устанавливаются свойства. Понимание этой последовательности необходимо в том случае, когда конструктор устанавливает для полей в объекте конкретные значения, а указываемые вами свойства эти значения изменяют.

В следующем упражнении будут показано, что инициализаторы объектов могут использоваться также с автоматически создаваемыми свойствами, не предназначенными только для чтения. В этом упражнении будет определен класс для моделирования правильных многоугольников, имеющих автоматически создаваемые свойства для предоставления доступа к информации о количестве сторон многоугольника и об их длине.



ПРИМЕЧАНИЕ Инициализировать таким образом автоматически создаваемые свойства, предназначенные только для чтения, невозможно, для этого придется воспользоваться одной из технологий, рассмотренных в предыдущем разделе.

Определение автоматически создаваемых свойств и использование инициализаторов объектов

Откройте в среде Visual Studio 2015 проект AutomaticProperties, который находится в папке \\Microsoft Press\\VCSBS\\Chapter 15\\AutomaticProperties вашей папки документов. В проекте AutomaticProperties содержится файл Program.cs, в котором находится определение класса Program с методами Main и doWork, уже встречавшимися в предыдущих упражнениях.

Щелкните правой кнопкой мыши в обозревателе решений на проекте AutomaticProperties, выберите пункт Добавить, а затем щелкните на пункте Класс, чтобы открылось диалоговое окно Добавить новый элемент – AutomaticProperties. Наберите в поле Имя строку Polygon.cs и щелкните на кнопке Добавить. Будет создан и добавлен в проект файл Polygon.cs, содержащий класс Polygon, код которого появится в окне редактора.

Добавьте к классу Polygon автоматически создаваемые свойства NumSides и SideLength, выделенные здесь жирным шрифтом:

```
class Polygon
{
    public int NumSides { get; set; }
    public double SideLength { get; set; }
}
```

Добавьте к классу Polygon следующий пассивный конструктор, выделенный жирным шрифтом:

```
class Polygon
{
    ...
    public Polygon()
    {
        this.NumSides = 4;
        this.SideLength = 10.0;
    }
}
```

Этот конструктор инициализирует исходными значениями поля NumSides и SideLength. В этом упражнении исходным многоугольником является квадрат со сторонами длиной 10 единиц.

Выполните в окно редактора файл Program.cs. Добавьте к методу doWork инструкции, выделенные здесь жирным шрифтом, заменив ими комментарий // TODO::

```
static void doWork()
{
    Polygon square = new Polygon();
    Polygon triangle = new Polygon { NumSides = 3 };
    Polygon pentagon = new Polygon { SideLength = 15.5, NumSides = 5 };
}
```

Эти инструкции создают `Polygon`-объекты. Переменная `square` инициализируется с помощью пассивного конструктора. Переменные `triangle` и `pentagon` также инициализируются с использованием пассивного конструктора, а затем этот код изменяет значения свойств, предоставляемых классом `Polygon`. В случае с переменной `triangle` для свойства `NumSides` устанавливается значение 3, но у свойства `SideLength` остается его исходное значение `10.0`. Для переменной `pentagon` код изменяет значения свойств `SideLength` и `NumSides`.

Добавьте к концу метода `doWork` следующий код, выделенный жирным шрифтом:

```
static void doWork()
{
    ...
    Console.WriteLine($"Square: number of sides is {square.NumSides}, length of
        each side is {square.SideLength}");
    Console.WriteLine($"Triangle: number of sides is {triangle.NumSides},
        length of each side is {triangle.SideLength}");
    Console.WriteLine($"Pentagon: number of sides is {pentagon.NumSides},
        length of each side is {pentagon.SideLength}");
}
```

Содержащиеся в нем инструкции выводят на экран значения свойств `NumSides` и `SideLength` для каждого объекта `Polygon`.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что выполняются сборка и запуск программы и она выводит в окне консоли показанные на рис. 15.2 сообщения.

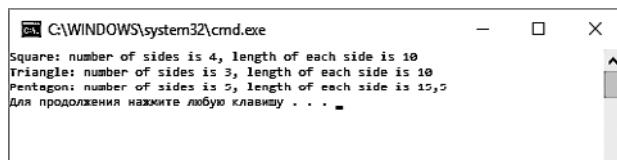


Рис. 15.2

Нажмите клавишу Ввод, чтобы закрыть приложение и вернуться в среду Visual Studio 2015.

Выводы

В этой главе было показано, как создаются и используются свойства, представляющие управляемый доступ к данным, находящимся в объекте. Вы также увидели, как свойства создаются в автоматическом режиме и как они используются для инициализации объектов.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 16 «Использование индексаторов».

Если сейчас хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Объявить для структуры или класса свойство, доступное для чтения и записи	Объявите тип свойства, его имя, а также методы доступа get и set, например: <pre>struct ScreenPosition { ... public int X { get { ... } set { ... } } ... }</pre>
Объявить для структуры или класса свойство, доступное только для чтения	Объявите свойство, указав для него только метод доступа get, например: <pre>struct ScreenPosition { ... public int X { get { ... } } ... }</pre>
Объявить для структуры или класса свойство, доступное только для записи	Объявите свойство, указав для него только метод доступа set, например: <pre>struct ScreenPosition { ... public int X { set { ... } } ... }</pre>

Чтобы	Сделайте следующее
Объявить свойство в интерфейсе	Объявите свойство, используя только ключевое слово get или set или же оба этих слова, например:
	<pre>interface IScreenPosition { int X { get; set; } // тела нет int Y { get; set; } // тела нет }</pre>
Реализовать свойство интерфейса в структуре или классе	Объявите в структуре или в классе, реализующем интерфейс, свойство и реализуйте его методы доступа, например:
	<pre>struct ScreenPosition : IScreenPosition { public int X { get { ... } set { ... } } public int Y { get { ... } set { ... } } }</pre>
Определить автоматически создаваемое свойство	Определите свойство в содержащем его классе или в структуре с пустыми методами доступа get и set, например:
	<pre>class Polygon { public int NumSides { get; set; } }</pre>
	Если определяется свойство, доступное только для чтения, его следует инициализировать либо в конструкторе объекта, либо сразу же при его определении, например:
	<pre>class Circle { public DateTime CircleCreatedDate { get; } = DateTime.Now; ... }</pre>
Использовать свойства для инициализации объекта	Укажите свойства и их значения в виде списка, заключенного в фигурные скобки, и поместите этот список в конструктор объекта, например:
	<pre>Triangle tri3 = new Triangle { Side2Length = 12, Side3Length = 17 };</pre>

16 Использование индексаторов

Прочитав эту главу, вы научитесь:

- инкапсулировать с помощью индексаторов логический доступ к объекту, работающий наподобие доступа к элементам массива;
- управлять доступом к индексаторам для чтения из них данных, объявляя методы доступа `get`;
- управлять доступом к индексаторам для записи в них данных, объявляя методы доступа `set`;
- создавать интерфейсы, объявляющие индексаторы;
- осуществлять реализацию в структурах и классах индексаторов, наследуемых из интерфейсов.

В главе 15 «Реализация свойств для доступа к полям» рассматривались способы реализации и использования свойств как средств предоставления управляемого доступа к полям класса. Свойства могут использоваться для зеркального отображения полей, содержащих единственное значение. Но индексаторы незаменимы при обеспечении доступа к элементам, содержащим несколько значений, и при реализации этого доступа с использованием понятного и знакомого синтаксиса.

Что такое индексатор?

Индексатор можно представить себе в качестве более разумно организованного массива, что во многом похоже на представление свойства в качестве более разумно организованного поля. Свойство инкапсулирует в классе одно значение, а индексатор делает то же самое для набора значений. Для индексатора используется точно такой же синтаксис, как и для массива.

Разобраться с тем, что такое индексатор, лучше всего на примере. Сначала будет рассмотрена задача, в которой не используются индексаторы. Затем та же задача

будет решена более эффективно за счет применения индексаторов. В задаче используются целые числа, или, если точнее, значения `int`-типа.

Пример без использования индексаторов

Обычно `int`-тип используется для хранения целочисленного значения. Внутри машины значения `int`-типа сохраняются в виде последовательности из 32 битов, где каждый бит может содержать либо 0, либо 1. Чаще всего это внутреннее двоичное представление вас абсолютно не касается, вы просто используете `int`-тип в качестве контейнера, содержащего целочисленное значение. Но иногда программисты применяют `int`-тип для других целей — некоторые из них используют `int` в качестве набора двоичных флагов и работают внутри значения `int`-типа с отдельными битами. Если вы такой же приверженец языка С старой закалки, как и я, то все, что вы прочтете дальше, навеет весьма живые воспоминания.



ПРИМЕЧАНИЕ В некоторых устаревших программах `int`-типы используются с целью экономии памяти. Эти программы создавались во времена, когда объемы компьютерной памяти измерялись в килобайтах, а не гигабайтах, как сегодня, и память была дефицитной. Тип `int` с одинарной точностью хранил 32 бита, каждый из которых мог быть либо единицей, либо нулем. В некоторых случаях программисты присваивали единицу, чтобы показать истинное значение (`true`), и нуль, чтобы показать ложное значение (`false`), а затем использовали `int`-тип как набор булевых значений.

В C# предоставляется набор операторов, которыми можно воспользоваться для доступа к отдельным битам в `int`-значении и работы с ними. К таким операторам относятся:

- **оператор НЕ (~).** Это унарный оператор, выполняющий поразрядное дополнение. Например, если взять 8-битное значение 11001100 (это десятичное число 204) и применить к нему оператор `~`, то в результате получится значение 00110011 (это десятичное число 51) — все единицы в исходном значении стали нулями, а нули превратились в единицы;



ПРИМЕЧАНИЕ Показанные здесь примеры имеют чисто иллюстративное назначение и верны только для 8 бит. В C# `int`-тип имеет размерность 32 бита, поэтому при попытке применения любого из этих примеров в приложении C# будет получаться 32-битный результат, который может отличаться от результата, показанного в данной подборке примеров. Например, при 32 битах число 204 имеет вид 0000000000000000000000000011001100, поэтому в C# `~204` будет иметь вид 11111111111111111111111100110011 (что фактически является представлением в C# десятичного числа -205).

- **оператор сдвига влево (<<).** Это бинарный оператор, выполняющий сдвиг влево. Выражение `204 << 2` возвращает значение 48. (В двоичном виде

десятичное число 204 отображается как 11001100, и его сдвиг влево на две позиции дает результат 00110000, или десятичное число 48.) Самые левые биты удаляются, а справа появляются нули. Существует также аналогичный оператор, осуществляющий сдвиг вправо (`>>`);

- **оператор ИЛИ (|).** Это бинарный оператор, выполняющий поразрядную операцию ИЛИ, возвращающую значение, содержащую единицу в каждом разряде, в котором в любом из operandов имеется единица. Например, выражение `204 | 24` дает значение 220 (204 — это 11001100, 24 — это 00011000, а 220 — это 11011100);
- **оператор И (&).** Этот оператор выполняет поразрядную операцию И. Оператор И похож на оператор ИЛИ, но он возвращает значение, содержащее единицу в каждом разряде, в котором у обоих operandов имеется единица. Поэтому выражение `204 & 24` дает результат 8 (204 — это 11001100, 24 — это 00011000, а 8 — это 00001000);
- **оператор исключающего ИЛИ (^).** Этот оператор выполняет поразрядную операцию исключающего ИЛИ, возвращая единицу в каждом разряде, в котором имеется единица в каком-либо из operandов, но не в обоих вместе. (Две единицы дают нуль, в этом и выражается «исключающая» часть оператора.) Следовательно, выражение `204 ^ 24` дает результат 212 (11001100 ^ 00011000 является 11010100).

Для определения значений отдельных битов в `int`-типе эти операторы могут использоваться вместе. Например, в следующем выражении используются операторы сдвига влево (`<<`) и поразрядного И (`&`), в результате чего определяется, какое значение имеет шестой разряд справа байтовой переменной по имени `bits` — нуль или единицу:

```
(bits & (1 << 5)) != 0
```



ПРИМЕЧАНИЕ Поразрядные операторы вычисляют позиции разрядов справа налево, а разряды отсчитываются, начиная с нуля. Соответственно, самым правым является нулевой разряд, а разряд в позиции с номером 5 — шестым справа.

Предположим, что переменная `bits` содержит десятичное число 42. В двоичном выражении оно имеет вид 00101010. Десятичное значение 1 соответствует двоичному значению 00000001, а выражение `1 << 5` имеет значение 00100000, где шестой разряд установлен в единицу. В двоичном виде выражение `bits & (1 << 5)` выглядит как 00101010 & 00100000, и значение этого выражения в том же двоичном виде выглядит как 00100000, то есть не является нулем. Если переменная `bits` содержит значение 65, или в двоичном виде 01000001, значение выражения выглядит как 01000001 & 00100000, что дает двоичный результат 00000000, или нуль.

Это довольно сложный пример, но по сравнению со следующим выражением, в котором для установки бита в позиции 6 в нуль используется оператор составного присваивания `&=`, он представляется простым:

```
bits &= ~(1 << 5)
```

Аналогично этому, если требуется установить бит в позиции 6 в единицу, можно воспользоваться поразрядным оператором ИЛИ (`|`). Следующее сложное выражение основано на применении составного оператора `|=`:

```
bits |= (1 << 5)
```

Применение таких примеров затруднено тем, что, несмотря на их полную состоятельность, в них неизвестно сложно разобраться. К тому же решение с их применением находится на одном из самых низких уровней программирования: на его основе невозможно создать абстракцию выполняемой задачи, что существенно усложняет сопровождение кода, выполняющего подобные операции.

Тот же пример с использованием индексаторов

Давайте немного отвлечемся от предыдущего низкоуровневого решения и вспомним, что представляет собой эта задача. Вам нужно воспользоваться `int`-типов не по прямому его назначению, а в качестве массива битов. Поэтому наилучшим способом решения этой задачи станет использование `int`-типа таким образом, будто он является массивом битов, или, иными словами, обеспечение возможности получения доступа к шестому биту справа в переменной `bits` путем использования следующего выражения (не забываем, что индексирование массивов начинается с нуля):

```
bits[5]
```

А для установки для четвертого бита справа значения `true` хотелось бы иметь возможность написать следующее:

```
bits[3] = true
```



ПРИМЕЧАНИЕ Для опытных разработчиков приложений на языке C булево значение `true` является синонимом двоичного значения 1, а булево значение `false` — двоичного значения 0. Соответственно, выражение `bits[3] = true` означает: «Установить в переменной `bits` для четвертого бита справа значение 1».

К сожалению, воспользоваться системой записи с квадратными скобками в отношении `int`-значения невозможно, она работает только в отношении массива или типа данных с поведением массива. Поэтому решение задачи заключается

в создании нового типа, работающего как массив, воспринимаемого как массив и используемого качестве массива булевых переменных, но реализуемого с использованием `int`-значения. Достичь желаемого результата можно с помощью индексатора. Давайте назовем этот новый тип `IntBits`. Он будет содержать `int`-значение (инициализированное в своем конструкторе), но суть замысла в том, что значение типа `IntBits` будет использоваться в качестве массива булевых переменных.



COBET Тип `IntBits` представляется весьма простым, поэтому имеет смысл создать его в виде структуры, а не в виде класса.

```
struct IntBits
{
    private int bits;

    public IntBits(int initialValue)
    {
        bits = initialValue;
    }

    // здесь будет написан индексатор
}
```

Для определения индексатора используется система записи, являющаяся гибридом записи свойства и массива. Индексатор вводится с помощью ключевого слова `this`, при этом указывается тип возвращаемого индексатором значения, а в квадратных скобках указывается тип значения, используемого в индексаторе в качестве индекса. В индексаторе для структуры `IntBits` в качестве типа его индекса используется целое число, а возвращает он булево значение:

```
struct IntBits
{
    ...
    public bool this [ int index ]
    {
        get
        {
            return (bits & (1 << index)) != 0;
        }
        set
        {
            if (value) // если value равно true, установка, а иначе - сброс бита
                bits |= (1 << index);
            else
                bits &= ~(1 << index);
        }
    }
}
```

Обратите внимание на следующие особенности.

- ❑ Индексатор не является методом: параметры в круглых скобках отсутствуют, а в квадратных скобках указывается индекс, который используется для указания того, к какому элементу происходит обращение.
- ❑ Индексаторы используют ключевое слово `this`. В классе или структуре может быть определено не более одного индексатора (хотя его можно переопределить и иметь несколько реализаций), и он всегда называется `this`.
- ❑ В индексаторах, как и в свойствах, содержатся методы доступа `get` и `set`. В данном примере методы доступа `get` и `set` содержат ранее рассмотренные сложные поразрядные выражения.
- ❑ Индекс, указанный в объявлении индексатора, заполняется индексным значением, указанным при вызове индексатора. Методы доступа `get` и `set` могут считать этот аргумент для определения элемента, к которому осуществляется доступ.



ПРИМЕЧАНИЕ Во избежание любых неожиданных исключений, выдаваемых в коде индексатора, в индексаторе нужно проверять значение индекса на его принадлежность к допустимому диапазону значений.

После объявления индексатора вы можете вместо `int`-переменной воспользоваться переменной типа `IntBits` и, как показано в следующем примере, применить к ней систему записи с использованием квадратных скобок:

```
int adapted = 126;           // Двоичный вид числа 126 – 01111110
IntBits bits = new IntBits(adapted);
bool peek = bits[6];         // извлечение булева значения по индексу 6;
                            // оно должно быть true (1)
bits[0] = true;             // установка бита по индексу 0 в true (1)
bits[3] = false;            // установка бита по индексу 3 в false (0)
                            // теперь в bits содержится значение 01110111,
                            // или 119 в десятичном исчислении
```

Вполне очевидно, что разобраться в этом синтаксисе будет гораздо проще. В нем довольно лаконично и непосредственно отражается суть задачи.

Основные сведения о методах доступа к индексаторам

При чтении с использованием индексатора компилятор автоматически преобразует ваш код, имеющий признаки массива, в вызов метода доступа `get` данного индексатора. Рассмотрим следующий пример:

```
bool peek = bits[6];
```

Эта инструкция преобразуется в вызов метода доступа `get` для индексатора `bits`, а для аргумента `index` устанавливается значение 6.

Аналогично этому при записи с использованием индексатора компилятор автоматически преобразует ваш код, имеющий признаки массива, в вызов метода доступа `set` данного индексатора, устанавливая для аргумента `index` значение, заключенное в квадратные скобки:

```
bits[3] = true;
```

Эта инструкция преобразуется в вызов метода доступа `set` для индексатора `bits`, а для аргумента `index` устанавливается значение 3. Как и в обычных свойствах, данные, записываемые в индексатор (в данном случае `true`), становятся доступными внутри метода доступа `set` при использовании ключевого слова `value`. Тип у переменной `value` такой же, как тип у самого индексатора (в данном случае `bool`).

Индексатор также можно применять в сложном контексте чтения/записи. В таком случае используются оба метода доступа, и `get`, и `set`. Посмотрите на следующую инструкцию, в которой используется оператор ИЛИ (^), чтобы инвертировать значение бита с индексом 6 в переменной `bits`:

```
bits[6] ^= true;
```

Это выражение будет автоматически преобразовано в следующий код:

```
bits[6] = bits[6] ^ true;
```

Работоспособность этого кода обеспечивается наличием обоих методов доступа, `get` и `set`.



ПРИМЕЧАНИЕ Можно объявить индексатор, содержащий только метод доступа `get` (индексатор только для чтения) или только метод доступа `set` (индексатор только для записи).

Сравнение индексаторов и массивов

При использовании индексатора намеренно применяется синтаксис, очень похожий на тот, что применяется в массивах. Но между индексаторами и массивами имеется ряд весьма важных различий.

- ❑ В индексаторах могут использоваться нечисловые индексы, например строки (как показано в следующем примере), а в массивах могут использоваться только целочисленные индексы:

```
public int this [ string name ] { ... } // Допустимо
```

- ❑ Индексаторы могут перегружаться (точно так же, как методы), а массивы не МОГУТ:

```
public Name this [ PhoneNumber number ] { ... }
public PhoneNumber this [ Name name ] { ... }
```

- ❑ Индексаторы не могут использоваться в качестве `ref`- или `out`-параметров, а массивы МОГУТ:

```
IntBits bits;           // bits содержит индексатор
Method(ref bits[1]); // ошибка в ходе компиляции
```

СВОЙСТВА, МАССИВЫ И ИНДЕКСАТОРЫ

Свойство может возвращать массив, но следует помнить, что массивы относятся к ссылочным типам, поэтому предоставление массива в качестве свойства позволяет случайно перезаписать большой массив данных. Посмотрите на следующую структуру, предоставляющую свойство по имени `Data`, содержащее массив:

```
struct Wrapper
{
    private int[] data;
    ...
    public int[] Data
    {
        get { return this.data; }
        set { this.data = value; }
    }
}
```

Теперь посмотрите на следующий код, использующий это свойство:

```
Wrapper wrap = new Wrapper();
...
int[] myData = wrap.Data;
myData[0]++;
myData[1]++;
```

Его внешний вид вроде бы ничем не настораживает. Но поскольку массивы относятся к ссылочным типам, переменная `myData` ссылается на тот же самый объект, что и закрытая переменная `data` в структуре `Wrapper`. Любые изменения, вносимые в элементы в `myData`, делаются в массиве `data`, а выражение `myData[0]++` производит точно такой же эффект, что и выражение `data[0]++`. Если это противоречит вашим замыслам, нужно в методах доступа `get` и `set` свойства `Data` воспользоваться методом `Clone`, чтобы возвращалась копия массива `data`, или сделать копию устанавливаемого значения, как показано в следующем примере кода. (Метод `Clone` для копирования массивов рассматривался в главе 8 «Основные сведения о значениях и ссылках».) Учтите, что метод `Clone` возвращает объект, который следует привести к типу целочисленного массива:

```
struct Wrapper
{
    private int[] data;
    ...
    public int[] Data
    {
        get { return this.data.Clone() as int[]; }
        set { this.data = value.Clone() as int[]; }
    }
}
```

Но с точки зрения использования памяти этот подход может оказаться слишком грубым и неэкономичным. А индексаторы позволяют решить эту задачу намного проще и естественнее — не нужно предоставлять в качестве свойства весь массив, вместо этого лучше через индексатор сделать доступными его отдельные элементы:

```
struct Wrapper
{
    private int[] data;
    ...
    public int this [int i]
    {
        get { return this.data[i]; }
        set { this.data[i] = value; }
    }
}
```

В следующем коде индексатор используется примерно так же, как и в показанном ранее свойстве:

```
Wrapper wrap = new Wrapper();
...
int[] myData = new int[2];
myData[0] = wrap[0];
myData[1] = wrap[1];
myData[0]++;
myData[1]++;
```

Теперь увеличение значений в массиве myData на единицу не влияет на исходный массив Wrapper-объекта. Если же действительно понадобится изменить данные во Wrapper-объекте, придется написать следующую инструкцию:

```
wrap[0]++;
```

Так будет намного понятнее и безопаснее!

Индексаторы в интерфейсах

Индексаторы можно объявлять в интерфейсе. Для этого указываются ключевое слово `get`, ключевое слово `set` или же оба этих слова, но вместо тела метода

доступа `get` или `set` ставится точка с запятой. Любые реализующие интерфейс класс или структура должны включать и реализацию методов доступа к индексатору, объявленному в интерфейсе:

```
interface IRawInt
{
    bool this [ int index ] { get; set; }
}
struct RawInt : IRawInt
{
    ...
    public bool this [ int index ]
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Если индексатор интерфейса реализуется в классе, объявление индексатора можно сделать виртуальным. Это позволит последующим производным классам перегружать методы доступа `get` и `set`:

```
class RawInt : IRawInt
{
    ...
    public virtual bool this [ int index ]
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Можно также выбрать реализацию индексатора с использованием синтаксиса явной реализации, рассмотренного в главе 13 «Создание интерфейсов и определение абстрактных классов». Явная реализация индексатора не может иметь модификатор доступа `public` и объявляться виртуальной (и, соответственно, не может быть перегружена):

```
struct RawInt : IRawInt
{
    ...
    bool IRawInt.this [ int index ]
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Использование индексаторов в приложении Windows

В следующем упражнении будет исследовано и завершено приложение простой телефонной книги. В классе PhoneBook будут созданы два индексатора: один, принимающий параметр типа `Name` (имя) и возвращающий значение типа `PhoneNumber` (номер телефона), и второй, действующий наоборот, то есть принимающий параметр `PhoneNumber` и возвращающий `Name`. (Структуры `Name` и `PhoneNumber` будут содержаться в упражнении в готовом виде.) Вам также нужно будет вызывать эти индексаторы из подходящих для этого мест в программе.

Ознакомьтесь с приложением

Откройте в среде Microsoft Visual Studio проект `Indexers`, который находится в папке `\Microsoft Press\VCBS\Chapter 16\Indexers` вашей папки документов.

С помощью этого графического приложения пользователь может искать номер телефона делового партнера, а также искать имя делового партнера по заданному номеру телефона.

Щелкните в меню Отладка на пункте Начать отладку. Произойдет сборка и запуск приложения и появится форма с пустыми текстовыми полями `Name` (Имя) и `Phone Number` (Номер телефона). Изначально в форме имеются две кнопки: одна для поиска телефона при заданном имени, другая для поиска имени при заданном номере телефона. Если раскрыть панель управления в нижней части формы, появится кнопка `Add` (Добавить), которая добавит к списку имен и номеров телефона, хранящемуся в приложении, еще одну пару «имя — номер телефона». Все кнопки, включая `Add` на панели управления, в данный момент ничего не делают. Приложение имеет вид, показанный на рис. 16.1.

Вам предстоит завершить приложение и заставить кнопки работать.

Вернитесь в среду Visual Studio 2015 и остановите отладку.

Выполните в окно редактора файл `Name.cs` проекта `Indexers`. Изучите структуру `Name`. Она предназначена для работы в качестве хранилища имен. Имя представляется конструктором в виде строки. Это имя может быть извлечено путем использования строкового свойства `Text`, предназначенного только для чтения. (Методы `Equals` и `GetHashCode` используются для сравнения имен при сквозном поиске в массиве из `Name`-значений, и пока их можно проигнорировать.)

Выполните в окно редактора файл `PhoneNumber.cs` и изучите структуру `PhoneNumber`. Она похожа на структуру `Name`.

Выполните в окно редактора файл `PhoneBook.cs` и изучите класс `PhoneBook`.

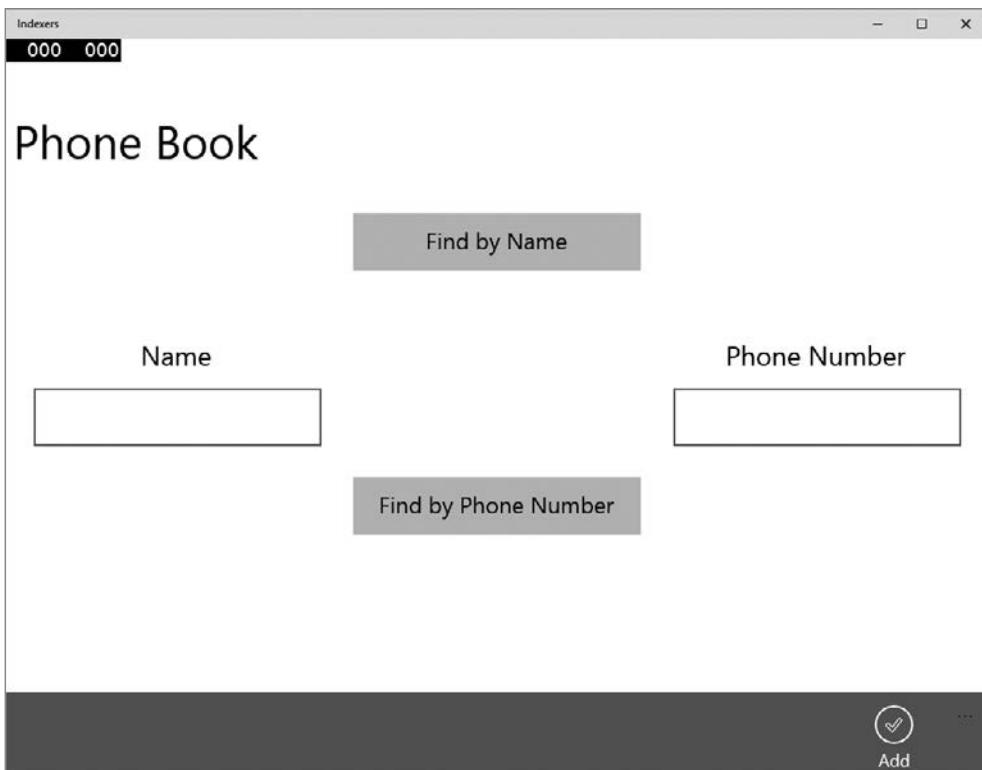


Рис. 16.1

В этом классе содержатся два закрытых массива: массив Name-значений по имени `names` и массив PhoneNumber-значений по имени `phoneNumbers`. Класс `PhoneBook` содержит также метод `Add`, добавляющий номер телефона и имя в телефонную книгу. Этот метод вызывается, когда пользователь щелкает на имеющейся в форме кнопке `Add`. Метод `enlargeIfFull` вызывается методом `Add` для проверки заполненности массивов, когда пользователь добавляет еще одну запись. Этот метод создает два новых, более крупных массива, копируя в них содержимое существующих массивов, после чего уничтожает старые массивы.

Метод `Add` намеренно упрощен и не проверяет, добавлялись ли уже в телефонную книгу данные имя или телефон.

Класс `PhoneBook` пока не предоставляет функциональных возможностей, с помощью которых пользователь смог бы найти имя или номер телефона. В следующем упражнении для реализации такой возможности вами будут добавлены два индексатора.

Создание индексаторов

Удалите из файла PhoneBook.cs комментарий `// TODO: write 1st indexer here`, заменив его открытым, предназначенный только для чтения индексатором для класса PhoneBook, выделенным в следующем примере кода жирным шрифтом. Индексатор будет возвращать Name-значение, получив в качестве своего индекса PhoneNumber-значение. Оставьте тело метода доступа `get` пустым.

Индексатор должен иметь следующий вид:

```
sealed class PhoneBook
{
    ...
    public Name this[PhoneNumber number]
    {
        get
        {
        }
    }
    ...
}
```

Реализуйте метод доступа `get`, выделенный в следующем примере кода жирным шрифтом.

Метод доступа предназначен для поиска имени, соответствующего указанному номеру телефона. Для этого вам нужно вызвать статический метод `IndexOf`, принадлежащий классу `Array`. Метод `IndexOf` осуществляет сквозной поиск в массиве, возвращая индекс первого же элемента массива, соответствующего указанному значению. Первым аргументом `IndexOf` является массив для выполнения сквозного поиска (`phoneNumbers`). Вторым аргументом `IndexOf` является искомый элемент. `IndexOf` возвращает целочисленный индекс элемента, если таковой будет найден, в противном случае возвращает `-1`. Если индексатор находит номер телефона, он должен возвращать соответствующее имя, а в противном случае — пустое Name-значение. (Учтите, что `Name` является структурой, следовательно, пассивный конструктор установит для его закрытого поля `name` значение `null`.)

```
sealed class PhoneBook
{
    ...
    public Name this [PhoneNumber number]
    {
        get
        {
            int i = Array.IndexOf(this.phoneNumbers, number);
            if (i != -1)
            {
                return this.names[i];
            }
            else
            {

```

```
        return new Name();
    }
}
...
}
```

Замените комментарий `// TODO: write 2nd indexer here` вторым открытым, предназначенным только для чтения индексатором для класса `PhoneBook`, возвращающим `PhoneNumber`-значение и принимающим один `Name`-параметр. Реализуйте этот индексатор точно так же, как и первый индексатор. (Тут снова следует учесть, что `PhoneNumber` является структурой, всегда имеющей пассивный конструктор.)

Второй индексатор должен иметь следующий вид:

```
sealed class PhoneBook
{
    ...
    public PhoneNumber this [Name name]
    {
        get
        {
            int i = Array.IndexOf(this.names, name);
            if (i != -1)
            {
                return this.phoneNumbers[i];
            }
            else
            {
                return new PhoneNumber();
            }
        }
    }
    ...
}
```

Обратите внимание на возможность сосуществования этих перегружаемых индексаторов, обусловленную тем, что их индексы имеют разные типы, а значит, и разные сигнатуры. Если бы структуры `Name` и `PhoneNumber` были заменены простыми строками (которые в них заключены), перегрузка имела бы одинаковую сигнатуру и класс не прошел бы компиляцию.

Щелкните в меню Сборка на пункте Собрать решение, исправьте все синтаксические ошибки, если таковые обнаружатся, а затем, если это необходимо, выполните повторную сборку решения.

Вызов индексаторов

Выполните в окне редактора файл `MainPage.xaml.cs` и найдите метод `findByNameClick`. Этот метод вызывается при щелчке на кнопке `Find By Name` (Найти по имени).

На данный момент код в нем отсутствует. Замените комментарий // TODO: кодом, выделенным в следующем примере жирным шрифтом. Этот код выполняет следующие задачи:

- ❑ считывает значение свойства **Text**, имеющегося в форме текстового поля **name**. В этой строке содержится имя делового партнера, набранное пользователем;
- ❑ если строка не является пустым значением, с помощью индексатора ищет номер телефона, соответствующий этому имени в телефонной книге **PhoneBook**. (Заметьте, что в классе **MainPage** содержится закрытое **PhoneBook**-поле по имени **phoneBook**.) Код создает из строки **Name**-объект и передает его в качестве параметра индексатору **PhoneBook**;
- ❑ если возвращенное индексатором свойство **Text**, принадлежащее структуре **PhoneNumber**, не содержит значение **null** или не является пустым, записывает значение этого свойства в имеющееся в форме текстовое поле **phoneNumber**. В противном случае в это поле выводится текст «**Not Found**», свидетельствующий о том, что поиск не удался.

В окончательном виде метод **findByNameClick** должен выглядеть следующим образом:

```
private void findByNameClick(object sender, RoutedEventArgs e)
{
    string text = name.Text;
    if (!String.IsNullOrEmpty(text))
    {
        Name personsName = new Name(text);
        PhoneNumber personsPhoneNumber = this.phoneBook[personsName];
        phoneNumber.Text = String.IsNullOrEmpty(personsPhoneNumber.Text) ?
            "Not Found" : personsPhoneNumber.Text;
    }
}
```

Кроме инструкции, которая обращается к индексатору, у этого кода есть еще две интересные особенности.

- ❑ Статический **String**-метод по имени **IsNullOrEmpty** используется для определения того, не является ли строка пустой и не содержит ли она **null**-значение. Это метод лучше всего подходит для тестирования строки на наличие в ней значения. Он возвращает **true**, если строка содержит значение **null** или является пустой, а в противном случае возвращает значение **false**.
- ❑ Инструкция, заполняющая свойство **Text** текстового поля **phoneNumber**, использует оператор **? :**, который действует как встроенная в выражение инструкция **if...else**. Этот оператор работает с тремя operandами: булевым выражением, выражением для вычисления и возвращения в случае вычисления булева выражения в **true** и еще одним выражением для вычисления и возвращения в случае вычисления булева выражения в **false**. Если в предыдущем

коде выражение `String.IsNullOrEmpty(personsPhoneNumber.Text)` вычисляется в `true`, что означает, что совпадений в записях телефонной книги найдено не было, то в форме выводится текст «*Not Found*»; в противном случае выводится значение, хранящееся в свойстве `Text` переменной `personsPhoneNumber`.

В общем виде оператор `? :` имеет следующий синтаксис:

Результат = <булево выражение> ? <Вычисляется, если true> : <Вычисляется, если false>

Найдите в файле `MainPage.xaml.cs` метод `findByPhoneNumberClick`, который находится ниже метода `findByNameClick`. Метод `findByPhoneNumberClick` вызывается при щелчке на кнопке `Find By Phone Number` (Найти по номеру телефона). Пока в этом методе нет ничего, кроме комментария `// TODO:`. Вам нужно реализовать этот метод, вставив в него код. (Полный код метода показан в следующем примере жирным шрифтом.)

Прочтайте значение свойства `Text` из имеющегося в форме текстового поля `phoneNumber`. Это строка, содержащая номер телефона, набранный пользователем.

Если в строке есть содержимое, воспользуйтесь индексатором для поиска имени, соответствующего этому номеру телефона в `PhoneBook`.

Запишите в принадлежащее форме поле `name` значение возвращенного индексатором свойства `Text`, принадлежащего структуре `Name`.

В окончательном варианте метод должен приобрести следующий вид:

```
private void findByPhoneNumberClick(object sender, RoutedEventArgs e)
{
    string text = phoneNumber.Text;
    if (!String.IsNullOrEmpty(text))
    {
        PhoneNumber personsPhoneNumber = new PhoneNumber(text);
        Name personsName = this.phoneBook[personsPhoneNumber];
        name.Text = String.IsNullOrEmpty(personsName.Text) ?
            "Not Found" : personsName.Text;
    }
}
```

Щелкните в меню Сборка на пункте Собрать решение и исправьте любые выявленные ошибки.

Тестирование приложения

Щелкните в меню Отладка на пункте Начать отладку. Наберите в соответствующих полях свое имя и номер телефона, затем раскройте панель управления и щелкните на кнопке Add (Добавить) (панель управления раскрывается по щелчку на символе многоточия). При щелчке на кнопке Add (Добавить) метод

Add сохраняет информацию в телефонной книге и очищает текстовые поля, подготавливая их к поиску.

Наберите несколько разных имен и номеров, чтобы в телефонной книге появилась подборка записей. Учтите, что приложение не проверяет вводимые имена и номера телефонов и одно и то же имя и один и тот же номер телефона можно ввести более одного раза. Чтобы не мешать демонстрации работы программы и исключить путаницу, постараитесь задавать разные имена и номера телефонов.

Наберите одно из ранее введенных имен в поле Name и щелкните на кнопке Find By Name (Найти по имени). Из телефонной книги будет извлечен и выведен в текстовое поле Phone Number номер телефона, ранее присвоенный вами этому деловому партнеру.

Наберите в поле Phone Number номер телефона другого делового партнера, а затем щелкните на кнопке Find By Phone Number (Найти по номеру телефона). Из телефонной книги будет извлечено и показано в поле Name имя делового партнера.

Наберите в поле Name имя, которое не вводилось в телефонную книгу, а затем щелкните на кнопке с надписью Find By Name (Найти по имени). На этот раз в поле с надписью Phone Number будет выведено сообщение о том, что номер не найден, — «Not Found».

Закройте форму и вернитесь в среду Visual Studio 2015.

Выводы

В этой главе вы узнали, как использовать индексаторы, предоставляющие к данным класса такой же доступ, как и к элементам массива. Вы научились создавать индексаторы, которые могут принимать индекс и возвращать соответствующее значение с использованием логики, определяемой в методе доступа `get`, и увидели, как используется с индексом метод доступа `set`, предназначенный для того, чтобы заполнить значение в индексаторе.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 17 «Введение в обобщения».

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню Файл щелкните на пункте Выход. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке Да и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Создать индексатор для класса или структуры	<p>Объявите тип индексатора, затем наберите ключевое слово <code>this</code>, а после него укажите в квадратных скобках аргументы индексатора. В теле индексатора может содержаться метод доступа <code>get</code> и/или <code>set</code>, например:</p> <pre>struct RawInt { ... public bool this [int index] { get { ... } set { ... } } ... }</pre>
Определить индексатор в интерфейсе	<p>Определите индексатор с ключевым словом <code>get</code> и/или <code>set</code>, например:</p> <pre>interface IRawInt { bool this [int index] { get; set; } }</pre>
Реализовать интерфейсный индексатор в классе или структуре	<p>Определите в классе или структуре, реализующей интерфейс, индексатор и реализуйте методы доступа к нему, например:</p> <pre>struct RawInt : IRawInt { ... public bool this [int index] { get { ... } set { ... } } ... }</pre>
Реализовать индексатор, определенный в интерфейсе, путем явной реализации интерфейса в классе или структуре	<p>Укажите индексатор в классе или в структуре, реализующей интерфейс, но не указывайте степень его доступности, например:</p> <pre>struct RawInt : IRawInt { ... bool IRawInt.this [int index] { get { ... } set { ... } } ... }</pre>

17

Введение в обобщения

Прочитав эту главу, вы научитесь:

- объяснять назначение обобщений;
- определять с использованием обобщений класс, не зависящий от используемых типов;
- создавать экземпляры класса-обобщения на основе типов, указанных в качестве параметров типов;
- реализовывать обобщенный интерфейс;
- определять обобщенный метод, реализующий алгоритм, независимый от типа данных, с которыми он работает.

В главе 8 «Основные сведения о значениях и ссылках» было показано, как переменная типа `object` используется для ссылки на экземпляр любого класса. Такая переменная может применяться для хранения значения любого типа, и когда в метод необходимо передать значения любых типов, параметры могут определяться с использованием переменных типа `object`. Метод также может возвращать значения любого типа путем указания `object` в качестве возвращаемого типа. При всей своей гибкости такой подход заставляет программиста держать в памяти, какого рода данные фактически используются. Если программист ошибается, это может привести к ошибкам в ходе выполнения программы. В этой главе вы узнаете об обобщениях, разработанных, чтобы помочь вам не допускать подобных ошибок.

Проблемы, связанные с типом `object`

Чтобы понять, что такое обобщения, стоит присмотреться к задаче, для решения которой они были разработаны. Предположим, нужно смоделировать структуру наподобие очереди, работающей по принципу «первым пришел — первым ушел». Для этого можно создать следующий класс:

```
class Queue
{
    private const int DEFAULTQUEUESIZE = 100;
    private int[] data;
    private int head = 0, tail = 0;
    private int numElements = 0;

    public Queue()
    {
        this.data = new int[DEFAULTQUEUESIZE];
    }

    public Queue(int size)
    {
        if (size > 0)
        {
            this.data = new int[size];
        }
        else
        {
            throw new ArgumentOutOfRangeException("size", "Must be greater than zero");
        }
    }

    public void Enqueue(int item)
    {
        if (this.numElements == this.data.Length)
        {
            throw new Exception("Queue full");
        }

        this.data[this.head] = item;
        this.head++;
        this.head %= this.data.Length;
        this.numElements++;
    }

    public int Dequeue()
    {
        if (this.numElements == 0)
        {
            throw new Exception("Queue empty");
        }

        int queueItem = this.data[this.tail];
        this.tail++;
        this.tail %= this.data.Length;
        this.numElements--;
        return queueItem;
    }
}
```

Для создания кольцевого буфера хранения данных в этом классе используется массив. Размер этого массива указан конструктором. Чтобы добавить элемент

в очередь, приложение использует метод `Enqueue`, а чтобы удалить его из очереди — метод `Dequeue`. Закрытые поля `head` и `tail` отслеживают, куда нужно вставить элемент в массив и из какого места извлечь элемент из массива. Поле `numElements` показывает, сколько элементов находится в массиве. Методы `Enqueue` и `Dequeue` используют эти поля для определения того, где сохранять или откуда извлекать элемент, и выполняют ряд простейших проверок на наличие ошибок. Как показано в следующем примере кода, приложение может создать `Queue`-объект и вызывать эти методы. Обратите внимание на то, что элементы извлекаются из очереди в том же порядке, в котором в нее попадают:

```
Queue queue = new Queue(); // Создание нового Queue-объекта
queue.Enqueue(100);
queue.Enqueue(-25);
queue.Enqueue(33);
Console.WriteLine($"{queue.Dequeue()}"); // Выводит на экран 100
Console.WriteLine($"{queue.Dequeue()}"); // Выводит на экран -25
Console.WriteLine($"{queue.Dequeue()}"); // Выводит на экран 33
```

Класс `Queue` хорошо работает для очередей из `int`-элементов, а как быть, если нужно создавать очереди из строк, или чисел с плавающей точкой, или даже более сложных типов, таких как `Circle` (см. главу 7 «Создание классов и объектов и управление ими»), `Horse` или `Whale` (см. главу 12 «Работа с наследованием»)? Проблема в том, что способ реализации класса `Queue` ограничивает тип элементов целочисленными значениями, и при попытке вставить элемент `Horse`-типа будет выдана ошибка компиляции:

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Ошибка в ходе компиляции: конвертация Horse в int
                        // невозможна
```

Один из способов обойти это ограничение заключается в указании того, что массив в классе `Queue` содержит элементы типа `object`, а также в изменении конструктора и методов `Enqueue` и `Dequeue` таким образом, чтобы они принимали параметр типа `object` и возвращали значение такого же типа:

```
class Queue
{
    ...
    private object[] data;
    ...
    public Queue()
    {
        this.data = new object[DEFAULTQUEUESIZE];
    }

    public Queue(int size)
    {
```

```
    ...
    this.data = new object[size];
    ...
}

public void Enqueue(object item)
{
    ...
}

public object Dequeue()
{
    ...
    object queueItem = this.data[this.tail];
    ...
    return queueItem;
}
}
```

Вспомним, что для ссылки на значение или переменную любого типа можно воспользоваться типом `object`. Все ссылочные типы автоматически наследуются (как непосредственно, так и косвенно) из класса `System.Object` среды Microsoft .NET Framework (в C# `object` является псевдонимом `System.Object`). Теперь, поскольку методы `Enqueue` и `Dequeue` распоряжаются объектами, в очередях можно работать с классами `Circles`, `Horses`, `Whales` или любыми другими, встречавшимися ранее в этой книге. Но важно учесть, что возвращаемые методом `Dequeue` значения следует приводить к соответствующему типу, поскольку компилятор не будет выполнять автоматическое преобразование из типа `object`:

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Теперь это допустимо, поскольку Horse – это объект

...
Horse dequeuedHorse = (Horse)queue.Dequeue(); // Следует привести тип object к типу
                                                // Horse
```

Если не выполнить приведение типа возвращаемого значения, будет получена ошибка компиляции, сообщающая о невозможности явного приведения типа `object` к типу `Horse`. Это требование явного приведения типа во многом снижает присущую типу `object` гибкость. Кроме того, совсем не трудно допустить ошибку, написав следующий код:

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse);

...
Circle myCircle = (Circle)queue.Dequeue(); // Ошибка в ходе выполнения
```

Несмотря на то что этот код пройдет компиляцию, он не сможет работать и в ходе выполнения выдаст исключение неверного приведения типов `System`.

`InvalidCastException`. Допущенная ошибка связана с попыткой при извлечении из очереди ссылки на `Horse` сохранить ее в `Circle`-переменной, а эти два типа не совместимы друг с другом. Эта ошибка ничем не проявится до тех пор, пока код не будет запущен на выполнение, поскольку у компилятора недостаточно информации для выполнения проверки в ходе компиляции. Каков реальный тип объекта, извлекаемого из очереди, выясняется только при выполнении кода.

Еще одним недостатком использования подхода с применением значений типа `object` для создания обобщенных классов и методов является вероятность дополнительного потребления памяти и процессорного времени, если в ходе выполнения потребуется преобразовывать объект в значение типа и обратно. Рассмотрим следующий фрагмент кода, в котором ведется работа с очередью из целочисленных значений:

```
Queue queue = new Queue();
int myInt = 99;
queue.Enqueue(myInt); // Запаковка int в object
...
myInt = (int)queue.Dequeue(); // Распаковка object в int
```

Тип данных `Queue` предполагает, что содержащиеся в нем элементы имеют тип `object`, а он относится к ссылочным типам. Помещение в очередь элемента, относящегося к типам значений, например `int`-элемента, требует его упаковки с целью преобразования в ссылочный тип. Аналогично этому извлечение из очереди элемента с получением `int`-значения требует распаковки элемента с целью его преобразования в элемент, имеющий тип значения (дополнительные сведения можно получить в разделах «Упаковка» и «Распаковка» главы 8). Хотя упаковка и распаковка происходят без каких-либо дополнительных действий, они вызывают дополнительные издержки производительности, поскольку используют динамическое выделение памяти. Для отдельно взятого элемента эти издержки невелики, но когда программа создает очереди из большого количества элементов, имеющих тип значения, они существенно возрастают.

Решение, использующее обобщения

Чтобы избавиться от необходимости приведения типов, в C# предоставляются обобщения, сокращающие объем требуемых упаковок и упрощающие создание обобщенных классов и методов, которые принимают параметры типа, указывающие типы объектов, с которыми они работают. В C# указанием на то, что класс является обобщением, служит параметр типа, заключенный в угловые скобки:

```
class Queue<T>
{
    ...
}
```

Параметр типа *T* в этом примере работает как заполнитель для реального типа, применяемого в ходе компиляции. При написании кода для получения экземпляра класса-обобщения *Queue* вы предоставляетете тип, который нужно подставить вместо *T* (*Circle*, *Horse*, *int* и т. д.). При определении в классе полей и методов для указания их типов используется точно такой же заполнитель:

```
class Queue<T>
{
    ...
    private T[] data; // Массив из элементов типа 'T', где 'T' – параметр типа
    ...
    public Queue()
    {
        this.data = new T[DEFAULTQUEUESIZE]; // Использование 'T' в качестве типа
                                            // данных
    }
    public Queue(int size)
    {
        ...
        this.data = new T[size];
        ...
    }
    public void Enqueue(T item) // Использование 'T' в качестве типа параметра
                                // метода
    {
        ...
    }
    public T Dequeue() // Использование 'T' в качестве типа возвращаемого значения
    {
        ...
        T queueItem = this.data[this.tail]; // Данные в массиве относятся
                                            // к типу 'T'
        ...
        return queueItem;
    }
}
```

В качестве параметра типа *T* может использоваться любой допустимый в C# идентификатор, хотя чаще всего это одиночный символ «*T*». Вместо него ставится тип, указанный при создании *Queue*-объекта. В следующем примере создается *Queue*-объект из *int*-значений и *Queue*-объект из *Horse*-значений:

```
Queue<int> intQueue = new Queue<int>();
Queue<Horse> horseQueue = new Queue<Horse>();
```

Кроме этого, у компилятора теперь достаточно информации для строгой проверки типов при сборке приложения. Теперь больше не нужно приводить типы данных при вызове метода *Dequeue*, и компилятор может отлавливать ошибки несоответствия типов на ранней стадии:

```

intQueue.Enqueue(99);
int myInt = intQueue.Dequeue();           // Приведение типов не требуется
Horse myHorse = intQueue.Dequeue();        // Ошибка компилятора: выполнить неявное
                                            // преобразование типа 'int' в тип 'Horse'
                                            // невозможно

```

Следует иметь в виду, что такая подстановка конкретного типа вместо `T` не является простым механизмом текстового замещения. Компилятор выполняет полноценную семантическую подстановку, поэтому для `T` можно указать любой допустимый тип. Рассмотрим еще несколько примеров:

```

struct Person
{
    ...
}
...
Queue<int> intQueue = new Queue<int>();
Queue<Person> personQueue = new Queue<Person>();

```

В первом примере создается очередь из целочисленных значений, а во втором — из `Person`-значений. Для каждой очереди компилятор создает также версии методов `.Enqueue` и `.Dequeue`. Для очереди `intQueue` эти методы имеют следующий вид:

```

public void Enqueue(int item);
public int Dequeue();

```

А для очереди `personQueue` они имеют следующий вид:

```

public void Enqueue(Person item);
public Person Dequeue();

```

Сравните эти определения с теми упомянутыми в предыдущем разделе версиями класса `Queue`, которые основаны на применении типа `object`. В методах, происходящих из класса-обобщения, параметр `item` передается методу `.Enqueue` в виде значения, относящегося к типу значений, не требующих упаковки. Аналогично этому значение, возвращенное методом `Dequeue`, относится к типу значений, не нуждающихся в распаковке. Аналогичный набор методов создается и для двух других очередей.



ПРИМЕЧАНИЕ Пространство имен `System.Collections.Generic` в библиотеке классов .NET Framework предоставляет реализацию класса `Queue`, работающего почти так же, как и только что рассмотренный класс. Это пространство имен включает и несколько других классов коллекций, которые более подробно описываются в главе 18 «Использование коллекций».

Параметр типа не обязательно должен быть простым классом или принадлежать к типу значений. Можно, к примеру, создать очередь из очередей целочисленных значений (если, конечно, она когда-нибудь пригодится):

```
Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();
```

У класса-обобщения может быть несколько параметров типа. Например, класс-обобщение `Dictionary`, определение которого находится в пространстве имен `System.Collections.Generic` библиотеки классов .NET Framework, предполагает использование двух параметров типа: одного для ключей, а другого для значений (более подробно этот класс рассматривается в главе 18).



ПРИМЕЧАНИЕ Можно также, используя точно такой же синтаксис с параметрами типа, как и для обобщенных классов, объявлять обобщенные структуры и интерфейсы.

Сравнение классов-обобщений и обобщенных классов

Следует различать класс-обобщение, использующий параметры типа, и обобщенный класс, разработанный с целью получения параметров, которые могут приводиться к различным типам. Например, показанная ранее версия класса `Queue`, основанная на применении значений типа `object`, является обобщенным классом. Это обособленная реализация класса, и его методы получают параметры типа `object`, а также возвращают значения типа `object`. Этот класс можно использовать с целочисленными, строковыми и многими другими типами, но всякий раз используется экземпляр одного и того же класса и применяемые данные приходится приводить к `object`-типу и из `object`-типа.

Сравните это с классом `Queue<T>`. При каждом использовании этого класса с параметром типа (например, `Queue<int>` или `Queue<Horse>`) вы заставляете компилятор создавать абсолютно новый класс с функциональными возможностями, определяемыми классом-обобщением. Это означает, что `Queue<int>` по типу совершенно отличается от `Queue<Horse>`, одинаковым у них является только поведение. Класс-обобщение можно представить себе как определение шаблона, который затем по мере необходимости используется компилятором для создания новых классов с конкретным указанием типов. Версии класса-обобщения с конкретным указанием типов (`Queue<int>`, `Queue<Horse>` и т. д.) называются классами со сконструированными типами, и их можно рассматривать как совершенно разные типы (даже те, у которых один и тот же набор методов и свойств).

Обобщения и ограничения

Временами нужно будет, чтобы параметр типа, используемый классом-обобщением, идентифицировал тип, предоставляющий конкретные методы. Например, если определяется класс `PrintableCollection` (коллекция, пригодная для вывода на печать), могут понадобиться гарантии того, что все объекты,

храняющиеся в классе, имели метод `Print`. Это условие можно задать, используя *ограничение*.

Используя ограничение, можно ограничить параметры типа класса-обобщения теми типами, которые реализуют конкретный набор интерфейсов, и тем самым предоставлять методы, определенные этими интерфейсами. Например, если в интерфейсе `IPrintable` определяется метод `Print`, вы можете создать следующий класс `PrintableCollection`:

```
public class PrintableCollection<T> where T : IPrintable
```

При создании этого класса с параметром типа компилятор выполняет проверку, чтобы убедиться, что тип, используемый для `T`, действительно реализует интерфейс `IPrintable`. Если этот интерфейс в нем не реализуется, компиляция прекращает работу с выдачей ошибки.

Создание класса-обобщения

В пространстве имен `System.Collections.Generic` библиотеки классов .NET Framework содержится целый ряд готовых к использованию классов-обобщений. Можно также определять собственные классы-обобщения, чем вы и будете заниматься в данном разделе. Но прежде чем приступить к их созданию, давайте рассмотрим некоторые теоретические положения.

Теория двоичных деревьев

В следующих упражнениях будет определяться и использоваться класс, представляющий собой двоичное дерево, которое является полезной структурой данных, используемой для выполнения разнообразных операций, включая очень быструю сортировку и сквозной поиск данных. О характеристиках двоичных деревьев написаны целые тома, но задача подробного изучения этой темы в данной книге не ставилась. Поэтому будут рассмотрены только те факты, которые имеют отношение к нашей теме. При желании расширить свои познания в данной области обратитесь к книге Дональда Кнута (Donald E. Knuth) «Искусство программирования», том 3 «Сортировка и поиск» (вышла в издательстве Addison-Wesley Professional в 1998 году, переведена на русский язык издательством «Вильямс»). Несмотря на почтенный возраст, она считается фундаментальной работой по алгоритмам сортировки и поиска.

Двоичное дерево является рекурсивной (автореферентной) структурой данных, которая может быть пустой или содержать три элемента: единицу информации

(datum), которую обычно называют узлом, и два поддерева, которые также являются двоичными деревьями. Два под дерева традиционно называются левым и правым, поскольку обычно их изображают соответственно левее и правее узла. Каждое левое или правое поддерево является либо пустым, либо содержащим узел и другие поддеревья. Теоретически структура может быть бесконечной. Небольшое двоичное дерево показано на рис. 17.1.

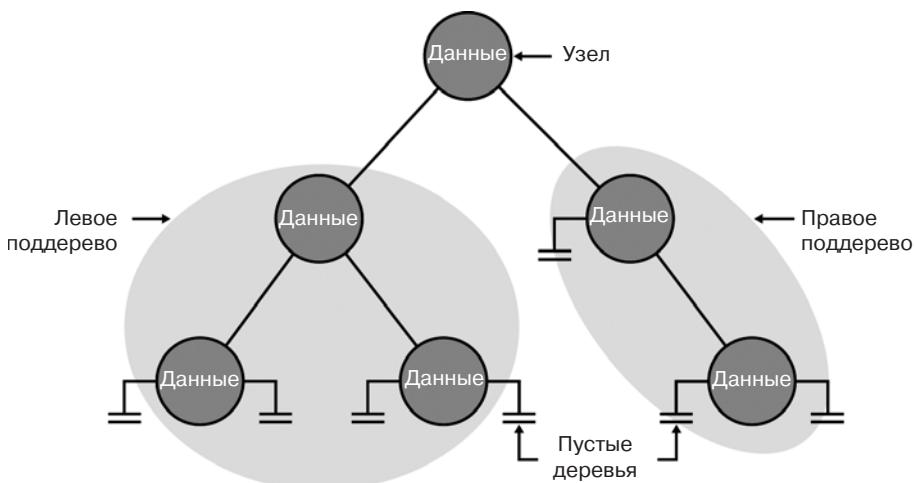


Рис. 17.1

Реальная эффективность двоичных деревьев проявляется при их использовании для сортировки данных. При наличии изначально неупорядоченной последовательности объектов одного типа можно выстроить упорядоченное двоичное дерево, а затем перемещаться по нему, заходя в каждый узел упорядоченной последовательности. Далее показан алгоритм вставки элемента «I» в упорядоченное двоичное дерево «B»:

Если дерево "B" пустое,
То

Создать новое дерево "B" с новым элементом "I" в качестве узла
и пустыми левым и правым поддеревьями

Иначе

Исследовать значения текущего узла "N" данного дерева "B"

Если значение "N" больше, чем значение нового элемента "I",
То

Если левое поддерево "B" пустое,
То

Создать новое левое поддерево дерева "B" с элементом "I" в качестве
узла и пустыми левым и правым поддеревьями

Иначе

Вставить "I" в левое поддерево дерева "B"

```

Завершить условие Если
Иначе
    Если правое поддерево дерева "B" пустое
        То
            Создать новое правое поддерево дерева "B" с элементом "I" в качестве
            узла и пустыми левым и правым поддеревьями
        Иначе
            Вставить "I" в правое поддерево дерева "B"
            Завершить условие Если
        Завершить условие Если
    Завершить условие Если

```

Обратите внимание на рекурсивность этого алгоритма, вызывающего самого себя для вставки элемента в левое или правое поддерево в зависимости от результата сравнения значения элемента со значением текущего узла в дереве.



ПРИМЕЧАНИЕ Определение выражения «больше чем» зависит от типа данных в элементе и в узле. Для числовых данных «больше чем» может быть простым арифметическим сравнением, а для текстовых данных оно может быть сравнением строк, но другим типам данных нужно давать их собственные средства сравнения значений. Более подробно данная тема будет изучена при реализации вами двоичного дерева в следующем разделе «Создание класса двоичного дерева с использованием обобщений».

Если изначально имеются пустое двоичное дерево и неупорядоченная последовательность объектов, можно устроить сквозной обход этой последовательности, вставляя каждый объект в двоичное дерево с использованием данного алгоритма и получив в результате упорядоченное дерево. Этапы построения дерева из набора, состоящего из пяти целых чисел, показаны на рис. 17.2.

После построения упорядоченного двоичного дерева его содержимое можно показать в виде последовательности путем поочередного посещения каждого узла и вывода на экран найденного значения. Алгоритм выполнения этой задачи также имеет рекурсивную природу:

```

Если левое поддерево не пустое
    То
        Вывести содержимое левого поддерева
    Завершить условие Если
    Вывести значение узла
    Если правое поддерево не пустое
        То
            Вывести содержимое правого поддерева
    Завершить условие Если

```

Этапы выведения содержимого дерева на экран показаны на рис. 17.3. Обратите внимание на то, что теперь целые числа выводятся на экран в порядке их возрастания.

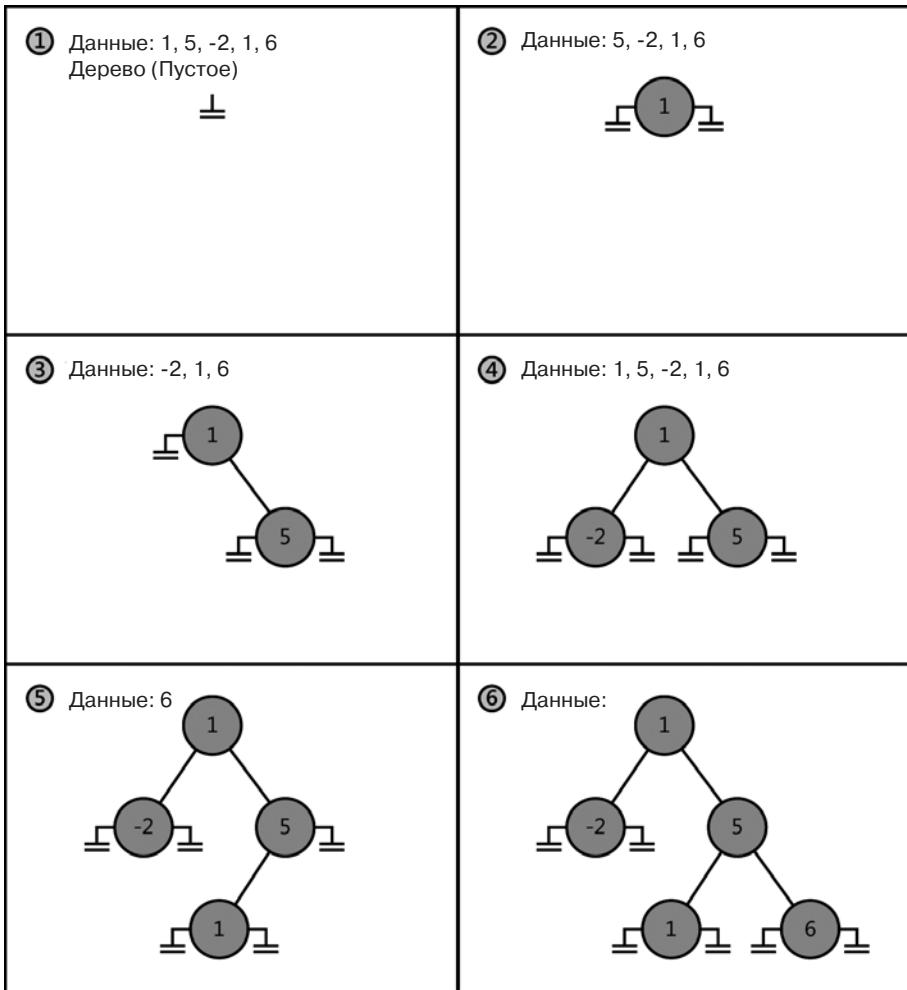


Рис. 17.2

Создание класса двоичного дерева с использованием обобщений

В следующем упражнении обобщения будут использованы для определения класса двоичного дерева, способного содержать почти любой тип данных. Единственное ограничение заключается в том, что тип данных должен предоставлять средства сравнения значений различных экземпляров.

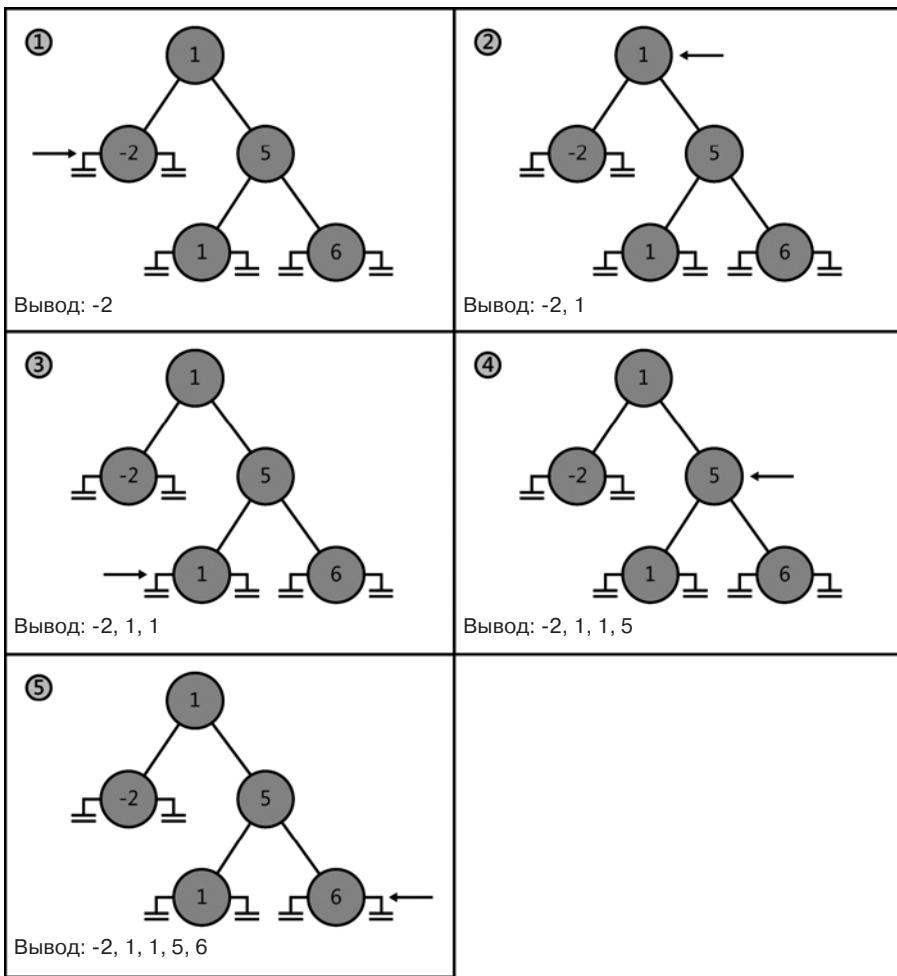


Рис. 17.3

Класс двоичного дерева может пригодиться в множестве различных приложений. Поэтому вы будете создавать его не в виде самостоятельного приложения, а в качестве библиотеки классов. Затем этот класс можно будет использовать где-нибудь в другом месте, при этом не понадобятся копирование исходного кода и его повторная компиляция. Библиотека классов представляет собой набор откомпилированных классов (и других типов, таких как структуры и делегаты), сохраненный в сборке. Сборка является файлом, имя которого обычно имеет расширение .dll. Элементы библиотеки классов могут использоваться другими проектами и приложениями посредством добавления ссылки на их сборку с последующим перенесением их пространств имен в область видимости путем

использования директивы `using`. Именно это вы и сделаете при тестировании класса двоичного дерева.

ИНТЕРФЕЙСЫ `SYSTEM.ICONPARABLE` И `SYSTEM.ICONPARABLE<T>`

Алгоритм вставки узла в двоичное дерево требует от вас сравнения значения вставляемого узла со значениями уже имеющихся в дереве узлов. Если используется числовой тип, например `int`, можно воспользоваться операторами `<`, `>` и `==`. Но как сравнивать объекты, если используется какой-нибудь другой тип, например `Mammal` или `Circle`, рассмотренные в предыдущих главах?

Если нужно создать класс, требующий от вас возможности сравнения значений в соответствии с неким естественным (а может, и неестественным) упорядочением, следует реализовать интерфейс `IComparable`. Этот интерфейс содержит метод `CompareTo`, принимающий один параметр, указывающий на объект, сравниваемый с текущим экземпляром, и возвращающий целое число, показывающее результат сравнения.

Значение	Означает
Меньше 0	Текущий экземпляр меньше значения параметра
0	Текущий экземпляр равен значению параметра
Больше 0	Текущий экземпляр больше значения параметра

Рассмотрим в качестве примера класс `Circle`, описание которого приводилось в главе 7. Давайте исследуем его еще раз:

```
class Circle
{
    public Circle(int initialRadius)
    {
        radius = initialRadius;
    }

    public double Area()
    {
        return Math.PI * radius * radius;
    }
    private double radius;
}
```

Сделать класс пригодным для сравнения можно, реализовав интерфейс `System.IComparable` и предоставив метод `CompareTo`. В данном примере метод `CompareTo` сравнивает `Circle`-объекты на основе их площадей. Круг с большей площадью считается больше круга с меньшей площадью:

```

class Circle : System.IComparable
{
    ...
    public int CompareTo(object obj)
    {
        Circle circObj = (Circle)obj; // Приведение параметра к его
                                       // настоящему типу
        if (this.Area() == circObj.Area())
            return 0;
        if (this.Area() > circObj.Area())
            return 1;

        return -1;
    }
}

```

Если изучить интерфейс System.IComparable, можно заметить, что его параметр определен как object. Но этот подход не дает полной независимости от применяемых типов. Чтобы понять, почему так, представьте, что получится, если попытаться передать методу CompareTo что-нибудь не относящееся к типу Circle. Интерфейс System.IComparable требует для доступа к методу Area использования приведения типа. Если параметр не относится к типу Circle и является объектом какого-либо другого типа, приведение даст сбой. Но в пространстве имен System определяется также интерфейс-обобщение IComparable<T>, в котором содержится следующий метод:

```
int CompareTo(T other);
```

Обратите внимание на то, что этот метод принимает не значение типа object, а параметр типа (T) и поэтому он намного безопаснее той версии интерфейса, которая не использует обобщение. Следующий код показывает, как этот интерфейс можно реализовать в классе Circle:

```

class Circle : System.IComparable<Circle>
{
    ...
    public int CompareTo(Circle other)
    {
        if (this.Area() == other.Area())
            return 0;
        if (this.Area() > other.Area())
            return 1;

        return -1;
    }
}

```

Параметр для метода CompareTo должен соответствовать типу, указанному в интерфейсе IComparable<Circle>. В целом предпочтительнее реализовывать интерфейс System.IComparable<T>, а не интерфейс System.IComparable. Можно также реализовать оба интерфейса, что и делают большинство типов в среде .NET Framework.

Создание класса Tree<TItem>

Укажите в меню Файл среды Microsoft Visual Studio 2015 на пункт Создать и щелкните на пункте Проект. В левой панели Шаблоны появившегося на экране диалогового окна Создание проекта щелкните на пункте Visual C#. Выберите в средней панели шаблон Библиотека классов. Наберите в поле Имя строку BinaryTree. Укажите в поле Расположение папку \Microsoft Press\VCSBS\Chapter 17 вашей папки документов, а затем щелкните на кнопке OK.



ПРИМЕЧАНИЕ Используя шаблон Class Library, вы можете создавать сборки, которые затем можно будет использовать в нескольких приложениях. Чтобы использовать класс в библиотеке классов, нужно сначала скопировать сборку, содержащую скомпилированный код для библиотеки классов, на свой компьютер (если вы не создали ее самостоятельно), а затем добавить ссылку на эту сборку.

Щелкните в обозревателе решений правой кнопкой мыши на файле Class1.cs, щелкните на пункте Переименовать, а затем измените имя файла на Tree.cs. При появлении предложения позвольте среде Visual Studio изменить имя класса, а также имя файла.

Измените в окне редактора определение класса Tree на Tree<TItem>, как показано в следующем коде жирным шрифтом:

```
public class Tree<TItem>
{
}
```

Измените в окне редактора определение класса Tree<TItem> таким образом, чтобы указать, что тип параметра TItem должен обозначать тип, в котором реализуется интерфейс-обобщение IComparable<TItem>. Необходимые изменения выделены в следующем примере кода жирным шрифтом. Измененное определение класса Tree<TItem> должно приобрести следующий вид:

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
}
```

Добавьте к классу Tree<TItem> три открытых автоматически создаваемых свойства: TItem-свойство по имени NodeData и Tree<TItem>-свойства с именами LeftTree и RightTree, выделенные в следующем примере кода жирным шрифтом:

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
    public TItem NodeData { get; set; }
    public Tree<TItem> LeftTree { get; set; }
    public Tree<TItem> RightTree { get; set; }
}
```

Добавьте к классу `Tree<TItem>` конструктор, принимающий единственный `TItem`-параметр по имени `nodeValue`. Установите в конструкторе для свойства `NodeData` значение `nodeValue` и инициализируйте свойства `LeftTree` и `RightTree` null-значениями, как показано в выделенном жирным шрифтом фрагменте следующего кода:

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
    ...
    public Tree(TItem nodeValue)
    {
        this.NodeData = nodeValue;
        this.LeftTree = null;
        this.RightTree = null;
    }
}
```



ПРИМЕЧАНИЕ Обратите внимание на то, что имя конструктора не включает параметр типа: конструктор называется `Tree`, а не `Tree<TItem>`.

Добавьте к классу `Tree<TItem>` открытый метод по имени `Insert`, выделенный в следующем коде жирным шрифтом. Этот метод вставляет `TItem`-значение в дерево.

Определение метода должно иметь следующий вид:

```
public class Tree<TItem> where TItem: IComparable<TItem>
{
    ...
    public void Insert(TItem newItem)
    {
    }
}
```

Реализуйте в методе `Insert` рассмотренный ранее рекурсивный алгоритм для создания упорядоченного двоичного дерева. Конструктор создает исходный узел дерева, поэтому метод `Insert` может считать, что дерево не пустое. Следующий код является частью алгоритма, запускаемого после проверки дерева на пустоту. Он показан здесь, чтобы помочь вам разобраться в коде, который будет вами написан для метода `Insert` при реализации следующих этапов упражнения:

```
...
Изучить значение узла "N" дерева "B"
Если значение "N" больше значения нового элемента "I"
То
    Если левое поддерево "B" пустое
        Создать новое левое поддерево дерева "B" с элементом "I" в качестве узла,
        с пустыми левым и правым поддеревьями
```

Иначе

 Вставить "I" в левое поддерево дерева "B"

Завершить условие Если

...

Добавьте к методу `Insert` инструкцию, объявляющую локальную переменную типа `TItem` по имени `currentNodeValue`. Инициализируйте эту переменную, как показано в следующем коде жирным шрифтом, значением свойства `NodeData`, принадлежащего дереву:

```
public void Insert(TItem newItem)
{
    TItem currentNodeValue = this.NodeData;
}
```

Добавьте к методу `Insert` выделенную в следующем примере кода жирным шрифтом инструкцию `if-else`, поставив ее после определения переменной `currentNodeValue`. Эта инструкция использует метод `CompareTo`, определенный в интерфейсе `IComparable<T>`, чтобы выяснить, является ли значение текущего узла большим, чем значение нового элемента:

```
public void Insert(TItem newItem)
{
    TItem currentNodeValue = this.NodeData;
    if (currentNodeValue.CompareTo(newItem) > 0)
    {
        // Вставка нового элемента в левое поддерево
    }
    else
    {
        // Вставка нового элемента в правое поддерево
    }
}
```

Добавьте к части кода `if` сразу же после комментария `// Вставка нового элемента в левое поддерево` следующие инструкции:

```
if (this.LeftTree == null)
{
    this.LeftTree = new Tree<TItem>(newItem);
}
else
{
    this.LeftTree.Insert(newItem);
}
```

Эти инструкции проверяют левое поддерево на пустоту. Если оно пустое, создается новое поддерево с использованием нового значения, которое прикрепляется к текущему узлу, в противном случае новое значение вставляется в существующее левое поддерево путем рекурсивного вызова метода `Insert`.

Добавьте к части кода `else` самой внешней инструкции `if-else` сразу же после комментария // Вставка нового элемента в правое поддерево следующий эквивалентный код, вставляющий новый узел в правое поддерево:

```
if (this.RightTree == null)
{
    this.RightTree = new Tree<TItem>(newItem);
}
else
{
    this.RightTree.Insert(newItem);
}
```

Добавьте к классу `Tree<TItem>` после метода `Insert` еще один открытый метод по имени `WalkTree`. Этот метод осуществляет сквозной обход дерева, посещая последовательно каждый узел и создавая строку, представляющую данные, содержащиеся в дереве. Определение метода должно иметь следующий вид:

```
public string WalkTree()
{
}
```

Добавьте к методу `WalkTree` инструкции, выделенные в следующем примере кода жирным шрифтом. Эти инструкции реализуют алгоритм, рассмотренный ранее и предназначенный для обхода двоичного дерева. При посещении каждого узла его значение возвращается методом в строку:

```
public string WalkTree()
{
    string result = "";

    if (this.LeftTree != null)
    {
        result = this.LeftTree.WalkTree();
    }

    result += $" {this.NodeData.ToString()} ";

    if (this.RightTree != null)
    {
        result += this.RightTree.WalkTree();
    }

    return result;
}
```

Щелкните в меню Сборка на пункте Собрать решение. Класс должен без проблем пройти компиляцию, но при обнаружении ошибок исправьте код и выполните повторную сборку решения.

В следующем упражнении вы тестируете класс `Tree<TItem>` путем создания двоичных деревьев из целых чисел и строк.

Тестирование класса Tree<TItem>

Щелкните правой кнопкой мыши в обозревателе решений на решении BinaryTree, укажите на пункт Добавить и щелкните на пункте Создать проект.



ПРИМЕЧАНИЕ Щелкнуть правой кнопкой мыши нужно на решении BinaryTree, а не на проекте BinaryTree.

Добавьте новый проект, воспользовавшись шаблоном Консольное приложение. При-
войте проекту имя BinaryTreeTest. Укажите для его размещения папку \\Microsoft
Press\\VCSBS\\Chapter 17 вашей папки документов, а затем щелкните на кнопке OK.



ПРИМЕЧАНИЕ Решение среды Visual Studio 2015 может содержать более одного
проекта. Используйте эту особенность для добавления к решению BinaryTree второго
проекта для тестирования класса Tree<TItem>.

Щелкните в обозревателе решений правой кнопкой мыши на проекте BinaryTreeTest,
а затем щелкните на пункте Назначить автозагружаемым проектом (Set As Startup
Project). В обозревателе решений будет выделен проект BinaryTreeTest. При за-
пуске приложения будет фактически выполняться именно этот проект.

Щелкните в обозревателе решений правой кнопкой мыши на проекте BinaryTreeTest,
укажите на пункт Добавить, а затем щелкните на пункте Ссылка.

Раскройте в левой панели появившегося диалогового окна Менеджер ссылок —
BinaryTreeTest пункт Проекты (рис. 17.4), а затем щелкните на пункте Решение.
Выберите в средней панели проект BinaryTree, установив флажок слева от него,
после чего щелкните на кнопке OK.

На этом этапе к списку ссылок для проекта BinaryTreeTest в обозревателе решений
будет добавлена сборка BinaryTree. Если в обозревателе решений просмотреть
папку Ссылки для проекта BinaryTreeTest, то можно увидеть сбоку BinaryTree, по-
казанную в самом верху. Теперь вы сможете создавать в проекте BinaryTreeTest
объекты Tree<TItem>.



ПРИМЕЧАНИЕ Если проект библиотеки классов не является частью того же реше-
ния, что и использующий его проект, вы должны добавить ссылку на сборку (на файл
с расширением .dll), а не на проект библиотеки классов. Это можно сделать путем
поиска сборки в диалоговом окне менеджера ссылок. Этот прием будет использован
вами в заключительной подборке упражнений данной главы.

Добавьте в окне редактора, показывающего класс Program, находящийся в файле
Program.cs, следующую директиву using, поставив ее в список в самом начале
класса:

```
using BinaryTree;
```

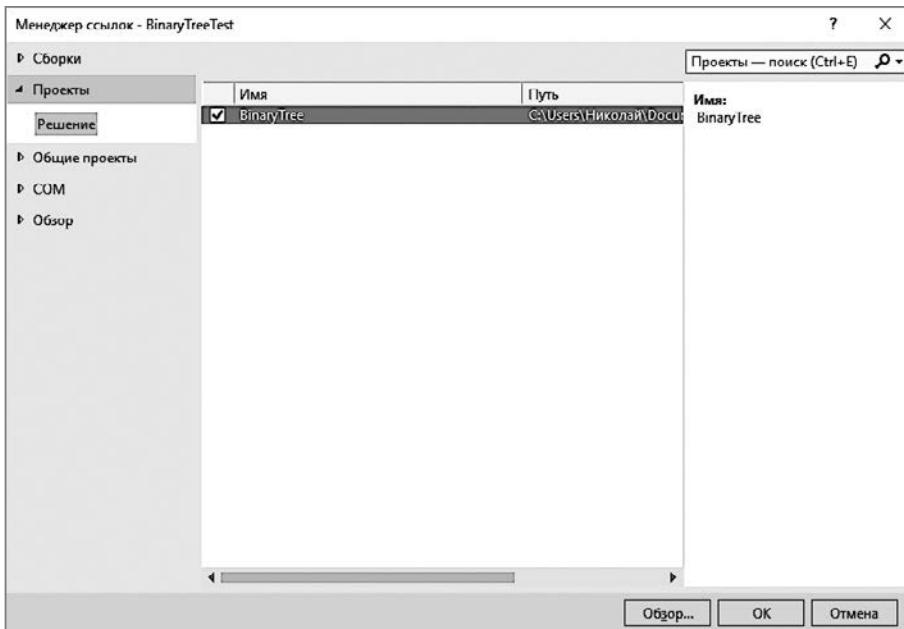


Рис. 17.4

Добавьте к методу `Main` инструкции, выделенные в следующем примере кода жирным шрифтом:

```
static void Main(string[] args)
{
    Tree<int> tree1 = new Tree<int>(10);
    tree1.Insert(5);
    tree1.Insert(11);
    tree1.Insert(5);
    tree1.Insert(-12);
    tree1.Insert(15);
    tree1.Insert(0);
    tree1.Insert(14);
    tree1.Insert(-8);
    tree1.Insert(10);
    tree1.Insert(8);
    tree1.Insert(8);

    string sortedData = tree1.WalkTree();
    Console.WriteLine($"Sorted data is: {sortedData}");
}
```

Эти инструкции создают новое двоичное дерево для хранения целочисленных значений. Конструктор создает исходный узел, содержащий значение 10. Инструкции `Insert` добавляют к дереву узлы, а метод `WalkTree` создает строку,

показывающую содержимое дерева, которое при выводе строки на экран должно появиться отсортированным в возрастающем порядке.



ПРИМЕЧАНИЕ Не забывайте, что ключевое слово `int` в C# является всего лишь псевдонимом для типа `System.Int32`: когда объявляется `int`-переменная, на самом деле объявляется структурная переменная типа `System.Int32`. Тип `System.Int32` реализует интерфейсы `IComparable` и `IComparable<T>`, благодаря чему вы можете создавать `Tree<int>`-объекты. Аналогично этому ключевое слово `string` является псевдонимом для `System.String`, где также реализуются интерфейсы `IComparable` и `IComparable<T>`.

Щелкните в меню Сборка на пункте Собрать решение и добейтесь того, чтобы решение прошло компиляцию. При необходимости исправьте ошибки.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что программа запускается и выводит значения в такой последовательности:

```
-12 -8 0 5 5 8 8 10 10 11 14 15
```

Нажмите клавишу Ввод и вернитесь в среду Visual Studio 2015.

Добавьте к методу `Main` класса `Program` ниже имеющегося кода следующие инструкции, выделенные жирным шрифтом:

```
static void Main(string[] args)
{
    ...
    Tree<string> tree2 = new Tree<string>("Hello");
    tree2.Insert("World");
    tree2.Insert("How");
    tree2.Insert("Are");
    tree2.Insert("You");
    tree2.Insert("Today");
    tree2.Insert("I");
    tree2.Insert("Hope");
    tree2.Insert("You");
    tree2.Insert("Are");
    tree2.Insert("Feeling");
    tree2.Insert("Well");
    tree2.Insert("!");
}

sortedData = tree2.WalkTree();
Console.WriteLine($"Sorted data is: {sortedData}");
```

Эти инструкции создают еще одно двоичное дерево для хранения строк, заполняют его тестовыми данными и выводят на экран. На этот раз данные сортируются в алфавитном порядке.

Щелкните в меню Сборка на пункте Собрать решение и добейтесь того, чтобы решение прошло компиляцию. При необходимости исправьте ошибки.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что программа запускается и выводит целочисленные значения, как и раньше, а за ними выводятся строковые значения в такой последовательности (рис. 17.5):

```
! Are Are Feeling Hello Hope How I Today Well World You You
```

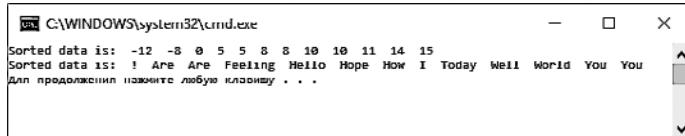


Рис. 17.5

Нажмите клавишу Ввод и вернитесь в среду Visual Studio 2015.

Создание методов-обобщений

Наряду с определением классов-обобщений можно создавать и методы-обобщения.

В методах-обобщениях можно указывать типы параметров и тип возвращаемого значения, используя параметр типа, который применяется практически так же, как и при определении класса-обобщения. Таким образом можно определить методы-обобщения, работа которых не зависит от типа, и избежать издержек на приведение типов (и в некоторых случаях на упаковку). Методы-обобщения зачастую используются в сочетании с классами-обобщениями. Вам они понадобятся как методы, получающие в качестве параметров обобщенные типы, или как методы, у которых тип возвращаемого значения обозначается обобщением.

Методы-обобщения определяются путем использования такого же синтаксиса параметра типа, который применялся при создании классов-обобщений (можно также указать ограничения). Например, метод-обобщение `Swap<T>`, показываемый в следующем фрагменте кода, выполняет для своих параметров обмен значениями. Поскольку эта функция полезна независимо от типа обмениваемых данных, есть смысл определить ее в виде метода-обобщения:

```
static void Swap<T>(ref T first, ref T second)
{
    T temp = first;
    first = second;
    second = temp;
}
```

Метод вызывается указанием соответствующего для его параметра типа. В следующем примере показано, как вызвать метод `Swap<T>` для обмена двух целых чисел и двух строк:

```
int a = 1, b = 2;
Swap<int>(ref a, ref b);
...
string s1 = "Hello", s2 = "World";
Swap<string>(ref s1, ref s2);
```



ПРИМЕЧАНИЕ Как создание экземпляров класса-обобщения с различными параметрами типа заставляет компилятор создавать различные типы, так и каждое отдельно взятое использование метода `Swap<T>` заставляет компилятор создавать различные версии метода. Метод `Swap<int>` не является эквивалентом метода `Swap<string>`, оба метода просто создаются из одного шаблона-обобщения, демонстрируя одинаковое поведение, но по отношению к различным типам.

Определение метода-обобщения для создания двоичного дерева

В предыдущем примере вы создали класс-обобщение для реализации двоичного дерева. Для добавления элементов данных в дерево в классе `Tree<TItem>` предоставляется метод `Insert`. Но если потребуется добавить большое количество элементов, повторные вызова метода `Insert` создадут явные неудобства. В следующем упражнении вы создадите метод-обобщение `InsertIntoTree`, который можно будет использовать для вставки в дерево списка элементов, предоставленных за один вызов. Этот метод будет протестирован путем его использования для вставки списка символов в дерево символов.

Написание метода `InsertIntoTree`

Создайте в среде Visual Studio 2015 новый проект, воспользовавшись для этого шаблоном консольного приложения. В диалоговом окне **Создание проекта** введите для проекта имя `BuildTree`. Укажите для его размещения папку `\Microsoft Press\VCBS\Chapter 17` вашей папки документов. В раскрывающемся списке **Решение** выберите пункт **Создать новое решение**, после чего щелкните на кнопке **OK**.

Щелкните в меню **Проект** на пункте **Добавить ссылку**. Щелкните в диалоговом окне **Менеджер ссылок** — `BuildTree` на кнопке **Обзор** (не перепутайте ее с одноименной вкладкой, находящейся в левой панели).

В диалоговом окне **Выберите файлы**, на которые нужно установить ссылки перейдите в папку `\Microsoft Press\VCBS\Chapter 17\BinaryTree\BinaryTree\bin\Debug` вашей папки документов, щелкните на файле `BinaryTree.dll`, после чего щелкните на кнопке **Добавить**.

Убедитесь в присутствии файла `BinaryTree.dll` в перечне сборок в диалоговом окне Менеджер ссылок — `BuildTree` и в том, что рядом с его именем установлен флажок, после чего щелкните на кнопке `OK`. Сборка `BinaryTree` будет добавлена к перечню ссылок, показанному в обозревателе решений.

Добавьте в окне редактора, показывающего содержимое файла `Program.cs`, в самое начало кода следующую директиву `using`:

```
using BinaryTree;
```

Следует напомнить, что класс `Tree<TItem>` находится именно в этом пространстве имен.

Добавьте после метода `Main` класса `Program` метод `InsertIntoTree`. Он должен быть объявлен статическим и пустым (`static void`), получающим параметр `Tree<TItem>` и `params`-массив из элементов типа `TItem` по имени `data`. Параметр `tree` следует передать по ссылке, исходя из соображений, которые будут рассмотрены при выполнении следующего этапа упражнения.

Определение метода должно иметь следующий вид:

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree,
    params TItem[] data)
{
}
```

Тип `TItem`, который используется для элементов, вставляемых в двоичное дерево, должен реализовывать интерфейс `IComparable<TItem>`. Измените определение метода `InsertIntoTree`, добавив условие `where`, выделенное в следующем коде жирным шрифтом:

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree,
    params TItem[] data) where TItem : IComparable<TItem>
{
}
```

Добавьте к методу `InsertIntoTree` инструкции, выделенные в следующем примере кода жирным шрифтом. Эти инструкции выполняют сквозной обход списка `params`, добавляя каждый элемент к дереву путем использования метода `Insert`. Если указанное параметром `tree` значение изначально `null`, создается новый экземпляр `Tree<TItem>` — именно поэтому параметр `tree` передается по ссылке:

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree,
    params TItem[] data) where TItem : IComparable<TItem>
{
    foreach (TItem datum in data)
    {
        if (tree == null)
```

```
        {
            tree = new Tree<TItem>(datum);
        }
    else
    {
        tree.Insert(datum);
    }
}
```

Тестирование метода InsertIntoTree

Добавьте к методу Main класса Program следующие инструкции, выделенные жирным шрифтом, которые создают новый экземпляр Tree для хранения символьных данных, заполняя его проверочными данными путем использования метода InsertIntoTree, а затем выведите его на экран, воспользовавшись методом WalkTree класса Tree:

```
static void Main(string[] args)
{
    Tree<char> charTree = null;
    InsertIntoTree<char>(ref charTree, 'M', 'X', 'A', 'M', 'Z', 'Z', 'N');
    string sortedData = charTree.WalkTree();
    Console.WriteLine($"Sorted data is: {sortedData}");
}
```

Щелкните в меню Сборка на пункте Собрать решение и добейтесь того, чтобы решение прошло компиляцию. При необходимости исправьте ошибки.

Щелкните в меню Отладка на пункте Запуск без отладки. Программа запустится и выведет символьные значения в следующем порядке:

A M M N X Z Z

Нажмите клавишу Ввод и вернитесь в среду Visual Studio 2015.

Вариантность и интерфейсы-обобщения

В главе 8 было показано, что тип object может использоваться для хранения значения или ссылки на любой другой объект. Например, вполне допустимо воспользоваться следующим кодом:

```
string myString = "Hello";
object myObject = myString;
```

Вспомним, что в понятиях наследования класс String является производным класса Object, следовательно, все строки являются объектами.

Рассмотрим теперь следующий интерфейс и класс, которые являются обобщениями:

```
interface IWrapper<T>
{
    void SetData(T data);
    T GetData();
}
class Wrapper<T> : IWrapper<T>
{
    private T storedData;

    void IWrapper<T>.SetData(T data)
    {
        this.storedData = data;
    }

    T IWrapper<T>.GetData()
    {
        return this.storedData;
    }
}
```

Класс `Wrapper<T>` предоставляет простую оболочку для указанного класса. Интерфейс `IWrapper` определяет метод `SetData`, который реализуется классом `Wrapper<T>` для сохранения данных, и метод `GetData`, который реализуется классом `Wrapper<T>` для извлечения данных. Вы можете создать экземпляр этого класса и воспользоваться им в качестве оболочки для строки:

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
Console.WriteLine($"Stored value is {storedStringWrapper.GetData()}");
```

Этот код создает экземпляр типа `Wrapper<string>`. Чтобы вызвать метод `SetData`, он ссылается на объект через интерфейс `IWrapper<string>`. (В типе `Wrapper<T>` его интерфейсы реализуются явным образом, поэтому вам нужно вызывать методы через соответствующую ссылку на интерфейс.) Код также через интерфейс `IWrapper<string>` вызывает метод `GetData`. Если запустить этот код, он выведет на экран сообщение «Stored value is Hello» («Сохраненным значением является Hello»).

Рассмотрим следующую строку кода:

```
IWrapper<object> storedObjectWrapper = stringWrapper;
```

Имеющаяся в ней инструкция похожа на ту, что создавала ссылку на `IWrapper<string>` в предыдущем примере кода, — разница в том, что параметром типа является `object`, а не `string`. Может ли этот код считаться допустимым?

Вспомним, что все строки по сути являются объектами (вы можете, как показано ранее, присвоить строковое значение ссылке на объект), следовательно, теоретически эта инструкция вселяет надежды. Но если попробовать ее в деле, она не пройдет компиляцию, вызвав появление сообщения «*Cannot implicitly convert type ‘Wrapper<string>’ to ‘IWrapper<object>’*» («Неявное преобразование типа ‘Wrapper<string>’ в тип ‘IWrapper<object>’ выполнить невозможно»).

Можно попробовать применить явное приведение типа:

```
IWrapper<object> storedObjectWrapper = (IWrapper<object>)stringWrapper;
```

Это код пройдет компиляцию, но даст сбой в ходе выполнения приложения с выдачей исключения о недопустимом приведении типа — *InvalidOperationException*. Проблема в том, что хотя все строки являются объектами, обратное утверждение неверно. Если бы выполнение этой инструкции было разрешено, то вы могли бы написать следующий код, который в конечном счете попытался бы сохранить *Circle*-объект в строковом поле:

```
IWrapper<object> storedObjectWrapper = (IWrapper<object>)stringWrapper;
Circle myCircle = new Circle();
storedObjectWrapper.SetData(myCircle);
```

Интерфейс *IWrapper<T>* называется инвариантным (неизменяемым). Вы не можете присвоить объект *IWrapper<A>* ссылке типа *IWrapper*, даже если тип *A* является производным от типа *B*. В C# это ограничение реализуется по умолчанию, чтобы обеспечить безопасность вашего кода в отношении типов.

Ковариантные интерфейсы

Предположим, что вместо интерфейса *IWrapper<T>* вы определили интерфейсы *IStoreWrapper<T>* и *IRetrieveWrapper<T>*, показанные в следующем примере, и реализовали их в классе *Wrapper<T>* следующим образом:

```
interface IStoreWrapper<T>
{
    void SetData(T data);
}

interface IRetrieveWrapper<T>
{
    T GetData();
}

class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>
{
    private T storedData;
```

```

void IStoreWrapper<T>.SetData(T data)
{
    this.storedData = data;
}

T IRetrieveWrapper<T>.GetData()
{
    return this.storedData;
}
}

```

Функционально класс `Wrapper<T>` остался таким же, как и прежде, за исключением того, что доступ к методам `SetData` и `GetData` осуществляется через разные интерфейсы:

```

Wrapper<string> stringWrapper = new Wrapper<string>();
IStoreWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;
Console.WriteLine($"Stored value is {retrievedStringWrapper.GetData()}");

```

Тогда допустим ли следующий код?

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

Можно дать быстрый ответ, что нет, и код не пройдет компиляцию с указанием той же ошибки, что и прежде. Но если вдуматься, становится понятным, что хотя компилятор C# посчитал эту инструкцию небезопасной с точки зрения типов, причины такого решения уже нельзя считать обоснованными. Интерфейс `IRetrieveWrapper<T>` позволяет с помощью метода `GetData` всего лишь считывать данные, хранящиеся в `Wrapper<T>`-объекте, и не предоставляет какого-либо способа изменения данных. В подобных ситуациях, когда в интерфейсе-обобщении параметр типа фигурирует только для возвращаемых методами значений, вы можете проинформировать компилятор, что неявные преобразования вполне допустимы и что он не должен принуждать вас к строгому соблюдению безопасности в отношении типов. Такое информирование осуществляется при объявлении параметра типа с помощью указания ключевого слова `out`:

```

interface IRetrieveWrapper<out T>
{
    T GetData();
}

```

Эта особенность называется *ковариантностью*. При условии допустимости преобразования из типа *A* в тип *B* или при условии, что тип *A* является производным от типа *B*, вы можете присвоить `IRetrieveWrapper<A>`-объект `IRetrieveWrapper`-ссылке. Следующий код теперь проходит компиляцию и выполняется вполне ожидаемым образом:

```
// тип string является производным от типа object, поэтому код является
// вполне допустимым
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

Квалификатор `out` можно указать с параметром типа, только если параметр типа фигурирует исключительно для типа возвращаемых методами значений. Если параметр типа используется для указания типа любого из параметров метода, применение квалификатора `out` недопустимо и ваш код не пройдет компиляцию. К тому же ковариантность работает только со ссылочными типами. Дело в том, что типы значений не могут создавать иерархии наследования. Следовательно, следующий код не пройдет компиляцию, поскольку `int` относится к типу значений:

```
Wrapper<int> intWrapper = new Wrapper<int>();
IStoreWrapper<int> storedIntWrapper = intWrapper; // Эта инструкция допустима
...
// Следующая инструкция недопустима – целочисленные значения не являются объектами
IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;
```

Ковариантность демонстрируют несколько интерфейсов, определенных в .NET Framework, включая интерфейс `IEnumerable<T>`, который подробно рассматривается в главе 19 «Перечисляемые коллекции».



ПРИМЕЧАНИЕ Ковариантными могут объявляться только интерфейсы и делегаты (рассматриваемые в главе 18). Для классов-обобщений модификатор `out` не указывается.

Контрвариантные интерфейсы

Контрвариантность следует тем же принципам, что и ковариантность, за исключением того, что она работает в противоположном направлении, позволяя использовать интерфейс-обобщение для ссылки на объект типа *B* через ссылку на тип *A* при условии, что тип *B* является производным от типа *A*. Сразу усвоить это нелегко, поэтому лучше посмотрим на пример из библиотеки классов .NET Framework.

Пространство имен `System.Collections.Generic` в .NET предоставляет интерфейс под названием `IComparer`, имеющий следующий вид:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

В классе, реализующем этот интерфейс, должен быть определен метод `Compare`, используемый для сравнения двух объектов того типа, который указан параметром типа `T`. Ожидается, что метод `Compare` будет возвращать целочисленное

значение: нуль, если у параметров *x* и *y* одинаковые значения, отрицательное число, если значение *x* меньше значения *y*, и положительное число, если значение *x* больше значения *y*. Следующий код показывает пример, где объекты сортируются в соответствии с их хэш-кодом. (Метод *GetHashCode* реализован классом *Object*. Он просто возвращает целое число, идентифицирующее объект. Все ссылочные типы наследуют этот метод и могут его перегрузить своими собственными реализациями.)

```
class ObjectComparer : IComparer<Object>
{
    int IComparer<Object>.Compare(Object x, Object y)
    {
        int xHash = x.GetHashCode();
        int yHash = y.GetHashCode();

        if (xHash == yHash)
            return 0;
        if (xHash < yHash)
            return -1;

        return 1;
    }
}
```

Для сравнения двух объектов можно создать *ObjectComparer*-объект и через интерфейс *IComparer<Object>* вызвать метод *Compare*:

```
Object x = ...;
Object y = ...;
ObjectComparer objectComparer = new ObjectComparer();
IComparer<Object> objectComparator = objectComparer;
int result = objectComparator.Compare(x, y);
```

Выглядит скучновато. Интереснее будет узнать, что вы можете сослаться на этот же самый объект через версию *IComparer*-интерфейса, сравнивающую строки, например:

```
IComparer<String> stringComparator = objectComparer;
```

На первый взгляд эта инструкция кажется нарушением всех правил обеспечения безопасности типов, которые только можно себе представить. Но если подумать о том, что именно делает интерфейс *IComparer<T>*, то этот подход обретает смысл. Цель метода *Compare* заключается в возвращении значения на основе сравнения переданных ему параметров. Если можно сравнить *Objects*, то, конечно же, можно будет сравнить и *Strings*, объекты которого являются всего лишь специализированными типами класса *Object*. В конце концов, *String* должен справляться со всем, с чем справляется *Object*, — именно в этом и состоит цель наследования.

Но это по-прежнему отчасти похоже на предположение. Как компилятор C# узнает, что вы не собираетесь выполнять какую-либо операцию в отношении конкретного типа в программном коде, принадлежащем методу `Compare`, который может дать сбой, если вызвать метод через интерфейс, основанный на другом типе? Если еще раз посмотреть на определение интерфейса `IComparer`, можно увидеть, что перед параметром типа стоит квалификатор `in`:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

Ключевое слово `in` сообщает компилятору C#, что вы можете передать методам в качестве параметра типа либо тип `T`, либо любой тип, являющийся производным от `T`. Вы не можете использовать `T` в качестве типа значения, возвращаемого из любых методов. По сути, это позволяет вам ссылаться на объект либо через интерфейс-обобщение, основанный на типе объекта, либо через интерфейс-обобщение, основанный на типе, являющемся производным от типа объекта. В общем виде, если тип `A` предоставляет некие операции, свойства или поля, то если тип `B` является производным от типа `A`, он также должен предоставлять те же самые операции (которые могут вести себя по-другому в случае их перегрузки), свойства и поля. Следовательно, должно быть абсолютно безопасно подставлять объект типа `B` вместо объекта типа `A`.

Ковариантность и контравариантность могут показаться в мире обобщений некоторыми дополнительными темами, но от них есть вполне определенная польза. Например, представляющий коллекцию класс-обобщение `List<T>` (в пространстве имен `System.Collections.Generic`) использует для реализации методов `Sort` и `BinarySearch` объекты `IComparer<T>`. Объект типа `List<Object>` может содержать коллекцию объектов любого типа, поэтому методам `Sort` и `BinarySearch` нужна возможность сортировать объекты любого типа. Без использования контравариантности методам `Sort` и `BinarySearch` потребовалось бы включать логику, определяющую реальные типы сортируемых или искомых элементов, а затем реализовывать механизм сортировки или поиска, соответствующий специфике типа. Но человеку, не связанному с математикой, будет нелегко вспомнить, что на самом деле представляют собой ковариантность и контравариантность. Способ, с помощью которого я это запомнил, основан на примерах, приведенных в данном разделе.

- ❑ **Пример ковариантности.** Если методы в интерфейсе-обобщении могут возвращать строки, то они могут возвращать и объекты. (Все строки являются объектами.)
- ❑ **Пример контравариантности.** Если методы в интерфейсе-обобщении могут получать в качестве параметров объекты, они могут получать и строковые

параметры. (Если можно выполнить операцию путем использования объекта, значит, ту же самую операцию можно выполнить и с использованием строки, поскольку все строки являются объектами.)



ПРИМЕЧАНИЕ Как и в случае с ковариантностью, контравариантными могут являться только интерфейсные и делегатные типы. Для классов-обобщений модификатор `in` не указывается.

Выводы

В этой главе вы научились использовать обобщения для создания классов, безопасных с точки зрения типов. Вы увидели, как указанием параметра типа создаются экземпляры обобщенных типов. Вы также узнали, как реализуется интерфейс-обобщение и определяется метод-обобщение. И наконец, научились определять ковариантные и контравариантные интерфейсы-обобщения, способные работать с иерархией типов.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 18.

Если сейчас хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Получить экземпляр объекта путем использования обобщенного типа	Укажите для обобщения соответствующий параметр типа, например: <code>Queue<int> myQueue = new Queue<int>();</code>
Создать новый обобщенный тип	Определите класс, используя параметр типа, например: <code>public class Tree<TItem></code> <code>{</code> <code> ...</code> <code>}</code>
Наложить ограничение на тип, который может быть подставлен вместо обобщенного параметра типа	При определении класса укажите ограничение путем использования условия <code>where</code> , например: <code>public class Tree<TItem></code> <code>where TItem : IComparable<TItem></code> <code>{</code> <code> ...</code> <code>}</code>

Чтобы	Сделайте следующее
Определить метод-обобщение	Определите метод путем использования параметров типа, например: <pre>static void InsertIntoTree<TItem> (Tree<TItem> tree, params TItem[] data) { ... }</pre>
Вызвать метод-обобщение	Предоставьте типы для каждого из параметров типа, например: <code>InsertIntoTree<char>(charTree, 'Z', 'X');</code>
Определить ковариантный интерфейс	Укажите для ковариантных параметров типа квалификатор <code>out</code> . Ссылайтесь на ковариантные параметры типа только в качестве типов значений, возвращаемых из методов, но не в качестве типов для параметров метода: <pre>interface IRetrieveWrapper<out T> { T GetData(); }</pre>
Определить контрвариантный интерфейс	Укажите для контрвариантных параметров типа квалификатор <code>in</code> . Ссылайтесь на контрвариантные параметры типа только в качестве типов параметров метода, но не в качестве типов возвращаемых значений: <pre>public interface IComparer<in T> { int Compare(T x, T y); }</pre>

18 Использование коллекций

Прочитав эту главу, вы научитесь:

- объяснять, какие функциональные возможности предоставляются в различных классах коллекций, доступных в среде .NET Framework;
- создавать безопасные по отношению к типам коллекции;
- заполнять коллекции набором данных;
- получать доступ к элементам данных, хранящимся в коллекциях, и оперировать ими;
- искать в списочной коллекции соответствующие элементы с помощью предиката.

В главе 10 «Использование массивов» были представлены массивы, предназначенные для хранения данных. В этом качестве массивы, безусловно, очень полезны, но у них имеются ограничения. Массивы предоставляют весьма скромные функциональные возможности, например, нелегко дается увеличение или уменьшение размера массива, да и сортировать хранящиеся в массиве данные не так-то просто. Кроме того, массивы фактически предоставляют только одно средство доступа к данным — целочисленный индекс. Если ваше приложение нуждается в сохранении данных с использованием какого-либо другого механизма, например, основанного на применении очереди, работающей по принципу «первым пришел — первым ушел», рассмотренного в главе 17 «Введение в обобщения», массивы могут оказаться не самой удобной структурой данных. И здесь свою пользу могут доказать коллекции.

Что такое классы коллекций?

Среда Microsoft .NET предоставляет несколько классов, которые собирают элементы вместе таким образом, чтобы приложение могло получать доступ к элементам конкретно указанными способами. Это уже упоминавшиеся

в главе 17 классы коллекций, которые находятся в пространстве имен `System.Collections.Generic`.

Из названия этого пространства имен видно, что коллекции относятся к обобщенным типам: все они ожидают от вас предоставления параметра типа, показывающего разновидность тех данных, которые будет в них хранить ваше приложение. Каждый класс коллекции оптимизирован под конкретную форму хранения данных и доступа к ним, и каждый из них предоставляет специализированные методы, поддерживающие эти функциональные возможности. Например, класс `Stack<T>` реализует модель «последним пришел — первым ушел», где элемент добавляется к вершине стека путем использования метода `Push` и забирается с вершины стека путем использования метода `Pop`. Метод `Pop` всегда извлекает самый последний помещенный в стек элемент и удаляет его из стека. В отличие от этого тип `Queue<T>` предоставляет методы `Enqueue` и `Dequeue`, рассмотренные в главе 17. Метод `Enqueue` добавляет элемент к очереди, а метод `Dequeue` извлекает элементы из очереди в порядке, реализующем модель «первым пришел — первым ушел». Доступно также множество других классов коллекций, наиболее востребованные из которых сведены в табл. 18.1.

Таблица 18.1

Коллекция	Описание
<code>List<T></code>	Список объектов, доступных по индексу, как в случае с массивом, но с дополнительными методами, позволяющими вести поиск в списке и сортировать его содержимое
<code>Queue<T></code>	Структура данных, работающая по принципу «первым пришел — первым ушел», с методами для добавления элемента к одному из концов очереди, удаления элемента с другого конца и изучения содержимого элемента без его удаления
<code>Stack<T></code>	Структура данных, работающая по принципу «первым пришел — последним ушел», с методами для помещения элемента на вершину стека, извлечения элемента с вершины стека и изучения элемента, находящегося на вершине стека, без его удаления
<code>LinkedList<T></code>	Двусторонний упорядоченный список, оптимизированный под поддержку вставки и удаления элементов с любого конца. Эта коллекция может работать как очередь или как стек, но, как и список, поддерживает произвольный доступ
<code>HashSet<T></code>	Неупорядоченный набор значений, оптимизированный под быстрое извлечение данных. Предоставляет ориентированные на наборы методы для определения того, содержат ли элементы поднаборы таких же элементов в другом <code>HashSet<T></code> -объекте, а также для вычисления пересечений и объединений <code>HashSet<T></code> -объектов

Таблица 18.1 (окончание)

Коллекция	Описание
Dictionary< TKey, TValue >	Коллекция значений, которые можно идентифицировать и извлечь с помощью ключей, а не индексов
SortedList< TKey, TValue >	Отсортированный список пар «ключ–значение». Ключи должны реализовывать IComparable< T >-интерфейс

Краткий обзор этих классов коллекций дается в следующем разделе. Более подробные сведения о каждом классе можно найти в документации библиотеки классов .NET Framework.



ПРИМЕЧАНИЕ В библиотеке классов .NET Framework в пространстве имен System.Collections предоставляется еще один набор типов коллекций. Это необобщенные коллекции, разработанные до того, как в C# появилась поддержка типов-обобщений (обобщения были добавлены к версии C#, разработанной для .NET Framework версии 2.0). За единственным исключением, все эти типы хранят ссылки на объекты, и от вас при сохранении или извлечении элементов требуется соответствующее приведение типов. Эти классы включены в библиотеку с целью обратной совместимости с существующими приложениями, и применять их при создании новых решений не рекомендуется. Фактически при создании приложений универсальной платформы Windows (UWP) эти классы недоступны.

Единственным не содержащим ссылок на объект является класс BitArray. Этот класс реализует компактный массив булевых значений за счет использования int-значений, в которых каждый бит означает true (1) или false (0). Конечно же, это напоминает структуру IntBits, которая уже встречалась в главе 16 «Использование индексаторов». Класс BitArray доступен для создания UWP-приложений.

Доступен и другой набор коллекций, чьи классы определены в пространстве имен System.Collections.Concurrent. Это коллекции классов, предназначенные для безопасной работы в многопоточной среде, которыми можно воспользоваться при создании многопоточных приложений. Более подробно эти классы будут рассмотрены в главе 24 «Сокращение времени отклика путем выполнения асинхронных операций».

Класс коллекций List<T>

Класс-обобщение List< T > является самым простым из классов коллекций. Его можно использовать практически так же, как массив, ссылаясь на существующий в коллекции List< T > элемент с использованием обычной для массивов системы записи с квадратными скобками и индексом элемента, хотя для добавления новых элементов эту систему записи использовать нельзя. Но в целом класс List< T > обеспечивает более высокую степень гибкости, чем массивы, и был разработан для преодоления следующих свойственных массивам ограничений.

- ❑ При необходимости изменения размера массива приходится создавать новый массив, копировать элементы (отбрасывая при этом некоторые из них, если массив становится меньше), после чего обновлять все ссылки на исходный массив, чтобы теперь они указывали на новый массив.
- ❑ При необходимости удаления элемента из массива приходится перемещать все следующие за ним элементы на одну позицию. Но даже это еще не все, поскольку у вас останутся две копии последнего элемента.
- ❑ При необходимости вставить элемент в массив приходится перемещать элементы на одну позицию, чтобы появилось свободное место. Но при этом последний элемент массива будет потерян!

Класс коллекций `List<T>` предоставляет следующие возможности, устраниющие перечисленные ограничения.

- ❑ Указывать емкость коллекции `List<T>` при ее создании не нужно. Коллекция может расширяться и сужаться по мере добавления или удаления элементов. Подобный динамичный характер поведения не обходится без издержек, и при необходимости можно указать начальный размер. Но если он будет превышен, то в силу необходимости коллекция `List<T>` просто расширится.
- ❑ Для удаления из коллекции `List<T>` указанного элемента можно воспользоваться методом `Remove`. Элементы коллекции `List<T>` автоматически перестроятся, закрывая прореху. С помощью метода `RemoveAt` можно также удалить элемент, указав его позицию в коллекции `List<T>`.
- ❑ Можно добавить элемент к концу коллекции `List<T>`, воспользовавшись имеющимся в ее классе методом `Add`, которому предоставляется добавляемый элемент. Размер коллекции изменяется `List<T>` автоматически.
- ❑ Можно вставить элемент в середину коллекции `List<T>`, воспользовавшись для этого методом `Insert`. При этом размер коллекции `List<T>` также изменится автоматически.
- ❑ Можно без особого труда отсортировать данные `List<T>`-объекта, вызвав для этого метод `Sort`.



ПРИМЕЧАНИЕ Как и при работе с массивами, если для последовательного обхода элементов коллекции `List<T>` используется цикл `foreach`, то воспользоваться переменной итерации для изменения содержимого коллекции вы не сможете. Кроме того, в цикле `foreach`, обходящем все элементы коллекции `List<T>`, нельзя использовать метод `Remove`, `Add` или `Insert` — любая попытка такого рода приведет к выдаче исключения `InvalidOperationException`.

Рассмотрим пример, показывающий, как можно создавать содержимое коллекции `List<int>`, работать с этим содержимым и выполнять последовательный обход элементов:

```
using System;
using System.Collections.Generic;
...
List<int> numbers = new List<int>();

// Заполнение List<int> с помощью метода Add
foreach (int number in new int[12]{10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1})
{
    numbers.Add(number);
}

// Вставка элемента в предпоследнюю позицию списка и сдвиг последнего элемента
// вверх по списку
// В первом параметре указывается позиция, а во втором – вставляемое значение
numbers.Insert(numbers.Count-1, 99);

// Удаление первого элемента, чье значение равно 7 (4-й элемент с индексом 3)
numbers.Remove(3);
// Удаление элемента, который теперь находится на 7-й позиции с индексом 6 (10)
numbers.RemoveAt(6);

// Последовательный обход оставшихся 11 элементов с использованием инструкции for
Console.WriteLine("Iterating using a for statement:");
for (int i = 0; i < numbers.Count; i++)
{
    int number = numbers[i]; // Обратите внимание на использование такого же
                            // синтаксиса, как и у массива
    Console.WriteLine(number);
}

// Обход тех же 11 элементов с использованием инструкции foreach
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

А вот как выглядит информация, выведенная данным кодом на экран:

```
Iterating using a for statement:
10
9
8
7
6
5
4
3
2
99
1
Iterating using a foreach statement:
10
9
8
```

7
6
5
4
3
2
99
1



ПРИМЕЧАНИЕ Способ определения количества элементов в коллекции `List<T>` отличается от запроса количества элементов в массиве. При использовании коллекции `List<T>` исследуется свойство `Count`, а при использовании массива — свойство `Length`.

Класс коллекций `LinkedList<T>`

Класс коллекций `LinkedList<T>` реализует двусвязный список. В каждом элементе списка содержится значение элемента со ссылкой на следующий элемент списка (свойство `Next`) и его предыдущий элемент (свойство `Previous`). У элемента в начале списка свойство `Previous` имеет значение `null`, и такое же значение имеет свойство `Next` у элемента, расположенного в конце списка.

В отличие от класса `List<T>`, в классе `LinkedList<T>` при вставке или исследовании элементов система записи, присущая массивам, не поддерживается. Вместо этого можно воспользоваться методом `AddFirst` для вставки элемента в начало списка с перемещением предыдущего первого элемента дальше по списку и установки в качестве значения его свойства `Previous` ссылки на новый элемент. Аналогично этому для вставки элемента в конец списка можно воспользоваться методом `AddLast`, что повлечет за собой установку в качестве значения свойства `Next` предыдущего последнего элемента ссылки на новый элемент. Для вставки элемента перед указанным элементом списка или после него (с предварительным извлечением этого элемента) можно также воспользоваться методами `AddBefore` и `AddAfter`.

Первый элемент коллекции `LinkedList<T>` можно найти, запросив значение свойства `First`, а свойство `Last` даст ссылку на последний элемент списка. Для последовательного обхода элементов связного списка можно приступить к этой операции с одного конца и пошагово применять ссылки из свойства `Next` или `Previous`, пока не будет найден элемент, у которого это свойство имеет значение `null`. В качестве альтернативного варианта можно воспользоваться инструкцией `foreach`, которая выполнит последовательный обход элементов вперед по списку `LinkedList<T>`-объекта, автоматически остановившись в конце.

Удаление элемента из коллекции `LinkedList<T>` осуществляется с помощью методов `Remove`, `RemoveFirst` и `RemoveLast`.

В следующем примере показана работа с элементами коллекции `LinkedList<T>`. Обратите внимание на то, что код, реализующий обход элементов списка с помощью инструкции `for`, пошагово использует ссылки из свойства `Next` или `Previous`, останавливаясь только в случае встречи ссылки со значением `null`, имеющейся в конце списка:

```
using System;
using System.Collections.Generic;
...
LinkedList<int> numbers = new LinkedList<int>();

// Заполнение List<int> с помощью метода AddFirst
foreach (int number in new int[] { 10, 8, 6, 4, 2 })
{
    numbers.AddFirst(number);
}

// Итерация с использованием инструкции for
Console.WriteLine("Iterating using a for statement:");
for (LinkedListNode<int> node = numbers.First; node != null; node = node.Next)
{
    int number = node.Value;
    Console.WriteLine(number);
}

// Итерация с использованием инструкции foreach
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// Итерация в обратном направлении
Console.WriteLine("\nIterating list in reverse order:");
for (LinkedListNode<int> node = numbers.Last; node != null; node = node.Previous)
{
    int number = node.Value;
    Console.WriteLine(number);
}
```

А вот как выглядит информация, выведенная данным кодом на экран:

```
Iterating using a for statement:
2
4
6
8
10
Iterating using a foreach statement:
2
4
6
8
10
```

```
Iterating list in reverse order:  
10  
8  
6  
4  
2
```

Класс коллекций Queue<T>

В классе `Queue<T>` реализован механизм «первым пришел — первым ушел». Элемент вставляется в конец очереди (операция `Enqueue`) и удаляется из начала очереди (операция `Dequeue`).

В следующем коде показаны пример `Queue<int>`-коллекции и самые распространенные из проводимых с ней операций:

```
using System;  
using System.Collections.Generic;  
...  
Queue<int> numbers = new Queue<int>();  
  
// Заполнение очереди  
Console.WriteLine("Populating the queue:");  
foreach (int number in new int[4]{9, 3, 7, 2})  
{  
    numbers.Enqueue(number);  
    Console.WriteLine($"{number} has joined the queue");  
}  
  
// Последовательный обход элементов очереди  
Console.WriteLine("\nThe queue contains the following items:");  
foreach (int number in numbers)  
{  
    Console.WriteLine(number);  
}  
  
// Опустошение очереди  
Console.WriteLine("\nDraining the queue:");  
while (numbers.Count > 0)  
{  
    int number = numbers.Dequeue();  
    Console.WriteLine($"{number} has left the queue");  
}
```

А вот как выглядит информация, выведенная данным кодом на экран:

```
Populating the queue:  
9 has joined the queue  
3 has joined the queue  
7 has joined the queue  
2 has joined the queue  
The queue contains the following items:  
9  
3
```

```

7
2
Draining the queue:
9 has left the queue
3 has left the queue
7 has left the queue
2 has left the queue

```

Класс коллекций Stack<T>

В классе `Stack<T>` реализован механизм «последним пришел — первым ушел». Элемент присоединяется к вершине стека (операция `pop`). Представьте себе стопку тарелок: новые тарелки кладут на вершину стопки и удаляют их с вершины, при этом последняя помещенная в стек тарелка удаляется первой. (Тарелка в самом низу используется редко, и прежде чем класть в нее еду, ее нужно помыть, поскольку к этому времени она сильно запылится!) Рассмотрим пример, обращая внимание на порядок, в котором инструкция `foreach` выводит элементы на экран:

```

using System;
using System.Collections.Generic;
...
Stack<int> numbers = new Stack<int>();

// Заполнение стека
Console.WriteLine("Pushing items onto the stack:");
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Push(number);
    Console.WriteLine($"{number} has been pushed on the stack");
}

// Последовательный обход элементов стека
Console.WriteLine("\nThe stack now contains:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// Опустошение стека
Console.WriteLine("\nPopping items from the stack:");
while (numbers.Count > 0)
{
    int number = numbers.Pop();
    Console.WriteLine($"{number} has been popped off the stack");
}

```

А вот как выглядит информация, выведенная программой на экран:

```

Pushing items onto the stack:
9 has been pushed on the stack

```

```
3 has been pushed on the stack
7 has been pushed on the stack
2 has been pushed on the stack
The stack now contains:
2
7
3
9
Popping items from the stack:
2 has been popped off the stack
7 has been popped off the stack
3 has been popped off the stack
9 has been popped off the stack
```

Класс коллекций `Dictionary< TKey, TValue >`

Массив и объекты типа `List< T >` предоставляют способ отображения на элемент целочисленного индекса. Целочисленный индекс указывается с помощью квадратных скобок (например, `[4]`), и извлекается элемент по индексу 4, будучи фактически пятым. Но иногда может понадобиться реализация отображения, при котором используется другой, не целочисленный тип, например `string`, `double` или `Time`. В других языках программирования такая организация хранения данных часто называется ассоциативным массивом. Эта функциональная возможность реализуется в классе `Dictionary< TKey, TValue >` путем внутреннего обслуживания двух массивов, один из которых предназначен для ключей, от которых выполняется отображение на одно из отображаемых значений. Когда в коллекцию `Dictionary< TKey, TValue >` вставляется пара «ключ–значение», класс автоматически отслеживает принадлежность ключа к значению, позволяя быстро и легко извлекать значение, связанное с указанным ключом. В конструкции класса `Dictionary< TKey, TValue >` имеется ряд важных особенностей.

- ❑ В коллекции `Dictionary< TKey, TValue >` не могут содержаться продублированные ключи. Если для добавления уже имеющегося в массиве ключа вызывается метод `Add`, выдается исключение. Но для добавления пары «ключ–значение» можно воспользоваться системой записи с использованием квадратных скобок (как показано в следующем примере), не опасаясь при этом выдачи исключения, даже если ключ уже был добавлен: любое значение с таким же самым ключом будет переписано новым значением. Протестировать наличие в коллекции `Dictionary< TKey, TValue >` конкретного ключа можно с помощью метода `ContainsKey`.
- ❑ По внутреннему устройству коллекция `Dictionary< TKey, TValue >` является разряженной структурой данных, работающей наиболее эффективно, когда в ее распоряжении имеется довольно большой объем памяти. По мере вставки элементов размер коллекции `Dictionary< TKey, TValue >` в памяти может очень быстро увеличиваться.

- Когда для последовательного обхода элементов коллекции `Dictionary< TKey, TValue >` используется инструкция `foreach`, возвращается элемент `KeyValuePair< TKey, TValue >`. Это структура, содержащая копию элементов ключа и значения, находящихся в коллекции `Dictionary< TKey, TValue >`, и доступ к каждому элементу можно получить через свойства `Key` и `Value`. Эти элементы доступны только для чтения, и их нельзя использовать для изменения данных в коллекции `Dictionary< TKey, TValue >`.

Далее показан пример, в котором данные о возрасте членов моей семьи связываются с их именами, а затем эта информация выводится на экран:

```
using System;
using System.Collections.Generic;
...
Dictionary<string, int> ages = new Dictionary<string, int>();

// Заполнение словаря Dictionary
ages.Add("John", 51); // Использование метода Add
ages.Add("Diana", 50);
ages["James"] = 23;    // Использование такой же системы записи, что и у массивов
ages["Francesca"] = 21;

// Последовательный обход элементов с использованием инструкции foreach,
// при этом оператор создает элемент KeyValuePair
Console.WriteLine("The Dictionary contains:");
foreach (KeyValuePair<string, int> element in ages)
{
    string name = element.Key;
    int age = element.Value;
    Console.WriteLine($"Name: {name}, Age: {age}");
}
```

А вот что данная программа выведет на экран:

```
The Dictionary contains:
Name: John, Age: 51
Name: Diana, Age: 50
Name: James, Age: 23
Name: Francesca, Age: 21
```



ПРИМЕЧАНИЕ Пространство имен `System.Collections.Generic` включает также тип коллекций `SortedDictionary< TKey, TValue >`. Этот класс содержит коллекцию в порядке, получаемом сортировкой ключей.

Класс коллекций `SortedList< TKey, TValue >`

Класс `SortedList< TKey, TValue >` очень похож на класс `Dictionary< TKey, TValue >` тем, что его можно использовать для связи ключей со значениями. Основное отличие заключается в том, что массив ключей всегда находится в отсортированном состоянии, поэтому класс и носит название `SortedList`. В большинстве

случаев времени на вставку данных в объект типа `SortedList<TKey, TValue>` тратится больше, чем на вставку в объект `SortedDictionary<TKey, TValue>`, но данные зачастую извлекаются быстрее (во всяком случае так же быстро), и класс `SortedList<TKey, TValue>` использует меньше памяти.

Когда в коллекцию `SortedList<TKey, TValue>` вставляется пара «ключ–значение», ключ вставляется в массив ключей на место с правильным индексом, чтобы массив ключей оставался отсортированным. Затем значение вставляется в массив значений на место с таким же индексом. Класс `SortedList<TKey, TValue>` автоматически обеспечивает поддержку синхронизированности ключей и значений даже при добавлении и удалении элементов. Это означает, что вставлять пары «ключ–значение» в `SortedList<TKey, TValue>` можно в любой последовательности — они всегда будут отсортированы на основе значений ключей.

Как и класс `Dictionary<TKey, TValue>`, коллекция `SortedList<TKey, TValue>` не может содержать продублированные ключи. Когда для последовательного обхода элементов коллекции `SortedList<TKey, TValue>` используется инструкция `foreach`, возвращается элемент `KeyValuePair<TKey, TValue>`. Но элементы `KeyValuePair<TKey, TValue>` будут возвращаться отсортированными по значению свойства `Key`.

Далее показан тот же пример, в котором данные о возрасте членов моей семьи связываются с их именами, после чего информация выводится на экран, но эта версия настроена на использование вместо коллекции `Dictionary<TKey, TValue>` объекта `SortedList<TKey, TValue>`:

```
using System;
using System.Collections.Generic;
...
SortedList<string, int> ages = new SortedList<string, int>();

// заполнение SortedList
ages.Add("John", 51); // Использование метода Add
ages.Add("Diana", 50);
ages["James"] = 23; // Использование такой же системы записи, что и у массивов
ages["Francesca"] = 21;

// Последовательный обход элементов с использованием инструкции foreach,
// при этом итератор создает элемент KeyValuePair
Console.WriteLine("The SortedList contains: ");
foreach (KeyValuePair<string, int> element in ages)
{
    string name = element.Key;
    int age = element.Value;
    Console.WriteLine($"Name: {name}, Age: {age}");
}
```

Информация, выведенная этой программой, отсортирована в алфавитном порядке по именам членов моей семьи:

```
The SortedList contains:
Name: Diana, Age: 50
Name: Francesca, Age: 21
Name: James, Age: 23
Name: John, Age: 51
```

Класс коллекций HashSet<T>

Класс `HashSet<T>` оптимизирован для выполнения набора таких операций, как обнаружение присутствия элемента в наборе и создание объединений и пересечений наборов.

Элементы в коллекцию `HashSet<T>` добавляются с помощью метода `Add`, а удаляются из нее с помощью метода `Remove`. Но настоящая эффективность класса `HashSet<T>` определяется предоставлением методов `IntersectWith`, `UnionWith` и `ExceptWith`. Эти методы изменяют коллекцию `HashSet<T>`, создавая из нее новый набор, который содержит либо элементы, встречающиеся в обоих указанных коллекциях, либо элементы из обеих коллекций, либо элементы, которых нет в указанной `HashSet<T>`-коллекции. Эти операции носят деструктивный характер, поскольку в результате их выполнения исходное содержимое `HashSet<T>`-объекта переписывается новым набором данных. Можно также определить, являются ли данные исходной `HashSet<T>`-коллекции поднабором данных другой коллекции и наоборот, для чего используются методы `IsSubsetOf`, `IsSupersetOf`, `IsProperSubsetOf` и `IsProperSupersetOf`. Эти методы возвращают булевые значения и не разрушают данные коллекции.

Внутри `HashSet<T>`-коллекции хранится хэш-таблица, позволяющая выполнить быстрый поиск элементов. Но для быстрой работы с данными крупной `HashSet<T>`-коллекции может потребоваться большой объем памяти.

В следующем примере показывается, как заполнить `HashSet<T>`-коллекцию, и иллюстрируется использование метода `IntersectWith` для поиска данных, присутствующих в обоих наборах:

```
using System;
using System.Collections.Generic;
...
HashSet<string> employees = new HashSet<string>(new string[]
    {"Fred", "Bert", "Harry", "John"});
HashSet<string> customers = new HashSet<string>(new string[]
    {"John", "Sid", "Harry", "Diana"});

employees.Add("James");
customers.Add("Francesca");

Console.WriteLine("Employees:");
foreach (string name in employees)
{
```

```
Console.WriteLine(name);
}

Console.WriteLine("\nCustomers:");
foreach (string name in customers)
{
    Console.WriteLine(name);
}

Console.WriteLine("\nCustomers who are also employees:");
customers.IntersectWith(employees);
foreach (string name in customers)
{
    Console.WriteLine(name);
}
```

Этот код выведет на экран следующую информацию:

```
Employees:
Fred
Bert
Harry
John
James
Customers:
John
Sid
Harry
Diana
Francesca
Customers who are also employees:
John
Harry
```



ПРИМЕЧАНИЕ В пространстве имен System.Collections.Generic предоставляется также тип коллекций SortedSet<T>, работа которого похожа на работу класса HashSet<T>. Основное отличие, судя по названию, состоит в том, что данные в этой коллекции содержатся в отсортированном порядке. Классы SortedSet<T> и HashSet<T> обладают функциональной совместимостью, позволяющей, к примеру, объединить коллекцию SortedSet<T> с коллекцией HashSet<T>.

Использование инициализаторов коллекций

В примерах из предыдущих разделов было показано, как к коллекции с помощью наиболее походящего для нее метода (`Add` для коллекции `List<T>`, `Enqueue` для коллекции `Queue<T>`, `Push` для коллекции `Stack<T>` и т. д.) добавляются отдельные элементы. Некоторые типы коллекций можно инициализировать при их объявлении, используя для этого синтаксис, похожий на тот, который

поддерживается массивами. Например, следующая инструкция создает и инициализирует показанный ранее `List<int>`-объект `numbers`, демонстрируя альтернативу многократному вызову метода `Add`:

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

Внутри среды компилятор C# преобразует эту инициализацию в серию вызовов метода `Add`. Следовательно, этот синтаксис может использоваться только для коллекций, поддерживающих метод `Add`. (Классы `Stack<T>` и `Queue<T>` его не поддерживают.)

Для более сложных коллекций, принимающих пары «ключ–значение», например коллекций класса `Dictionary< TKey, TValue >`, можно указывать значения каждого ключа, пользуясь системой записи индексатора:

```
Dictionary<string, int> ages = new Dictionary<string, int>()
{
    ["John"] = 51,
    ["Diana"] = 50,
    ["James"] = 23,
    ["Francesca"] = 21
};
```

При желании можно также указать в списке инициализации каждую пару «ключ–значение» в виде безымянных типов:

```
Dictionary<string, int> ages = new Dictionary<string, int>()
{
    {"John", 51},
    {"Diana", 50},
    {"James", 23},
    {"Francesca", 21}
};
```

В данном случае первым элементом каждой пары является ключ, а вторым — значение. Для придания коду максимальной разборчивости при инициализации словарного типа рекомендуется везде, где только можно, использовать систему записи, присущую индексаторам.

Методы Find, предикаты и лямбда-выражения

Используя словарно-ориентированные коллекции `Dictionary< TKey, TValue >`, `SortedDictionary< TKey, TValue >` и `SortedList< TKey, TValue >`, можно быстро найти значение, указав искомый ключ, и, как уже было показано в предыдущих примерах, для доступа к значению можно воспользоваться системой записи, присущей массивам. В других коллекциях, поддерживающих произвольный доступ без

применения ключей, например в коллекциях классов `List<T>` и `LinkedList<T>`, использование системы записи, присущей массивам, не поддерживается, но вместо этого для обнаружения элемента предоставляется метод `Find`. В этих классах аргументом для метода `Find` служит предикат, в котором указывается критерий поиска. Формой предиката является метод, исследующий каждый элемент коллекции и возвращающий булево значение, показывающее, соответствует ли элемент критерию поиска. В случае использования метода `Find`, как только будет найдено первое же совпадение, возвращается соответствующий элемент. Следует учесть, что классы `List<T>` и `LinkedList<T>` поддерживают и другие методы, например метод `FindLast`, возвращающий последний совпадающий объект, а класс `List<T>` кроме этого поддерживает метод `FindAll`, возвращающий коллекцию `List<T>`, состоящую из всех элементов, соответствующих условиям поиска.

Самым простым методом указания предиката является использование лямбда-выражения. Так называется выражение, возвращающее метод. Это звучит несколько необычно, поскольку большинство встречавшихся до сих пор в C# выражений возвращали значение. Если вам знакомы языки функционального программирования, например Haskell, то вы, вероятно, знакомы и с этим понятием. Если же нет, не стоит переживать: в лямбда-выражениях нет ничего слишком сложного, и после освоения новых синтаксических особенностей вы сможете убедиться в том, что они весьма полезны.



ПРИМЕЧАНИЕ Если вас заинтересовало функциональное программирование на языке Haskell, зайдите на веб-сайт, посвященный этому языку, который находится по адресу <http://www.haskell.org/haskellwiki/Haskell>.

В главе 3 «Создание методов и применение областей видимости» объясняется, что обычный метод состоит из четырех элементов: возвращаемого типа, имени метода, списка параметров и тела метода. Лямбда-выражение содержит два из этих элементов: список параметров и тело метода. В лямбда-выражении не определяется имя метода, а возвращаемый тип (при наличии такового) выводится из контекста, в котором используется лямбда-выражение. В случае использования метода `Find` предикат по очереди обрабатывает каждый элемент коллекции: тело предиката должно исследовать элемент и в зависимости от его соответствия критерию поиска возвратить `true` или `false`. В следующем примере метод `Find` (выделенный жирным шрифтом) показан в коллекции `List<Person>`, где `Person` является структурой. Метод `Find` возвращает первый элемент списка, у которого для свойства `ID` установлено значение 3:

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
```

```

}
...
// Создание и заполнение штатного расписания
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 51 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};

// Поиск элемента списка, у которого свойство ID имеет значение 3
Person match = personnel.Find((Person p) => { return p.ID == 3; });

Console.WriteLine($"ID: {match.ID}\nName: {match.Name}\nAge: {match.Age}");

```

А вот что этот код выводит на экран:

```

ID: 3
Name: Fred
Age: 34

```

В вызове метода `Find` аргумент `(Person p) => { return p.ID == 3; }` является лямбда-выражением, фактически выполняющим всю работу. В нем содержатся следующие элементы синтаксиса:

- ❑ список параметров, заключенный в круглые скобки. Как и в обычном методе, если определяемый метод (как в предыдущем примере) не принимает никаких параметров, вы все равно должны поставить круглые скобки. В случае использования метода `Find` предикату по очереди предоставляется каждый элемент коллекции, и этот элемент передается в качестве параметра лямбда-выражению;
- ❑ оператор `=>`, который показывает компилятору C#, что это лямбда-выражение;
- ❑ тело метода. Здесь приведен очень простой пример, содержащий всего одну инструкцию, возвращающую булево значение, показывающее, соответствует ли элемент, указанный в параметре, критерию поиска. Но в лямбда-выражении может содержаться несколько инструкций, которые можно отформатировать для наилучшей читаемости. Нужно лишь не забыть поставить после каждой из них точку с запятой, как это делается в обычном методе.



ВНИМАНИЕ В главе 3 также было показано, как оператор `=>` используется для определения методов, тело которых заключено в выражение. По совершенно непонятной причине оператор `=>` имеет такое вот совместное использование. При некоторой схожести системы записи методы, имеющие тело, заключенное в выражение, и лямбда-выражения семантически (и функционально) — совершенно разные вещи, которые не следует путать друг с другом.

Собственно говоря, тело лямбда-выражения может быть телом метода, содержащим несколько инструкций, или одним выражением. Если тело лямбда-выражения содержит только одно выражение, то фигурные скобки и точку с запятой можно не ставить (но для завершения всей инструкции точка с запятой все же понадобится). Кроме того, если выражение получает один параметр, вы можете не ставить вокруг него круглые скобки. И наконец, во многих случаях можно не указывать тип параметров, поскольку компилятор сам извлекает эту информацию из того контекста, в котором вызывается лямбда-выражение. Упрощенная форма ранее показанной инструкции `Find` имеет следующий вид, в котором намного проще разобраться:

```
Person match = personnel.Find(p => p.ID == 3);
```

Формы лямбда-выражений

Лямбда-выражения являются весьма эффективными конструкциями, и чем глубже вы станете вникать в программирование на C#, тем чаще они будут встречаться. Формы самих выражений могут немного отличаться друг от друга. Изначально лямбда-выражения были частью математической формы записи под названием «лямбда-исчисление», которая предназначалась для описания функций. (Функции можно рассматривать как методы, возвращающие значения.) Хотя в имеющейся в C# реализации лямбда-выражений используются расширенный синтаксис и семантика лямбда-исчисления, многие исходные принципы по-прежнему применимы. Далее приведен ряд примеров, показывающих различные формы лямбда-выражений, доступных в C#:

```
(ref int x, int y) => { x++; return x / y; }      // Несколько параметров
                                                       // с явно указанными типами.
                                                       // Параметр x передан по ссылке,
                                                       // поэтому эффект от операции ++
                                                       // не ограничивается выражением.
```

Чтобы подвести итог, перечислим ряд свойств лямбда-выражений, о которых вам следует знать.

- ❑ Если лямбда-выражение получает параметры, их нужно указать в круглых скобках слева от оператора `=>`. Типы параметров можно не указывать, и тогда компилятор C# выведет их типы из контекста самого лямбда-выражения. Если нужно, чтобы лямбда-выражение могло изменять значения параметров не только локально, параметры можно передавать по ссылке, используя ключевое слово `ref`, но делать это не рекомендуется.
- ❑ Лямбда-выражения могут возвращать значения, но их тип должен совпадать с типом соответствующего делегата.
- ❑ Тело лямбда-выражения может быть простым выражением или блоком кода C#, состоящим из нескольких инструкций, вызовов методов, определений переменных и других элементов кода.
- ❑ Переменные, определенные в методе лямбда-выражения, по завершении работы метода исчезают из области видимости.
- ❑ Лямбда-выражение может получать доступ ко всем переменным за его пределами, находящимися в той области видимости, в которой определено это лямбда-выражение, и вносить в них изменения. Но пользоваться этим свойством нужно весьма осмотрительно!

ЛЯМБДА-ВЫРАЖЕНИЯ И БЕЗЫМЯННЫЕ МЕТОДЫ

Лямбда-выражения были добавлены к языку C# в версии 3.0. А в C# версии 2.0 были введены безымянные методы, способные выполнять сходные задачи, но не обладающие такой же гибкостью. Безымянные методы были введены главным образом для того, чтобы вы могли определять делегаты без необходимости создания именованных методов, просто указывая вместо имени метода определение его тела:

```
this.stopMachinery += delegate { folder.StopFolding(0); };
```

Безымянный метод также можно передать вместо делегата в качестве параметра:

```
control.Add(delegate { folder.StopFolding(0); } );
```

Обратите внимание на то, что при вставке безымянного метода перед ним нужно ставить ключевое слово `delegate`. Кроме того, все необходимые параметры указываются в круглых скобках, которые, как показано в следующем примере, следуют за ключевым словом `delegate`:

```
control.Add(delegate(int param1, string param2))
{ /* код, использующий param1 и param2 */ ... });
```

Лямбда-выражения позволяют применить более лаконичный и естественный синтаксис, чем безымянные методы, и они охватывают более совершенные аспекты C#, в чём вы сможете убедиться в следующих главах книги. Стало быть, в своем коде вам стоит отдавать предпочтение не безымянным методам, а лямбда-выражениям.

Сравнение массивов и коллекций

Далее перечислены важные отличия массивов от коллекций.

- ❑ Экземпляр массива имеет фиксированный размер и не может увеличиваться или уменьшаться. Коллекция может динамически изменяться в размере по мере надобности.
- ❑ У массива может быть более одной размерности. Коллекция носит линейный характер. Но элементы коллекции могут сами быть коллекциями, поэтому многомерный массив можно имитировать в виде коллекции коллекций.
- ❑ Элементы в массиве сохраняются и извлекаются с помощью индекса. Этот принцип поддерживается лишь некоторыми коллекциями. Например, для сохранения элемента в коллекции `List<T>` используется метод `Add` или `Insert`, а для извлечения элемента — метод `Find`.
- ❑ Многие классы коллекций для создания и заполнения массива, содержащего элементы коллекции, предоставляют метод `ToArrayList`. Элементы копируются в массив без удаления их из коллекции. Кроме того, эти коллекции представляют конструкторы, способные заполнять коллекцию непосредственно из массива.

Применение классов коллекций к игральным картам

В следующем упражнении карточная игра, разработанная в главе 10, будет переделана под использование не массивов, а коллекций.

Использование коллекций для реализации карточной игры

Откройте в среде Microsoft Visual Studio 2015 проект `Cards`, который находится в папке `\Microsoft Press\VCSBS\Chapter 18\Cards` вашей папки документов.

Этот проект содержит измененную версию проекта из главы 10, занимавшегося раздачей карт с помощью массивов. Класс `PlayingCard` изменен таким образом, чтобы демонстрировать достоинство и масть карты в качестве свойств, предназначенных только для чтения.

Выполните в окно редактора файл `Pack.cs`. Добавьте к началу файла следующую директиву `using`:

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

Внесите в класс `Pack` изменения, показанные жирным шрифтом, заменив определение двумерного массива `cardPack` на объект `Dictionary<Suit, List<PlayingCard>>`:

```
class Pack
{
    ...
    private Dictionary<Suit, List<PlayingCard>> cardPack;
    ...
}
```

В исходном приложении двумерный массив использовался для представления карточной колоды. В этом коде массив заменен словарем (`Dictionary`), где ключ указывает на масть, а значение является перечнем карт в этой масти.

Найдите конструктор `Pack`. Внесите в первую инструкцию конструктора изменения, выделенные жирным шрифтом, позволяющие создавать в ней экземпляр переменной `cardPack` в виде новой `Dictionary`-коллекции, а не массива:

```
public Pack()
{
    this.cardPack = new Dictionary<Suit, List<PlayingCard>>(NumSuits);
    ...
}
```

Несмотря на то что `Dictionary`-коллекция будет изменять свой размер автоматически по мере добавления элементов, если вероятность изменения размера коллекции мала, то при создании ее экземпляра можно указать начальный размер. Это упростит выделение памяти (но при этом, если размер окажется недостаточным, коллекция по-прежнему сможет его увеличить). В данном случае `Dictionary`-коллекция будет содержать коллекцию из четырех списков (по одному для каждой масти), следовательно, будет выделено пространство для четырех элементов (`NumSuits` является константой со значением 4).

Объявите во внешнем цикле `for` объект коллекции `List<PlayingCard>` по имени `cardsInSuit`, достаточно большой, чтобы содержать количество карт, которое имеется в каждой масти, используя для этого константу `CardsPerSuit`. Соответствующий код выделен жирным шрифтом:

```
public Pack()
{
    this.cardPack = new Dictionary<Suit, List<PlayingCard>>(NumSuits);
    for (Suit = Suit.Clubs; suit <= Suit.Spades; suit++)
    {
        List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
        for (Value value = Value.Two; value <= Value.Ace; value++)
        {
            ...
        }
    }
}
```

Внесите изменения, показанные жирным шрифтом, во внутренний цикл `for`, добавив к этой коллекции не массив, а новые объекты `PlayingCard`:

```
for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
{
    List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        cardsInSuit.Add(new PlayingCard(suit, value));
    }
}
```

Добавьте к `Dictionary`-коллекции `cardPack` после внутреннего цикла `for` объект типа `List`, указав в качестве ключа к этому элементу переменную `suit`:

```
for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
{
    List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        cardsInSuit.Add(new PlayingCard(suit, value));
    }
    this.cardPack.Add(suit, cardsInSuit);
}
```

Найдите метод `DealCardFromPack`. Этот метод выбирает произвольную карту из колоды, удаляет ее и возвращает вызывавшему метод коду. Логика выбора карты не требует никаких изменений, но инструкция в конце метода, извлекающая карту из массива, должна быть изменена для использования вместо массива `Dictionary`-коллекции. Кроме того, нужно изменить код, удаляющий карту (которая уже была сдана) из массива: эту карту нужно найти в списке, а затем удалить оттуда. Для поиска карты воспользуйтесь методом `Find` и укажите предикат, с помощью которого можно будет отыскать карту с совпадающим значением. Параметром предиката должен быть объект типа `PlayingCard` (в списке содержатся элементы типа `PlayingCard`).

Как показано жирным шрифтом в следующем примере, измененные инструкции находятся после закрывающей фигурной скобки второго цикла `while`:

```

public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
    while (this.IsSuitEmpty(suit))
    {
        suit = (Suit)randomCardSelector.Next(NumSuits);
    }

    Value value = (Value)randomCardSelector.Next(CardsPerSuit);
    while (this.IsCardAlreadyDealt(suit, value))
    {
        value = (Value)randomCardSelector.Next(CardsPerSuit);
    }

    List<PlayingCard> cardsInSuit = this.cardPack[suit];
    PlayingCard card = cardsInSuit.Find(c => c.CardValue == value);
    cardsInSuit.Remove(card);
    return card;
}

```

Найдите метод `IsCardAlreadyDealt`. Он определяет, была ли карта уже сдана, проверяя для этого, установлено ли для соответствующего элемента в массиве значение `null`. Вам нужно изменить этот метод, чтобы он определял, имеется ли в списке соответствующей масти в `Dictionary`-коллекции `cardPack` карта указанного достоинства.

Чтобы определить, есть ли элемент в `List<T>`-коллекции, воспользуйтесь методом `Exists`. Этот метод похож на метод `Find` тем, что он получает в качестве аргумента предикат. Этому предикату поочередно передается каждый элемент коллекции, и он должен возвратить `true`, если элемент соответствует указанному критерию, и `false`, если не соответствует. В данном случае коллекция типа `List<T>` содержит `PlayingCard`-объекты, и критерий для предиката метода `Exists` должен быть выбран таким, чтобы возвращалось значение `true`, если ему передан элемент `PlayingCard` с мастью и достоинством, которые соответствуют параметрам, переданным методу `IsCardAlreadyDealt`. Внесите в метод изменения, показанные в следующем примере жирным шрифтом:

```

private bool IsCardAlreadyDealt(Suit suit, Value value)
{
    List<PlayingCard> cardsInSuit = this.cardPack[suit];
    return (!cardsInSuit.Exists(c => c.CardSuit == suit && c.CardValue == value));
}

```

Выполните в окне редактора файл `Hand.cs`. Добавьте к списку в самом начале файла следующую директиву `using`:

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

В данный момент класс `Hand` для хранения разданных игральных карт использует массив по имени `cards`. Измените, как показано далее жирным

шрифтом, определение переменной `cards`, чтобы она стала `List<PlayingCard>`-коллекцией:

```
class Hand
{
    public const int HandSize = 13;
    private List<PlayingCard> cards = new List<PlayingCard>(HandSize);
    ...
}
```

Найдите метод `AddCardToHand`. Сейчас он занимается проверкой окончания раздачи карт: если карты еще не розданы, добавляет карту, предоставляемую в качестве параметра, в массив `cards` по индексу, указанному переменной `playingCardCount`. Измените этот метод, чтобы в нем вместо этого использовался метод `Add` из `List<PlayingCard>`-коллекции.

В результате изменения исчезнет также надобность в явном отслеживании количества карт, содержащихся в коллекции, поскольку вместо этого вы можете воспользоваться присущим коллекции карт свойством `Count`. Следовательно, вам нужно удалить из класса переменную `playingCardCount` и изменить инструкцию `if`, проверяющую заполнение раздачи, воспользовавшись ссылкой на свойство `Count`, принадлежащее коллекции карт.

Метод должен приобрести следующий окончательный вид, где все изменения выделены жирным шрифтом:

```
public void AddCardToHand(PlayingCard cardDealt)
{
    if (this.cards.Count >= HandSize)
    {
        throw new ArgumentException("Too many cards");
    }
    this.cards.Add(cardDealt);
}
```

Щелкните в меню Отладка на пункте Начать отладку, чтобы инициировать сборку и запуск приложения.

После появления формы Card Game щелкните на кнопке Deal.



ПРИМЕЧАНИЕ Кнопка Deal расположена на панели команд. Для ее обнаружения может понадобиться раскрыть эту панель.

Убедитесь в том, что карты розданы на руки и эта раздача выглядит точно так же, как и прежде. Еще раз щелкните на кнопке Deal, чтобы создать еще один произвольный набор розданных на руки карт.

Приложение в процессе работы показано на рис. 18.1.

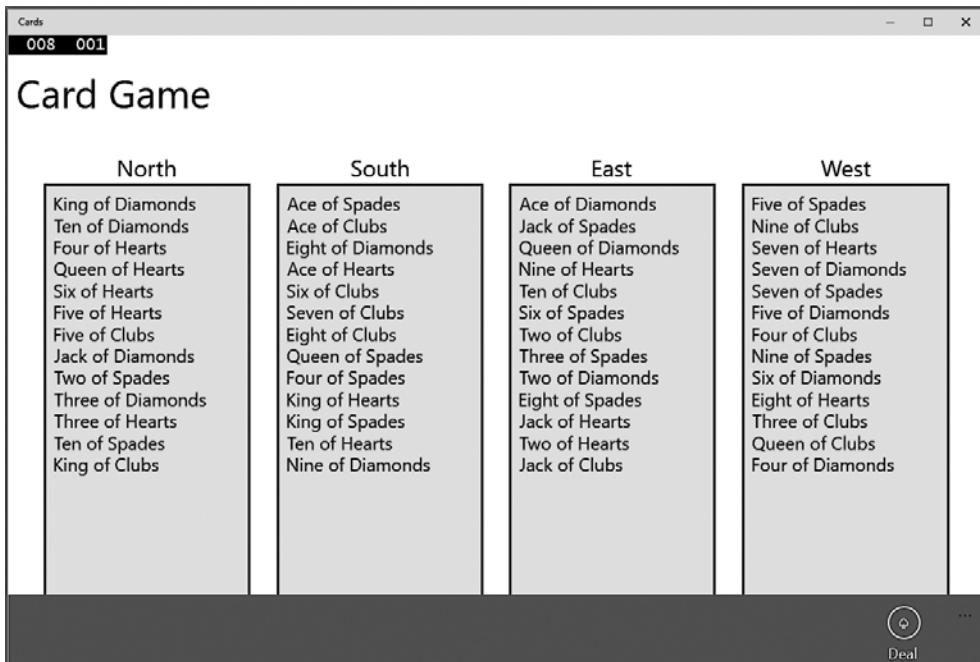


Рис. 18.1

Вернитесь в среду Visual Studio 2015 и остановите отладку.

Выводы

В этой главе вы научились использовать некоторые самые распространенные классы коллекций, применяемые для хранения данных и обеспечения доступа к ним. В частности, вы изучили порядок использования классов коллекций-обобщений для создания коллекций, безопасных в отношении применяемых типов. Вы также научились создавать лямбда-выражения для поиска в коллекциях указанных элементов.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 19 «Перечисляемые коллекции».

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Создать новую коллекцию	Воспользуйтесь конструктором для класса коллекции, например: <pre>List<PlayingCard> cards = new List<PlayingCard>();</pre>
Добавить к коллекции еще один элемент	Для списков, хэш-наборов и словарно-ориентированных коллекций воспользуйтесь (в зависимости от обстоятельств) методами Add или Insert. Для Queue<T>-коллекций воспользуйтесь методом Enqueue, а для Stack<T>-коллекций — методом Push, например: <pre>HashSet<string> employees = new HashSet<string>(); employees.Add("John"); ... LinkedList<int> data = new LinkedList<int>(); data.AddFirst(101); ... Stack<int> numbers = new Stack<int>(); numbers.Push(99);</pre>
Удалить элемент из коллекции	Для списков, хэш-наборов и словарно-ориентированных коллекций воспользуйтесь методом Remove. Для Queue<T>-коллекций воспользуйтесь методом Dequeue, а для Stack<T>-коллекций — методом Pop, например: <pre>HashSet<string> employees = new HashSet<string>(); employees.Remove("John"); ... LinkedList<int> data = new LinkedList<int>(); data.Remove(101); ... Stack<int> numbers = new Stack<int>(); ... int item = numbers.Pop();</pre>
Определить количество элементов коллекции	Воспользуйтесь свойством Count, например: <pre>List<PlayingCard> cards = new List<PlayingCard>(); ... int noOfCards = cards.Count;</pre>
Найти в коллекции указанный элемент	Для словарно-ориентированных коллекций воспользуйтесь системой записи, присущей массивам. Для списков воспользуйтесь методом Find, например: <pre>Dictionary<string, int> ages = new Dictionary<string, int>(); ages.Add("John", 47); int johnsAge = ages["John"]; ... List<Person> personnel = new List<Person>(); Person match = personnel.Find(p => p.ID == 3);</pre>

Чтобы	Сделайте следующее
	<p>Примечание: в классах коллекций Stack<T>, Queue<T> и хэш-наборов поиск не поддерживается, хотя в хэш-наборе вы можете протестировать наличие того или иного элемента, воспользовавшись для этого методом Contains</p>
Совершить последовательный обход всех элементов коллекции	<p>Воспользуйтесь инструкцией for или foreach, например:</p> <pre>LinkedList<int> numbers = new LinkedList<int>(); ... for (LinkedListNode<int> node = numbers.First; node != null; node = node.Next) { int number = node.Value; Console.WriteLine(number); } ... foreach (int number in numbers) { Console.WriteLine(number); }</pre>

19 Перечисляемые коллекции

Прочитав эту главу, вы научитесь:

- самостоятельно определять нумератор с возможностью его использования для последовательного обхода элементов коллекции;
- создавать нумератор в автоматическом режиме путем создания итератора;
- предоставлять дополнительные итераторы для пошагового обхода элементов коллекции в различных последовательностях.

В главе 10 «Использование массивов» и в главе 18 «Использование коллекций» было показано, как работать с массивами и классами коллекций для хранения последовательностей или наборов данных. В главе 10 также подробно рассмотрена инструкция `foreach`, которую можно использовать для пошагового обхода или перебора элементов в коллекции. В этих главах инструкция `foreach` использовалась в качестве быстрого и удобного способа доступа к содержимому массива или коллекции, но теперь настало время получить больше сведений о том, как работает эта инструкция. Эта тема приобретает важность при определении ваших собственных классов коллекций, и в данной главе рассказывается, как сделать коллекции перечисляемыми.

Перечисление элементов коллекции

В главе 10 представлен пример использования инструкции `foreach` для перечисления элементов в простом массиве. Код выглядит следующим образом:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
    Console.WriteLine(pin);
}
```

Конструктор `foreach` предоставляет весьма изящный механизм, существенно упрощающий создаваемый код, но он может быть применен только при определенных обстоятельствах: им можно воспользоваться только для пошагового обхода перечисляемой коллекции.

Но что на самом деле представляет собой перечисляемая коллекция? Если давать краткое определение, то это коллекция, реализующая интерфейс `System.Collections.IEnumerable`.



ПРИМЕЧАНИЕ Следует помнить, что все массивы в C# фактически являются экземплярами класса `System.Array`. Этот класс является классом коллекции, реализующим интерфейс `IEnumerable`.

В интерфейсе `IEnumerable` содержится всего один метод `GetEnumerator`:

```
IEnumerator GetEnumerator();
```

Этот метод должен возвратить объект-нумератор, реализующий интерфейс `System.Collections.IEnumerator`. Объект-нумератор используется для пошагового обхода (перечисления) элементов коллекции. Интерфейс `IEnumerator` определяет следующие свойства и методы:

```
object Current { get; }  
bool MoveNext();  
void Reset();
```

Нумератор следует представлять в виде указателя на элементы списка. В исходном состоянии указатель стоит перед первым элементом. Для перемещения указателя на следующий (первый) элемент списка нужно вызвать метод `MoveNext`, который должен возвратить `true`, если следующий элемент имеется в списке, и `false`, если такого элемента в нем нет. Для доступа к элементу, на котором стоит указатель, используется свойство `Current`, а для возвращения указателя назад и установки его перед первым элементом списка — метод `Reset`. Использованием для нумератора метода коллекции `GetEnumerator` выполняется многократный вызов метода `MoveNext`, а применяя нумератор для извлечения значения свойства `Current`, можно пошагово продвигаться вперед по элементам. Именно этим и занимается инструкция `foreach`. Следовательно, если нужно создать свой перечисляемый класс коллекции, вы должны реализовать в коллекции интерфейс `IEnumerable`, а также предоставить реализацию интерфейса `IEnumerator` для его возвращения методом `GetEnumerator` класса коллекции.



ВНИМАНИЕ На первый взгляд интерфейсы `IEnumerable` и `IEnumerator` имеют очень похожие имена, в которых несложно запутаться. Убедитесь в том, что вы их не перепутали.

Если присмотреться, можно заметить, что свойством `Current` интерфейса `IEnumerator` демонстрируется поведение, не обеспечивающее безопасности по отношению к типам, поскольку им возвращается объект, а не конкретный тип. Но вам, вероятно, интересно будет узнать, что библиотека классов Microsoft .NET Framework также предоставляет интерфейс-обобщение `IEnumerator<T>`, у которого имеется свойство `Current`, возвращающее вместо этого `T`. Также в этой библиотеке имеется интерфейс `IEnumerable<T>`, содержащий метод `GetEnumerator`, который возвращает объект `IEnumerator<T>`. Оба этих интерфейса определены в пространстве имен `System.Collections.Generic`, и если вы создаете приложение для .NET Framework версии 2.0 и выше, то при определении перечисляемых коллекций следует воспользоваться именно этими интерфейсами-обобщениями, а не версиями, не являющимися обобщениями.

Самостоятельная реализация нумератора

В следующем упражнении вами будет определен класс, реализующий интерфейс-обобщение `IEnumerator<T>` и создающий нумератор для класса двоичного дерева, который рассмотрен в главе 17 «Введение в обобщения».

В главе 17 было показано, как можно без особых трудностей выполнить обход двоичного дерева и вывести на экран его содержимое. Из-за этого может возникнуть мнение, что определение нумератора, извлекающего каждый элемент в двоичном дереве в том же порядке, станет довольно простой задачей. Как ни печально, но это не так. Основная проблема заключается в том, что при определении нумератора нужно помнить, что вы находитесь в структуре и поэтому последовательные вызовы метода `MoveNext` могут соответственно обновить позицию. Рекурсивные алгоритмы, подобные тому, что используется для обхода двоичного дерева, сами по себе не предоставляют простым и доступным образом сохраняемую информацию о состоянии между вызовами метода. По этой причине вы сначала проведете предварительную обработку данных в двоичном дереве, превращая его в более податливую структуру данных (а именно в очередь), а замен сделает эту структуру перечисляемой. Разумеется, эта хитрость будет скрыта от пользователя, совершающего обход элементов двоичного дерева!

Создание класса `TreeEnumerator`

Откройте в среде Microsoft Visual Studio 2015 решение `BinaryTree`, которое находится в папке `\Microsoft Press\VCSBS\Chapter 19\BinaryTree` вашей папки документов. В этом решении содержится рабочая копия проекта `BinaryTree`, созданного в главе 17. Вы добавите к этому проекту новый класс, в котором будет реализован нумератор для класса `BinaryTree`.

Щелкните в обозревателе решений на проекте `BinaryTree`. Щелкните в меню `Проект` на пункте `Добавить класс`, чтобы открыть диалоговое окно `Добавить новый`

элемент — `BinaryTree`. Выберите в средней панели шаблон Класс, наберите в поле Имя строку `TreeEnumerator.cs`, а затем щелкните на кнопке Добавить. Будет создан класс `TreeEnumerator`, создающий нумератор для `Tree<TItem>`-объекта. Чтобы гарантировать классу безопасность в отношении типов, следует предоставить параметр типа и реализовать `IEnumerator<T>`-интерфейс. Кроме того, параметр типа должен быть допустимым типом для `Tree<TItem>`-объекта, который класс будет превращать в перечисляемый, поэтому он должен быть принужден к реализации `IComparable<TItem>`-интерфейса (класс `BinaryTree` требует, чтобы для сортировки элементы дерева предоставляли средства, позволяющие сравнивать их друг с другом).

Измените в окне редактора, показывающего содержимое файла `TreeEnumerator.cs`, определение класса `TreeEnumerator`, чтобы оно отвечало этим требованиям. Соответствующие изменения показаны в следующем примере жирным шрифтом:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
}
```

Добавьте к классу `TreeEnumerator<TItem>` следующие три закрытые переменные, выделенные здесь жирным шрифтом:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
    private Tree<TItem> currentData = null;
    private TItem currentItem = default(TItem);
    private Queue<TItem> enumData = null;
}
```

Переменная `currentData` будет использоваться для хранения ссылки на дерево, превращаемое в перечисляемое, а переменная `currentItem` — для хранения значения, возвращаемого свойством `Current`. Очередь `enumData` будет заполняться значениями, извлекаемыми из узлов дерева, а метод `MoveNext` будет по порядку возвращать каждый элемент из этой очереди. Значение ключевого слова `default` объясняется в этой главе чуть позже во врезке «Инициализация переменной, определяемой с параметром типа».

Добавьте конструктор, который забирает в класс `TreeEnumerator<TItem>` `Tree<TItem>`-параметр по имени `data`. Добавьте в тело конструктора инструкцию, инициализирующую переменную `currentData` значением `data`:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
    ...
    public TreeEnumerator(Tree<TItem> data)
    {
        this.currentData = data;
    }
}
```

Добавьте к классу `TreeEnumerator<TItem>` сразу же после конструктора следующий закрытый метод `populate`:

```
class TreeEnumerator<TItem> : IEnumarator<TItem> where TItem : IComparable<TItem>
{
    ...
    private void populate(Queue<TItem> enumQueue, Tree<TItem> tree)
    {
        if (tree.LeftTree != null)
        {
            populate(enumQueue, tree.LeftTree);
        }

        enumQueue.Enqueue(tree.NodeData);

        if (tree.RightTree != null)
        {
            populate(enumQueue, tree.RightTree);
        }
    }
}
```

Этот метод совершаает обход двоичного дерева, добавляя содержащиеся в нем данные в очередь. В нем используется почти такой же алгоритм, как и тот, что использовался в методе `WalkTree` класса `Tree<TItem>`, описание которого приводилось в главе 17. Основное отличие заключается в том, что вместо добавления значений `NodeData` к строке метод сохраняет их в очереди.

Вернитесь к определению класса `TreeEnumerator<TItem>`. Поставьте в определении класса указатель мыши над текстом `IEnumarator<TItem>`. Щелкните в появившемся раскрывающемся контекстном меню (со значком горящей лампочки) на пункте Реализовать интерфейс явно. Это действие приведет к созданию заглушек для методов в интерфейсе `IEnumarator<TItem>` и интерфейсе `IEnumarator`, а также к добавлению его к концу класса. Будет также создан метод `Dispose` для интерфейса `IDisposable`.



ПРИМЕЧАНИЕ Интерфейс `IEnumarator<TItem>` наследуется из интерфейсов `IEnumarator` и `IDisposable`, из-за чего в нем появляются и их методы. Фактически единственным элементом, принадлежащим интерфейсу `IEnumarator<TItem>`, является свойство-обобщение `Current`. Методы `MoveNext` и `Reset` принадлежат необобщенному интерфейсу `IEnumarator`. А описание интерфейса `IDisposable` можно найти в главе 14 «Использование сборщика мусора и управление ресурсами».

Исследуйте созданный код. В телах свойств и методов содержится исходная реализация, которая просто выдает исключение `NotImplementedException`. На следующих этапах вы замените этот код реальной реализацией.

Обновите тело метода `MoveNext` кодом, выделенным здесь жирным шрифтом:

```
bool IEnumator.MoveNext()
{
    if (this.enumData == null)
    {
        this.enumData = new Queue<TItem>();
        populate(this.enumData, this.currentData);
    }

    if (this.enumData.Count > 0)
    {
        this.currentItem = this.enumData.Dequeue();
        return true;
    }

    return false;
}
```

Принадлежащий нумератору метод `MoveNext` выполняет двойную задачу. При первом вызове он должен инициализировать данные, используемые нумератором, и перейти к первой части данных, которые должны быть возвращены. (До того как метод `MoveNext` будет вызван в первый раз, значение, возвращаемое свойством `Current`, будет неопределенным, что должно привести к выдаче исключения.) В данном случае процесс инициализации состоит из создания экземпляра очереди с последующим вызовом метода `populate` для заполнения очереди данными, извлеченными из дерева.

Все последующие вызовы метода `MoveNext` должны просто приводить к перемещению по элементам данных, пока их уже не останется, с извлечением элементов из очереди, пока она, как в данном примере, не опустеет. Важно понимать, что фактически `MoveNext` элементы данных не возвращает, этим занимается свойство `Current`. Метод `MoveNext` лишь обновляет внутреннее состояние нумератора (то есть устанавливает для переменной `currentItem` значение элемента данных, извлеченного из очереди) с целью его использования свойством `Current`, возвращая при этом `true` при наличии следующего значения и `false` — при его отсутствии.

Внесите в определение метода доступа `get` свойства-обобщения `Current` следующие изменения, выделенные жирным шрифтом:

```
TItem IEnumator<TItem>.Current
{
    get
    {
        if (this.enumData == null)
        {
            throw new InvalidOperationException("Use MoveNext before calling
                                              Current");
        }

        return this.currentItem;
    }
}
```

ИНИЦИАЛИЗАЦИЯ ПЕРЕМЕННОЙ, ОПРЕДЕЛЯЕМОЙ С ПАРАМЕТРОМ ТИПА

Вы, наверное, заметили, что инструкция, определяющая и инициализирующая переменную `currentItem`, использует ключевое слово `default`. Переменная `currentItem` определяется с помощью параметра типа `TItem`. Когда программа написана и скомпилирована, реальный тип, который будет подставлен вместо `TItem`, может быть неизвестен — этот вопрос решается только в ходе выполнения кода. Это затрудняет указание на способ инициализации переменной. Возникает соблазн установить для нее значение `null`. Но если тип, подставляемый вместо `TItem`, является типом значений, такое присваивание будет недопустимым. (Для типов значений нельзя устанавливать значение `null`, это можно делать только для ссылочных типов.) Аналогично этому, если установить для этой переменной значение 0, предполагая при этом, что тип будет числовым, это значение будет недопустимо, если в реальности будет использоваться ссылочный тип. Но есть и другая возможность — вместо `TItem` может подставляться, к примеру, булев тип. Эта проблема решается с помощью ключевого слова `default`. Значение, используемое для инициализации переменной, будет определяться при выполнении инструкции. Если `TItem` становится ссылочным типом, то `default(TItem)` возвращает `null`, если `TItem` становится числовым типом, то `default(TItem)` возвращает 0, а если `TItem` становится булевым типом, то `default(TItem)` возвращает `false`. Если же `TItem` становится структурой, то отдельные поля в структуре инициализируются таким же образом. (Ссылочные поля устанавливаются в `null`, числовые поля — в 0, а булевые поля — в `false`.)



ВНИМАНИЕ Код следует добавить к соответствующей реализации свойства `Current`. А необобщенную версию `System.Collections.IEnumerator.Current` нужно оставить в исходном состоянии, выдающем исключение `NotImplementedException`.

Чтобы убедиться, что был вызван метод `MoveNext`, свойство `Current` исследует переменную `enumData`. (Эта переменная до первого вызова `MoveNext` будет иметь значение `null`.) Если вызова не было, свойство выдает исключение `InvalidOperationException`, что является общепринятым механизмом, используемым приложениями .NET Framework в качестве оповещения о том, что операция в текущем состоянии не может быть выполнена. Если предварительно был вызван метод `MoveNext`, то он уже обновил значение переменной `currentItem`, поэтому свойству `Current` остается лишь возвратить значение этой переменной.

Найдите метод `IDisposable.Dispose`. Закомментируйте строку `throw new NotImplementedException();`, выделенную в следующем примере кода жирным шрифтом. Нумератор не использует какие-либо ресурсы, требующие явного высвобождения, поэтому данному методу ничего не нужно делать. Но он все же должен присутствовать. Дополнительные сведения о методе `Dispose` можно найти в главе 14.

```
void IDisposable.Dispose()
{
    // throw new NotImplementedException();
}
```

Выполните сборку решения и исправьте ошибки, если о них будут выданы сообщения.

Реализация интерфейса `IEnumerable`

В следующем упражнении вам предстоит внести изменения в класс двоичного дерева, чтобы реализовать в нем интерфейс `IEnumerable<T>`. Объект `TreeEnumerator<TItem>` будет возвращаться методом `GetEnumerator`.

Реализация интерфейса `IEnumerable<TItem>` в классе `Tree<TItem>`

В обозревателе решений дважды щелкните на файле `Tree.cs`, чтобы в окне редактора отобразился класс `Tree<TItem>`. Внесите в определение класса `Tree<TItem>` изменения, выделенные в следующем примере кода жирным шрифтом, чтобы в нем реализовывался интерфейс `IEnumerable<TItem>`:

```
public class Tree<TItem> : IEnumerable<TItem> where TItem : IComparable<TItem>
```

Обратите внимание на то, что ограничения всегда помещаются в конце определения класса.

В определении класса наведите указатель мыши на интерфейс `IEnumerable<TItem>`. В раскрывающемся контекстном меню щелкните на пункте Реализовать интерфейс явно. В результате этого будет создана реализация методов `IEnumerable<TItem>.GetEnumerator` и `IEnumerable.GetEnumerator`, которая будет добавлена к классу. Не являющийся обобщением метод `IEnumerable` этого интерфейса реализуется за счет того, что интерфейс-обобщение `IEnumerable<TItem>` является наследником интерфейса `IEnumerable`.

Найдите метод-обобщение `IEnumerable<TItem>.GetEnumerator`, который находится в нижней части кода класса. Внесите изменения, показанные в следующем примере кода жирным шрифтом, в тело метода `GetEnumerator()`, заменив новой строкой существующую инструкцию `throw`:

```
IEnumerable<TItem> IEnumerable<TItem>.GetEnumerator()
{
    return new TreeEnumerator<TItem>(this);
}
```

Целью метода `GetEnumerator` является создание объекта-нумератора для сквозного обхода элементов коллекции. В данном случае вам нужно лишь создать новый `TreeEnumerator<TItem>`-объект, используя данные, находящиеся в дереве.

Выполните сборку решения и исправьте ошибки, если о них будут выданы сообщения.

Теперь вы протестируете измененный класс `Tree<TItem>`, используя для сквозного обхода элементов двоичного дерева и вывода на экран его содержимого инструкцию `foreach`.

Тестирование нумератора

В обозревателе решений щелкните правой кнопкой мыши на решении `BinaryTree`, укажите на пункт **Добавить**, а затем щелкните на пункте **Создать проект**. Добавьте новый проект, воспользовавшись шаблоном **Консольное приложение**. В качестве его имени укажите `EnumeratorTest`, в качестве места размещения — папку `\Microsoft Press\VCBS\Chapter 19\BinaryTree` своей папки документов, после чего щелкните на кнопке **OK**.



ПРИМЕЧАНИЕ Убедитесь в том, что в списке шаблонов Visual C# выбран именно шаблон **Консольное приложение**. В диалоговом окне **Добавить новый проект** изначально отображаются шаблоны для **Visual Basic** или **C++**.

В обозревателе решений щелкните правой кнопкой мыши на проекте `EnumeratorTest`, а затем щелкните на пункте **Назначить автозагружаемым проектом**.

Щелкните в меню **Проект** на пункте **Добавить ссылку**. Раскройте в левой панели диалогового окна **Менеджер ссылок** — `EnumeratorTest` узел **Проекты** и щелкните на пункте **Решение**. Установите флажок возле проекта `BinaryTree`, а затем щелкните на кнопке **OK**. В перечне ссылок для проекта `EnumeratorTest` в обозревателе решений появится сборка `BinaryTree`.

Выполните в окно редактора содержимое класса `Program` и добавьте к списку в начале файла следующую директиву `using`:

```
using BinaryTree;
```

Добавьте к методу `Main` инструкции, выделенные в следующем примере кода жирным шрифтом. Эти инструкции создают двоичное дерево и заполняют его целочисленными значениями:

```
static void Main(string[] args)
{
    Tree<int> tree1 = new Tree<int>(10);
    tree1.Insert(5);
    tree1.Insert(11);
    tree1.Insert(5);
```

```
tree1.Insert(-12);
tree1.Insert(15);
tree1.Insert(0);
tree1.Insert(14);
tree1.Insert(-8);
tree1.Insert(10);
}
```

Добавьте выделенную далее жирным шрифтом инструкцию `foreach`, которая займется перечислением содержимого дерева и выводом результатов на экран:

```
static void Main(string[] args)
{
    ...
    foreach (int item in tree1)
    {
        Console.WriteLine(item);
    }
}
```

Щелкните в меню Отладка на пункте Запуск без отладки. Программа запустится и выведет на экран значения в следующей последовательности (рис. 19.1):

-12, -8, 0, 5, 5, 10, 10, 11, 14, 15

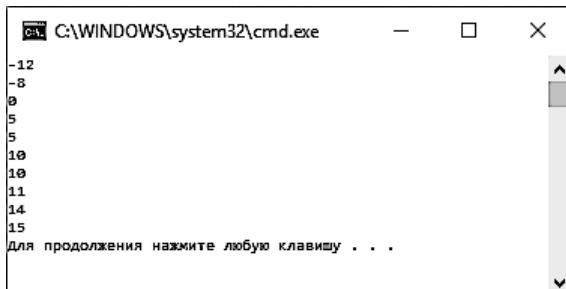


Рис. 19.1

Нажмите Ввод, чтобы вернуться в среду Visual Studio 2015.

Реализация нумератора с использованием итератора

Вполне очевидно, что процесс превращения коллекции в перечисляемую может усложниться и он не застрахован от ошибок. Чтобы упростить решение задачи, C# предоставляет итераторы, способные во многом автоматизировать ход этого процесса.

Итератор представляет собой блок кода, выдающий упорядоченную последовательность значений. Итератор не является компонентом перечисляемого класса, а указывает последовательность, которую нумератор должен использовать для возвращения своих значений. Иными словами, итератор — это просто описание последовательности перечисления, которой компилятор C# может воспользоваться для создания собственного нумератора. Усвоение этой концепции требует некоторого осмыслиения, поэтому давайте рассмотрим следующий простой пример.

Простой итератор

Принципы реализации итератора будут проиллюстрированы на примере класса `BasicCollection<T>`. Для хранения данных и предоставления метода `FillList` для заполнения списка в классе используется `List<T>`-объект. Обратите внимание также на то, что что в классе `BasicCollection<T>` реализуется интерфейс `IEnumerable<T>`. Метод `GetEnumerator` реализуется с помощью итератора:

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

class BasicCollection<T> : IEnumerable<T>
{
    private List<T> data = new List<T>();

    public void FillList(params T [] items)
    {
        foreach (var datum in items)
        {
            data.Add(datum);
        }
    }

    IEnumrator<T> IEnumerable<T>.GetEnumerator()
    {
        foreach (var datum in data)
        {
            yield return datum;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // В данном примере не реализован
        throw new NotImplementedException();
    }
}
```

При всей своей очевидной простоте метод `GetEnumerator` заслуживает более тщательного исследования. Первое, на что следует обратить внимание: он не

возвращает тип `IEnumerator<T>`. Вместо этого он осуществляет циклический обход элементов в массиве данных, возвращая по очереди каждый элемент. Основным моментом является использование ключевого слова `yield`. Оно указывает на значение, которое должно быть возвращено каждой итерацией. Если это вам поможет, то можете считать, что инструкция `yield` временно останавливает метод, возвращая значение вызвавшему его коду. Когда этому коду требуется следующее значение, метод `GetEnumerator` продолжает работу с того места, на котором она прервалась, осуществляет новый проход цикла, после чего выдает следующее значение. Со временем данные будут исчерпаны, цикл финиширует и метод `GetEnumerator` закончит свою работу. На этом итерация будет завершена.

Запомните, что в обычном понимании этот метод нельзя назвать нормальным. Код в методе `GetEnumerator` определяется как итератор. Компилятор использует этот код для создания реализации класса `IEnumerator<T>`, содержащего методы `Current` и `MoveNext`. Эта реализация в точности соответствует тем функциональным возможностям, которые указаны методом `GetEnumerator`. Но увидеть этот сгенерированный код вам, скорее всего, не удастся (если только не декомпилировать сборку, содержащую скомпилированный код), однако это не такая уж высокая плата за удобство и сокращение объема кода, который вам необходимо создать. Нумератор, созданный итератором, можно вызвать обычным образом, что и показано в следующем блоке кода, который выводит на экран слова первой строки стихотворения Льюиса Кэрролла «Бармаглот»:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
foreach (string word in bc)
{
    Console.WriteLine(word);
}
```

Этот код просто выводит содержимое объекта `bc` в следующем порядке:

```
Twas, brillig, and, the, slithy, toves
```

Если для представления данных в другой последовательности нужен альтернативный механизм итерации, можно создать дополнительные свойства, в которых реализуется интерфейс `IEnumerable`, и использовать для возвращения данных итератор. Например, показанное здесь свойство `Reverse` класса `BasicCollection<T>` выдает данные, находящиеся в списке, в обратном порядке:

```
class BasicCollection<T> : IEnumerable<T>
{
    ...
    public IEnumerable<T> Reverse
    {
```

```
get
{
    for (int i = data.Count - 1; i >= 0; i--)
    {
        yield return data[i];
    }
}
```

Это свойство можно вызвать следующим образом:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
foreach (string word in bc.Reverse)
{
    Console.WriteLine(word);
}
```

Этот код выводит на экран содержимое объекта `bc` в обратном порядке:

```
toves, slithy, the, and, brillig, Twas
```

Определение нумератора для класса `Tree<TItem>` путем использования итератора

В следующем упражнении нумератор для класса `Tree<TItem>` будет реализован с использованием итератора. В отличие от предыдущего набора упражнений, требовавшего предварительной обработки данных, находящихся в дереве, и помещения их в очередь с использованием метода `MoveNext`, здесь вы можете определить итератор, обеспечивающий последовательный обход элементов дерева, путем использования более естественного рекурсивного механизма, похожего на тот, что использовался в методе `WalkTree`, рассмотренном в главе 17.

Добавление нумератора к классу `Tree<TItem>`

Откройте в среде Visual Studio 2015 решение `BinaryTree`, которое находится в папке `\Microsoft Press\VCSBS\Chapter 19\IteratorBinaryTree` вашей папки документов. Это решение содержит еще одну копию проекта `BinaryTree`, созданного в главе 17.

Откройте в окне редактора файл `Tree.cs`. Внесите в определение класса `Tree<TItem>` следующее изменение, выделенное здесь жирным шрифтом, показывающее, что в нем реализуется интерфейс `IEnumerable<TItem>`:

```
public class Tree<TItem> : IEnumerable<TItem> where TItem : IComparable<TItem>
{
    ...
}
```

Наведите указатель мыши на обозначенный в определении класса интерфейс `IEnumerable<TItem>`. Щелкните в появившемся контекстном меню на пункте Реализовать интерфейс явно, чтобы добавить к концу класса методы `IEnumerable<TItem>.GetEnumerator` и `IEnumerable.GetEnumerator`.

Найдите метод-обобщение `IEnumerable<TItem>.GetEnumerator`. Замените содержимое метода `GetEnumerator` кодом, выделенным здесь жирным шрифтом:

```
IEnumerator<TItem> IEnumerable<TItem>.GetEnumerator()
{
    if (this.LeftTree != null)
    {
        foreach (TItem item in this.LeftTree)
        {
            yield return item;
        }
    }

    yield return this.NodeData;

    if (this.RightTree != null)
    {
        foreach (TItem item in this.RightTree)
        {
            yield return item;
        }
    }
}
```

На первый взгляд, может быть, это и не будет очевидным, но этот код следует такому же рекурсивному алгоритму, который использовался в главе 17 для обхода содержимого двоичного дерева. Если значение у `LeftTree` не пустое, первая инструкция `foreach` неявным образом вызывает в отношении себя же метод `GetEnumerator`, который именно сейчас и определяется. Этот процесс продолжается до тех пор, пока не будет найден узел без левого поддерева. В этот момент свойство `NodeData` уже возвращено, и таким же образом выполняется исследование правого поддерева. Когда данные в правом поддереве будут исчерпаны, процесс отматывается до родительского узла, возвращая родительское свойство `NodeData` и исследуя правое поддерево родительского узла. Такой порядок действий соблюдается до тех пор, пока процесс нумерации не охватит все дерево и не будут возвращены все узлы.

Тестирование нового нумератора

В обозревателе решений щелкните правой кнопкой мыши на решении `BinaryTree`, укажите на пункт Добавить и щелкните на пункте Существующий проект. В диалоговом окне Добавить существующий проект перейдите в папку `\Microsoft Press\VCBS\Chapter 19\BinaryTree\EnumeratorTest`, выберите файл проекта `EnumeratorTest`,

а затем щелкните на кнопке Открыть. Это проект, созданный для тестирования нумератора, который ранее в этой главе вы разработали собственными силами.

В обозревателе решений щелкните правой кнопкой мыши на проекте `EnumeratorTest`, а затем щелкните на пункте Назначить автозагружаемым проектом. Раскройте в обозревателе решений узел Ссылки, относящийся к проекту `EnumeratorTest`, щелкните правой кнопкой мыши на ссылке `BinaryTree`, после чего щелкните на пункте Удалить.

В меню Проект щелкните на пункте Добавить ссылку. Раскройте в левой панели диалогового окна Менеджер ссылок — `EnumeratorTest` узел Проекты и щелкните на пункте Решение. Установите в средней панели флажок возле проекта `BinaryTree`, а затем щелкните на кнопке ОК.



ПРИМЕЧАНИЕ Эти два действия гарантируют вам, что проект `EnumeratorTest` ссылается на нужную версию сборки `BinaryTree`. Здесь должна использоваться сборка, реализующая нумератор путем использования итератора, а не та версия, которая была создана в предыдущем наборе упражнений данной главы.

Выполните в окне редактора файл `Program.cs` проекта `EnumeratorTest`. Посмотрите в этом файле на метод `Main`. Вы должны вспомнить, что при тестировании ранее созданного нумератора этот метод создавал экземпляр `Tree<int>`-объекта, наполнял его данными и затем, используя инструкцию `foreach`, выводил на экран его содержимое. Выполните сборку решения и исправьте ошибки, если о них будут выданы сообщения. Щелкните в меню Отладка на пункте Запуск без отладки. Программа запустится и выведет значения в том же порядке, что и раньше:

```
-12, -8, 0, 5, 5, 10, 10, 11, 14, 15
```

Нажмите Ввод и вернитесь в среду Visual Studio 2015.

Выходы

В этой главе вы узнали, как для того, чтобы приложения могли выполнять сквозной обход элементов коллекции, в классе этой коллекции реализуются интерфейсы `IEnumerable<T>` и `IEnumerator<T>`. Вы также увидели, как нумератор реализуется с помощью итератора.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 20 «Отделение логики приложения и обработка событий».

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню Файл щелкните на пункте Выход. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке Да и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Превратить класс коллекции в перечисляемый, позволяя ему поддерживать конструкцию foreach	<p>Реализуйте интерфейс <code>IEnumerable</code> и предоставьте метод <code>GetEnumerator</code>, возвращающий объект <code>IEnumerator</code>, например:</p> <pre>public class Tree<TItem> : IEnumerable<TItem> { ... IEnumerator<TItem> GetEnumerator() { ... } }</pre>
Реализовать нумератор без использования итератора	<p>Определите класс нумератора, реализующий интерфейс <code>IEnumerator</code>, а также предоставляющий свойство <code>Current</code> и метод <code>MoveNext</code> (и, возможно, дополнительный метод <code>Reset</code>), например:</p> <pre>public class TreeEnumerator<TItem> : IEnumerator<TItem> { ... TItem Current { get { ... } } bool MoveNext() { ... } }</pre>
Определить нумератор путем использования итератора	<p>Реализуйте нумератор для указания того, какие элементы должны быть возвращены (воспользовавшись инструкцией <code>yield</code>) и в каком порядке, например:</p> <pre>IEnumerator<TItem> GetEnumerator() { for (...) { yield return ... } }</pre>

20 Отделение логики приложения и обработка событий

Прочитав эту главу, вы научитесь:

- объявлять тип делегата для создания абстракции сигнатуры метода;
- создавать экземпляр делегата для ссылки на конкретный метод;
- вызывать метод через делегата;
- определять лямбда-выражение для конкретного указания кода, выполняемого делегатом;
- объявлять поле события;
- обрабатывать событие с помощью делегата;
- инициировать событие.

Во многих примерах и упражнениях данной книги делался акцент на тщательное определение классов и структур с целью выполнения инкапсуляции. Благодаря этому реализацию методов в таких типах можно изменять без излишнего воздействия на использующие их приложения. Но иногда инкапсулировать все функциональные средства типа невозможно или нежелательно. Например, логика в методе класса может зависеть от того, каким компонентом или приложением вызывается этот метод, для чего в качестве составной части работы метода может потребоваться выполнить специфичную для приложения или компонента обработку. Но при создании такого класса и реализации его методов вы можете не знать, какие приложения или компоненты собираются его использовать, и вам нужно обойтись без предоставления зависимостей в вашем коде, которые могут накладывать ограничения на использование класса. Идеальным решением этой проблемы являются делегаты, позволяющие полностью отделить логику приложения в ваших методах от вызывающих их приложений.

События в C# придерживаются примерно такого же сценария. В коде, который используется в данной книге для упражнений, зачастую предполагается последовательное выполнение инструкций. Хотя в большинстве случаев все так и происходит, иногда требуется прервать текущий ход выполнения программы для решения другой, более важной задачи. Когда выполнение этой задачи завершится, программа может продолжить выполнение с того места, где оно было прервано. Классическими примерами этого стиля программирования являются формы универсальной платформы Windows (UWP), которые вы использовали в упражнениях по разработке графических приложений. Форма показывает на экране такие элементы управления, как кнопки и текстовые поля. При щелчке на кнопке или наборе текста в текстовом поле вы ожидаете от формы немедленной реакции. Приложению приходится приостанавливать свою работу и обрабатывать введенные вами данные.

Действия такого рода применяются не только к графическим пользовательским интерфейсам, но и к любым приложениям, где операции должны выполняться безотлагательно (такова, например, операция по заглушке реактора на атомной электростанции при чрезмерном повышении температуры). Чтобы справиться с подобной обработкой, системе выполнения нужно предоставить две вещи: средства, обозначающие, что случилось что-то неотложное, и способ указания кода, который должен быть запущен при возникновении неотложного события. Инфраструктуру, с которой можно реализовать системы, применяющие этот подход, предоставляют события в связке с делегатами.

Сначала давайте рассмотрим работу делегатов.

Основные сведения о делегатах

Делегат является ссылкой на метод. Это весьма простое, но чрезвычайно эффективное средство. Позвольте пояснить.



ПРИМЕЧАНИЕ Своё название делегаты получили потому, что при вызове они делегируют обработку тому методу, на который ссылаются.

Обычно при написании инструкции, вызывающей метод, указывается имя метода (и, возможно, объект или структура, которой этот метод принадлежит). По коду можно абсолютно точно понять, какой метод запускается и где именно вы его запускаете. Рассмотрим следующий простой пример, в котором вызывается метод `performCalculation` объекта, относящегося к типу `Processor` (сейчас совершенно неважно, чем этот метод занимается или как определен класс `Processor`):

```
Processor p = new Processor();
p.performCalculation();
```

Делегат — это объект, ссылающийся на метод. Ссылку на метод можно присвоить делегату практически так же, как можно присвоить `int`-значение `int`-переменной. В следующем примере создается делегат по имени `performCalculationDelegate`, ссылающийся на метод `performCalculation`, принадлежащий объекту типа `Processor`. Я умышленно не стал показывать некоторые элементы инструкции, объявляющие делегат, поскольку сейчас важнее понять саму концепцию, а не разбираться в синтаксисе (вскоре будет показан и полный синтаксис):

```
Processor p = new Processor();
delegate ... performCalculationDelegate ...;
performCalculationDelegate = p.performCalculation;
```

Следует усвоить, что инструкция, присваивающая делегату ссылку на метод, не запускает метод именно в этот момент, а после имени метода не ставятся круглые скобки и не указываются какие-либо параметры (если метод их принимает). Это всего лишь инструкция присваивания.

После сохранения в делегате ссылки на метод `performCalculation`, принадлежащий `Processor`-объекту, приложение может в дальнейшем вызывать метод через делегат:

```
performCalculationDelegate();
```

Это похоже на обычный вызов метода, и не зная, в чем именно дело, можно подумать, что запускается метод `performCalculationDelegate`. Но общезыковая среда выполнения (common language runtime (CLR)) знает, что это делегат, поэтому извлекает метод, на который он ссылается, и запускает этот метод. Позже вы можете вносить изменения в метод, на который ссылается делегат, следовательно, инструкция,зывающая делегат, может фактически при каждом выполнении вызывать разные методы. Кроме того, делегат может одновременно ссылаться более чем на один метод (что можно представить в виде коллекции ссылок на методы), и при вызове делегата будут запущены все методы, на которые он ссылается.



ПРИМЕЧАНИЕ Те, кто знаком с языком C++, могут считать делегат аналогом указателя на функцию. Но в отличие от указателей на функции, делегаты обладают полной безопасностью относительно используемых типов. Делегат можно заставить ссылаться только на тот метод, который соответствует сигнатуре делегата, и вы не можете вызвать делегат, который не ссылается на подходящий метод.

Примеры делегатов в библиотеке классов .NET Framework

Библиотека классов Microsoft .NET Framework широко использует делегаты для многих своих типов, примерами могут послужить рассмотренные в главе 18 «Использование коллекций» методы `Find` и `Exists` класса `List<T>`. Если помните, эти методы выполняют сквозной поиск в коллекции `List<T>`, либо возвращая

соответствующий элемент, либо проверяя наличие этого элемента. При реализации класса `List<T>` его разработчики не имели ни малейшего представления о том, что в коде вашего приложения должно подтверждаться соответствие, поэтому они позволили вам определить это путем предоставления своего собственного кода в форме предиката. Фактически предикат является делегатом, возвращающим булево значение.

Напомнить порядок использования метода `Find` поможет следующий код:

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
...
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 47 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};
...
// Поиск элемента списка, у которого ID равен 3
Person match = personnel.Find(p => p.ID == 3);
```

Другими примерами методов, использующих делегаты для выполнения своих операций в классе `List<T>`, могут послужить `Average`, `Max`, `Min`, `Count` и `Sum`. Эти методы получают в качестве параметра делегат `Func`. Он ссылается на метод, возвращающий значение (то есть на функцию). В следующих примерах метод `Average` используется для вычисления среднего возраста элементов в коллекции `personnel` (делегат `Func<T>` просто возвращает значение, находящееся в поле `Age` каждого элемента коллекции), метод `Max` используется для определения элемента с самым высоким `ID`, а метод `Count` вычисляет, сколько элементов имеют значение `Age` между 30 и 39 включительно:

```
double averageAge = personnel.Average(p => p.Age);
Console.WriteLine($"Average age is {averageAge}");
...
int id = personnel.Max(p => p.ID);
Console.WriteLine($"Person with highest ID is {id}");
...
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
Console.WriteLine($"Number of personnel in their thirties is {thirties}");
```

Этот код выводит на экран следующую информацию:

```
Average age is 32.75
Person with highest ID is 4
Number of personnel in their thirties is 1
```

В книге часто будут встречаться примеры этих и других типов делегатов, используемых библиотекой классов .NET Framework. Можно определять и собственные делегаты. Досконально разобраться в том, как и когда они могут понадобиться, лучше всего путем наблюдения их в действии, поэтому давайте проработаем ряд примеров.

ТИПЫ ДЕЛЕГАТОВ FUNC<T, ...> И ACTION<T, ...>

Параметр, получаемый Average, Max, Count и другими методами класса List<T>, по сути является делегатом-обобщением Func<T, TResult>; параметры типа ссылается на тип параметра, переданного делегату, и на тип возвращаемого значения. Для методов Average, Max и класса List<Person>, показанного в тексте, первым параметром T является тип данных в списке (структура Person), а параметр типа TResult определяется контекстом, в котором используется делегат. В следующем примере типом TResult является int, потому что значение, возвращаемое методом Count, должно быть целым числом:

```
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
```

Следовательно, в этом примере метод Count ожидает, что типом делегата будет Func<Person, int>.

Все это можно назвать теорией, потому что компилятор автоматически генерирует делегата на основе типа List<T>, но вас все же стоило ознакомить с этой особенностью, поскольку такое происходит в библиотеке классов .NET Framework практически повсеместно. Фактически в пространстве имен System определяется целое семейство типов делегатов Func, от Func<TResult> для функций, возвращающих результат, не получая каких-либо параметров, до Func<T1, T2, T3, T4, ..., T16, TResult> для функций, получающих 16 параметров. Когда вам придется создавать собственный тип делегата, соответствующий этой схеме, нужно будет присмотреться к использованию соответствующего типа делегата Func. Очередная встреча с типами делегатов Func вам предстоит в главе 21 «Запрос данных, находящихся в памяти, с помощью выражений запросов».

В дополнение к Func в пространстве имен System определяется также серия типов делегатов Action. Делегаты типа Action используются для ссылок на метод, выполняющий действие, а не возвращающий значение (методы типа void). Семейство типов делегатов Action доступно также в диапазоне от Action<T> (определяющем делегата, получающего всего один параметр) до Action<T1, T2, T3, T4, ..., T16>.

Сценарий автоматизированной фабрики

Предположим, что вы создаете систему управления для автоматизированной фабрики. Фабрика состоит из большого количества различных машин, каждая

из которых выполняет отдельную задачу в производстве конечной продукции, выпускаемой фабрикой: обработка и укладка металлических листов, сварка листов, их покраска и т. д. Каждая машина была собрана и установлена специалистом поставщика. Всеми машинами управляет компьютер, и каждый поставщик предоставил набор функций, который можно использовать для управления его машиной. Ваша задача заключается в объединении различных систем, используемых машинами, в единую программу управления. Одним из аспектов, на котором вы собрались сосредоточить свои усилия, является разработка средств выключения всех машин, причем быстрого выключения в случае возникновения острой необходимости!

В следующей сводке показано, что у каждой машины есть собственный уникальный процесс безопасного выключения, которым управляет компьютер (и собственные функции):

```
StopFolding();           // Машина для обработки и укладки
FinishWelding();         // Машина для сварки
PaintOff();              // Машина для покраски
```

Реализация системы управления фабрикой без использования делегатов

К реализации функции выключения машин в программе управления можно применить следующий простой подход:

```
class Controller
{
    // Поля, представляющие различные машины
    private FoldingMachine folder;
    private WeldingMachine welder;
    private PaintingMachine painter;

    ...
    public void ShutDown()
    {
        folder.StopFolding();
        welder.FinishWelding();
        painter.PaintOff();
    }
    ...
}
```

Этот подход позволяет создать вполне работоспособную, но недостаточно хорошо расширяемую и гибкую систему. Если на фабрике будет закуплена новая машина, код придется изменить, поскольку класс `Controller` и код для управления машинами имеют весьма тесную связь.

Реализация системы управления фабрикой с использованием делегата

Хотя имена у всех методов разные, у них один и тот же «профиль»: они не получают никаких параметров и не возвращают никаких значений. (Что произойдет при других условиях, вы увидите чуть позже, так что немного потерпите.) То есть для всех методов используется следующий общий формат:

```
void methodName();
```

Именно здесь и может пригодиться делегат. Вы можете воспользоваться делегатом, соответствующим этому профилю, для ссылки на любой метод остановки оборудования. Объявить делегат можно так:

```
delegate void stopMachineryDelegate();
```

Обратите внимание на следующие особенности:

- ❑ здесь используется ключевое слово `delegate`;
- ❑ здесь указываются возвращаемый тип (в данном случае `void`), имя делегата (`stopMachineryDelegate`) и любые параметры (в рассматриваемом случае их нет).

После объявления делегата можно создать экземпляр и заставить его ссылаться на соответствующий метод, воспользовавшись оператором составного присваивания `+=`. Сделать это можно в конструкторе класса контроллера:

```
class Controller
{
    delegate void stopMachineryDelegate();           // Тип делегата
    private stopMachineryDelegate stopMachinery; // Экземпляр делегата
    ...
    public Controller()
    {
        this.stopMachinery += folder.StopFolding;
    }
    ...
}
```

Освоить этот синтаксис нетрудно. К делегату добавляется новый метод, причем следует запомнить, что в данный момент этот метод не вызывается. Чтобы получить новое назначение при использовании с делегатами, оператор `+` перегружается. (Более подробно перегрузка операторов рассматривается в главе 22 «Перегрузка операторов».) Обратите внимание на то, что здесь просто указывается имя метода без круглых скобок и параметров.

В отношении неинициализированного делегата оператор `+=` можно использовать без всяких опасений. Инициализация будет выполнена автоматически.

В качестве альтернативного варианта для явной инициализации делегата с единственным указанным методом можно воспользоваться ключевым словом `new`:

```
this.stopMachinery = new stopMachineryDelegate(folder.StopFolding);
```

Метод можно вызвать, задействуя для этого делегата:

```
public void ShutDown()
{
    this.stopMachinery();
    ...
}
```

Чтобы применить делегата, следует использовать тот же синтаксис, что и для вызова метода. Если метод, на который ссылается делегат, получает параметры, то при задействовании делегата их нужно указать в круглых скобках.



ПРИМЕЧАНИЕ При попытке задействовать делегата, не прошедшего инициализацию и не ссылающегося ни на один метод, будет выдано исключение `NullReferenceException`.

Важным преимуществом от использования делегата является то, что он может одновременно ссылаться на более чем один метод. Чтобы добавить методы к делегату, нужно просто воспользоваться оператором `+=`:

```
public Controller()
{
    this.stopMachinery += folder.StopFolding;
    this.stopMachinery += welder.FinishWelding;
    this.stopMachinery += painter.PaintOff;
}
```

Когда в методе `Shutdown` класса `Controller` задействуется делегат `this.stopMachinery()`, каждый метод вызывается по очереди в автоматическом режиме. Метод `Shutdown` не нуждается в сведениях о том, сколько машин имеется и как называются их методы.

Метод можно удалить из делегата, воспользовавшись для этого оператором составного присваивания `-=`:

```
this.stopMachinery -= folder.StopFolding;
```

Используемая в данный момент схема добавляет методы управления машинами к делегату в конструкторе `Controller`. Чтобы сделать класс `Controller` полностью независимым от различных машин, нужно объявить тип `stopMachineryDelegate` открытым и предоставить средства, позволяющие внешним по отношению

к `Controller` классам добавлять методы к делегату. Для этого есть несколько вариантов.

- ❑ Сделать переменную делегата `stopMachinery` открытой:

```
public stopMachineryDelegate stopMachinery;
```

- ❑ Оставить переменную делегата `stopMachinery` закрытой, но предоставить к ней доступ, создав свойство для чтения и записи:

```
private stopMachineryDelegate stopMachinery;  
...  
public stopMachineryDelegate StopMachinery  
{  
    get  
    {  
        return this.stopMachinery;  
    }  
    set  
    {  
        this.stopMachinery = value;  
    }  
}
```

- ❑ Предоставить полную инкапсуляцию, реализовав отдельные методы `Add` и `Remove`. Метод `Add` получает метод в качестве параметра и добавляет его к делегату, а метод `Remove` удаляет указанный метод из делегата (обратите внимание на то, что метод указывается в качестве параметра путем использования типа делегата):

```
public void Add(stopMachineryDelegate stopMethod)  
{  
    this.stopMachinery += stopMethod;  
}  
  
public void Remove(stopMachineryDelegate stopMethod)  
{  
    this.stopMachinery -= stopMethod;  
}
```

Ярый сторонник объектно-ориентированного подхода, наверное, выбрал бы вариант с методами `Add` и `Remove`. Но и другие подходы являются вполне жизнеспособными и довольно часто используемыми альтернативами, именно поэтому они здесь и показаны.

Какой бы из приемов вы ни выбрали, код, добавляющий методы управления машинами к делегату, нужно убирать из конструктора `Controller`. Тогда можно будет создать экземпляр класса `Controller` и объекты, представляющие другие подобные этим машины (в этом примере применяется подход с использованием `Add` и `Remove`):

```
Controller control = new Controller();
FoldingMachine folder = new FoldingMachine();
WeldingMachine welder = new WeldingMachine();
PaintingMachine painter = new PaintingMachine();
...
control.Add(folder.StopFolding);
control.Add(welder.FinishWelding);
control.Add(painter.PaintOff);
...
control.ShutDown();
...
```

Объявление и использование делегатов

В следующих упражнениях вы завершите разработку приложения, формирующего часть системы для компании Wide World Importers, которая занимается импортом и продажей строительных материалов и инструментов. Приложение, над которым вы будете работать, дает клиентам возможность просматривать товарные позиции, имеющиеся на складах компании Wide World Importers, и размещать заказы на них. Приложение содержит форму, показывающую доступные на данный момент товары, а также панель со списком уже выбранных клиентом товаров. Когда клиент желает разместить заказ, он может щелкнуть на кнопке формы **Checkout** (Разместить заказ). Затем заказ обрабатывается и панель очищается.

На момент размещения клиентом заказа производится несколько действий.

- Клиент получает требование об оплате.
- Товарные позиции поочередно проверяются на наличие для некоторых из них возрастных ограничений (например, когда дело касается электроинструментов), а также проверяются и отслеживаются характеристики заказа.
- Для доставки товара формируется сопроводительная накладная, содержащая краткие сведения о заказе.

Логика процессов проверки и доставки не зависит от логики размещения заказа, и порядок, в котором эти процессы происходят, роли не играет. Более того, любой из этих элементов в будущем может быть откорректирован, и по мере того как в перспективе станут изменяться условия ведения бизнеса или нормативные требования, для операции размещения заказа может потребоваться дополнительная обработка. Поэтому, чтобы проще было сопровождать и корректировать программу, желательно отделить логику оплаты и оформления заказа от процессов проверки и доставки. Начнем с исследования приложения, чтобы увидеть, что на данный момент оно не в состоянии выполнить эту задачу. Затем вы измените структуру приложения, чтобы удалить зависимости между логикой размещения заказа и логикой проверки и доставки.

Исследование приложения Wide World Importers

Откройте в среде Microsoft Visual Studio 2015 проект Delegates, который находится в папке \Microsoft Press\VCBS\Chapter 20\Delegates вашей папки документов.

Щелкните в меню Отладка на пункте Начать отладку. Проект будет собран и запущен. Появится форма, показывающая доступные товары, а также панель с данными о заказе (изначально она пуста). Приложение отображает товары в элементе управления GridView, который позволяет осуществлять горизонтальную прокрутку (рис. 20.1).

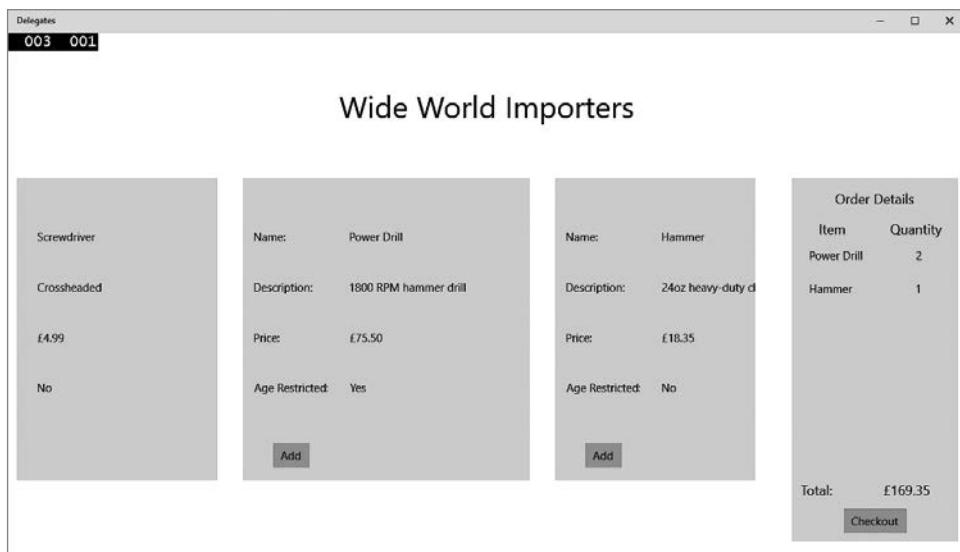


Рис. 20.1

Выберите один или несколько товаров, после чего щелкните на кнопке Add (Добавить), чтобы включить их в корзину покупателя. При этом выберите хотя бы один товар с возрастными ограничениями на приобретение (с пометкой Age Restricted Yes). Как только товар будет добавлен, он тут же появится на расположенной справа панели Order Details (Сведения о заказе). Обратите внимание на то, что при многократном добавлении одного и того же товара его количество с каждым щелчком будет увеличиваться. (В этой версии приложения возможность удаления товара из корзины не реализована.)

На панели Order Details щелкните на кнопке Checkout (Разместить заказ). Появится сообщение, показывающее, что заказ был размещен. Заказу присваивается уникальный идентификационный номер (ID), и этот ID отображается вместе со стоимостью заказа. Щелкните на кнопке Закрыть, чтобы убрать сообщение с экрана, а затем вернитесь в среду Visual Studio 2015, чтобы остановить отладку.

Раскройте в обозревателе решений узел проекта Delegates, после чего откройте файл Package.appxmanifest. Появится конструктор манифеста приложения. Щелкните в нем на вкладке Упаковка (Packaging). Обратите внимание на значение в поле Имя пакета (Package Name). Оно имеет форму глобально-уникального идентификатора (globally unique identifier (GUID)).

Используя Проводник, зайдите в папку %USERPROFILE%\AppData\Local\Packages\yyy\LocalStorage, где *yyy* — это значение идентификатора, начинающегося с GUID, на который вы только что обратили внимание. Это локальная папка для приложения Wide World Importers. В ней вы должны увидеть два файла, один с именем audit-nnnnnn.xml (где *nnnnnn* — идентификационный номер показанного ранее заказа), а другой с именем dispatch-nnnnnn.txt. Первый файл был создан проверочным компонентом приложения, а второй является сопроводительной накладной, созданной компонентом доставки товара.



ПРИМЕЧАНИЕ Если файла audit-nnnnnn.xml не будет, значит вы при размещении заказа не выбрали ни одного товара с возрастными ограничениями на приобретение. В таком случае вернитесь в приложение и создайте новый заказ, включающий один или несколько таких товаров.

Откройте в среде Visual Studio файл audit-nnnnnn.xml. В нем содержатся список имеющихся в заказе товаров с возрастными ограничениями на приобретение, а также номер и дата заказа. Он должен иметь следующий вид (рис. 20.2).



Рис. 20.2

Когда закончите изучение списка, закройте файл в среде Visual Studio.

Откройте в Блокноте файл `dispatch-nnnnnnn.txt`. Он содержит краткие сведения о заказе с перечислением ID заказа и его стоимости. Он должен иметь примерно следующий вид (рис. 20.3).



Рис. 20.3

Закройте Блокнот, вернитесь в среду Visual Studio 2015 и остановите отладку. Обратите внимание на то, что в среде Visual Studio решение состоит из следующих проектов.

- ❑ **Delegates.** Этот проект содержит само приложение. Пользовательский интерфейс определен в файле `MainPage.xaml`, а логика приложения содержится в файле `MainPage.xaml.cs`.
- ❑ **AuditService.** Этот проект содержит компоненты, реализующие процесс проверки. Он упакован как библиотека классов и содержит всего один класс `Auditor`. Этот класс предоставляет единственный открытый метод `AuditOrder`, который проверяет заказ и создает файл `audit-nnnnnnn.xml`, если в заказе содержится какой-либо товар с возрастными ограничениями на приобретение.
- ❑ **DeliveryService.** Этот проект содержит компонент, выполняющий логику доставки, упакованный как библиотека классов. Функциональные средства, связанные с доставкой, содержат класс `Shipper`, предоставляющий открытый метод `ShipOrder`, который обрабатывает процесс доставки, а также создает сопроводительную накладную.



ПРИМЕЧАНИЕ Вы, конечно, можете изучить код классов `Auditor` и `Shipper`, но глубоко вникать во внутреннюю работу этих компонентов в данном приложении нет никакой необходимости.

- ❑ **DataTypes.** Этот проект содержит типы данных, используемые другими проектами. В классе `Product` определяются особенности товаров, показываемых приложением, а данные товаров хранятся в классе `ProductsDataSource`. (В данный момент в приложении используется небольшой жестко заданный набор товаров. В производственной системе эта информация должна извлекаться из базы данных или из веб-сервиса.) В классах `Order` и `OrderItem`

реализуется структура заказа, в каждом заказе (*order*) содержится одна или несколько товарных позиций (*order items*).

Вызовите в окно редактора файл MainPage.xaml.cs, принадлежащий проекту Delegates, и изучите в нем закрытые поля и конструктор MainPage. Для нас важны следующие элементы:

```
...
private Auditor auditor = null;
private Shipper shipper = null;
public MainPage()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
}
```

В полях *auditor* и *shipper* содержатся ссылки на экземпляры классов *Auditor* и *Shipper*, а конструктор создает экземпляры этих объектов.

Найдите метод *CheckoutButtonClicked*. Этот метод запускается после щелчка пользователя на кнопке Checkout, предназначеннной для размещения заказа. Первые несколько строк имеют следующий вид:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Выполнение обработки, связанной с размещением заказа
        if (this.requestPayment())
        {
            this.auditor.AuditOrder(this.order);
            this.shipper.ShipOrder(this.order);
        }
        ...
    }
    ...
}
```

Это метод реализует обработку, связанную с размещением заказа. Он требует от пользователя оплаты заказа, после чего задействует метод *AuditOrder* объекта *auditor*, а затем метод *ShipOrder* объекта *shipper*. Сюда будет добавляться любая востребуемая в будущем дополнительная бизнес-логика. Весь остальной код этого метода (после инструкции *if*) занимается управлением пользовательским интерфейсом: отображением для пользователя окна сообщения и очисткой панели сведений о заказе Order Details.



ПРИМЕЧАНИЕ В целях упрощения кода имеющийся в данном приложении метод *requestPayment* в настоящий момент просто возвращает *true*, чтобы показать, что платеж был получен. В реальном мире этот метод должен был бы выполнить полноценную обработку платежа.

Хотя приложение работает в точном соответствии с тем, как было анонсировано, компоненты *Auditor* и *Shipper* довольно сильно интегрированы в обработку, связанную с размещением заказа. Если эти компоненты будут изменены, приложение придется обновлять. К тому же, если придется вносить в процесс, связанный с размещением заказа, какую-либо дополнительную логику, возможно, выполняемую с использованием других компонентов, эту часть приложения придется переработать.

В следующем упражнении вы увидите, как можно отделить от приложения обработку бизнес-информации, связанной с размещением заказа. В процессе этой обработки по-прежнему придется задействовать компоненты *Auditor* и *Shipper*, но при этом нужно получить возможность для расширения, достаточную для простого внедрения дополнительных компонентов. Эта цель будет достигаться созданием компонента под названием *CheckoutController*. Он займется реализацией бизнес-логики для процесса размещения заказа и предоставлением делегата, позволяющего приложению указывать, какие компоненты и методы должны быть включены в этот процесс. Компонент *CheckoutController* будет задействовать эти методы путем использования делегата.

Создание компонента *CheckoutController*

В обозревателе решений щелкните правой кнопкой мыши на решении *Delegates*, укажите на пункт **Добавить**, а затем щелкните на пункте **Создать проект**. Раскройте в левой панели диалогового окна **Добавить новый проект** узел **Windows**, а затем щелкните на узле **Универсальные**. Выберите в средней панели шаблон **Библиотека классов (Универсальное приложение Windows)**. Наберите в поле **Имя** строку **CheckoutService**, а затем щелкните на кнопке **OK**.

В обозревателе решений раскройте проект *CheckoutService*, щелкните правой кнопкой мыши на файле *Class1.cs* и на пункте **Переименовать**. Измените имя файла на *CheckoutController.cs*, а затем нажмите **Ввод**. Получив предложение, позвольте среди Visual Studio переименовать все ссылки на *Class1* в ссылки на *CheckoutController*. В проекте *CheckoutService* щелкните правой кнопкой мыши на узле **Ссылки**, а затем щелкните на пункте **Добавить ссылку**. Щелкните в левой панели диалогового окна **Менеджер ссылок** — *CheckoutService* на пункте **Решение**. Выберите в средней панели проект *DataTypes*, а затем щелкните на кнопке **OK**. Класс *CheckoutController* будет использовать класс *Order*, определенный в проекте *DataTypes*.

Выполните в окне редактора файл *CheckoutController.cs* и добавьте к списку в самом начале файла следующую директиву **using**:

```
using DataTypes;
```

Добавьте к классу *CheckoutController* открытый тип делегата по имени *CheckoutDelegate*, выделенный в следующем примере кода жирным шрифтом:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
}
```

Этот тип делегата можно использовать для ссылки на методы, получающие параметр типа Order и не возвращающие результат. Это соответствует профилю методов AuditOrder и ShipOrder классов Auditor и Shipper.

Добавьте основанный на этом типе делегата открытый делегат по имени CheckoutProcessing:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;
}
```

Выполните в окне редактора файл MainPage.xaml.cs проекта Delegates и найдите метод requestPayment (в конце файла). Вырежьте этот метод из класса MainPage. Вернитесь к файлу CheckoutController.cs и вставьте метод requestPayment в класс CheckoutController, добавив тем самым к нему код, выделенный в следующем примере жирным шрифтом:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;

    private bool requestPayment()
    {
        // Здесь производится обработка платежа

        // Логика обработки платежа в данном примере не реализована,
        // метод просто возвращает true, показывая, что платеж принят
        return true;
    }
}
```

Добавьте к классу CheckoutController метод StartCheckoutProcessing, выделенный здесь жирным шрифтом:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;

    private bool requestPayment()
    {
        ...
    }

    public void StartCheckoutProcessing(Order order)
```

```

{
    // Обработка данных для размещения заказа
    if (this.requestPayment())
    {
        if (this.CheckoutProcessing != null)
        {
            this.CheckoutProcessing(order);
        }
    }
}
}

```

Этот метод предоставляет функциональные средства для размещения заказа, ранее реализованные в методе `CheckoutButtonClicked` класса `MainPage`. Он запрашивает проведение оплаты заказа, а затем исследует делегата `CheckoutProcessing`: если его значение не равно `null` (то есть он ссылается на один или несколько методов), метод задействует делегата. В этом случае запускаются все методы, на которые ссылается этот делегат.

В обозревателе решений в проекте `Delegates` щелкните правой кнопкой мыши на узле `Ссылки`, а затем щелкните на пункте `Добавить ссылку`. Щелкните на левой панели диалогового окна `Менеджер ссылок — Delegates` на пункте `Решение`. Выберите в средней панели проект `CheckoutService`, а затем щелкните на кнопке `OK`.

Вернитесь к файлу `MainPage.xaml.cs` проекта `Delegates` и добавьте к списку в его начале следующую директиву `using`:

```
using CheckoutService;
```

Добавьте к классу `MainPage` показанную жирным шрифтом закрытую переменную по имени `checkoutController`, имеющую тип `CheckoutController`, и инициализируйте ее значением `null`:

```

public ... class MainPage : ...
{
    ...
    private Auditor auditor = null;
    private Shipper shipper = null;
private CheckoutController checkoutController = null;
    ...
}

```

Найдите конструктор `MainPage`. Создайте после инструкций, создающих компоненты `Auditor` и `Shipper`, экземпляр компонента `CheckoutController`, выделенный здесь жирным шрифтом:

```

public MainPage()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
this.checkoutController = new CheckoutController();
}

```

Добавьте к конструктору после только что введенной инструкции следующие инструкции, выделенные жирным шрифтом:

```
public MainPage()
{
    ...
    this.checkoutController = new CheckoutController();
    this.checkoutController.CheckoutProcessing += this.auditor.AuditOrder;
    this.checkoutController.CheckoutProcessing += this.shipper.ShipOrder;
}
```

Этот код добавляет к делегату `CheckoutProcessing` объекта `CheckoutController` ссылки на методы `AuditOrder` и `ShipOrder` объектов `Auditor` и `Shipper`.

Найдите метод `CheckoutButtonClicked`. Замените в блоке `try` код, выполняющий обработку данных для размещения заказа (блок инструкции `if`), инструкцией, показанной здесь жирным шрифтом:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Обработка данных для размещения заказа
        this.checkoutController.StartCheckoutProcessing(this.order);

        // Вывод на экран кратких сведений о заказе
        ...
    }
    ...
}
```

Теперь логика, связанная с размещением заказа, отделена от компонентов, использующих эту обработку данных для размещения заказа. Компоненты `CheckoutController`, которые должны использоваться, определяются бизнес-логикой в классе `MainPage`.

Тестирование приложения

Щелкните в меню Отладка на пункте Начать отладку. Приложение будет собрано и запущено. Когда появится форма Wide World Importers, выберите несколько товаров (включая хотя бы один с возрастными ограничениями на его приобретение), а затем щелкните на кнопке `Checkout`.

Когда появится сообщение о размещении заказа — `Order Placed`, запишите номер заказа, после чего щелкните на кнопке `Close` или `OK`.

Перейдите в Проводник и зайдите в папку `%USERPROFILE%\AppData\Local\Packages\yyy\LocalStorage`, где `yyy` является значением идентификатора, начинающегося с ранее записанного GUID. Убедитесь в том, что были созданы новые файлы

`audit-nnnnnn.xml` и `dispatch-nnnnnn.txt`, где `nnnnnn` является числом, идентифицирующим новый заказ. Изучите эти файлы и убедитесь в том, что в них содержатся данные заказа.

Вернитесь в среду Visual Studio 2015 и остановите отладку.

Лямбда-выражения и делегаты

Во всех показанных до сих пор примерах добавления метода к делегату использовалось имя метода. Например, в ранее рассмотренном сценарии автоматизированной фабрики метод `StopFolding` объекта `folder` добавлялся к делегату `stopMachinery` следующим образом:

```
this.stopMachinery += folder.StopFolding;
```

Этот подход хорош в том случае, когда имеется метод, соответствующий сигнатуре делегата. А что делать, если у метода `StopFolding` будет следующая сигнатура:

```
void StopFolding(int shutDownTime); // Завершение работы через указанное количество  
// секунд
```

Теперь эта сигнатура отличается от сигнатурой методов `FinishWelding` и `PaintOff`, и поэтому использовать одного и того же делегата для управления всеми тремя методами нельзя. Так что делать?

Создание метода-адаптера

Одним из способов обхода данной проблемы является создание еще одного метода по имени `FinishFolding`, но такого, который сам по себе никаких параметров не принимает:

```
void FinishFolding()  
{  
    folder.StopFolding(0); // Немедленное завершение работы  
}
```

Затем метод `FinishFolding` можно добавить к делегату `stopMachinery` вместо метода `StopFolding`, используя для этого тот же синтаксис, что и раньше:

```
this.stopMachinery += folder.FinishFolding;
```

При задействовании делегат `stopMachinery` вызывает `FinishFolding`, который в свою очередь вызывает метод `StopFolding`, передавая ему параметр `0`.



ПРИМЕЧАНИЕ Метод `FinishFolding` является классическим примером адаптера — метода, который преобразует (или адаптирует) метод, чтобы придать ему другую сигнатуру. Этот шаблон используется довольно широко и относится к набору шаблонов, описание которых дано в книге Эриха Гамма (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (Addison-Wesley Professional, 1994).

Зачастую такие методы-адаптеры имеют небольшой размер и запросто могут затеряться среди целого моря методов, особенно в большом по объему классе. Более того, метод вряд ли будет вызываться, за исключением тех случаев, когда он используется для адаптации метода `StopFolding` при использовании его делегатом. Для подобных ситуаций в C# предоставляется лямбда-выражение. Такие выражения рассматривались в главе 18, и примеры с ними ранее уже не раз были показаны в данной главе. В сценарии фабрики можно использовать следующее лямбда-выражение:

```
this.stopMachinery += ((_) => folder.StopFolding(0));
```

При задействовании делегат `stopMachinery` запустит код, определяемый лямбда-выражением, который в свою очередь вызовет метод `StopFolding` с соответствующим параметром.

Включение уведомлений путем использования событий

Теперь вы знаете, как объявляется тип делегата, вызывается делегат и создаются экземпляры делегата. Но это лишь половина истории. Несмотря на возможность опосредованного вызова любого количества методов с помощью делегатов, вам все же приходится задействовать делегата явным образом. Зачастую было бы полезно запускать делегаты автоматически при каком-то значительном событии. Например, в сценарии автоматизированной фабрики может быть жизненно необходимым наличие возможности задействования делегата `stopMachinery` и остановки работы оборудования в случае обнаружения перегрева машины.

В .NET Framework предоставляются события, которыми можно воспользоваться для определения и перехвата важных действий и организовать задействование делегата, позволяющего справиться с ситуацией. События предоставляются многими классами .NET Framework. Многие элементы управления, которые можно поместить в форму UWP-приложения, а также сами классы Windows используют события для запуска кода в случае, если пользователь, к примеру, щелкает на кнопке или что-нибудь вводит в поле. Можно также объявлять собственные события.

Объявление события

Событие объявляется в классе, предназначенном для действия в качестве источника события. Обычно источником события является класс, отслеживающий свою среду и инициирующий событие, когда происходит что-либо значимое. На автоматизированной фабрике источником события может быть класс, отслеживающий температуру каждой машины. Такой класс будет инициировать событие «перегрев машины», если обнаружит, что у машины превышен порог теплового излучения, то есть она перегрелась. При инициировании события у него есть список методов, подлежащих вызову. Иногда эти методы называют подписчиками. Эти методы должны быть подготовлены к обработке события «перегрев машины» и к принятию необходимого для исправления ситуации действия — остановке машин.

Событие объявляется точно так же, как и поле. Но поскольку события предназначены для использования с делегатами, типом события должен быть делегат, а перед объявлением должно ставиться ключевое слово `event`. Для объявления событий следует использовать такой синтаксис:

```
event delegateTypeName eventName
```

Возьмем, к примеру, делегата `StopMachineryDelegate` из автоматизированной фабрики. Он был перемещен в класс по имени `TemperatureMonitor`, который предоставляет интерфейс для различных датчиков контроля температуры оборудования (по логике это место больше подходит для события, чем класс `Controller`):

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    ...
}
```

Событие `MachineOverheating`, задействующее делегата `stopMachineryDelegate`, можно определить следующим образом:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
}
```

Логика в классе `TemperatureMonitor` (которая здесь не показана) в случае необходимости инициирует событие `MachineOverheating`. (Как инициируется событие, будет показано в следующем разделе.) К тому же методы добавляются

к событию (этот процесс называется подпиской), а не к делегату, на котором основано это событие. Этот аспект событий сейчас и будет рассмотрен.

Подписка на событие

Как и делегаты, события поставляются в готовом виде с оператором `+=`. С помощью этого оператора осуществляется подписка на событие. В автоматизированной фабрике программные средства, управляющие каждой машиной, могут быть подготовлены под вызов методов, останавливающих их работу, при выдаче события `MachineOverheating` следующим образом:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
}
...
TemperatureMonitor tempMonitor = new TemperatureMonitor();
...
tempMonitor.MachineOverheating += (() => { folder.StopFolding(0); });
tempMonitor.MachineOverheating += welder.FinishWelding;
tempMonitor.MachineOverheating += painter.PaintOff;
```

Обратите внимание на то, что здесь используется точно такой же синтаксис, что и при добавлении метода к делегату. Можно даже подписаться на событие, используя лямбда-выражение. Когда будет инициировано событие `tempMonitor.MachineOverheating`, оно вызовет все подписавшиеся на него методы и остановит машины.

Отмена подписки на событие

Зная, что для прикрепления делегата к событию используется оператор `+=`, можно догадаться, что для открепления делегата от события используется оператор `-=`. Вызов оператора `-=` приводит к удалению метода из внутренней коллекции делегата, принадлежащего событию. Зачастую это действие называют отменой подписки на событие.

Иницирование события

Иницировать событие можно, вызвав его наподобие метода. При инициировании события вызываются по порядку все прикрепленные делегаты. Например,

в следующем фрагменте кода показан класс `TemperatureMonitor` с закрытым методом `Notify`, инициирующим событие `MachineOverheating`:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
    private void Notify()
    {
        if (this.MachineOverheating != null)
        {
            this.MachineOverheating();
        }
    }
    ...
}
```

Это общая идиома. Проверка на `null` необходима, поскольку поле события изначально имеет значение `null` и становится непустым, только когда на событие с использованием оператора `+=` подписывается какой-нибудь метод. Если попытаться инициировать событие, имеющее значение `null`, будет выдано исключение `NullReferenceException`. Если делегат, определяющий событие, ожидает получения каких-либо параметров, то при инициировании события должны быть предоставлены соответствующие аргументы. Примеры этого будут показаны чуть позже.



ВНИМАНИЕ У событий есть очень полезное встроенное свойство обеспечения безопасности. Открытое событие, такое как `MachineOverheating`, может быть инициировано только методами в том классе, в котором оно определено (в классе `TemperatureMonitor`). Любые попытки инициировать событие за пределами класса приведут к тому, что компилятор выдаст ошибку.

Основные сведения о событиях пользовательского интерфейса

Как уже упоминалось, события широко применяются классами .NET Framework и элементами управления, используемыми для создания графических интерфейсов пользователя. Например, класс `Button`, являющийся производным от класса `ButtonBase`, наследует открытое событие по имени `Click`, имеющее тип `RoutedEventHandler`. Делегат `RoutedEventHandler` ожидает передачи двух параметров: ссылки на объект, ставший причиной инициирования события, и объекта `RoutedEventArgs`, который содержит дополнительную информацию о событии:

```
public delegate void RoutedEventHandler(Object sender, RoutedEventArgs e);
```

Класс **Button** имеет следующий вид:

```
public class ButtonBase: ...
{
    public event RoutedEventHandler Click;
    ...
}

public class Button: ButtonBase
{
    ...
}
```

Когда происходит щелчок на показанной на экране кнопке, класс **Button** автоматически инициирует событие **Click**. Это обстоятельство упрощает создание делегата для избранного метода и прикрепление этого делегата к требуемому событию. В следующем примере показан пример кода для UWP-формы, содержащей кнопку, названную **okay**, и код для подключения к событию **Click** кнопки **okay** к методу **okayClick**:

```
partial class MainPage :
    global::Windows.UI.Xaml.Controls.Page,
    global::Windows.UI.Xaml.Markup.IComponentConnector,
    global::Windows.UI.Xaml.Markup.IComponentConnector2
{
    ...
    public void Connect(int connectionId, object target)
    {
        switch(connectionId)
        {
            case 1:
            {
                this.okay = (global::Windows.UI.Xaml.Controls.Button)(target);
                ...
                ((global::Windows.UI.Xaml.Controls.Button)this.okay).Click +=
                    this.okayClick;
                ...
            }
            break;
        default:
            break;
        }
        this._contentLoaded = true;
    }
    ...
}
```

Обычно этот код скрыт от ваших глаз. При использовании в среде Visual Studio 2015 окна конструктора и установке в описании формы на языке XAML для свойства **Click** кнопки **okay** метода **okayClick** среда Visual Studio 2015 создает этот код самостоятельно. Вам останется только записать логику своего приложения.

в обрабатывающем событие методе `okayClick`, в той части кода, к которой у вас есть доступ и которая в данном случае находится в файле `MainPage.xaml.cs`:

```
public sealed partial class MainPage : Page
{
    ...
    private void okayClick(object sender, RoutedEventArgs e)
    {
        // Ваш код для обработки события Click
    }
}
```

События, инициируемые различными элементами управления пользовательского графического интерфейса, неизменно развиваются по этому шаблону. События являются типом делегата, чья сигнатура имеет тип возвращаемого значения `void` и два аргумента. Первым аргументом всегда является отправитель (источник) события — `sender`, вторым аргументом всегда является экземпляр класса `EventArgs` (или класса, производного от `EventArgs`).

Наличие аргумента `sender` позволяет многократно использовать один и тот же метод для нескольких событий. Метод делегата может исследовать аргумент `sender` и отреагировать на него соответствующим образом. Например, один и тот же метод можно использовать для подписки на событие `Click` для двух кнопок. (Один и тот же метод добавляется к двум разным событиям.) При инициировании события имеющийся в методе код может исследовать аргумент `sender`, чтобы выяснить, на какой из кнопок был сделан щелчок.

Использование событий

В предыдущем упражнении вы внесли в приложение Wide World Importers изменения, позволяющие отделить логику проверки и доставки заказа от процесса его размещения. Созданный вами класс `CheckoutController` вызывает компоненты проверки и доставки путем использования делегата и ничего не знает об этих компонентах или о запускаемых ими методах — эта обязанность возлагается на приложение, создающее объект типа `CheckoutController` и добавляющее соответствующие ссылки к делегату. Но, может быть, компоненту было бы полезно иметь возможность оповестить приложение о завершении возложенной на него обработки данных и позволить приложению навести необходимый порядок.

Поначалу это может показаться несколько странным, поскольку есть уверенность в том, что при задействовании приложением делегата в объекте типа `CheckoutController` все методы, на которые ссылается этот делегат, будут запущены и приложение продолжит работу, выполняя следующую инструкцию только после того, как эти методы завершат выполнение своей задачи. Но так

бывает не всегда! В главе 24 «Сокращение времени отклика путем выполнения асинхронных операций» показано, что методы могут запускаться в асинхронном режиме и при вызове метода он может не завершить свою работу до того, как выполнение приложения продолжится с использованием следующей инструкции. Это особенно актуально для UWP-приложений, в которых затратные по времени операции выполняются в потоках, запускаемых в фоновом режиме, чтобы позволить пользовательскому интерфейсу восстановить свою способность к откликам. В методе `CheckoutButtonClicked` приложения Wide World Importers за кодом, который задействует делегата, следует инструкция, выводящая диалоговое окно с сообщением о том, что заказ был размещен:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Выполнение действий, связанных с размещением заказа
        this.checkoutController.StartCheckoutProcessing(this.order);

        // Вывод общих сведений о заказе
        MessageDialog dlg = new MessageDialog(...);
        dlg.ShowAsync();
        ...
    }
    ...
}
```

По сути, нет никаких гарантий того, что ко времени появления диалогового окна обработка данных, осуществляемая делегируемыми методами, уже завершилась, поэтому сообщение фактически может вводить пользователя в заблуждение. Именно здесь и можно получить неоценимую помощь от использования события. Оба компонента, и `Auditor`, и `Shipper`, могут опубликовать событие, на которое подписывается приложение. Этот событие может быть инициировано компонентами только при завершении ими возложенной на них обработки данных. Когда приложение получит это событие, оно сможет показать сообщение, пребывая в полной уверенности, что теперь это сообщение соответствует действительности.

В следующем упражнении вы внесете изменения в классы `Auditor` и `Shipper`, позволяющие инициировать событие, возникающее только при условии завершения ими возложенной на них обработки данных. Приложение будет подписано на событие каждого из компонентов и при его возникновении выведет на экран соответствующее сообщение.

Добавление события к классу `CheckoutController`

Выполните в среде Visual Studio 2015 решение `Delegates`. Откройте в окне редактора файл `Auditor.cs`, принадлежащий проекту `AuditService`. Дополните класс `Auditor`

открытым делегатом по имени `AuditingCompleteDelegate`. Этот делегат укажет на метод, получающий строковый параметр по имени `message` и возвращающий значение типа `void`. Определение этого делегата показано жирным шрифтом:

```
class Auditor
{
    public delegate void AuditingCompleteDelegate(string message);
    ...
}
```

Добавьте к классу `Auditor` после делегата `AuditingCompleteDelegate` открытое событие по имени `AuditProcessingComplete`. Это событие, показанное жирным шрифтом, должно быть основано на делегате `AuditingCompleteDelegate`:

```
class Auditor
{
    public delegate void AuditingCompleteDelegate(string message);
    public event AuditingCompleteDelegate AuditProcessingComplete;
    ...
}
```

Найдите метод `AuditOrder`. Это метод, запускаемый при задействовании делегата в `CheckoutController`-объекте. Он вызывает еще один закрытый метод по имени `doAuditing` для реального выполнения операции проверки. Метод имеет следующий вид:

```
public void AuditOrder(Order order)
{
    this.doAuditing(order);
}
```

Прокрутите код до метода `doAuditing`. Код этого метода заключен в блок `try-catch`. Для создания XML-представления проверяемого заказа и сохранения его в файле им используются XML API-интерфейсы библиотеки классов среды .NET Framework. (Подробности внутренней работы этих интерфейсов в данной книге не рассматриваются.)

Добавьте в поле блока `catch` блок `finally`, инициирующий событие `AuditProcessingComplete` и показанный далее жирным шрифтом:

```
private async void doAuditing(Order order)
{
    List<OrderItem> ageRestrictedItems = findAgeRestrictedItems(order);
    if (ageRestrictedItems.Count > 0)
    {
        try
        {
            ...
        }
        ...
    }
}
```

```
        catch (Exception ex)
        {
            ...
        }
    finally
    {
        if (this.AuditProcessingComplete != null)
        {
            this.AuditProcessingComplete(
                $"Audit record written for Order {order.OrderID}");
        }
    }
}
```

Откройте в окне редактора файл `Shipper.cs`, принадлежащий проекту `DeliveryService`. Дополните класс `Shipper` открытым делегатом `ShippingCompleteDelegate`. Этот делегат укажет на метод, получающий строковый параметр по имени `message` и возвращающий значение типа `void`. Определение делегата показано далее жирным шрифтом:

```
class Shipper
{
    public delegate void ShippingCompleteDelegate(string message);
    ...
}
```

Добавьте к классу `Shipper` показанное жирным шрифтом открытое событие `ShipProcessingComplete`, основанное на делегате `ShippingCompleteDelegate`:

```
class Shipper
{
    public delegate void ShippingCompleteDelegate(string message);
    public event ShippingCompleteDelegate ShipProcessingComplete;
    ...
}
```

Найдите метод `doShipping`, выполняющий логику доставки. После имеющегося в этом методе блока `catch` добавьте показанный далее жирным шрифтом блок `finally`, инициирующий событие `ShipProcessingComplete`:

```
private async void doShipping(Order order)
{
    try
    {
        ...
    }
    catch (Exception ex)
    {
        ...
    }
}
```

```

    finally
    {
        if (this.ShipProcessingComplete != null)
        {
            this.ShipProcessingComplete(
                $"Dispatch note generated for Order {order.OrderID}");
        }
    }
}

```

Выполните в окне конструктора разметку для файла MainPage.xaml, принадлежащего проекту Delegates. В панели XAML прокрутите код до первой установки значений для элементов RowDefinition. Код в формате XAML должен иметь следующий вид:

```

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid Margin="12,0,12,0" Loaded="MainPageLoaded">
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="10*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        ...
    </Grid>

```

Измените значение свойства Height последнего элемента RowDefinition, показанного далее жирным шрифтом, на 2*:

```

<Grid.RowDefinitions>
    ...
    <RowDefinition Height="10*"/>
    <RowDefinition Height="2*"/>
</Grid.RowDefinitions>

```

Это изменение разметки делает доступным пространство в нижней части формы, которое будет использоваться в качестве области для отображения сообщений, получаемых от компонентов Auditor и Shipper после того, как они инициируют свои события. Более подробно разметка пользовательского интерфейса с помощью элемента управления Grid рассматривается в главе 25 «Реализация пользовательского интерфейса для приложений универсальной платформы Windows».

Прокрутите код в XAML-панели до самого конца. Добавьте перед предпоследним тегом </Grid> следующие выделенные жирным шрифтом элементы ScrollViewer и TextBlock:

```

    ...
</Grid>
<ScrollViewer Grid.Row="4" VerticalScrollBarVisibility="Visible">

```

```
<TextBlock x:Name="messageBar" FontSize="18" />
</ScrollViewer>
</Grid>
</Grid>
</Page>
```

Эта разметка добавляет к области в нижней части экрана элемент управления **TextBlock** по имени **messageBar**. Этот элемент управления будет использоваться для вывода сообщений от объектов типа **Auditor** и **Shipper**.

Выведите в окно редактора файл **MainPage.xaml.cs**. Найдите метод **CheckoutButtonClicked** и удалите код, выводящий на экран краткие сведения о заказе. После этого блок **try** должен приобрести следующий вид:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Выполнение действий, связанных с размещением заказа
        this.checkoutController.StartCheckoutProcessing(this.order);

        // Удаление сведений о заказе, чтобы пользователь мог
        // начать все сначала и сделать новый заказ

        this.order = new Order { Date = DateTime.Now, Items = new
        List<OrderItem>(),
        OrderID = Guid.NewGuid(), TotalValue = 0 };
        this.orderDetails.DataContext = null;
        this.orderValue.Text = $"{order.TotalValue:C}";
        this.listViewHeader.Visibility = Visibility.Collapsed;
        this.checkout.IsEnabled = false;
    }
    catch (Exception ex)
    {
        ...
    }
}
```

Добавьте к классу **MainPage** закрытый метод по имени **displayMessage**. Этот метод будет получать строковый параметр по имени **message** и возвращать значение типа **void**. Добавьте к телу этого метода инструкцию, которая присоединяет значение параметра **message** к свойству **Text** элемента управления **messageBar** **TextBlock**, за которым следует символ новой строки (добавляемый код выделен жирным шрифтом):

```
private void displayMessage(string message)
{
    this.messageBar.Text += message + "\n";
}
```

Этот код заставляет сообщение появляться в области сообщений в нижней части формы.

Найдите конструктор для класса `MainPage` и добавьте к нему код, выделенный жирным шрифтом:

```
public MainPage()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
    this.checkoutController = new CheckoutController();
    this.checkoutController.CheckoutProcessing += this.auditor.AuditOrder;
    this.checkoutController.CheckoutProcessing += this.shipper.ShipOrder;

    this.auditor.AuditProcessingComplete += this.displayMessage;
    this.shipper.ShipProcessingComplete += this.displayMessage;
}
```

Эти инструкции осуществляют подписку на события, предоставляемые объектами типа `Auditor` и `Shipper`. При инициировании событий запускается метод `displayMessage`. Обратите внимание на то, что обработка обоих событий осуществляется одним и тем же методом.

Щелкните в меню Отладка на пункте Начать отладку, чтобы выполнить сборку и запустить приложения.

Когда появится форма `Wide World Importers`, выберите несколько товаров, включая хотя бы один с возрастными ограничениями на его приобретение, а затем щелкните на кнопке `Checkout`.

Убедитесь в том, что в элементе управления `TextBlock` в нижней части формы сначала появляется сообщение о выполнении записи с результатом проверки «`Audit record written`», а затем сообщение о создании сопроводительной на-кладной «`Dispatch note generated`» (рис. 20.4).

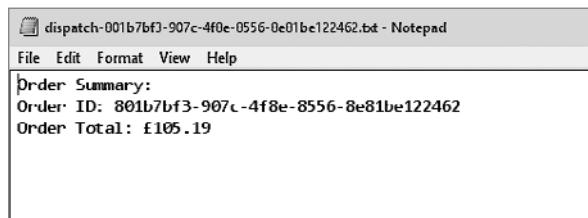


Рис. 20.4

Разместите еще несколько заказов и обратите внимание на новые сообщения, появляющиеся при каждом щелчке на кнопке `Checkout` (чтобы их увидеть, по мере заполнения области сообщений может понадобиться прокрутить текст).

Завершив работу, вернитесь в среду Visual Studio 2015 и остановите отладку.

Выводы

В этой главе вы научились использовать делегаты для ссылки на методы и вызова этих методов. Вы также узнали, как определяются лямбда-выражения, которые могут запускаться с помощью делегатов. И наконец, вы научились определять и использовать события для запуска метода на выполнение.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 21.

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Объявить тип делегата	Напишите ключевое слово <code>delegate</code> , после него укажите тип возвращаемого значения, имя типа делегата и любые типы параметров, например: <code>delegate void myDelegate();</code>
Создать экземпляр делегата, инициализированный одним указанным методом	Воспользуйтесь таким же синтаксисом, который использовался для класса или структуры: напишите ключевое слово <code>new</code> , после него укажите имя типа (имя делегата), а дальше в круглых скобках укажите аргументы. Аргумент должен быть методом, чья сигнатура в точности совпадает с сигнатурой делегата, например: <code>delegate void myDelegate();</code> <code>private void myMethod() { ... }</code> <code>...</code> <code>myDelegate del = new myDelegate(this.myMethod);</code>
Вызвать делегата	Используйте такой же синтаксис, что и для вызова метода, например: <code>myDelegate del;</code> <code>...</code> <code>del();</code>
Объявить событие	Напишите ключевое слово <code>event</code> , затем укажите имя типа (он должен относиться к делегатам), после чего укажите имя события, например: <code>class MyClass</code> <code>{</code> <code> public delegate void MyDelegate();</code>

Чтобы	Сделайте следующее
	<pre style="text-align: center;">... public event myDelegate MyEvent; }</pre>
Подписаться на событие	<p>Создайте экземпляр делегата (того же типа, что и событие) и присоедините его к событию, воспользовавшись оператором <code>+=</code>, например:</p> <pre>class MyEventHandlingClass { private MyClass myClass = new MyClass(); ... public void Start() { myClass.MyEvent += new myClass.MyDelegate (this.eventHandlingMethod); } private void eventHandlingMethod() { ... } }</pre> <p>Можно также заставить компилятор создать нового делегата автоматически, просто указав подписывающийся на событие метод:</p> <pre>public void Start() { myClass.MyEvent += this.eventHandlingMethod; }</pre>
Отменить подписку на событие	<p>Создайте экземпляр делегата (того же типа, что и событие) и отсоедините его от события, воспользовавшись оператором <code>-=</code>, например:</p> <pre>class MyEventHandlingClass { private MyClass myClass = new MyClass(); ... public void Stop() { myClass.MyEvent -= new myClass.MyDelegate (this.eventHandlingMethod); } ... }</pre> <p>Или:</p> <pre>public void Stop() { myClass.MyEvent -= this.eventHandlingMethod; }</pre>

Чтобы	Сделайте следующее
Инициировать событие	<p>Воспользуйтесь таким же синтаксисом, что и при вызове метода. Предоставляемые аргументы должны совпадать по типу с параметрами, ожидаемыми делегатом, на которого ссылается событие. Не забудьте проверить событие на содержание null-значения, например:</p> <pre>class MyClass { public event myDelegate MyEvent; ... private void RaiseEvent() { if (this.MyEvent != null) { this.MyEvent(); } ... } }</pre>

21

Запрос данных, находящихся в памяти, с помощью выражений в виде запросов

Прочитав эту главу, вы научитесь:

- определять запросы на встроенном в C# расширении (Language-Integrated Query (LINQ)) для исследования содержимого перечисляемых коллекций;
- использовать методы расширения LINQ и операторы запросов;
- объяснять, как LINQ откладывает вычисление запроса и как можно принудить систему к немедленному выполнению LINQ-запроса и кэшированию его результатов.

Большинство функциональных особенностей языка C# уже рассмотрено. Но до сих пор я обходил молчанием один важный аспект, который был бы наверняка полезен во многих приложениях: имеется в виду предоставляемая C# поддержка составления запросов данных. Вы уже поняли, что для моделирования данных можно определять структуры и классы и что для временного хранения данных в памяти можно использовать коллекции и массивы. Но как выполнить такую широко востребованную задачу, как поиск в коллекции элементов, отвечающих конкретному набору критериев? Например, если есть коллекция объектов `Customer`, как в ней найти всех клиентов, проживающих в Лондоне, или как определить, в каких городах находится большинство клиентов, пользующихся вашими услугами? Можно написать собственный код для сквозного обхода элементов коллекции и исследования полей каждого объекта, но подобного рода задачи настолько широко востребованы, что разработчики C# решили включить в язык функциональные средства для минимизации обязательного для написания объема кода. В этой главе вы научитесь использовать усовершенствования языка C#, созданные с целью запроса данных и работы с ними.

Что такое LINQ?

Обрабатывать данные приходится всем приложениям, за исключением, может быть, самых примитивных. Исторически сложилось так, что большинство приложений предоставляло для выполнения связанных с этим операций свою собственную логику. Но такая стратегия может привести к тому, что код в приложении станет тесно связанным со структурой обрабатываемых данных. Если структура данных изменится, может потребоваться внесение большого объема изменений в код, обрабатывающий данные. Разработчики среды Microsoft .NET Framework долго и упорно обдумывали пути решения этих проблем и решили облегчить жизнь разработчикам приложений, предоставив функциональные возможности, отделяющие механизм, используемый приложением для запроса данных, от самого кода приложения. Эти возможности были названы Language-Integrated Query (LINQ).

Создатели LINQ непредвзято присмотрелись к способу, используемому реляционными системами управления базами данных, такими как Microsoft SQL Server, отделив язык, используемый для запросов к базе данных, от внутреннего формата данных в базе данных. Разработчики обращались к базе данных SQL Server, выдавая инструкции на языке структурных запросов (Structured Query Language (SQL)). SQL предоставляет высокоуровневое описание данных, которые разработчику нужно извлечь, но не дает точных указаний, как именно система управления базами данных должна извлекать эти данные. Эти тонкости являются прерогативой самой системы управления базами данных. Следовательно, приложение, выдающее SQL-инструкции, не интересуется тем, как система управления базами данных физически сохраняет или извлекает данные. Формат, используемый системой управления базами данных, может изменяться (например, при выпуске новой версии), не требуя от разработчиков приложений внесения изменений в SQL-инструкции, используемые приложениями.

LINQ предоставляет синтаксис и семантику, которые напоминают SQL и во многом обеспечивают такие же преимущества. Основную структуру запрашиваемых данных можно изменять, не испытывая при этом необходимости в изменении кода, фактически выполняющего запросы. Следует иметь в виду: несмотря на то что LINQ-запрос похож на SQL, он намного гибче и способен управлять широким диапазоном логических структур данных. LINQ может обрабатывать данные с иерархической организацией, к примеру, данные, которые могут храниться в XML-документе. Но в этой главе основное внимание уделяется использованию LINQ в реляционной форме.

Использование LINQ в приложении на C#

Возможно, проще всего объяснить использование функциональных средств C#, поддерживающих LINQ, путем проработки ряда простых примеров, основанных на следующих наборах информации о клиентах и адресах (табл. 21.1 и 21.2).

Таблица 21.1

CustomerID (Идентификатор клиента)	FirstName (Имя)	LastName (Фамилия)	CompanyName (Название компании)
1	Kim	Abercrombie	Alpine Ski House
2	Jeff	Hay	Coho Winery
3	Charlie	Herb	Alpine Ski House
4	Chris	Preston	Trey Research
5	Dave	Barnett	Wingtip Toys
6	Ann	Beebe	Coho Winery
7	John	Kane	Wingtip Toys
8	David	Simpson	Trey Research
9	Greg	Chapman	Wingtip Toys
10	Tim	Litton	Wide World Importers

Таблица 21.2

CompanyName (Название компании)	City (Город)	Country (Страна)
Alpine Ski House	Berne	Switzerland
Coho Winery	San Francisco	United States
Trey Research	New York	United States
Wingtip Toys	London	United Kingdom
Wide World Importers	Tetbury	United Kingdom

LINQ-расширению требуется, чтобы данные хранились в структуре данных, реализующей интерфейс `IEnumerable` или `IEnumerable<T>` в соответствии с описанием, которое дано в главе 19 «Перечисляемые коллекции». При этом абсолютно неважно, какая структура используется (массив, `HashSet<T>`, `Queue<T>` или любые другие типы коллекций или даже тип коллекции, определенный вами), при условии, что она будет перечисляемой. Но чтобы ничего не усложнять, примеры в этой главе предполагают, что информация о клиентах и об адресах хранится в массивах `customers` и `addresses`, показанных в следующем примере кода.



ПРИМЕЧАНИЕ В реальном приложении содержимое этих массивов будет наполняться путем считывания данных из файла или из базы данных.

```
var customers = new[] {
    new { CustomerID = 1, FirstName = "Kim", LastName = "Abercrombie",
          CompanyName = "Alpine Ski House" },
    new { CustomerID = 2, FirstName = "Jeff", LastName = "Hay",
          CompanyName = "Coho Winery" },
    new { CustomerID = 3, FirstName = "Charlie", LastName = "Herb",
          CompanyName = "Alpine Ski House" },
    new { CustomerID = 4, FirstName = "Chris", LastName = "Preston",
          CompanyName = "Trey Research" },
    new { CustomerID = 5, FirstName = "Dave", LastName = "Barnett",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 6, FirstName = "Ann", LastName = "Beebe",
          CompanyName = "Coho Winery" },
    new { CustomerID = 7, FirstName = "John", LastName = "Kane",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 8, FirstName = "David", LastName = "Simpson",
          CompanyName = "Trey Research" },
    new { CustomerID = 9, FirstName = "Greg", LastName = "Chapman",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 10, FirstName = "Tim", LastName = "Litton",
          CompanyName = "Wide World Importers" }
};

var addresses = new[] {
    new { CompanyName = "Alpine Ski House", City = "Berne",
          Country = "Switzerland" },
    new { CompanyName = "Coho Winery", City = "San Francisco",
          Country = "United States" },
    new { CompanyName = "Trey Research", City = "New York",
          Country = "United States" },
    new { CompanyName = "Wingtip Toys", City = "London",
          Country = "United Kingdom" },
    new { CompanyName = "Wide World Importers", City = "Tetbury",
          Country = "United Kingdom" }
};
```



ПРИМЕЧАНИЕ В разделах «Выбор данных», «Фильтрация данных», «Упорядочение, группировка и статистическая обработка данных» и «Объединение данных» будут показаны основные возможности и синтаксис, применяемый для запроса данных с помощью методов LINQ. Временами синтаксис будет немного усложняться, но когда вы дойдете до раздела «Использование операторов запросов», то поймете, что запоминать, как весь этот синтаксис работает, совершенно ни к чему. Однако вам будет полезно по крайней мере просмотреть все эти разделы, чтобы получить более полное представление о том, как операторы запросов, предоставляемые C#, справляются со своими задачами.

Выбор данных

Предположим, вам нужно вывести на экран список, состоящий из имен всех клиентов, присутствующих в массиве `customers`. Эту задачу можно выполнить с помощью следующего кода:

```
IEnumerable<string> customerFirstNames =  
    customers.Select(cust => cust.FirstName);  
  
foreach (string name in customerFirstNames)  
{  
    Console.WriteLine(name);  
}
```

Несмотря на весьма скромный размер блока кода, он проделывает большую работу и требует некоторых разъяснений, начинающихся с использования метода `Select` массива `customers`.

С помощью метода `Select` можно извлечь из массива конкретные данные, в нашем случае это всего лишь значение поля `FirstName` каждого элемента массива. Как это работает? Параметром метода `Select` фактически является еще один метод, получающий строку из массива `customers` и возвращающий из этой строки выбранные данные. Для выполнения этой задачи можно определить свой собственный специализированный метод, но, как показано в предыдущем примере, проще всего воспользоваться лямбда-выражением и определить безымянный метод. А теперь следует усвоить три важных обстоятельства:

- ❑ Переменная `cust` является параметром, переданным методу. Параметр `cust` можно воспринимать как псевдоним для каждой строки массива `customers`. Компилятор выводит это из того факта, что метод `Select` вызывается в отношении массива `customers`. Вместо `cust` можно использовать любой допустимый идентификатор C#.
- ❑ На этой стадии метод `Select` не извлекает данные, он просто возвращает перечисляемый объект, который будет получать данные, идентифицированные

методом `Select` при последующем проведении итерации. К этому аспекту LINQ мы еще вернемся в разделе «LINQ и отложенное вычисление».

- Метод `Select` фактически не является методом типа `Array`. Это метод расширения класса `Enumerable`, который находится в пространстве имен `System.Linq` и предоставляет изрядный набор статических методов для запросов объектов, реализующих интерфейс-обобщение `IEnumerable<T>`.

В предыдущем примере метод `Select` массива `customers` используется для создания `IEnumerable<string>`-объекта по имени `customerFirstNames`. (Он имеет тип `IEnumerable<string>`, поскольку метод `Select` возвращает перечисляемую коллекцию имен клиентов, являющихся строками.) Инструкция `foreach` выполняет сквозной обход этой коллекции строк, выводя на экран имена всех клиентов в следующей последовательности:

```
Kim
Jeff
Charlie
Chris
Dave
Ann
John
David
Greg
Tim
```

Теперь можно вывести на экран имя каждого клиента. А как получить имя и фамилию каждого клиента? Эта задача решается немного сложнее. Если изучить определение метода `Enumerable.Select` в пространстве имен `System.Linq` по документации, предоставленной Microsoft Visual Studio 2015, то можно увидеть его таким:

```
public static IEnumerable<TResult> Select<TSource, TResult> (
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)
```

По сути, это свидетельствует о том, что `Select` является методом-обобщением, получающим два параметра с именами `TSource` и `TResult`, а также два обычных параметра с именами `source` и `selector`. Параметр `TSource` является типом коллекции, для которой создается перечисляемый набор результатов (в данном случае объектов `customer`), а `TResult` является типом данных в перечисляемом наборе результатов (в данном случае строковых объектов). Следует напомнить, что `Select` является методом расширения, поэтому параметр `source` фактически ссылается на тип, подвергшийся расширению (обобщенную коллекцию объектов `customer`, реализующих в данном примере интерфейс `IEnumerable`). Параметр `selector` указывает на метод-обобщение, идентифицирующий извлекаемые

поля. (Следует напомнить, что `Func` — это название типа делегата-обобщения в .NET Framework, которым можно воспользоваться для инкапсуляции метода-обобщения, возвращающего результат.) Метод, на который ссылается параметр `selector`, получает параметр типа `TSource` (в данном случае `customer`) и выдает объект типа `TResult` (в данном случае `string`). Значение, возвращаемое методом `Select`, является перечисляемой коллекцией объектов типа `TResult` (и опять это `string`).



ПРИМЕЧАНИЕ Как работают методы расширения и какова роль первого параметра для метода расширения, объясняется в главе 12 «Работа с наследованием».

Из предыдущего абзаца важно понять, что метод `Select` возвращает перечисляемую коллекцию, основанную на едином типе. Если нужно, чтобы нумератор возвращал несколько элементов данных, например имя и фамилию каждого клиента, то имеются как минимум два варианта.

- Можно в методе `Select` объединить имя и фамилию в одну строку:

```
IEnumerable<string> customerNames =
    customers.Select(cust => $"{cust.FirstName} {cust.LastName}");
```

- Можно определить новый тип, заключающий в себе имя и фамилию, и воспользоваться методом `Select` для создания экземпляров этого типа:

```
class FullName
{
    public string FirstName{ get; set; }
    public string LastName{ get; set; }
}
...
IEnumerable<FullName> customerNames =
    customers.Select(cust => new FullName
    {
        FirstName = cust.FirstName,
        LastName = cust.LastName
    });

```

Возможно, второй вариант предпочтительнее, но если это единственный случай использования в вашем приложении типа `FullName`, то лучше будет, наверное, воспользоваться безымянным типом, особенно для единственной операции:

```
var customerNames =
    customers.Select(cust => new { FirstName = cust.FirstName,
                                    LastName = cust.LastName } );
```

Обратите внимание на использование здесь для определения типа перечисляемой коллекции ключевого слова `var`. Типы объектов в коллекции являются безымянными, поэтому конкретный тип объектов в коллекции вам неизвестен.

Фильтрация данных

Используя метод `Select`, можно указать или выдать поля, которые требуется включить в перечисляемую коллекцию. Но кроме этого может потребоваться наложить ограничения на строки, которые будут содержаться в этой перечисляемой коллекции. Предположим, к примеру, что нужно составить список, состоящий только из тех называний компаний в массиве `addresses`, которые находятся на территории США. Для этого можно воспользоваться методом `Where`:

```
IEnumerable<string> usCompanies =
    addresses.Where(addr => String.Equals(addr.Country, "United States"))
        .Select(usComp => usComp.CompanyName);

foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

Синтаксически метод `Where` похож на метод `Select`. Он ожидает получения параметров, определяющих метод, фильтрующий данные в соответствии с указанным критерием. В этом примере используется еще одно лямбда-выражение. Переменная `addr` является псевдонимом для строки в массиве `addresses`, а лямбда-выражение возвращает все строки, в которых поле `Country` соответствует строковому значению «`United States`». Метод `Where` возвращает перечисляемую коллекцию строк, содержащих каждое поле из исходной коллекции. Затем к строкам применяется метод `Select`, чтобы передать из этой перечисляемой коллекции только одно поле `CompanyName` и вернуть еще одну перечисляемую коллекцию строковых объектов. (Переменная `usComp` является псевдонимом для типа каждой строки в перечисляемой коллекции, возвращенной методом `Where`.) Следовательно, типом результата всего этого выражения будет `IEnumerable<string>`. Полезно усвоить такую последовательность операций: сначала для фильтрации строк применяется метод `Where`, затем для указания полей применяется метод `Select`. Инструкция `foreach`, выполняющая сквозной обход элементов этой коллекции, выводит на экран названия следующих компаний:

```
Coho Winery
Trey Research
```

Упорядочение, группировка и статистическая обработка данных

Тем, кто уже знаком с SQL, известно, что этот язык кроме простых выделения и фильтрации позволяет выполнять широкий спектр реляционных операций. Например, можно указать, что данные должны возвращаться в определенном

порядке, можно сгруппировать возвращаемые строки по одному или нескольким ключевым полям, а можно подсчитать суммарные значения на основе строк в каждой группе. LINQ предоставляет такие же функциональные возможности.

Чтобы извлечь данные в определенном порядке, можно воспользоваться методом `OrderBy`. Так же, как и методы `Select` и `Where`, метод `OrderBy` в качестве своего аргумента ожидает предоставления метода. Этот метод определяет выражения, которые нужно использовать для сортировки данных. Например, можно вывести на экран название каждой компании из массива `addresses` в возрастающем порядке:

```
IEnumerable<string> companyNames =
    addresses.OrderBy(addr => addr.CompanyName).Select(comp => comp.CompanyName);

foreach (string name in companyNames)
{
    Console.WriteLine(name);
}
```

Этот блок кода выведет на экран компании, показанные в таблице адресов, в алфавитном порядке:

```
Alpine Ski House
Coho Winery
Trey Research
Wide World Importers
Wingtip Toys
```

Если нужно перечислить данные в убывающем порядке, можно воспользоваться методом `OrderByDescending`. Если данные нужно упорядочить по нескольким ключевым значениям, можно после `OrderBy` или `OrderByDescending` воспользоваться методом `ThenBy` или `ThenByDescending`.

Чтобы сгруппировать данные в соответствии с общими значениями в одном или нескольких полях, можно воспользоваться методом `GroupBy`. В следующем примере показано, как сгруппировать компании в массиве `addresses` по странам:

```
var companiesGroupedByCountry =
    addresses.GroupBy(addr => addr.Country);

foreach (var companiesPerCountry in companiesGroupedByCountry)
{
    Console.WriteLine(
        $"Country: {companiesPerCountry.Key}\t{companiesPerCountry.Count()} 
        companies");
    foreach (var companies in companiesPerCountry)
    {
        Console.WriteLine("\t{companies.CompanyName}");
    }
}
```

Теперь общая схема уже должна быть вам ясна. Метод `GroupBy` ожидает передачи ему метода, указывающего поля, по которым следует группировать данные. Но метод `GroupBy` немного отличается от других методов, рассмотренных ранее.

Интереснее всего отсутствие необходимости использования метода `Select` для выделения полей, попадающих в результат. Перечисляемый набор, возвращаемый `GroupBy`, содержит все поля исходной коллекции, но строки выстроены в набор перечисляемых коллекций на основе поля, определяемого методом, указываемым методу `GroupBy`. Иными словами, результатом работы метода `GroupBy` является перечисляемый набор групп, каждая из которых представляет собой перечисляемый набор строк. В только что показанном примере перечисляемый набор `companiesGroupedByCountry` является набором стран. Элементы этого набора сами по себе являются перечисляемыми коллекциями, содержащими компании для каждой из стран, выстроенные в очередь. Код, выводящий компании в каждой стране, использует для сквозного обхода набора `companiesGroupedByCountry` цикл `foreach`, а для сквозного обхода компаний каждой страны — вложенный цикл `foreach`. Обратите внимание на то, что во внешнем цикле `foreach` доступ к группируемому значению можно получить путем использования поля `Key` каждого элемента, а суммарные данные для каждой группы можно подсчитать, воспользовавшись для этого такими методами, как `Count`, `Max`, `Min` и многими другими. Выводимая кодом примера информация имеет следующий вид:

```
Country: Switzerland 1 companies
    Alpine Ski House
Country: United States 2 companies
    Coho Winery
    Trey Research
Country: United Kingdom 2 companies
    Wingtip Toys
    Wide World Importers
```

Многими из методов, выдающих статистическую информацию, например `Count`, `Max` и `Min`, можно воспользоваться, применяя их непосредственно к результатам работы метода `Select`. Если нужно узнать, сколько компаний находится в массиве `addresses`, можно использовать следующий блок кода:

```
int numberOfCompanies = addresses.Select(addr => addr.CompanyName).Count();
Console.WriteLine($"Number of companies: {numberOfCompanies}");
```

Обратите внимание на то, что результатом работы этих методов является одно скалярное значение, а не перечисляемая коллекция. Предыдущий блок кода выведет на экран следующую информацию:

```
Number of companies: 5
```

Но здесь следует сделать некоторые предостережения. Методы, выдающие статистические итоги, не различают строки в обрабатываемом наборе, содержащие

продублированные значения в выделяемых полях. Строго говоря, это означает, что в предыдущем примере показывается только количество строк в массиве `addresses`, содержащих значение в поле `CompanyName`. Если нужно определить, сколько различных стран упомянуто в данной таблице, может возникнуть соблазн воспользоваться следующим кодом:

```
int numberOfCountries = addresses.Select(addr => addr.Country).Count();
Console.WriteLine($"Number of countries: {numberOfCountries}");
```

Он выведет на экран следующую информацию:

```
Number of countries: 5
```

Но на самом деле в массиве `addresses` содержатся только три страны — так уж вышло, что и United States, и United Kingdom фигурируют в нем дважды. Убрать дубликаты из вычисления можно с помощью метода `Distinct`:

```
int numberOfCountries =
    addresses.Select(addr => addr.Country).Distinct().Count();
Console.WriteLine($"Number of countries: {numberOfCountries}");
```

Вот теперь инструкция `Console.WriteLine` выведет на экран вполне ожидаемый результат:

```
Number of countries: 3
```

Объединение данных

Точно так же, как и язык SQL, расширение LINQ позволяет объединять несколько наборов данных по одному или нескольким общим ключевым полям. В следующем примере показано, как вывести на экран имя и фамилию каждого клиента вместе с названием страны его проживания:

```
var companiesAndCustomers = customers
    .Select(c => new { c.FirstName, c.LastName, c.CompanyName })
    .Join(addresses, custs => custs.CompanyName, addrs => addrs.CompanyName,
        (custs, addrs) => new {custs.FirstName, custs.LastName, addrs.Country });

foreach (var row in companiesAndCustomers)
{
    Console.WriteLine(row);
}
```

Имена и фамилии клиентов доступны в массиве `customers`, но страна для каждой компании, в которой работают клиенты, хранится в массиве `addresses`. Общим ключом для массивов `customers` и `addresses` является название страны. В методе

`Select` указываются интересующие вас поля в массиве `customers` (`FirstName` и `LastName`), а также поле, содержащее общий ключ (`CompanyName`). Для объединения данных, которые были определены методом `Select`, с другой перечисляемой коллекцией используется метод `Join`. В этом методе используются следующие параметры:

- ❑ перечисляемая коллекция, с которой происходит объединение;
- ❑ метод, определяющий общие ключевые поля из данных, которые были определены методом `Select`;
- ❑ метод, определяющий общие ключевые поля, по которым объединяются выбранные данные;
- ❑ метод, указывающий столбцы, требующиеся в перечисляемом результирующем наборе, возвращаемом методом `Join`.

В данном примере метод `Join` объединяет перечисляемую коллекцию, содержащую поля `FirstName`, `LastName` и `CompanyName` из массива `customers`, со строками в массиве `addresses`. Два набора данных объединяются там, где значение в поле `CompanyName` в массиве `customers` соответствует значению в поле `CompanyName` в массиве `addresses`. В получающийся в результате этого набор включаются строки, содержащие поля `FirstName` и `LastName` из массива `customers`, а также поле `Country` из массива `addresses`. Код, выводящий на экран данные из коллекции `companiesAndCustomers`, показывает следующую информацию:

```
{ FirstName = Kim, LastName = Abercrombie, Country = Switzerland }
{ FirstName = Jeff, LastName = Hay, Country = United States }
{ FirstName = Charlie, LastName = Herb, Country = Switzerland }
{ FirstName = Chris, LastName = Preston, Country = United States }
{ FirstName = Dave, LastName = Barnett, Country = United Kingdom }
{ FirstName = Ann, LastName = Beebe, Country = United States }
{ FirstName = John, LastName = Kane, Country = United Kingdom }
{ FirstName = David, LastName = Simpson, Country = United States }
{ FirstName = Greg, LastName = Chapman, Country = United Kingdom }
{ FirstName = Tim, LastName = Litton, Country = United Kingdom }
```



ПРИМЕЧАНИЕ Следует помнить, что коллекции в памяти не являются эквивалентом таблиц в реляционной базе данных и содержащиеся в них данные не подпадают под действие тех же ограничений целостности данных. Вполне допустимо предполагать, что в реляционной базе данных у каждого клиента имеется соответствующая компания и у каждой компании есть собственный уникальный адрес. В коллекциях отсутствует принудительная целостность данных на таком же уровне, значит, в них запросто может оказаться клиент, ссылающийся на компанию, отсутствующую в массиве `addresses`, и в этом же массиве какая-то компания может встречаться несколько раз. В подобных ситуациях получаемые результаты могут быть точными, но неожиданными. Операции объединения работают лучше в том случае, когда понятны взаимоотношения между объединяемыми данными.

Использование операторов запросов

В предыдущих разделах были рассмотрены многие функциональные возможности, доступные для создания запросов к находящимся в памяти данным путем использования методов расширения для класса `Enumerable`, определенного в пространстве имен `System.Linq`. В синтаксисе использовался ряд расширенных свойств языка C#, и получающийся в результате код порой мог быть трудным для понимания и сопровождения. Чтобы избавить вас от этих осложнений, разработчики C# добавили к языку операторы запросов, позволяющие использовать функциональные возможности расширения LINQ, применяя синтаксис, больше похожий на синтаксис языка SQL.

Из примеров, показанных ранее в этой главе, можно было уяснить, что имя каждого клиента извлекается следующим образом:

```
IEnumerable<string> customerFirstNames =
    customers.Select(cust => cust.FirstName);
```

Эту инструкцию можно переделать, используя операторы запросов `from` и `select`:

```
var customerFirstNames = from cust in customers
                           select cust.FirstName;
```

В ходе своей работы компилятор C# преобразует это выражение в соответствующий вызов метода `Select`. Оператор `from` определяет псевдоним для коллекции источника, а оператор `select` указывает поля, извлекаемые с использованием этого псевдонима. В результате получается перечисляемая коллекция имен клиентов. Тем, кто знаком с SQL, следует обратить внимание на то, что оператор `from` ставится перед оператором `select`.

Продолжая в том же духе, для извлечения имени и фамилии каждого клиента можно воспользоваться следующей инструкцией. (Вы можете обратиться к рассмотренному ранее примеру такой же инструкции, основанной на применении метода расширения `Select`.)

```
var customerNames = from cust in customers
                       select new { cust.FirstName, cust.LastName };
```

Для фильтрации данных используется оператор `where`. В следующем примере показано, как из массива `addresses` можно получить названия компаний, находящихся в США:

```
var usCompanies = from a in addresses
                       where String.Equals(a.Country, "United States")
                       select a.CompanyName;
```

Для упорядочения данных используется оператор `orderby`:

```
var companyNames = from a in addresses
                    orderby a.CompanyName
                    select a.CompanyName;
```

Группировка данных осуществляется с помощью оператора `group`:

```
var companiesGroupedByCountry = from a in addresses
                                    group a by a.Country;
```

Заметьте, что здесь по аналогии с рассмотренным ранее примером группировки данных оператор `select` не предоставляется и сквозной обход результатов можно выполнить с помощью такого же кода, как и там:

```
foreach (var companiesPerCountry in companiesGroupedByCountry)
{
    Console.WriteLine(
        $"Country: {companiesPerCountry.Key}\t{companiesPerCountry.Count()} 
        companies");
    foreach (var companies in companiesPerCountry)
    {
        Console.WriteLine($" \t{companies.CompanyName}");
    }
}
```

Функции статистической обработки, например `Count`, можно вызвать в отношении коллекции, возвращенной перечисляемой коллекцией, с помощью следующего кода:

```
int numberOfCompanies = (from a in addresses
                           select a.CompanyName).Count();
```

Заметьте, что выражение заключено в круглые скобки. Если требуется проигнорировать продублированные значения, воспользуйтесь методом `Distinct`:

```
int numberOfCountries = (from a in addresses
                           select a.Country).Distinct().Count();
```



СОВЕТ Зачастую требуется просто подсчитать количество строк в коллекции, а не сумму чисел в полях во всех строках этой коллекции. В таком случае можно метод `Count` вызывать непосредственно в отношении исходной коллекции:

```
int numberOfCompanies = addresses.Count();
```

Оператор `join` можно использовать для объединения двух коллекций по общему ключу. В следующем примере показан запрос, возвращающий данные из массивов `customers` и `addresses` на основе столбца `CompanyName` каждой коллекции, переделанный в этот раз под использование оператора `join`. Для указания характера связи двух коллекций с оператором `equals` используется условие `on`.



ПРИМЕЧАНИЕ В настоящее время в LINQ поддерживается только объединение equi-joins (объединение на основе равенства). Тому, кто занимался разработкой баз данных и привык к использованию SQL, могут быть знакомы объединения на основе других операторов, таких как `>` и `<`, но в LINQ подобные функциональные возможности не предоставляются.

```
var countriesAndCustomers = from a in addresses
                             join c in customers
                             on a.CompanyName equals c.CompanyName
                             select new { c.FirstName, c.LastName, a.Country };
```



ПРИМЕЧАНИЕ В отличие от SQL, порядок выражений в условии `on` LINQ-выражения играет важную роль. Объединяемый элемент из другой коллекции (ссылающийся на данные в коллекции в условии `from`) должен находиться слева от оператора `equals`, а элемент, с которым происходит объединение (ссылающийся на данные в коллекции в условии `join`), — справа от него.

Для получения сводной информации и объединения, а также для группировки и ведения поиска в данных расширением LINQ предоставляется большое количество других методов. В этом разделе были рассмотрены лишь наиболее востребованные из них. К примеру, в LINQ предоставляются методы `Intersect` и `Union`, которые могут использоваться для выполнения широкого спектра операций. Также в этом расширении предоставляются такие методы, как `Any` и `All`, которые можно использовать для определения наличия хотя бы одного элемента в коллекции или соответствия всех элементов коллекции определенному предикату. Значения в перечисляемой коллекции можно разбить на части, используя методы `Take` и `Skip`. Дополнительную информацию обо всех этих методах можно найти в материалах в разделе «LINQ» документации, предоставляемой средой Visual Studio 2015.

Запрос данных в объектах `Tree<TItem>`

В рассмотренных ранее примерах главы было показано, как данные запрашиваются из массива. Точно такую же технологию можно применять для любого класса коллекции, в котором реализован интерфейс-обобщение `IEnumerable<T>`. В следующем упражнении будет определен новый класс для моделирования работников компании. Вам будет создан объект `BinaryTree`, содержащий коллекцию объектов типа `Employee`, а затем для запроса нужной информации будет использовано расширение LINQ. Сначала методы расширения LINQ станут вызываться напрямую, а затем код будет изменен под использование операторов запросов.

Извлечение данных из `BinaryTree` с помощью методов расширения

Откройте в среде Visual Studio 2015 решение `QueryBinaryTree`, которое находится в папке `\Microsoft Press\VCBS\Chapter 21\QueryBinaryTree` вашей папки документов.

В проекте имеется файл Program.cs, в котором определяется класс `Program` с методами `Main` и `doWork`, которые уже встречались в предыдущих упражнениях.

В обозревателе решений щелкните правой кнопкой мыши на проекте `QueryBinaryTree`, укажите на пункт **Добавить** и щелкните на пункте **Класс**. Наберите в поле **Имя** диалогового окна **Добавить новый элемент — Query BinaryTree** строку `Employee.cs` и щелкните на кнопке **Добавить**.

Добавьте к классу `Employee` автоматически создаваемые свойства, выделенные в следующем фрагменте кода жирным шрифтом:

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Department { get; set; }
    public int Id { get; set; }
}
```

Добавьте к классу `Employee` выделенный в следующем примере кода жирным шрифтом метод `ToString`. Типы в среде .NET Framework используют этот метод при преобразовании объекта в строковое представление точно так же, как это делается при его выводе на экран с помощью инструкции `Console.WriteLine`:

```
class Employee
{
    ...
    public override string ToString()
    {
        return
            $"Id: {this.Id}, Name: {this.FirstName} {this.LastName}, Dept: {this.
                Department}";
    }
}
```

Измените определение класса `Employee` так, чтобы в нем реализовывался интерфейс `IComparable<Employee>`:

```
class Employee : IComparable<Employee>
{
}
```

Необходимость этого объясняется тем, что класс `BinaryTree` указывает на то, что его элементы должны быть сравниваемыми.

Наведите указатель мыши на интерфейс `IComparable<Employee>` в определении класса, щелкните на появившемся значке с горящей лампочкой, а затем щелкните в контекстном меню на пункте **Реализовать интерфейс явно**. В результате будет создана исходная реализация метода `CompareTo`. Следует напомнить, что класс `BinaryTree` вызывает этот метод, когда ему нужно сравнить элементы при вставке их в дерево.

Замените тело метода `CompareTo` следующим кодом, выделенным жирным шрифтом. Эта реализация метода `CompareTo` сравнивает объекты типа `Employee` на основе значения поля `Id`:

```
int IComparable<Employee>.CompareTo(Employee other)
{
    if (other == null)
    {
        return 1;
    }

    if (this.Id > other.Id)
    {
        return 1;
    }

    if (this.Id < other.Id)
    {
        return -1;
    }
    return 0;
}
```



ПРИМЕЧАНИЕ Описание интерфейса `IComparable<T>` дается в главе 19.

В обозревателе решений щелкните правой кнопкой мыши на решении `QueryBinaryTree`, укажите на пункт Добавить и щелкните на пункте Существующий проект. В диалоговом окне Добавить существующий проект перейдите в папку `Microsoft Press\VCBS\Chapter 21\BinaryTree` вашей папки документов, щелкните на проекте `BinaryTree`, а затем на кнопке Открыть. В проекте `BinaryTree` содержится копия перечисляемого класса `BinaryTree`, который был реализован в главе 19.

В обозревателе решений щелкните правой кнопкой мыши на проекте `QueryBinaryTree`, укажите на пункт Добавить и щелкните на пункте Ссылка. Щелкните в левой панели диалогового окна Менеджер ссылок — `QueryBinaryTree` на пункте Решение. Установите в средней панели флажок напротив проекта `BinaryTree`, а затем щелкните на кнопке OK.

Выполните в окне редактора файл `Program.cs` проекта `QueryBinaryTree` и убедитесь в том, что в список директив `using` в начале файла включена следующая строка кода:

```
using System.Linq;
```

Добавьте в список в начале файла `Program.cs` директиву `using`, чтобы поместить в область видимости пространство имен `BinaryTree`:

```
using BinaryTree;
```

Удалите из метода doWork в классе Program комментарий // TODO: и добавьте инструкции, выделенные в следующем примере кода жирным шрифтом, чтобы создать и заполнить экземпляр класса BinaryTree:

```
static void doWork()
{
    Tree<Employee> empTree = new Tree<Employee>(
        new Employee { Id = 1, FirstName = "Kim", LastName = "Abercrombie",
                      Department = "IT"});
    empTree.Insert(
        new Employee { Id = 2, FirstName = "Jeff", LastName = "Hay",
                      Department = "Marketing"});
    empTree.Insert(
        new Employee { Id = 4, FirstName = "Charlie", LastName = "Herb",
                      Department = "IT"});
    empTree.Insert(
        new Employee { Id = 6, FirstName = "Chris", LastName = "Preston",
                      Department = "Sales"});
    empTree.Insert(
        new Employee { Id = 3, FirstName = "Dave", LastName = "Barnett",
                      Department = "Sales"});
    empTree.Insert(
        new Employee { Id = 5, FirstName = "Tim", LastName = "Litton",
                      Department="Marketing"
    });
}
```

Добавьте к концу метода doWork следующие выделенные жирным шрифтом инструкции. Этот код вызывает метод Select для вывода списка подразделений, найденных в двоичном дереве.

```
static void doWork()
{
    ...
    Console.WriteLine("List of departments");
    var depts = empTree.Select(d => d.Department);

    foreach (var dept in depts)
    {
        Console.WriteLine($"Department: {dept}");
    }
}
```

Щелкните в меню Отладка на пункте Запуск без отладки.

Приложение должно вывести на экран следующий список подразделений:

```
List of departments
Department: IT
Department: Marketing
Department: Sales
Department: IT
Department: Marketing
Department: Sales
```

Каждое подразделение фигурирует дважды, поскольку в каждом подразделении по два работника. Порядок следования подразделений определяется методом `CompareTo` класса `Employee`, который использует для сортировки данных свойство `Id` каждого работника. Первое подразделение показано для работника с `Id`, имеющим значение 1, второе подразделение показано для работника с `Id`, имеющим значение 2, и т. д.

Нажмите клавишу **Ввод**, чтобы вернуться в среду Visual Studio 2015.

Измените в методе `doWork` класса `Program` инструкцию, создающую перечисляемую коллекцию подразделений, выделенную в данном примере жирным шрифтом:

```
var depts = empTree.Select(d => d.Department).Distinct();
```

Метод `Distinct` удаляет из перечисляемой коллекции дубликаты строк.

Щелкните в меню Отладка на пункте Запуск без отладки.

Убедитесь в том, что теперь приложение выводит подразделения только по одному разу:

```
List of departments  
Department: IT  
Department: Marketing  
Department: Sales
```

Нажмите клавишу **Ввод**, чтобы вернуться в среду Visual Studio 2015.

Добавьте к концу метода `doWork` следующие инструкции, показанные далее жирным шрифтом. В этом блоке кода для фильтрации работников и возвращения только тех из них, кто работает в IT-подразделении, используется метод `Where`. Метод `Select` возвращает всю строку, не выделяя конкретные столбцы:

```
static void doWork()  
{  
    ...  
    Console.WriteLine("\nEmployees in the IT department");  
    var ITEmployees =  
        empTree.Where(e => String.Equals(e.Department, "IT"))  
        .Select(emp => emp);  
  
    foreach (var emp in ITEmployees)  
    {  
        Console.WriteLine(emp);  
    }  
}
```

После показанного выше кода добавьте к концу метода `doWork` следующий код, выделенный здесь жирным шрифтом. В этом коде для группировки работников,

найденных в двоичном дереве по подразделению, используется метод `GroupBy`. Внешняя инструкция `foreach` выполняет сквозной обход каждой группы, выводя на экран название подразделения. Внутренняя инструкция `foreach` выводит на экран имена работников каждого подразделения:

```
static void doWork()
{
    ...
    Console.WriteLine("\nAll employees grouped by department");
    var employeesByDept = empTree.GroupBy(e => e.Department);

    foreach (var dept in employeesByDept)
    {
        Console.WriteLine($"Department: {dept.Key}");
        foreach (var emp in dept)
        {
            Console.WriteLine($"{emp.FirstName} {emp.LastName}");
        }
    }
}
```

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что приложение выводит на экран следующие данные:

```
List of departments
Department: IT
Department: Marketing
Department: Sales

Employees in the IT department
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 4, Name: Charlie Herb, Dept: IT

All employees grouped by department
Department: IT
    Kim Abercrombie
    Charlie Herb
Department: Marketing
    Jeff Hay
    Tim Litton
Department: Sales
    Dave Barnett
    Chris Preston
```

Нажмите Ввод, чтобы вернуться в среду Visual Studio 2015.

Извлечение данных из BinaryTree с помощью операторов запросов

Закомментируйте в методе `doWork` инструкцию, создающую коллекцию подразделений, и вставьте вместо нее эквивалентную инструкцию, выделенную жирным шрифтом, в которой используются операторы запросов `from` и `select`:

```
// var depts = empTree.Select(d => d.Department).Distinct();
var depts = (from d in empTree
             select d.Department).Distinct();
```

Закомментируйте инструкцию, создающую коллекцию работников ИТ-подразделения, и вставьте следующий код, выделенный жирным шрифтом:

```
// var ITEmployees =
//     empTree.Where(e => String.Equals(e.Department, "IT"))
//     .Select(emp => emp);
var ITEmployees = from e in empTree
                  where String.Equals(e.Department, "IT")
                  select e;
```

Закомментируйте инструкцию, создающую коллекцию группирующую работников по подразделениям, и вставьте вместо нее инструкции, выделенные в следующем коде жирным шрифтом:

```
// var employeesByDept = empTree.GroupBy(e => e.Department);
var employeesByDept = from e in empTree
                      group e by e.Department;
```

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что приложение выводит на экран тот же результат, что и прежде:

```
List of departments
Department: IT
Department: Marketing
Department: Sales

Employees in the IT department
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 4, Name: Charlie Herb, Dept: IT

All employees grouped by department
Department: IT
    Kim Abercrombie
    Charlie Herb
Department: Marketing
    Jeff Hay
    Tim Litton
Department: Sales
    Dave Barnett
    Chris Preston
```

Нажмите Ввод, чтобы вернуться в среду Visual Studio 2015.

LINQ и отложенное вычисление

Когда расширение LINQ применяется для определения перечисляемой коллекции путем использования либо методов расширения LINQ, либо операторов запросов, следует помнить, что приложение фактически не создает коллекцию

в ходе выполнения метода расширения LINQ — коллекция перечисляется, только когда выполняется ее обход. Это означает, что за время между выполнением LINQ-запроса и извлечением данных, определяемых запросом, данные в исходной коллекции могут измениться: извлекаться неизменно будут наиболее свежие данные. Например, следующий запрос (который уже был показан ранее) определяет перечисляемую коллекцию компаний, находящихся в США:

```
var usCompanies = from a in addresses
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

Данные из массива `addresses` не извлекаются, и никакие условия, указанные в фильтре `Where`, не вычисляются, пока не будет выполняться сквозной обход элементов коллекции `usCompanies`:

```
foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

Если изменить данные в массиве `addresses` в период между определением коллекции `usCompanies` и сквозным обходом коллекции (например, если добавить новую компанию, находящуюся в США), эти новые данные станут видны. Такая стратегия называется отложенным вычислением.

Если LINQ-запрос определяет и создает статическую кэшируемую коллекцию, вычисление можно сделать вынужденным. Получаемая в результате коллекция является копией исходных данных и не будет меняться при изменении данных в исходной коллекции. Для создания статического `List`-объекта расширение LINQ предоставляет метод `ToList`, содержащий кэшированную копию данных. Она используется следующим образом:

```
var usCompanies = from a in addresses.ToList()
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

На этот раз список компаний фиксируется при создании запроса. Если вы добавляете к массиву `address` еще больше компаний из США, то их при сквозном обходе элементов коллекции `usCompanies` видно не будет. Расширение LINQ также предоставляет метод `ToArrayList`, сохраняющий кэшированную коллекцию в массиве.

В заключительном упражнении этой главы будет выполнено сравнение эффектов использования отложенного вычисления LINQ-запроса с эффектами создания кэшированной коллекции.

Изучение эффектов отложенного и кэшированного вычисления LINQ-запроса

Выведите в окно редактора Visual Studio 2015 файл Program.cs проекта QueryBinaryTree. Закомментируйте часть содержимого метода doWork, кроме инструкций, создающих двоичное дерево empTree:

```
static void doWork()
{
    Tree<Employee> empTree = new Tree<Employee>(
        new Employee { Id = 1, FirstName = "Kim", LastName = "Abercrombie",
                      Department = "IT"});
    empTree.Insert(
        new Employee { Id = 2, FirstName = "Jeff", LastName = "Hay",
                      Department = "Marketing"});
    empTree.Insert(
        new Employee { Id = 4, FirstName = "Charlie", LastName = "Herb",
                      Department = "IT"});
    empTree.Insert(
        new Employee { Id = 6, FirstName = "Chris", LastName = "Preston",
                      Department = "Sales"});
    empTree.Insert(
        new Employee { Id = 3, FirstName = "Dave", LastName = "Barnett",
                      Department = "Sales"});
    empTree.Insert(
        new Employee { Id = 5, FirstName = "Tim", LastName = "Litton",
                      Department = "Marketing"});
    // Закомментируйте всю остальную часть метода
    ...
}
```



СОВЕТ Закомментировать блок кода можно, выбрав весь блок в окне редактора и щелкнув на кнопке панели инструментов. Закомментировать выделенные строки можно, нажав клавишу **Ctrl** и, удерживая ее, последовательно нажав клавиши **K** и **C**.

Добавьте к методу doWork сразу после кода, создающего и заполняющего двоичное дерево empTree, следующий код, выделенный жирным шрифтом:

```
static void doWork()
{
    ...
    Console.WriteLine("All employees");
    var allEmployees = from e in empTree
                      select e;

    foreach (var emp in allEmployees)
    {
        Console.WriteLine(emp);
    }
    ...
}
```

Этот код создает перечисляемую коллекцию работников `allEmployees`, а затем выполняет сквозной обход элементов этой коллекции, выводя на экран сведения, касающиеся каждого работника.

Добавьте сразу же после только что набранных инструкций следующий код:

```
static void doWork()
{
    ...
    empTree.Insert(new Employee
    {
        Id = 7,
        FirstName = "David",
        LastName = "Simpson",
        Department = "IT"
    });
    Console.WriteLine("\nEmployee added");

    Console.WriteLine("All employees");
    foreach (var emp in allEmployees)
    {
        Console.WriteLine(emp);
    }
    ...
}
```

Инструкции этого кода добавляют к дереву `empTree` нового работника, а затем еще раз выполняют сквозной обход элементов коллекции `allEmployees`.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что приложение выводит следующие данные:

```
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales

Employee added
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
Id: 7, Name: David Simpson, Dept: IT
```

Обратите внимание на то, что при втором сквозном обходе элементов коллекции `allEmployees` выводимый на экран список включает работника `David`

Simpson, даже при том что он был добавлен только после определения коллекции `allEmployees`.

Нажмите Ввод, чтобы вернуться в среду Visual Studio 2015.

Внесите изменения, показанные жирным шрифтом, в метод `doWork`, в инструкцию, создающую коллекцию `allEmployees`, чтобы происходило определение и немедленное кэширование данных:

```
var allEmployees = from e in empTree.ToList<Employee>()
                    select e;
```

Расширение LINQ предоставляет обобщенные и необобщенные версии методов `ToList` и `ToArray`. По возможности лучше использовать обобщенные версии этих методов, чтобы гарантировать безопасность результата в отношении типов. Данные, возвращаемые оператором `select`, являются `Employee`-объектом, а только что показанный код создает коллекцию-обобщение `allEmployees` типа `List<Employee>`.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что приложение выводит следующие данные:

```
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
```

```
Employee added
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
```

Обратите внимание на то, что при втором сквозном обходе элементов коллекции `allEmployees` в выведенном на экран списке отсутствует работник `David Simpson`. В данном случае запрос был вычислен, а результат кэширован до того, как `David Simpson` был добавлен к двоичному дереву `empTree`.

Нажмите Ввод, чтобы вернуться в среду Visual Studio 2015.

Выводы

В этой главе были изучены приемы использования расширением LINQ интерфейса `IEnumerable<T>`, а также применение методов расширения для предоставления механизма запроса данных. Вы также увидели, как эти функциональные средства поддерживают синтаксис выражений запросов в C#.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 22 «Перегрузка операторов».

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Выделить указанные поля из перечисляемой коллекции	<p>Воспользуйтесь методом <code>Select</code> и укажите лямбда-выражение, определяющее выделяемые поля, например:</p> <pre>var customerFirstNames = customers.Select(cust => cust.FirstName);</pre> <p>Или же воспользуйтесь операторами запросов <code>from</code> и <code>select</code>, например:</p> <pre>var customerFirstNames = from cust in customers select cust.FirstName;</pre>
Отфильтровать строки из перечисляемой коллекции	<p>Воспользуйтесь методом <code>Where</code> и укажите лямбда-выражение, содержащее критерий, которому должны соответствовать строки, например:</p> <pre>var usCompanies = addresses.Where(addr => String.Equals(addr.Country, «United States»)) .Select(usComp => usComp.CompanyName);</pre> <p>Или же воспользуйтесь оператором запросов <code>where</code>, например:</p> <pre>var usCompanies = from a in addresses where String.Equals(a.Country, «United States») select a.CompanyName;</pre>
Перечислить данные в определенном порядке	Воспользуйтесь методом <code>OrderBy</code> и укажите лямбда-выражение, определяющее поле, используемое для упорядочения строк, например:

Чтобы	Сделайте следующее
	<pre>var companyNames = addresses.OrderBy(addr => addr.CompanyName) .Select(comp => comp.CompanyName); Или же воспользуйтесь оператором запросов orderby, например:</pre> <pre>var companyNames = from a in addresses orderby a.CompanyName select a.CompanyName;</pre>
Сгруппировать данные по значению в поле	<p>Воспользуйтесь методом GroupBy и укажите лямбда-выражение, определяющее поле, используемое для группировки строк, например:</p> <pre>var companiesGroupedByCountry = addresses.GroupBy(addr => addr.Country);</pre> <p>Или же воспользуйтесь оператором запросов group by, например:</p> <pre>var companiesGroupedByCountry = from a in addresses group a by a.Country;</pre>
Объединить данные, хранящиеся в двух разных коллекциях	<p>Воспользуйтесь методом Join, указав коллекцию, с которой нужно объединиться, критерий объединения и поля для вывода результата, например:</p> <pre>var countriesAndCustomers = customers .Select(c => new { c.FirstName, c.LastName, c.CompanyName }). Join(addresses, custs => custs.CompanyName, addrs => addrs.CompanyName, (custs, addrs) => new { custs.FirstName, custs.LastName, addrs.Country }); Или же воспользуйтесь оператором запросов join, например:</pre> <pre>var countriesAndCustomers = from a in addresses join c in customers on a.CompanyName equals c.CompanyName select new { c.FirstName, c.LastName, a.Country };</pre>
Добиться от LINQ-запроса немедленного создания результата	<p>Воспользуйтесь методом ToList или методом ToArray для создания списка или массива, содержащего результаты, например:</p> <pre>var allEmployees = from e in empTree.ToList<Employee>() select e;</pre>

22 Перегрузка операторов

Прочитав эту главу, вы научитесь:

- реализовывать для своих собственных типов бинарные операторы;
- реализовывать для своих собственных типов унарные операторы;
- создавать для своих собственных типов операторы инкремента и декремента;
- осознавать необходимость реализации некоторых операторов в виде пар;
- реализовывать для своих собственных типов неявную конверсию операторов;
- реализовывать для своих собственных типов явную конверсию операторов.

В примерах книги для выполнения стандартных операций (например, сложения и вычитания) широко использовались стандартные символы операторов (например, + и -). Многие встроенные типы поставляются для каждого оператора со своим предопределенным поведением. Можно также определить, как операторы должны себя вести с вашими собственными структурами и классами. Именно эта тема и будет рассмотрена в данной главе.

Общие сведения об операторах

Перед тем как углубиться в подробности работы операторов и способы их перегрузки, стоит вспомнить об основных аспектах, кратко перечисленных в следующем списке.

- Операторы используются для объединения operandов в выражениях. У каждого оператора в зависимости от типа, с которым он работает, имеется

собственная семантика. Например, когда используются числовые типы, оператор `+` означает «сложить», а когда строки — «объединить».

- ❑ У каждого оператора есть приоритет. Например, приоритет оператора `*` выше приоритета оператора `+`. Это означает, что выражение `a + b *` с является эквивалентом выражения `a + (b * c)`.
- ❑ Для каждого оператора определена ассоциативность, задающая направление связанного с ним вычисления: слева направо или справа налево. Например, оператор `=` имеет правую ассоциативность (выражение с ним вычисляется справа налево), следовательно, выражение `a = b = c` эквивалентно выражению `a = (b = c)`.
- ❑ Оператор, имеющий только один operand, называется унарным. Например, к унарным относится оператор инкремента (`++`).
- ❑ Оператор, имеющий два операнда, называется бинарным. Например, к бинарным относится оператор умножения (`*`).

Ограничения, накладываемые на операторы

В этой книге представлено множество примеров того, как в C# при определении собственных типов можно перегружать методы. В C# для своих собственных типов можно также перегружать многие из существующих символов операторов, хотя синтаксис при этом может немного различаться. При такой перегрузке реализуемые вами операторы автоматически попадают в четко определенную среду со следующими правилами.

- ❑ Изменить приоритет или ассоциативность оператора невозможно, поскольку они определяются символом оператора (например, `+`), а не типом, в отношении которого используется этот символ оператора (например, `int`). Следовательно, выражение `a + b *` с всегда будет эквивалентно выражению `a + (b * c)` независимо от типов `a`, `b` и `c`.
- ❑ Нельзя изменять количество operandов оператора. Например, `*` (символ умножения) является бинарным оператором. Если объявлять оператор `*` для собственного типа, он должен быть бинарным оператором.
- ❑ Нельзя изобретать новые символы операторов. Например, для возведения числа в степень другого числа нельзя создать такой оператор, как `**`. Для этого нужно определить метод.
- ❑ Нельзя изменять значение операторов, когда они применяются ко встроенным типам. Например, выражение `1 + 2` имеет предопределенное значение, переопределять которое не разрешается. Если бы такое было возможно, то все бы чрезвычайно усложнилось.

- ❑ Есть ряд символов операторов, не подлежащих перегрузке. Например, нельзя перегрузить оператор точки (.), который служит признаком доступа к компоненту класса. Если бы и это было возможно, то возникли бы ненужные осложнения.



СОВЕТ Для имитации в качестве оператора квадратных скобок [] можно использовать индексаторы. По аналогии с этим для имитации в качестве оператора знака равенства (=) можно использовать свойства, а для имитации в качестве оператора вызова функции — воспользоваться делегатами.

Перегруженные операторы

Чтобы определить выбранный оператор, задав ему нужное вам поведение, нужно его перегрузить. При этом используется синтаксис, похожий на синтаксис метода, с возвращаемым значением и параметрами, но в качестве имени метода используются ключевое слово `operator` и символ объявляемого оператора. Например, следующий код показывает определяемую пользователем структуру по имени `Hour`, в которой определяется бинарный оператор + для сложения двух экземпляров `Hour`:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }
    ...
    private int value;
}
```

Обратите внимание на следующие обстоятельства.

- ❑ Оператор является открытым (`public`). Все операторы должны быть открытыми.
- ❑ Оператор является статическим (`static`). Все операторы должны быть статическими. К операторам никогда не применяется полиморфизм, и в отношении них не могут использоваться модификаторы `virtual`, `abstract`, `override` или `sealed`.
- ❑ У бинарного оператора (такого, как +, показанный в данном примере) имеются два явно указанных аргумента, а у унарного оператора имеется один

явно указанный аргумент. (Программисты, работавшие на C++, должны заметить, что у операторов параметр `this` никогда не скрывается.)



СОВЕТ При объявлении сильно стилизованных функциональных средств, таких как операторы, полезно внедрить для параметров соглашение о присваивании имен. Например, для бинарных операторов разработчики часто используют имена `lhs` и `rhs` (акронимы для операнда, находящегося слева от оператора (*left-hand side*), и операнда, находящегося справа от него (*right-hand side*), соответственно).

При использовании оператора `+` в отношении двух выражений типа `Hour` компилятор C# автоматически превращает ваш код в вызов метода вашего оператора `+`. Компилятор C# превращает код

```
Hour Example(Hour a, Hour b)
{
    return a + b;
}
```

в такой:

```
Hour Example(Hour a, Hour b)
{
    return Hour.operator +(a,b); // Псевдокод
}
```

Но следует заметить, что этот синтаксис является псевдокодом, недопустимым в C#. Бинарные операторы можно использовать только с их стандартной инфиксной системой записи (с символом между операндами).

Существует еще одно, итоговое правило, которого нужно придерживаться при объявлении оператора: как минимум один параметр должен содержать тип. В предыдущем примере `operator +` для класса `Hour` один из параметров, `a` или `b`, должен быть `Hour`-объектом. В данном примере оба параметра относятся к типу `Hour`. Но случается, что нужно определить дополнительные реализации `operator +` для сложения, к примеру, целого числа (количество часов) с объектом `Hour` — у первого параметра должен быть тип `Hour`, а второй параметр может быть целым числом. Если выполняется это правило, компилятору проще узнать, на что нужно ориентироваться при попытке разрешения вызова оператора, а также гарантировать невозможность изменения значения встроенных операторов.

Создание симметричных операторов

В предыдущем разделе было показано, как объявляется бинарный оператор `+` для сложения двух экземпляров типа `Hour`. У структуры `Hour` имеется также конструктор, создающий `Hour` из `int`. Это означает возможность сложения `Hour` и `int`, просто нужно сначала, как показано в следующем примере, воспользоваться `Hour`-конструктором для преобразования `int` в `Hour`:

```
Hour a = ...;
int b = ...;
Hour sum = a + new Hour(b);
```

Этот код заведомо допустим, но он менее понятен или выразителен, чем непосредственное сложение `Hour` и `int`:

```
Hour a = ...;
int b = ...;
Hour sum = a + b;
```

Чтобы легализовать выражение `a + b`, нужно указать, что оно означает сложение `Hour` (левого операнда `a`) с `int` (правым операндом `b`). Иными словами, нужно объявить бинарный оператор `+`, чьим первым параметром будет тип `Hour`, а вторым — тип `int`. Рекомендуемый подход показан в следующем примере кода:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }
    public static Hour operator +(Hour lhs, int rhs)
    {
        return lhs + new Hour(rhs);
    }
    ...
    private int value;
}
```

Заметьте, что во второй версии оператора всего лишь создается значение типа `Hour` из аргумента, имеющего тип `int`, с последующим вызовом первой версии. Таким образом, реальная логика оператора содержится в одном месте. Главное здесь то, что дополнительный оператор `+` просто упрощает использование существующих функциональных возможностей. Также следует заметить, что вам не нужно предоставлять множество различных версий этого оператора, у каждой из которых будет другой второй параметр, — вместо этого подстройтесь только под наиболее распространенные и значимые случаи применения и позвольте пользователю класса предпринять любые дополнительные шаги в том случае, когда ему придется иметь дело с необычными вариантами.

Оператор `+` объявляет, как сложить значение типа `Hour`, фигурирующее в качестве левого операнда, со значением типа `int`, фигурирующим в качестве правого операнда. В нем не объявляется, как сложить значение типа `int`, фигурирующее

в качестве левого операнда, и значение типа `Hour`, фигурирующее в качестве правого операнда:

```
int a = ...;
Hour b = ...;
Hour sum = a + b; // Ошибка в ходе компиляции
```

Это противоречит здравому смыслу. Если можно написать выражение `a + b`, то вполне ожидаемо, что можно написать и выражение `b + a`. Поэтому вы должны предоставить еще одну перегрузку оператора `+`:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator +(Hour lhs, int rhs)
    {
        return lhs + new Hour(rhs);
    }
    public static Hour operator +(int lhs, Hour rhs)
    {
        return new Hour(lhs) + rhs;
    }
    ...
    private int value;
}
```



ПРИМЕЧАНИЕ Программистам, работавшим с языком C++, следует заметить, что перегрузку вы должны предоставлять самостоятельно. Компилятор не станет создавать код за вас или молча менять порядок следования двух operandов, чтобы найти подходящий оператор.

ОПЕРАТОРЫ И ЯЗЫКОВАЯ СОВМЕСТИМОСТЬ

Перегрузка операторов поддерживается или понимается не всеми языками, выполнение программ на которых осуществляется в общеязыковой среде выполнения (common language runtime (CLR)). Чтобы мог использоваться класс из языков, не поддерживающих перегрузку операторов, при их перегрузке нужно предоставлять альтернативный механизм, реализующий такую же функциональную возможность. Предположим, к примеру, что вы реализуете оператор `+` для структуры `Hour` следующим образом:

```
public static Hour operator +(Hour lhs, int rhs)
{
    ...
}
```

Если нужно предоставить возможность использования вашего класса из приложения, написанного на языке Microsoft Visual Basic, вы также должны предоставить метод Add, выполняющий ту же самую задачу:

```
public static Hour Add(Hour lhs, int rhs)
{
    ...
}
```

Основные сведения о вычислении составного присваивания

Оператор составного присваивания (например, `+=`) всегда вычисляется в понятиях его связанности с простым оператором (например, `+`). Иными словами, инструкция

```
a += b;
```

автоматически вычисляется в

```
a = a + b;
```

В общем виде выражение `a @= b`, где символом @ представлен любой допустимый оператор, всегда вычисляется в `a = a @ b`. Если вы перегрузили соответствующий простой оператор, то при использовании соответствующего ему составного оператора присваивания автоматически вызывается перегруженная версия:

```
Hour a = ...;
int b = ...;
a += a; // То же самое, что и a = a + a
a += b; // То же самое, что и a = a + b
```

Первое выражение составного присваивания `a += a` вполне допустимо, поскольку `a` относится к типу `Hour`, а в типе `Hour` объявляется бинарный оператор `+`, оба параметра которого относятся к типу `Hour`. Аналогично этому вполне допустимо и второе выражение составного присваивания `a += b`, поскольку `a` относится к типу `Hour`, а `b` — к типу `int`. В типе `Hour` также объявляется бинарный оператор `+`, чьим первым параметром является `Hour`-переменная, а вторым — `int`-переменная. Но следует иметь в виду, что вы не можете написать выражение `b += a`, поскольку это то же самое, что написать `b = b + a`. Хотя сложение допустимо, но присваивание недопустимо, поскольку способов присваивания значения типа `Hour` переменной, относящейся к встроенному типу `int`, не существует.

Объявление операторов инкремента и декремента

В C# допустимо объявление своих собственных версий операторов инкремента (++) и декремента (--). При объявлении этих операторов нужно применять обычные правила: они должны быть открытыми, статическими и унарными — получающими только один параметр. Оператор инкремента для структуры `Hour` объявляется следующим образом:

```
struct Hour
{
    ...
    public static Hour operator ++(Hour arg)
    {
        arg.value++;
        return arg;
    }
    ...
    private int value;
}
```

Операторы инкремента и декремента уникальны тем, что их можно использовать в префиксной и постфиксной форме. Языку C# присуще разумное использование одного и того же оператора как для префиксной, так и для постфиксной версии. Результатом постфиксного выражения является определение значения операнда перед тем, как будет вычислено само выражение. Иными словами, компилятор фактически преобразует код

```
Hour now = new Hour(9);
Hour postfix = now++;
```

в код

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // Псевдокод, недопустимый в C#
```

Результатом префиксного выражения будет значение, возвращаемое оператором, следовательно, компилятор C# фактически преобразует код

```
Hour now = new Hour(9);
Hour prefix = ++now;
```

в код

```
Hour now = new Hour(9);
now = Hour.operator ++(now); // Псевдокод, недопустимый в C#
Hour prefix = now;
```

Эта эквивалентность означает, что тип возвращаемого значения операторов инкремента и декремента должен совпадать с типом параметра.

Операторы сравнения в структурах и классах

Следует иметь в виду, что реализация оператора инкремента в структуре `Hour` работает только потому, что `Hour` является структурой. Если сделать `Hour` классом, но оставить реализацию его оператора инкремента без изменений, то обнаружится, что постфиксная трансляция не дает правильного ответа. Вы, наверное, помните, что класс является ссылочным типом, и если еще раз взглянуть на рассмотренную ранее трансляцию, выполняемую компилятором, то в следующем примере вы сможете понять, почему операторы для класса `Hour` больше не работают так, как от них ожидалось:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // Псевдокод, недопустимый в C#
```

Если `Hour` является классом, то инструкция присваивания `postfix =` теперь заставляет переменную `postfix` ссылаться на тот же объект, на который ссылается переменная `now`. Обновление `now` автоматически приведет к обновлению `postfix`! Если `Hour` является структурой, то инструкция присваивания делает копию `now` в `postfix` и любые изменения `now` не приводят к изменению `postfix`, то есть вы получаете желаемый результат.

Корректная реализация оператора инкремента для класса `Hour` будет иметь следующий вид:

```
class Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator ++(Hour arg)
    {
        return new Hour(arg.value + 1);
    }
    ...
    private int value;
}
```

Обратите внимание на то, что теперь оператор `++` создает на основе данных оригинала новый объект. Данные в новом объекте инкрементируются, но данные

в оригинале не изменяются. Это вполне работоспособный вариант, но компилятор при трансляции оператора инкремента всякий раз создает новый объект, что может дорого обойтись из-за расходования памяти и издержек на сборку мусора. Поэтому рекомендуется при определении типов не увлекаться перегрузкой операторов. Эта рекомендация касается всех операторов, а не только оператора инкремента.

Определение пар операторов

Особенностью некоторых операций является применение пар операторов. Например, если есть возможность сравнивать два значения типа `Hour` путем использования оператора `!=`, то вполне логично ожидать и сравнения двух `Hour`-значений с помощью оператора `==`. Компилятор C# подкрепляет эти весьма резонные ожидания, настаивая на том, что при определении либо оператора `==`, либо оператора `!=` вы должны определить оба оператора. Правило «все или ничего» применимо и к операторам `<` и `>`, а также `<=` и `>=`. Сам компилятор ни одного из этих партнерских операторов для вас не создает. При всей своей очевидности все партнерские операторы должны быть написаны вами самостоятельно и выражены явным образом. Определения операторов `==` и `!=` для структуры `Hour` имеют следующий вид:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static bool operator ==(Hour lhs, Hour rhs)
    {
        return lhs.value == rhs.value;
    }
    public static bool operator !=(Hour lhs, Hour rhs)
    {
        return lhs.value != rhs.value;
    }
    ...
    private int value;
}
```

Тип возвращаемого этими операторами значения может и не быть булевым. Но у вас должны быть весьма веские причины для использования какого-либо другого типа, в противном случае данные операторы могут стать источником невообразимой путаницы.

ПЕРЕГРУЗКА ОПЕРАТОРОВ РАВЕНСТВА

Если в классе определяются операторы `==` и `!=`, нужно также перегружать методы `Equals` и `GetHashCode`, унаследованные от `System.Object` (или от `System.ValueType`, если создается структура). Метод `Equals` должен демонстрировать точно такое же поведение, что и оператор `==`. (Вы должны определить один из них посредством другого.) Метод `GetHashCode` используется другими классами в среде Microsoft .NET Framework. (Когда, к примеру, объект используется в качестве ключа хэш-таблицы, то, чтобы помочь вычислить хэш-значение, в отношении этого объекта вызывается метод `GetHashCode`. Более подробно этот вопрос изложен в справочной документации по .NET Framework, предоставляемой средой Visual Studio 2015.) Этот метод должен всего лишь возвращать отличающееся целочисленное значение. Не возвращайте из метода `GetHashCode` то же самое целочисленное значение, поскольку это сведет на нет эффективность алгоритмов хэширования.

Реализация операторов

В следующем упражнении вы разработаете класс, имитирующий комплексные числа.

У комплексного числа имеются два элемента: вещественная и мнимая части. Обычно комплексное число представляется в форме $(x + yi)$, где x — это вещественная часть, а yi — мнимая. Значения x и y являются обычными целыми числами, а i представляет собой квадратный корень из -1 (именно поэтому yi называется мнимой частью). Несмотря на то что комплексные числа кажутся слишком сложными для понимания и больше привязанными к теории, чем к практике, они широко применяются в электронике, прикладной математике и физике, а также во многих аспектах проектирования. При желании получить более подробные сведения о пользе комплексных чисел и о том, в чем она выражается, обратитесь к Википедии, где имеется весьма полезная и информативная статья.



ПРИМЕЧАНИЕ В Microsoft .NET Framework версии 4.0 и более поздних имеется тип под названием `Complex`, который находится в пространстве имен `System.Numerics` и реализует комплексные числа, поэтому нет никакого смысла в определении собственной версии этого типа. Тем не менее весьма поучительно посмотреть, как для этого типа реализуются некоторые наиболее широко используемые операторы.

Комплексные числа будут реализованы вами в виде пар целочисленных значений, представляющих коэффициенты x и y для вещественной и мнимой частей. Вы также реализуете операторы, необходимые для выполнения простых арифметических операций с использованием комплексных чисел. В табл. 22.1 дается сводка способов выполнения четырех арифметических операций над парами комплексных чисел, $(a + bi)$ и $(c + di)$.

Таблица 22.1

Операция	Вычисление
$(a + bi) + (c + di)$	$((a + c) + (b + d)i)$
$(a + bi) - (c + di)$	$((a - c) + (b - d)i)$
$(a + bi)(c + di)$	$((ac - bd) + (bc + ad)i)$
$(a + bi)/(c + di)$	$((ac + bd)/(cc + dd)) + ((bc - ad)/(cc + dd))i$

Создание класса Complex и реализация арифметических операторов

Откройте в среде Visual Studio 2015 проект ComplexNumbers, который находится в папке `\Microsoft Press\VCSBS\Chapter 22\ComplexNumbers` вашей папки документов. Для сборки и тестирования кода будет использоваться консольное приложение. В файле `Program.cs` содержится уже знакомый вам метод `doWork`.

В обозревателе решений щелкните на проекте `ComplexNumbers`. В меню **Проект** щелкните на пункте **Добавить класс**. В поле **Имя** диалогового окна **Добавить новый элемент** — `ComplexNumbers` наберите строку `Complex.cs`, а затем щелкните на кнопке **Добавить**. Среда Visual Studio создаст класс `Complex` и откроет файл `Complex.cs` в окне редактора. Добавьте к классу `Complex` автоматически создаваемые целочисленные свойства `Real` и `Imaginary`, показанные в следующем примере кода жирным шрифтом.

```
class Complex
{
    public int Real { get; set; }
    public int Imaginary { get; set; }
}
```

В этих свойствах будут храниться вещественная и мнимая части комплексного числа.

Добавьте к классу `Complex` конструктор, показанный далее жирным шрифтом:

```
class Complex
{
    ...
    public Complex (int real, int imaginary)
    {
        this.Real = real;
        this.Imaginary = imaginary;
    }
}
```

Этот конструктор получает два `int`-параметра и использует их для заполнения свойств `Real` и `Imaginary`. Переопределите метод `ToString`, применив код, выделенный далее жирным шрифтом:

```
class Complex
{
    ...
    public override string ToString()
    {
        return $"{this.Real} + {this.Imaginary}i";
    }
}
```

Этот метод возвращает строку, представляющую комплексное число в формате $(x + yi)$. Добавьте к классу `Complex` перегружаемый оператор `+`, выделенный далее жирным шрифтом:

```
class Complex
{
    ...
    public static Complex operator +(Complex lhs, Complex rhs)
    {
        return new Complex(lhs.Real + rhs.Real, lhs.Imaginary + rhs.Imaginary);
    }
}
```

Это бинарный оператор сложения. Он получает два `Complex`-объекта и складывает их, выполняя вычисление, показанное в табл. 22.1. Оператор возвращает новый `Complex`-объект, содержащий результаты этого вычисления.

Добавьте к классу `Complex` перегружаемый оператор `-`:

```
class Complex
{
    ...
    public static Complex operator -(Complex lhs, Complex rhs)
    {
        return new Complex(lhs.Real - rhs.Real, lhs.Imaginary - rhs.Imaginary);
    }
}
```

Этот оператор придерживается такого же формата, что и перегружаемый оператор `+`.

Реализуйте операторы `*` и `/`:

```
class Complex
{
    ...
    public static Complex operator *(Complex lhs, Complex rhs)
    {
```

```
        return new Complex(lhs.Real * rhs.Real - lhs.Imaginary * rhs.Imaginary,
                           lhs.Imaginary * rhs.Real + lhs.Real * rhs.Imaginary);
    }
    public static Complex operator /(Complex lhs, Complex rhs)
    {
        int realElement = (lhs.Real * rhs.Real + lhs.Imaginary * rhs.Imaginary) /
                           (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);

        int imaginaryElement = (lhs.Imaginary * rhs.Real - lhs.Real *
                               rhs.Imaginary) / (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);

        return new Complex(realElement, imaginaryElement);
    }
}
```

Эти операторы придерживаются того же формата, что и предыдущие два оператора, хотя выполнить вычисление немного сложнее. (Вычисление для оператора `/` было разбито на две части, чтобы строки кода не получались слишком длинными.)

Выведите в окно редактора файл Program.cs. Добавьте к методу `doWork` следующие инструкции, выделенные далее жирным шрифтом, и удалите комментарий `// TODO::`:

```
static void doWork()
{
    Complex first = new Complex(10, 4);
    Complex second = new Complex(5, 2);

    Console.WriteLine($"first is {first}");
    Console.WriteLine($"second is {second}");

    Complex temp = first + second;
    Console.WriteLine($"Add: result is {temp}");

    temp = first - second;
    Console.WriteLine($"Subtract: result is {temp}");

    temp = first * second;
    Console.WriteLine($"Multiply: result is {temp}");

    temp = first / second;
    Console.WriteLine($"Divide: result is {temp}");
}
```

Этот код создает два `Complex`-объекта, которые представляют комплексные значения ($10 + 4i$) и ($5 + 2i$). Код выводит их на экран, а затем тестирует все только что определенные вами операторы, выводя результаты для каждого случая.

В меню Отладка щелкните на пункте Запуск без отладки. Убедитесь в том, что приложение выводит на экран результаты, показанные на рис. 22.1.

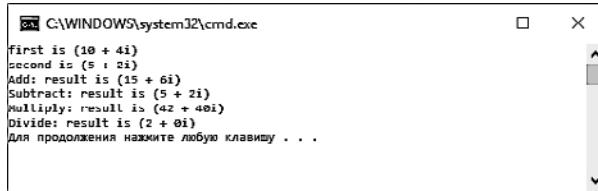


Рис. 22.1

Закройте приложение и вернитесь в среду Visual Studio 2015.

Вы только что создали тип, моделирующий комплексные числа и поддерживающий основные арифметические операции. В следующем упражнении вы расширите класс `Complex` и предоставите для него операторы равенства `==` и `!=`.

Реализация операторов равенства

Выведите в окно редактора среды Visual Studio 2015 файл `Complex.cs`. Добавьте к классу `Complex` операторы `==` и `!=`, показанные далее жирным шрифтом:

```
class Complex
{
    ...
    public static bool operator ==(Complex lhs, Complex rhs)
    {
        return lhs.Equals(rhs);
    }
    public static bool operator !=(Complex lhs, Complex rhs)
    {
        return !(lhs.Equals(rhs));
    }
}
```

Обратите внимание на то, что оба эти оператора используют метод `Equals`, который сравнивает экземпляр класса с другим экземпляром класса, указанным в качестве аргумента. Он возвращает `true`, если их значения равны, и `false`, если они не равны.

Щелкните в меню Сборка на пункте Пересобрать решение.

В окне Список ошибок появятся следующие предупреждающие сообщения:

```
'ComplexNumbers.Complex' defines operator == or operator != but does not override
Object.Equals(object o)
'ComplexNumbers.Complex' defines operator == or operator != but does not override
Object.GetHashCode()
```

Если вы определили операторы `!=` и `==`, то нужно также переопределить методы `Equals` и `GetHashCode`, унаследованные от `System.Object`.



ПРИМЕЧАНИЕ Если список ошибок не показан, щелкните в меню Вид на пункте Список ошибок.

Переопределите метод `Equals` в классе `Complex`, как показано далее жирным шрифтом:

```
class Complex
{
    ...
    public override bool Equals(Object obj)
    {
        if (obj is Complex)
        {
            Complex compare = (Complex)obj;
            return (this.Real == compare.Real) &&
                   (this.Imaginary == compare.Imaginary);
        }
        else
        {
            return false;
        }
    }
}
```

В качестве параметра метод `Equals` получает значение типа `Object`. Этот код проверяет, что тип параметра фактически является `Complex`-объектом. Если так оно и есть, код сравнивает значения в свойствах `Real` и `Imaginary` в текущем экземпляре и в переданном ему параметре. Если они одинаковы, метод возвращает `true`, а если нет — `false`. Если переданный параметр не является `Complex`-объектом, метод возвращает `false`.



ВНИМАНИЕ У вас может возникнуть соблазн написать метод `Equals` следующим образом:

```
public override bool Equals(Object obj)
{
    Complex compare = obj as Complex;
    if (compare != null)
    {
        return (this.Real == compare.Real) &&
               (this.Imaginary == compare.Imaginary);
    }
    else
    {
        return false;
    }
}
```

Но выражение `compare != null` вызывает оператор `!=` класса `Complex`, который опять вызывает метод `Equals`, что приводит к рекурсивному циклу.

Переопределите метод `GetHashCode`. Эта реализация просто вызывает метод, унаследованный от класса `Object`, но если хотите, то для создания хэш-кода для объекта можете предоставить собственный механизм:

```
Class Complex
{
    ...
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}
```

Щелкните в меню Сборка на пункте Пересобрать решение. Убедитесь в том, что теперь решение проходит сборку без каких-либо предупреждений.

Вызовите в окно редактора файл `Program.cs` и добавьте к концу метода `doWork` следующий код, выделенный жирным шрифтом:

```
static void doWork()
{
    ...
    if (temp == first)
    {
        Console.WriteLine("Comparison: temp == first");
    }
    else
    {
        Console.WriteLine("Comparison: temp != first");
    }
    if (temp == temp)
    {
        Console.WriteLine("Comparison: temp == temp");
    }
    else
    {
        Console.WriteLine("Comparison: temp != temp");
    }
}
```



ПРИМЕЧАНИЕ Выражение `temp == temp` вызывает вывод предупреждающего сообщения «`Comparison made to same variable; did you mean to compare to something else?`» («Сравнение проводится с той же самой переменной; возможно, вы предполагали провести сравнение с чем-нибудь другим?») В данном случае можете проигнорировать предупреждение, поскольку такое сравнение выполнено намеренно, чтобы проверить, что оператор `==` работает так, как ожидалось.

В меню Отладка щелкните на пункте Запуск без отладки. Убедитесь в том, что последние два сообщения, выведенные на экран, выглядят следующим образом:

```
Comparison: temp != first
Comparison: temp == temp
```

Закройте приложение и вернитесь в среду Visual Studio 2015.

Основные сведения об операторах преобразования

Иногда нужно преобразовать выражение одного типа в выражение другого типа. Например, следующий метод объявлен с единственным параметром типа `double`:

```
class Example
{
    public static void MyDoubleMethod(double parameter)
    {
        ...
    }
}
```

Вполне логично ожидать, что когда ваш код вызывает метод `MyDoubleMethod`, в качестве аргумента могут использоваться только значения типа `double`, но это не так. Компилятор C# также позволяет методу `MyDoubleMethod` быть вызванным с аргументом какого-либо другого типа, но только если значение аргумента может быть преобразовано в `double`. Например, если предоставить аргумент типа `int`, компилятор при вызове метода создаст код, преобразующий значение аргумента в `double`.

Предоставление встроенных преобразований

Во встроенных типах имеются некоторые встроенные преобразования. Например, как уже упоминалось, `int`-значение может быть неявным образом преобразовано в значение типа `double`. Неявное преобразование не требует специального синтаксиса и никогда не выдает исключений:

```
Example.MyDoubleMethod(42); // Неявное преобразование int в double
```

Иногда неявное преобразование называют расширяющим преобразованием, поскольку результат оказывается шире исходного значения — в нем содержится больше информации, чем в исходном значении, и при этом ничего не теряется. В случае с типами `int` и `double` диапазон `double` шире диапазона `int` и все `int`-значения имеют эквивалентные им `double`-значения. Но обратное утверждение будет неверным, и `double`-значение не может быть неявным образом преобразовано в `int`-значение:

```
class Example
{
    public static void MyIntMethod(int parameter)
    {
        ...
    }
}
...
Example.MyIntMethod(42.0); // Ошибка в ходе компиляции
```

При преобразовании `double` в `int` есть риск потери информации, поэтому преобразование не выполняется в автоматическом режиме. (Подумайте, что получилось бы, если бы аргументом `MyIntMethod` было число `42.5`. Как бы оно могло быть преобразовано?) Значение типа `double` может быть преобразовано в значение типа `int`, но преобразование требует явной записи (приведения типа):

```
Example.MyIntMethod((int)42.0);
```

Явное преобразование иногда называют сужающим преобразованием, поскольку результат получается менее широким, чем исходное значение (то есть в нем может содержаться меньше информации), и может выдать исключение `OverflowException`, если результат значения выходит за допустимый диапазон целевого типа. В C# можно создать операторы преобразования для самостоятельно определяемых вами типов, чтобы контролировать целесообразность преобразования значений в другие типы. Также можно указать, будет ли это преобразование неявным или явным.

Реализация операторов преобразований, определяемых пользователем

Синтаксис для объявления операторов преобразования, определяемых пользователем, отчасти похож на объявление перегружаемого оператора, но у него также имеется несколько важных отличий. Вот как выглядит оператор преобразования, позволяющий `Hour`-объекту быть неявным образом преобразованным в `int`-значение:

```
struct Hour
{
    ...
    public static implicit operator int (Hour from)
    {
        return from.value;
    }

    private int value;
}
```

Оператор преобразования должен быть открытым (`public`), а также статическим (`static`). Тип, из которого осуществляется преобразование, объявляется в качестве параметра (в данном случае это `Hour`), а тип, в который осуществляется преобразование, объявляется в качестве названия типа после ключевого слова `operator` (в данном случае это `int`). Возвращаемого значения, указываемого перед ключевым словом `operator`, здесь нет.

При объявлении собственных операторов преобразования нужно указать, какими именно они являются — неявными или явными. Это делается путем

использования ключевых слов `implicit` и `explicit`. Оператор преобразования `Hour` в `int`, показанный в предыдущем примере, является неявным, а это означает, что компилятор C# может использовать его, не требуя при этом приведения типа:

```
class Example
{
    public static void MyOtherMethod(int parameter) { ... }
    public static void Main()
    {
        Hour lunch = new Hour(12);
        Example.MyOtherMethod(lunch); // Неявное преобразование Hour в int
    }
}
```

Если оператор преобразования был объявлен явным, то есть с ключевым словом `explicit`, предыдущий пример не будет скомпилирован, поскольку оператор явного преобразования требует приведения типа.

```
Example.MyOtherMethod((int)lunch); // Явное преобразование Hour в int
```

Когда нужно объявлять оператор преобразования явным, а когда неявным? Если преобразование всегда проводится безопасно, без риска потери информации, и не может выдать исключение, оно может быть определено как неявное. В противном случае оно должно быть объявлено как явное. Преобразование из `Hour` в `int` всегда безопасно — для каждого значения `Hour` имеется соответствующее значение `int`, следовательно, есть смысл сделать его неявным. Оператор, преобразующий строковое значение `string` в значение `Hour`, должен осуществлять явное преобразование, поскольку не все строки являются допустимым представлением значений для часа — `Hours`. (Строка «7» подходит, а как можно преобразовать в `Hour` строку «Hello, World»?)

Повторное обращение к теме симметричных операторов

Операторы преобразования предоставляют вам альтернативный способ решения проблемы симметричных операторов. Например, вместо предоставления показанных ранее трех версий оператора `+` (`Hour + Hour`, `Hour + int` и `int + Hour`) для структуры `Hour` вы можете предоставить единственную версию оператора `+` (которая принимает два `Hour`-параметра) и неявное преобразование `int` в `Hour`:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
```

```

public static Hour operator +(Hour lhs, Hour rhs)
{
    return new Hour(lhs.value + rhs.value);
}

public static implicit operator Hour (int from)
{
    return new Hour (from);
}
...
private int value;
}

```

Если сложить `Hour` и `int` (в любом порядке), компилятор C# автоматически преобразует `int` в `Hour`, а затем вызовет оператор `+` с двумя аргументами типа `Hour`:

```

void Example(Hour a, int b)
{
    Hour eg1 = a + b; // b преобразуется в Hour
    Hour eg2 = b + a; // b преобразуется в Hour
}

```

Написание операторов преобразований

В заключительном упражнении этой главы вы добавите операторы преобразования к классу `Complex`. Сначала вы напишете пару операторов преобразования, позволяющих осуществлять преобразование между типами `int` и `Complex`. Преобразование `int` в объект типа `Complex` всегда выполняется безопасно и никогда не приводит к потере информации, поскольку `int` фактически является простым комплексным числом без мнимой части. Оно будет реализовано в виде оператора неявного преобразования. Но обратное утверждение будет неверным — чтобы преобразовать `Complex`-объект в `int`, нужно отбросить мнимую часть. Следовательно, реализация будет выполнена в виде создания оператора явного преобразования.

Реализация операторов преобразований

Вернитесь в среду Visual Studio 2015 и выведите в окно редактора файл `Complex.cs`. Добавьте к коду, который следует за классом `Complex`, сразу после существующего конструктора и перед методом `ToString` еще один конструктор, показанный далее жирным шрифтом. Этот новый конструктор получит один `int`-параметр, который используется им для инициализации свойства `Real`. Для свойства `Imaginary` устанавливается значение 0:

```

class Complex
{
    ...
    public Complex(int real)

```

```
{  
    this.Real = real;  
    this.Imaginary = 0;  
}  
...  
}
```

Добавьте к классу `Complex` следующий оператор неявного преобразования:

```
class Complex  
{  
    ...  
    public static implicit operator Complex(int from)  
    {  
        return new Complex(from);  
    }  
    ...  
}
```

Этот оператор осуществляет преобразование из `int` в `Complex`-объект, возвращая новый экземпляр класса `Complex` с помощью только что созданного вами конструктора.

Добавьте к классу `Complex` следующий оператор явного преобразования, показанный далее жирным шрифтом:

```
class Complex  
{  
    ...  
    public static explicit operator int(Complex from)  
    {  
        return from.Real;  
    }  
    ...  
}
```

Этот оператор получает `Complex`-объект и возвращает значение свойства `Real`. При этом преобразовании отбрасывается мнимая часть комплексного числа.

Выведите в окно редактора файл `Program.cs` и добавьте к концу метода `doWork` следующий код, показанный далее жирным шрифтом:

```
static void doWork()  
{  
    ...  
    Console.WriteLine($"Current value of temp is {temp}");  
  
    if (temp == 2)  
    {  
        Console.WriteLine("Comparison after conversion: temp == 2");  
    }  
    else  
    {  
        Console.WriteLine("Comparison after conversion: temp != 2");  
    }  
}
```

```

    }

    temp += 2;
    Console.WriteLine($"Value after adding 2: temp = {temp}");
}

```

Эти инструкции выполняют тестирование оператора неявного преобразования `int` в `Complex`-объект. Инструкция `if` сравнивает `Complex`-объект с `int`. Компилятор создает код, который сначала преобразует `int` в `Complex`-объект, а затем вызывает оператор `==` класса `Complex`. Инструкция, которая прибавляет 2 к значению переменной `temp`, преобразует `int`-значение 2 в `Complex`-объект, а затем использует оператор `+` класса `Complex`.

Добавьте к концу метода `doWork` следующие инструкции:

```

static void doWork()
{
    ...
    int tempInt = temp;
    Console.WriteLine($"Int value after conversion: tempInt == {tempInt}");
}

```

Первая инструкция пытается присвоить `Complex`-объект `int`-переменной.

Щелкните в меню Сборка на пункте Пересобрать решение. Сборка пройдет неудачно, и компилятор сообщит в окне Список ошибок о следующей ошибке:

```
Cannot implicitly convert type 'ComplexNumbers.Complex' to 'int'. An explicit conversion exists (are you missing a cast?)
```

Оператор, осуществляющий преобразование из `Complex`-объекта в `int`-значение, является оператором явного преобразования, поэтому нужно указать приведение типа.

Измените инструкцию, которая пытается сохранить `Complex`-значение в `int`-переменной, так, чтобы в ней использовалось приведение типа:

```
int tempInt = (int)temp;
```

В меню Отладка щелкните на пункте Запуск без отладки. Убедитесь в том, что теперь решение проходит сборку и выведенные на экран последние четыре сообщения имеют следующий вид:

```
Current value of temp is (2 + 0i)
Comparison after conversion: temp == 2
Value after adding 2: temp = (4 + 0i)
Int value after conversion: tempInt == 4
```

Закройте приложение и вернитесь в среду Visual Studio 2015.

Выводы

В этой главе вы научились перегружать операторы и предоставлять соответствующие функциональные средства применительно к классу или структуре. Вами были реализованы несколько арифметических операторов, а также созданы операторы, с помощью которых можно сравнивать экземпляры класса. И наконец, вы научились создавать операторы неявного и явного преобразования.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 23 «Повышение производительности путем использования задач».

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Реализовать оператор	<p>Напишите ключевые слова <code>public</code> и <code>static</code>, укажите после них возвращаемое значение, затем ключевое слово <code>operator</code>, а за ним — символ объявляемого оператора, после которого укажите в круглых скобках соответствующие параметры. Реализуйте в теле метода логику для оператора, например:</p> <pre>class Complex { ... public static bool operator ==(Complex lhs, Complex rhs) { ... // Реализация логики для оператора == } ... }</pre>
Определить оператор преобразования	<p>Напишите ключевые слова <code>public</code> и <code>static</code>, укажите после них ключевое слово <code>implicit</code> или <code>explicit</code>, а за ним — ключевое слово <code>operator</code> и тип, в который осуществляется преобразование, после чего укажите преобразуемый тип в виде единственного параметра, заключенного в круглые скобки, например:</p> <pre>class Complex { ... public static implicit operator Complex(int from) { ... // Код для преобразования из int } ... }</pre>

Часть IV

Создание приложений универсальной платформы Windows с использованием C#

Итак, вы уже изучили основы синтаксиса и семантики языка C#. Настало время проверить, как можно воспользоваться этими знаниями применительно к функциональным средствам, предоставляемым Windows для создания приложений, работающих в неизменном виде на широком диапазоне устройств, от настольных компьютеров до смартфонов. Создавать приложения, работающие в различных средах, позволяет универсальная платформа Windows — UWP. UWP-приложения могут определять характеристики оборудования, на котором запускаются, и подстраиваться под него. Они в состоянии получать информацию с сенсорного экрана и могут быть разработаны с учетом осведомленности о местоположении и ориентации устройства, на котором запускаются. Возможности Windows 10 позволяют также создавать приложения, ориентированные на подключение к средам облачных вычислений, не привязанным к конкретному компьютеру, и способные сопровождать пользователей, регистрирующихся с другого устройства. Короче говоря, Windows 10 предоставляет платформу для разработки высокомобильных универсальных приложений, использующих графику высокого разрешения и скоростное интернет-подключение.

В части IV предлагается знакомство с требованиями, предъявляемыми к созданию UWP-приложений. Вам будут показаны примеры асинхронной модели программирования, разработанной в качестве части среды .NET Framework. Также будет рассмотрено создание UWP-приложения, подключающегося к облаку для извлечения и представления сложной информации, во вполне естественном и легко управляемом стиле.

23

Повышение производительности путем использования задач

Прочитав эту главу, вы научитесь:

- объяснять преимущества реализации параллельных операций в приложении;
- использовать класс `Task` для создания и запуска параллельных операций в приложении;
- использовать класс `Parallel` для распараллеливания некоторых наиболее часто встречающихся в программировании конструкций;
- отменять долго выполняющиеся задачи и обрабатывать исключения, выдаваемые при выполнении параллельных операций.

В основной части предыдущих глав книги вы изучали использование C# для написания программ, выполняемых в однопоточном режиме. Под однопоточностью я подразумеваю то, что в любой отдельно взятый момент времени программа выполняет одну инструкцию. Но порой такой подход может быть не самым эффективным для приложения. При наличии соответствующих вычислительных ресурсов некоторые приложения могут работать значительно быстрее, если разбить их на части, которые могут выполняться параллельно в одно и то же время. Эта глава посвящена вопросам повышения производительности приложений путем интенсификации использования доступной вычислительной мощности. В частности, в этой главе вы узнаете о порядке использования `Task`-объектов с целью применения эффективной многозадачности для приложений, занимающихся интенсивными вычислениями.

Зачем нужна многозадачность, достигаемая параллельной обработкой данных?

Применение многозадачности в приложении может понадобиться по двум основным причинам.

- **Для повышения оперативности реагирования.** Продолжительные операции могут включать задачи, не требующие процессорного времени. К числу наиболее распространенных примеров относятся операции, связанные с вводом-выводом, например чтение с локального диска или запись на него, а также отправка и получение данных по сети. В обоих этих случаях не имеет смысла заставлять программу впустую тратить циклы работы центрального процессора, ожидая завершения операций, когда программа может делать вместо этого что-либо полезное, например реагировать на пользовательский ввод. Большинство пользователей мобильных устройств принимают эту форму реагирования как нечто само собой разумеющееся и не предполагают, что их планшетный компьютер просто замрет, отправляя или получая сообщение электронной почты. Более подробно эти особенности раскрыты в главе 24 «Сокращение времени отклика путем выполнения асинхронных операций».
- **Для повышения масштабируемости.** Если операция связана с использованием ресурсов центрального процессора, то масштабируемость можно повысить путем более эффективного использования доступных вычислительных ресурсов, задействуя их для сокращения времени, требующегося на выполнение операции. Разработчик может определить, какие операции включают задачи, которые могут быть выполнены в параллельном режиме, и принять меры к тому, чтобы они выполнялись одновременно. Чем больше будет добавляться вычислительных ресурсов, тем больше таких задач сможет выполняться в параллельном режиме. До относительно недавнего времени эта модель подходила только для научных и инженерных систем, у которых либо имелось несколько центральных процессоров, либо была возможность распределить обработку между различными компьютерами, связанными сетью. Но теперь большинство компьютерных устройств содержат мощные центральные процессоры, способные поддерживать реальную многозадачность, и многие операционные системы предоставляют элементарные процедуры, позволяющие относительно легко распараллеливать задачи.

Взлет многоядерных процессоров

На рубеже веков стоимость приличного персонального компьютера колебалась от 800 до 1500 долларов. Сегодня достойный внимания персональный компьютер, даже после 15 лет ценовой инфляции, стоит примерно столько же.

Характеристика типового компьютера в наши дни включает процессор с тактовой частотой 2–3 ГГц, накопитель на жестком диске емкостью свыше 1000 Гбайт, оперативную память объемом 4–8 Гбайт, высокоскоростное графическое устройство с высоким разрешением, быстрые сетевые интерфейсы и DVD-привод с возможностью перезаписи дисков. Пятнадцать лет назад тактовая частота процессора типовой машины была между 500 МГц и 1 ГГц, емкость большого жесткого диска — 80 Гбайт, Windows вполне удовлетворялась оперативной памятью 256 Мбайт и даже меньше, а приводы перезаписываемых компакт-дисков стоили более 100 долларов. (Перезаписывающие DVD-приводы встречались крайне редко и стоили очень дорого.) Тем и вызывает восхищение технологический прогресс: все более быстрое и мощное оборудование продаётся по все более низким ценам.

Эта тенденция не нова. В 1965 году Гордон Мур, сооснователь компании Intel, опубликовал статью под названием «Cramming More Components onto Integrated Circuits» («Заполнение интегральных микросхем все большим количеством компонентов»), речь в которой шла о том, что рост миниатюризации компонентов, позволяющий размещать на кремниевой микросхеме больше транзисторов, и снижение стоимости производства с ростом доступности технологий приведут к тому, что к 1975 году станет вполне рентабельным уместить на одной микросхеме примерно 65 000 компонентов. Эти наблюдения позволили ему вывести часто упоминаемый закон Мура, основное утверждение которого гласит, что количество транзисторов, размещаемых на недорогой интегральной микросхеме, будет увеличиваться экспоненциально, удваиваясь примерно каждые два года. (Вообще-то Гордон Мур сначала давал более оптимистичный прогноз, утверждая, что объем транзисторов будет, скорее всего, увеличиваться вдвое каждый год, но позже скорректировал свои подсчеты.) Возможность совместного размещения транзисторов привела к возможности более быстрой передачи данных между ними. Это означает, что мы вправе ожидать, что производители микросхем станут выпускать более быстрые и мощные микропроцессоры практически неизменными темпами, позволяя разработчикам программных средств создавать все более сложные программы, способные выполняться еще быстрее.

Закон Мура, касающийся миниатюризации электронных компонентов, все еще в силе, даже полстолетия спустя. Но в дело стала вмешиваться физика. Возникли ограничения, связанные с невозможностью передавать сигналы между транзисторами на отдельно взятой микросхеме еще быстрее, независимо от того, насколько они малы или плотно упакованы. Наиболее заметным результатом этого ограничения для разработчиков программных средств стало прекращение роста скорости работы процессоров. Десять лет назад быстрый процессор работал на частоте 3 ГГц. И сегодня быстрый процессор по-прежнему работает на частоте 3 ГГц.

Ограничение по скорости передачи процессорами данных между компонентами заставили компании по производству микросхем искать альтернативные

механизмы повышения объемов работ, с которыми мог бы справиться процессор. В результате большинство современных процессоров имеют два и более процессорных ядер. В действительности производители микросхем поместили несколько процессоров на один и тот же кристалл и добавили логику, необходимую для их взаимосвязи и координации. Сейчас нередко можно встретить четырех- и восьмиядерные процессоры. Доступны и кристаллы с 16, 32 и 64 ядрами, а цена двух- и четырехъядерных процессоров существенно снизилась, и их присутствие вполне ожидаемо в ноутбуках, планшетных компьютерах и смартфонах. Поэтому, несмотря на прекращение роста тактовой частоты процессоров, сегодня можно рассчитывать на получение их большего количества на одном кристалле.

А что это означает для разработчиков приложений на C#?

До появления многоядерных процессоров ускорить работу приложения, выполняемого в одном потоке, можно было просто за счет его запуска на более быстром процессоре. С появлением многоядерных процессоров ситуация изменилась. Однопоточные приложения будут работать с одинаковой скоростью на одно-, двух- или четырехъядерных процессорах с одной и той же тактовой частотой. Разница лишь в том, что при условии выполнения приложения на двухъядерном процессоре одно из ядер будет находиться в простое, а на четырехъядерном процессоре в ожидании работы будут простоявать три ядра. Для рационального использования многоядерных процессоров нужно создавать такие приложения, которые получали бы преимущества от применения многозадачности.

Реализация многозадачности с помощью Microsoft .NET Framework

Многозадачность представляет собой возможность одновременного совершения сразу нескольких действий. Это одно из тех понятий, которые легко поддавались описанию, но реализация которых до последнего времени вызывала существенные затруднения.

Согласно оптимальному сценарию приложение, запущенное на многоядерном процессоре, одновременно выполняет столько задач, сколько для него имеется доступных процессорных ядер, загружая задачами каждое из них. Но для реализации одновременного выполнения нужно решить множество вопросов, включая следующие.

- ❑ Как разбить приложение на набор одновременно выполняемых операций?
- ❑ Как подстроиться под набор операций, одновременно выполняемых на нескольких процессорах?

- ❑ Как обеспечить попытку одновременного выполнения такого количества операций, которое в точности соответствует количеству доступных процессоров?
- ❑ Как обнаружить заблокированные операции (например, ожидающие завершения ввода-вывода) и настроить процессор для запуска другой операции вместо того, чтобы находиться в простое?
- ❑ Как определить, когда именно завершится одна или несколько одновременно выполняемых операций?

Для разработчика приложений первый из этих вопросов относится к проектированию приложения. Все остальные зависят от программной инфраструктуры. Чтобы помочь в решении этих вопросов, компания Microsoft предоставляет класс `Task` и коллекцию связанных с ним типов, находящуюся в пространстве имен `System.Threading.Tasks`.



ВНИМАНИЕ Основным здесь является вопрос, касающийся проектирования приложения. Если приложение реализуется без прицела на многозадачность, то неважно, сколько процессорных ядер будет на него задействовано, оно будет работать ничуть не быстрее, чем на одноядерной машине.

Задачи, потоки и ThreadPool

Класс `Task` является абстракцией операции, выполняемой одновременно с другими операциями. `Task`-объект создается для запуска блока кода. Можно создать несколько экземпляров `Task`-объектов и, при условии достаточного количества доступных процессоров или процессорных ядер, запустить их в параллельном режиме выполнения.



ПРИМЕЧАНИЕ Впредь я буду использовать понятие «процессор» для обозначения либо одноядерного процессора, либо отдельно взятого ядра многоядерного процессора.

Внутри среды Windows Runtime (WinRT) реализация задач и их диспетчеризация для выполнения осуществляются с помощью объектов `Thread` и класса `ThreadPool`. Многопоточность и пулы потоков были доступны в .NET Framework, начиная с версии 1.0, и при создании обычных приложений для настольного компьютера класс `Thread`, находящийся в пространстве имен `System.Threading`, можно использовать в коде напрямую. Но для приложений универсальной платформы Windows класс `Thread` недоступен, а вместо него используется класс `Task`.

Класс `Task` предоставляет весьма эффективную абстракцию, позволяющую довольно легко разобраться в разнице между степенью распараллеливания в приложении (задачами) и блоками для параллельного выполнения (потоками). На однопроцессорном компьютере эти понятия обычно не различаются.

Но на компьютере с несколькими процессорами или с многоядерным процессором они отличаются друг от друга. При проектировании программы, основанной непосредственно на потоках, может оказаться, что приложение не слишком хорошо поддается масштабированию: программа будет использовать то количество потоков, которое было создано явным образом, а операционная система станет заниматься диспетчеризацией только этого количества потоков. Если количество потоков существенно превышает количество доступных процессоров, это может привести к превышению уровня допустимой нагрузки и к плохому показателю времени отклика, а если количество потоков меньше количества процессоров, это может обусловить неэффективность использования оборудования и низкую производительность.

WinRT оптимизирует количество потоков, требующееся для реализации конкретных задач, и осуществляет их эффективную диспетчеризацию в соответствии с количеством доступных процессоров. Для распределения рабочей нагрузки между набором потоков, реализуемых за счет использования объекта типа `ThreadPool`, эта среда организует механизм постановки в очередь. Когда программа создает `Task`-объект, задача добавляется к глобальной очереди. Когда поток становится доступен, задача удаляется из глобальной очереди и выполняется этим потоком. Класс `ThreadPool` выполняет ряд оптимизаций, а для того чтобы обеспечить эффективность диспетчеризации потоков, использует алгоритм на основе перехвата работы (*work-stealing algorithm*).



ПРИМЕЧАНИЕ Класс `ThreadPool` был доступен и в предыдущих редакциях .NET Framework, но в .NET Framework 4.0 он был существенно усовершенствован для поддержки экземпляров класса `Task`.

Следует заметить, что количество потоков, созданных для обработки ваших задач, не равно количеству процессоров. В зависимости от характеристики рабочей нагрузки один или несколько процессоров должны быть заняты выполнением высокоприоритетной работы для других приложений и служб. Следовательно, оптимальное количество потоков для вашего приложения может быть меньше, чем количество процессоров в машине. Кроме того, один или несколько потоков в приложении могут находиться в состоянии ожидания завершения продолжительных операций обращения к памяти, ввода-вывода или обмена данными по сети, освобождая тем самым соответствующие процессоры от нагрузки. В таком случае оптимальное количество потоков может быть больше количества доступных процессоров. Для динамического определения идеального количества потоков для текущей рабочей нагрузки среда WinRT следует итеративной стратегии, известной как алгоритм восхождения на гору (*hill-climbing algorithm*).

Важно усвоить, что единственной вашей обязанностью при работе с кодом является деление приложения, или его разбиение на задачи, которые могут выполняться в параллельном режиме. Среда WinRT берет на себя обязанности по созданию

соответствующего количества потоков на основе архитектуры процессора и рабочей нагрузки вашего компьютера, связывая задачи с этими потоками и подстраивая одно под другое с целью эффективного выполнения кода. Не страшно, если вы разделите свою работу на слишком много задач — WinRT попытается запустить ровно столько параллельных потоков, сколько имеет смысл выполнять. Фактически вам рекомендуется разбивать свою работу на большее количество частей, поскольку это поможет гарантировать масштабирование приложения при его перемещении на компьютер, имеющий большее количество доступных процессоров.

Создание задач, их выполнение и управление ими

Создать Task-объекты можно с помощью Task-конструктора. Это перегружаемый конструктор, но все его версии ждут от вас предоставления в качестве параметра Action-делегата. В главе 20 «Отделение логики приложения и обработка событий» было показано, что Action-делегат ссылается на метод, не возвращающий значение. Task-объект задействует этого делегата при диспетчеризации выполнения его кода. В следующем примере создается Task-объект, использующий делегата для запуска метода по имени doWork:

```
Task task = new Task(doWork);
...
private void doWork()
{
    // Этот код выполняется задачей при запуске
    ...
}
```



COBET Изначально тип Action ссылается на метод, не получающий параметры. Другие переопределения конструктора Task получают параметр Action<object>, представляющий собой делегата, который ссылается на метод, получающий один параметр типа object. Используя эти переопределения, можно передавать данные методу, запускаемому задачей. Примером может послужить следующий код:

```
Action<object> action;
action = doWorkWithObject;
object parameterData = ...;
Task task = new Task(action, parameterData);
...
private void doWorkWithObject(object o)
{
    ...
}
```

После создания объекта типа Task его можно запустить на выполнение, используя метод Start:

```
Task task = new Task(...);
task.Start();
```

Метод `Start` перегружается, позволяя дополнительно указать объект типа `TaskCreationOptions` для предоставления сведений о том, как планировать и запускать задачу.



ПРИМЕЧАНИЕ За дополнительными сведениями о перечислении `TaskCreationOptions` обращайтесь к поставляемой с Visual Studio документации с описанием библиотеки классов .NET Framework.

Создание и запуск задачи — весьма распространенный процесс, и класс `Task` предоставляет статический метод `Run`, с помощью которого можно эти операции объединить. Метод `Run` получает `Action`-делегата, указывающего на выполняемую операцию (подобно `Task`-конструктору), но тут же запускает выполнение задачи. Он возвращает ссылку на `Task`-объект. Этим можно воспользоваться следующим образом:

```
Task task = Task.Run(() => doWork());
```

Когда запущенный задачей метод завершит свою работу, задача завершится и поток, использовавшийся для запуска задачи, может быть использован повторно для выполнения другой задачи.

Когда задача завершается, можно подготовиться к другой задаче, запланированной на немедленное выполнение, создав *продолжение*. Для этого нужно вызвать принадлежащий `Task`-объекту метод `ContinueWith`. Когда действие, выполняемое `Task`-объектом, завершается, диспетчер автоматически создает новый `Task`-объект для запуска действия, указанного методом `ContinueWith`. Метод, указанный продолжением, ожидает `Task`-параметр, и диспетчер передает в метод ссылку на завершенную задачу. Значение, возвращенное методом `ContinueWith`, является ссылкой на новый `Task`-объект. В следующем примере кода создается `Task`-объект, запускающий метод `doWork` и указывающий продолжение, запускающее метод `doMoreWork` в новой задаче, когда будет завершена первая задача:

```
Task task = new Task(doWork);
task.Start();
Task newTask = task.ContinueWith(doMoreWork);
...
private void doWork()
{
    // Этот код задача выполняет при запуске
    ...
}
...
private void doMoreWork(Task task)
{
    // Этот код продолжение выполняет по завершении работы метода doWork
    ...
}
```

Метод `ContinueWith` активно перегружается, и вы, включая значение `TaskContinuationOptions`, можете предоставить ряд параметров, указывающих дополнительные элементы. Тип `TaskContinuationOptions` является перечислением, содержащим расширенный набор значений, содержащихся в перечислении типа `TaskCreationOptions`. К числу дополнительных доступных значений относятся:

- ❑ `NotOnCanceled` и `OnlyOnCanceled`. Необязательный параметр `NotOnCanceled` указывает, что продолжение должно запускаться, только если предыдущее действие завершилось и не было отменено, а необязательный параметр `OnlyOnCanceled` указывает, что продолжение должно запускаться, только если предыдущее действие было отменено. Отмена задачи рассматривается далее в разделе «Отмена задач и обработка исключений»;
- ❑ `NotOnFaulted` и `OnlyOnFaulted`. Необязательный параметр `NotOnFaulted` показывает, что продолжение должно быть запущено, только если предыдущее действие завершилось и не выдало необрабатываемое исключение. Необязательный параметр `OnlyOnFaulted` заставляет продолжение запускаться, только если предыдущее действие выдало необрабатываемое исключение. Дополнительные сведения по управлению исключениями в задаче даны в разделе «Отмена задач и обработка исключений»;
- ❑ `NotOnRanToCompletion` и `OnlyOnRanToCompletion`. Необязательный параметр `NotOnRanToCompletion` указывает, что продолжение должно запускаться, только если предыдущее действие не достигло успешного завершения (либо оно может быть отменено), или выдать исключение. Необязательный параметр `OnlyOnRanToCompletion` заставляет продолжение запуститься, только если предыдущее действие завершилось успешно.

В следующих примерах кода показано, как к задаче добавляется продолжение, запускаемое, только если исходное действие не выдало необрабатываемое исключение:

```
Task task = new Task(doWork);
task.ContinueWith(doMoreWork, TaskContinuationOptions.NotOnFaulted);
task.Start();
```

Общим требованием к приложениям, активизирующими операции в параллельном режиме, является синхронизация задач. Класс `Task` предоставляет метод `Wait`, реализующий простой механизм координации задач. Используя этот метод, можно приостановить выполнение текущего потока до завершения указанной задачи:

```
Task task2 = ...
task2.Start();
...
task2.Wait(); // Ожидание, пока не завершится task2
```

Можно ожидать завершения целого набора задач, воспользовавшись для этого статическими методами `WaitAll` и `WaitAny` класса `Task`. Оба метода получают массив `params`, содержащий набор `Task`-объектов. Метод `WaitAll` ждет завершения всех указанных задач, а метод `WaitAny` ждет завершения хотя бы одной из указанных задач. Воспользоваться ими можно следующим образом:

```
Task.WaitAll(task, task2); // Ожидание завершения обеих задач, task и task2  
Task.WaitAny(task, task2); // Ожидание завершения любой из задач, либо task,  
// либо task2
```

Использование класса Task для реализации выполнения программы в параллельном режиме

В следующем упражнении класс `Task` будет использоваться для распараллеливания кода приложения, интенсивно загружающего процессор, и для показа того, как распараллеливание сокращает время, затрачиваемое приложением на выполнение вычислений, путем их распределения среди нескольких ядер процессора.

Приложение с названием `GraphDemo` состоит из страницы, использующей элемент управления `Image` для вывода графики. Приложение выстраивает точки для графики, выполняя сложное вычисление.



ПРИМЕЧАНИЕ Упражнения в этой главе предназначены для запуска на компьютере с многоядерным процессором. Если у вас одноядерный процессор, вы не сможете наблюдать те же эффекты. Кроме того, не сможете запускать в ходе выполнения упражнений дополнительные программы или службы, потому что это может отразиться на наблюдаемых результатах.

Изучение и запуск однопоточного приложения `GraphDemo`

Откройте в среде Microsoft Visual Studio 2015 решение `GraphDemo`, которое находится в папке `\Microsoft Press\VCBS\Chapter 23\GraphDemo` вашей папки документов. Это приложение предназначено для работы на платформе UWP.

В обозревателе решений дважды щелкните на файле `MainPage.xaml` проекта `GraphDemo` для отображения формы в окне конструктора. Кроме элемента управления `Grid`, определяющего разметку, в форме имеются следующие важные элементы управления:

- ❑ элемент управления изображением `Image` с названием `graphImage`. Он выводит графику, воспроизводимую приложением;
- ❑ элемент управления `Button` (кнопка) с названием `plotButton`. Пользователи щелкают на этой кнопке для генерации данных для графики и ее вывода в элемент управления `graphImage`;



ПРИМЕЧАНИЕ Чтобы не усложнять операции этого приложения, оно выводит кнопку на страницу. В коммерческих UWP-приложениях кнопки такого рода должны располагаться на панели управления.

- элемент управления `TextBlock` с названием `duration`. В этой надписи приложение показывает время, затраченное на генерацию данных и вывод на экран графики, построенной по этим данным.

В обозревателе решений раскройте файл `MainPage.xaml`, а затем дважды щелкните на файле `MainPage.xaml.cs`, чтобы код формы был выведен в окно редактора.

Для вывода графики в форме используется объект типа `WriteableBitmap` (он определен в пространстве имен `Windows.UI.Xaml.Media.Imaging`) по имени `graphBitmap`. Горизонтальное и вертикальное разрешение для `WriteableBitmap`-объекта указывается в переменных `pixelWidth` и `pixelHeight` соответственно:

```
public partial class MainPage : Window
{
    // Если места недостаточно, уменьшите значения pixelWidth и pixelHeight
    private int pixelWidth = 15000;
    private int pixelHeight = 10000;

    private WriteableBitmap graphBitmap = null;
    ...
}
```



ПРИМЕЧАНИЕ Это приложение было разработано и протестировано на настольном компьютере с 8 Гбайт памяти. Если на вашем компьютере доступен меньший объем памяти, может понадобиться уменьшить значение переменных `pixelWidth` и `pixelHeight`, в противном случае приложение может выдать исключение `OutOfMemoryException`. Если же у вас имеется больший объем доступной памяти, то для полноценного наблюдения за эффектом, получаемым в этом упражнении, может потребоваться увеличить значения переменных.

Изучите последние три строки кода конструктора `MainPage`:

```
public MainPage()
{
    ...
    int dataSize = bytesPerPixel * pixelWidth * pixelHeight;
    data = new byte[dataSize];

    graphBitmap = new WriteableBitmap(pixelWidth, pixelHeight);
}
```

В первых двух строках создается экземпляр байтового массива, в котором будут содержаться данные для графики. Размер этого массива зависит от разрешения `WriteableBitmap`-объекта, определяемого полями `pixelWidth` и `pixelHeight`. Кроме того, этот размер должен увеличиваться на тот объем памяти, который

потребуется для вывода на экран каждого пикселя: класс `WriteableBitmap` использует на каждый пиксель 4 байта, которые определяют красную, зеленую и синюю насыщенность в каждом пикселе и значение в нем альфа-смешения. (Альфа-смешение определяет степень прозрачности и яркости пикселя.)

Последняя инструкция создает `WriteableBitmap`-объект с указанным разрешением.

Изучите код метода `plotButton_Click`:

```
private void plotButton_Click(object sender, RoutedEventArgs e)
{
    Random rand = new Random();
    redValue = (byte)rand.Next(0xFF);
    greenValue = (byte)rand.Next(0xFF);
    blueValue = (byte)rand.Next(0xFF);
    Stopwatch watch = Stopwatch.StartNew();
    generateGraphData(data);

    duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";

    Stream pixelStream = graphBitmap.PixelBufferAsStream();
    pixelStream.Seek(0, SeekOrigin.Begin);
    pixelStream.Write(data, 0, data.Length);
    graphBitmap.Invalidate();
    graphImage.Source = graphBitmap;
}
```

Этот метод запускается, когда пользователь щелкает на кнопке `plotButton`.

Далее вы будете щелкать на этой кнопке несколько раз, что позволит увидеть, как при каждом создании приложением произвольного набора значений для насыщенности красного, зеленого и синего цвета выводимой на экран точки будет рисоваться новая версия графического изображения. (После каждого щелчка на этой кнопке графика будет другого цвета.)

Переменная `watch` является `System.Diagnostics.Stopwatch`-объектом. Тип `StopWatch` применяется для операций отсчета времени. Статический метод `StartNew`, относящийся к типу `StopWatch`, создает новый экземпляр объекта `StopWatch` и запускает его в работу. Время работы `StopWatch`-объекта можно получить, изучив свойство `ElapsedMilliseconds`.

Метод `generateGraphData` заполняет массив данными для графики, выводимой на экран с помощью объекта типа `WriteableBitmap`. Это метод будет изучен на следующем этапе выполнения упражнения.

По завершении работы метода `generateGraphData` затраченное на нее время (в миллисекундах) появится в элементе управления `TextBox` по имени `duration`.

Завершающий блок кода берет информацию, хранящуюся в массиве данных, и копирует ее в `WriteableBitmap`-объект для вывода графики на экран. Самый

простой технологический прием предполагает создание потока данных в памяти, который может использоваться для наполнения свойства `PixelBuffer` объекта типа `WriteableBitmap`. Затем метод `Write` этого потока можно использовать для копирования содержимого массива данных в этот буфер. Метод `Invalidate` класса `WriteableBitmap` требует, чтобы операционная система перерисовала побитовое изображение с использованием информации, хранящейся в буфере. Свойство `Source` элемента управления `Image` указывает на данные, которые должны быть отображены этим элементом. Последняя инструкция устанавливает в качестве значения свойства `Source` объект типа `WriteableBitmap`.

Изучите код метода `generateGraphData`:

```
private void generateGraphData(byte[] data)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    for (int x = 0; x < a; x++)
    {
        int s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y +
                (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y +
                (pixelHeight / 2)));
        }
    }
}
```

Этот метод выполняет ряд вычислений для построения точек весьма сложного графического изображения. (Само по себе вычисление роли не играет, оно просто генерирует привлекательную графику.) По мере вычисления каждой точки он вызывает метод `plotXY` для установки в массиве данных байтов, соответствующих этой точке. Точки для графического изображения отражаются относительно оси *x*, следовательно, метод `plotXY` для каждого вычисления вызывается два раза: первый раз для положительного значения *x*-координаты, второй раз — для отрицательного.

Изучите метод `plotXY`:

```
private void plotXY(byte[] data, int x, int y)
{
    int pixelIndex = (x + y * pixelWidth) * bytesPerPixel;
    data[pixelIndex] = blueValue;
```

```
    data[pixelIndex + 1] = greenValue;
    data[pixelIndex + 2] = redValue;
    data[pixelIndex + 3] = 0xBF;
}
```

Этот метод устанавливает значения байтов в массиве данных, соответствующих x - и y -координатам, передаваемых ему в качестве параметров. Каждая выводимая на экран точка соответствует пикселу, а каждый пиксель, как уже упоминалось, состоит из 4 байтов. Пиксели, оставшиеся неустановленными, отображаются в виде черных точек. Значение 0xBF для байта альфа-смешения показывает, что соответствующий пиксель должен быть выведен на экран с умеренной интенсивностью. Если это значение уменьшить, пиксель станет тусклее, а если значение будет установлено в 0xFF (максимальное значение для байта), пиксель отобразится с максимально большой интенсивностью.

Щелкните в меню Отладка на пункте Начать отладку, чтобы инициировать сборку и запуск приложения.

Когда появится окно Graph Demo, щелкните на кнопке Plot Graph (Построить изображение) и дождитесь результата. Здесь вам придется проявить терпение. Для создания и вывода на экран графического изображения приложению понадобится несколько секунд, и приложение не будет отвечать на запросы, пока это не произойдет. (Почему так происходит и как избежать подобного поведения, объясняется в главе 24.) Графическое изображение показано на рис. 23.1. Обратите внимание на значение под надписью Duration (ms). В данном случае для построения изображения понадобилось 4907 мс. Учтите, что сюда не было включено время, затраченное на сам вывод изображения на экран, что могло занять еще несколько секунд.



ПРИМЕЧАНИЕ Приложение было запущено на компьютере, имеющем 8 Гбайт памяти и четырехъядерный процессор, работающий с тактовой частотой 2,4 ГГц. Если вы пользуетесь более медленным или более быстрым процессором с другим количеством ядер или компьютером с более или менее емкой памятью, то ваши показатели времени могут быть другими.

Щелкните на кнопке Plot Graph еще раз и заметьте время, понадобившееся для перерисовки изображения. Повторите это действие несколько раз, чтобы получить среднее значение этого времени.



ПРИМЕЧАНИЕ Может оказаться, что от случая к случаю изображение появляется значительно позже (на его построение тратится более 30 с). Чаще всего такое случается, если вы приближаетесь к границе емкости памяти своего компьютера и Windows вынуждена организовывать постраничный обмен данными между памятью и диском. Если столкнетесь с подобным феноменом, не берите показанное время в расчет при вычислении среднего времени выполнения.

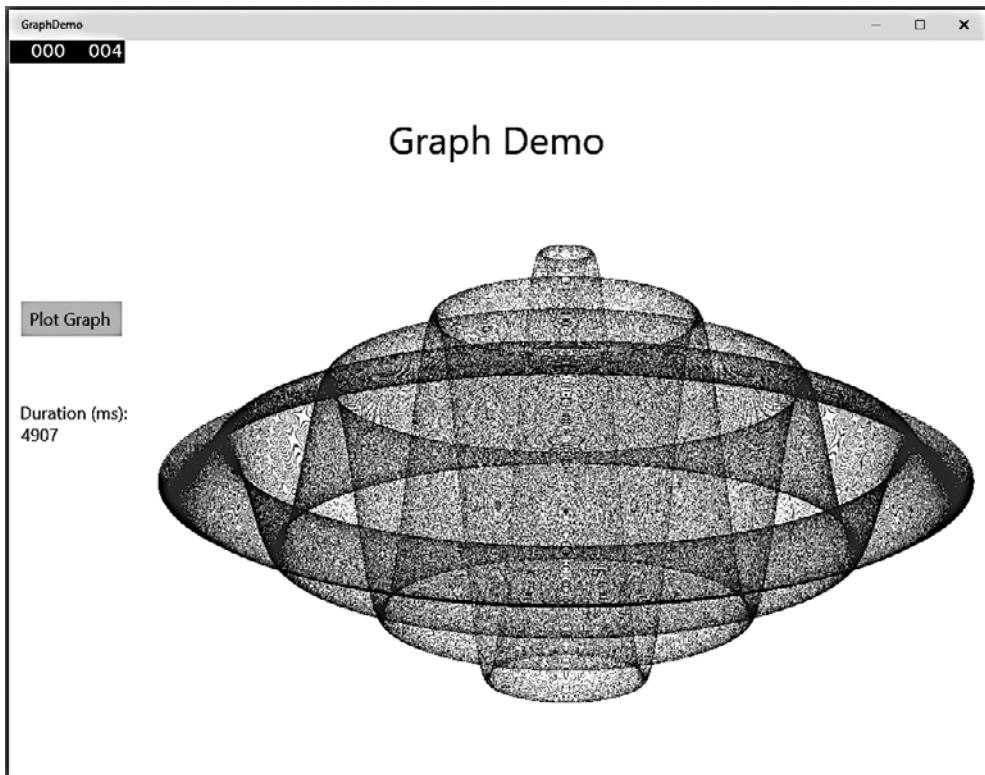


Рис. 23.1

Пускай приложение работает, а вы щелкните правой кнопкой мыши на пустом месте панели задач. Щелкните в появившемся контекстном меню на пункте Диспетчер задач. Щелкните в окне диспетчера задач на вкладке Производительность (Performance) и выведите на экран график использования центрального процессора. Если вкладка Производительность не видна, щелкните на пункте Подробнее (More Details). Щелкните правой кнопкой мыши на графике использования центрального процессора (CPU), укажите на пункт Изменить график (Change Graph To), а затем щелкните на пункте Общая загрузка (Overall Utilization). Тем самым вы заставите диспетчера задач показать на одном графике использование всех ядер процессора, запущенных на вашем компьютере. Настроенная таким образом на моем компьютере вкладка Производительность диспетчера задач показана на рис. 23.2.

Вернитесь в приложение Graph Demo и измените размер и местоположение окна приложения, а также окна диспетчера задач, чтобы они оба находились в поле зрения. Дождитесь выравнивания графика использования центрального процессора, а затем щелкните в окне Graph Demo на кнопке Plot Graph.

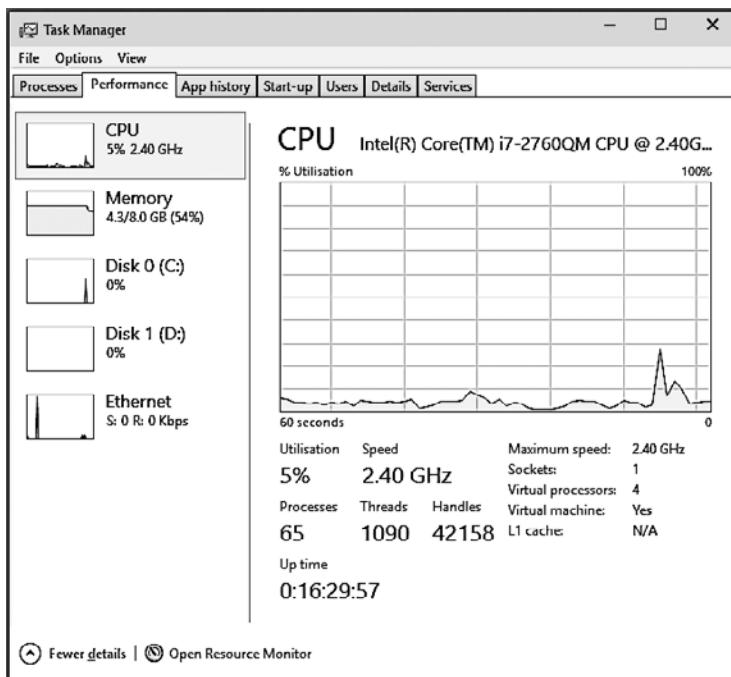


Рис. 23.2

Опять дождитесь выравнивания графика использования центрального процессора, а затем еще раз щелкните на кнопке Plot Graph.

Повторите последнее действие 16 раз, между щелчками дожидаясь выравнивания графика использования центрального процессора.

Понаблюдайте в окне диспетчера задач за использованием центрального процессора. Результаты, получаемые разными читателями, будут варьироваться, но на двухъядерном процессоре при создании изображения использование центрального процессора будет, скорее всего, на уровне 50–55 %. Как показано на рис. 23.3, на четырехъядерной машине использование процессора, вероятнее всего, окажется на уровне 25–30 %. Учтите, что на производительность могут влиять и другие факторы, например тип графической карты вашего компьютера: Вернитесь в среду Visual Studio 2015 и остановите отладку.

Теперь у вас есть от чего отталкиваться при анализе времени, затрачиваемого приложением на выполнение вычислений. Но при рассмотрении использования центрального процессора, показанного диспетчером задач, становится ясно, что приложение задействует не все его доступные ресурсы. На двухъядерной машине будет задействовано чуть больше половины мощности центрального процессора, а на четырехъядерной машине — чуть больше четверти. Причина

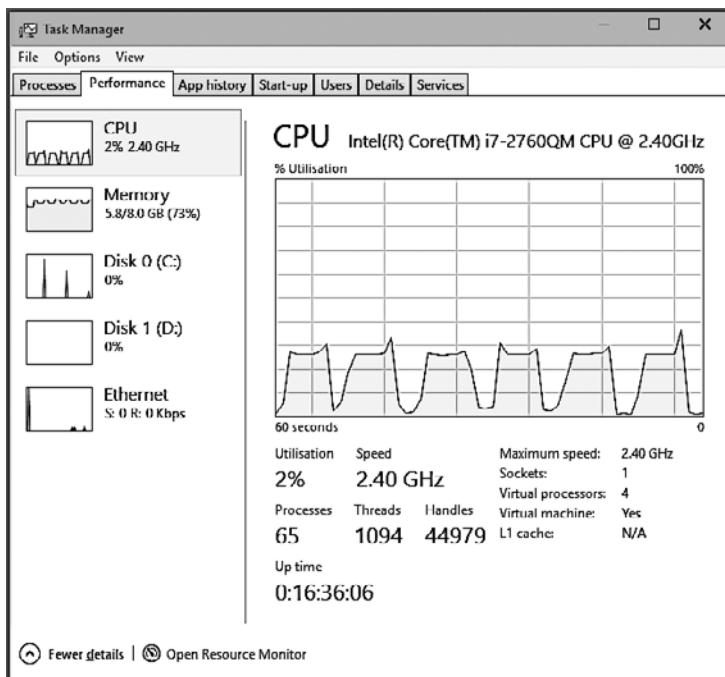


Рис. 23.3

этого феномена в однопоточности приложения, а в приложении Windows один поток может предоставить работу только одному ядру многоядерного процессора. Чтобы распределить нагрузку на все доступные ядра, нужно разбить приложение на задачи и организовать выполнение каждой задачи в отдельном потоке, где каждый поток будет запускаться на другом ядре. Именно этим вы дальше и займитесь.

Использование обозревателя производительности для обнаружения узких мест, где образуется большая нагрузка на центральный процессор

Приложение GraphDemo было специально разработано для создания в заранее известном месте (в методе `generateGraphData`) затора в работе центрального процессора. В реальном мире вы можете быть предупреждены, что в вашем приложении имеется нечто заставляющее его работать медленнее и не реагировать на действия пользователей, но вы можете не знать, где находится вредоносный код. В таком случае можно воспользоваться бесценной услугой обозревателя производительности и профилировщика среды Visual Studio.

Профилировщик может периодически замерять состояние выполнения приложения и собирать информацию о том, какая инструкция выполняется в данный

момент. Чем чаще выполняется конкретная строка кода и чем больше времени занимает ее выполнение, тем чаще эта инструкция будет наблюдаться. Профилировщик использует эти данные для генерации профиля выполнения и выдачи отчета, содержащего сведения о проблемных местах вашего кода. Эти сведения могут пригодиться при определении областей, на которых требуется сфокусировать свои усилия по оптимизации. Вы пройдете через этот процесс, выполнив следующие дополнительные действия.



ПРИМЕЧАНИЕ Обозреватель производительности и профилировщик в среде Visual Studio 2015 Community Edition недоступны.

В меню Отладка среды Visual Studio укажите на пункт Профилировщик, затем на пункт Обозреватель производительности, после чего щелкните на пункте Новый сеанс производительности. В среде Visual Studio должно появиться окно обозревателя производительности (рис. 23.4).

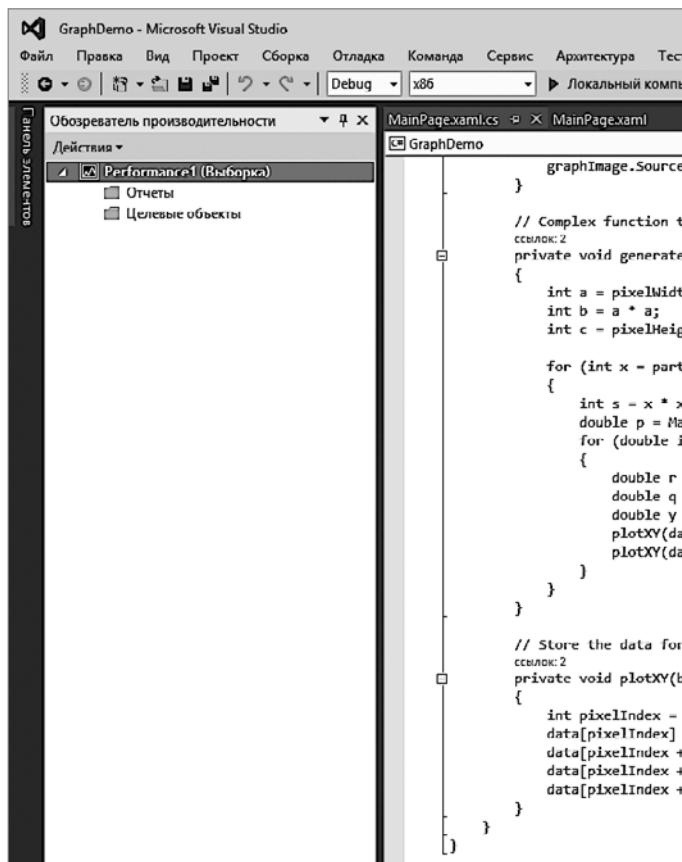


Рис. 23.4

В обозревателе производительности щелкните правой кнопкой мыши на пункте Целевые объекты, а затем на пункте Добавить целевой проект. В качестве целевого проекта будет добавлено приложение GraphDemo.

В панели меню обозревателя производительности щелкните на пункте Действия, а затем на пункте Запустить профилирование. Начнется выполнение приложения GraphDemo.

Щелкните на кнопке Plot Graph и дождитесь создания графического изображения. Повторите этот процесс несколько раз, после чего закройте приложение GraphDemo.

Вернитесь в среду Visual Studio и подождите, пока профилировщик проанализирует собранные данные замеров и сгенерирует отчет, который должен иметь примерно следующий вид (рис. 23.5).

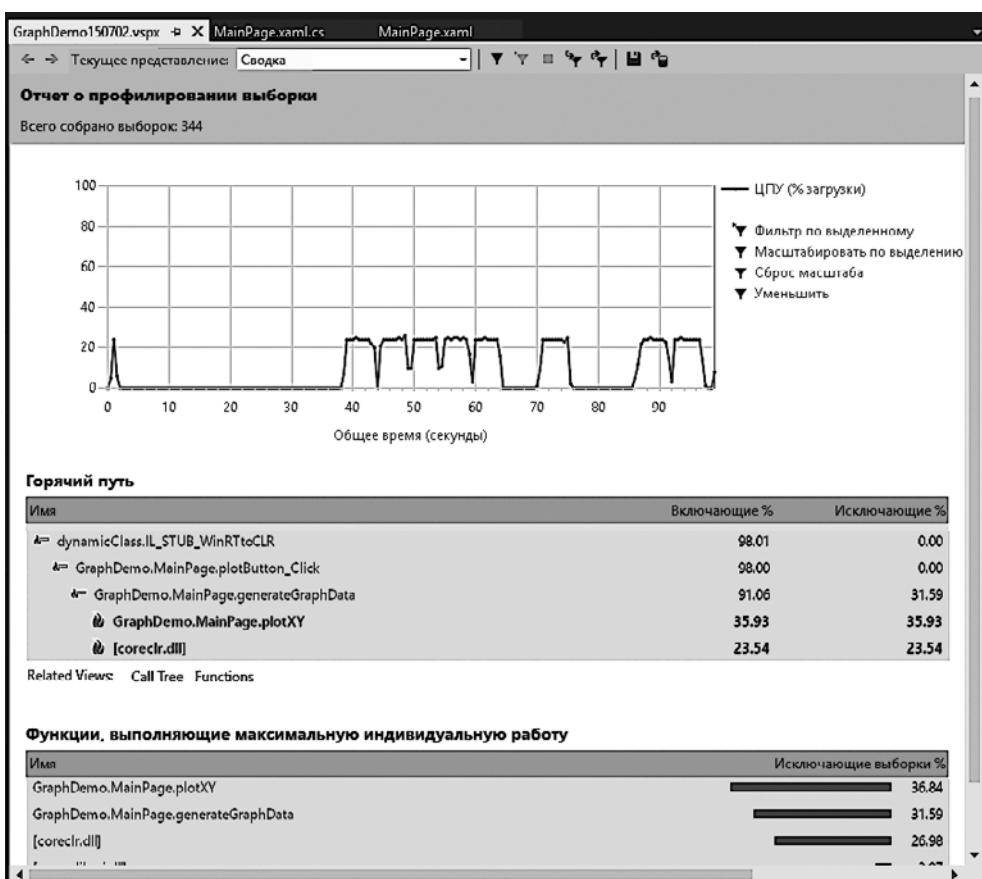


Рис. 23.5

В этом отчете показан график использования центрального процессора (он должен быть похож на тот, который вы видели ранее в диспетчере задач, с пиками, возникающими при каждом щелчке на кнопке Plot Graph) и Горячий путь для приложения. Этот путь идентифицирует ту последовательность в приложении, которая потребляет основные вычислительные мощности. В данном случае приложение тратит 98 % времени на метод `plotButton_Click`, 91,06 % – на выполнение метода `generateGraphData` и 35,93 % – на выполнение метода `plotXY`. Существенное количество времени (23,54 %) было потрачено также средой выполнения (`coreclr.dll`).

Обратите внимание на то, что вы можете увеличить конкретную область графика использования центрального процессора (щелчком кнопки мыши и перетаскиванием) и отфильтровать отчет, чтобы охватить только увеличенную часть выборочных данных.

В области Горячий путь щелкните на методе `GraphDemo.MainPage.generateGraphData`. Подробности, касающиеся метода, а также доля времени центрального процессора, потраченная на выполнение наиболее затратных инструкций, показываются в окне отчета (рис. 23.6).

В данном случае можно увидеть, что код в цикле `for` должен стать первоочередной целью для осуществления любых действий по оптимизации.

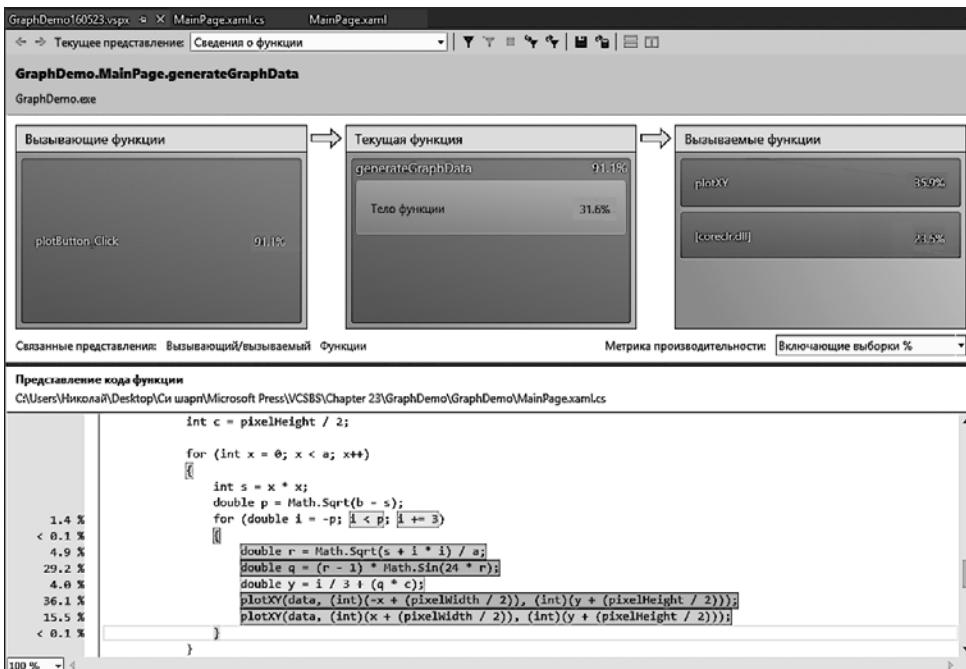


Рис. 23.6

Изменение приложения GraphDemo под использование Task-объектов

Вернитесь в среду Visual Studio 2015 и выведите в окно редактора файл MainPage.xaml.cs. Изучите метод generateGraphData.

Этот метод предназначен для заполнения массива данных элементами. С помощью внешнего **for**-цикла, основанного на переменной управления циклом **x** и выделенного в следующем примере кода жирным шрифтом, он совершает сквозной обход элементов массива:

```
private void generateGraphData(byte[] data)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;
    for (int x = 0; x < a; x++)
    {
        int s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y +
                (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y +
                (pixelHeight / 2)));
        }
    }
}
```

Вычисление, производимое одной итерацией этого цикла, не зависит от вычислений, производимых другими итерациями. Поэтому есть смысл разбить работу, выполняемую этим циклом, и запустить разные итерации на отдельных процессорах.

Измените определение метода generateGraphData, чтобы обеспечить прием двух дополнительных **int**-параметров с именами **partitionStart** и **partitionEnd**, дополнив его кодом, показанным жирным шрифтом:

```
private void generateGraphData(byte[] data, int partitionStart,
                                int partitionEnd)
{
    ...
}
```

Измените в методе generateGraphData внешний цикл **for** для проведения итераций между значениями **partitionStart** и **partitionEnd**, применив код, выделенный здесь жирным шрифтом:

```
private void generateGraphData(byte[] data, int partitionStart,
                               int partitionEnd)
{
    ...
    for (int x = partitionStart; x < partitionEnd; x++)
    {
        ...
    }
}
```

Добавьте к началу файла MainPage.xaml.cs в окне редактора следующую директиву **using**:

```
using System.Threading.Tasks;
```

Закомментируйте в методе **plotButton_Click** инструкцию, вызывающую метод **generateGraphData**, и добавьте к нему показанную жирным шрифтом инструкцию, создающую объект задачи типа **Task** и запускающую его на выполнение:

```
...
Stopwatch watch = Stopwatch.StartNew();
// generateGraphData(data);
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4));
...
```

Задача запускает код, указанный лямбда-выражением. Значения параметров **partitionStart** и **partitionEnd** показывают, что **Task**-объект вычисляет данные для первой половины графического изображения. (Данные для всего графического изображения содержат точки, выстраиваемые для значений между 0 и **pixelWidth / 2**.)

Добавьте еще одну инструкцию, показанную жирным шрифтом, которая создает и запускает в другом потоке второй объект задачи типа **Task**:

```
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4,
                                                pixelWidth / 2));
...
```

Этот **Task**-объект вызывает метод **generateGraphData** и вычисляет данные для значений между **pixelWidth / 4** и **pixelWidth / 2**.

Добавьте следующую показанную жирным шрифтом инструкцию, которая, прежде чем продолжить выполнение, будет ожидать завершения выполнения работы обоих **Task**-объектов:

```
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4,
                                                pixelWidth / 2));
Task.WaitAll(first, second);
...
```

Щелкните в меню Отладка на пункте Начать отладку, инициируя сборку и запуск приложения. Настройте изображение на экране так, чтобы можно было увидеть окно диспетчера задач, показывающее данные об использовании центрального процессора.

Щелкните в окне Graph Demo на кнопке Plot Graph. Дождитесь выравнивания графика использования центрального процессора в окне диспетчера задач.

Повторите предыдущее действие еще несколько раз, дожидаясь выравнивания графика использования центрального процессора между щелчками. Заметьте продолжительность, записываемую при каждом щелчке на кнопке, а затем вычислите ее среднее значение.

Вы должны увидеть, что приложение выполняется намного быстрее, чем раньше. На моем компьютере время сократилось на 2858 мс, то есть примерно на 40 %.

В большинстве случаев время, необходимое для выполнения вычислений, будет делиться примерно пополам, но в приложении все еще имеются однопоточные элементы, такие как логика, выводящая на экран графическое изображение после того, как будут сгенерированы данные. Именно поэтому общее время все же больше половины времени, затрачиваемого предыдущей версией приложения.

Перейдите в окно диспетчера задач. Вы должны заметить, что приложение использует больше ядер центрального процессора. На моей четырехъядерной машине использование центрального процессора после каждого щелчка на кнопке Plot Graph взлетает примерно на 50 %. Дело в том, что каждое из двух заданий запускается на отдельном ядре, но оставшиеся два ядра остаются незанятыми. Если у вас двухъядерная машина, то, скорее всего, вы увидите, что при каждом построении графического изображения использование процессора будет ненадолго достигать 100 % (рис. 23.7).

Если у вас четырехъядерный компьютер, степень использования центрального процессора можно повысить, еще больше сократив время путем добавления еще двух заданий, то есть двух Task-объектов, и разбиения работы метода `plotButton_Click` на четыре части, для чего нужно воспользоваться кодом, выделенным здесь жирным шрифтом:

```
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 8));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 8,
pixelWidth / 4));
Task third = Task.Run(() => generateGraphData(data, pixelWidth / 4,
pixelWidth * 3 / 8));
Task fourth = Task.Run(() => generateGraphData(data, pixelWidth * 3 / 8,
pixelWidth / 2));
Task.WaitAll(first, second, third, fourth);
...
```

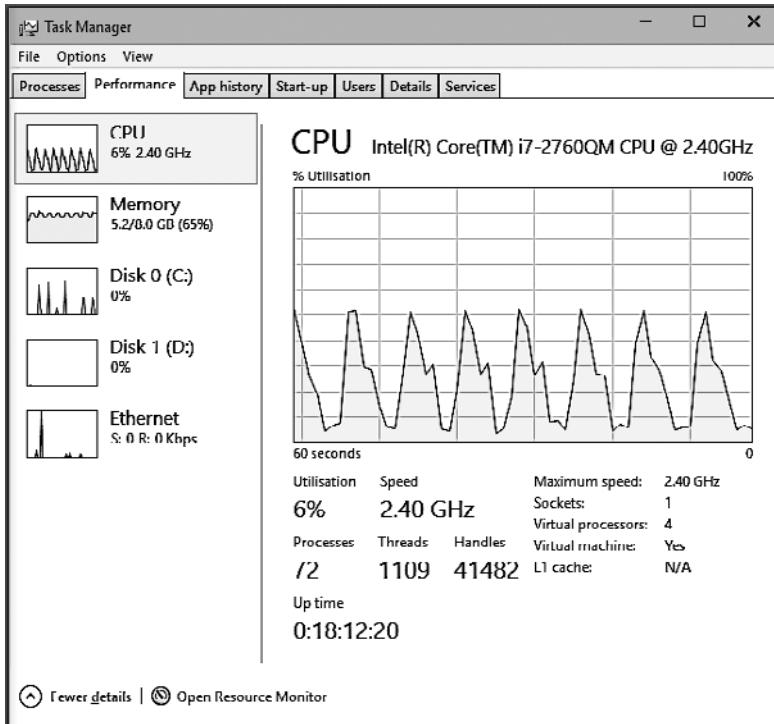


Рис. 23.7

Если у вашего процессора только два ядра, вы все равно можете попробовать пойти на эти изменения и должны заметить небольшой выигрыш по времени. Причина заключается в первую очередь в том, что алгоритм, используемый средой CLR, оптимизирует способ диспетчеризации потоков для каждой задачи.

Абстрагирование задач путем использования класса Parallel

Использование класса `Task` позволяет получить полный контроль над количеством задач, создаваемых вашим приложением. Но чтобы приспособиться к использованию `Task`-объектов, приходится вносить изменения в конструкцию приложения. Кроме того, нужно добавить код для синхронизации операций: приложение может выводить на экран графическое изображение только по завершении всех задач. В сложных приложениях синхронизация задач может стать весьма непростым процессом, в ходе которого весьма легко допустить ошибки.

Используя класс `Parallel`, вы можете распараллелить некоторые наиболее часто встречающиеся конструкции программирования, не меняя общую конструкцию

самого приложения. Выполняя внутреннюю работу, класс `Parallel` создает собственный набор задач в виде `Task`-объектов и автоматически синхронизирует эти задачи по мере их завершения. Класс `Parallel` находится в пространстве имен `System.Threading.Tasks` и предоставляет небольшой набор статических методов, который может использоваться для обозначения того, что код по возможности следует выполнять в параллельном режиме. К этим методам относятся:

- ❑ **Parallel.For.** Этот метод можно использовать вместо имеющейся в C# инструкции `for`. Он определяет цикл, в котором итерации с помощью определения задач могут выполняться параллельно. Этот метод имеет множество перегружаемых вариантов, но главный принцип у всех них один и тот же: вы указываете начальное значение, конечное значение и ссылку на метод, получающий целочисленный параметр. Метод выполняется для каждого значения между стартовым значением и значением, предшествующим указанному конечному значению, а параметр загружается целым числом, указывающим текущее значение. Рассмотрим, к примеру, следующий простой цикл `for`, который последовательно выполняет каждую итерацию:

```
for (int x = 0; x < 100; x++)  
{  
    // Выполнение циклической обработки  
}
```

В зависимости от обработки, выполняемой в теле цикла, может появиться возможность заменить этот цикл конструкцией `Parallel.For`, способной выполнять итерации в параллельном режиме:

```
Parallel.For(0, 100, performLoopProcessing);  
...  
private void performLoopProcessing(int x)  
{  
    // Выполнение циклической обработки  
}
```

Используя перегрузки метода `Parallel.For`, можно предоставить локальные данные, закрытые для каждого потока, указав различные необязательные параметры для создания задач, запускаемых методом `For`, и создать `ParallelLoopState`-объект, которым можно воспользоваться для передачи информации о состоянии другим одновременно выполняемым итерациям цикла. (Использование объекта типа `ParallelLoopState` будет рассмотрено чуть позже.)

- ❑ **Parallel.ForEach<T>.** Этим методом можно воспользоваться вместо имеющейся в C# инструкции `foreach`. Как и метод `For`, метод `ForEach` определяет цикл, итерации в котором могут выполняться в параллельном режиме. Вы указываете коллекцию, реализующую интерфейс-обобщение `IEnumerable<T>`,

и ссылку на метод, получающий один параметр типа *T*. Этот метод выполняется в отношении каждого элемента коллекции, и этот элемент передается методу в виде параметра. Доступны варианты перегрузки метода, с помощью которых можно предоставить закрытые локальные потоковые данные и указать необязательные параметры для создания задач, запускаемых методом *ForEach*.

- **Parallel.Invoke.** Этим методом можно воспользоваться для реализации в качестве параллельно выполняемых задач вызовов набора методов без параметров. Для этого с помощью делегата указывается список вызовов методов (или лямбда-выражений), не получающих параметры и не возвращающих значения. Каждый вызов метода может запускаться в отдельном потоке в любом порядке. Например, следующий код выполняет серию вызовов методов:

```
doWork();
doMoreWork();
doYetMoreWork();
```

Эти инструкции можно заменить следующим кодом,зывающим эти методы с использованием серий задач:

```
Parallel.Invoke(
    doWork,
    doMoreWork,
    doYetMoreWork
);
```

Следует иметь в виду, что класс *Parallel* определяет фактическую степень распараллеливания, подходящую для среды и рабочей нагрузки компьютера. Например, если метод *Parallel.ForEach* используется для реализации цикла, выполняющего 1000 итераций, классу *Parallel* совершенно необязательно создавать 1000 одновременно выполняемых задач (если только у вас не исключительно мощный компьютер с тысячью ядер). Вместо этого класс *Parallel* создаст оптимальное количество задач, поддерживая разумный баланс между доступными ресурсами и требованиями загруженности процессоров. В одной задаче могут выполняться несколько итераций, и задачи координируются друг с другом для определения того, какие именно итерации будет выполнять каждая из них. Важным последствием этого является невозможность гарантировать соблюдение порядка выполнения итераций, поэтому вы должны быть уверены в отсутствии зависимостей между итерациями, в противном случае можно, как будет показано далее, получить весьма неожиданные результаты.

В следующем упражнении вы вернетесь к исходной версии приложения *GraphDemo* и воспользуетесь для одновременного выполнения операций классом *Parallel*.

Использование класса Parallel для распараллеливания операций в приложении GraphDemo

Откройте в среде Visual Studio 2015 решение GraphDemo, которое находится в папке \Microsoft Press\VCBS\Chapter 23\Parallel GraphDemo вашей папки документов. Это копия исходной версии приложения GraphDemo без использования задач.

Раскройте в проекте GraphDemo, показанном в обозревателе решений, узел MainPage.xaml и дважды щелкните на файле MainPage.xaml.cs, чтобы его код был выведен в окно редактора.

Добавьте к списку в начале файла следующую директиву using:

```
using System.Threading.Tasks;
```

Найдите метод generateGraphData. Он должен выглядеть следующим образом:

```
private void generateGraphData(byte[] data)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    for (int x = 0; x < a; x++)
    {
        int s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y +
                (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y +
                (pixelHeight / 2)));
        }
    }
}
```

Первым кандидатом на распараллеливание является внешний цикл `for`, совершающий итерации под управлением целочисленной переменной `x`. Можно также присмотреться к внутреннему циклу, управляемому переменной `i`, но для распараллеливания этого цикла из-за типа, к которому принадлежит `i`, потребуется больше усилий. (Методы в классе `Parallel` предполагают, что управляющая переменная будет целочисленной.) Кроме того, при наличии вложенных циклов, таких как те, которые встречаются в этом коде, лучше сначала распараллеливать внешние циклы, а затем выполнять тестирование, чтобы определить, является ли производительность приложения достаточной. Если нет, следует пройти через вложенные циклы и распараллелить их в направлении от внешних к внутренним, тестируя производительность после изменения каждого из

них. Обнаружится, что во многих случаях распараллеливание внешних циклов оказывает на производительность наиболее существенное влияние, в то время как эффект от изменения внутренних циклов становится все более скромными.

Вырежьте код из тела цикла `for` и создайте на основе этого кода новый закрытый `void`-метод по имени `calculateData`. Этот метод должен получать `int`-параметр по имени `x` и байтовый массив по имени `data`. Кроме этого, переместите инструкции, объявляющие локальные переменные `a`, `b` и `c`, из метода `generateGraphData` в начало метода `calculateData`. Метод `generateGraphData` после удаления кода и метод `calculateData` показаны в следующем примере (этот код пока что не нужно компилировать):

```
private void generateGraphData(byte[] data)
{
    for (int x = 0; x < a; x++)
    {
    }
}
private void calculateData(int x, byte[] data)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    int s = x * x;
    double p = Math.Sqrt(b - s);
    for (double i = -p; i < p; i += 3)
    {
        double r = Math.Sqrt(s + i * i) / a;
        double q = (r - 1) * Math.Sin(24 * r);
        double y = i / 3 + (q * c);
        plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
    }
}
```

Замените в методе `generateGraphData` цикл `for` следующей инструкцией, вызывающей статический метод `Parallel.For`:

```
private void generateGraphData(byte[] data)
{
    Parallel.For(0, pixelWidth / 2, x => calculateData(x, data));
}
```

Этот код является вычисляемым параллельно эквивалентом исходного цикла `for`. Он перебирает значения от 0 до `pixelWidth / 2 - 1` включительно. Каждый вызов выполняется с использованием задачи, а каждая задача может иметь более одной итерации. Метод `Parallel.For` завершает свою работу только тогда, когда работу завершат все созданные им задачи. Вспомним, что в качестве последнего параметра метод `Parallel.For` ожидает указания метода, получающего

один целочисленный параметр. Он вызывает этот метод, передавая в качестве параметра текущий индекс цикла. В данном примере метод `calculateData` не соответствует требуемой сигнатуре, поскольку он получает два параметра: целочисленное значение и байтовый массив. Поэтому код использует лямбда-выражение, которое действует в качестве адаптера,зывающего метод `calculateData` с подходящими аргументами.

В меню Отладка щелкните на пункте Начать отладку, чтобы инициировать сборку и запуск приложения.

В окне Graph Demo щелкните на кнопке Plot Graph. Когда в окне Graph Demo появится графическое изображение, запишите время, затраченное на его создание. Чтобы получить среднее значение, повторите это действие несколько раз.

Нетрудно будет заметить, что приложение выполняется практически так же быстро, как и его предыдущая версия, в которой использовались Task-объекты (и даже, может быть, быстрее, в зависимости от количества доступных центральных процессоров). Если изучить данные диспетчера задач, можно будет заметить, что пик использования центрального процессора приближается к 100 % независимо от того, какой компьютер используется, двух- или четырехядерный (рис. 23.8).

Вернитесь в среду Visual Studio и остановите отладку.

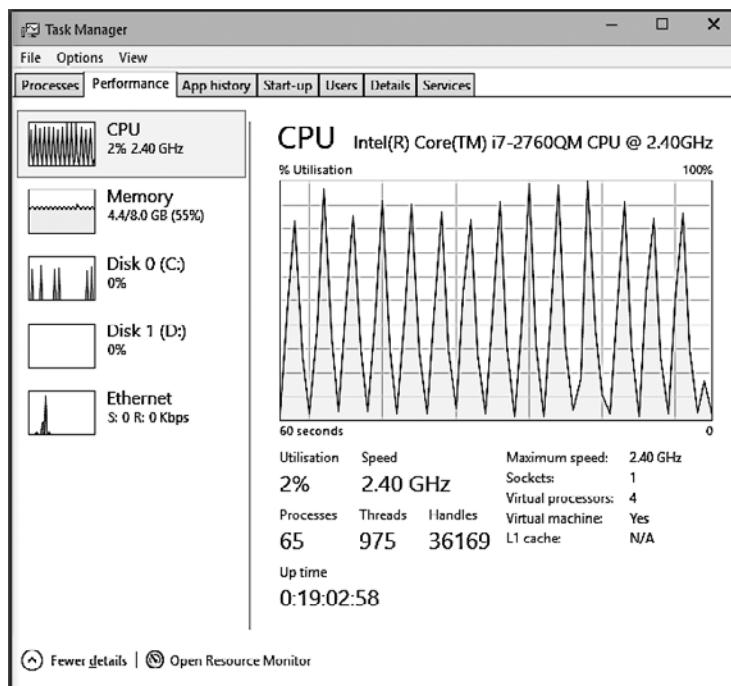


Рис. 23.8

Когда не нужно использовать класс Parallel

Следует знать, что несмотря на широкую представленность команды разработчиков .NET Framework в компании Microsoft и приложенные ею немалые усилия, класс `Parallel` не волшебный, его нельзя использовать без должной осмотрительности, просто ожидая, что ваше приложение вдруг заработает намного быстрее и выдаст те же результаты. Класс `Parallel` предназначен для распараллеливания требовательных к вычислительной мощности центрального процессора независимых областей кода.

Если выполняемый код не требует от центрального процессора больших вычислительных мощностей, распараллеливание может не привести к повышению производительности. В этом случае издержки на создание задачи, ее запуск в отдельном потоке и ожидание ее завершения будут, скорее всего, выше затрат на непосредственный запуск того или иного метода. Дополнительные издержки при каждом вызове метода могут исчисляться всего несколькими миллисекундами, но вы должны учитывать количество запусков метода. Если вызов метода находится во вложенном цикле и выполняется тысячи раз, то все эти небольшие издержки будут складываться. Главное правило для `Parallel.Invoke` заключается в том, что его нужно использовать, лишь когда это целесообразно. Метод `Parallel.Invoke` нужно оставить для операций, производящих интенсивные вычисления, в противном случае издержки на создание задач и управление ими могут фактически замедлить выполнение приложения.

Другое основное соображение для использования класса `Parallel` заключается в том, что параллельно выполняемые операции должны быть независимыми. Например, если попытаться воспользоваться методом `Parallel.For` для распараллеливания цикла, итерации в котором зависят друг от друга, результаты будут непредсказуемыми.

Чтобы понять, что я имею в виду, посмотрите на следующий код (этот пример можно найти в решении `ParallelLoop`, которое находится в папке `\Microsoft Press\VCBS\Chapter 23\ParallelLoop` вашей папки документов):

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace ParallelLoop
{
    class Program
    {
        private static int accumulator = 0;

        static void Main(string[] args)
        {
            for (int i = 0; i < 100; i++)
            {
                Parallel.Invoke(
                    () => accumulator += i * i,
                    () => accumulator += i * i
                );
            }
        }
    }
}
```

```
        {
            AddToAccumulator(i);
        }
        Console.WriteLine($"Accumulator is {accumulator}");
    }

private static void AddToAccumulator(int data)
{
    if ((accumulator % 2) == 0)
    {
        accumulator += data;
    }
    else
    {
        accumulator -= data;
    }
}
```

Эта программа последовательно перебирает значения от 0 до 99 и поочередно для каждого значения вызывает метод `AddToAccumulator`. Этот метод исследует текущее значение переменной `accumulator` и, если оно является четным числом, складывает значение параметра со значением переменной `accumulator`, в противном случае вычитает значение параметра. Результат выводится на экран в конце программы. По окончании работы этой программы должно получиться 100.

Чтобы повысить степень распараллеливания в этом простом приложении, у вас может возникнуть соблазн заменить цикл `for` в методе `Main` вызовом метода `Parallel.For`:

```
static void Main(string[] args)
{
    Parallel.For (0, 100, AddToAccumulator);
    Console.WriteLine($"Accumulator is {accumulator}");
}
```

Но это не дает гарантий того, что задачи, созданные для выполнения различных вызовов метода `AddToAccumulator`, будут выполняться в какой-то определенной последовательности. (Код также не защищен от одновременной работы нескольких потоков, поскольку потоки, выполняющие задачи, могут пытаться обновлять значение переменной `accumulator` одновременно.) Значение, вычисляемое методом `AddToAccumulator`, зависит от обслуживаемой последовательности, поэтому результат внесенных изменений выразится в том, что теперь при каждом запуске приложение может выдавать разные значения. В данном наиболее простом случае вы можете фактически не увидеть никаких отличий вычисленного значения от предыдущих значений, поскольку метод `AddToAccumulator` выполняется очень быстро и среда .NET Framework может выбрать последовательный запуск каждого вызова путем использования одного и того же потока. Но если

внести в метод `AddToAccumulator` следующее изменение (выделенное жирным шрифтом), вы станете получать разные результаты:

```
private static void AddToAccumulator(int data)
{
    if ((accumulator % 2) == 0)
    {
        accumulator += data;
        Thread.Sleep(10); // Ожидание в течение 10 мс
    }
    else
    {
        accumulator -= data;
    }
}
```

Метод `Thread.Sleep` просто заставляет текущий поток находиться в состоянии ожидания в течение указанного времени. Это изменение имитирует поток, выполняющий дополнительную обработку, и влияет на способ диспетчеризации задач классом `Parallel`, который теперь запускает задачи в различных потоках, что выражается в разной последовательности их выполнения.

Основное правило использования методов `Parallel.For` и `Parallel.ForEach` заключается в том, что ими следует пользоваться только при условии независимости каждой итерации цикла и тщательного тестирования кода. Подобные соображения применяются и к методу `Parallel.Invoke`: использование этой конструкции для вызовов методов допустимо при условии их независимости друг от друга и при том, что приложение не зависит от их запуска в определенной последовательности.

Отмена задач и обработка исключений

Основным требованием к приложениям, выполняющим долговременные операции, является наличие возможности в случае необходимости остановить эти операции. Но вы не можете просто прервать задачу, поскольку данные вашего приложения могут остаться в неопределенном состоянии. Вместо этого класс `Task` реализует стратегию согласованной отмены, позволяющей задаче выбрать удобную точку остановки обработки, а также позволяет классу при необходимости произвести откат любой работы, выполненной до отмены.

Механизмы согласованной отмены

Согласованная отмена основана на понятии признака отмены. Признак отмены является структурой, представляющей запрос на отмену выполнения одной или

нескольких задач. Метод, выполняемый задачей, должен включать параметр `System.Threading.CancellationToken`. Приложение, которому требуется отменить задачу, устанавливает булево свойство `IsCancellationRequested` этого параметра в `true`. Метод, выполняемый в задаче, может запросить это свойство на различных этапах своего выполнения. Если на любом этапе это свойство установлено в `true`, он знает, что приложение запросило отмену задачи. Также метод знает, какую работу он проделал до сих пор, следовательно, он, если нужно, может выполнить откат любых изменений и только потом завершить работу. Как вариант, метод может просто проигнорировать запрос и продолжить выполнение.



СОВЕТ Признак отмены в задаче можно исследовать довольно часто, но частота проверок не должна негативно сказываться на производительности выполнения задачи. По возможности нужно нацелиться на проверку признака отмены не реже чем каждые 10 мс, но не чаще чем 1 мс.

Приложение получает `CancellationToken` путем создания объекта типа `System.Threading.CancellationTokenSource` и запроса свойства `Token` этого объекта. Затем приложение может передать этот `CancellationToken`-объект в качестве параметра любому методу, запускаемому задачами, создаваемыми и выполняемыми приложением. Если приложение нуждается в отмене задач, оно вызывает метод `Cancel` объекта типа `CancellationTokenSource`. Этот метод устанавливает значение свойства `IsCancellationRequested` объекта `CancellationToken`, передаваемое всем задачам.

В следующем примере кода показано, как создается признак отмены и как он используется для отмены задачи. Метод `initiateTasks` создает экземпляр переменной `cancellationTokenSource` и получает ссылку на `CancellationToken`-объект, доступный через эту переменную. Затем код создает и запускает задачу, выполняющую метод `doWork`. Чуть позже код вызывает метод `Cancel`, принадлежащий источнику признака отмены, который устанавливает признак отмены. Метод `doWork` запрашивает свойство `IsCancellationRequested`, принадлежащее признаку отмены. Если свойство установлено в `true`, метод завершает работу, в противном случае он продолжает выполнение.

```
public class MyApplication
{
    ...
    // Метод, создающий задачу и управляющий ею
    private void initiateTasks()
    {
        // Создание признака отмены и получение этого признака
        CancellationTokenSource cancellationTokenSource = new
            CancellationTokenSource();
        CancellationToken cancellationToken = cancellationTokenSource.Token;
        // Создание задачи и запуск ее на выполнение метода doWork
        Task myTask = Task.Run(() => doWork(cancellationToken));
    }
}
```

```
...
if (...)

{
    // Отмена задачи
    cancellationSource.Cancel();
}

...

}

// Метод, запускаемый задачей
private void doWork(CancellationToken token)
{
    ...

    // Если признак отмены установлен в true, завершение обработки
    if (token.IsCancellationRequested)
    {
        // Уборка и завершение
        ...
        return;
    }
    // Если задача не отменена, продолжение обычного выполнения
    ...
}
}
```

В добавок к предоставлению высокой степени контроля над отменой обработки этот подход масштабируется на любое количество задач: можно запустить несколько задач и передать каждой из них один и тот же `CancellationToken`-объект. Если вызвать в объекте типа `CancellationTokenSource` метод, каждая задача проверит установку для свойства `IsCancellationRequested` значения `true` и будет действовать соответствующим образом.

Воспользовавшись методом `Register`, можно также зарегистрировать метод обратного вызова (в форме `Action`-делегата) с признаком отмены. Когда приложение вызывает метод `Cancel` соответствующего объекта типа `CancellationTokenSource`, запускается этот метод обратного вызова. Но вы не можете указать конкретный момент выполнения этого метода — это может случиться до или после выполнения задачами их собственной обработки отмены или даже в ходе этого процесса:

```
...
cancellationToken.Register(doAdditionalWork);
...
private void doAdditionalWork()
{
    // Выполнение дополнительной обработки отмены
}
```

В следующем упражнении вы добавите возможность отмены к приложению `GraphDemo`.

Добавление возможности отмены к приложению GraphDemo

Откройте в среде Visual Studio 2015 решение GraphDemo, которое находится в папке \Microsoft Press\VCSBS\Chapter 23\GraphDemo With Cancellation вашей папки документов.

Это полная копия приложения GraphDemo из ранее показанного упражнения, в которой для повышения производительности вычислений используются задачи. Также пользовательский интерфейс включает кнопку `cancelButton`, которой можно воспользоваться для остановки задач, вычисляющих данные для графического изображения.

В проекте GraphDemo в обозревателе решений сделайте двойной щелчок на файле `MainPage.xaml`, чтобы вывести форму в окно конструктора. Обратите внимание на появление в левой панели формы кнопки `Cancel` (Отмена).

Откройте в окне редактора файл `MainPage.xaml.cs`. Найдите метод `Button_Click`. Этот метод запускается, когда пользователь щелкает на кнопке `Cancel` (Отмена). Код в нем пока отсутствует.

Добавьте к списку в начале файла следующую директиву `using`:

```
using System.Threading;
```

В этом пространстве имен размещаются типы, используемые для согласованной отмены.

Добавьте к классу `MainPage` поле типа `CancellationTokenSource` по имени `tokenSource` и, как показано далее жирным шрифтом, инициализируйте его значением `null`:

```
public sealed partial class MainPage : Page
{
    ...
    private byte redValue, greenValue, blueValue;
    private CancellationTokenSource tokenSource = null;
    ...
}
```

Найдите метод `generateGraphData` и добавьте к определению метода параметр типа `CancellationToken` под названием `token`, код которого выделен здесь жирным шрифтом:

```
private void generateGraphData(byte[] data, int partitionStart,
                               int partitionEnd, CancellationToken token)
{
    ...
}
```

Добавьте в начало внутреннего цикла `for` в методе `generateGraphData` следующий код, показанный жирным шрифтом, для проверки запроса отказа. Если отказ запрошен, выполняется возврат из метода, в противном случае продолжается вычисление значений и построение графического изображения:

```
private void generateGraphData(byte[] data, int partitionStart,
                               int partitionEnd, CancellationToken token)
{
    int a = pixelWidth / 2;
    int b = a * a;
    int c = pixelHeight / 2;

    for (int x = partitionStart; x < partitionEnd; x++)
    {
        int s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            if (token.IsCancellationRequested)
            {
                return;
            }

            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y +
                (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y +
                (pixelHeight / 2)));
        }
    }
}
```

Добавьте в метод `plotButton_Click` следующие инструкции, выделенные жирным шрифтом, которые создают экземпляр переменной типа `tokenSource` и извлекают `CancellationToken`-объект в переменную по имени `token`:

```
private void plotButton_Click(object sender, RoutedEventArgs e)
{
    Random rand = new Random();
    redValue = (byte)rand.Next(0xFF);
    greenValue = (byte)rand.Next(0xFF);
    blueValue = (byte)rand.Next(0xFF);

    tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;
    ...
}
```

Измените инструкции, создающие и запускающие две задачи, и передайте методу `generateGraphData` в качестве последнего параметра переменную `token`:

```
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4,
                                               token));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4,
                                                pixelWidth / 2, token));
...
```

Отредактируйте определение метода `plotButton_Click` и добавьте модификатор `async`, показанный здесь жирным шрифтом:

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
}
```

Закомментируйте в теле метода `plotButton_Click` инструкцию `Task.WaitAll`, которая ожидает завершения выполнения задач, и замените ее следующими инструкциями, выделенными жирным шрифтом, которые используют оператор `await`:

```
...
// Task.WaitAll(first, second);
await first;
await second;

duration.Text = string.Format(...);
...
```

Потребности в изменениях, произведенных на двух последних этапах, связаны с однопоточной природой пользовательского интерфейса Windows. В нормальных условиях, когда приступает к работе обработчик события для такого элемента пользовательского интерфейса, как кнопка, обработчики событий для других компонентов пользовательского интерфейса блокируются, пока не завершит свою работу первый обработчик (даже если в обработчике события используются задачи). В данном примере использование метода `Task.WaitAll` для ожидания завершения выполнения задач превратит `Cancel` в бесполезную кнопку, поскольку обработчик события для кнопки `Cancel` не будет запущен, пока не завершит свою работу обработчик для кнопки `Plot Graph`, и в этом случае в попытках отмены операции не будет никакого смысла. Фактически, как уже было упомянуто, когда происходит щелчок на кнопке `Plot Graph`, пользовательский интерфейс становится совершенно невосприимчивым до тех пор, пока графическое изображение не появится на экране и метод `plotButton_Click` не завершит свою работу.

Для того чтобы справиться с подобными ситуациями, был придуман оператор `await`. Этим оператором можно воспользоваться только внутри метода, помеченного ключевым словом `async`. Его целями являются освобождение текущего

потока и ожидание завершения задачи в фоновом режиме. Как только задача завершится, управление вернется методу, который продолжит свое выполнение со следующей инструкции. В данном примере две инструкции `await` просто позволяют каждой из задач завершиться в фоновом режиме. После того как вторая задача завершится, метод продолжит свое выполнение, показывая время, затраченное на выполнение этих задач, в элементе управления типа `TextBlock` по имени `duration`. Применение `await` в отношении задачи, которая уже завершилась, ошибкой не считается, просто оператор `await` тут же передаст управление следующей инструкции.



ПРИМЕЧАНИЕ Более подробно модификатор `async` и оператор `await` рассматриваются в главе 24.

Найдите метод `cancelButton_Click`. Добавьте к этому методу код, показанный жирным шрифтом:

```
private void cancelButton_Click(object sender, RoutedEventArgs e)
{
    if (tokenSource != null)
    {
        tokenSource.Cancel();
    }
}
```

Этот код проверяет наличие экземпляра переменной `tokenSource`. Если он имеется, код вызывает в отношении этой переменной метод `Cancel`.

В меню Отладка щелкните на пункте Начать отладку, инициируя сборку и запуск приложения.

В окне `GraphDemo` щелкните на кнопке `Plot Graph` и убедитесь в появлении, как и прежде, графического изображения. Но теперь вы должны заметить, что на построение изображения затрачено немного больше времени, чем раньше. Причина кроется в дополнительной проверке, выполняемой методом `generateGraphData`.

Щелкните на кнопке `Plot Graph` еще раз, а затем сразу же щелкните на кнопке `Cancel`.

Если вы не промедлили и щелкнули на кнопке `Cancel` до того, как были сгенерированы данные для графического изображения, это действие повлечет за собой возвращение из методов, запущенных задачами. Данные до конца не сформировались, поэтому графическое изображение появится с пустотами (рис. 23.9). (Несмотря на пустоты, предыдущее графическое изображение будет по-прежнему отображаться на экране, а размеры этих пустот зависят от того, насколько быстро вы щелкнули на кнопке `Cancel`.)

Вернитесь в среду Visual Studio и остановите отладку.

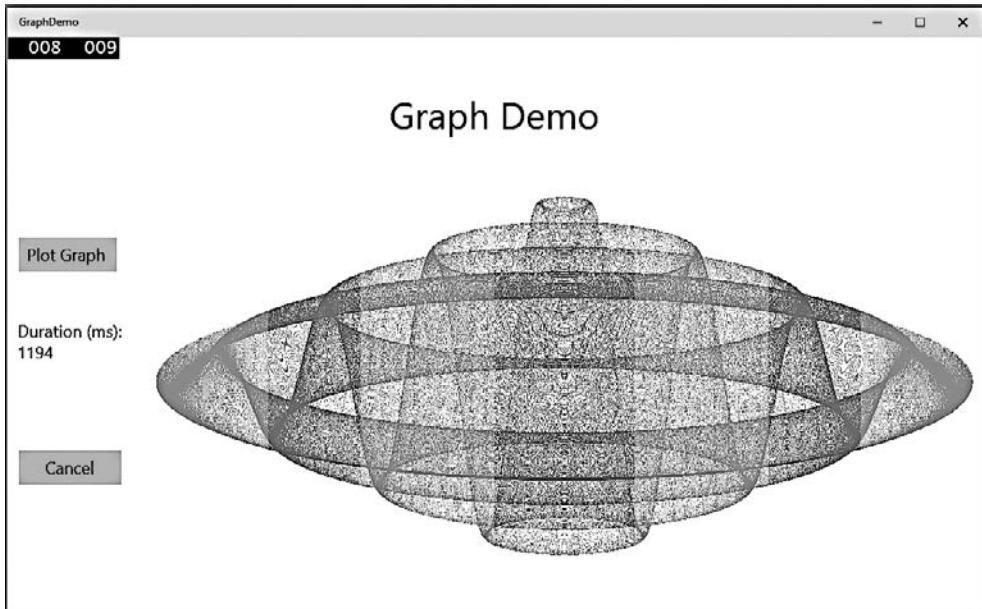


Рис. 23.9

Изучив свойство `Status` объекта типа `Task`, можно определить, была задача завершена или же прервана. Свойство `Status` содержит значение из перечисления типа `System.Threading.Tasks.TaskStatus`. Некоторые значения состояния, с которыми вы можете столкнуться, рассмотрены в следующем списке (есть и другие значения).

- Created.** Это начальное состояние задачи. Она была создана, но еще не спланирована на выполнение.
- WaitingToRun.** Задача была спланирована на выполнение, но оно еще не началось.
- Running.** Задача выполняется потоком.
- RanToCompletion.** Задача успешно завершилась без каких-либо необработанных исключений.
- Canceled.** Задача была прервана до того, как началось ее выполнение, или же она распознала отмену и завершилась без выдачи исключения.
- Faulted.** Задача прекратила выполнение по причине возникновения исключения.

В следующем упражнении вы предпримете попытку создания отчета о состоянии каждой задачи, чтобы получить возможность узнать о ее завершении или отмене.

ОТМЕНА ВЫПОЛНЯЕМОГО В ПАРАЛЛЕЛЬНОМ РЕЖИМЕ ЦИКЛА FOR ИЛИ LOOP

Методы Parallel.For и Parallel.ForEach не предоставляют вам прямого доступа к созданным Task-объектам. Фактически вы даже не знаете, сколько задач запущено, — среда .NET Framework использует свои собственные эвристические алгоритмы для определения оптимального количества используемых задач на основе доступных ресурсов и текущей рабочей нагрузки компьютера.

Если нужно остановить работу метода Parallel.For или Parallel.ForEach досрочно, следует воспользоваться объектом типа ParallelLoopState. Метод, указанный в качестве тела цикла, должен включать дополнительный параметр ParallelLoopState. Класс Parallel создает ParallelLoopState-объект и передает его методу в качестве параметра. Класс Parallel использует этот объект для хранения информации о каждом вызове метода. Метод может вызвать метод Stop этого объекта, чтобы выставить признак отмены классом Parallel попыток выполнения любых итераций, кроме тех, которые уже были запущены и завершены. В следующем примере показан метод Parallel.For, вызывающий для каждой итерации метод doLoopWork. Метод doLoopWork исследует значение переменной итерации: если оно больше 600, метод вызывает метод Stop параметра ParallelLoopState. Это приводит к тому, что метод Parallel.For прекращает запуск последующих итераций цикла. (Уже запущенные итерации могут продолжиться до завершения.)



ПРИМЕЧАНИЕ Не забудьте, что итерации в цикле Parallel.For не запускаются в определенной последовательности. Следовательно, прекращение цикла при получении переменной итерации значения 600 не гарантирует, что предыдущие 599 итераций уже отработали. Более того, уже могут быть завершены некоторые итерации со значениями больше 600:

```
Parallel.For(0, 1000, doLoopWork);
...
private void doLoopWork(int i, ParallelLoopState p)
{
    ...
    if (i > 600)
    {
        p.Stop();
    }
}
```

Отображение состояния каждой задачи

Выполните в окне конструктора среды Visual Studio файл MainPage.xaml. Добавьте в XAML-панели для определения формы MainPage следующую разметку, выделенную жирным шрифтом, расположив ее до предпоследнего </Grid>-тега:

```
<Image x:Name="graphImage" Grid.Column="1" Stretch="Fill" />
</Grid>
<TextBlock x:Name="messages" Grid.Row="4" FontSize="18"
HorizontalAlignment="Left"/>
</Grid>
</Page>
```

Эта разметка добавляет к нижней части формы элемент управления типа `TextBlock` по имени `messages`.

Выполните в окне редактора файл `MainPage.xaml.cs` и найдите метод `plotButton_Click`. Добавьте к этому методу код, показанный ниже жирным шрифтом. Имеющиеся в нем инструкции создают строку, которая содержит состояние каждой задачи после завершения ее выполнения, а затем выведите эту строку в элементе управления `messages` типа `TextBlock`, который находится в нижней части формы.

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    await first;
    await second;

    duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";

    string message = $"Status of tasks is {first.Status}, {second.Status}";
    messages.Text = message;
    ...
}
```

В меню Отладка щелкните на пункте Начать отладку.

В окне `GraphDemo` щелкните на кнопке `Plot Graph`, но не щелкайте на кнопке `Cancel`. Убедитесь в том, что в выведенном сообщении о состоянии задач говорится (два раза), что их состояние соответствует выполнению до полного завершения — `RanToCompletion`.

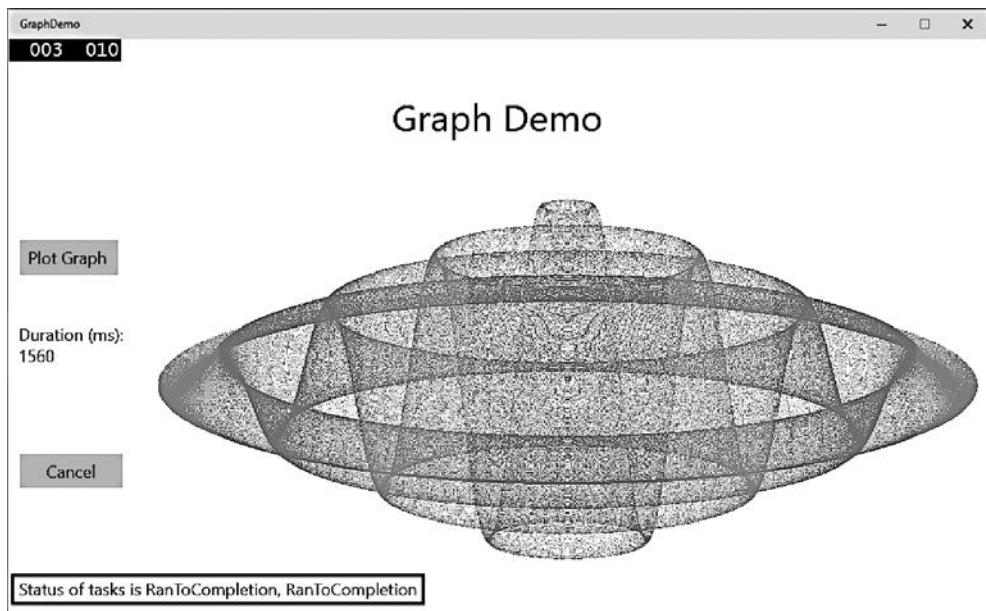
Щелкните еще раз в окне `GraphDemo` на кнопке `Plot Graph`, а затем без промедления — на кнопке `Cancel`.

Как ни удивительно, но появившееся сообщение по-прежнему свидетельствует о состоянии каждой задачи `RanToCompletion`, даже при том что графическое изображение появляется с пустотами (рис. 23.10).

Дело в том, что, несмотря на отправку каждой задаче запроса на отмену путем использования признака отмены, методы, запущенные этими задачами, просто возвращают управление. Среда выполнения .NET Framework не знает, были ли задачи отменены или же им было позволено доработать до конца, и она просто игнорирует запросы на отмену.

Вернитесь в среду Visual Studio и остановите отладку.

Итак, как же тогда создать признак того, что задача была отменена и не получила возможности быть выполненной до конца? Ответ кроется в объекте типа `CancellationToken`, передаваемом в качестве параметра методу, запускаемому задачей. Класс `CancellationToken` предоставляет метод по имени `Throw-`

**Рис. 23.10**

`IfCancellationRequested`. Этот метод тестирует свойство `IsCancellationRequested` признака отмены: если оно имеет значение `true`, метод выдает исключение `OperationCanceledException` и прерывает выполнение метода, запущенного задачей.

Приложение, запустившее поток, должно быть подготовлено к перехвату и обработке этого исключения, но тогда возникает еще один вопрос. Если задача прерывается путем выдачи исключения, то фактически она возвращается к состоянию `Faulted`. Это касается даже такого исключения, как `OperationCanceledException`. Задача входит в состояние `Canceled`, только если она отменяется без выдачи исключения. Так как же задаче выдать исключение `OperationCanceledException`, чтобы оно не рассматривалось в качестве исключения?

На этот раз ответ кроется в самой задаче. Чтобы задача в управляемом режиме признала, что исключение `OperationCanceledException` стало результатом отмены, а не просто исключением, вызванным другими обстоятельствами, ей следует знать, что операция была на самом деле отменена. Сделать это можно, только если ей удастся исследовать признак отмены. Вы передаете этот признак в качестве параметра тому методу, который запускается задачей, но задача фактически не проверяет какой-либо из этих параметров. Вместо этого вы указываете признак отмены при создании и запуске задачи. Код, показанный в следующем примере, основан на приложении `GraphDemo`. Обратите внимание

не только на передачу признака отмены методу `generateGraphData` (как и прежде), но и на передачу его в качестве параметра методу `Run`:

```
tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4, token),
token);
```

Теперь, как только метод, запущенный задачей, выдаст исключение `OperationCanceledException`, инфраструктура, стоящая за задачей, изучает значение переменной типа `CancellationToken`. Если это значение свидетельствует об отмене задачи, инфраструктура устанавливает состояние задачи в `Canceled`. Если для ожидания завершения задач используется оператор `await`, вам также понадобится быть готовыми к перехвату и обработке исключения `OperationCanceledException`. Именно этим вы и займитесь в следующем упражнении.

Распознавание отмены и обработка исключения `OperationCanceledException`

Вернитесь в среде Visual Studio в окно редактора, показывающее содержимое файла `MainPage.xaml.cs`. В методе `plotButton_Click` внесите изменения в инструкции, показанные далее жирным шрифтом, создающие и запускающие задачи и указывающие `CancellationToken`-объект в качестве второго параметра для метода `Run` (а также в качестве параметра для метода `generateGraphData`):

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;

    ...
    Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4,
    token), token);
    Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4,
    pixelWidth / 2, token), token);
    ...
}
```

Заключите инструкции, создающие и запускающие задачи, ожидающие их выполнения и выводящие затраченное время, в блок `try`. Добавьте блок `catch`, обрабатывающий исключение `OperationCanceledException`. В этом обработчике исключений выведите в элементе управления типа `TextBlock` по

имени `duration` причину выдачи исключения, отчет о котором находится в свойстве `Message` объекта исключения. Все вносимые изменения показаны далее жирным шрифтом:

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    try
    {
        await first;
        await second;

        duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";
    }
    catch (OperationCanceledException oce)
    {
        duration.Text = oce.Message;
    }

    string message = $"Status of tasks is {first.Status}, {second.Status}";
    ...
}
```

Закомментируйте в методе `generateGraphData` инструкцию `if`, которая проверяет значение свойства `IsCancellationRequested` объекта типа `CancellationToken`, и добавьте инструкцию, выделенную далее жирным шрифтом, которая вызывает метод `ThrowIfCancellationRequested`:

```
private void generateGraphData(byte[] data, int partitionStart,
                               int partitionEnd, CancellationToken token)
{
    ...
    for (int x = partitionStart; x < partitionEnd; x++)
    {
        ...
        for (double i = -p; i < p; i += 3)
        {
            //if (token.IsCancellationRequested)
            //{
            //    return;
            //}
            token.ThrowIfCancellationRequested();
            ...
        }
    }
    ...
}
```

Укажите в меню Отладка на пункт Окна и щелкните после этого на пункте Параметры исключений. Снимите в окне параметров исключений флагок Common

Language Runtime Exceptions. Щелкните правой кнопкой мыши на пункте Common Language Runtime Exceptions и убедитесь в том, что режим Continue When Unhandled in User Code (Продолжить после необработанной ошибки в коде пользователя) включен (рис. 23.11).

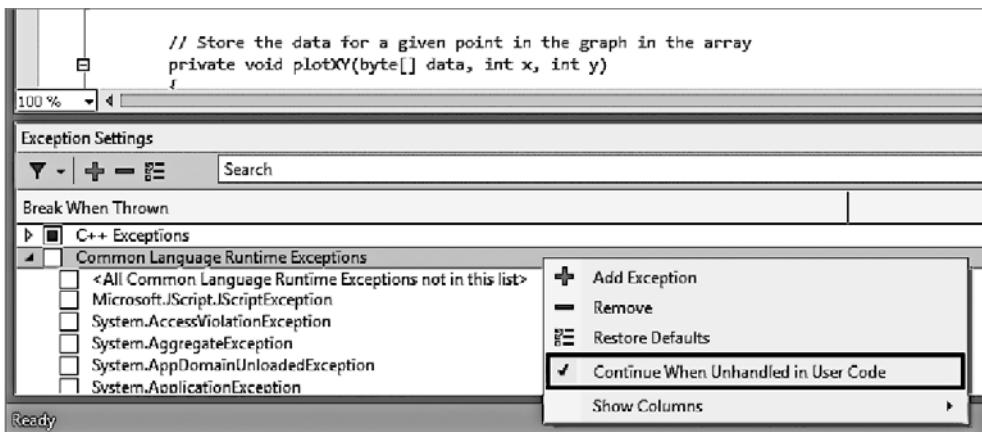


Рис. 23.11

Такая настройка необходима для того, чтобы отладчик среды Visual Studio не перехватывал исключение `OperationCanceledException`, которое будет выдано при запуске приложения в режиме отладки.

В меню Отладка щелкните на пункте Начать отладку.

В окне Graph Demo щелкните на кнопке Plot Graph, дождитесь появления графического изображения и убедитесь в том, что состояние обеих задач отображено как `RanToCompletion` и изображение создано.

Щелкните еще раз на кнопке Plot Graph и тут же щелкните на кнопке Cancel.

Если второй щелчок был сделан без промедления, состояние обеих задач будет отображено как `Canceled`, в элементе управления типа `TextBox` под названием `duration` должен быть выведен текст «The operation was canceled», а графическое изображение должно быть выведено на экран с пустотами. Если вы все же промедлили, попробуйте выполнить последнее действие еще раз (рис. 23.12).

Вернитесь в среду Visual Studio и остановите отладку.

Укажите в меню Отладка на пункт Окна и щелкните после этого на пункте Параметры исключений. В панели инструментов окна Параметры исключений щелкните на кнопке Восстановить для списка параметры по умолчанию, после чего щелкните на кнопке OK.

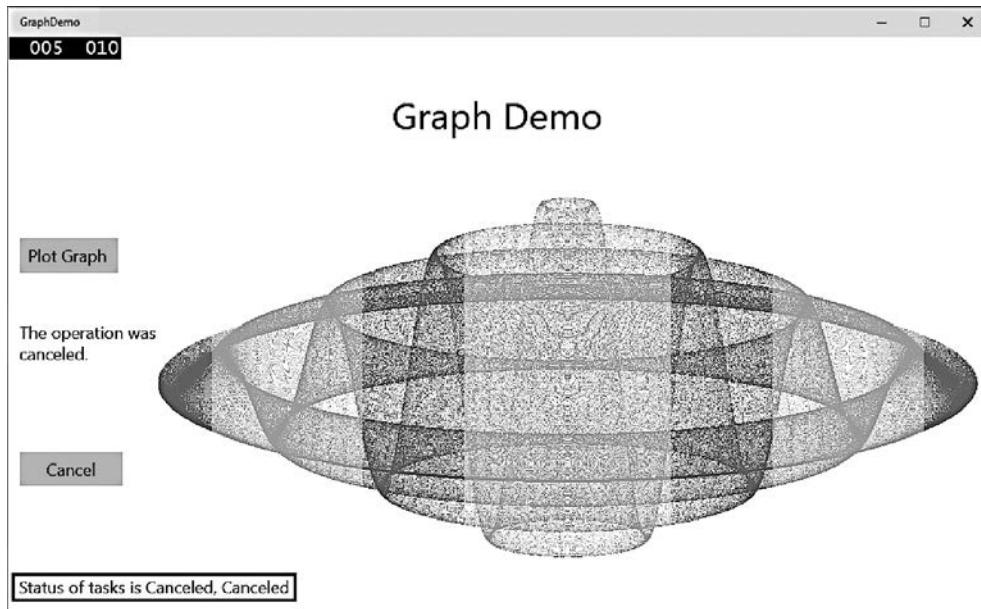


Рис. 23.12

ОБРАБОТКА ИСКЛЮЧЕНИЙ ЗАДАЧ ПУТЕМ ИСПОЛЬЗОВАНИЯ КЛАССА AGGREGATEEXCEPTION

На основе изученного в книге материала уже стало ясно, что обработка исключений является весьма важным элементом любого коммерческого приложения. Те конструкции обработки исключений, которые вам уже попадались, весьма просты в использовании, и при их аккуратном применении перехватить исключения и определить ту часть кода, из которой они были выданы, не составит труда. Но когда вы начинаете разбивать работу на несколько одновременно выполняемых задач, отслеживание и обработка исключений усложняются. В предыдущем упражнении было показано, как можно перехватить исключение OperationCanceledException, выдаваемое при отмене задачи. Но существует множество других исключений, которые также могут быть выданы, и различные задачи могут также выдавать собственные исключения. Поэтому вам нужен способ для перехвата и обработки нескольких исключений, которые могут быть выданы одновременно.

Когда для ожидания завершения нескольких задач используется один из методов ожидания класса Task (экземпляр метода Wait или статические методы Task.WaitAll и Task.WaitAny), любые исключения, выдаваемые методами, запускаемыми этими задачами, собираются вместе в одно исключение, известное нам как исключение AggregateException. Это исключение ведет себя как оболочка для коллекции исключений. Каждое из исключений в коллекции может быть выдано различными задачами. В своем приложении вы можете перехватить

исключение AggregateException, а затем выполнить перебор элементов этой коллекции и любую необходимую обработку. Вам в помощь класс AggregateException предоставляет метод Handle. Этот метод получает делегата Func<Exception, bool>, ссылающегося на метод, получающий в качестве параметра объект типа Exception и возвращающий булево значение. При вызове метода Handle для каждого исключения в коллекции в объекте типа AggregateException запускается метод, на который имеется ссылка. Этот запускаемый по ссылке метод может исследовать исключение и предпринять соответствующее действие. Если этот метод обрабатывает исключение, он должен вернуть значение true. Если нет, он должен вернуть false. Когда метод Handle завершает свою работу, любые необработанные исключения собираются вместе в новое выдаваемое исключение AggregateException. Затем это исключение может перехватить и обработать следующий внешний обработчик исключений.

В следующем фрагменте кода показан пример метода, который может быть зарегистрирован в качестве обработчика исключения AggregateException. Этот метод при обнаружении исключения DivideByZeroException просто выводит на экран сообщение о попытке деления на нуль «Division by zero occurred» или при выдаче исключения IndexOutOfRangeException выводит на экран сообщение о выходе за пределы индексации массива «Array index out of bounds». Любые другие исключения остаются необработанными.

```
private bool handleException(Exception e)
{
    if (e is DivideByZeroException)
    {
        displayErrorMessage("Division by zero occurred");
        return true;
    }

    if (e is IndexOutOfRangeException)
    {
        displayErrorMessage("Array index out of bounds");
        return true;
    }
    return false;
}
```

Когда используются методы ожидания класса Task, можно перехватить исключение AggregateException и зарегистрировать метод handleException:

```
try
{
    Task first = Task.Run(...);
    Task second = Task.Run(...);
    Task.WaitAll(first, second);
}
catch (AggregateException ae)
{
    ae.Handle(handleException);
}
```

Если любая из задач выдает исключение DivideByZeroException или исключение IndexOutOfRangeException, метод handleException выведет на экран соответствующее сообщение и подтвердит, что исключение обработано. Любые другие исключения классифицируются как необработанные и распространяются из обработчика исключения AggregateException обычным способом.

Но есть одно осложнение, о котором нужно знать. При отмене задачи вы видели, что среда CLR выдает исключение OperationCanceledException, и отчет об этом исключении выводится в том случае, если для ожидания завершения задачи используется оператор await. Но если вы используете один из методов ожидания класса Task, это исключение трансформируется в исключение TaskCanceledException, и это тот самый тип исключения, к обработке которого нужно подготовиться в обработчике исключения AggregateException.

Использование продолжений с отмененными и давшими сбой задачами

Если вам нужно выполнить дополнительную работу, когда задача была отменена или выдала необработанное исключение, запомните, что для этого можно воспользоваться методом ContinueWith с соответствующим TaskContinuationOptions-значением. Например, следующий код создает задачу, которая выполняет метод doWork. Если задача отменена, метод ContinueWith указывает на то, что должна быть создана одна задача, запускающая метод doCancellationWork. Этот метод может выполнить ряд простых регистрационных записей или навести порядок в использовании ресурсов. Если задача не отменена, продолжение не запускается.

```
Task task = new Task(doWork);
task.ContinueWith(doCancellationWork, TaskContinuationOptions.OnlyOnCanceled);
task.Start();
...
private void doWork()
{
    // Задача выполняет этот код после запуска
    ...
}
...
private void doCancellationWork(Task task)
{
    // Задача выполняет этот код, когда работа метода doWork завершается
    ...
}
```

Кроме того, определив значение TaskContinuationOptions.OnlyOnFaulted, можно указать продолжение, которое будет запущено при условии выдачи необработанного исключения исходным методом, запускаемым задачей.

Выводы

В этой главе вы узнали о том, насколько важно создавать приложения, способные распространять свое выполнение на несколько процессоров и процессорных ядер. Вы увидели, как класс `Task` используется для выполнения операций в параллельном режиме и как синхронизируются одновременно выполняемые операции и задается ожидание их завершения. Вы научились использовать класс `Parallel` для распараллеливания кода в некоторых наиболее распространенных конструкциях программирования, а также увидели, когда распараллеливать код не имеет смысла. Вы использовали вместе задачи и потоки в графическом пользовательском интерфейсе с целью повышения отзывчивости приложения и скорости обработки данных и увидели, как задачи отменяются вполне понятным и контролируемым образом.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 24.

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню Файл щелкните на пункте Выход. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке Да и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Создать задачу и запустить ее на выполнение	<p>Чтобы одним действием создать и запустить задачу, воспользуйтесь статическим методом <code>Run</code> класса <code>Task</code>:</p> <pre>Task task = Task.Run(() => doWork()); ... private void doWork() { // Задача выполняет этот код после запуска ... }</pre> <p>Или создайте новый <code>Task</code>-объект, ссылающийся на выполняемый метод, и вызовите метод <code>Start</code>:</p> <pre>Task task = new Task(doWork); task.Start();</pre>
Дождаться окончания выполнения задачи	<p>Вызовите метод <code>Wait</code> объекта типа <code>Task</code>:</p> <pre>Task task = ...; ... task.Wait();</pre> <p>Или воспользуйтесь оператором <code>await</code> (только в <code>async</code>-методе):</p> <pre>await task;</pre>

Чтобы	Сделайте следующее
Дождаться окончания выполнения нескольких задач	<p>Вызовите статический метод WaitAll класса Task и укажите задачи, окончания выполнения которых нужно дождаться:</p> <pre>Task task1 = ...; Task task2 = ...; Task task3 = ...; Task task4 = ...; ... Task.WaitAll(task1, task2, task3, task4);</pre>
Указать метод для запуска в новой задаче, когда задача завершит свое выполнение	<p>Вызовите в отношении задачи метод ContinueWith и укажите метод, запускаемый в качестве продолжения:</p> <pre>Task task = new Task(doWork); task.ContinueWith(doMoreWork, TaskContinuationOptions.NotOnFaulted);</pre>
Выполнить итерации цикла и последовательности инструкций путем использования параллельно запускаемых задач	<p>Для выполнения итераций цикла с использованием задач воспользуйтесь методами Parallel.For и Parallel.ForEach:</p> <pre>Parallel.For(0, 100, performLoopProcessing); ... private void performLoopProcessing(int x) { // Выполнение циклической обработки }</pre> <p>Для выполнения одновременного вызова методов путем использования отдельных задач воспользуйтесь методом Parallel.Invoke:</p> <pre>Parallel.Invoke(doWork, doMoreWork, doYetMoreWork);</pre>
Обработать исключения, выдаваемые одной или несколькими задачами	<p>Перехватите исключение AggregateException. Воспользуйтесь методом Handle для указания метода, способного обработать каждое исключение в объекте типа AggregateException. Если метод обработки исключения производит такую обработку, возвращайте значение true, а если нет, возвращайте значение false:</p> <pre>try { Task task = Task.Run(...); task.Wait(); ... } catch (AggregateException ae) { ae.Handle(handleException); }</pre>

Чтобы	Сделайте следующее
	<pre> ... private bool handleException(Exception e) { if (e is TaskCanceledException) { ... return true; } else { return false; } } </pre>
Разрешить отмену задачи	<p>Реализуйте согласованную отмену путем создания CancellationTokenSource-объекта и использования CancellationToken-параметра в методе, запускаемом задачей. Вызовите в этом методе метод ThrowIfCancellationRequested, принадлежащий параметру типа CancellationToken, для выдачи исключения OperationCanceledException и прекращения выполнения задачи:</p> <pre> private void generateGraphData(..., CancellationToken token) { ... token.ThrowIfCancellationRequested(); ... } </pre>

24

Сокращение времени отклика путем выполнения асинхронных операций

Прочитав эту главу, вы научитесь:

- определять асинхронные методы и пользоваться ими для сокращения времени отклика приложений, работающих в интерактивном режиме и выполняющих операции, занимающие много времени или связанные с вводом-выводом данных;
- объяснять, как сократить время, затрачиваемое на выполнение сложных LINQ-запросов, используя распараллеливание;
- пользоваться параллельными классами коллекций для безопасного совместного использования данных параллельно выполняемыми задачами.

В главе 23 «Повышение производительности путем использования задач» демонстрировались способы использования класса Task для выполнения операций в параллельном режиме и повышения производительности работы приложений, связанных с интенсивными вычислениями. Но наряду с тем, что максимальное использование приложением доступной вычислительной мощности может заставить его работать намного быстрее, важным аспектом является также возможность этого приложения реагировать на действия пользователя. Следует помнить, что пользовательский интерфейс Windows работает с использованием одного потока выполнения, но пользователи ожидают от приложения возможной ответной реакции при щелчке на кнопке формы, даже если в данный момент приложение выполняет весьма объемное и сложное вычисление. Кроме того, некоторые задачи могут затрачивать много времени на выполнение, даже если они не связаны с интенсивными вычислениями (это, к примеру, задачи, связанные с вводом-выводом данных, ожидающие получения информации по сети от

удаленного веб-сайта), а блокирование взаимодействия с пользователем на время ожидания события, которое может занять неопределенное количество времени до того, как это событие произойдет, несомненно, не лучший вариант практики проектирования. У обеих этих проблем одно и то же решение: задачи следует выполнять в асинхронном режиме, оставляя свободным поток пользовательского интерфейса для обработки взаимодействия с пользователем.

Вопросы, связанные со временем отклика, не ограничиваются пользовательскими интерфейсами. К примеру, в главе 21 «Запрос данных, находящихся в памяти, с помощью выражений в виде запросов» было показано, как получить доступ к данным, хранящимся в памяти, заявительным способом, используя интегрированный в C# язык запросов (Language-Integrated Query (LINQ)). Обычный LINQ-запрос создает перечисляемый набор результатов, и для извлечения данных можно организовать последовательный перебор элементов набора. Если источник данных, используемый для создания набора результатов, слишком объемный, выполнение LINQ-запроса может занять много времени. Многие системы управления базами данных сталкиваются с проблемой оптимизации запросов, решая ее путем использования алгоритмов, разбивающих процесс идентификации данных для запроса на серию задач с последующим запуском этих задач в параллельном режиме и объединением результатов, и получения полного набора результатов, когда выполнение задач завершится. Разработчики среды Microsoft .NET Framework решили снабдить расширение LINQ подобной способностью, в результате чего появилось расширение Parallel LINQ, или PLINQ. Это расширение будет рассмотрено во второй части данной главы.

Реализация асинхронных методов

Асинхронным называется метод, не блокирующий текущий поток, в котором началось его выполнение. Когда приложение вызывает асинхронный метод, по негласному соглашению ожидается, что метод возвратит управление вызывающей среде довольно быстро и станет выполнять свою работу в отдельном потоке. Определение достаточности нельзя охарактеризовать в количественных математических показателях, но ожидания заключаются в том, что если асинхронный метод выполняет операцию, которая может привести к заметной для вызывающей стороны задержке, он должен выполнять эту операцию с использованием потока, работающего в фоновом режиме, позволяя тем самым вызывающей стороне продолжить работу на текущем потоке. Судя по описанию, это довольно сложный процесс, и в самом деле в ранних версиях среды .NET Framework так оно и было. Но сейчас C# предоставляет модификатор методов под названием `async` и оператор `await`, которые возлагают основные сложности этого процесса на компилятор, а это означает (в большинстве случаев), что вам самим теперь уже не нужно связываться с тонкостями многопоточности.

АСИНХРОННОСТЬ И МАСШТАБИРУЕМОСТЬ

Асинхронность является весьма эффективной концепцией, которую следует уяснить при создании крупномасштабных решений, таких как корпоративные веб-приложения и сервисы. Обычно у веб-сервера имеется ограниченный набор ресурсов для обработки запросов от потенциально широкой аудитории, каждый представитель которой ожидает, что его запрос будет обработан с высокой скоростью. Во многих случаях пользовательский запрос может вызывать серию операций, каждая из которых в отдельности может занять весьма значительное время — возможно, порядка 1–2 с. Рассмотрим, к примеру, систему электронной торговли, в которой пользователь запрашивает каталог товаров или размещает заказ. Обе эти операции обычно требуют чтения и записи данных, хранящихся в базе данных, которая может управляться сервером базы данных, удаленным от веб-сервера. Многие веб-серверы могут поддерживать только ограниченное количество одновременных подключений, и если поток, связанный с подключением, находится в режиме ожидания завершения операции ввода-вывода, это подключение фактически блокируется. Если поток создает для асинхронного ввода-вывода отдельную задачу, он может быть освобожден и подключение будет снова использовано уже другим пользователем. Этот подход предоставляет значительно более широкие возможности для масштабирования, чем тот, при котором операции проводятся в синхронном режиме.

Примеры и подробные объяснения, почему в данной ситуации выполнение синхронного ввода-вывода нельзя считать приемлемым, можно найти в материалах об антишаблоне синхронного ввода-вывода (*Synchronous I/O anti-pattern*) в открытом хранилище Microsoft Patterns & Practices по адресу <https://github.com/mspnp/performance-optimization/tree/master/SynchronousIO>.

Определение асинхронных методов — суть проблемы

Вы уже видели, как можно реализовать одновременно выполняемые операции с использованием `Task`-объектов. Краткое напоминание: когда задача инициируется путем использования методов `Start` или `Run`, относящихся к типу `Task`, среда выполнения (*common language runtime (CLR)*) использует для выделения задаче потока свой собственный алгоритм диспетчеризации и назначает выполнение этого потока на время, удобное для операционной системы при доступности достаточных ресурсов. Этот подход освобождает код от требований по определению рабочей нагрузки вашего компьютера и управлению ею. Если по завершении конкретной задачи нужно выполнить еще одну операцию, можно воспользоваться следующими вариантами.

- Самостоятельно задать ожидание завершения, воспользовавшись одним из `Wait`-методов, предоставляемых `Task`-типом. Затем можно инициировать новую операцию, возможно, путем определения еще одной задачи.
- Определить продолжение, которое просто указывает на операцию, выполняемую по завершении заданной операции. Среда .NET Framework

автоматически выполняет операцию продолжения в качестве задачи, планируемой к выполнению после завершения исходной задачи. Продолжение использует тот же поток, что и исходная задача.

Тем не менее даже несмотря на то что `Task`-тип предоставляет удобное обобщение операций, вам все еще зачастую приходится создавать потенциально нескладный код для решения ряда часто встречающихся проблем, с которыми сталкиваются разработчики при использовании фоновых потоков. Предположим, к примеру, что вы определили следующий метод, выполняющий ряд длительных операций, которые требуют последовательного запуска, после чего выводит на экран сообщение в элементе управления типа `TextBox`:

```
private void slowMethod()
{
    doFirstLongRunningOperation();
    doSecondLongRunningOperation();
    doThirdLongRunningOperation();
    message.Text = "Processing Completed";
}

private void doFirstLongRunningOperation()
{
    ...
}

private void doSecondLongRunningOperation()
{
    ...
}

private void doThirdLongRunningOperation()
{
    ...
}
```

Если вызвать `slowMethod` из какой-то части кода пользовательского интерфейса (например, из обработчика события `Click` для элемента управления типа кнопки), пользовательский интерфейс перестанет реагировать на действия пользователя вплоть до завершения выполнения метода. Можно повысить его отзывчивость при выполнении метода `slowMethod` за счет использования `Task`-объекта, запускающего метод `doFirstLongRunningOperation` и определяющего продолжение для той же задачи, в которой поочередно запускаются методы `doSecondLongRunningOperation` и `doThirdLongRunningOperation`:

```
private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
```

```
task.Start();
message.Text = "Processing Completed"; // Когда появится это сообщение?
}

private void doFirstLongRunningOperation()
{
    ...
}

private void doSecondLongRunningOperation(Task t)
{
    ...
}

private void doThirdLongRunningOperation(Task t)
{
    ...
}
```

Хотя такая реорганизация кода представляется довольно простой, следует отметить ряд особенностей. А именно, чтобы приспособить методы `doSecondLongRunningOperation` и `doThirdLongRunningOperation` к требованиям продолжений, нужно изменить их сигнатуры (методам продолжений передается в качестве параметра `Task`-объект, инициирующий продолжение). Еще важнее задать вопрос: «Когда сообщение будет выведено в элементе управления типа `TextBox`?». Что касается второго пункта, то дело обстоит следующим образом: несмотря на то что метод `Start` инициирует запуск задачи `Task`, он не дожидается ее завершения, следовательно, сообщение появляется не по завершении задачи, а в ходе ее выполнения.

Особой сложностью этот пример не отличается, но важен сам принцип. Есть как минимум два решения. Первое заключается в том, чтобы перед выводом сообщения дождаться завершения задачи:

```
private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.Start();
    task.Wait();
    message.Text = "Processing Completed";
}
```

Но теперь вызов метода `Wait` блокирует поток, выполняющий метод `slowMethod`, и дискредитирует главную цель использования класса `Task`.



ВНИМАНИЕ Вообще-то непосредственного вызова метода `Wait` в потоке пользователяского интерфейса нужно избегать.

Более удачное решение предусматривает определение продолжения, которое выводит сообщение на экран и подстраивается под запуск только при завершении выполнения метода `doThirdLongRunningOperation`, тогда вызов метода `Wait` можно будет удалить. Может возникнуть соблазн реализовать это продолжение в виде делегата, показанного далее жирным шрифтом (не забудьте, что продолжение передает `Task`-объект в виде аргумента, именно для этого и предназначен параметр делегата `t`):

```
private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.ContinueWith((t) => message.Text = "Processing Complete");
    task.Start();
}
```

К сожалению, этот подход вскрывает существование еще одной проблемы. Если попытаться запустить этот код в режиме отладки, окажется, что последнее продолжение выдаст исключение `System.Exception` с весьма туманным сообщением о том, что приложение вызвало интерфейс, маршиализованный для другого потока: «The application called an interface that was marshaled for a different thread». Дело в том, что манипулировать элементами пользовательского интерфейса может только поток пользовательского интерфейса, а здесь вы попытались сделать запись в элемент управления типа `TextBox` из другого потока, того, что был задействован для запуска задачи `Task`-объекта. Эту проблему можно решить за счет использования `Dispatcher`-объекта. Этот объект является компонентом инфраструктуры пользовательского интерфейса, и ему путем вызова его же метода `RunAsync` можно отправлять запросы для выполнения работы в потоке пользовательского интерфейса. Этот метод получает `Action`-делегата, указывающего на запускаемый код. Рассмотрение подробностей, касающихся `Dispatcher`-объекта и метода `RunAsync`, не входит в круг вопросов, рассматриваемых в данной книге, но в следующем примере показано, как ими можно воспользоваться для вывода из продолжения сообщения, требующегося методу `slowMethod`:

```
private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.ContinueWith((t) => this.Dispatcher.RunAsync(
        CoreDispatcherPriority.Normal,
        () => message.Text = "Processing Complete"));
    task.Start();
}
```

Хотя этот код работает, но в нем будет трудно разобраться и его нелегко будет сопровождать. Теперь у вас есть делегат (продолжение), указывающий на другого делегата (код, запускаемый методом `RunAsync`).



ПРИМЕЧАНИЕ Дополнительные сведения об объекте типа `Dispatcher` и о методе `RunAsync` можно найти на веб-сайте компании Microsoft по адресу <https://msdn.microsoft.com/library/windows.ui.core.coredispatcher.runasync>.

Определение асинхронных методов: Решение

Ключевые слова `async` и `await` предназначены в C# для того, чтобы позволить вам определять и вызывать методы, способные выполняться в асинхронном режиме. Это означает, что вы не обязаны заниматься указанием продолжений или диспетчеризацией кода для запуска `Dispatcher`-объектов с целью обеспечения манипуляции данными в надлежащих потоках. Все очень просто:

- ❑ модификатор `async` показывает, что метод содержит функции, которые должны выполняться в асинхронном режиме;
- ❑ оператор `await` указывает места, в которых функции должны выполняться в асинхронном режиме.

В следующем примере кода показан метод `slowMethod`, реализованный с помощью модификатора `async` и операторов `await` в виде асинхронного метода:

```
private async void slowMethod()
{
    await doFirstLongRunningOperation();
    await doSecondLongRunningOperation();
    await doThirdLongRunningOperation();
    message.Text = "Processing Complete";
}
```

Теперь этот метод выглядит в высшей степени похожим на исходную версию, чем и характеризуется эффективность `async` и `await`. Эта магия является ничем иным, как упражнением по переделке вашего кода компилятором C#. Когда в методе, объявленном с ключевым словом `async`, этому компилятору встречается оператор `await`, происходит фактическое переформатирование операнда, который следует за этим оператором, в задачу, запускаемую в том же потоке, что и `async`-метод. Весь остальной код превращается в продолжение, запускаемое после завершения задачи, которое снова запускается в том же потоке. Теперь, поскольку поток, в котором был запущен `async`-метод, был потоком, в котором запущен пользовательский интерфейс, у него есть непосредственный доступ к элементам управления окна, следовательно, он может обновлять их напрямую, не прокладывая маршрут через `Dispatcher`-объект.

Хотя на первый взгляд этот подход выглядит весьма простым, нужно обязательно иметь в виду следующие особенности и избегать возможного недопонимания.

- ❑ Модификатор `async` не является признаком того, что метод запускается в асинхронном режиме в отдельном потоке. Его роль ограничивается указанием на то, что код в методе может быть разделен для получения одного или нескольких продолжений. Когда запускаются эти продолжения, они выполняются в том же самом потоке, что и исходный метод.
- ❑ Оператор `await` указывает место, с которого компилятор C# может разбить код на продолжение. Сам оператор `await` ожидает в качестве своего операнда объект, допускающий ожидание. Этот объект относится к типу, предоставляющему метод `GetAwaiter`, который возвращает объект, который в свою очередь предоставляет методы для запуска кода и ожидания завершения его выполнения. Компилятор C# превращает ваш код в инструкции, использующие эти методы для создания соответствующих продолжений.



ВНИМАНИЕ Оператор `await` можно использовать только в методе с пометкой `async`. Вне `async`-метода ключевое слово рассматривается как обычный идентификатор (можно даже создать переменную по имени `await`, хотя делать это не рекомендуется).

В текущей реализации оператора `await` объект, допускающий ожидание, ждет, что в качестве операнда вы укажете объект типа `Task`. Это означает, что нужно внести ряд изменений в методы `doFirstLongRunningOperation`, `doSecondLongRunningOperation` и `doThirdLongRunningOperation`. Если конкретнее, то каждый метод для выполнения своей работы должен теперь создать и запустить `Task`-объект и вернуть ссылку на этот `Task`-объект. Следующий пример показывает исправленную версию метода `doFirstLongRunningOperation`:

```
private Task doFirstLongRunningOperation()
{
    Task t = Task.Run(() => { /* сюда помещается код для этого метода */ });
    return t;
}
```

Также имеет смысл разобраться с тем, есть ли возможность разбить работу, выполняемую методом `doFirstLongRunningOperation`, на ряд параллельных операций. Если такая возможность есть, работу можно разбить на набор задач (`Task`-объектов) в соответствии с описаниями, которые были даны в главе 23. Но какие из этих `Task`-объектов нужно вернуть в качестве результата метода?

```
private Task doFirstLongRunningOperation()
{
    Task first = Task.Run(() => { /* код для первой операции */ });
    Task second = Task.Run(() => { /* код для второй операции */ });
    return ...; // Какой из объектов возвращать, first или second?
}
```

Если метод возвращает `first`, оператор `await` в `slowMethod` будет дожидаться только завершения этой задачи, но не задачи `second`. Похожая логика применяется, если метод возвращает `second`. Решение заключается в определении метода `doFirstLongRunningOperation` с указанием модификатора `async` и применением оператора `await` в отношении каждой из задач:

```
private async Task doFirstLongRunningOperation()
{
    Task first = Task.Run(() => { /* код для первой операции */ });
    Task second = Task.Run(() => { /* код для второй операции */ });
    await first;
    await second;
}
```

Следует напомнить, что компилятор, встретив оператор `await`, создает код, ожидающий завершения выполнения той задачи, которая была указана в качестве аргумента, вместе с продолжением, запускающим последующие инструкции. Значение, возвращаемое `async`-методом, можно рассматривать в качестве ссылки на задачу, запускающую это продолжение (это не совсем точное описание, но для целей данной главы оно является вполне подходящей моделью). Итак, метод `doFirstLongRunningOperation` создает и запускает в режиме параллельного выполнения задачи `first` и `second`, компилятор переформатирует операторы `await` в код, ожидающий завершения задачи `first`, за которой следует продолжение, ожидающее завершения задачи `second`, а модификатор `async` заставляет компилятор возвращать ссылку на продолжение. Учтите, что из-за того что компилятор не определяет возвращаемое значение метода, вы больше не указываете возвращаемое значение самостоятельно (фактически, если вы попытаетесь в данном случае вернуть значение, код не пройдет компиляцию).



ПРИМЕЧАНИЕ Если не включить инструкцию `await` в `async`-метод, этот метод просто станет ссылкой на `Task`-объект, выполняющий код в теле метода. В результате при вызове метода он не будет запущен в асинхронном режиме. В таком случае компилятор предупредит вас сообщением, что в `async`-методе отсутствуют операторы `await` и он будет выполнен в синхронном режиме: «*This async method lacks await operators and will run synchronously*».



СОВЕТ Модификатор `async` можно использовать в качестве префикса делегата, позволяя создавать делегатов, объединяющих асинхронную обработку путем использования оператора `await`.

В следующем упражнении вы будете работать с приложением `GraphDemo` из главы 23 и внесете в него изменения для создания данных, используемых при построении графического изображения путем использования асинхронного метода.

Внесение изменения в приложение GraphDemo для использования асинхронного метода

Откройте в среде Microsoft Visual Studio 2015 решение GraphDemo, которое находится в папке \Microsoft Press\VCBS\Chapter 24\GraphDemo вашей папки документов.

Раскройте в обозревателе решений узел MainPage.xaml и откройте к окне редактора файл MainPage.xaml.cs.

Найдите в классе MainPage метод plotButton_Click. Его код должен выглядеть следующим образом:

```
private void plotButton_Click(object sender, RoutedEventArgs e)
{
    Random rand = new Random();
    redValue = (byte)rand.Next(0xFF);
    greenValue = (byte)rand.Next(0xFF);
    blueValue = (byte)rand.Next(0xFF);

    tokenSource = new CancellationSource();
    CancellationToken token = tokenSource.Token;

    Stopwatch watch = Stopwatch.StartNew();

    try
    {
        generateGraphData(data, 0, pixelWidth / 2, token);
        duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";
    }
    catch (OperationCanceledException oce)
    {
        duration.Text = oce.Message;
    }
    Stream pixelStream = graphBitmap.PixelBuffer.AsStream();
    pixelStream.Seek(0, SeekOrigin.Begin);
    pixelStream.Write(data, 0, data.Length);
    graphBitmap.Invalidate();
    graphImage.Source = graphBitmap;
}
```

Это упрощенная версия приложения из предыдущей главы. Здесь метод generateGraphData вызывается непосредственно из потока пользовательского интерфейса, а объекты типа Task для создания данных, используемых при построении графического изображения, в параллельном режиме не используются.



ПРИМЕЧАНИЕ Если в упражнении в главе 23 вы в целях экономии памяти уменьшили значения полей pixelWidth и pixelHeight, то, прежде чем продолжить выполнение данного упражнения, сделайте это еще раз.

В меню Отладка щелкните на пункте Начать отладку. Щелкните в окне GraphDemo на кнопке Plot Graph. Попробуйте в ходе создания данных щелкнуть на кнопке Cancel.

Заметьте, что пользовательский интерфейс при создании и выводе графического изображения на экран ни на что не реагирует. Дело в том, что метод `plotButton_Click` всю свою работу, включая создание данных, используемых при построении графического изображения, выполняет в синхронном режиме.

Вернитесь в среду Visual Studio и остановите отладку.

В окне редактора, показывающем код класса `MainPage`, добавьте выше метода `generateGraphData` новый закрытый метод по имени `generateGraphDataAsync`.

Этот метод будет получать такой же список параметров, что и метод `generateGraphData`, но вместо `void` он должен вернуть `Task`-объект. Также у метода должна быть пометка `async`, и он должен выглядеть следующим образом:

```
private async Task generateGraphDataAsync(byte[] data,
    int partitionStart, int partitionEnd,
    CancellationToken token)
{
}
```



ПРИМЕЧАНИЕ Асинхронные методы принято называть с применением суффикса `Async`.

Добавьте к методу `generateGraphDataAsync` инструкции, выделенные здесь жирным шрифтом:

```
private async Task generateGraphDataAsync(byte[] data, int partitionStart, int
partitionEnd, CancellationToken token)
{
    Task task = Task.Run(() => generateGraphData(data, partitionStart,
        partitionEnd, token));
    await task;
}
```

Этот код создает `Task`-объект, который запускает метод `generateGraphData` и использует оператор `await` для ожидания завершения задачи `Task`-объекта. Задача, созданная компилятором в результате использования оператора `await`, является значением, возвращенным методом.

Вернитесь к методу `plotButton_Click` и измените определение этого метода, включив в него, как показано далее жирным шрифтом, модификатор `async`:

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
}
```

Вставьте в блок `try` метода `plotButton_Click` инструкцию, создающую данные для графического изображения и показанную далее жирным шрифтом, чтобы вызвать метод `generateGraphDataAsync` в асинхронном режиме:

```

try
{
    await generateGraphDataAsync(data, 0, pixelWidth / 2, token);
    duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds})";
}
...

```

В меню Отладка укажите на пункт Окна и после этого щелкните на пункте Параметры исключений. Снимите в окне параметров исключений флажок Common Language Runtime Exceptions. Щелкните правой кнопкой мыши на пункте Common Language Runtime Exceptions и убедитесь в том, что режим Продолжить после необработанной ошибки в коде пользователя включен.

В меню Отладка щелкните на пункте Начать отладку. В окне GraphDemo щелкните на кнопке Plot Graph и убедитесь в том, что приложение успешно создает графическое изображение.

Щелкните на кнопке Plot Graph, а затем в ходе создания данных — на кнопке Cancel. На этот раз пользовательский интерфейс прореагирует. Будет создана только часть графического изображения, а в элементе управления типа `TextBlock` по имени `duration` будет выведено сообщение об отмене операции: «The operation was cancelled» (рис. 24.1).

Вернитесь в среду Visual Studio и остановите отладку.

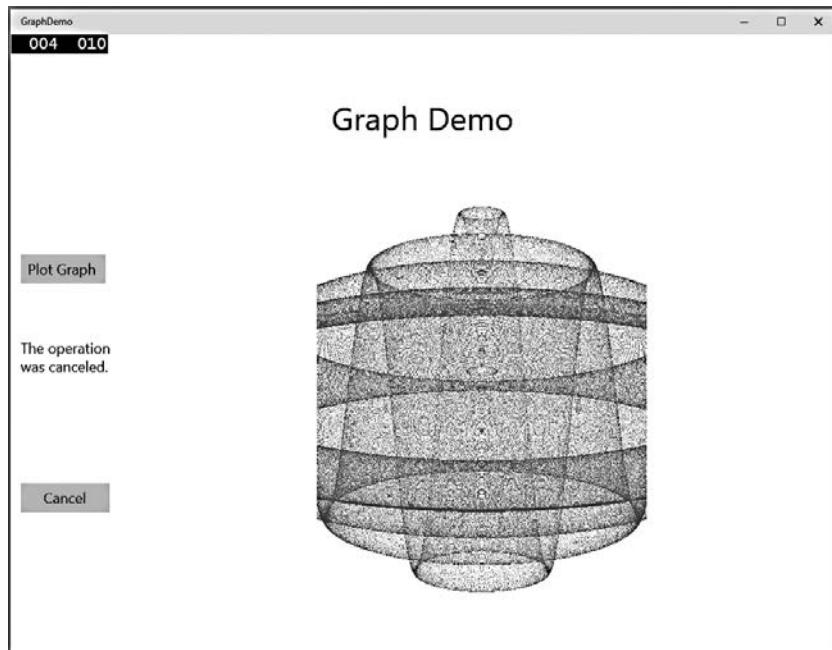


Рис. 24.1

Определение асинхронных методов, возвращающих значения

До сих пор во всех показанных упражнениях для выполнения части работы использовался Task-объект, не возвращающий значение. Но вы также используете задачи для запуска методов, вычисляющих результат. Для этого применяется класс-обобщение `Task<TResult>`, где параметр типа, `TResult`, указывает тип результата.

Объект типа `Task<TResult>` создается и запускается почти так же, как и обычный Task-объект. Основное отличие заключается в том, что выполняемый код должен вернуть значение. Например, метод по имени `calculateValue`, показанный в следующем примере кода, создает целочисленный результат. Для вызова этого метода с помощью задачи создается и запускается объект типа `Task<int>`. Значение, возвращаемое методом, вы получаете путем запроса свойства `Result`, принадлежащего `Task<int>`-объекту. Если задача не завершила выполнение метода и результат еще не доступен, свойство `Result` блокирует вызывающий код. Это означает, что вам не нужно самостоятельно что-либо предпринимать в плане синхронизации и что вы знаете: как только свойство `Result` вернет значение, задача завершит свою работу:

```
Task<int> calculateValueTask = Task.Run(() => calculateValue(...));  
...  
int calculatedData = calculateValueTask.Result; // Блокируется до завершения работы  
// calculateValueTask  
...  
private int calculateValue(...)  
{  
    int someValue;  
    // Выполнение вычисления и присваивание значения переменной someValue  
    ...  
    return someValue;  
}
```

Тип-обобщение `Task<TResult>` является также основой механизма определения асинхронных методов, возвращающих значения. В предыдущих примерах было показано, что асинхронные `void`-методы реализуются путем возвращения Task-объекта. Если асинхронный метод действительно создает результат, он, как показано в следующем примере, создающем асинхронную версию метода `calculateValue`, должен вернуть объект типа `Task<TResult>`:

```
private async Task<int> calculateValueAsync(...)  
{  
    // Вызов calculateValue с использованием Task  
    Task<int> generateResultTask = Task.Run(() => calculateValue(...));  
    await generateResultTask;  
    return generateResultTask.Result;  
}
```

Этот метод выглядит немного странно, поскольку возвращаемый тип указан как `Task<int>`, а инструкция `return` на самом деле возвращает значение типа `int`. Следует напомнить, что при определении `async`-метода компилятор реорганизует ваш код и на самом деле возвращает ссылку на `Task`-объект, который запускает продолжение для инструкции, возвращающей `generateResultTask.Result`. Это продолжение возвращает выражение типа `int`, поэтому возвращаемым типом метода является `Task<int>`.

Для вызова асинхронного метода, возвращающего значение, нужно воспользоваться оператором `await`:

```
int result = await calculateValueAsync(...);
```

Этот оператор извлекает значение из `Task`-объекта путем использования метода `calculateValueAsync` и в данном случае присваивает его переменной `result`.

Трудности, связанные с асинхронными методами

Программисты считают, что модификатор `async` и оператор `await` вносят в код путаницу. Поэтому важно усвоить следующее.

- ❑ Если метод объявлен с модификатором `async`, это еще не означает, что он выполняется в асинхронном режиме. Это означает, что метод может содержать инструкции, которые могут выполняться в асинхронном режиме.
- ❑ Оператор `await` показывает, что метод должен быть запущен в отдельной задаче и что вызывающий код приостанавливается, пока не будет завершен вызов метода. Поток, используемый вызывающим кодом, высвобождается и может быть использован повторно. Это важно в том случае, если это тот самый поток, который используется пользовательским интерфейсом, поскольку это позволяет сохранить его отзывчивость на действия пользователя.
- ❑ Оператор `await` не является функциональным аналогом принадлежащего задаче метода `Wait`, который всегда блокирует текущий поток и не допускает его повторного использования, пока задача не завершится.
- ❑ Изначально код, возобновляющий выполнение после оператора `await`, пытается получить исходный поток, который был использован для вызова асинхронного метода. Если этот поток занят, код будет блокирован. Чтобы указать, что выполнение кода может быть возобновлено в любом доступном потоке, и сократить шансы на его блокировку, можно воспользоваться методом `ConfigureAwait(false)`. Особую пользу это принесет веб-приложениям и сервисам, которым может понадобиться обслуживать многие тысячи одновременно поступающих запросов.

- Метод `ConfigureAwait(false)` нельзя использовать, если код, запускаемый после оператора `await`, должен выполняться в исходном потоке. В ранее рассмотренном примере добавление `ConfigureAwait(false)` к каждой операции с `await` приведет к тому, что с высокой вероятностью продолжения, создаваемые компилятором, будут запускаться в отдельных потоках. Это касается и продолжений, пытающихся установить для свойства `Text` строковое значение с сообщением, что снова приведет к выдаче исключения, свидетельствующего о том, что приложение вызвало интерфейс, маршированный для другого потока: «The application called an interface that was marshaled for a different thread».

```
private async void slowMethod()
{
    await doFirstLongRunningOperation().ConfigureAwait(false);
    await doSecondLongRunningOperation().ConfigureAwait(false);
    await doThirdLongRunningOperation().ConfigureAwait(false);
    message.Text = "Processing Complete";
}
```

- Неосмотрительное использование асинхронных методов, возвращающих результаты и запускаемых в потоке пользовательского интерфейса, может привести к возникновению взаимных блокировок и стать причиной зависания приложения. Рассмотрим следующий пример:

```
private async void myMethod()
{
    var data = generateResult();
    ...
    message.Text = $"result: {data.Result}";
}

private async Task<string> generateResult()
{
    string result;
    ...
    result = ...
    return result;
}
```

Метод `generateResult` в этом примере кода возвращает строковое значение. Но метод `myMethod` фактически не запускает задачу, выполняющую метод `generateResult`, пытаясь получить доступ к свойству `data.Result`. Переменная `data` является ссылкой на задачу, и если свойство `Result` недоступно по причине того, что задача не была запущена, то обращение к этому свойству заблокирует текущий поток до завершения работы метода `generateResult`. Кроме того, когда метод завершит работу, задача, используемая для запуска метода `generateResult`, попытается возобновить работу потока, на котором она была вызвана (потока пользовательского интерфейса), но теперь этот поток

окажется заблокированным. В результате всего этого метод `myMethod` не сможет завершить свою работу, пока не завершится работа метода `generateResult`, а метод `generateResult` не сможет завершиться до тех пор, пока не завершится метод `myMethod`.

Решением этой проблемы является использование оператора `await` в отношении задачи, которая выполняет метод `generateResult`. Сделать это можно следующим образом:

```
private async void myMethod()
{
    var data = generateResult();
    ...
    message.Text = $"result: {await data}";
}
```

Асинхронные методы и API-интерфейсы Windows Runtime

Разработчики Windows 8 и последующих версий этой операционной системы хотели обеспечить максимально возможную отзывчивость приложений, поэтому при реализации WinRT ими было принято решение, что любая операция, которая может занять более 50 мс, должна быть доступна только через асинхронный API-интерфейс. В этой книге уже встречалась пара примеров такого подхода. К примеру, для вывода на экран сообщения для пользователя можно воспользоваться объектом типа `MessageDialog`. Но при выводе этого сообщения нужно воспользоваться методом `ShowAsync`:

```
using Windows.UI.Popups;
...
MessageDialog dlg = new MessageDialog("Message to user");
await dlg.ShowAsync();
```

Объект типа `MessageDialog` выводит сообщение на экран и ждет, пока пользователь не щелкнет на кнопке `Close` (Закрыть), появляющейся в качестве части этого диалогового окна. Любая форма взаимодействия с пользователем может занять неопределенное время (прежде чем щелкнуть на кнопке `Close`, пользователь может пойти на обед), и зачастую бывает важно не заблокировать приложение или не препятствовать ему при выполнении других операций (например, при реагировании на события), пока на экране находится диалоговое окно. Класс `MessageDialog` не предоставляет синхронную версию метода `ShowAsync`, но если нужно вывести диалоговое окно в синхронном режиме, можно воспользоваться методом `dlg.ShowAsync()` без оператора `await`.

Другой часто встречающийся пример асинхронной обработки касается класса `FileOpenPicker`, который вам уже встречался в главе 5 «Использование

инструкций составного присваивания и итераций». Класс `FileOpenPicker` выводит на экран список файлов, в котором пользователь может сделать свой выбор. Как и в случае использования класса `MessageDialog`, пользователь может потратить на просмотр и выбор файла довольно много времени, поэтому данная операция не должна блокировать работу приложения. В следующем примере показано, как использовать класс `FileOpenPicker` для отображения файлов, находящихся в папке документов пользователя, и ожидать выбора пользователем отдельного файла из списка:

```
using Windows.Storage;
using Windows.Storage.Pickers;
...
FileOpenPicker fp = new FileOpenPicker();
fp.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
fp.ViewMode = PickerViewMode.List;
fp.FileTypeFilter.Add("*");
StorageFile file = await fp.PickSingleFileAsync();
```

Ключевая инструкция находится в строке, где вызывается метод `PickSingleFileAsync`. Это метод, отображающий список файлов и позволяющий пользователю перемещаться по файловой системе и выбирать файл. (Класс `FileOpenPicker` также предоставляет метод `PickMultipleFilesAsync`, с помощью которого пользователь может выбрать сразу несколько файлов.) Значение, возвращаемое этим методом, имеет тип `Task<StorageFile>`, а оператор `await` извлекает из этого результата объект типа `StorageFile`. Класс `StorageFile` предоставляет абстрактное представление файла, хранящегося на жестком диске. Используя этот класс, можно открыть файл и производить с ним операции чтения и записи данных.



ПРИМЕЧАНИЕ Собственно говоря, метод `PickSingleFileAsync` возвращает объект типа `IAsyncOperation<StorageFile>`. WinRT использует свое собственное абстрактное представление асинхронных операций и отображает Task-объекты .NET Framework на эту абстракцию; интерфейс `IAsyncOperation` реализуется в классе `Task`. Если вы программируете на C#, ваш код не затрагивается этим преобразованием и вы можете просто использовать Task-объекты, не задумываясь о том, как они получат отображение на асинхронные операции WinRT.

Еще одним источником потенциально медленно выполняемых операций является файловый ввод-вывод, и в классе `StorageFile` реализуется ряд асинхронных методов, с помощью которых эти операции могут выполняться, не влияя на отзывчивость приложения. К примеру, в главе 5, после того как пользователь выбирал файл, используя `FileOpenPicker`-объект, код открывал этот файл для чтения в асинхронном режиме:

```
StorageFile file = await fp.PickSingleFileAsync();
...
var fileStream = await file.OpenAsync(FileAccessMode.Read);
```

И еще один, последний пример, имеющий непосредственное отношение к упражнениям, которые вы уже видели в этой и предыдущей главах, касается записи в поток данных. Вы могли заметить, что хотя время в отчете о создании данных для графического изображения составляло порядка нескольких секунд, могло пройти вдвое больше времени, прежде чем появится само изображение. Дело в способе записи данных в растровое изображение. Это изображение отображает данные, хранящиеся в буфере в качестве части объекта типа `WriteableBitmap`, а метод расширения `AsStream` предоставляет этому буферу интерфейс типа `Stream` (поток данных). Данные записываются в буфер через этот поток с помощью метода `Write`:

```
...
Stream pixelStream = graphBitmap.PixelBufferAsStream();
pixelStream.Seek(0, SeekOrigin.Begin);
pixelStream.Write(data, 0, data.Length);
...
```

Если вы не уменьшили значения полей `pixelWidth` и `pixelHeight` для экономии памяти, объем данных, записываемых в буфер, будет составлять чуть более 570 Мбайт ($15\ 000 \cdot 10\ 000 \cdot 4$ байта), поэтому на эту операцию `Write` уйдет несколько секунд. Для сокращения времени отклика эту операцию можно выполнить в асинхронном режиме, воспользовавшись методом `WriteAsync`:

```
await pixelStream.WriteAsync(data, 0, data.Length);
```

Таким образом, создавая приложение под операционную систему Windows, вы должны изыскивать возможность применения асинхронного режима везде, где только можно.

ШАБЛОН ПРОЕКТИРОВАНИЯ IASYNCRESULT В РАННИХ ВЕРСИЯХ .NET FRAMEWORK

Асинхронность уже давно признана ключевым элементом в создании отзывчивых приложений с помощью .NET Framework и понятием, предшествующим введению класса `Task` в .NET Framework версии 4.0. Чтобы справиться с ситуациями, требующими асинхронности, компания Microsoft ввела шаблон проектирования `IAsyncResult` на основе делегата типа `AsyncCallback`. Рассмотрение подробностей работы этого шаблона не входит в круг вопросов, рассматриваемых в данной книге, но с точки зрения программистов реализация шаблона означает, что многие типы в библиотеке классов среды .NET Framework предоставляют длительные операции двумя способами: в синхронном виде, состоящем из одного метода, и в асинхронном виде, использующем пару методов с именами в формате *НачалоИмяОперации* (`BeginOperationName`) и *КонецИмяОперации* (`EndOperationName`), где *ИмяОперации* указывает на выполняемую операцию. Например, класс `MemoryStream` в пространстве имен `System.IO` предоставляет метод `Write` для записи данных в синхронном режиме в поток данных в памяти, но он также предоставляет методы `BeginWrite` и `EndWrite` для выполнения той же операции в асинхронном режиме.

Метод `BeginWrite` иницирует операцию записи, выполняемую в новом потоке, и ждет от программиста предоставления ссылки на метод обратного вызова, который выполняется по завершении операции записи, — эта ссылка дается в виде `AsyncCallback`-делегата. В этом методе программист должен реализовать любое подходящее наведение порядка в использовании ресурсов и, чтобы обозначить завершение операции, вызвать метод `EndWrite`. Применение этого шаблона показано в следующем примере:

```
...
Byte[] buffer = ...; // Заполнение данными для записи в MemoryStream
MemoryStream ms = new MemoryStream();
AsyncCallback callback = new AsyncCallback(handleWriteCompleted);
ms.BeginWrite(buffer, 0, buffer.Length, callback, ms);
...
private void handleWriteCompleted(IAsyncResult ar)
{
    MemoryStream ms = ar.AsyncState as MemoryStream;
    ... // Выполнение любых подходящих действий по наведению порядка
    ms.EndWrite(ar);
}
```

Параметром метода обратного вызова (`handleWriteCompleted`) является `IAsyncResult`-объект, содержащий информацию о состоянии асинхронной операции и любую другую информацию о состоянии. В этом параметре вы можете передать методу обратного вызова информацию, определенную пользователем, — в этот параметр запаковывается последний аргумент, предоставляемый методу `BeginOperationName`. В данном примере методу обратного вызова передается ссылка на `MemoryStream`.

При всей работоспособности этой последовательности сам подход представляется нечетко выраженным, не обозначающим выполняемые операции достаточно ясно. Код для операции разбит на два метода, предполагаемую связь между которыми при сопровождении кода легко упустить из виду. При использовании `Task`-объектов эту модель можно упростить путем вызова статического метода `FromAsync`, принадлежащего классу `TaskFactory`. Этот метод получает метод `BeginOperationName` и метод `EndOperationName` и заключает их в код, выполняемый с использованием задачи, определяемой `Task`-объектом. При этом отпадает необходимость использования `AsyncCallback`-делегата, поскольку он создается за сценой с помощью метода `FromAsync`. Следовательно, операцию, показанную в предыдущем примере, можно выполнить следующим образом:

```
...
Byte[] buffer = ...;
MemoryStream s = new MemoryStream();
Task t = Task<int>.Factory.FromAsync(s.Beginwrite, s.EndWrite, buffer, 0,
                                         buffer.Length, null);
t.Start();
await t;
...
```

Эта технология пригодится в том случае, если нужно получить доступ к возможностям асинхронного выполнения, предоставляемым типами, разработанными в ранних версиях среди .NET Framework.

Использование PLINQ для распараллеливания декларативного доступа к данным

Еще одной областью, для которой время отклика играет важную роль, является доступ к данным. Особенно это касается создания приложений, которым приходится вести поиск в протяженных структурах данных. В предыдущих главах вы смогли убедиться в эффективности применения расширения LINQ для извлечения данных из перечисляемых структур, но показанные примеры были по своей природе однопоточными. PLINQ предоставляет LINQ-набор расширений, основанных на применении задач в виде `Task`-объектов, помогающих существенно ускорить выполнение и распараллелить некоторые операции запросов.

PLINQ работает за счет разбиения набора данных на части и последующего применения задач для извлечения данных, соответствующих критериям, указанным в запросе параллельно для каждой части. Когда задачи завершат свое выполнение, результаты, извлеченные для каждой части, объединяются в один перечисляемый набор результатов. PLINQ идеально подходит для сценариев, где фигурируют наборы данных с большим количеством элементов, или для тех случаев, когда критерии, указанные для подходящих данных, предполагают проведение сложных операций, требующих больших вычислительных мощностей. Важной целью расширения PLINQ является его минимально возможное вторжение в код. Для преобразования LINQ-запроса в PLINQ-запрос используется метод расширения `AsParallel`. Этот метод возвращает `ParallelQuery`-объект, действующий аналогично исходному перечисляемому объекту, за исключением того, что он предоставляет возможность параллельной реализации множества LINQ-операторов, таких как `join` и `where`. Эти реализации LINQ-операторов основаны на задачах и используют различные алгоритмы в попытках запуска частей вашего LINQ-запроса в параллельном режиме везде, где только возможно. Но как и все в мире параллельных вычислений, метод `AsParallel` не волшебный. Вы не можете гарантировать, что выполнение кода ускорится, — все зависит от природы ваших LINQ-запросов и от того, поддаются ли распараллеливанию выполняемые ими задачи.

Чтобы разобраться в работе PLINQ и понять, в каких ситуациях это расширение принесет пользу, лучше обратиться к примерам. В упражнениях следующих разделов показаны два простых сценария.

Использование PLINQ для повышения производительности при переборе элементов коллекции

Первый сценарий довольно прост. Давайте рассмотрим LINQ-запрос, перебирающий элементы коллекции и извлекающий их из нее на основе вычисления,

интенсивно нагружающего процессор. Эта форма запроса может получить преимущество от параллельного выполнения при условии, что вычисления независимы друг от друга. Элементы в коллекции могут быть разбиты на несколько частей, точное число которых зависит от текущей нагрузки на компьютер и количества доступных центральных процессоров. Элементы в каждой части могут обрабатываться в отдельном потоке. После того как будут обработаны все части, результаты можно объединить. Таким образом можно управлять любой коллекцией, поддерживающей доступ к элементам посредством индекса, например массивом или коллекцией, реализующей интерфейс `IList<T>`.

Распараллеливание LINQ-запроса в отношении простой коллекции

Откройте в среде Visual Studio 2015 решение PLINQ, которое находится в папке `\Microsoft Press\VCSBS\Chapter 24\PLINQ` вашей папки документов.

Дважды щелкните в обозревателе решений на файле `Program.cs` проекта PLINQ, чтобы его код появился в окне редактора. Приложение, с которым ведется работа, консольное. Его основная структура уже создана. Класс `Program` содержит методы с именами `Test1` и `Test2`, созданными, чтобы проиллюстрировать два наиболее распространенных сценария. Метод `Main` вызывает каждый из этих тестовых методов по очереди. Оба тестовых метода имеют одинаковую основную структуру: они создают LINQ-запрос (добавлять код к ним вы будете в наборе упражнений чуть позже), запускают его на выполнение и выводят на экран затраченное на это время. Код для каждого из этих методов практически полностью отделен от инструкций, создающих и запускающих запросы.

Давайте изучим метод `Test1`. Этот метод создает большой массив, состоящий из целых чисел, и заполняет его набором случайных чисел в диапазоне от 0 до 200. Для генератора случайных чисел применяется начальное число, поэтому при запуске приложения вы должны получать одни и те же результаты.

Добавьте к этому методу сразу же после первого комментария `ToDo` LINQ-запрос, показанный здесь жирным шрифтом:

```
// TO DO: Создать LINQ-запрос, извлекающий все числа больше 100
var over100 = from n in numbers
              where TestIfTrue(n > 100)
              select n;
```

Этот LINQ-запрос извлекает все элементы в массиве чисел со значением больше 100. Сам по себе тест $n > 100$ не может считаться содержащим интенсивное вычисление, достаточное для того, чтобы показать преимущества распараллеливания этого запроса, поэтому код вызывает метод по имени `TestIfTrue`, который создает небольшое замедление, выполняя операцию `SpinWait`. Метод `SpinWait`

заставляет процессор за короткое время многократно выполнять цикл специальных инструкций «пустая команда», нагружая процессор, но не выполняя никакой работы (это действие называется пробуксовкой). Метод `TestIfTrue` выглядит следующим образом:

```
public static bool TestIfTrue(bool expr)
{
    Thread.Sleep(1000);
    return expr;
}
```

Добавьте в метод `Test1` после второго комментария `TO DO` код, показанный здесь жирным шрифтом:

```
// TO DO: Run the LINQ query, and save the results in a List<int> object
List<int> numbersOver100 = new List<int>(over100);
```

Следует напомнить, что LINQ-запросы используют отложенное выполнение, поэтому они не запускаются на выполнение, пока вы не станете извлекать из них результаты. Эта инструкция создает объект типа `List<int>` и заполняет его результатами выполнения запроса `over100`.

Добавьте в метод `Test1` после третьего комментария `TO DO` инструкцию, показанную далее жирным шрифтом:

```
// TO DO: Display the results
Console.WriteLine($"There are {numbersOver100.Count} numbers over 100");
```

Щелкните в меню Отладка на пункте Запуск без отладки. Запишите время выполнения `Test1` и количество элементов массива, значение которых превышает 100.

Запустите приложение несколько раз и вычислите среднее время. Убедитесь в том, что количество элементов со значением, превышающим 100, одинаково при каждом запуске (чтобы обеспечить повторяемость тестов, приложение использует при запуске одни и те же случайные числа). Когда закончите, вернитесь в среду Visual Studio.

Логика, выбирающая каждый элемент, возвращенный LINQ-запросом, независима от выборочной логики для всех остальных элементов, поэтому запрос является идеальным кандидатом на разделение. Внесите изменения, показанные здесь жирным шрифтом, в инструкцию, определяющую LINQ-запрос, и укажите метод расширения `AsParallel` для массива `numbers`:

```
var over100 = from n in numbers.AsParallel()
              where TestIfTrue(n > 100)
              select n;
```



ПРИМЕЧАНИЕ Если логика выбора или вычисления требует совместного использования данных, вы должны синхронизировать задачи, выполняемые в параллельном режиме, в противном случае результат может быть непредсказуем. Но синхронизация может привести к издержкам и свести на нет все преимущества от распараллеливания запроса.

В меню Отладка щелкните на пункте Запуск без отладки. Убедитесь в том, что количество элементов, попадающее в отчет `Test1`, такое же, как и прежде, а время, затрачиваемое на выполнение, существенно снизилось. Запустите тест несколько раз и выведите среднее значение продолжительности тестирования.

Если тест выполняется на двухъядерном процессоре (или на компьютере с двумя процессорами), вы должны заметить, что время сократилось на 40–45 %. Если у вас больше процессорных ядер, сокращение будет еще большим (на моей четырехъядерной машине время обработки уменьшилось с 10,3 до 2,8 с).

Закройте приложение и вернитесь в среду Visual Studio.

Предыдущее упражнение показало повышение производительности, которую можно достичь путем внесения небольших изменений в LINQ-запрос. Но следует иметь в виду, что подобные результаты можно увидеть только при условии, что вычисление, выполняемое запросом, занимает довольно длительное время. Я немного схитрил за счет пробуксовки процессора. Без этой издержки параллельная версия запроса будет выполняться медленнее последовательной. В следующем упражнении вы увидите LINQ-запрос, объединяющий два массива в памяти. На этот раз в упражнении используются более реалистичные объемы данных, поэтому в искусственном замедлении запроса нет необходимости.

Распараллеливание LINQ-запроса, объединяющего две коллекции

В обозревателе решений щелкните на файле `Data.cs`, откройте его в окне редактора и найдите класс `CustomersInMemory`.

В этом классе содержится открытый строковый массив по имени `Customers`. Каждая строка в массиве `Customers` хранит данные об отдельном клиенте, а поля отделены друг от друга запятыми. Такой формат обычно применяется для данных, которые приложение может считать из текстового файла, использующего поля, разделенные запятыми. В первом поле содержится идентификатор клиента, во втором — название компании, которую он представляет, а в остальных полях — адрес, город, страна или почтовый код.

Найдите класс `OrdersInMemory`. Этот класс похож на класс `CustomersInMemory`, за исключением того, что в нем содержится строковый массив `Orders`. В первом поле каждой строки содержится номер заказа, во втором — идентификатор клиента, в третьем — данные о размещении заказа.

Найдите класс `OrderInfo`. Он содержит четыре поля: идентификатор клиента, название компании, идентификатор заказа и дату заказа. Для заполнения коллекции объектов типа `OrderInfo` из данных, находящихся в массивах `Customers` и `Orders`, будет использоваться LINQ-запрос.

Выполните в окне редактора файл `Program.cs` и найдите в классе `Program` метод `Test2`. В этом методе будет создан LINQ-запрос, объединяющий массивы `Customers` и `Orders` путем использования идентификатора пользователя для возврата списка клиентов и заказов, размещенных каждым клиентом. Запрос будет сохранять каждую строку результата в объекте типа `OrderInfo`.

Добавьте в `try`-блок этого метода после первого комментария `TO DO` код, показанный жирным шрифтом:

```
// TO DO: Create a LINQ query that retrieves customers and orders from arrays
// Store each row returned in an OrderInfo object
var orderInfoQuery = from c in CustomersInMemory.Customers
    join o in OrdersInMemory.Orders
    on c.Split(',')[0] equals o.Split(',')[1]
    select new OrderInfo
    {
        CustomerID = c.Split(',')[0],
        CompanyName = c.Split(',')[1],
        OrderID = Convert.ToInt32(o.Split(',')[0]),
        OrderDate = Convert.ToDateTime(o.Split(',')[2]),
        new CultureInfo("en-US"))
    };

```

Добавленная инструкция определяет LINQ-запрос. Обратите внимание на то, что для разбиения каждой строки на массив строк в нем используется метод `Split`, принадлежащий классу `String`. Строки разбиваются по запятым, при этом запятые удаляются. Одно из осложнений заключается в том, что даты хранятся в формате United States English, поэтому код, помещая их в `OrderInfo`-объект, преобразует их в объекты типа `DateTime`, задавая формат United States English. Если вы пользуетесь указателем формата по умолчанию для вашей локализации, разбор дат может пройти неправильно. В конечном счете для создания каждой записи этот запрос выполняет существенный объем работы.

Добавьте в метод `Test2` после второго комментария `TO DO` следующий код, выделенный жирным шрифтом:

```
// TO DO: Run the LINQ query, and save the results in a List<OrderInfo> object
List<OrderInfo> orderInfo = new List<OrderInfo>(orderInfoQuery);
```

Эти инструкции запускают запрос и заполняют коллекцию `orderInfo`.

Добавьте после третьего комментария `TO DO` инструкцию, показанную жирным шрифтом:

```
// TO DO: Display the results
Console.WriteLine($"There are {orderInfo.Count} orders");
```

Закомментируйте в методе `Main` инструкцию, вызывающую метод `Test1`, и удалите символы комментария из строки с инструкцией, вызывающей метод `Test2`, как показано далее жирным шрифтом:

```
static void Main(string[] args)
{
    // Test1();
    Test2();
}
```

В меню Отладка щелкните на пункте Запуск без отладки.

Убедитесь в том, что `Test2` извлекает 830 заказов, и зафиксируйте продолжительность выполнения теста. Запустите приложение несколько раз для получения среднего значения, а затем вернитесь в среду Visual Studio.

Внесите в `Test2` изменения, показанные далее жирным шрифтом, скорректировав LINQ-запрос и добавив метод расширения `AsParallel` к массивам `Customers` и `Orders`:

```
var orderInfoQuery = from c in CustomersInMemory.Customers.AsParallel()
                      join o in OrdersInMemory.Orders.AsParallel()
                      on c.Split(',')[0] equals o.Split(',')[1]
                      select new OrderInfo
                      {
                          CustomerID = c.Split(',')[0],
                          CompanyName = c.Split(',')[1],
                          OrderID = Convert.ToInt32(o.Split(',')[0]),
                          OrderDate = Convert.ToDateTime(o.Split(',')[2]),
                          New CultureInfo("en-US"))
                      };
};
```



ВНИМАНИЕ При объединении двух источников данных таким способом оба они должны быть объектами типа `IEnumerable` или `ParallelQuery`. Это означает, что при указании метода `AsParallel` для одного источника вы также должны указать `AsParallel` и для другого. Если этого не сделать, ваш код не запустится — его выполнение будет прервано с выдачей ошибки.

Запустите приложение несколько раз. Обратите внимание на то, что на выполнение `Test2` будет тратиться значительно меньше времени, чем раньше. Чтобы оптимизировать операции объединения путем извлечения данных для каждой объединяемой части в параллельном режиме, PLINQ может воспользоваться несколькими потоками.

Закройте приложение и вернитесь в среду Visual Studio.

Эти два простых упражнения показали вам эффективность применения метода расширения `AsParallel` и расширения PLINQ. Следует заметить, что PLINQ является развивающейся технологией, и весьма вероятно, что со временем ее

внутренняя реализация изменится. Кроме того, объемы данных и вычислительной работы, выполняемой в запросе, также имеют отношение к эффективности использования PLINQ. Поэтому вы не должны рассматривать эти упражнения в качестве определения жестких правил, подлежащих неукоснительному соблюдению. Они просто иллюстрируют положение, согласно которому вы должны очень тщательно подходить к оценке возможного повышения производительности или получения других преимуществ использования PLINQ со своими собственными данными и в собственной вычислительной среде.

Отмена PLINQ-запроса

В отличие от обычных LINQ-запросов, PLINQ-запрос можно отменить. Для этого указывается `CancellationToken`-объект из `CancellationTokenSource` и используется метод расширения `WithCancellation`, принадлежащий классу `ParallelQuery`:

```
CancellationToken tok = ...;  
...  
var orderInfoQuery =  
    from c in CustomersInMemory.Customers.AsParallel().WithCancellation(tok)  
    join o in OrdersInMemory.Orders.AsParallel()  
    on ...
```

В запросе `WithCancellation` указывается однократно. Отмена применяется ко всем источникам запроса. Если объект типа `CancellationTokenSource`, используемый для создания `CancellationToken`, содержит отмену, запрос останавливается с выдачей исключения `OperationCanceledException`.

Синхронизация одновременного доступа к данным

Технология PLINQ не может быть всегда наиболее подходящей технологией для вашего приложения. Если вы создаете задачи самостоятельно, то для правильной работы нужно обеспечить их скоординированность. Библиотека классов .NET Framework предоставляет методы, с помощью которых вы можете ожидать завершения задач, и эти методы можно использовать для координации выполнения задач на весьма примитивном уровне. Но представьте себе, что произойдет, если две задачи пытаются получить доступ к одним и тем же данным и внести в них изменения. Если обе задачи выполняются одновременно, их параллельно осуществляемые операции могут испортить данные. Эта ситуация может привести к возникновению ошибок, исправить которые будет очень трудно главным образом по причине их непредсказуемости.

Класс `Task` предоставляет эффективную структуру, с помощью которой можно проектировать и создавать приложения, пользующиеся преимуществами применения нескольких ядер центрального процессора для выполнения задач в параллельном режиме. Но при сборке решений, использующих одновременно выполняемые операции, нужно проявлять осмотрительность, особенно если эти операции реализуют совместный доступ к данным. Средств управления диспетчеризацией параллельно выполняемых операций или даже степенью распараллеливания, которую может предоставить операционная система приложению, спроектированному путем использования задач, у вас немного. Подобные решения остаются на усмотрение среды выполнения и зависят от рабочей нагрузки и возможностей оборудования компьютера, на котором выполняется приложение. Этот уровень абстракции был преднамеренным решением части команды разработчиков компании Microsoft: при создании приложений, требующих использования параллельных потоков, он избавляет вас от необходимости разбираться в низкоуровневом управлении потоками и в подробностях диспетчеризации. Но за эту абстрагированность приходится платить. Хотя все вроде бы работает волшебным образом само по себе, но вам все же нужно приложить усилия, чтобы разобраться в том, как работает код. В противном случае у вас, как показано в следующем примере кода (этот пример доступен в проекте `ParallelTest` в папке, содержащей код для главы 24), получатся приложения с непредсказуемым (и неправильным) поведением:

```
using System;
using System.Threading;

class Program
{
    private const int NUMELEMENTS = 10;

    static void Main(string[] args)
    {
        SerialTest();
    }

    static void SerialTest()
    {
        int[] data = new int[NUMELEMENTS];
        int j = 0;

        for (int i = 0; i < NUMELEMENTS; i++)
        {
            j = i;
            doAdditionalProcessing();
            data[i] = j;
            doMoreAdditionalProcessing();
        }

        for (int i = 0; i < NUMELEMENTS; i++)
    }
```

```

        Console.WriteLine($"Element {i} has value {data[i]}");
    }
}

static void doAdditionalProcessing()
{
    Thread.Sleep(10);
}

static void doMoreAdditionalProcessing()
{
    Thread.Sleep(10);
}
}

```

Метод `SerialTest` заполняет (довольно многословным способом) целочисленный массив набором значений, а затем выполняет последовательный перебор получившегося списка, выводя на экран индекс каждого элемента массива и значение соответствующего элемента. Методы `doAdditionalProcessing` и `doMoreAdditionalProcessing` просто имитируют выполнение довольно продолжительных операций в качестве части обработки, которая может заставить среду выполнения приступить к управлению распределением рабочей нагрузки процессора. Программа выводит следующие данные:

```

Element 0 has value 0
Element 1 has value 1
Element 2 has value 2
Element 3 has value 3
Element 4 has value 4
Element 5 has value 5
Element 6 has value 6
Element 7 has value 7
Element 8 has value 8
Element 9 has value 9

```

Теперь рассмотрим показанный далее метод `ParallelTest`. Этот метод аналогичен методу `SerialTest`, за исключением того, что для заполнения массива `data` с помощью запуска одновременно выполняемых задач в нем используется конструкция `Parallel.For`. Код в лямбда-выражении, запускаемом каждой задачей, идентичен коду в исходном цикле `for` в методе `SerialTest`:

```

using System.Threading.Tasks;
...

static void ParallelTest()
{
    int[] data = new int[NUMELEMENTS];
    int j = 0;

    Parallel.For (0, NUMELEMENTS, (i) =>
    {

```

```

j = i;
doAdditionalProcessing();
data[i] = j;
doMoreAdditionalProcessing();
});

for (int i = 0; i < NUMELEMENTS; i++)
{
    Console.WriteLine($"Element {i} has value {data[i]}");
}
}
}

```

Предполагается, что метод `ParallelTest` будет выполнять ту же операцию, что и метод `SerialTest`, за исключением того, что за счет использования одновременно выполняемых задач и при удачном стечении обстоятельств делаться это будет немного быстрее. Проблема в том, что порой этот код может работать весьма неожиданным образом. Вот один из примеров вывода, созданного методом `ParallelTest`:

```

Element 0 has value 1
Element 1 has value 1
Element 2 has value 4
Element 3 has value 8
Element 4 has value 4
Element 5 has value 1
Element 6 has value 4
Element 7 has value 8
Element 8 has value 8
Element 9 has value 9

```

Значения, присвоенные каждому элементу массива `data`, не всегда совпадают со значениями, созданными при использовании метода `SerialTest`. Кроме того, при последующих выполнениях метода `ParallelTest` могут получаться разные наборы результатов.

Если исследовать логику конструкции `Parallel.For`, можно заметить то место, где кроется проблема. Лямбда-выражение содержит следующие инструкции:

```

j = i;
doAdditionalProcessing();
data[i] = j;
doMoreAdditionalProcessing();

```

Внешне вроде все нормально. Код копирует текущее значение переменной `i` (индексной переменной, показывающей, какая итерация цикла выполняется) в переменную `j`, а чуть позже он сохраняет значение переменной `j` в элементе массива `data`, индексируемом значением переменной `i`. Если `i` содержит 5, `j` присваивается значение 5, и чуть позже значение `j` сохраняется в элементе `data[5]`. Но между присваиванием значения переменной `j` и считыванием его обратно код проделывает дополнительную работу — вызывает метод `doAdditionalProcessing`.

Если выполнение этого метода занимает много времени, среда выполнения может приостановить поток и спланировать выполнение другой задачи. При этом может запуститься параллельно выполняемая задача, осуществляющая другую итерацию конструкции `Parallel.For`, и присвоить переменной `j` новое значение. Вследствие этого, когда возобновится выполнение исходной задачи, значение `j`, которое она будет присваивать элементу `data[5]`, уже не будет значением, которое было сохранено этим потоком, и в результате произойдет порча данных. Еще более тревожным моментом является то, что иногда этот код может выполняться в точном соответствии с вашими ожиданиями и давать правильные результаты, а иногда он может работать совершенно по-другому — все зависит от степени занятости компьютера и момента диспетчеризации выполнения различных задач. Поэтому ошибки такого рода могут не проявляться во время тестирования, а затем совершенно внезапно обнаруживаться в производственной среде.

Переменная `j` совместно используется всеми одновременно выполняемыми задачами. Если задача сохраняет значение в переменной `j`, а чуть позже считывает его обратно, она должна гарантировать, что никакая другая задача не изменила значение `j` за это время. Для этого требуется синхронизированный доступ к переменной, распространяемый на все одновременно выполняемые задачи, которые могут получить к ней доступ. Одним из способов получения синхронизированного доступа является блокировка данных.

Блокировка данных

Семантика блокировки, которой можно воспользоваться для обеспечения исключительного доступа к ресурсам, предоставляется в языке C# посредством ключевого слова `lock`. Ключевое слово `lock` можно использовать следующим образом:

```
object myLockObject = new object();
...
lock (myLockObject)
{
    // Код, требующий исключительного доступа к общему ресурсу
    ...
}
```

Инструкция `lock` пытается получить взаимоисключающую блокировку указанного объекта (можно вообще-то использовать любой ссылочный тип, а не только объект), и ее выполнение блокируется, если тот же объект уже заблокирован другим потоком. Когда поток становится хозяином блокировки, выполняется код, находящийся в блоке, который следует за инструкцией `lock`. В конце этого блока блокировка снимается. Если другой поток заблокирован в ожидании получения блокировки, он может затем захватить блокировку и продолжить выполняться.

Примитивы синхронизации для координации выполнения задач

Применение ключевого слова `lock` является вполне подходящим средством для многих простых сценариев, но в некоторых ситуациях у вас могут быть более сложные требования. Пространство имен `System.Threading` включает несколько дополнительных примитивов синхронизации, которыми можно воспользоваться в подобных ситуациях. Эти примитивы синхронизации являются классами, разработанными для использования вместе с задачами. Они предоставляют механизмы блокировки, ограничивающие доступ к ресурсу на время удержания блокировки со стороны задачи. Они поддерживают различные технологии блокировки, которыми можно воспользоваться для реализации различных стилей одновременного доступа в диапазоне от простых исключающих блокировок, где одна задача имеет исключительный доступ к ресурсу, до семафоров, где несколько задач могут обращаться к ресурсу одновременно, но управляемым образом, и блокировок по чтению-записи, позволяющих различным задачам иметь совместный доступ по чтению к ресурсу наряду с гарантированным исключительным доступом для потока, которому необходимо внести в ресурс изменение.

Некоторые из этих примитивов приведены в следующем перечне. Дополнительные сведения и примеры можно найти в документации, предоставляемой средой Visual Studio 2015.



ПРИМЕЧАНИЕ Со времен своего первого выпуска среда .NET Framework включает в себя солидный набор примитивов синхронизации. В следующем перечне перечислены только самые поздние примитивы, включенные в пространство имен `System.Threading`. Между новыми и предшествующими примитивами существует некоторое функциональное перекрытие. Там, где оно имеет место, вы должны использовать более поздние альтернативные варианты, поскольку они были разработаны и оптимизированы под компьютеры с несколькими центральными процессорами.

Подробное рассмотрение теории, касающейся всевозможных механизмов синхронизации, доступных для создания многопоточных приложений, в круг вопросов, рассматриваемых в данной книге, не входит. Дополнительные сведения, касающиеся общей теории многопоточности и синхронизации, можно найти в теме «Синхронизация данных для многопоточности» в документации, предоставляемой вместе со средой Visual Studio 2015.

- **ManualResetEventSlim.** Класс `ManualResetEventSlim` предоставляет функциональные средства, с помощью которых ожидать событие может одна или несколько задач.

Объект типа `ManualResetEventSlim` может находиться в одном из двух состояний: сигнальном (`true`) и несигнальном (`false`). Задача создает `ManualResetEventSlim`-объект и указывает его исходное состояние. Другие задачи могут

ожидать, пока `ManualResetEventSlim`-объект не станет сигнальным, вызывая для этого метод `Wait`. Если объект типа `ManualResetEventSlim` находится в несигнальном состоянии, метод `Wait` блокирует задачи. Другая задача может изменить состояние `ManualResetEventSlim`-объекта на сигнальное путем вызова метода `Set`. Это действие освобождает все задачи, ожидающие изменения состояния `ManualResetEventSlim`-объекта, позволяя им возобновить выполнение. Метод `Reset` изменяет состояние `ManualResetEventSlim`-объекта, возвращая его в несигнальное.

- ❑ **SemaphoreSlim.** Класс `SemaphoreSlim` можно использовать для управления доступом к пулу ресурсов.

Объект типа `SemaphoreSlim` имеет исходное значение (неотрицательное целое число) и необязательное максимальное значение. Обычно исходное значение `SemaphoreSlim`-объекта является количеством ресурсов в пуле. Задачи, обращающиеся к ресурсам в пуле, сначала вызывают метод `Wait`. Этот метод пытается уменьшить значение объекта типа `SemaphoreSlim` на единицу, и если результат получается ненулевым, потоку разрешается продолжить выполнение и взять ресурс из пула. Когда задача завершает работу, она должна вызвать в отношении `SemaphoreSlim`-объекта метод `Release`. Это действие повысит значение семафора на единицу.

Если задача вызывает метод `Wait` и в результате уменьшения значения `SemaphoreSlim`-объекта получается отрицательное значение, задача ждет, пока другая задача не вызовет метод `Release`.

Класс `SemaphoreSlim` также предоставляет свойство `CurrentCount`, которым можно воспользоваться для определения того, пройдет ли операция `Wait` успешно или ее выполнение выльется в блокировку.

- ❑ **CountdownEvent.** Класс `CountdownEvent` можно представлять как нечто среднее между обратным семафором и сбросом события вручную.

Когда задача создает `CountdownEvent`-объект, она указывает начальное значение (неотрицательное целое число). Одна или несколько задач могут вызывать метод `Wait` объекта типа `CountdownEvent`, и если его значение отличается от нуля, задачи блокируются. Метод `Wait` не уменьшает значение `CountdownEvent`-объекта на единицу, вместо этого, чтобы уменьшить значение, другие задачи могут вызвать метод `Signal`. Когда значение `CountdownEvent`-объекта достигает нуля, все заблокированные задачи получают сигнал и могут возобновить свое выполнение.

Задача может вернуть значение `CountdownEvent`-объекта к значению, указанному в его конструкторе, воспользовавшись для этого методом `Reset`, а еще задача может увеличить это значение, вызвав метод `AddCount`. Свойство `CurrentCount` позволяет определить, насколько высока вероятность быть заблокированным при вызове метода `Wait`.

- **ReaderWriterLockSlim.** Класс `ReaderWriterLockSlim` является самым современным примитивом синхронизации, поддерживающим одну записывающую задачу и множество читающих задач. Идея заключается в том, что изменение ресурса (запись в него) требует исключительного доступа, а чтение ресурса этого не требует — одновременный доступ к ресурсу могут получить сразу несколько читающих задач, но не в то время, когда его получила записывающая задача.

Задача, желающая прочитать ресурс, вызывает метод `EnterReadLock`, принадлежащий объекту типа `ReaderWriterLockSlim`. Это действие захватывает блокировку объекта по чтению. Когда задача завершает работу с ресурсом, она вызывает метод `ExitReadLock`, который снимает блокировку по чтению. Один и тот же ресурс могут читать сразу несколько задач, и каждая задача получает собственную блокировку по чтению.

Когда задача вносит в ресурс изменения, она может вызвать метод `EnterWriteLock` того же `ReaderWriterLockSlim`-объекта, чтобы получить блокировку по записи. Если у одной или нескольких задач в этот момент имеется блокировка по чтению относительно этого объекта, метод `EnterWriteLock` блокируется до тех пор, пока все эти блокировки не будут освобождены. После того как задача получит блокировку по записи, она сможет внести изменения в ресурс и вызвать метод `ExitWriteLock`, чтобы освободить блокировку.

У объекта типа `ReaderWriterLockSlim` имеется только одна блокировка по записи. Если другая задача попытается получить блокировку по записи, она будет заблокирована до тех пор, пока первая задача не освободит эту блокировку по записи.

Чтобы обеспечить невозможность бесконечной блокировки записывающих задач, как только задача запрашивает блокировку по записи, все последующие вызовы метода `EnterReadLock`, осуществляемые другими задачами, блокируются до тех пор, пока блокировка по записи не будет получена и освобождена.

- **Barrier.** Используя класс `Barrier`, можно временно остановить выполнение набора задач в конкретном месте приложения и возобновить его только тогда, когда все задачи достигнут этого места. Он пригодится для синхронизации задач, которым нужно выполнить последовательно, одна за другой, серию параллельных операций.

Когда задача создает объект типа `Barrier`, она указывает то количество задач в наборе, которое будет синхронизировано. Это значение можно представить себе в виде счетчика задач, обслуживаемого внутренними механизмами класса `Barrier`. Позже это значение может быть скорректировано путем вызова метода добавления участника — `AddParticipant` или метода удаления участника — `RemoveParticipant`. Когда задача достигает места синхронизации, она вызывает метод `SignalAndWait`, принадлежащий `Barrier`-объекту, который

уменьшает значение счетчика потоков внутри `Barrier`-объекта. Если значение этого счетчика больше нуля, задача блокируется. Только когда значение счетчика достигнет нуля, все задачи, ожидающие продолжения выполнения из-за использования объекта типа `Barrier`, освобождаются и могут продолжить свое выполнение.

Класс `Barrier` предоставляет свойство `ParticipantCount`, показывающее количество синхронизируемых задач, и свойство `ParticipantsRemaining`, показывающее, сколько задач нуждаются в вызове метода `SignalAndWait` до поднятия барьера и возможности заблокированным задачам продолжить выполнение.

В конструкторе класса `Barrier` можно также указать делегата. Этот делегат может ссылаться на метод, запускаемый, когда барьера достигнут все задачи. В качестве параметра этому методу передается `Barrier`-объект. Пока этот метод не завершит работу, барьер поднят не будет и задачи не получат свободу.

Отмена синхронизации

Все классы, `ManualResetEventSlim`, `SemaphoreSlim`, `CountdownEvent` и `Barrier`, поддерживают отмену, следуя ее модели, рассмотренной в главе 23. Операции ожидания для каждого из этих классов могут получать необязательный параметр `CancellationToken`, извлекаемый из объекта, имеющего тип `CancellationTokenSource`. Если вызвать метод `Cancel` объекта типа `CancellationTokenSource`, каждая операция ожидания, ссылающаяся на признак отмены `CancellationToken`, созданный этим источником, прерывается с выдачей исключения `OperationCanceledException` (которое, возможно, будет заключено в исключение `AggregateException` в зависимости от контекста операции ожидания).

Как вызывается метод `Wait` объекта типа `SemaphoreSlim` и указывается признак отмены, показано в следующем коде. Если операция ожидания отменяется, выполняется обработчик, перехватывающий исключение `OperationCanceledException`.

```
CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
CancellationToken cancellationToken = cancellationTokenSource.Token;
...
// Семафор, защищающий пул из трех ресурсов
SemaphoreSlim semaphoreSlim = new SemaphoreSlim(3);
...
// Ожидание на семафоре и перехват исключения OperationCanceledException,
// если другой поток вызывает Cancel в отношении cancellationTokenSource
try
{
    semaphoreSlim.Wait(cancellationToken);
}
catch (OperationCanceledException e)
{
    ...
}
```

Классы коллекций, к которым осуществляется одновременный доступ

Ко многим многопоточным приложениям довольно часто предъявляется требование по сохранению данных в коллекции и их извлечению. Стандартные классы коллекций, предоставляемые средой .NET Framework, изначально не считаются безопасными, хотя вы можете воспользоваться примитивами синхронизации, рассмотренными в предыдущем разделе, в которые заключается код, добавляющий элементы в коллекцию, отправляющий в отношении нее запросы и извлекающий из нее элементы. Но этот процесс потенциально предрасположен к ошибке и не обладает широкими возможностями масштабирования, поэтому в библиотеку классов среды .NET Framework включены небольшие наборы классов коллекций и интерфейсов для безопасной работы в многопоточной вычислительной среде, которые находятся в пространстве имен `System.Collections.Concurrent` и разработаны специально для использования с задачами. Краткое общее описание основных типов этого пространства имен приводится в следующем перечне.

- ❑ **ConcurrentBag<T>**. Это класс-обобщение для хранения неупорядоченной коллекции элементов. Он включает методы для вставки (`Add`), удаления (`TryTake`) и исследования (`TryPeek`) элементов коллекции. Эти методы безопасны в многопоточной среде. Коллекция также является перечисляемой, допускающей последовательный перебор содержимого путем использования инструкции `foreach`.
- ❑ **ConcurrentDictionary< TKey, TValue >**. Этот класс реализует безопасную в многопоточной среде версию класса-обобщения коллекции `Dictionary< TKey, TValue >`, рассмотренного в главе 18 «Использование коллекций». Он предоставляет методы `TryAdd`, `ContainsKey`, `TryGetValue`, `TryRemove` и `TryUpdate`, которыми можно воспользоваться для добавления элементов в словарь, их запроса, удаления и изменения.
- ❑ **ConcurrentQueue<T>**. Этот класс предоставляет безопасную в многопоточной среде версию класса-обобщения `Queue< T >`, рассмотренного в главе 18. Он включает методы `Enqueue`, `TryDequeue` и `TryPeek`, которыми можно воспользоваться для добавления элементов в очередь, их удаления и запроса.
- ❑ **ConcurrentStack<T>**. Это безопасная в многопоточной среде реализация класса-обобщения `Stack< T >`, также рассмотренного в главе 18. Она предоставляет методы `Push`, `TryPop` и `TryPeek`, которыми можно воспользоваться для помещения элементов в стек, их извлечения из стека и запроса элементов, находящихся в стеке.



ПРИМЕЧАНИЕ Добавление в классы коллекций безопасности при работе в многопоточной среде требует дополнительных издержек при работе среды выполнения, поэтому эти классы работают не так быстро, как обычные классы коллекций. Вам следует помнить об этом при принятии решения о распараллеливании набора операций, требующих обращения к совместно используемой коллекции.

Использование коллекции с одновременным доступом и блокировка с целью реализации безопасного доступа к данным в многопоточной среде

В следующей подборке упражнений вами будет реализовано приложение, вычисляющее число π путем использования геометрического приближения. Сначала будет выполнено вычисление в однопоточном режиме, а затем в код будут внесены изменения для вычисления с использованием одновременно выполняемых задач. В ходе выполнения упражнений будет выявлен ряд вопросов синхронизации данных, на которые следует обратить внимание и которые будут разрешены путем использования класса коллекции, безопасно работающего в многопоточной вычислительной среде, и блокировкой, обеспечивающей должную координацию действий, выполняемых в задачах.

Реализуемый вами алгоритм вычисляет число π на основе простых математических и статистических выборок. Если нарисовать окружность с радиусом r и квадрат со сторонами, соприкасающимися с окружностью, то длина сторон квадрата составит $2r$ (рис. 24.2).

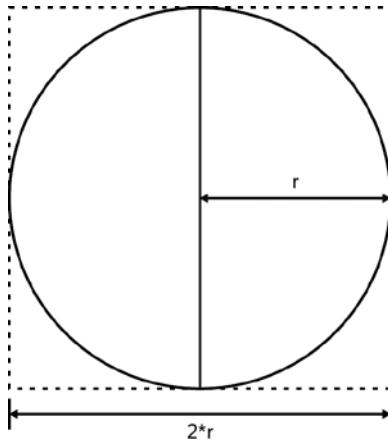


Рис. 24.2

Площадь квадрата S можно вычислить следующим образом:

$$S = (2r)(2r)$$

или

$$S = 4rr.$$

А площадь круга C вычисляется так:

$$C = \pi rr.$$

После перегруппировки формул можно увидеть, что

$$rr = C/\pi$$

и

$$rr = S/4.$$

Объединив этих равенства, получим следующий результат:

$$S/4 = C/\pi.$$

Следовательно:

$$\pi = 4C/S.$$

Весь фокус заключается в том, чтобы определить отношение площади круга C к площади квадрата S . Именно здесь и пригодится статистическая выборка. Можно создать набор случайных точек внутри квадрата и подсчитать, сколько точек попадает и в круг. Если создать довольно большую случайную выборку, отношение точек, попадающих в круг, к точкам, попадающим в квадрат (а также в круг), даст приблизительное соотношение площадей двух фигур C/S . Остается лишь подсчитать эти точки.

Как определить, что точка находится внутри круга? Чтобы сделать решение более наглядным, нарисуйте на листке бумаги квадрат с центром в точке начала координат $(0, 0)$. После этого можно сгенерировать пару значений, или координат, находящихся в диапазоне от $(-r, -r)$ до $(+r, +r)$. Затем можно определить, находится ли любой набор координат (x, y) внутри круга, применив для определения расстояния от начала координат теорему Пифагора. Расстояние d можно вычислить как квадратный корень из $(xx + yy)$. Если d меньше радиуса окружности r или равно ему, значит, как показано на графике на рис. 24.3, координаты (x, y) указывают на точку внутри круга.

Ситуацию можно упростить еще больше, генерируя координаты, которые попадают только в верхний правый квадрант графика, тогда останется лишь генерировать пары случайных чисел между 0 и r . Именно этот подход и будет использоваться в упражнениях.



ПРИМЕЧАНИЕ Упражнения в этой главе предназначены для выполнения на компьютере с многоядерным процессором. Если у вас одноядерный центральный процессор, эффектов, аналогичных описываемым, вы не увидите. Кроме того, между выполнением упражнений не нужно запускать никаких дополнительных программ или служб, поскольку это может повлиять на наблюдаемые результаты.

Вычисление числа π с использованием одного потока

Откройте в среде Visual Studio 2015 решение CalculatePI, которое находится в папке `\Microsoft Press\VCBS\Chapter 24\CalculatePI` вашей папки документов.

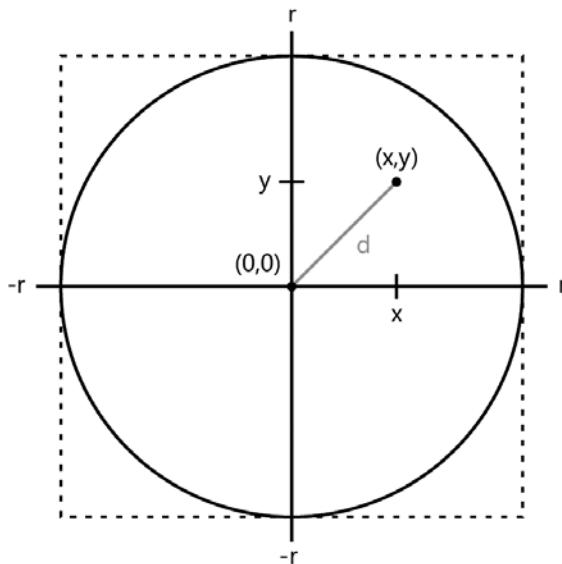


Рис. 24.3

Дважды щелкните на файле Program.cs проекта CalculatePI, показанного в обозревателе решений, чтобы вывести его содержимое в окно редактора. Это консольное приложение. Его основная структура уже создана.

Прокрутите экран до конца файла и изучите метод Main, имеющий следующий вид:

```
static void Main(string[] args)
{
    double pi = SerialPI();
    Console.WriteLine($"Geometric approximation of PI calculated serially: {pi}");

    Console.WriteLine();
    // pi = ParallelPI();
    // Console.WriteLine($"Geometric approximation of PI calculated in
    // parallel: {pi}");
}
```

Этот код вызывает метод `SerialPI`, вычисляющий число π путем использования геометрического алгоритма, рассмотренного перед упражнением. Значение возвращается в виде числа с двойной точностью и выводится на экран. Код, который в данный момент закомментирован, вызывает метод `ParallelPI`, осуществляющий то же самое вычисление, но с использованием одновременно выполняемых задач. Выводимый на экран результат должен быть таким же, как и результат, возвращаемый методом `SerialPI`.

Изучите метод `SerialPI`.

```
static double SerialPI()
{
    List<double> pointsList = new List<double>();
    Random random = new Random(SEED);
    int numPointsInCircle = 0;
    Stopwatch timer = new Stopwatch();
    timer.Start();

    try
    {
        // TO DO: Реализовать геометрическую аппроксимацию  $\pi$ 
        return 0;
    }
    finally
    {
        long milliseconds = timer.ElapsedMilliseconds;
        Console.WriteLine($"SerialPI complete: Duration: {milliseconds} ms");
        Console.WriteLine(
            $"Points in pointsList: {pointsList.Count}. Points within circle:
            {numPointsInCircle}");
    }
}
```

Этот метод генерирует большой общий набор координат и вычисляет расстояние каждого набора координат от начальной точки. Размер общего набора указан константой NUMPOINTS в начале класса `Program`. Чем больше это значение, тем объемнее получается общий набор координат и тем точнее выходит значение π , вычисленное этим методом. Если у вашего компьютера имеется довольно большой объем памяти, вы можете увеличить значение NUMPOINTS. А если приложение при запуске выдает исключение, связанное с недостатком памяти — `OutOfMemoryException`, значение этой константы нужно уменьшить.

Расстояния от начала координат до всех точек хранятся в коллекции `pointsList`, имеющей тип `List<double>`. Данные для координат генерируются путем использования переменной `random`. Она является `Random`-объектом, для создания которого используется постоянное начальное число, чтобы при каждом запуске программы генерировался один и тот же набор случайных чисел. (Это поможет определить, правильно ли выполняется программа.) Если нужно, чтобы генератор случайных чисел имел другое начальное значение, вам следует изменить значение константы `SEED` в начале класса `Program`.

Для подсчета количества точек в коллекции `pointsList`, находящихся в пределах круга, используется переменная `numPointsInCircle`. Радиус окружности указывается в начале класса `Program` с помощью константы `RADIUS`.

Чтобы легче было сравнивать производительность этого метода с производительностью метода `ParallelPI`, в коде создается `Stopwatch`-переменная по имени `timer`, и объект, ссылка на который в ней содержится, запускается на выполнение. В последнем блоке определяется время, затраченное на вычисление,

и результат выводится на экран. По причинам, которые будут названы чуть позже, в последнем блоке на экран выводятся также количество элементов, хранящихся в коллекции `pointsList`, и количество точек, попадающих в круг.

Далее в несколько приемов в `try`-блок будет добавлен код, выполняющий вычисления.

Удалите из `try`-блока комментарий и инструкцию `return`, которая была предоставлена исключительно для обеспечения прохождения кодом компиляции. Добавьте в `try`-блок цикл `for` и инструкции, показанные в следующем примере кода жирным шрифтом:

```
try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    }
}
```

Добавленный блок кода создает пару значений координат в диапазоне от 0 до `RADIUS` и сохраняет ее в переменных `xCoord` и `yCoord`. Затем код использует теорему Пифагора, чтобы вычислить расстояние от начальной точки до этих координат и добавить результат к коллекции `pointsList`.



ПРИМЕЧАНИЕ Объем вычислительной работы в этом блоке кода невелик, а в настоящее приложение для научных расчетов будут, скорее всего, включены более сложные вычисления, загружающие процессор на более продолжительный период времени. Чтобы имитировать подобную ситуацию, в этом блоке кода вызывается еще один метод, `doAdditionalProcessing`. Как показано в следующем примере, он всего лишь отвлекает на себя некоторое количество циклов центрального процессора. Такой подход я выбрал с целью более яркой демонстрации требований, предъявляемых к синхронизации доступа к данным, не заставляя при этом вас для загрузки центрального процессора создавать приложение, выполняющее какое-либо весьма сложное вычисление, например быстрое преобразование Фурье:

```
private static void doAdditionalProcessing()
{
    Thread.SpinWait(SPINWAITS);
}
```

`SPINWAITS` — это еще одна константа, определение которой находится в начале класса `Program`.

Добавьте к методу `SerialPI` в `try`-блок, который находится после блока `for`, инструкцию `foreach`, выделенную в следующем примере кода жирным шрифтом.

```

try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        ...
    }

    foreach (double datum in pointsList)
    {
        if (datum <= RADIUS)
        {
            numPointsInCircle++;
        }
    }
}

```

Этот код выполняет последовательный обход элементов коллекции `pointsList` и по очереди исследует каждое значение. Если значение меньше или равно радиусу окружности, он увеличивает значение переменной `numPointsInCircle` на единицу. В конце этого цикла в переменной `numPointsInCircle` должно находиться общее количество координат, чье присутствие обнаружено внутри круга.

Добавьте к блоку `try` после инструкции `foreach` следующие инструкции, показанные жирным шрифтом:

```

try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        ...
    }

    foreach (double datum in pointsList)
    {
        ...
    }

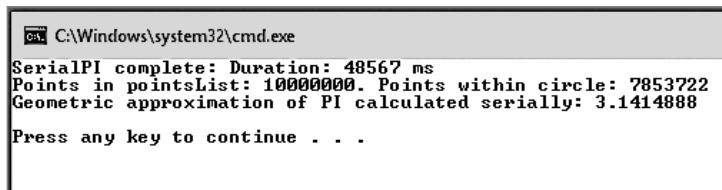
    double pi = 4.0 * numPointsInCircle / NUMPOINTS;
    return pi;
}

```

Первая инструкция вычисляет π на основании отношения числа точек, попавших в круг, к общему количеству точек с применением ранее рассмотренной формулы. Получившееся значение возвращается в качестве результата выполнения метода.

В меню Отладка щелкните на пункте Запуск без отладки.

Программа запустится и выведет приблизительное значение числа π (рис. 24.4). (На моем компьютере на это ушло около 49 с, поэтому наберитесь терпения.) Также будет выведено время, затраченное на вычисление.



```
C:\Windows\system32\cmd.exe
SerialPI complete: Duration: 48567 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888
Press any key to continue . . .
```

Рис. 24.4



ПРИМЕЧАНИЕ Если вы не изменяли значений констант NUMPOINTS, RADIUS или SEED, то за исключением того времени, которое было затрачено на вычисления, вы должны получить точно такой же результат.

Закройте окно консоли и вернитесь в среду Visual Studio.

Несомненно, областью, подлежащей распараллеливанию в методе `SerialPI`, является код в цикле `for`, который генерирует точки и вычисляет их удаление от начала координат. Именно этим распараллеливанием вы и займетесь в следующем упражнении.

Вычисление π с использованием одновременно выполняемых задач

Выполните в окне редактора файла `Program.cs`, дважды щелкнув на его имени в обозревателе решений. Найдите метод `ParallelPI`. В нем содержится точно такой же код, который был в исходной версии метода `SerialPI` до добавления в блок `try` кода вычисления числа π .

Удалите из блока `try` комментарий и инструкцию `return` и добавьте к нему инструкцию `Parallel.For`, показанную жирным шрифтом:

```
try
{
    Parallel.For (0, NUMPOINTS, (x) =>
    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    });
}
```

Эта конструкция является параллельным аналогом кода, который находится в цикле `for` метода `SerialPI`. Тело исходного цикла `for` заключено в лямбда-выражение. Следует напомнить, что каждая итерация цикла выполняется с использованием задачи, а задачи могут запускаться в параллельном режиме.

Степень распараллеливания зависит от количества ядер процессора и объема других ресурсов, доступного на вашем компьютере.

Добавьте к блоку `try` после инструкции `Parallel.For` следующий код, показанный жирным шрифтом. Этот код точно такой же, как и соответствующие ему инструкции в методе `SerialPI`:

```
try
{
    Parallel.For (...  

    {  

        ...  

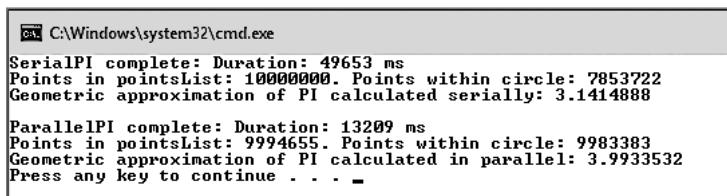
    });  
  

    foreach (double datum in pointsList)
    {
        if (datum <= RADIUS)
        {
            numPointsInCircle++;
        }
    }
    double pi = 4.0 * numPointsInCircle / NUMPOINTS;
    return pi;
}
```

Снимите в методе `Main`, который находится ближе к концу файла `Program.cs`, комментарий с вызова метода `ParallelPI` и с инструкции `Console.WriteLine`, выводящей результаты на экран.

В меню Отладка щелкните на пункте Запуск без отладки.

Программа запустится, и результатом ее работы станет вывод на экран информации, показанной на рис. 24.5 (полученное вами время, затраченное на вычисления, может отличаться от показанного, лично я использовал для работы четырехъядерный процессор).



```
C:\Windows\system32\cmd.exe  

SerialPI complete: Duration: 49653 ms  

Points in pointsList: 10000000. Points within circle: 7853722  

Geometric approximation of PI calculated serially: 3.1414888  

ParallelPI complete: Duration: 13209 ms  

Points in pointsList: 9994655. Points within circle: 9983383  

Geometric approximation of PI calculated in parallel: 3.9933532  

Press any key to continue . . . -
```

Рис. 24.5

Значение, вычисленное методом `SerialPI`, будет точно таким же, что и прежде, а вот результат работы метода `ParallelPI` выглядит как-то подозрительно. Генератор случайных чисел получил то же самое начальное число, которое

использовалось в методе `SerialPI`, следовательно, он должен выдать ту же последовательность случайных чисел с тем же результатом и с тем же количеством точек, попадающих в круг. Странно также то, что коллекция `pointsList` в методе `ParallelPI` содержит меньше точек, чем та же самая коллекция в методе `SerialPI`.



ПРИМЕЧАНИЕ Если коллекция `pointsList` содержит ожидаемое количество элементов, запустите приложение еще раз. В большинстве запусков (но не обязательно во всех из них) окажется, что точек в коллекции меньше, чем ожидалось.

Закройте окно консоли и вернитесь в среду Visual Studio.

В чем же причина столь странной работы параллельного вычисления? Лучше всего приступить к ее выяснению, начав с количества элементов в коллекции `pointsList`. Эта коллекция генерируется объектом-обобщением `List<double>`, но тип этого объекта не предназначен для безопасной работы в многопоточной среде. Код в инструкции `Parallel.For`, предназначенный для добавления элемента в коллекцию, вызывает метод `Add`, но не следует забывать, что этот код выполняется задачами, запускаемыми в одновременно выполняемых потоках. Следовательно, применительно к количеству элементов, добавляемых к коллекции, высока вероятность того, что некоторые вызовы метода `Add` будут мешать друг другу и становиться причиной искажения данных. Решить проблему можно путем применения какой-нибудь коллекции из пространства имен `System.Collections.Concurrent`, поскольку эти коллекции обеспечивают безопасную работу в многопоточной среде. Наверное, наиболее подходящая для нашего примера коллекция в этом пространстве имен будет создаваться классом-обобщением `ConcurrentBag<T>`.

Использование коллекции, безопасно работающей в многопоточной среде

Выполните в окне редактора файл `Program.cs`, дважды щелкнув на его имени в обозревателе решений. Добавьте к началу файла следующую директиву `using`:

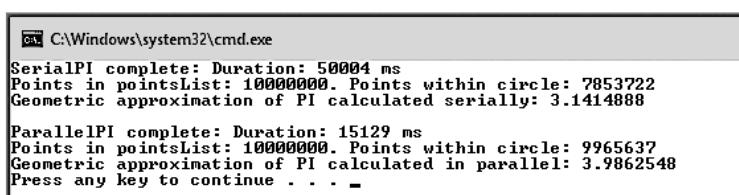
```
using System.Collections.Concurrent;
```

Найдите метод `ParallelPI`. Замените в начале этого метода инструкцию, создающую экземпляр коллекции `List<double>`, кодом, показанным в следующем примере жирным шрифтом, который создает коллекцию `ConcurrentBag<double>`:

```
static double ParallelPI()
{
    ConcurrentBag<double> pointsList = new ConcurrentBag<double>();
    Random random = ...;
    ...
}
```

Заметьте, что вы не можете указать для этого класса исходный объем коллекции, поэтому конструктор не получает никаких параметров. Весь остальной код данного метода в изменениях не нуждается, поскольку элемент к коллекции `ConcurrentBag<T>` добавляется с помощью метода `Add`, то есть используется тот же самый механизм, который использовался для добавления элемента к коллекции `List<T>`.

Щелкните в меню Отладка на пункте Запуск без отладки. Программа запустится и выведет приблизительное значение числа π с использованием методов `SerialPI` и `ParallelPI`. Информация, выводимая программой на экран, показана на рис. 24.6.



```
C:\Windows\system32\cmd.exe
SerialPI complete: Duration: 50004 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 15129 ms
Points in pointsList: 10000000. Points within circle: 9965637
Geometric approximation of PI calculated in parallel: 3.9862548
Press any key to continue . . .
```

Рис. 24.6

На этот раз коллекция `pointsList` в методе `ParallelPI` содержит правильное количество точек, но количество точек, попадающих в круг, по-прежнему слишком велико, а оно должно быть таким же, как количество, показанное после работы метода `SerialPI`.

Следует также заметить, что время, затраченное методом `ParallelPI`, по сравнению с временем, показанным в предыдущем упражнении, увеличилось. Дело в том, что для безопасности данных при работе в многопоточной среде метод в классе `ConcurrentBag<T>` вынужден заниматься блокировкой и разблокированием доступа к этим данным, и к общим издержкам добавляются еще и вызовы соответствующих методов. Это обстоятельство следует учитывать, принимая решение о том, стоит ли распараллеливать операцию.

Закройте окно консоли и вернитесь в среду Visual Studio.

Теперь в коллекции `pointsList` хранится правильное количество точек, но под вопросом остается значение, записанное для каждой из них. Код в конструкции `Parallel.For` вызывает метод `Next` объекта типа `Random`, но так же, как и метод в классе-обобщении `List<T>`, этот метод не обеспечивает безопасную работу в многопоточной среде. К сожалению, версии класса `Random` для работы в параллельном режиме не существует, поэтому для обеспечения последовательного режима вызовов метода `Next` нужно обратиться к использованию альтернативной технологии. Поскольку каждый вызов этого метода занимает относительно немного времени, есть смысл для защиты его вызовов воспользоваться простой блокировкой.

Использование блокировки для обеспечения вызовов метода в последовательном режиме

Выполните в окно редактора файл Program.cs, дважды щелкнув в обозревателе решений на его имени. Найдите метод ParallelPI. Измените код в лямбда-выражении инструкции Parallel.For для защиты вызовов random.Next путем использования инструкции lock. Укажите в качестве предмета блокировки, как показано далее жирным шрифтом, коллекцию pointsList:

```
static double ParallelPI()
{
    ...
    Parallel.For(0, NUMPOINTS, (x) =>
    {
        int xCoord;
        int yCoord;

        lock(pointsList)
        {
            xCoord = random.Next(RADIUS);
            yCoord = random.Next(RADIUS);
        }

        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    });
    ...
}
```

Обратите внимание на то, что переменные xCoord и yCoord объявляются за пределами инструкции lock. Дело в том, что инструкция lock определяет собственное пространство имен и любая переменная, определенная внутри блока, указывающего пространство имен инструкции lock, исчезает, как только происходит выход из конструкции.

Щелкните в меню Отладка на пункте Запуск без отладки.

На этот раз значения π , вычисленные методами SerialPI и ParallelPI, совпадают. Единственная разница заключается в том, что метод ParallelPI выполняется гораздо быстрее (рис. 24.7).

Закройте окно консоли и вернитесь в среду Visual Studio.

```
C:\Windows\system32\cmd.exe
SerialPI complete: Duration: 51366 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 15736 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated in parallel: 3.1414888
Press any key to continue . . .
```

Рис. 24.7

Выводы

В этой главе было показано, как с помощью модификатора `async` и оператора `await` определяются асинхронные методы, работа которых основана на применении задач, используемых для выполнения обработки данных в асинхронном режиме, при этом оператор `await` указывает на те места, в которых задача может использоваться для выполнения асинхронной обработки.

Также вкратце было рассмотрено применение метода расширения `AsParallel`, позволяющего распараллелить некоторые LINQ-запросы. Применение технологии PLINQ является темой, заслуживающей особого внимания, поэтому в данной главе показаны только подступы к ней. Дополнительные сведения можно найти в теме «Parallel LINQ (PLINQ)» документации, предоставленной средой Visual Studio.

В этой главе было показано, как осуществляется синхронизация доступа к данным в одновременно выполняемых задачах путем использования примитивов синхронизации, предоставляемых для использования с задачами. Вы также увидели, как применяются классы коллекций, предназначенные для работы с данными в многопоточной вычислительной среде.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 25 «Реализация пользовательского интерфейса для приложений универсальной платформы Windows».

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Реализовать асинхронный метод	<p>Определите метод с помощью модификатора <code>async</code> и измените тип метода, чтобы он возвращал <code>Task</code> (или <code>void</code>). Используйте в теле метода оператор <code>await</code>, чтобы указать места, в которых может выполняться асинхронная обработка данных, например:</p> <pre>private async Task<int> calculateValueAsync(...) { // Вызов calculateValue с помощью Task Task<int> generateResultTask = Task.Run(() => calculateValue(...)); await generateResultTask; return generateResultTask.Result; }</pre>

Чтобы	Сделайте следующее
Распараллелить LINQ-запрос	Укажите метод расширения AsParallel с источником данных в запросе, например: <pre>var over100 = from n in numbers.AsParallel() where ... select n;</pre>
Позволить отмену в PLINQ-запросе	Воспользуйтесь в PLINQ-запросе методом WithCancellation класса ParallelQuery и укажите признак отмены, например: <pre>CancellationToken tok = ...; ... var orderInfoQuery = from c in CustomersInMemory.Customers.AsParallel(). WithCancellation(tok) join o in OrdersInMemory.Orders.AsParallel() on</pre>
Синхронизировать одну или несколько задач для реализации безопасного в многопоточной среде исключительного доступа к совместно используемым данным	Для обеспечения исключительного доступа к данным воспользуйтесь инструкцией lock, например: <pre>object myLockObject = new object(); ... lock (myLockObject) { // Код, требующий исключительного доступа // к совместно используемому ресурсу ... }</pre>
Синхронизировать потоки и заставить их ожидать наступления события	Для синхронизации неопределенного количества потоков воспользуйтесь объектом типа ManualResetEventSlim. Чтобы получить сигнал о наступлении события определенное количество раз, воспользуйтесь объектом типа CountdownEvent. Для координации конкретного количества потоков и их синхронизации в конкретном месте операции воспользуйтесь объектом типа Barrier
Синхронизировать доступ к общему пулу ресурсов	Воспользуйтесь объектом типа SemaphoreSlim. Укажите в конструкторе количество элементов в пуле. Перед обращением к ресурсу в общем пуле вызовите метод Wait. Завершив использование ресурса, вызовите метод Release, например: <pre>SemaphoreSlim semaphore = new SemaphoreSlim(3); ... semaphore.Wait(); // Обращение к ресурсу из пула ... semaphore.Release();</pre>

Чтобы	Сделайте следующее
Предоставить исключительный доступ к ресурсу по записи, но совместный доступ к нему по чтению	<p>Воспользуйтесь объектом типа ReaderWriterLockSlim. Прежде чем выполнять чтение из совместно используемого ресурса, вызовите метод EnterReadLock. Завершив работу, вызовите метод ExitReadLock. Перед записью в общий ресурс вызовите метод EnterWriteLock. Завершив операцию записи, вызовите метод ExitWriteLock, например:</p> <pre data-bbox="425 436 1045 842"> ReaderWriterLockSlim readerWriterLock = new ReaderWriterLockSlim(); Task readerTask = Task.Factory.StartNew(() => { readerWriterLock.EnterReadLock(); // Чтение из совместно используемого ресурса readerWriterLock.ExitReadLock(); }); Task writerTask = Task.Factory.StartNew(() => { readerWriterLock.EnterWriteLock(); // Запись в совместно используемый ресурс readerWriterLock.ExitWriteLock(); });</pre>
Отменить операцию ожидания блокировки	<p>Создайте признак отмены из объекта CancellationTokenSource и укажите этот признак в качестве параметра операции ожидания. Для отмены операции ожидания вызовите метод Cancel объекта типа CancellationTokenSource, например:</p> <pre data-bbox="425 1023 1058 1351"> CancellationTokenSource cancellationTokenSource = new CancellationTokenSource(); CancellationToken cancellationToken = cancellationTokenSource.Token; ... // Семафор, защищающий пул из трех ресурсов SemaphoreSlim semaphoreSlim = new SemaphoreSlim(3); ... // Ожидание на семафоре и выдача исключения // OperationCanceledException, если // еще один поток вызвал Cancel в отношении // cancellationTokenSource semaphore.Wait(cancellationToken);</pre>

25

Реализация пользовательского интерфейса для приложений универсальной платформы Windows

Прочитав эту главу, вы научитесь:

- характеризовать особенности обычного приложения для универсальной платформы Windows;
- создавать масштабируемый пользовательский интерфейс для приложения универсальной платформы Windows, способный адаптироваться к различным форм-факторам и ориентации устройств;
- создавать стили и применять их к приложению для универсальной платформы Windows.

В последних версиях Windows появилась платформа для создания и запуска приложений с высокой степенью интерактивности, постоянно подключенным и сенсорно управляемым пользовательским интерфейсом и поддержкой встроенных сенсорных устройств. Обновленная система безопасности приложений и модель их жизненного цикла изменяют способ совместной работы пользователей и приложений. Эта платформа называется Windows Runtime (WinRT), и на нее уже встречались ссылки в этой книге. Среду Visual Studio можно использовать для создания WinRT-приложений, подстраиваемых под широкий спектр форм-факторов устройств, от планшетных компьютеров до настольных вычислительных систем с большими экранами высокого разрешения. Используя Windows 8 и Visual Studio 2013, вы можете также публиковать эти приложения в магазине Windows Store в качестве принадлежащих ему приложений.

Помимо этого, для разработки и реализации приложений, запускаемых на устройствах под управлением Windows Phone 8, вы можете воспользоваться

средой Windows Phone SDK 8.0, интегрированной в Visual Studio. В этих приложениях много общего с их ближайшими родственниками, разработанными для планшетных и настольных компьютеров, но они работают в более ограниченной среде, имеющей, как правило, более скромные ресурсы и требующей поддержки другой разметки пользовательского интерфейса. Вследствие этого приложения, разрабатываемые под Windows Phone 8, используют другую версию WinRT, которая называется Windows Phone Runtime, и у вас есть возможность вывести приложения Windows Phone 8 на рынок в качестве приложений магазина Windows Phone Store. С помощью имеющегося в Visual Studio шаблона Библиотека классов (переносимая) вы можете создать библиотеку классов для совместного использования логики приложения и бизнес-логики с работающим под Windows приложением для планшетных и настольных компьютеров и приложением для Windows Phone 8. Но приложения в магазине Windows Store и в магазине Windows Phone Store — это разные приложения, отличающиеся друг от друга предоставляемыми функциями.

Впоследствии компания Microsoft стремилась сблизить эти две платформы и сократить количество различий. Кульминацией этой стратегии стала операционная система Windows 10 с приложениями универсальной платформы Windows, где используется скорректированная версия WinRT, которая и называется универсальной платформой (Universal Windows Platform (UWP)). Используя UWP, можно создавать приложения, которые будут работать на широком спектре устройств под управлением Windows 10, при этом исключается необходимость сопровождения отдельных баз исходного кода. Кроме множества смартфонов, планшетных и настольных компьютеров, UWP доступна также на устройствах Xbox.



ПРИМЕЧАНИЕ Платформа UWP определяет основной набор функций и средств их реализации. В UWP устройства делятся на семейства настольных компьютеров, мобильных устройств, Xbox-устройств и т. д. Для каждого семейства определяется набор API-интерфейсов и устройств, на которых они реализуются. В дополнение к этому в семействе универсального устройства определяется основной набор функций и средств их реализации, доступных на всех семействах устройств. Библиотеки, доступные для каждого семейства устройств, включают условные методы, позволяющие приложению тестировать, на каком семействе устройств оно в данный момент выполняется.

В этой главе дается краткое описание концепций, положенных в основу платформы UWP, которое поможет вам приступить к использованию среды Visual Studio 2015 для создания приложений, работающих в этом окружении. В главе будет рассмотрен ряд функций и инструментальных средств, включенных в Visual Studio 2015 для создания UWP-приложений, с помощью которых будет создано приложение, соответствующее особенностям интерфейса Windows 10. Основное внимание будет уделено изучению способов реализации пользовательского интерфейса, способного к масштабированию и адаптации к различным разрешающим возможностям и форм-факторам устройств, а также способов применения стилей для придания приложению особого оформления.



ПРИМЕЧАНИЕ Чтобы представить полноценный трактат о создании UWP-приложений, в книге слишком мало места. Поэтому ее последние главы сконцентрированы на рассмотрении основных принципов построения интерактивных приложений, использующих пользовательский интерфейс Windows 10. Более подробную информацию о создании UWP-приложений можно получить на странице «Руководство по работе с приложениями универсальной платформы Windows (UWP)» веб-сайта компании Microsoft по адресу <https://msdn.microsoft.com/library/dn894631.aspx>.

Характерные особенности приложения универсальной платформы Windows

Многие современные мобильные устройства и планшетные компьютеры позволяют пользователям взаимодействовать с приложениями с использованием сенсорного ввода. UWP-приложение нужно проектировать на основе этого стиля пользователя восприятия (user experience (UX)). Windows 10 включает обширную коллекцию элементов управления на основе сенсорного ввода, которые работают также с мышью и клавиатурой. Вам в своих приложениях не нужно разделять свойства сенсорного ввода и мыши — просто проектируйте приложения под сенсорный ввод, а пользователи в силу своих предпочтений или использования устройств, не поддерживающих сенсорный ввод, по-прежнему смогут работать с ними, используя мышь и клавиатуру.

Способ, которым графический пользовательский интерфейс (GUI) реагирует на жесты для обеспечения обратной связи с пользователем, может существенно улучшить восприятие ваших приложений — они будут производить впечатление программных продуктов, созданных на высоком профессиональном уровне. Имеющиеся в среде Visual Studio 2015 шаблоны для универсальных приложений Windows включают библиотеку анимации, которой можно воспользоваться в ваших приложениях для приведения этой обратной связи к стандартам операционной системы и программного обеспечения, предоставляемого компанией Microsoft.



ПРИМЕЧАНИЕ Жестикуляция относится к ручным операциям, которые пользователь может выполнять на устройствах сенсорного ввода. Например, пользователь может прикоснуться к элементу пальцем, и реакция на этот жест будет такой же, как и на щелчок на этом элементе мышью. Но жесты могут быть гораздо выразительнее простых операций, фиксируемых при использовании мыши. Например, вращательный жест предполагает прикосновение к экрану двумя пальцами и их перемещение по воображаемой дуге окружности, в обычном приложении, разработанном под Windows 10, этот жест заставит пользовательский интерфейс выполнить вращение выбранного объекта в направлении, показанном движениями пальцев пользователя. К прочим жестам относятся разведение пальцев на экране для увеличения элемента и его более детального отображения, прикосновение, кратковременное удержание пальца на элементе для получения о нем дополнительной информации (этот жест аналогичен щелчку правой кнопкой мыши) и проведение пальцем по экрану для выбора элемента и его перемещения в другое место экрана.

Платформа UWP предназначена для работы на широком спектре устройств с различными размерами и разрешениями экранов. Поэтому при создании UWP-приложения нужно сконструировать свои программные средства так, чтобы они адаптировались к среде, в которой выполняются, и автоматически масштабировались под размер экрана и ориентацию устройства. Такой подход откроет для ваших программ более широкий рынок сбыта. Кроме того, многие современные устройства со встроенными сенсорами и акселерометрами способны определять свою ориентацию в пространстве и скорость, с которой пользователь меняет ее. UWP-приложения могут адаптировать свою разметку под наклоны и вращения устройства, позволяя пользователю работать в наиболее комфортном для него режиме. Следует также понимать, что мобильность является ключевым требованием для многих современных приложений и, применяя UWP-приложения, пользователи могут перемещаться, а их данные могут мигрировать по облачным хранилищам, на каких бы устройствах ни запускалось ваше приложение в данный конкретный момент.

Жизненный цикл UWP-приложения не такой, как у обычного приложения, предназначенного для настольного компьютера. Вам следует разрабатывать приложения, способные работать на таких устройствах, как смартфоны, которые приостанавливают свое выполнение, когда пользователь переключается на другое приложение, а затем возобновляют его, когда пользователь к ним возвращается. На устройствах с ограниченным объемом ресурсов такой подход способствует их экономии, а также уменьшению расхода энергии аккумуляторной батареи. Операционная система Windows может даже принять решение о закрытии приостановленного приложения, если обнаружится, что ей нужно высвободить системные ресурсы, например память. При следующем запуске приложение должно обладать способностью возвращаться на то место, где было приостановлено его выполнение. Это означает, что при определенном стечении обстоятельств вам следует быть готовым управлять в своем коде информацией о состоянии приложения, сохранять эту информацию на жестком диске и впоследствии восстанавливать ее.



ПРИМЕЧАНИЕ Дополнительные сведения об управлении жизненным циклом UWP-приложения можно найти на странице «Guidelines for app suspend and resume» веб-сайта Microsoft по адресу <https://msdn.microsoft.com/library/windows/apps/hh465088.aspx>.

После создания нового UWP-приложения его можно запаковать с помощью инструментальных средств, предоставляемых средой Visual Studio 2015, и выложить в магазин Windows Store. Другие пользователи могут зайти в магазин, скачать ваше приложение и установить его на своем устройстве. Вы можете взимать плату за свои приложения или открыть к ним бесплатный доступ. Возможность использования этого механизма распространения и развертывания

зависит от того, насколько ваше приложение надежно и в какой степени оно соответствует политикам безопасности, установленным компанией Microsoft. Когда приложение выкладывают в магазине Windows Store, оно подвергается ряду проверок, позволяющих убедиться, что в нем не содержится вредоносный код и оно соответствует требованиям безопасности, предъявляемым к UWP-приложениям. Эти ограничения, связанные с выполнением требований безопасности, предписывают порядок доступа вашего приложения к ресурсам компьютера, на который оно устанавливается. Например, изначально UWP-приложение не может вести запись непосредственно в файловую систему или прослушивать входящие сетевые запросы (это два стиля поведения, присущие вирусам и другим вредоносным программам). Но если ваше приложение нуждается в выполнении операций, на которые наложены ограничения, их можно указать в качестве возможностей в данных манифеста приложения, хранящегося в файле Package.appxmanifest. Эта информация записывается в метаданных вашего приложения и сигнализирует Microsoft о необходимости выполнения дополнительных тестов для проверки способа использования приложением этих функциональных средств.

Файл Package.appxmanifest является XML-документом, но вы можете его отредактировать в среде Visual Studio с помощью конструктора манифеста. Пример показан на рис. 25.1. Здесь для указания подпадающих под ограничения операций, которые приложение может выполнять, используется вкладка Возможности.

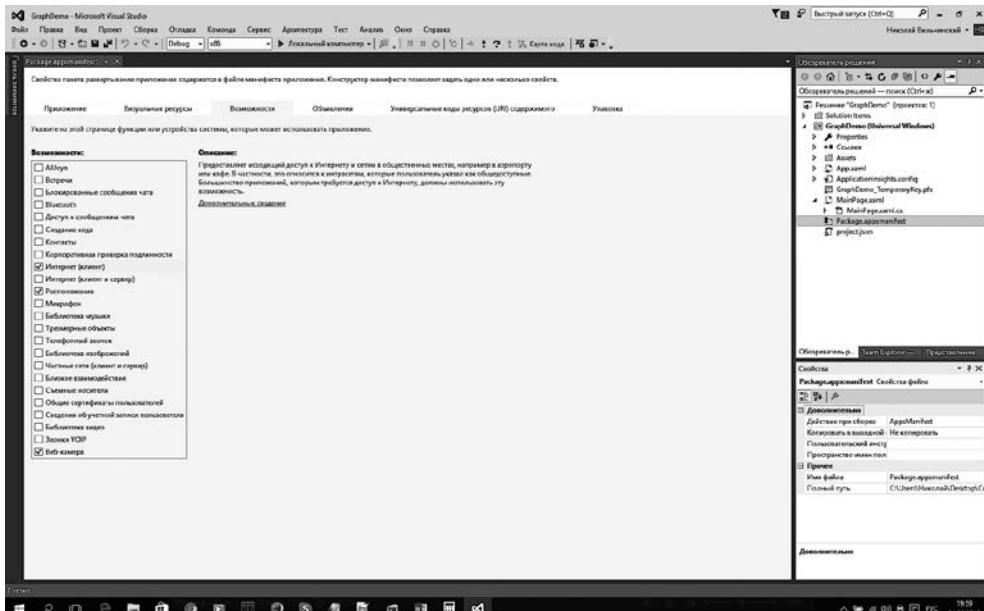


Рис. 25.1

В этом примере приложение объявляет, что ему нужно:

- ❑ получать данные, поступающие из Интернета, но не иметь возможности действовать в качестве сервера и не иметь доступа к локальной сети;
- ❑ считывать и записывать данные, хранящиеся в папке документов пользователя;
- ❑ иметь доступ к информации GPS, позволяющей определить местоположение устройства;
- ❑ получать доступ к видеоканалу встроенной камеры или внешней веб-камеры.

Эти требования становятся известны пользователю, и в любом случае он может отключить настройки после установки приложения, а приложение должно обнаружить подобное отключение и быть готовым к откату, альтернативному решению или отключению функциональной возможности, требующей выполнения данных функций.



ПРИМЕЧАНИЕ Дополнительные сведения о возможностях, поддерживаемых UWP-приложением, можно найти на странице «Объявления возможностей приложения» веб-сайта Microsoft по адресу <http://msdn.microsoft.com/library/windows/apps/hh464936.aspx>.

Хватит теории, давайте приступим к созданию UWP-приложения.

Использование шаблона пустого приложения для создания приложения универсальной платформы Windows

Проще всего создать UWP-приложение, воспользовавшись шаблонами универсального приложения Windows, включенными в среду Visual Studio 2015, работающую под управлением Windows 10. Многие основанные на графическом интерфейсе пользователя приложения, реализованные в предыдущих главах, использовали шаблон Пустое приложение, являющийся вполне подходящим местом для начала работы.

В следующих упражнениях вы будете проектировать пользовательский интерфейс для простого приложения, разрабатываемого для вымышленной компании под названием Adventure Works. Эта компания производит и поставляет велосипеды и сопутствующие товары. Приложение позволит пользователю вводить и править сведения о клиентах компании Adventure Works.

Создание приложения Adventure Works Customers

В меню Файл среды Visual Studio 2015 укажите на пункт Создать, а затем щелкните на пункте Проект.

В левой панели диалогового окна Создание проекта раскройте пункт Шаблоны, затем подпункт Visual C#, подпункт Windows и щелкните на пункте Универсальные. Щелкните в средней панели на значке Пустое приложение (Универсальное приложение Windows). Наберите в поле Имя строку Customers. Наберите в поле Расположение строку \Microsoft Press\VCBS\Chapter 25 с префиксом в виде вашей папки документов. Щелкните на кнопке OK. Будет создано новое приложение, и в окно редактора будет выведен файл App.xaml.cs. Пока его можно проигнорировать.

В обозревателе решений дважды щелкните на файле MainPage.xaml. Появится окно конструктора с отображением пустой страницы. Чтобы добавить нужные приложению элементы управления, их, как было показано в главе 1 «Добро пожаловать в C#», можно перетащить из панели элементов. Но для выполнения задач данного упражнения поучительнее будет обратиться к определяющей формат XAML-разметке. У нее должен быть следующий вид:

```
<Page
    x:Class="Customers.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Customers"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    </Grid>
</Page>
```

Форма начинается с XAML-тега `<Page>` и завершается закрывающим тегом `</Page>`. Все, что находится между этими двумя тегами, определяет содержимое страницы.

В атрибутах `<Page>`-тега содержится несколько объявлений в формате `xmlns:id = "..."`. Это объявления пространства имен XAML, которые действуют аналогично директивам C# `using` в том смысле, что вводят элементы в область видимости. Многие элементы управления и другие элементы, которые можно добавлять на страницу, определены в этих пространствах имен XAML, и большинство этих объявлений можно проигнорировать. Но здесь есть одно довольно любопытное объявление, на которое следует обратить внимание:

```
xmlns:local="using:Customers"
```

Это объявление помещает элементы в область видимости C# `Customers`. В коде XAML на классы и другие типы в этом пространстве имен можно ссылаться, используя для них префикс `local`. Пространство имен `Customers` сгенерировано для кода в вашем приложении.

В обозревателе решений раскройте узел `MainPage.xaml`, а затем дважды щелкните на файле `MainPage.xaml.cs`, чтобы его содержимое отобразилось в окне редактора. Из предыдущих упражнений этой книги можно было понять, что это файл C#, в котором находятся логика приложения и обработчики событий для формы. Он имеет следующий вид (инструкции `using`, находящиеся в верхней части файла, для экономии пространства опущены):

```
// Документацию по шаблону элемента "Пустая страница" см. по адресу
// http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409
namespace Customers
{
    /// <summary>
    /// Пустая страница, которую можно использовать саму по себе или для
    /// перехода внутри фрейма.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

В этом файле определяются типы в пространстве имен `Customers`. Страница реализуется классом по имени `MainPage`, являющимся наследником класса `Page`. Класс `Page` реализует исходные функциональные возможности XAML-страницы для UWP-приложения, и вам остается лишь написать в классе `MainPage` код, определяющий логику, специфичную для вашего приложения.

Вернитесь в окне конструктора к файлу `MainPage.xaml`. Посмотрите на XAML-разметку для страницы и убедитесь, что тег `<Page>` включает следующий атрибут:

```
x:Class="Customers.MainPage"
```

Этот атрибут подключает XAML-разметку, определяющую формат страницы, к классу `MainPage`, предоставляющему логику,ложенную в основу страницы.

Это основное подключение простого UWP-приложения. Разумеется, ценность графического приложения определяет способ предоставления информации пользователю. Но не всегда все так просто, как кажется. Проектирование привлекательного и удобного в использовании графического интерфейса требует особых навыков, которые имеются не у всех разработчиков (я это знаю,

поскольку сам испытываю их дефицит). Но многие художники-графики, обладающие подобными навыками, не являются программистами и, обладая способностью спроектировать великолепный пользовательский интерфейс, могут быть не в состоянии реализовать логику, придающую этому интерфейсу полезные свойства. К счастью, среда Visual Studio 2015 позволяет отделить дизайн пользовательского интерфейса от бизнес-логики, в результате чего художник-график и разработчик могут сотрудничать, создавая приложение, имеющее великолепный внешний вид и отличные рабочие параметры. Разработчику остается сконцентрироваться на основной разметке приложения, а работу по стилевому оформлению предоставить художнику.

Реализация масштабируемого пользовательского интерфейса

Основа разметки пользовательского интерфейса для UWP-приложения заключается в понимании того, как его можно заставить масштабироваться и подстраиваться под разные форм-факторы, доступные устройствам, на которых пользователи могут запускать приложение. Способы достижения способности к масштабированию будут исследованы в следующих упражнениях.

Разметка страницы для приложения Customers

Обратите внимание на то, что в панели инструментов в верхней части окна конструктора (рис. 25.2) имеются раскрывающийся список, позволяющий выбрать разрешение и форм-фактор рабочей области конструирования, и две кнопки, позволяющие выбрать ориентацию (книжную или альбомную) для тех устройств, которые поддерживают вращение (планшеты и смартфоны его поддерживают, а настольные компьютеры — нет). Цель в том, чтобы позволить вам использовать эти настройки для быстрого просмотра того, как пользовательский интерфейс будет выглядеть на различных устройствах.

Изначально разметка показывается для смартфона с 5-дюймовым экраном в книжной ориентации. Выберите в раскрывающемся списке 0" Desktop (1280 x 720) 100% масштаб. Обратите внимание на то, что исходной ориентацией для этого форм-фактора является альбомная. Посмотрите на XAML-разметку для страницы MainPage. Она содержит один элемент управления Grid:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
</Grid>
```



ПРИМЕЧАНИЕ Обращать внимание на способ указания свойства фона Background для элемента управления Grid пока не нужно. Это пример использования стиля, а стили будут рассмотрены в этой главе чуть позже.

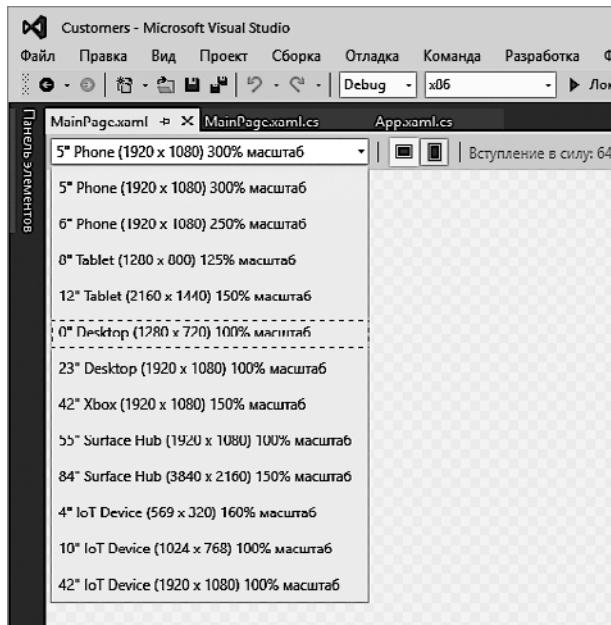


Рис. 25.2

При построении масштабируемых и гибких пользовательских интерфейсов основную роль играет понимание того, как работает элемент управления **Grid**. Элемент **Page** может содержать только один элемент, и, как показано в следующем примере, если нужно, элемент управления **Grid** можно заменить элементом управления **Button** (кнопка).



ПРИМЕЧАНИЕ Не набирайте следующий код, он показан исключительно в учебных целях:

```
<Page
  ...
  <Button Content="Click Me"/>
</Page>
```

Но польза от получившегося в результате приложения весьма сомнительна, потому что форма, содержащая кнопку и больше ничего не показывающая, вряд ли завоюет приз как самое удачное приложение в мире. При попытке добавления на страницу второго элемента управления, например текстового поля типа **TextBox**, ваш код не откомпилируется и будет выдана ошибка (рис. 25.3).

Элемент управления **Grid** предназначен для того, чтобы помочь добавить на страницу несколько элементов. Этот элемент является примером элемента управления типа «контейнер» — в нем могут содержаться несколько других

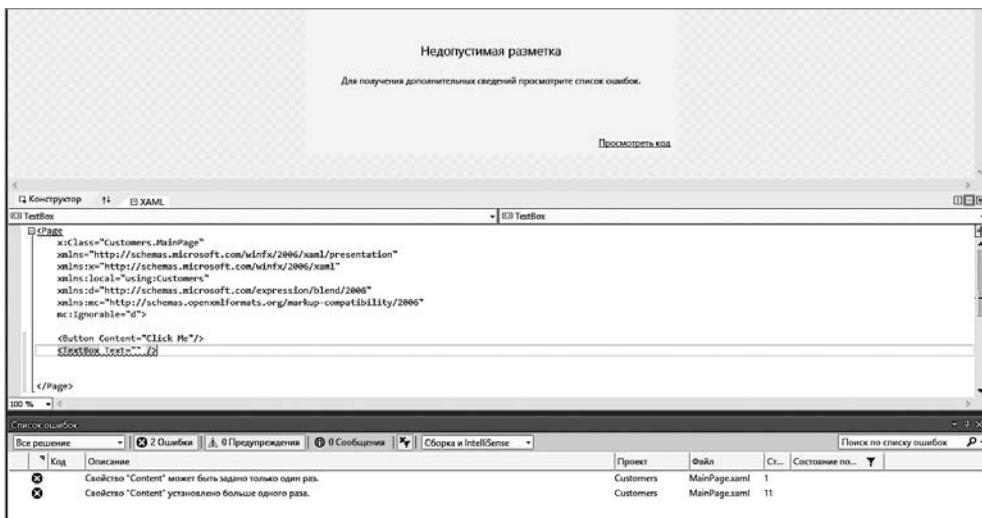


Рис. 25.3

элементов управления, и вы можете указать позиции этих элементов внутри данного контейнера. Допустимо использование и других элементов управления типа «контейнер». Например, элемент управления `StackPanel` автоматически размещает содержащиеся в нем элементы управления в вертикальной раскладке с позиционированием каждого элемента управления непосредственно под его предшественником.

В данном упражнении `Grid` будет использован для содержания элементов управления, необходимых пользователю для ввода и просмотра данных о клиенте.

Добавьте на страницу элемент управления `TextBlock`, перетащив его из панели элементов или введя строку `<TextBlock />` непосредственно на панели XAML в пустой строке после открывающего тега `<Grid>`:

```

<Grid Background="{ThemeResource ApplicationBackgroundThemeBrush}">
    <TextBlock />
</Grid>
  
```



СОВЕТ Если панель элементов не появилась, щелкните на пункте Панель элементов в меню Вид, и она должна быть выведена в левой части окна. Также утите, что вам можно набрать код элемента управления непосредственно в окне XAML для данной страницы — перетаскивать элементы из панели элементов совсем не обязательно.

Этот блок текста типа `TextBlock` предоставляет заголовок страницы. Установите для элемента управления типа `TextBlock` свойства, воспользовавшись значениями, показанными в табл. 25.1.

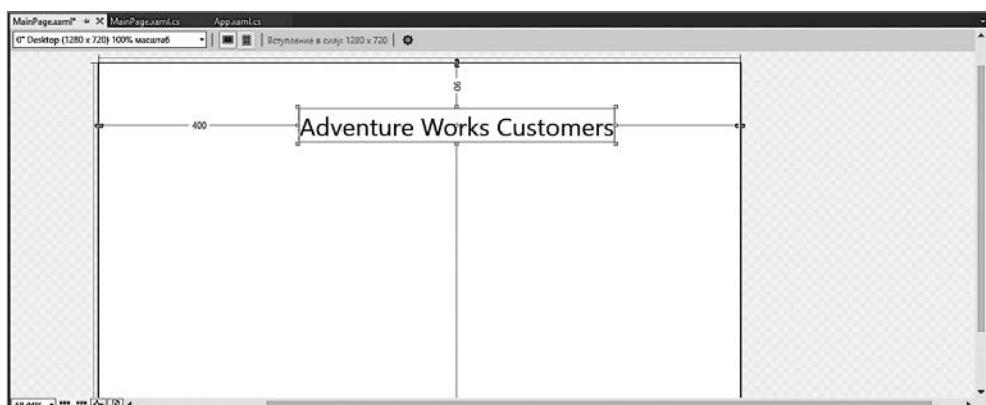
Таблица 25.1

Свойство	Значение
HorizontalAlignment	Left
Margin	400,90,0,0
TextWrapping	Wrap
Text	Adventure Works Customers
VerticalAlignment	Top
FontSize	50

Эти свойства можно установить, воспользовавшись окном Свойства или путем ввода эквивалентной XAML-разметки, показанной жирным шрифтом, в окне XAML:

```
<TextBlock HorizontalAlignment="Left" Margin="400,90,0,0" TextWrapping="Wrap"
Text="Adventure Works Customers" VerticalAlignment="Top" FontSize="50"/>
```

Получившийся в результате текст должен отобразиться в окне конструктора (рис. 25.4).

**Рис. 25.4**

Заметьте, что при перетаскивании элемента управления из панели элементов в форму с двух сторон элемента появляются линии, соединяющие эти стороны с краями элемента управления типа «контейнер», куда помещается перетаскиваемый элемент, с указанием расстояния между соединяемыми точками. В предыдущем примере соединительные линии для элемента управления типа

`TextBlock` содержали надписи 400 (от левого края контейнера) и 90 (от его верхнего края). Если в ходе выполнения программы размеры элемента управления `Grid` изменяются, `TextBlock` будет перемещаться для сохранения этого расстояния, что в данном случае может привести к изменению расстояния в пикселях, на которое `TextBlock` будет удален от правого и нижнего краев элемента `Grid`. Можно указать край или края, к которым элемент будет привязан, задав значения свойств `HorizontalAlignment` и `VerticalAlignment`. Расстояние от краев, к которым осуществлена привязка, указывается в свойстве `Margin`. В данном примере для свойства `HorizontalAlignment` элемента `TextBlock` установлено значение `Left`, а для свойства `VerticalAlignment` — значение `Top`, благодаря чему элемент управления привязан к левому и верхнему краям контейнера. Свойство `Margin` имеет четыре значения, в которых указываются расстояния от левой, верхней, правой и нижней сторон (именно в таком порядке) элемента управления до соответствующих краев контейнера. Если одна из сторон элемента управления не привязана к краю контейнера, в свойстве `Margin` соответствующее значение можно установить равным 0.

Добавьте на страницу еще четыре элемента типа `TextBlock`. Они являются надписями, помогающими пользователю определять данные, отображаемые на странице. Для установки свойств этих элементов управления воспользуйтесь значениями из табл. 25.2.

Таблица 25.2

Элемент управления	Свойство	Значение
First Label	HorizontalAlignment	Left
	Margin	330,190,0,0
	TextWrapping	Wrap
	Text	ID
	VerticalAlignment	Top
	FontSize	20
Second Label	HorizontalAlignment	Left
	Margin	460,190,0,0
	TextWrapping	Wrap
	Text	Title
	VerticalAlignment	Top
	FontSize	20

Таблица 25.2 (окончание)

Элемент управления	Свойство	Значение
Third Label	HorizontalAlignment	Left
	Margin	620,190,0,0
	TextWrapping	Wrap
	Text	First Name
	VerticalAlignment	Top
	FontSize	20
Fourth Label	HorizontalAlignment	Left
	Margin	975,190,0,0
	TextWrapping	Wrap
	Text	Last Name
	VerticalAlignment	Top
	FontSize	20

Как и прежде, можно либо перетащить эти элементы управления из панели элементов и воспользоваться для установки их свойств окном Свойства, либо набрать в панели XAML следующую XAML-разметку, поместив ее после элемента управления типа `TextBlock` и перед закрывающим тегом `</Grid>`:

```
<TextBlock HorizontalAlignment="Left" Margin="330,190,0,0" TextWrapping="Wrap"
Text="ID" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="460,190,0,0" TextWrapping="Wrap"
Text="Title" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="620,190,0,0" TextWrapping="Wrap"
Text="First Name" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="975,190,0,0" TextWrapping="Wrap"
Text="Last Name" VerticalAlignment="Top" FontSize="20"/>
```

Ниже элементов управления `TextBlock` добавьте три элемента управления типа `TextBox`, которые отображают текст идентификационного номера, имени и фамилии. Для установки значений этих элементов управления воспользуйтесь табл. 25.3. Обратите внимание на то, что для свойства `Text` должно быть установлено значение пустой строки (""). Также заметьте, что элемент управления `id` типа `TextBox` помечен как доступный только для чтения. Дело в том, что клиентские идентификационные номера (ID) станут генерироваться в коде, который будет добавлен чуть позже в автоматическом режиме.

Таблица 25.3

Элемент управления	Свойство	Значение
First TextBox	x:Name	Id
	HorizontalAlignment	Left
	Margin	300,240,0,0
	TextWrapping	Wrap
	Text	—
	VerticalAlignment	Top
	FontSize	20
	IsReadOnly	True
Second TextBox	x:Name	firstName
	HorizontalAlignment	Left
	Margin	550,240,0,0
	TextWrapping	Wrap
	Text	—
	VerticalAlignment	Top
	FontSize	20
Third TextBox	x:Name	lastName
	HorizontalAlignment	Left
	Margin	875,240,0,0
	TextWrapping	Wrap
	Text	—
	VerticalAlignment	Top
	FontSize	20

Эквивалентная XAML-разметка для этих элементов управления показана в следующем примере кода:

```
<TextBox x:Name="id" HorizontalAlignment="Left" Margin="300,240,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" FontSize="20"
IsReadOnly="True"/>
<TextBox x:Name="firstName" HorizontalAlignment="Left" Margin="550,240,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="300" FontSize="20"/>
<TextBox x:Name="lastName" HorizontalAlignment="Left" Margin="875,240,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="300" FontSize="20"/>
```

Свойство `Name` для элемента управления обязательным не является, но оно пригодится, если на этот элемент нужно будет сослаться в коде C# для приложения. Обратите внимание на то, что у свойства `Name` имеется префикс `x:`. Это ссылка на пространство имен XML `http://schemas.microsoft.com/winfx/2006/xaml`, указанное в атрибутах страницы в верхней части XAML-разметки. Это пространство имен определяет свойство `Name` для всех элементов управления.



ПРИМЕЧАНИЕ Разбираться в том, почему свойство `Name` определено таким образом, совсем не обязательно, но для получения дополнительных сведений можно прочитать статью «`x:Name Directive`» по адресу <http://msdn.microsoft.com/library/ms752290.aspx>.

В свойстве `Width` указывается ширина элемента управления, а в свойстве `TextWrapping` показано, что произойдет, если пользователь попытается ввести в элемент управления информацию, превышающую его ширину. В данном случае все элементы управления типа `TextBox` будут переносить текст на новую строку той же ширины (элемент управления увеличится по вертикали). Альтернативное значение, `NoWrap`, приведет к тому, что текст по мере ввода пользователем будет прокручиваться по горизонтали.

Добавьте к форме элемент управления типа `ComboBox`, поместив его ниже элемента управления `Title` типа `TextBlock` и расположив между элементами `id` и `firstName` типа `TextBox`. Установите для свойств этого элемента управления следующие значения (табл. 25.4).

Таблица 25.4

Свойство	Значение
<code>x:Name</code>	<code>Title</code>
<code>HorizontalAlignment</code>	<code>Left</code>
<code>Margin</code>	<code>420,240,0,0</code>
<code>VerticalAlignment</code>	<code>Top</code>
<code>Width</code>	<code>100</code>
<code>FontSize</code>	<code>20</code>

Эквивалентная XAML-разметка для этого элемента управления имеет следующий вид:

```
<ComboBox x:Name="title" HorizontalAlignment="Left" Margin="420,240,0,0"
VerticalAlignment="Top" Width="100" FontSize="20"/>
```

Элемент управления типа `ComboBox` используется для отображения списка значений, из которых пользователь может выбрать нужные.

В окне конструктора щелкните на элементе управления ComboBox. Раскройте в окне Свойства категорию свойств Общие. Затем щелкните на кнопке с многоточием, появившейся рядом со свойством Items. Откроется окно Редактор коллекций "Object". Выберите в списке в левом нижнем углу окна элемент ComboBoxItem, а затем щелкните на кнопке Добавить. Раскройте в правой панели, отображающей свойства элемента, раздел Общие, а затем наберите в поле свойства Content строку Mr (рис. 25.5).

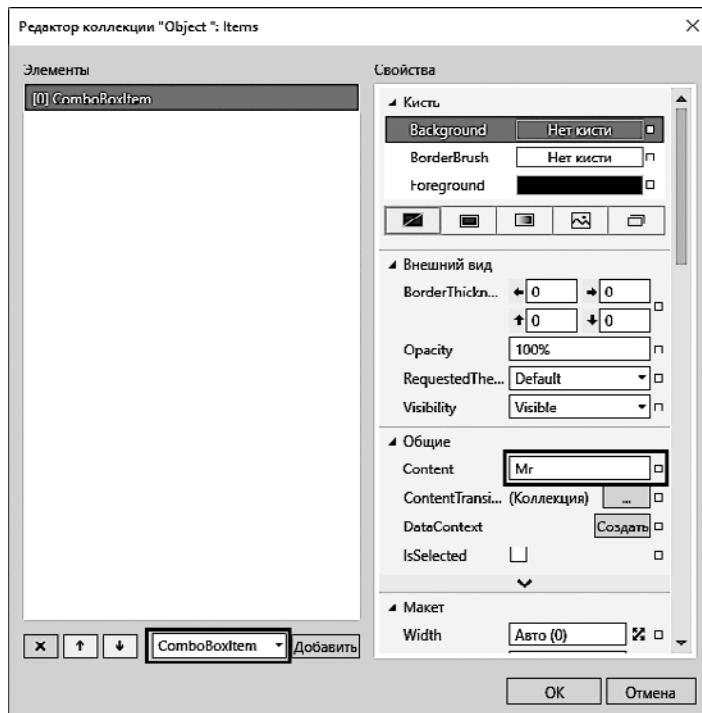


Рис. 25.5

Щелкните на кнопке OK. Редактор коллекций закроется. Если посмотреть на XAML-разметку для элемента title типа ComboBox, то в ней должен быть следующий код:

```
<ComboBox x:Name="title" HorizontalAlignment="Left" Margin="420,240,0,0"
          VerticalAlignment="Top" Width="100" FontSize="20"/>
    <ComboBoxItem Content="Mr"/>
</ComboBox>
```

Здесь следует отметить два обстоятельства. Первое заключается в том, что разметка ComboBox была разбита на открывающий тег <ComboBox> и закрывающий

тег </ComboBox>. А второе заключается в том, что между этими тегами среда Visual Studio добавила элемент `ComboBoxItem` со свойством `Content`, для которого установлено значение `Mr`. Это значение при работе приложения будет показано в качестве элемента раскрывающегося списка.

Добавьте к элементу `title` типа `ComboBox` значения `Mrs`, `Ms` и `Miss`. Для этого можете либо воспользоваться редактором коллекций, либо набрать XAML-разметку вручную. Получившаяся в результате этого разметка должна иметь следующий вид:

```
<ComboBox x:Name="title" HorizontalAlignment="Left" Margin="420,240,0,0"
    VerticalAlignment="Top" Width="75" FontSize="20">
    <ComboBoxItem Content="Mr"/>
    <ComboBoxItem Content="Mrs"/>
    <ComboBoxItem Content="Ms"/>
    <ComboBoxItem Content="Miss"/>
</ComboBox>
```



ПРИМЕЧАНИЕ Элемент управления типа `ComboBox` может выводить на экран простые элементы, такие как набор элементов управления типа `ComboBoxItem`, показывающих текст, но может выводить и более сложные элементы, такие как кнопки, флаги и переключатели. При добавлении простых элементов типа `ComboBoxItem`, наверное, будет проще набрать XAML-разметку вручную, но при добавлении более сложных элементов управления весьма полезным может оказаться редактор коллекций `Object`. Но особо изощряться в поле со списком не стоит. Самыми удачными получаются приложения, предоставляющие наиболее интуитивно понятные пользовательские интерфейсы, а вставка в поле со списком сложных элементов управления может только сбить пользователя с толку.

Добавьте к форме еще два элемента управления типа `TextBox` и еще два элемента управления типа `TextBlock`. Элементы управления `TextBox` позволяют пользователю вводить адреса электронной почты и номера телефонов клиентов, а элементы управления `TextBlock` предоставляют надписи для текстовых полей. Для установки свойств элементов управления воспользуйтесь значениями из табл. 25.5.

Таблица 25.5

Элемент управления	Свойство	Значение
Первый <code>TextBlock</code>	HorizontalAlignment	Left
	Margin	300,390,0,0
	TextWrapping	Wrap
	Text	Email
	VerticalAlignment	Top
	FontSize	20

Элемент управления	Свойство	Значение
Первый TextBox	x:Name	Email
	HorizontalAlignment	Left
	Margin	450,390,0,0
	TextWrapping	Wrap
	Text	Leave Empty
	VerticalAlignment	Top
	Width	400
	FontSize	20
Второй TextBlock	HorizontalAlignment	Left
	Margin	300,540,0,0
	TextWrapping	Wrap
	Text	Phone
	VerticalAlignment	Top
	FontSize	20
Второй TextBox	x:Name	phone
	HorizontalAlignment	Left
	Margin	450,540,0,0
	TextWrapping	Wrap
	Text	Leave Empty
	VerticalAlignment	Top
	Width	200
	FontSize	20

XAML-разметка для этих элементов управления должна иметь следующий вид:

```
<TextBlock HorizontalAlignment="Left" Margin="300,390,0,0" TextWrapping="Wrap"
Text="Email" VerticalAlignment="Top" FontSize="20"/>
<TextBox x:Name="email" HorizontalAlignment="Left" Margin="450,390,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="400" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="300,540,0,0" TextWrapping="Wrap"
Text="Phone" VerticalAlignment="Top" FontSize="20"/>
<TextBox x:Name="phone" HorizontalAlignment="Left" Margin="450,540,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="200" FontSize="20"/>
```

Форма в окне конструктора должна выглядеть следующим образом (рис. 25.6).

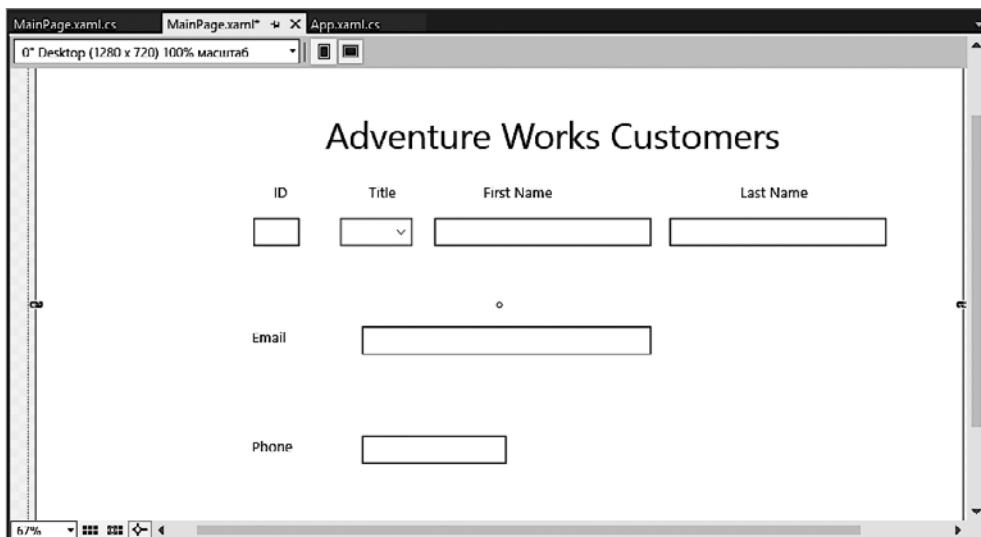


Рис. 25.6

Щелкните в меню Отладка на пункте Начать отладку, чтобы инициировать сборку и запуск приложения.

Приложение будет запущено и выведет на экран форму, изображенную на рис. 25.7. Вы можете вводить данные и выбирать из поля со списком вид

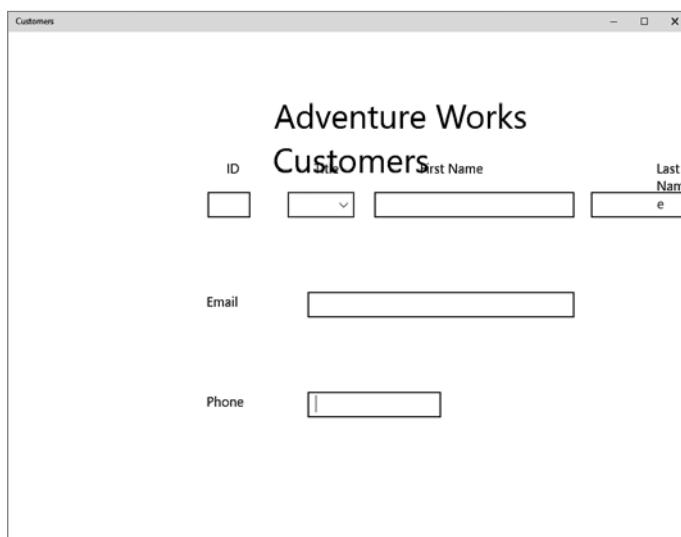


Рис. 25.7

обращения к клиенту, но пока это все, что вам доступно. Куда более серьезной проблемой является весьма неприглядный внешний вид формы. Правая сторона отображаемой формы получилась обрезанной, много текста было перенесено на новую строку, а текстовое поле *Last Name* оказалось усеченным.

Перетащите правую сторону окна, чтобы расширить область отображения и добиться вывода на экран текста и элементов управления в том виде, в котором они появлялись в окне конструктора среды Visual Studio. Это и будет оптимальным размером разработанной формы.

Сделайте ширину окна, отображающего приложение *Customer*, минимальной. Основная часть формы исчезнет. Часть содержимого элементов типа *TextBlock* будет отображена с переносом на новые строки, но форма в этом представлении станет совершенно бесполезной.

Вернитесь в среду Visual Studio и в меню Отладка щелкните на пункте Остановить отладку.

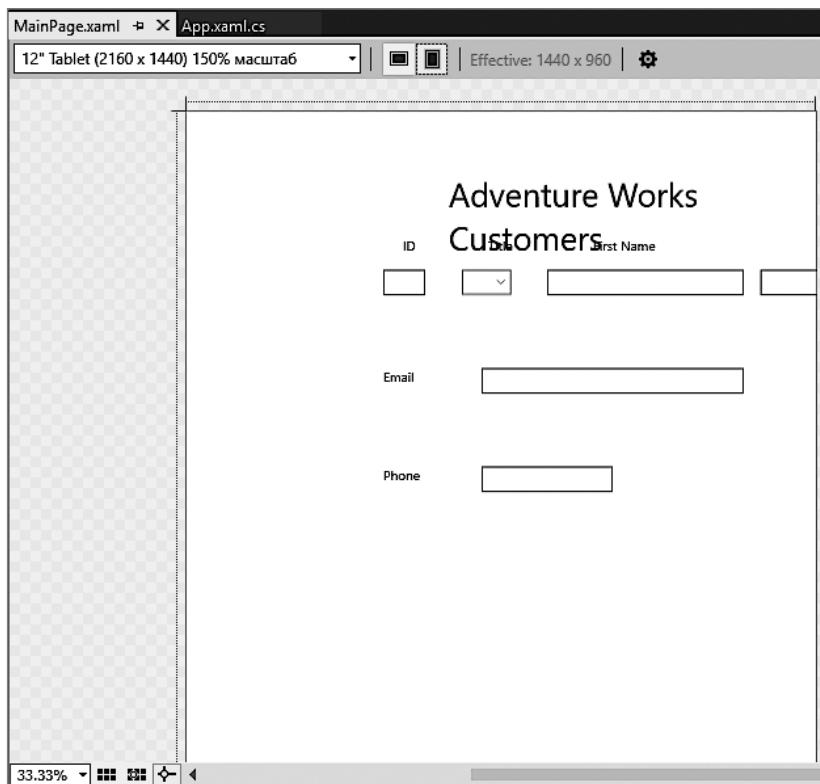


Рис. 25.8

ИСПОЛЬЗОВАНИЕ СИМУЛЯТОРА ДЛЯ ТЕСТИРОВАНИЯ ПРИЛОЖЕНИЙ УНИВЕРСАЛЬНОЙ ПЛАТФОРМЫ WINDOWS

Используя симулятор, предоставляемый средой Visual Studio 2015, вы можете протестировать свои UWP-приложения даже при отсутствии планшетного компьютера и увидеть, как они ведут себя на мобильном устройстве. Симулятор имитирует планшетное устройство, предоставляя возможность эмулировать такие жесты пользователя, как сведение и разведение пальцев на объектах и проведение пальцем по объекту, а также вращение и смену разрешения устройства.

Для запуска приложения в симуляторе откройте в панели инструментов Visual Studio раскрывающийся список целевого объекта отладки (Debug Target). Изначально в качестве целевого объекта отладки устанавливается Локальный компьютер, что заставляет приложение запускаться в полноэкранном режиме вашего компьютера, но в этом списке можно выбрать пункт Симулятор, запустив при отладке приложения симулятор. Заметьте, что вы также можете указать в качестве целевого объекта отладки другой компьютер на тот случай, если вам нужно выполнить удаленную отладку (при выборе этого варианта будет предложено ввести сетевой адрес удаленного компьютера). Список выбора целевого объекта отладки показан на рис. 25.9.

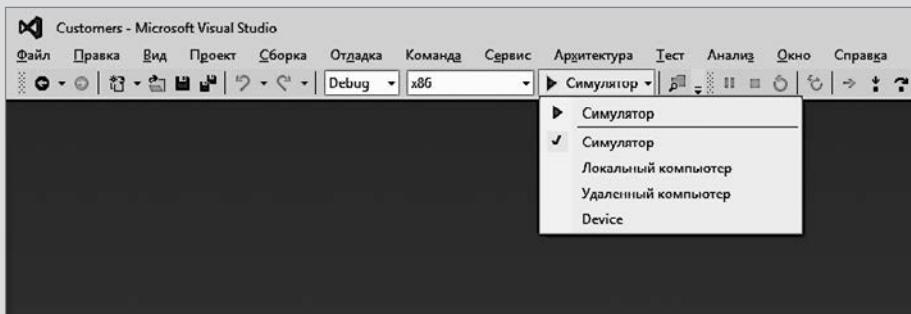


Рис. 25.9

После выбора варианта Симулятор при запуске приложения в Visual Studio из меню Отладка будет запущен симулятор, который выведет приложение на экран. Панель инструментов, расположенная с правой стороны окна симулятора, содержит подборку инструментов, позволяющих имитировать жесты пользователя с помощью мыши. Можно даже имитировать местоположение пользователя, если приложению требуется информация о географическом расположении устройства. Но для тестирования разметки приложения наиболее важными инструментами будут кнопки Повернуть по часовой стрелке на 90 градусов, Повернуть против часовой стрелки на 90 градусов и Изменить разрешение. На рис. 25.10 показано приложение Customers, запущенное в симуляторе. Приложение было раскрыто на весь экран. Надписи описывают функции всех кнопок симулятора.

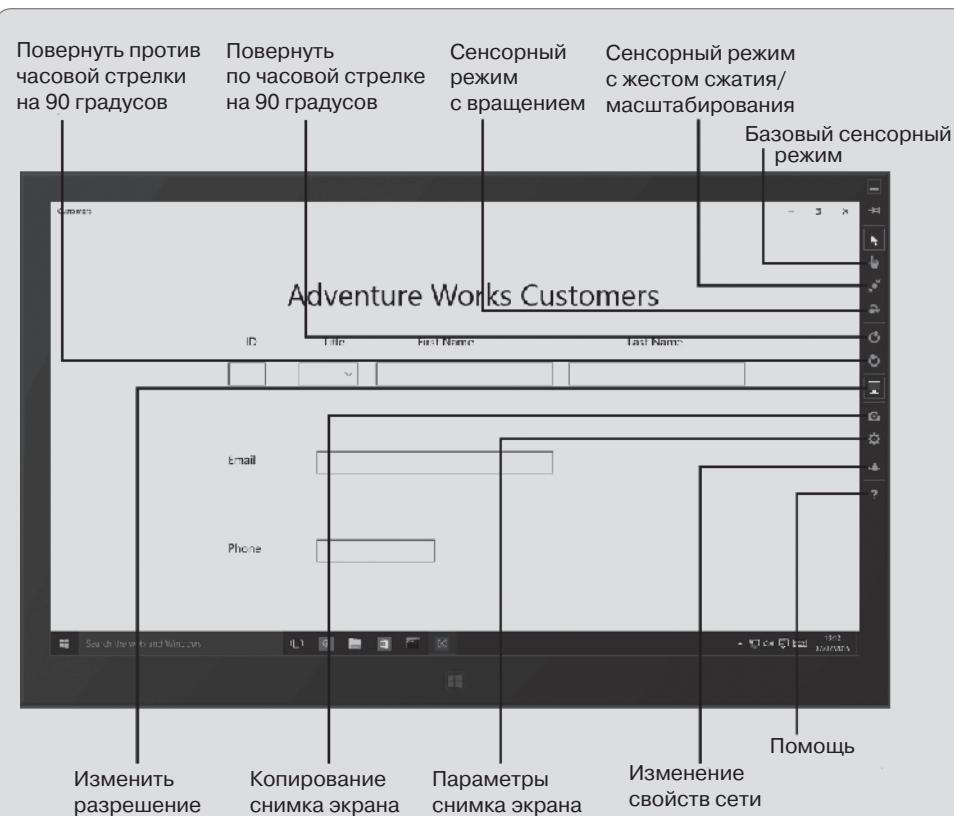


Рис. 25.10



ПРИМЕЧАНИЕ Копии экрана, демонстрируемые в данном разделе, были сделаны на компьютере с симулятором, запущенным при разрешении 1366 × 768, представляющим дисплей с диагональю 10,6 дюйма. Чтобы получить точно такие же результаты при использовании другого разрешения экрана, может понадобиться щелкнуть на кнопке Изменение разрешения и переключиться на разрешение 1366 × 768.

На рис. 25.11 показано то же самое приложение после щелчка пользователя на кнопке Повернуть по часовой стрелке на 90 градусов, что заставило приложение выполнять в книжной ориентации.

Можно также попробовать посмотреть, как приложение поведет себя, если изменить разрешение симулятора. На рис. 25.12 показано приложение Customers, запущенное при установленном на симуляторе высоком разрешении (2560 × 1440, которое является обычным разрешением 27-дюймового монитора). Видно, что область отображения приложения втиснута в верхний левый угол экрана.

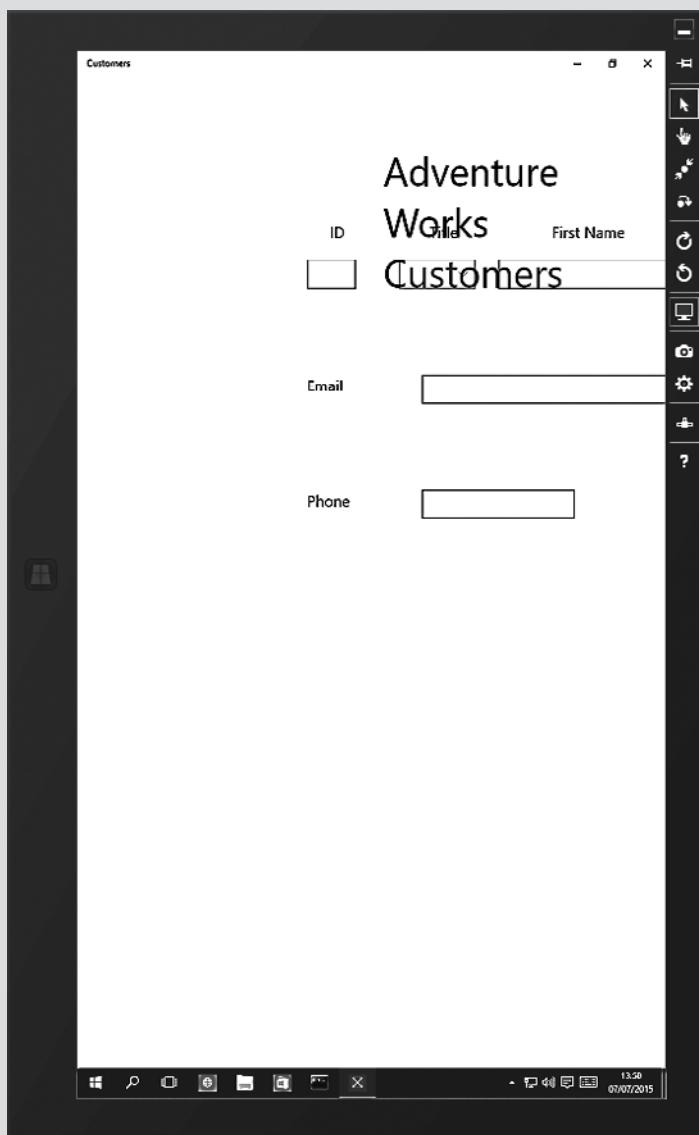


Рис. 25.11

Симулятор ведет себя точно так же, как компьютер, работающий под управлением Windows 10 (это, по сути, подключение к вашему компьютеру удаленного Рабочего стола). Чтобы остановить симулятор, щелкните на кнопке Windows (в симуляторе, а не на вашем Рабочем столе), а затем на пункте Выключение и на пункте Отключиться.

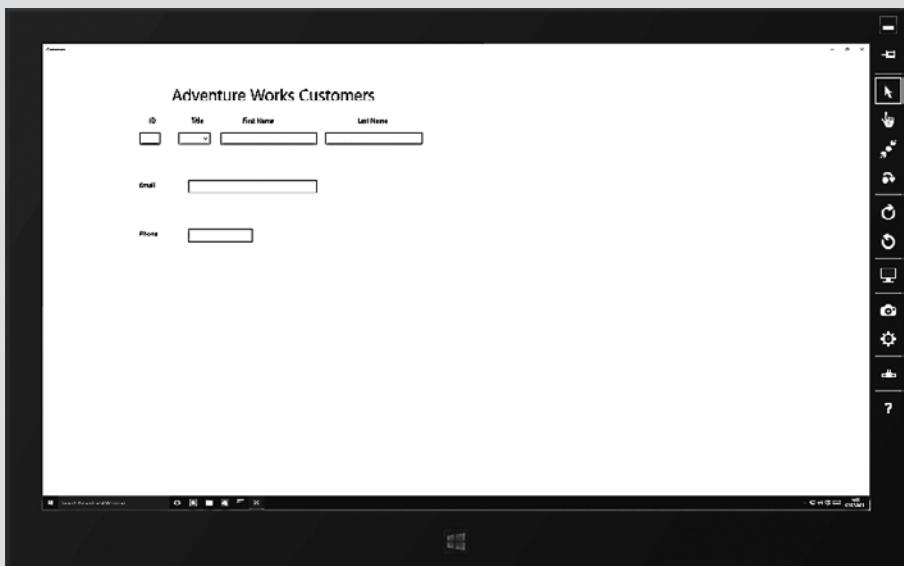


Рис. 25.12

Следует заметить, что среда Visual Studio поддерживает также эмуляторы для конкретных мобильных устройств. Некоторые из них могут быть перечислены в раскрывающемся списке симулятора, но вы сможете загружать новые эмуляторы, выбрав пункт загрузки новых эмуляторов.

Вы получили весьма поучительный урок, показывающий важность тщательной разметки приложения. Несмотря на то что приложение при запуске в окне того же размера, что и окно конструктора, имеет вполне приемлемый внешний вид, как только окно сужается, польза от приложения уменьшается или же оно становится абсолютно бесполезным. Кроме того, в приложении предполагается, что пользователь будет просматривать экран на устройстве с альбомной ориентацией. Если временно переключить окно конструктора на форм-фактор 12" Tablet и щелкнуть на кнопке ориентации Книжная, можно увидеть, что форма станет выглядеть так, будто пользователь запустил приложение на планшетном компьютере, поддерживающем различные варианты ориентации, и повернул устройство для переключения в книжный режим (рис. 25.8). (Не забудьте после этого переключить режим отображения обратно на форм-фактор 0" Desktop.)

Проблема в том, что рассмотренная на данный момент технология разметки не позволяет выполнять масштабирование и адаптироваться к различным форм-факторам и ориентации экрана. К счастью, для решения этой проблемы

можно воспользоваться свойствами элемента управления `Grid` и еще одним функциональным средством, которое называется Диспетчером визуальных состояний.

Реализация табличной разметки путем использования элемента управления `Grid`

Для реализации табличной разметки можно воспользоваться элементом управления `Grid`. В `Grid` содержатся строки и столбцы, и можно указать, в какие строки и столбцы должны быть помещены другие элементы управления. Привлекательность элемента управления `Grid` состоит в том, что вы можете указать размеры содержащихся в нем строк и столбцов в качестве относительных значений: по мере того как таблица сжимается или расширяется, адаптируясь под различные форм-факторы и ориентацию, на которые может переключиться пользователь, строки и столбцы могут сжиматься и расширяться пропорционально таблице. Пересечение строк и столбцов в таблице определяет ячейки, и если вы помещаете элементы управления в ячейки, то по мере сжатия или расширения строк и столбцов они будут перемещаться. Поэтому ключом к реализации масштабируемого пользовательского интерфейса является его разбиение на коллекцию ячеек и помещение связанных друг с другом элементов в одну и ту же ячейку. Ячейка может содержать еще одну таблицу, позволяя точно настроить позиционирование каждого элемента.

Если присмотреться к приложению *Customers*, можно заметить, что пользовательский интерфейс разбивается на две основные области: заголовок, содержащий название, и тело, содержащее подробные данные о клиенте. Как показано на следующей схеме (рис. 25.13), каждой из этих областей можно выделить относительные размеры с учетом некоторого промежутка между ними и отступа в нижней части формы.

На схеме показаны только начальные прикидки, где строка для заголовка вдвое выше строки-промежутка под ним. Стока для тела в 10 раз выше этого промежутка, а нижний отступ в два раза выше него.

Для хранения элементов в каждой области можно определить таблицу с четырьмя строками и поместить в каждую строку соответствующие элементы. Но как показано на рис. 25.14, тело формы может быть описано с помощью другой, более сложной таблицы.

Здесь опять высота каждой строки, как и ширина каждого столбца, указаны в относительных выражениях. Кроме того, совершенно очевидно, что `TextBox`-элементы для `Email` и `Phone` не вписываются в точности в эту табличную схему. Обладая достаточной степенью педантизма, можно выбрать вариант

определения внутри тела формы дополнительных таблиц, чтобы поместить в них эти элементы. Но при этом нужно помнить о назначении данной таблицы, заключающемся в определении относительного позиционирования и создании промежутков между элементами. Поэтому выход элемента за границы ячейки табличной разметки вполне приемлем.

The screenshot shows a Windows application window titled "Adventure Works Customers". Inside, there are several input fields with labels: "ID", "Title", "First Name", "Last Name", "Email", and "Phone". The "Email" field is a long horizontal input, and the "Phone" field is a shorter one. To the right of these fields, there are two columns of text: "2x", "x", and "10x". Below the "Phone" field, there is another "2x" column. The "Email" field is positioned such that its right edge aligns with the "10x" label. The "Phone" field is positioned such that its right edge aligns with the second "2x" label below it.

Рис. 25.13

	ID	Title		First Name		Last Name		<i>y</i>
	<input type="text"/>	<input type="button" value="▼"/>		<input type="text"/>		<input type="text"/>		<i>y</i>
								<i>2y</i>
Email				<input type="text"/>		<input type="text"/>		<i>y</i>
								<i>2y</i>
Phone				<input type="text"/>	<input type="text"/>	<input type="text"/>		<i>y</i>
								<i>4y</i>
<i>z</i>	<i>z</i>	<i>z</i>		<i>2z</i>		<i>2z</i>		<i>z</i>

Рис. 25.14

В следующем упражнении вам предстоит усовершенствовать разметку приложения *Customers*, воспользовавшись для позиционирования элементов управления этим табличным форматом.

Изменение разметки с целью ее масштабирования под различные форм-факторы и варианты ориентации

Добавьте в существующий *Grid*-элемент в XAML-панели приложения *Customers* еще один элемент *Grid*. Задайте, как показано в следующем примере жирным шрифтом, для нового *Grid*-элемента отступы по 10 пикселов от левой и правой сторон родительского *Grid*-элемента и по 20 пикселов от его верхней и нижней сторон:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid Margin="10,20,10,20">
        </Grid>
        <TextBlock HorizontalAlignment="Left" TextWrapping="Wrap"
Text="Adventure Works Customers" ... />
        ...
    </Grid>
```

Строки и столбцы можно определить как часть существующего *Grid*-элемента, но чтобы по внешнему виду приложение ничем не отличалось от других UWP-приложений, нужно оставить в левой и верхней частях страницы немного пустого пространства.

Добавьте к новому *Grid*-элементу следующий *<Grid.RowDefinitions>*-раздел, показанный жирным шрифтом:

```
<Grid Margin="10,20,10,20">
    <Grid.RowDefinitions>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="10*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
</Grid>
```

Раздел *<Grid.RowDefinitions>* определяет строки таблицы. В данном примере определены четыре строки. Можно указать размер строки в виде абсолютного значения в пикселях или же воспользоваться оператором *, показывающим, что размеры являются относительными и что Windows должна высчитать размеры строк самостоятельно при выполнении приложения в зависимости от форм-фактора и разрешения экрана. Значения, использованные в данном примере, соответствуют относительным размерам показанных на предыдущей схеме строк для заголовка, тела, разделяющего их пространства и нижнего отступа формы *Customers*.

Переместите в `Grid` элемент управления типа `TextBlock`, содержащий текст «Adventure Works Customers», поставив его сразу же после закрывающего тега `</Grid.RowDefinitions>`.

Добавьте к элементу управления `TextBlock` атрибут `Grid.Row` и установите для него значение 0.

Это укажет на необходимость позиционирования `TextBlock` внутри первой строки `Grid`. (В элементах управления `Grid` нумерация строк и столбцов начинается с нуля.)



ПРИМЕЧАНИЕ Атрибут `Grid.Row` является примером прикрепляемого свойства. Прикрепляемым называется свойство, которое элемент управления получает от элемента управления типа «контейнер», в котором он размещен. За пределами таблицы у `TextBlock` нет свойства `Row` (его наличие было бы бессмысленным), но при позиционировании внутри таблицы свойство `Row` прикрепляется к `TextBlock` и элемент управления `TextBlock` может присвоить ему значение. Затем элемент управления `Grid` использует это значение, чтобы определить, где нужно отобразить элемент управления `TextBlock`. Прикрепляемые свойства легко обнаружить, поскольку они имеют форму `ТипКонтейнера.ИмяСвойства`.

Удалите свойство `Margin` и установите для свойств `HorizontalAlignment` и `VerticalAlignment` значение `Center`. Это заставит `TextBlock` появляться в строке по центру.

XAML-разметка для элементов управления `Grid` и `TextBlock` должна приобрести следующий вид (изменения в `TextBlock` выделены жирным шрифтом):

```
<Grid Margin="10,20,10,20">
    <Grid.RowDefinitions>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="10*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Adventure Works Customers" VerticalAlignment="Center" FontSize="50"/>
    ...
</Grid>
```

Добавьте после элемента управления `TextBlock` еще одну вложенную таблицу в виде элемента управления `Grid`. Этот `Grid`-контейнер будет использоваться для разметки элементов управления внутри тела формы и должен появиться в третьей строке внешнего `Grid`-контейнера (с размером строки, указанным как `10*`), поэтому, как показано далее жирным шрифтом, установите для свойства `Grid.Row` значение 2:

```
<Grid Margin="10,20,10,20">
    <Grid.RowDefinitions>
```

```
<RowDefinition Height="2*"/>
<RowDefinition Height="*"/>
<RowDefinition Height="10*"/>
<RowDefinition Height="2*"/>
</Grid.RowDefinitions>
<TextBlock Grid.Row="0" HorizontalAlignment="Center" .../>
<Grid Grid.Row="2">
</Grid>
...
</Grid>
```

Добавьте к новому элементу управления `Grid` разделы `<Grid.RowDefinitions>` и `<Grid.ColumnDefinitions>`:

```
<Grid Grid.Row="2">
<Grid.RowDefinitions>
<RowDefinition Height="*"/>
<RowDefinition Height="*"/>
<RowDefinition Height="2*"/>
<RowDefinition Height="*"/>
<RowDefinition Height="2*"/>
<RowDefinition Height="*"/>
<RowDefinition Height="4*"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="20"/>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="20"/>
<ColumnDefinition Width="2*"/>
<ColumnDefinition Width="20"/>
<ColumnDefinition Width="2*"/>
<ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
</Grid>
```

Эти определения строк и столбцов указывают высоту и ширину каждой строки и каждого столбца, показанных ранее на схеме, изображающей структуру тела формы. Между каждым столбцом, в котором будут содержаться элементы управления, имеется небольшое пространство в 20 пикселов.

Переместите элементы управления `TextBlock`, показывающие надписи `ID`, `Title`, `Last Name` и `First Name`, поместив их внутри вложенного элемента управления `Grid` сразу же после закрывающего тега `</Grid.ColumnDefinitions>`.

Установите для свойства `Grid.Row` каждого элемента управления `TextBlock` значение `0` (эти надписи будут появляться в первой строке таблицы). Установите для свойства `Grid.Column` надписи `ID` значение `1`, для свойства `Grid.Column` надписи `Title` — значение `3`, для свойства `Grid.Column` надписи `First Name` — значение `5` и для свойства `Grid.Column` надписи `Last Name` — значение `7`.

Удалите свойство `Margin` из каждого элемента управления `TextBlock` и установите для их свойств `HorizontalAlignment` и `VerticalAlignment` значение `Center`. XAML-разметка для этих элементов управления должна получить следующий вид (все изменения выделены жирным шрифтом):

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0" Grid.Column="1" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="ID" VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Row="0" Grid.Column="3" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Title" VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Row="0" Grid.Column="5" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="First Name" VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Row="0" Grid.Column="7" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Last Name" VerticalAlignment="Center" FontSize="20"/>
</Grid>
```

Переместите элементы управления типа `TextBox` с именами `id`, `firstName` и `lastName`, а также элемент управления типа `ComboBox` с именем `title`, поместив их внутри вложенного элемента управления `Grid` сразу же после элемента управления `TextBlock` с текстом `Last Name`.

Поместите эти элементы управления в строку 1 элемента управления `Grid`. Поместите элемент управления `id` в столбец 1, элемент управления `title` — в столбец 3, элемент управления `firstName` — в столбец 5, а элемент управления `lastName` — в столбец 7.

Удалите свойство `Margin` каждого из этих элементов управления и установите для свойства `VerticalAlignment` значение `Center`. Удалите свойство `Width` и установите для свойства `HorizontalAlignment` значение `Stretch`. Это заставит элементы управления занять при отображении всю ячейку, а также расширяться и сужаться при изменении размера ячейки.

Полный вид XAML-разметки для этих элементов управления с выделенными жирным шрифтом изменениями должен выглядеть следующим образом:

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    ...
    <TextBlock Grid.Row="0" Grid.Column="7" ... Text="Last Name" .../>
```

```

<TextBox Grid.Row="1" Grid.Column="1" x:Name="id"
    HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
    VerticalAlignment="Center" FontSize="20" IsReadOnly="True"/>
<TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName"
    HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
    VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName"
    HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
    VerticalAlignment="Center" FontSize="20"/>
<ComboBox Grid.Row="1" Grid.Column="3" x:Name="title"
    HorizontalAlignment="Stretch" VerticalAlignment="Center" FontSize="20">
    <ComboBoxItem Content="Mr"/>
    <ComboBoxItem Content="Mrs"/>
    <ComboBoxItem Content="Ms"/>
    <ComboBoxItem Content="Miss"/>
</ComboBox>
</Grid>

```

Переместите элемент управления `TextBlock` для надписи `Email` и элемент управления `TextBox` по имени `email` во вложенный элемент управления `Grid`, поставив их сразу же после закрывающего тега элемента управления `ComboBox` по имени `title`. Поместите эти элементы управления в строку 3 элемента управления `Grid`. Поместите надпись `Email` в столбец 1, а элемент управления `TextBox` по имени `email` — в столбец 3. Кроме того, установите для свойства `Grid.ColumnSpan` элемента `TextBox` по имени `email` значение 5, чтобы, как показано на предыдущей схеме, элемент мог распространяться на значение, указанное в его свойстве `Width` через пять столбцов.

Установите для свойства `HorizontalAlignment` элемента управления надписи `Email` значение `Center`, а для свойства `HorizontalAlignment` элемента `TextBox` по имени `email` оставьте прежнее значение `Left` — этот элемент управления должен оставаться выровненным по левому краю относительно первого столбца, поверх которого он располагается, а не по центру относительно всех этих столбцов.

Установите для свойства `VerticalAlignment` надписи `Email` и элемента управления `TextBox` по имени `email` значение `Center`. Удалите для обоих этих элементов свойство `Margin`.

Полное определение этих элементов управления показано в следующей XAML-разметке:

```

<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    ...
    <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" ...>
        ...

```

```
</ComboBox>
<TextBlock Grid.Row="3" Grid.Column="1" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Email" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="5" x:Name="email"
HorizontalAlignment="Left" TextWrapping="Wrap" Text="" 
VerticalAlignment="Center" Width="400" FontSize="20"/>
</Grid>
```

Переместите элемент управления `TextBlock` для надписи `Phone` и элемент управления `TextBox` по имени `phone` во вложенный элемент управления `Grid` и поставьте сразу же после элемента управления `TextBox` по имени `email`.

Поместите эти элементы управления в строку 5 элемента управления `Grid`. Поместите элемент управления с надписью `Phone` в столбец 1, а элемент управления `TextBox` по имени `phone` — в столбец 3. Установите для свойства `Grid.ColumnSpan` элемента управления `TextBox` по имени `phone` значение 3.

Установите для свойства `HorizontalAlignment` элемента управления с надписью `Phone` значение `Center`, а для свойства `HorizontalAlignment` элемента `TextBox` по имени `phone` оставьте значение `Left`.

Установите для свойства `VerticalAlignment` обоих элементов управления значение `Center` и удалите свойство `Margin`.

Полное определение этих элементов управления показано в следующей XAML-разметке:

```
<Grid Grid.Row="2">
  <Grid.RowDefinitions>
    ...
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    ...
  </Grid.ColumnDefinitions>
  ...
  <TextBox ... x:Name="email" .../>
  <TextBlock Grid.Row="5" Grid.Column="1" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Phone" VerticalAlignment="Center" FontSize="20"/>
  <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3" x:Name="phone"
HorizontalAlignment="Left" TextWrapping="Wrap" Text="" 
VerticalAlignment="Center" Width="200" FontSize="20"/>
</Grid>
```

На панели инструментов Visual Studio выберите в раскрывающемся списке целевого объекта отладки пункт **Симулятор**. Приложение будет запущено в симуляторе, чтобы вы могли увидеть, как разметка адаптируется к различным разрешениям экрана и форм-факторам.

В меню **Отладка** щелкните на пункте **Начать отладку**. Симулятор начнет работу, и приложение `Customers` запустится. Раскройте окно приложения так, чтобы оно заняло весь экран симулятора. Щелкните на кнопке **Изменить разрешение**,

а затем настройте симулятор на отображение приложения с использованием разрешения экрана 1366 × 768. Кроме того, убедитесь, что симулятор отображается в альбомной ориентации (если он отображается в книжной ориентации, щелкните на кнопке Повернуть по часовой стрелке на 90 градусов). Убедитесь в равномерном распределении элементов управления в этой ориентации.

Щелкните на кнопке Повернуть против часовой стрелки на 90 градусов, чтобы изменить ориентацию симулятора на книжную.

Приложение *Customers* должно подстроить разметку пользовательского интерфейса, и элементы управления должны быть по-прежнему равномерно распределены по форме и пригодны к работе (рис. 25.15).

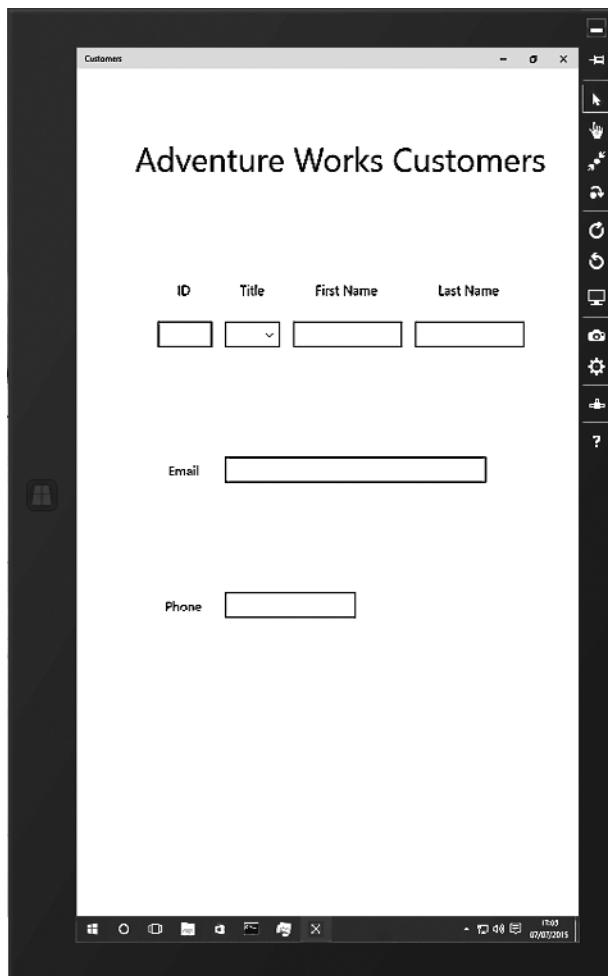


Рис. 25.15

Щелкните на кнопке Повернуть против часовой стрелки на 90 градусов, чтобы вернуть симулятору альбомную ориентацию, а затем щелкните на кнопке Изменить разрешение и переключите разрешение симулятора на 2560×1400 .

Заметьте, что элементы управления по-прежнему равномерно распределены по форме, хотя надписи, если только у вас нет экрана с диагональю 27 дюймов, могут читаться с большим трудом.

Еще раз щелкните на кнопке Изменить разрешение и переключите разрешение симулятора на 1024×768 .

Еще раз обратите внимание на то, что расстояния между элементами и их размеры отрегулированы так, чтобы поддерживать равномерный баланс элементов пользовательского интерфейса (рис. 25.16).

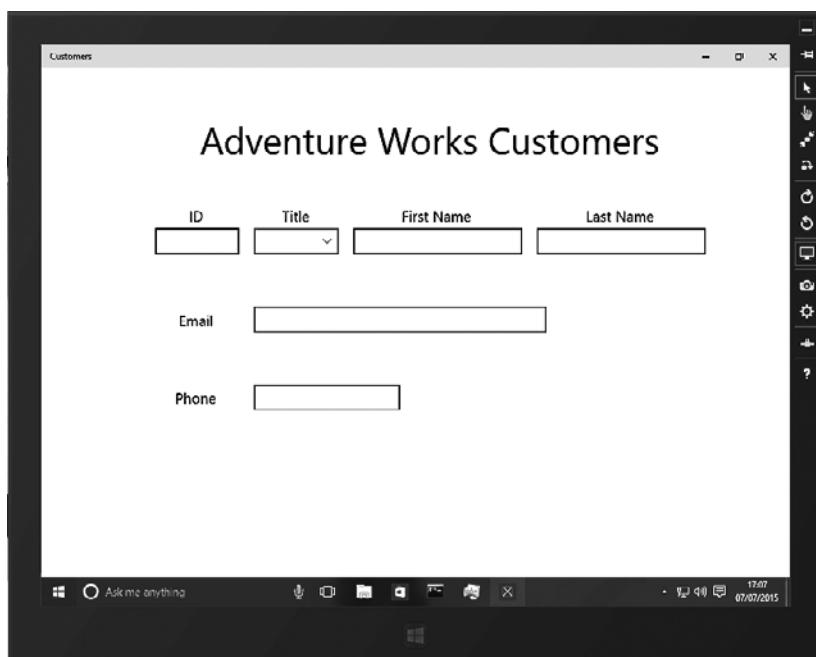


Рис. 25.16

Еще раз щелкните на кнопке Изменить разрешение и опять переключите разрешение симулятора на 1366×768 .

Дважды щелкните в симуляторе на верхнем крае формы, чтобы отображение вернулось в режим окна, а затем перетащите границу окна, изменив его размер так, чтобы форма отображалась в левой половине экрана. Уменьшите ширину окна до минимума. Таким приложение может появляться на смартфонах.

Все элементы управления остаются в поле зрения, но текст для надписи Phone и заголовок переносятся на новую строку, затрудняя чтение и усложняя тем самым использование элементов управления (рис. 25.17).

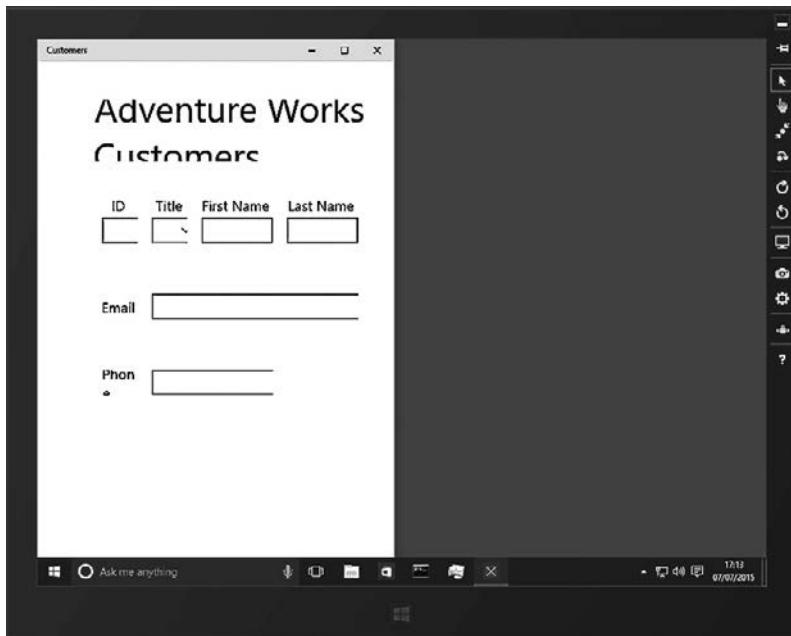


Рис. 25.17

Щелкните в симуляторе на кнопке Пуск (со значком Windows), затем на пункте Выключение и пункте Отключиться. Симулятор закроется, и вы вернетесь в среду Visual Studio.

На панели инструментов Visual Studio в раскрывающемся списке целевого объекта отладки выберите пункт Локальный компьютер.

Подстраивание разметки с помощью Диспетчера визуальных состояний

Пользовательский интерфейс для приложения Customers масштабируется для различных разрешений экрана и форм-факторов, но его работа при уменьшении ширины области просмотра по-прежнему оставляет желать лучшего, и его внешний вид на смартфоне, ширина экрана которого еще меньше, теряет привлекательность. Если вдуматься, то решение проблемы в таком случае заключается не в масштабировании элементов управления, а в другом

способе их разметки. Например, было бы разумнее, если бы при узкой области просмотра у формы Customers был следующий внешний вид (рис. 25.18).

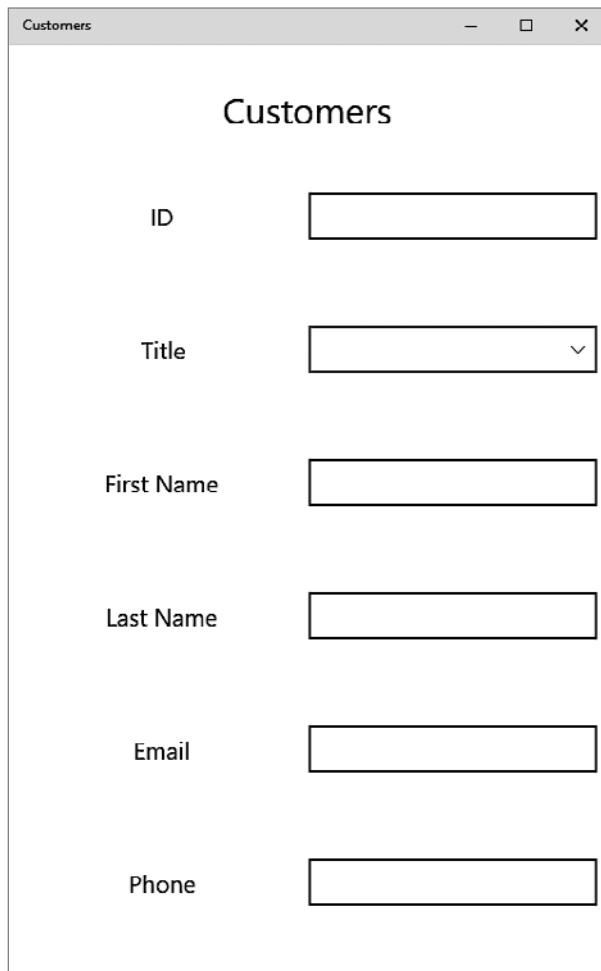


Рис. 25.18

Такого эффекта можно достичь несколькими способами.

- ❑ Можно создать несколько версий файла MainPage.xaml, по одной для каждого семейства устройств. Каждый из этих XAML-файлов может быть связан с одной и той же программной базой (MainPage.xaml.cs), чтобы все они запускали один и тот же код. Например, для создания XAML-файла для смартфона добавьте к проекту папку по имени DeviceFamily-Mobile (имя здесь будет играть важную роль), а затем добавьте в папку новое XAML-представление

по имени `MainPage.xaml`, воспользовавшись командой меню Добавить новый элемент. Создайте в этой папке страницы разметку элементов управления, соответствующую их отображению на смартфоне. XAML-представление будет автоматически привязано к существующему файлу `MainPage.xaml.cs`. В ходе выполнения универсальная платформа Windows выберет подходящее представление на основе типа устройства, на котором запущено приложение.

- ❑ Можно воспользоваться Диспетчером визуальных состояний, изменения с его помощью разметку страницы в ходе выполнения программы. Все UWP-приложения реализуют Диспетчер визуальных состояний, отслеживающий визуальное состояние приложения. Он может определить момент изменения высоты и ширины окна, и вы можете добавить XAML-разметку, позиционирующую элементы управления в зависимости от размера окна. Эта разметка может перемещать элементы или же делать их видимыми и невидимыми.
- ❑ Можно использовать Диспетчер визуальных состояний для переключения между представлениями на основе высоты и ширины экрана. Этот подход является сочетанием первых двух рассмотренных здесь вариантов, но в нем легче разобраться (для него не требуется большого объема хитроумного XAML-кода, вычисляющего наиболее подходящие места для каждого элемента управления), а также он наиболее гибок (будет работать, если уменьшить ширину окна на том же самом устройстве).

В следующих упражнениях вы станете придерживаться третьего из этих подходов. На первом этапе будет определена разметка для данных о клиенте, которые должны появляться при узкой области просмотра.

Определение разметки для узкой области просмотра

Добавьте в панели XAML для приложения `Customers` к элементу управления `Grid` свойства `x:Name` и `Visibility`, показанные далее жирным шрифтом:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid x:Name="customersTabularView" Margin="10,20,10,20"
          Visibility="Collapsed">
        ...
    </Grid>
</Grid>
```

Этот элемент управления `Grid` будет содержать исходное представление формы. Позже на этот элемент управления `Grid` будут делаться ссылки в другой XAML-разметке из этого набора упражнений, поэтому нужно дать ему имя. Свойство `Visibility` указывает, виден этот элемент управления (`Visible`) или скрыт (`Collapsed`). Значением по умолчанию является `Visible`, но до некоторых пор, пока определяется другая таблица для отображения данных в формате столбцов, этот `Grid` будет скрыт.

Добавьте после закрывающего тега `</Grid>` для Grid-элемента `customersTabularView` еще один элемент управления `Grid`. Установите для его свойства `x:Name` значение `customersColumnarView`, для свойства `Margin` — значение `10,20,10,20`, а для свойства `Visibility` — значение `Visible`.



СОВЕТ Щелкните на значках «+» и «-», появляющихся вдоль левого края XAML-разметки в панели XAML, принадлежащей окну конструктора, вы можете развернуть и свернуть элементы, упрощая тем самым чтение структуры.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid x:Name="customersTabularView" Margin="10,20,10,20"
          Visibility="Collapsed">
        ...
    </Grid>
    <Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    </Grid>
</Grid>
```

Этот элемент управления `Grid` будет содержать «узкое» представление формы. Поля в этой таблице будут размечены по столбцам в соответствии с ранее рассмотренным описанием.

Добавьте в элемент управления `Grid` по имени `customersColumnarView` следующее определение строк:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="10*"/>
    </Grid.RowDefinitions>
</Grid>
```

Верхняя строка будет использоваться для вывода заголовка, а вторая, более крупная, — для вывода элементов управления, в которые пользователь будет вводить данные.

Добавьте сразу же после определения строк элемент управления `TextBlock`, показанный далее жирным шрифтом. Этот элемент показывает в первой строке Grid-элемента усеченный заголовок `Customers`. Установите для свойства `FontSize` значение `30`:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
              Text="Customers" VerticalAlignment="Center" FontSize="30"/>
</Grid>
```

Добавьте к строке 1 Grid-элемента по имени `customersColumnarView` сразу же после элемента управления `TextBlock`, содержащего заголовок `Customers`, еще один элемент управления `Grid`. Этот Grid-элемент будет в двух столбцах отображать надписи и элементы управления для ввода данных, поэтому добавьте к нему определения строк и столбцов, показанные в следующем примере кода жирным шрифтом:

```
<TextBlock Grid.Row="0" ... />
<Grid Grid.Row="1">
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
</Grid>
```

Обратите внимание на то, что если все строки или столбцы в наборе имеют одинаковую высоту или ширину, указывать их размеры не нужно.

После только что добавленного определения строк скопируйте в новый Grid-элемент XAML-разметку элементов управления типа `TextBlock` с названиями `ID`, `Title`, `First Name` и `Last Name` из Grid-элемента по имени `customersTabularView`. Поместите элемент управления для `ID` в строку 0, элемент управления для `Title` — в строку 1, элемент управления для `First Name` — в строку 2 и элемент управления для `Last Name` — в строку 3. Поместите все элементы управления в столбец 0:

```
<Grid.RowDefinitions>
  ...
</Grid.RowDefinitions>
<TextBlock Grid.Row="0" Grid.Column="0" HorizontalAlignment="Center"
  TextWrapping="Wrap" Text="ID" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="1" Grid.Column="0" HorizontalAlignment="Center"
  TextWrapping="Wrap" Text="Title" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="2" Grid.Column="0" HorizontalAlignment="Center"
  TextWrapping="Wrap" Text="First Name" VerticalAlignment="Center"
  FontSize="20"/>
<TextBlock Grid.Row="3" Grid.Column="0" HorizontalAlignment="Center"
  TextWrapping="Wrap" Text="Last Name" VerticalAlignment="Center"
  FontSize="20"/>
```

Сразу же после элементов управления `TextBox` скопируйте в новый Grid-элемент XAML-разметку для элементов управления типа `ComboBox` и `TextBox` с именами

`id`, `title`, `firstName` и `lastName` из `Grid`-элемента по имени `customersTabularView`. Поместите элемент `id` в строку 0, элемент `title` — в строку 1, элемент `firstName` — в строку 2 и элемент `lastName` — в строку 3. Поместите все четыре элемента управления в столбец 1. Также измените имена элементов управления, поставив префикс в виде буквы `c` (от слова `column`, означающего разметку по столбцам). Последнее изменение необходимо, чтобы избежать совпадения с именами существующих элементов управления в `Grid`-элементе по имени `customersTabularView`:

```
<TextBlock Grid.Row="3" Grid.Column="0" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Last Name" .../>
<TextBox Grid.Row="0" Grid.Column="1" x:Name="cId"
    HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
    VerticalAlignment="Center" FontSize="20" IsReadOnly="True"/>
<TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName"
    HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
    VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName"
    HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
    VerticalAlignment="Center" FontSize="20"/>
<ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Center" FontSize="20">
    <ComboBoxItem Content="Mr"/>
    <ComboBoxItem Content="Mrs"/>
    <ComboBoxItem Content="Ms"/>
    <ComboBoxItem Content="Miss"/>

```

`</ComboBox>`

Скопируйте в новый `Grid`-элемент элементы управления типа `TextBlock` и `TextBox` для адреса электронной почты (`email`) и номера телефона из `Grid`-элемента по имени `customersTabularView`, поместив их после элемента управления типа `ComboBox` по имени `cTitle`. Поместите элементы управления типа `TextBlock` в столбец 0, в строки 4 и 5, а элементы управления типа `TextBox` — в столбец 1, в строки 4 и 5. Измените имя элемента `email` типа `TextBox` на `cEmail` и имя элемента `phone` типа `TextBox` — на `cPhone`. Удалите свойства `Width` у элементов `cEmail` и `cPhone` и установите для их свойств `HorizontalAlignment` значения `Stretch`:

```
<ComboBox ...>
    ...
</ComboBox>
<TextBlock Grid.Row="4" Grid.Column="0" HorizontalAlignment="Center"
    TextWrapping="Wrap" Text="Email" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail"
    HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
    VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="5" Grid.Column="0" HorizontalAlignment="Center"
    TextWrapping="Wrap" Text="Phone" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone"
    HorizontalAlignment="Stretch" TextWrapping="Wrap" Text=""
    VerticalAlignment="Center" FontSize="20"/>
```

В окне конструктора должна отобразиться примерно такая столбцовая разметка (рис. 25.19).

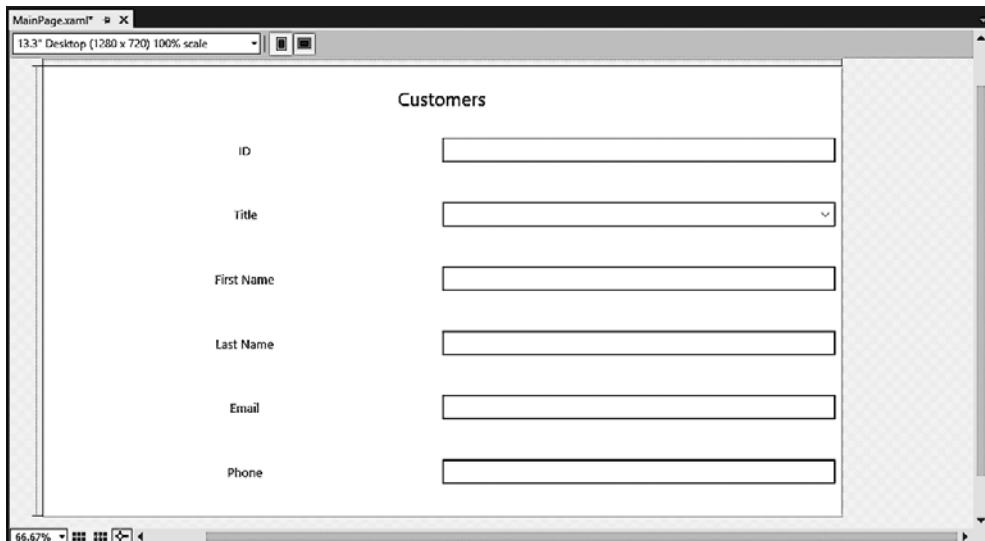


Рис. 25.19

Вернитесь к XAML-разметке для Grid-элемента по имени `customersTabularView` и установите для свойства `Visibility` значение `Visible`:

```
<Grid x:Name="customersTabularView" Margin="10,20,10,20" Visibility="Visible">
```

Для свойства `Visibility` в XAML-разметке Grid-элемента по имени `customersColumnarView` установите значение `Collapsed`:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Collapsed">
```

В окне конструктора должна отобразиться исходная табличная разметка формы `Customers`. Это исходное представление, которое будет использоваться приложением.

Теперь вы определили разметку, которая будет отображаться при узкой области просмотра. То, что все сделанное вами по сути является дублированием множества элементов управления и разметкой их в другой манере, не может не вызывать беспокойства. Если запустить форму и переключиться между представлениями, то как данные из одного представления будут перемещаться в другое представление? К примеру, если сведения о клиенте вводятся, когда приложение работает в полноэкранном режиме, а затем происходит переключение в режим узкой области просмотра, то заново отображаемые элементы управления не будут содержать те же самые данные, которые только что были

введены. В UWP-приложениях эта проблема решается с помощью привязки данных. Это технология, с помощью которой можно связать один и тот же фрагмент данных с несколькими элементами управления, так что по мере изменения данных все элементы управления будут показывать обновленную информацию. Работа этой технологии будет показана в главе 26. А пока вы будете заниматься только приемами использования Диспетчера визуальных состояний для переключения между разметками при изменениях области просмотра.

Когда какое-нибудь свойство дисплея (например, высота или ширина) изменяется, вы можете воспользоваться инициаторами, подающими сигнал Диспетчеру визуальных состояний. Переходы визуального состояния, выполняемые этими инициаторами, можно определить в XAML-разметке вашего приложения. Именно этим вы и займитесь в следующем упражнении.

Использование Диспетчера визуальных состояний для изменения разметки

Добавьте в панели XAML приложения `Customers` после закрывающего тега `</Grid>`, принадлежащего Grid-элементу по имени `customersColumnarView`, следующую разметку:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    ...
</Grid>
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout">
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Переходы визуального состояния определяются путем реализации одной или нескольких групп визуального состояния. Каждая группа визуального состояния определяет переходы, которые должны произойти, когда Диспетчер визуальных состояний переключится в это состояние. Каждому состоянию должно быть дано значимое имя, помогающее определить его предназначение.

Добавьте к группе визуального состояния следующий инициатор (trigger), показанный жирным шрифтом:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="660"/>
            </VisualState.StateTriggers>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Этот инициатор будет срабатывать, как только ширина окна станет меньше 660 пикселов. При этой ширине в форме *Customers* начинается перенос элементов управления и надписей на новую строку и с ними становится трудно работать.

Добавьте к XAML-разметке после определения инициатора следующий код, выделенный жирным шрифтом:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="660"/>
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter Target="customersTabularView.Visibility"
                    Value="Visible"/>
                <Setter Target="customersColumnarView.Visibility"
                    Value="Collapsed"/>
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Этот код определяет действия, производимые при срабатывании инициатора. В данном примере действия определяются с помощью элементов-установщиков *Setter*. Элемент *Setter* определяет устанавливаемое свойство и значение, которое для него должно быть установлено. Для данного представления команды *Setter* изменяют значения указанных свойств: Grid-элемент *customersTabularView* становится видимым, а Grid-элемент *customersColumnarView* скрывается (стается невидимым).

Добавьте после определения визуального состояния *TabularLayout* следующую разметку, показанную жирным шрифтом, которая определяет эквивалентные функциональные возможности для узкой области просмотра:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout">
            ...
        </VisualState>
        <VisualState x:Name="ColumnarLayout">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="0"/>
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter Target="customersTabularView.Visibility"
                    Value="Collapsed"/>
                <Setter Target="customersColumnarView.Visibility"
                    Value="Visible"/>
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Переход будет осуществляться, когда ширина окна станет меньше 660 пикселов. Приложение переключится в состояние `ColumnarLayout`: `Grid`-элемент `customersTabularView` скроется, а `Grid`-элемент `customersColumnarView` станет видимым.

В меню Отладка щелкните на пункте Начать отладку. Приложение запустится и отобразит полноэкранную форму `Customers`. Данные будут отображены с применением табличной разметки.



ПРИМЕЧАНИЕ Если вы используете дисплей с разрешением менее чем 1366 × 768, запускайте приложение в ранее рассмотренном симуляторе. Настройте симулятор на разрешение 1366 × 768.

Измените размер окна приложения `Customer` для отображения формы в узкой области просмотра. Когда ширина окна станет меньше 660 пикселов, отображение переключится на столбцовую разметку.

Измените размер окна приложения `Customers`, сделав его шире 660 пикселов (или переключите его в полноэкранный режим). Форма `Customers` опять перейдет к табличной разметке.

Вернитесь в среду Visual Studio и остановите отладку.

Применение стилей к пользовательскому интерфейсу

Следующим шагом после того, как будет решен вопрос с механикой основной разметки приложения, станет применение к пользовательскому интерфейсу стилей для добавления ему привлекательности. У элементов управления UWP-приложения имеются самые разнообразные свойства, которыми можно воспользоваться для изменения таких характеристик, как шрифт, цвет, размер, а также других атрибутов элемента. Значения этих свойств можно устанавливать отдельно для каждого элемента, но если одно и то же стилевое оформление нужно применить к нескольким элементам управления, такой подход может привести к утомительным и однообразным действиям. Кроме того, в лучших приложениях для пользовательского интерфейса выдерживается единство стиля, а при повторяющихся установках значений для одних и тех же свойств по мере добавления или удаления элементов управления сохранить его бывает нелегко. Зачастую приходится совершать однообразные действия, и при этом весьма высока вероятность хотя бы раз сделать что-нибудь не так!

Создавая UWP-приложения, можно определять многократно используемые стили. Их нужно реализовывать в виде общедоступных во всем приложении ресурсов, создав словарь ресурсов, и тогда они станут доступны всем элементам управления на всех страницах приложения. Можно также воспользоваться

локальными ресурсами, применимыми только к одной странице, определив их в XAML-разметке для этой страницы. В следующем упражнении вы определите несколько простых стилей для приложения *Customers*, после чего они будут применены к элементам управления формы *Customers*.

Определение стилей для формы *Customers*

В обозревателе решений щелкните правой кнопкой мыши на проекте *Customers*, укажите на пункт **Добавить** и щелкните на пункте **Создать элемент**.

В диалоговом окне **Добавить новый элемент** — *Customers* щелкните на пункте **Словарь ресурсов**. Наберите в поле **Имя** строку *AppStyles.xaml*, а затем щелкните на кнопке **Добавить**.

В окне редактора появится файл *AppStyles.xaml*. Словарь ресурсов является XAML-файлом, содержащим ресурсы, которыми может воспользоваться приложение. Файл *AppStyles.xaml* выглядит следующим образом:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Customers">

</ResourceDictionary>
```

Стили являются одним из примеров ресурса, но вы можете также добавлять другие элементы. Фактически, первым ресурсом, который вы добавите, будет не стиль, а элемент **ImageBrush** (кисть), используемый для рисования фона самого внешнего элемента управления **Grid**, имеющегося в форме *Customers*.

В обозревателе решений щелкните правой кнопкой мыши на проекте *Customers*, укажите на пункт **Добавить**, а затем щелкните на пункте **Создать папку**. Измените имя новой папки на *Images*. Щелкните правой кнопкой мыши на папке *Images*, укажите на пункт **Добавить** и щелкните на пункте **Существующий элемент**.

В диалоговом окне **Добавление существующего элемента** — *Customers* перейдите в папку *\Microsoft Press\VCSBS\Chapter 25\Resources folder* вашей папки документов, щелкните на файле *wood.jpg*, а затем на пункте **Добавить**.

Файл *wood.jpg* добавляется к папке *Images* проекта *Customers*. Этот файл содержит изображение стильного фона, похожего на срез ствола дерева, который будет использован для формы *Customers*.

Выполните в окне редактора файл *AppStyles.xaml* и добавьте к нему следующую XAML-разметку, показанную жирным шрифтом:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
xmlns:local="using:Customers"

<ImageBrush x:Key="WoodBrush" ImageSource="Images/wood.jpg"/>
</ResourceDictionary>
```

Эта разметка создает ресурс типа `ImageBrush` по имени `WoodBrush`, основанный на файле `wood.jpg`. Этот ресурс можно использовать для установки фона элемента управления, и он выведет на экран изображение, находящееся в файле `wood.jpg`.

Добавьте к файлу `AppStyles.xaml` ниже ресурса `ImageBrush` следующий стиль, показанный жирным шрифтом:

```
<ResourceDictionary
    ...>

    <ImageBrush x:Key="WoodBrush" ImageSource="Images/wood.jpg"/>
    <Style x:Key="GridStyle" TargetType="Grid">
        <Setter Property="Background" Value="{StaticResource WoodBrush}"/>
    </Style>
</ResourceDictionary>
```

Эта разметка показывает, как определить стиль. У элемента `Style` должно быть имя (ключ, позволяющий ссылаться на него из любого места приложения), и в нем должен определяться тип элемента управления, к которому может применяться стиль. Вы будете использовать этот стиль с элементом управления `Grid`.

Тело стиля состоит из одного или нескольких элементов `Setter`. В данном примере для свойства фона `Background` устанавливается значение ресурса `WoodBrush`, относящегося к типу `ImageBrush`. Но синтаксис выглядит немного странно. При указании значения свойства можно либо сослаться на одно из подходящих значений из тех, что определяются системой (например, `Red`, если нужно установить для фона насыщенный красный цвет), либо указать ресурс, который был где-либо определен. Для ссылки на определенный в каком-то месте ресурс используется ключевое слово `StaticResource`, а все выражение заключается в фигурные скобки.

Перед тем как получить возможность использовать этот стиль, требуется обновить словарь глобальных ресурсов для приложения, который находится в файле `App.xaml`, добавив ссылку на файл `AppStyles.xaml`. В обозревателе решений дважды щелкните на файле `App.xaml`, чтобы вывести его в окно редактора. Файл `App.xaml` имеет следующий вид:

```
<Application
    x:Class="Customers.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Customers"
    RequestedTheme="Light">

</Application>
```

На данный момент в файле App.xaml определяется только объект приложения, и в область видимости вводятся несколько пространств имен, а словарь глобальных ресурсов пока что пуст.

Добавьте к файлу App.xaml код, выделенный жирным шрифтом:

```
<Application
    x:Class="Customers.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Customers"
    RequestedTheme="Light">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="AppStyles.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

Эта разметка добавляет к списку ресурсов, доступных в глобальном словаре ресурсов, те ресурсы, которые определены в файле AppStyles.xaml. Теперь они доступны для использования по всему приложению.

Теперь переключитесь на файл MainPage.xaml для отображения пользовательского интерфейса, определенного в форме Customers. Найдите на панели XAML самый внешний элемент управления **Grid**:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

В XAML-разметке этого элемента управления замените свойство **Background** выделенным здесь жирным шрифтом свойством **Style**, которое ссылается на стиль **GridStyle**:

```
<Grid Style="{StaticResource GridStyle}">
```

Фон элемента управления **Grid** в окне конструктора должен переключиться и приобрести вид деревянной панели (рис. 25.20).



ПРИМЕЧАНИЕ В идеале нужно добиться того, чтобы любое фоновое изображение, применяемое к странице или элементу управления, сохраняло эстетическую привлекательность при изменении форм-фактора или ориентации устройства. Изображение, которое хорошо смотрится на 30-дюймовом мониторе, на смартфоне Windows может быть искажено и сжато. Возможно, для разных представлений и ориентации придется предоставлять альтернативные изображения и использовать Диспетчер визуальных состояний для внесения изменений в свойство **Background** элемента управления с целью переключения с одного изображения на другое по мере изменения визуального состояния.

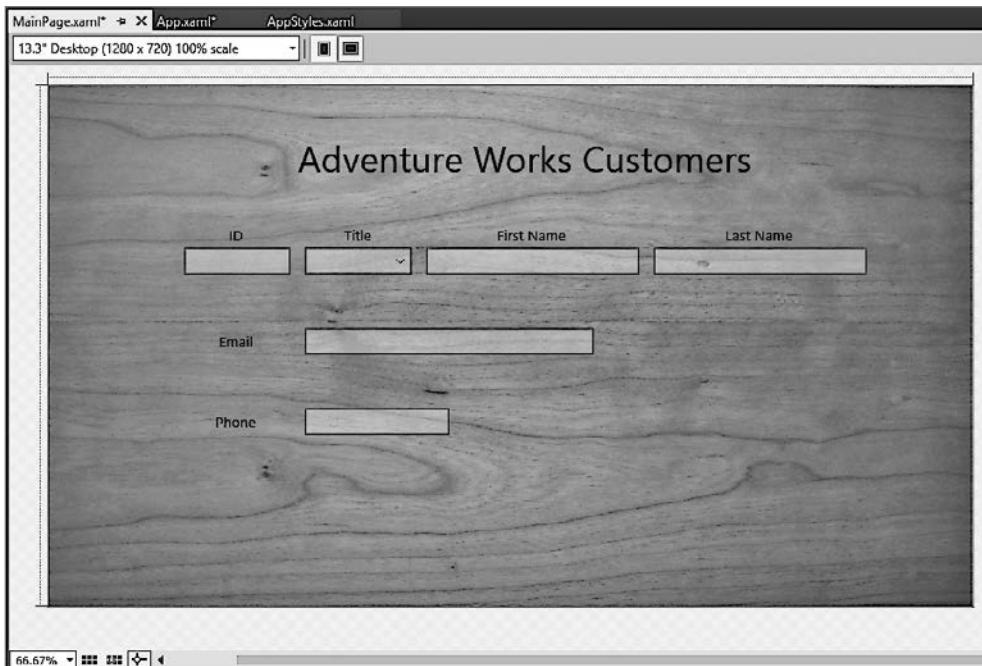


Рис. 25.20

Вернитесь в окне редактора к файлу AppStyles.xaml и добавьте после стиля `GridStyle` стиль `FontStyle`:

```
<Style x:Key="GridStyle" TargetType="Grid">
    ...
</Style>
<Style x:Key="FontStyle" TargetType="TextBlock">
    <Setter Property="FontFamily" Value="Segoe Print"/>
</Style>
```

Этот стиль применяется к элементам типа `TextBlock` и изменяет шрифт на `Segoe Print`. Данные, выведенные этим шрифтом, напоминают рукописный текст.

На данном этапе вполне возможно дать ссылку на стиль `FontStyle` в каждом элементе управления `TextBlock`, которому требуется этот шрифт, но такой подход не даст никаких преимуществ над простой непосредственной установкой `FontFamily` в разметке для каждого элемента управления. Реальная эффективность стилей проявляется при сочетании сразу нескольких свойств, в чем вы убедитесь, выполняя следующие действия.

Добавьте к файлу AppStyles.xaml выделенный жирным шрифтом стиль `HeaderStyle`:

```
<Style x:Key="FontStyle"
TargetType="TextBlock">
    ...
</Style>
<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource
    FontStyle}">
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
    <Setter Property="Foreground" Value="SteelBlue"/>
</Style>
```

Это составной стиль, устанавливающий для элементов управления типа `TextBlock` свойства `HorizontalAlignment`, `TextWrapping`, `VerticalAlignment` и `Foreground`. Кроме того, с помощью свойства `BasedOn` стиль `HeaderStyle` ссылается на стиль `FontStyle`. Свойство `BasedOn` предоставляет для стилей простую форму наследования.

Этот стиль будет использоваться для форматирования надписей, появляющихся в верхней части элементов управления `customersTabularView` и `customersColumnarView`. Но у этих заголовков разные размеры шрифта (заголовок для табличной разметки больше заголовка для столбцовой разметки), поэтому вы создадите еще два стиля, расширяющих стиль `HeaderStyle`.

Добавьте к файлу `AppStyles.xaml` следующие стили:

```
<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource
    FontStyle}">
    ...
</Style>
<Style x:Key="TabularHeaderStyle" TargetType="TextBlock"
BasedOn="{StaticResource HeaderStyle}">
    <Setter Property="FontSize" Value="40"/>
</Style>

<Style x:Key="ColumnarHeaderStyle" TargetType="TextBlock"
BasedOn="{StaticResource HeaderStyle}">
    <Setter Property="FontSize" Value="30"/>
</Style>
```

Заметьте, что размеры шрифта для этих стилей немного меньше размеров шрифта, используемых для заголовков в элементах управления `Grid` на данный момент. Дело в том, что шрифт `Segoe Print` крупнее шрифта, используемого по умолчанию.

Вернитесь к файлу `MainPage.xaml` и найдите XAML-разметку для элемента управления типа `TextBlock` с надписью `Adventure Works Customers` в `Grid`-элементе `customersTabularView`:

```
<TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Adventure Works Customers" VerticalAlignment="Center" FontSize="50"/>
```

Измените свойства этого элемента управления, указав в них ссылку на стиль **TabularHeaderStyle**, выделенную жирным шрифтом:

```
<TextBlock Grid.Row="0" Style="{StaticResource TabularHeaderStyle}"  
Text="Adventure Works Customers"/>
```

У заголовка, отображаемого в окне конструктора, должны измениться цвет, размер и шрифт, и он должен приобрести следующий вид (рис. 25.21).

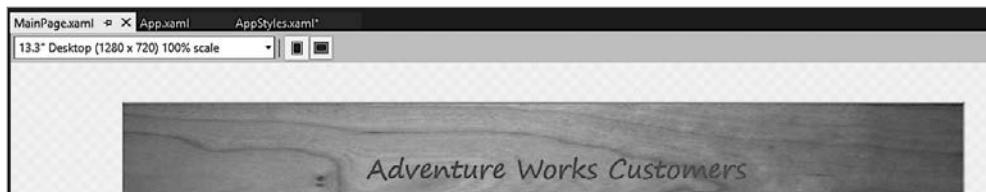


Рис. 25.21

Найдите в Grid-элементе **customersColumnarView** XAML-разметку для элемента управления типа **TextBlock** с надписью *Customers*:

```
<TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"  
Text="Customers" VerticalAlignment="Center" FontSize="30"/>
```

Измените разметку этого элемента управления, указав в ней ссылку на стиль **ColumnarHeaderStyle**, выделенную жирным шрифтом:

```
<TextBlock Grid.Row="0" Style="{StaticResource ColumnarHeaderStyle}"  
Text="Customers"/>
```

Но в окне конструктора это изменение не отобразится, поскольку по умолчанию Grid-элемент **customersColumnarView** скрыт. Тем не менее результат изменения проявится в этом упражнении при последующем запуске приложения.

Вернитесь в окне редактора к файлу **AppStyles.xaml**. Измените стиль **HeaderStyle**, добавив к нему дополнительные элементы установки свойств **Setter**, выделенные жирным шрифтом:

```
<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource  
FontStyle}">  
    <Setter Property="HorizontalAlignment" Value="Center"/>  
    <Setter Property="TextWrapping" Value="Wrap"/>  
    <Setter Property="VerticalAlignment" Value="Center"/>  
    <Setter Property="Foreground" Value="SteelBlue"/>  
    <Setter Property="RenderTransformOrigin" Value="0.5,0.5"/>  
    <Setter Property="RenderTransform">  
        <Setter.Value>  
            <CompositeTransform Rotation="-5"/>  
        </Setter.Value>  
    </Setter>  
</Style>
```

Эти элементы с помощью преобразования поворачивают текст, отображаемый в заголовке, вокруг его центра на 5°.



ПРИМЕЧАНИЕ В этом примере показано простое преобразование. Применяя свойство `RenderTransform`, можно выполнять множество других разнообразных преобразований элемента, сочетая при этом сразу несколько преобразований. Например, можно переместить элемент по осям `x` и `y`, придать элементу наклон и задать ему новый масштаб.

Следует также отметить, что значением свойства `RenderTransform` является еще одна пара «свойство–значение» (свойство в ней `Rotation`, а значение `-5`). В подобных случаях значение указывается с использованием тега `<Setter.Value>`.

Перейдите к файлу `MainPage.xaml`. Теперь в окне конструктора заголовок должен отображаться как бы немного набекрень (рис. 25.22).

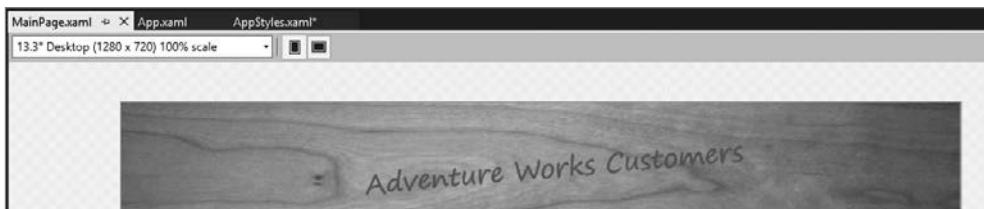


Рис. 25.22

Добавьте к файлу `AppStyles.xaml` следующий стиль:

```
<Style x:Key="LabelStyle" TargetType="TextBlock" BasedOn="{StaticResource
    FontStyle}">
    <Setter Property="FontSize" Value="30"/>
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
    <Setter Property="Foreground" Value="AntiqueWhite"/>
</Style>
```

Этот стиль будет применяться к элементам типа `TextBlock`, предоставляющим надписи для различных элементов типа `TextBox` и `ComboBox`, используемых пользователями для ввода информации о клиентах. В стиле имеется ссылка на тот же самый стиль шрифта, что и для заголовков, но установлены другие, более подходящие для надписей значения свойств.

Вернитесь к файлу `MainPage.xaml`. Измените в панели XAML разметку для элементов управления типа `TextBlock` для каждой надписи в Grid-элементах `customersTabularView` и `customersColumnarView`. Удалите свойства

`HorizontalAlignment`, `TextWrapping`, `VerticalAlignment` и `FontSize` и укажите ссылку на стиль `LabelStyle`, выделенную далее жирным шрифтом:

```
<Grid x:Name="customersTabularView" Margin="10,20,10,20" Visibility="Visible">
    ...
    <Grid Grid.Row="2">
        ...
        <TextBlock Grid.Row="0" Grid.Column="1" Style="{StaticResource
            LabelStyle}" Text="ID"/>
        <TextBlock Grid.Row="0" Grid.Column="3" Style="{StaticResource
            LabelStyle}" Text="Title"/>
        <TextBlock Grid.Row="0" Grid.Column="5" Style="{StaticResource
            LabelStyle}" Text="First Name"/>
        <TextBlock Grid.Row="0" Grid.Column="7" Style="{StaticResource
            LabelStyle}" Text="Last Name"/>
        ...
        <TextBlock Grid.Row="3" Grid.Column="1" Style="{StaticResource
            LabelStyle}" Text="Email"/>
        ...
        <TextBlock Grid.Row="5" Grid.Column="1" Style="{StaticResource
            LabelStyle}" Text="Phone"/>
        ...
    </Grid>
</Grid>
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Collapsed">
    ...
    <Grid Grid.Row="1">
        ...
        <TextBlock Grid.Row="0" Grid.Column="0" Style="{StaticResource
            LabelStyle}" Text="ID"/>
        <TextBlock Grid.Row="1" Grid.Column="0" Style="{StaticResource
            LabelStyle}" Text="Title"/>
        <TextBlock Grid.Row="2" Grid.Column="0" Style="{StaticResource
            LabelStyle}" Text="First Name"/>
        <TextBlock Grid.Row="3" Grid.Column="0" Style="{StaticResource
            LabelStyle}" Text="Last Name"/>
        ...
        <TextBlock Grid.Row="4" Grid.Column="0" Style="{StaticResource
            LabelStyle}" Text="Email"/>
        ...
        <TextBlock Grid.Row="5" Grid.Column="0" Style="{StaticResource
            LabelStyle}" Text="Phone"/>
        ...
    </Grid>
</Grid>
```

Надписи в форме должны измениться: теперь они выведены шрифтом Segoe Print, белого цвета, размер 30 пунктов (рис. 25.23).

Щелкните в меню Отладка на пункте Начать отладку, инициируя сборку и запуск приложения.

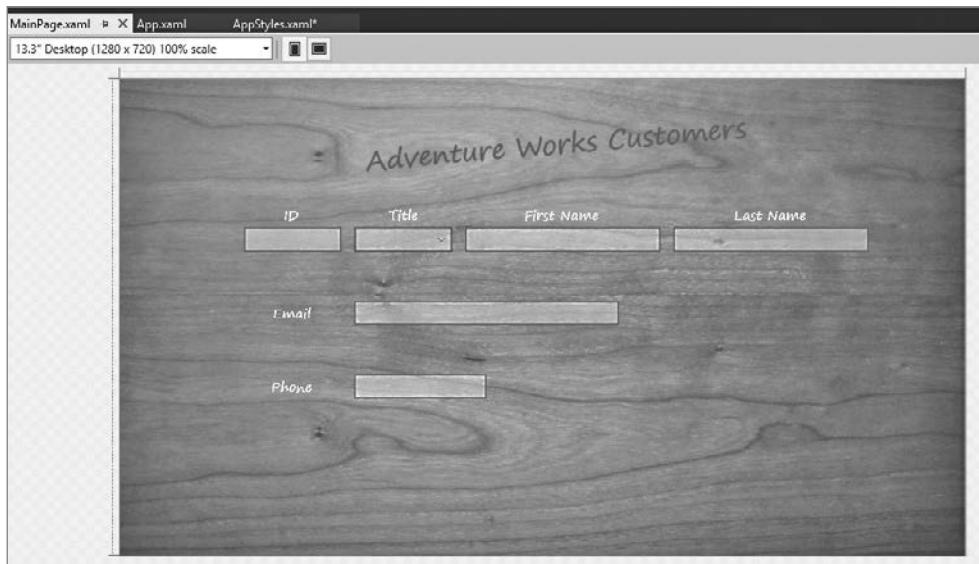


Рис. 25.23



ПРИМЕЧАНИЕ Если запуск приложения осуществляется на дисплее с разрешением менее 1366 × 768, воспользуйтесь симулятором.

Должна появиться форма Customers, имеющая точно такое же стилевое оформление, как и в окне конструктора в среде Visual Studio. Заметьте, что при вводе в различные поля формы любого текста для элементов управления типа TextBox будут применяться шрифт и стиль, используемые по умолчанию.



ПРИМЕЧАНИЕ Шрифт Segoe Print больше подходит для названий и надписей и не рекомендуется в качестве шрифта для полей ввода данных, потому что некоторые символы в нем практически неотличимы друг от друга. Например, буква «l» в нижнем регистре очень похожа на цифру 1, а буква «O» в верхнем регистре мало чем отличается от цифры 0. Поэтому есть смысл оставить для элементов управления типа TextBox шрифт, используемый по умолчанию.

Измените размер окна, уменьшив его ширину, и убедитесь в том, что к элементам управления применяется стилевое оформление, определенное в Grid-элементе customersColumnarView. Форма должна приобрести следующий вид (рис. 25.24).

Вернитесь в среду Visual Studio и остановите отладку.

Чтобы вы могли получить полное представление, на рис. 25.25 показано приложение Customers, запущенное на эмуляторе устройства, работающего под управлением операционной системы Windows Phone 10.

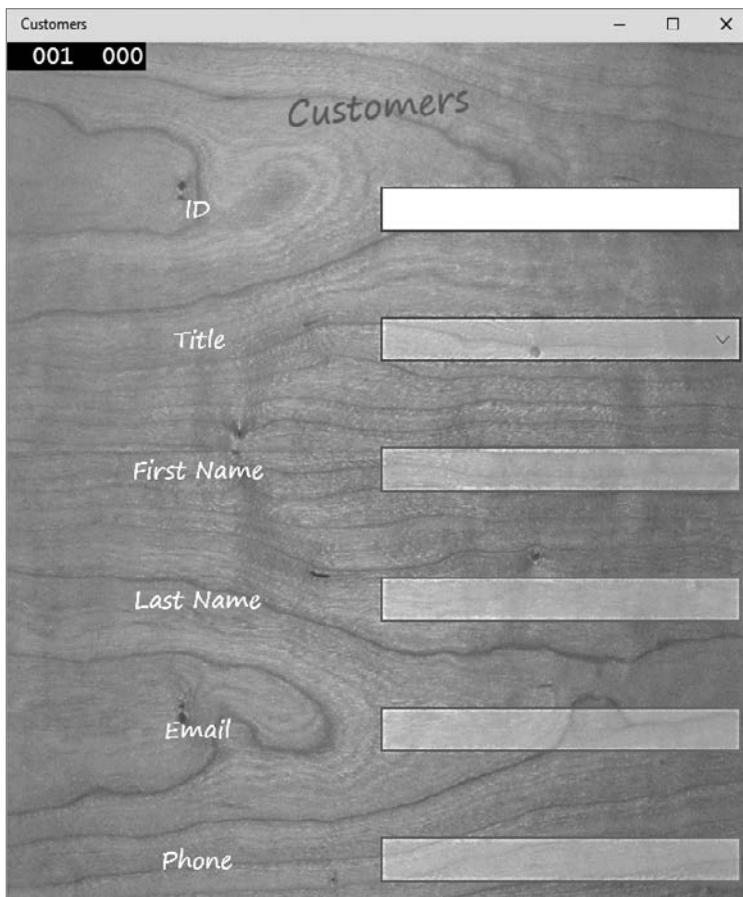


Рис. 25.24

У вас есть возможность убедиться в том, что использование стилей позволяет легко и просто реализовывать ряд весьма впечатляющих эффектов. Кроме того, аккуратное использование стилей облегчает сопровождение кода по сравнению с тем, каким бы оно было, если бы значения для свойств устанавливались в отдельно взятых элементах управления. К примеру, если в приложении *Customers* вам понадобится переключить шрифт, используемый для надписей и заголовков, придется всего лишь внести небольшое изменение в стиль *FontStyle*. В принципе, стили нужно использовать везде, где только можно, поскольку это не только облегчает сопровождение кода, но и помогает поддерживать чистоту и лаконичность XAML-разметки для форм, в которой нужно будет указывать только элементы управления и их расположение, а не то, как именно они должны выглядеть в форме. Для определения сложных стилей, встраиваемых в приложения, можно также воспользоваться средством Microsoft Blend для среды



Рис. 25.25

Visual Studio 2015. Профессиональные художники-графики могут применять Blend для разработки оригинальных стилей и предоставлять их в форме XAML-разметок разработчикам, создающим приложения. Разработчику для ссылки на нужные стили останется всего лишь добавить соответствующие теги *Style* к элементам пользовательского интерфейса.

Выводы

В этой главе вы узнали, как для реализации пользовательского интерфейса используются элементы управления *Grid*, позволяющие выполнять масштабирование под различные форм-факторы и ориентацию устройств. Вы также научились использовать Диспетчер визуальных состояний для адаптации разметки элементов управления при изменении пользователем размеров окна,

в котором отображается приложение. И наконец, вы узнали, как создаются оригинальные стили и как они применяются к элементам управления формы. Следующей задачей после того, как вы определили пользовательский интерфейс, станет добавление к приложению функциональных средств, позволяющих пользователю выводить на экран данные и изменять их. Ее решению и будут посвящены заключительные главы книги.

Если хотите продолжить работу и изучить следующую главу, оставьте открытой среду Visual Studio 2015 и переходите к главе 26.

Если сейчас вы хотите выйти из среды Visual Studio 2015, то в меню **Файл** щелкните на пункте **Выход**. Увидев диалоговое окно с предложением сохранить изменения, щелкните на кнопке **Да** и сохраните проект.

Краткий справочник

Чтобы	Сделайте следующее
Создать новое UWP-приложение	Воспользуйтесь одним из UWP-шаблонов, имеющихся в среде Visual Studio 2015, например шаблоном пустого приложения
Реализовать пользовательский интерфейс, способный масштабироваться под различные форм-факторы и ориентацию устройств	Воспользуйтесь элементом управления Grid. Разбейте Grid-элемент на строки и столбцы и вместо указания для элементов управления абсолютного местоположения относительно сторон Grid-элемента поместите их в эти строки и столбцы
Реализовать пользовательский интерфейс, способный подстраиваться под различные значения ширины дисплея	Создайте разные разметки для каждого представления, отображающие элементы управления подобающим образом. Для выбора разметки, отображаемой при изменении визуального состояния, воспользуйтесь Диспетчером визуальных состояний
Создать оригинальные стили	Добавьте к приложению словарь ресурсов. Определите в этом словаре стили, воспользовавшись элементом <code><Style></code> , и укажите свойства, изменяемые каждым стилем, например:
	<code><Style x:Key="GridStyle" TargetType="Grid"></code> <code> <Setter Property="Background"</code> <code> Value="{StaticResource WoodBrush}" /></code> <code></Style></code>
Применить к элементу управления оригинальный стиль	Установите для элемента управления свойство <code>Style</code> и сошлитесь на стиль по имени, например:
	<code><Grid Style="{StaticResource GridStyle}"></code>

26

Отображение и поиск данных в приложении универсальной платформы Windows

Прочитав эту главу, вы научитесь:

- объяснять, как шаблон Model — View — ViewModel (модель — представление — модель представления) используется для реализации логики приложения универсальной платформы Windows;
- использовать в представлении привязку данных для их отображения и изменения;
- создавать модель представления (ViewModel), с помощью которой представление сможет взаимодействовать с моделью.

В главе 25 «Реализация пользовательского интерфейса для приложений универсальной платформы Windows» были продемонстрированы способы конструирования пользовательского интерфейса, способного адаптироваться под различные форм-факторы, варианты ориентации и области просмотра устройств, которые может использовать клиент, запускающий ваше приложение. Учебное приложение, разработанное в этой главе, является одним из простейших приложений, сконструированных для отображения и редактирования данных о клиентах.

Далее вы увидите, как отобразить данные в пользовательском интерфейсе, и узнаете о характерных особенностях Windows 10, используя которые можно вести поиск данных в приложении. В ходе выполнения этих задач вы также изучите способы, которыми можно воспользоваться для структурирования UWP-приложения. В этой главе охвачено множество основных вопросов. В частности, вы увидите, как используется привязка данных для связи пользовательского интерфейса с отображаемыми им данными и как создается модель представления (ViewModel), предназначенная для отделения логики пользовательского интерфейса от модели данных и бизнес-логики приложения.

Реализация шаблона Model-View-ViewModel

В графических приложениях, имеющих четко выраженную структуру, конструкция пользовательского интерфейса отделена от данных, используемых приложением, и от бизнес-логики, охватывающей функциональные возможности приложения. Это помогает устраниить зависимости между различными компонентами, позволяет иметь различные представления данных, не испытывая при этом необходимости изменять бизнес-логику илиложенную в основу модель данных. Этот подход также открывает пути конструирования и реализации различных элементов специалистами узкого профиля. Например, художник-график может сконцентрироваться на разработке привлекательного и интуитивно понятного пользовательского интерфейса, специалист по базам данных — сосредоточиться на реализации оптимизированного набора структур данных для хранения данных и доступа к ним, а разработчик, владеющий C#, — направить свои усилия на реализацию бизнес-логики приложения. Эта же общая цель ставится при разработке многих программных средств, не только UWP-приложений, и на протяжении последних нескольких лет для построения структуры приложения именно в таком плане было придумано множество технологических приемов.

Пожалуй, наиболее популярным подходом является следование шаблону разработки Model – View – ViewModel (MVVM). В этом шаблоне модель (Model) предоставляет данные, используемые приложением, а представление (View) задает способ отображения данных в пользовательском интерфейсе. Модель представления (ViewModel) содержит логику, которая связывает два первых компонента, принимая пользовательский ввод и превращая его в команды, выполняющие бизнес-операции в отношении модели, а также забирая данные из модели и форматируя их ожидаемым представлением образом. На следующей схеме (рис. 26.1) показаны упрощенные взаимоотношения между элементами шаблона MVVM. Обратите внимание на то, что приложение может предоставлять несколько представлений одних и тех же данных. Например, в UWP-приложении можно реализовать несколько состояний представлений, способных предоставлять информацию, используя различные разметки экрана. Одной из задач модели представления является обеспечение возможности отображения данных одной и той же модели и работы с ними с использованием множества различных представлений. Для связи с данными, предоставляемыми моделью представления, в представлении UWP-приложения может использоваться привязка данных. Кроме того, используя вызов команд, реализуемых моделью представления, представление может запросить у модели представления обновление данных в модели или выполнение бизнес-задач.

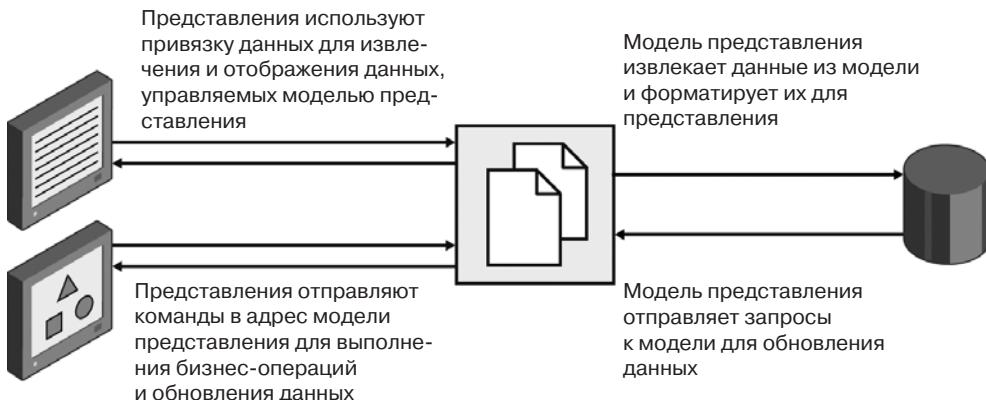


Рис. 26.1

Отображение данных путем использования привязки данных

Перед тем как вы приступите к реализации модели представления для приложения *Customers*, будет полезно более тщательно разобраться в привязке данных и в способе применения этой технологии для отображения данных в пользовательском интерфейсе. Используя привязку данных, можно связать свойство элемента управления со свойством объекта: если значение указанного свойства объекта изменяется, то изменяется и свойство связанного с объектом элемента управления. Кроме того, привязка данных может быть двунаправленной: если изменяется значение свойства в элементе управления, использующем привязку данных, изменение распространяется и на объект, к которому привязан этот элемент управления. В следующем упражнении дается краткое введение в порядок использования привязки данных для отображения данных. Это упражнение основано на приложении *Customers* из главы 25.

Использование привязки данных для отображения информации о клиенте в приложении *Customers*

Откройте в среде Visual Studio 2015 проект *Customers*, который находится в папке `\Microsoft Press\VCSBS\Chapter 26\Data Binding` вашей папки документов. Это версия приложения *Customers*, которая была разработана в главе 25, но разметка пользовательского интерфейса немного изменена — чтобы элементы управления стали заметнее, они отображаются на синем фоне.

В обозревателе решений щелкните правой кнопкой мыши на проекте *Customers*, укажите на пункт Добавить, а затем щелкните на пункте Класс. Убедитесь в том,

Что в диалоговом окне Добавить новый элемент — Customers выбран шаблон Класс. Наберите в поле Имя строку Customer.cs, а затем щелкните на кнопке Добавить.



ПРИМЕЧАНИЕ Синий фон был создан путем использования элемента управления Rectangle (прямоугольник), охватывающего те же строки и столбцы, что и элементы управления TextBlock и TextBox, в которых отображаются заголовки и данные. Прямоугольник закрашивается с помощью кисти LinearGradientBrush, которая постепенно меняет его цвет от синего средней насыщенности вверху до темно-синего внизу. XAML-разметка для элемента управления Rectangle, который отображается в Grid-элементе по имени customersTabularView, выглядит следующим образом (XAML-разметка для Grid-элемента customersColumnarView Grid включает такой же элемент управления Rectangle, охватывающий строки и столбцы, используемые этой разметкой):

```
<Rectangle Grid.Row="0" Grid.RowSpan="6" Grid.Column="1"
           Grid.ColumnSpan="7" ...>
    <Rectangle.Fill>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="#FF0E3895"/>
            <GradientStop Color="#FF141415" Offset="0.929"/>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Этот класс будет использоваться для реализации типа данных Customer и привязки данных к отображению сведений, содержащихся в Customer-объектах, в пользовательском интерфейсе.

Вызовите в окно редактора файл Customer.cs, сделайте класс Customer открытым и добавьте к нему следующие закрытые поля и свойства, выделенные жирным шрифтом:

```
public class Customer
{
    public int _customerID;
    public int CustomerID
    {
        get { return this._customerID; }
        set { this._customerID = value; }
    }

    public string _title;
    public string Title
    {
        get { return this._title; }
        set { this._title = value; }
    }

    public string _firstName;
    public string FirstName
    {
        get { return this._firstName; }
        set { this._firstName = value; }
    }
}
```

```
public string _lastName;
public string LastName
{
    get { return this._lastName; }
    set { this._lastName = value; }
}

public string _emailAddress;
public string EmailAddress
{
    get { return this._emailAddress; }
    set { this._emailAddress = value; }
}

public string _phone;
public string Phone
{
    get { return this._phone; }
    set { this._phone = value; }
}
```

Вы можете удивиться, почему эти свойства не реализованы в качестве автоматических, учитывая то, что они всего лишь извлекают и устанавливают значение в закрытом поле. Дело в том, что в выполняемом далее упражнении к этим свойствам будет добавляться дополнительный код.

В обозревателе решений в проекте *Customers* дважды щелкните на файле *MainPage.xaml*, чтобы в окне конструктора отобразился пользовательский интерфейс приложения. Найдите в XAML-панели разметку для элемента управления типа *TextBox* по имени *id*. Замените XAML-разметку, которая устанавливает свойство *Text* для этого элемента управления, следующей разметкой, показанной далее жирным шрифтом:

```
<TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
Text="{Binding CustomerID}" .../>
```

Синтаксис *Text="{Binding Path}"* указывает на то, что значение свойства *Text* будет предоставлено значением выражения *Path* (путь) в ходе выполнения приложения. В данном случае для выражения *Path* установлено значение *CustomerID*, поэтому этим элементом управления будет отображено значение, хранящееся в выражении *CustomerID*. Но чтобы указать, что *CustomerID* фактически является свойством объекта *Customer*, вам следует предоставить дополнительную информацию. Для этого нужно установить значение для свойства *DataContext* элемента управления, что вам вскоре и предстоит сделать.

Добавьте в форме к каждому второму элементу ввода текста следующие выражения привязки. Примените привязку данных, показанную в следующем коде жирным шрифтом, к элементам управления типа *TextBox* в *Grid*-элементах *customersTabularView* и *customersColumnarView*. (Элементы управления типа

ComboBox требуют несколько другого обращения, которое будет рассмотрено далее в разделе «Использование привязки данных к элементам управления типа ComboBox».)

```
<Grid x:Name="customersTabularView" ...>
    ...
    <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
Text="{Binding CustomerID}" .../>
    ...
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...
Text="{Binding FirstName}" .../>
    <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...
Text="{Binding LastName}" .../>
    ...
    <TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="3"
x:Name="email" ... Text="{Binding EmailAddress}" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3"
x:Name="phone" ... Text="{Binding Phone}" .../>
</Grid>
<Grid x:Name="customersColumnarView" Margin="10,20,10,20"
Visibility="Collapsed">
    ...
    <TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...
Text="{Binding CustomerID}" .../>
    ...
    <TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...
Text="{Binding FirstName}" .../>
    <TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...
Text="{Binding LastName}" .../>
    ...
    <TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...
Text="{Binding EmailAddress}" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...
Text="{Binding Phone}" .../>
</Grid>
```

Обратите внимание на то, что одни и те же выражения привязки могут использоваться с более чем одним элементом управления. Например, на выражение {Binding CustomerID} имеются ссылки в элементах управления типа TextBox с именами id и cId, что заставляет оба элемента управления отображать одни и те же данные.

В обозревателе решений откройте файл MainPage.xaml, а затем двойным щелчком откройте в окне редактора файл MainPage.xaml.cs с кодом для формы MainPage.xaml. Добавьте в конструктор MainPage инструкцию, выделенную далее жирным шрифтом.

```
public MainPage()
{
    this.InitializeComponent();
```

```
Customer customer = new Customer
{
    CustomerID = 1,
    Title = "Mr",
    FirstName = "John",
    LastName = "Sharp",
    EmailAddress = "john@contoso.com",
    Phone = "111-1111"
};
}
```

Этот код создает новый экземпляр класса `Customer` и наполняет его данными, взятыми для примера. После кода, создающего новый объект `Customer`, добавьте следующую инструкцию, выделенную жирным шрифтом:

```
Customer customer = new Customer
{
    ...
};

this.DataContext = customer;
```

Эта инструкция указывает объект, к которому должны быть привязаны элементы управления формы `MainPage`. XAML-разметка `Text="{Binding Path}"` в каждом элементе управления будет разрешена в отношении этого объекта. Например, как для элемента управления типа `TextBox` с именем `id`, так и для элемента управления типа `TextBox` с именем `cId` указывается атрибут `Text="{Binding CustomerID}"`, следовательно, они будут отображать значение, найденное в свойстве `CustomerID` объекта типа `Customer`, к которому привязана форма.



ПРИМЕЧАНИЕ В данном примере вами было установлено значение для имеющегося у формы свойства `DataContext`, поэтому одна и та же привязка данных автоматически применяется ко всем элементам управления формы. Если нужно привязать конкретные элементы управления к разным объектам, можно также установить значения для свойств `DataContext`, принадлежащих отдельным элементам управления.

В меню Отладка щелкните на пункте Начать отладку, чтобы выполнить сборку и запуск приложения. Убедитесь в том, что форма заняла весь экран и, как показано на рис. 26.2, отобразила данные клиента John Sharp.

Измените размер окна, чтобы приложение отобразилось в более узкой области просмотра. Убедитесь в том, что отображаются те же данные, которые показаны на рис. 26.3.

Элементы управления, отображаемые в узкой области просмотра, привязаны к тем же данным, что и элементы управления, отображаемые в полноэкранной области просмотра.

Измените в узкой области просмотра адрес электронной почты (Email) на `john@treyresearch.com`. Расширьте окно приложения, чтобы произошло переключение



Рис. 26.2

к представлению в широкой области просмотра. Заметьте, что адрес электронной почты, отображаемый в этом представлении, не изменился.

Вернитесь в среду Visual Studio и остановите отладку.

Вызовите в окно редактора среды Visual Studio код класса `Customer` и установите контрольную точку на методе доступа `set` для свойства `EmailAddress`.

Щелкните в меню Отладка на пункте Начать отладку для повторной сборки и запуска приложения.

Когда отладчик дойдет до контрольной точки первый раз, нажмите клавишу `F5`, чтобы продолжить выполнение приложения.

Когда появится пользовательский интерфейс для приложения `Customers`, измените размер окна приложения, чтобы отобразилось представление для узкой области просмотра, и измените адрес электронной почты на `john@treyresearch.com`.

Снова увеличьте окно приложения, чтобы вернуться к представлению для широкой области просмотра.

Заметьте, что отладчик не дойдет до контрольной точки на строке с методом доступа `set` для свойства `EmailAddress`: когда элемент управления типа `TextBox` по имени `email` теряет фокус, обновленное значение не записывается обратно в объект типа `Customer`.



Рис. 26.3

Вернитесь в среду Visual Studio и остановите отладку.

Удалите контрольную точку из строки с методом доступа `set` свойства `EmailAddress` в классе `Customer`.

Изменение данных путем использования привязки данных

В предыдущем упражнении было показано, насколько просто за счет использования привязки данных, имеющихся в объекте, эти данные можно отобразить в форме приложения. Но привязка данных по умолчанию является односторонней операцией, и любые изменения, вносимые в отображаемые данные,

обратно в источник данных не копируются. В упражнении это было заметно при изменении адреса электронной почты в представлении для узкой области просмотра: когда произошло переключение на представление для широкой области просмотра, данные там не изменились. Реализовать двунаправленную привязку данных можно изменением в XAML-разметке элемента управления параметра **Mode**, принадлежащего спецификации **Binding**. Этот параметр показывает, какой именно является привязка данных, одно- или двунаправленной. Его изменением вы сейчас и займитесь.

Реализация двунаправленной привязки данных для изменения информации о клиенте

Выполните в окне редактора файл MainPage.xaml и измените, как показано в следующем примере кода жирным шрифтом, XAML-разметку для каждого элемента управления типа **TextBox**:

```
<Grid x:Name="customersTabularView" ...>
    ...
    <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
Text="{Binding CustomerID, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...
Text="{Binding FirstName, Mode=TwoWay}" .../>
    <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...
Text="{Binding LastName, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="3"
x:Name="email" ... Text="{Binding EmailAddress, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3"
x:Name="phone" ... Text="{Binding Phone, Mode=TwoWay}" .../>
</Grid>
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" ...>
    ...
    <TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...
Text="{Binding CustomerID, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...
Text="{Binding FirstName, Mode=TwoWay}" .../>
    <TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...
Text="{Binding LastName, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...
Text="{Binding EmailAddress, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...
Text="{Binding Phone, Mode=TwoWay}" .../>
</Grid>
```

Параметр **Mode** в спецификации **Binding** показывает, какой является привязка данных, односторонней (эта установка используется по умолчанию) или

дву направленной. Установка для параметра Mode значения TwoWay заставляет любые изменения, произведенные пользователем, передавать назад объекту, к которому привязан элемент управления.

В меню Отладка щелкните на пункте Начать отладку, чтобы снова выполнить сборку и запуск приложения. При используемом для приложения представлении, предназначенном для широкой области просмотра, измените адрес электронной почты на `john@treyresearch.com`, а затем измените размер окна, чтобы для отображения приложения использовалось представление, предназначенное для узкой области просмотра.

Обратите внимание на то, что, несмотря на изменения привязки данных на режим TwoWay, адрес электронной почты, отображаемый в представлении для узкой области просмотра, не обновился и остался прежним — `john@contoso.com`.

Вернитесь в среду Visual Studio и остановите отладку.

Очевидно, что-то не сработало! Теперь проблема не в том, что не обновились данные, а в том, что представление не отобразило самую последнюю версию данных. (Если в классе `Customer` восстановить контрольную точку на методе доступа `set` свойства `EmailAddress` и запустить приложение в отладчике, вы увидите, что отладчик дойдет до контрольной точки при изменении значения адреса электронной почты и перемещении фокуса с элемента управления типа `TextBox`.) Несмотря на внешние признаки, процесс привязки данных не волшебный и привязка данных не получает информации об изменении этих данных. Объект должен проинформировать привязку данных о любых изменениях отправкой пользовательскому интерфейсу события `PropertyChanged`. Это событие является частью интерфейса `INotifyPropertyChanged`, и все объекты, поддерживающие двунаправленную привязку данных, должны иметь реализацию этого интерфейса. Его реализацией вы займетесь в следующем упражнении.

Реализация интерфейса `INotifyPropertyChanged` в классе `Customer`

Выполните в окне редактора среды Visual Studio файл `Customer.cs`.

Добавьте к списку в начале файла следующую директиву `using`:

```
using System.ComponentModel;
```

В этом пространстве имен определен интерфейс `INotifyPropertyChanged`.

Измените определение класса `Customer`, чтобы указать на реализацию этим классом интерфейса `INotifyPropertyChanged` (изменение показано жирным шрифтом):

```
class Customer : INotifyPropertyChanged
```

Добавьте после свойства Phone в классе Customer событие PropertyChanged, выделенное в следующем примере жирным шрифтом:

```
class Customer : INotifyPropertyChanged
{
    ...
    public string _phone;
    public string Phone {
        get { return this._phone; }
        set { this._phone = value; }
    }
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Это событие является единственным элементом, определяемым интерфейсом `INotifyPropertyChanged`. Все объекты, реализующие этот интерфейс, должны предоставлять это событие и инициировать его, как только им потребуется уведомить внешний мир об изменении значения свойства.

Добавьте к классу `Customer` после события `PropertyChanged` метод `OnPropertyChanged`, показанный далее жирным шрифтом:

```
class Customer : INotifyPropertyChanged
{
    ...
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Метод `OnPropertyChanged` инициирует событие `PropertyChanged`. Параметр `PropertyChangedEventArgs` события `PropertyChanged` должен указывать имя измененного свойства. Это значение передается в качестве параметра методу `OnPropertyChanged`.

Измените в классе `Customer` методы доступа к свойству `set` для каждого свойства, чтобы в них вызывался метод `OnPropertyChanged`, как только содержащееся в них значение подвергнется изменению (все изменения выделены далее жирным шрифтом):

```
class Customer : INotifyPropertyChanged
{
    public int _customerID;
    public int CustomerID
```

```
{  
    get { return this._customerID; }  
    set  
    {  
        this._customerID = value;  
        this.OnPropertyChanged(nameof(CustomerID));  
    }  
}  
  
public string _title;  
public string Title  
{  
    get { return this._title; }  
    set  
    {  
        this._title = value;  
        this.OnPropertyChanged(nameof(Title));  
    }  
}  
  
public string _firstName;  
public string FirstName  
{  
    get { return this._firstName; }  
    set  
    {  
        this._firstName = value;  
        this.OnPropertyChanged(nameof(FirstName));  
    }  
}  
  
public string _lastName;  
public string LastName  
{  
    get { return this._lastName; }  
    set  
    {  
        this._lastName = value;  
        this.OnPropertyChanged(nameof(LastName));  
    }  
}  
  
public string _emailAddress;  
public string EmailAddress  
{  
    get { return this._emailAddress; }  
    set  
    {  
        this._emailAddress = value;  
        this.OnPropertyChanged(nameof(EmailAddress));  
    }  
}  
  
public string _phone;  
public string Phone  
{
```

```

        get { return this._phone; }
        set
        {
            this._phone = value;
            this.OnPropertyChanged(nameof(Phone));
        }
    }
    ...
}

```

ОПЕРАТОР NAMEOF

Оператор nameof, показанный в классе Customer, является редко используемым, но весьма полезным в таком коде, как этот, средством C#. Он возвращает имя переменной, переданной ему в качестве параметра, в виде строки. Без использования оператора nameof вам пришлось бы воспользоваться точными указаниями строковых значений, например:

```

public int CustomerID
{
    get { return this._customerID; }
    set
    {
        this._customerID = value;
        this.OnPropertyChanged("CustomerID");
    }
}

```

Хотя использование строковых значений позволяет набирать меньше текста, подумайте о том, что произойдет, если когда-либо в будущем вам потребуется изменить имя переменной. При использовании подхода, предполагающего использование строкового значения, вам придется вносить изменения еще и в это строковое значение. Если этого не сделать, код будет по-прежнему проходить компиляцию и запускаться, но любые изменения, внесенные в значение свойства в ходе выполнения приложения, не будут регистрироваться, что приведет к возникновению трудно отслеживаемых ошибок. Если при использовании оператора nameof вы измените имя свойства, но забудете изменить аргумент для оператора nameof, код не пройдет компиляцию и вы тут же получите предупреждение о наличии ошибки, которую можно будет легко и быстро устраниТЬ.

В меню Отладка щелкните на пункте Начать отладку, чтобы еще раз выполнить сборку и запуск приложения.

Когда появится форма Customers, измените адрес электронной почты на john@treyresearch.com, а номер телефона — на 222-2222.

Измените размер окна, чтобы приложение отображалось в представлении для узкой области просмотра, и убедитесь в том, что адрес электронной почты и номер телефона изменились.

Измените имя на `James`, увеличьте размер окна, чтобы приложение отображалось в представлении для широкой области просмотра, и убедитесь в том, что имя изменилось.

Вернитесь в среду Visual Studio и остановите отладку.

Использование привязки данных к элементам управления типа ComboBox

Использование привязки данных к элементам управления типа `TextBox` или `TextBlock` особых вопросов не вызывает. А вот элементам управления типа `ComboBox` требуется уделить немного больше внимания. Дело в том, что элемент управления типа `ComboBox` фактически имеет две отображаемые категории: список значений в раскрывающемся списке, из которого пользователь может выбрать элемент, и значение текущего выбранного элемента. Если реализовать привязку данных для отображения списка элементов `ComboBox`, то значение, выбранное пользователем, должно входить в этот список. В приложении `Customers` вы можете настроить привязку данных для выбранного значения в элементе управления типа `ComboBox` по имени `title`, установив значение для свойства `SelectedValue`:

```
<ComboBox ... x:Name="title" ... SelectedValue="{Binding Title}" ... />
```

Но следует помнить, что список значений для раскрывающегося списка конкретно определен в XAML-разметке:

```
<ComboBox ... x:Name="title" ... >
  <ComboBoxItem Content="Mr"/>
  <ComboBoxItem Content="Mrs"/>
  <ComboBoxItem Content="Ms"/>
  <ComboBoxItem Content="Miss"/>
</ComboBox>
```

До создания элемента управления эта разметка не применяется, следовательно, значение, указанное в привязке данных, не будет найдено в списке, поскольку при построении привязки данных списка еще не существует. В результате этого значение не будет выведено на экран. Если хотите, можете проверить — настройте привязку для свойства `SelectedValue`, как только что было показано, и запустите приложение. Элемент управления `title` типа `ComboBox` при начальном отображении будет пуст, несмотря на то что обращение к пользователю было в форме `Mr`.

Эту проблему можно решить несколькими способами, но проще всего создать источник данных, содержащий список допустимых значений, а затем указать, что элемент управления типа `ComboBox` должен использовать этот список при

установке своих значений для раскрывающегося списка. Кроме того, вам нужно сделать это до применения к элементу управления типа ComboBox привязки данных.

Реализация привязки данных для элементов управления типа ComboBox по имени title

Откройте в окне редактора среды Visual Studio файл MainPage.xaml.cs. Добавьте к конструктору MainPage следующий код, выделенный жирным шрифтом:

```
public MainPage()
{
    this.InitializeComponent();

    List<string> titles = new List<string>
    {
        "Mr", "Mrs", "Ms", "Miss"
    };

    this.title.ItemsSource = titles;
    this.cTitle.ItemsSource = titles;

    Customer customer = new Customer
    {
        ...
    };

    this.DataContext = customer;
}
```

Этот код создает список строковых значений, содержащий допустимые обращения, которые могут быть применены к клиентам. Затем код устанавливает в качестве значения свойства `ItemsSource` для обоих элементов управления типа ComboBox по имени `title` ссылку на этот список (вспомним, что элемент управления типа ComboBox имеется в каждом представлении).



ПРИМЕЧАНИЕ В коммерческом приложении список значений, отображаемых элементом управления типа ComboBox, будет извлекаться, скорее всего, из базы данных или какого-нибудь другого источника, а не из фиксированного списка, как показано в этом примере.

Важно определить для этого кода правильно место. Он должен запускаться перед инструкцией, устанавливающей свойство `DataContext` формы `MainPage`, поскольку эта инструкция привязывает данные к элементам управления формы.

Выполните в окне редактора файл `MainPage.xaml`. Измените XAML-разметку для элементов управления `title` и `cTitle`, относящихся к типу ComboBox (изменения выделены жирным шрифтом):

```
<Grid x:Name="customersTabularView" ...>
    ...
    <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" ...
SelectedValue="{Binding Title, Mode=TwoWay}">
    </ComboBox>
    ...
</Grid>
<Grid x:Name="customersColumnarView" ...>
    ...
    <ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle" ...
SelectedValue="{Binding Title, Mode=TwoWay}">
    </ComboBox>
    ...
</Grid>
```

Заметьте, что список `ComboBoxItem`-элементов для каждого элемента управления был перемещен и свойство `SelectedValue` настроено на использование привязки данных к полю `Title` в объекте типа `Customer`.

В меню Отладка щелкните на пункте Начать отладку, чтобы выполнить сборку и запуск приложения.

Убедитесь в том, что значение обращения к клиенту отображено правильно (оно должно выглядеть как `Mr`). Щелкните на стрелке раскрытия списка в элементе типа `ComboBox` и убедитесь в том, что он содержит значения `Mr`, `Mrs`, `Ms` и `Miss`.

Измените размер окна, чтобы приложение выводилось в представлении для узкой области просмотра, и проконтролируйте наличие тех же самых данных. Заметьте, что вы можете изменить обращение и при переключении на представление для широкой области просмотра будет отображаться новая форма обращения.

Вернитесь в среду Visual Studio и остановите отладку.

Создание модели представления (ViewModel)

Вы увидели, как настраивается привязка данных для связи источника данных с элементами управления в пользовательском интерфейсе, но используемый источник данных был очень простым, состоящим из данных всего лишь одного клиента. В реальном мире источники данных намного сложнее и состоят из коллекций различных типов объектов. Вспомним, что в понятиях MVVM источник данных зачастую предоставляется моделью, а пользовательский интерфейс (или представление) связывается с моделью только опосредованно, через объект модели представления (`ViewModel`). Рациональным зерном такого подхода является выполнение требования независимости модели и представления, отображающего данные, предоставляемые моделью: вам не нужно изменять модель в случае изменения пользовательского интерфейса и не требуется подстраивать пользовательский интерфейс при внесении изменений в базовую модель.

Связь между представлением и моделью обеспечивает модель представления, в которой реализуется также бизнес-логика приложения. Эта бизнес-логика должна быть независима от представления и модели. Модель представления предоставляет бизнес-логику представлению, реализуя коллекцию команд. Пользовательский интерфейс может инициировать эти команды, основываясь на способе перемещения пользователя по приложению. В следующем упражнении вы расширите приложение *Customers* за счет реализации модели, содержащей список *Customer*-объектов, и создания модели представления, предоставляющей команды, с помощью которых пользователь может осуществлять в представлении переходы между сведениями о клиентах.

Создание модели представления для управления информацией о клиентах

Откройте проект *Customers*, который находится в папке `\Microsoft Press\VCBS\Chapter 26\ViewModel` вашей папки документов. Он содержит полную версию приложения *Customers* из предыдущего набора упражнений, но если хотите, можете и дальше использовать собственную версию проекта.

В обозревателе решений щелкните правой кнопкой мыши на проекте *Customers*, укажите на пункт **Добавить**, а затем щелкните на пункте **Класс**.

Наберите в поле **Имя** диалогового окна **Добавить новый элемент** — *Customers* строку `ViewModel.cs`, а затем щелкните на кнопке **Добавить**.

Этот класс предоставляет основную модель представления (*ViewModel*), в которой содержится коллекция объектов типа *Customer*. Пользовательский интерфейс будет привязан к данным, предоставляемым этой моделью представления.

Пометьте в окне редактора, показывающем файл `ViewModel.cs`, класс открытым (`public`) и добавьте к классу `ViewModel` код, показанный в следующем примере жирным шрифтом:

```
public class ViewModel
{
    private List<Customer> customers;

    public ViewModel()
    {
        this.customers = new List<Customer>
        {
            new Customer
            {
                CustomerID = 1,
                Title = "Mr",
                FirstName="John",
                LastName="Sharp",
                EmailAddress="john@contoso.com",
            }
        }
    }
}
```

```

        Phone="111-1111"
    },
    new Customer
{
    CustomerID = 2,
    Title = "Mrs",
    FirstName="Diana",
    LastName="Sharp",
    EmailAddress="diana@contoso.com",
    Phone="111-1112"
},
new Customer
{
    CustomerID = 3,
    Title = "Ms",
    FirstName="Francesca",
    LastName="Sharp",
    EmailAddress="frankie@contoso.com",
    Phone="111-1113"
}
);
}
}
}

```

Класс `ViewModel` использует в качестве своей модели объект типа `List<Customer>`, и конструктор заполняет этот список образцом данных. Строго говоря, эти данные должны храниться в отдельном классе `Model`, но, преследуя цели именно этого упражнения, мы будем работать с образцом данных.

Добавьте к классу `ViewModel` закрытую переменную `currentCustomer`, показанную в следующем примере кода жирным шрифтом, и инициализируйте ее в конструкторе нулевым значением:

```

class ViewModel
{
    private List<Customer> customers;
private int currentCustomer;

    public ViewModel()
    {
        this.currentCustomer = 0;
        this.customers = new List<Customer>
        {
            ...
        }
    }
}

```

Класс `ViewModel` будет использовать эту переменную для отслеживания того объекта типа `Customer`, который в данный момент выводится на экран.

Добавьте к классу `ViewModel` сразу после конструктора свойство `Current`, выделенное в следующем примере кода жирным шрифтом:

```

class ViewModel
{
    ...

    public ViewModel()
    {
        ...
    }

    public Customer Current
    {
        get { return this.customers.Count > 0 ?
              this.customers[currentCustomer] : null; }
    }
}

```

Свойство `Current` предоставляет в модели доступ к текущему `Customer`-объекту. Если клиенты отсутствуют, оно возвращает объект со значением `null`.



ПРИМЕЧАНИЕ Предоставление управляемого доступа к модели данных считается устоявшейся практикой — возможность внесения изменений в модель должна быть только у `ViewModel`. Но это ограничение не препятствует тому, чтобы представление могло обновлять данные, предоставляемые `ViewModel`, — оно просто не может переключить модель и заставить ее ссылаться на другой источник данных.

В окне редактора откройте файл `MainPage.xaml.cs`. Удалите из конструктора `MainPage` код, создающий объект типа `Customer`, заменив его инструкцией, создающей экземпляр класса `ViewModel`. Измените инструкцию, устанавливающую значение для свойства `DataContext` объекта типа `MainPage`, указав ссылку на новый объект типа `ViewModel` (изменения выделены жирным шрифтом):

```

public MainPage()
{
    ...
    this.cTitle.ItemsSource = titles;

    ViewModel viewModel = new ViewModel();
    this.DataContext = viewModel;
}

```

В окне конструктора откройте файл `MainPage.xaml`. Измените в панели XAML привязки данных для элементов управления типа `TextBox` и `ComboBox`, чтобы в них были ссылки на свойства, задаваемые посредством объекта `Current`, предоставляемого моделью представления (изменения показаны жирным шрифтом):

```

<Grid x:Name="customersTabularView" ...>
    ...
    <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
Text="{Binding Current.CustomerID, Mode=TwoWay}" .../>
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...

```

```
Text="{Binding Current.FirstName, Mode=TwoWay }" .../>
    <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...
Text="{Binding Current.LastName, Mode=TwoWay }" .../>
    <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" ...
SelectedValue="{Binding Current.Title, Mode=TwoWay}">
    </ComboBox>
    ...
    <TextBox Grid.Row="3" Grid.Column="3" ... x:Name="email" ...
Text="{Binding Current.EmailAddress, Mode=TwoWay }" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="3" ... x:Name="phone" ...
Text="{Binding Current.Phone, Mode=TwoWay }" .../>
</Grid>
<Grid x:Name="customersColumnarView" Margin="20,10,20,110" ...>
    ...
    <TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...
Text="{Binding Current.CustomerID, Mode=TwoWay }" .../>
    <TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...
Text="{Binding Current.FirstName, Mode=TwoWay }" .../>
    <TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...
Text="{Binding Current.LastName, Mode=TwoWay }" .../>
    <ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle" ...
SelectedValue="{Binding Current.Title, Mode=TwoWay}">
    </ComboBox>
    ...
    <TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...
Text="{Binding Current.EmailAddress, Mode=TwoWay }" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...
Text="{Binding Current.Phone, Mode=TwoWay }" .../>
</Grid>
```

В меню Отладка щелкните на пункте Начать отладку, чтобы выполнить сборку и запуск приложения. Убедитесь в том, что приложение отображает информацию о клиенте John Sharp (о первом клиенте в списке).

Измените информацию о клиенте и переключитесь на другое представление, чтобы удостовериться, что привязка данных по-прежнему работает должным образом.

Вернитесь в среду Visual Studio и остановите отладку.

Модель представления предоставляет доступ к клиентской информации через свойство `Current`, но на данный момент она не обеспечивает какого-либо способа перехода от информации об одном клиенте к информации о другом клиенте. Вы можете создать методы, уменьшающие и увеличивающие значение переменной `currentCustomer` на единицу, чтобы в свойство `Current` извлекалась информация о различных клиентах, но это нужно сделать так, чтобы не привязывать представление (View) к модели представления (ViewModel). Чаще всего для этой цели применяется технология, использующая шаблон `Command`. В этом шаблоне модель представления предоставляет методы в форме команд, которые могут вызываться представлением. Замысел заключается в том, чтобы обойтись в коде

представления без явной ссылки на эти методы по именам. Чтобы добиться этой цели, XAML позволяет декларативно связывать команды с действиями, инициируемыми элементами управления в пользовательском интерфейсе, что и будет показано в упражнениях следующего раздела.

Добавление команд к модели представления

XAML-разметка, привязывающая действие в отношении элемента управления к команде, требует, чтобы команды, предоставляемые моделью представления, реализовывали интерфейс `ICommand`. Этот интерфейс определяет следующие элементы.

- **CanExecute.** Этот метод возвращает булево значение, показывающее, может ли команда быть запущена. Используя этот метод, модель представления может в зависимости от контекста включить или выключить команду. Например, команда, которая извлекает из списка данные о следующем клиенте, должна включаться для запуска, только если есть следующий клиент, если же клиентов больше нет, команда должна быть выключена.
- **Execute.** Этот метод запускается при вызове команды.
- **CanExecuteChanged.** Это событие инициируется, когда состояние модели представления изменяется. При таких обстоятельствах те команды, которые до этого могли запускаться, теперь могут быть выключены, и наоборот. Например, если пользовательский интерфейс вызывает команду, извлекающую из списка данные о следующем клиенте, и если этот клиент является последним, то следующие вызовы метода `CanExecute` должны возвращать `false`. При таких обстоятельствах должно инициироваться событие `CanExecuteChanged`, показывающее, что команда была выключена.

В следующем упражнении будет создан класс-обобщение, реализующий интерфейс `ICommand`.

Реализация класса Command

В среде Visual Studio щелкните правой кнопкой мыши на проекте `Customers`, укажите на пункт **Добавить** и щелкните на пункте **Класс**.

В диалоговом окне **Добавить новый элемент** — `Customers` выберите шаблон **Класс**. В поле **Имя** наберите строку `Command.cs`, а затем щелкните на кнопке **Добавить**.

В окне редактора, показывающем код файла `Command.cs`, добавьте к списку в начале файла следующую директиву `using`:

```
using System.Windows.Input;
```

В этом пространстве имен определен интерфейс `ICommand`.

Объявите класс `Command` открытым и укажите, что в нем реализуется интерфейс `ICommand`, добавив к его объявлению следующие элементы, показанные жирным шрифтом:

```
public class Command : ICommand  
{  
}
```

Добавьте к классу `Command` следующие закрытые поля:

```
public class Command : ICommand  
{  
    private Action methodToExecute = null;  
    private Func<bool> methodToDetectCanExecute = null;  
}
```

Типы `Action` и `Func` были вкратце рассмотрены в главе 20 «Отделение логики приложения и обработка событий». Тип `Action` является делегатом, которым можно пользоваться для ссылки на метод, не принимающий параметры и не возвращающий значение, тип `Func<T>` также является делегатом, который может ссылаться на метод, не принимающий параметры, но возвращающий значение того типа, который указан параметром типа `T`. В этом классе для ссылки на код, который `Command`-объект будет запускать после его вызова из представления, вы будете использовать поле `methodToExecute`. Поле `methodToDetectCanExecute` будет использоваться для ссылки на метод, который обнаруживает возможность запуска команды (она может быть выключена по каким-либо причинам, зависящим от состояния приложения или данных).

Добавьте к классу `Command` конструктор, который будет принимать два параметра: `Action`-объект и `Func<T>`-объект. Как показано далее жирным шрифтом, значения этих параметров нужно присвоить полям `methodToExecute` и `methodToDetectCanExecute`:

```
public Command : ICommand  
{  
    ...  
    public Command(Action methodToExecute, Func<bool> methodToDetectCanExecute)  
    {  
        this.methodToExecute = methodToExecute;  
        this.methodToDetectCanExecute = methodToDetectCanExecute;  
    }  
}
```

Модель представления создаст экземпляр этого класса для каждой команды. При вызове конструктора модель представления предоставит метод для запуска команды и метод для обнаружения того, следует ли включить команду.

Реализуйте методы `Execute` и `CanExecute` класса `Command`, воспользовавшись методами, на которые ссылаются поля `methodToExecute` и `methodToDetectCanExecute`:

```

public Command : ICommand
{
    ...
    public Command(Action methodToExecute,
        Func<bool> methodToDetectCanExecute)
    {
        ...
    }

    public void Execute(object parameter)
    {
        this.methodToExecute();
    }

    public bool CanExecute(object parameter)
    {
        if (this.methodToDetectCanExecute == null)
        {
            return true;
        }
        else
        {
            return this.methodToDetectCanExecute();
        }
    }
}

```

Заметьте, что если модель представления дает для параметра конструктора `methodToDetectCanExecute` ссылку `null`, то по умолчанию предполагается, что команда может быть запущена, и метод `CanExecute` возвращает `true`.

Добавьте к классу `Command` событие `CanExecuteChanged`:

```

public Command : ICommand
{
    ...
    public bool CanExecute(object parameter)
    {
        ...
    }

    public event EventHandler CanExecuteChanged;
}

```

Когда команда привязывается к элементу управления, он автоматически подписывается на это событие, которое будет инициировано `Command`-объектом, если состояние модели представления обновлено и значение, возвращенное методом `CanExecute`, изменяется. Самой простой стратегией будет использование таймера для иницирования события `CanExecuteChanged` примерно раз в секунду. Тогда элемент управления может вызвать `CanExecute`, чтобы определить, может ли команда по-прежнему быть выполнена, и предпринять действия по своему включению или выключению в зависимости от результата.

Добавьте к списку в начале файла следующую директиву `using`:

```
using Windows.UI.Xaml;
```

Добавьте к классу `Command` выше конструктора следующее поле, выделенное жирным шрифтом:

```
public class Command : ICommand
{
    ...
    private Func<bool> methodToDetectCanExecute = null;
    private DispatcherTimer canExecuteChangedEventTimer = null;

    public Command(Action methodToExecute,
        Func<bool> methodToDetectCanExecute)
    {
        ...
    }
}
```

Класс `DispatcherTimer`, определение которого находится в пространстве имен `Windows.UI.Xaml`, реализует таймер, который может инициировать событие через указанные интервалы времени. Для инициирования события `CanExecuteChanged` через односекундные интервалы вы будете использовать поле `canExecuteChangedEventTimer`.

Добавьте к классу `Command` метод `canExecuteChangedEventTimer_Tick`, показанный в следующем примере кода жирным шрифтом:

```
public class Command : ICommand
{
    ...
    public event EventHandler CanExecuteChanged;

    void canExecuteChangedEventTimer_Tick(object sender, object e)
    {
        if (this.CanExecuteChanged != null)
        {
            this.CanExecuteChanged(this, EventArgs.Empty);
        }
    }
}
```

Этот метод просто инициирует событие `CanExecuteChanged`, если как минимум один элемент управления привязан к команде. Строго говоря, этот метод перед инициированием события должен также проверять, не изменилось ли состояние объекта. Но чтобы свести к минимуму издержки от неэффективности работы кода, связанной с отказом от проверки состояния, вы установите интервал таймера на достаточно продолжительный (в понятиях обработки данных) период.

Добавьте к конструктору `Command` следующие инструкции, выделенные жирным шрифтом:

```
public class Command : ICommand
{
    ...
    public Command(Action methodToExecute, Func<bool> methodToDetectCanExecute)
    {
        this.methodToExecute = methodToExecute;
        this.methodToDetectCanExecute = methodToDetectCanExecute;

        this.canExecuteChangedEventTimer = new DispatcherTimer();
        this.canExecuteChangedEventTimer.Tick +=
            canExecuteChangedEventTimer_Tick;
        this.canExecuteChangedEventTimer.Interval = new TimeSpan(0, 0, 1);
        this.canExecuteChangedEventTimer.Start();
    }
    ...
}
```

Этот код инициализирует `DispatcherTimer`-объект и устанавливает перед запуском таймера значение его интервала на одну секунду.

В меню Сборка щелкните на пункте Собрать решение и убедитесь в том, что сборка приложения прошла без ошибок.

Теперь вы можете воспользоваться классом `Command` для добавления команд к классу `ViewModel`. В следующем упражнении вы определите команды, позволяющие пользователю переходить в представлении между сведениями о разных клиентах.

Добавление к классу `ViewModel` команд `NextCustomer` и `PreviousCustomer`

В окне редактора среды Visual Studio откройте файл `ViewModel.cs`. Добавьте к началу файла следующую директиву `using` и измените определение класса `ViewModel`, указав, что в нем реализуется интерфейс `INotifyPropertyChanged`:

```
...
using System.ComponentModel;

namespace Customers
{
    public class ViewModel : INotifyPropertyChanged
    {
        ...
    }
}
```

Добавьте к концу класса `ViewModel` событие `PropertyChanged` и метод `OnPropertyChanged`. Это точно такой же код, который был включен вами в класс `Customer`:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Не забудьте, что представление для различных входящих в него элементов управления ссылается на данные в выражениях привязки данных посредством свойства `Current`. Когда класс `ViewModel` переходит к данным другого клиента, он должен инициировать событие `PropertyChanged`, чтобы уведомить представление о том, что отображаемые данные были изменены.

Добавьте к классу `ViewModel` сразу же после конструктора следующие поля и свойства:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public ViewModel()
    {
        ...
    }

    private bool _isAtStart;
    public bool IsAtStart
    {
        get { return this._isAtStart; }
        set
        {
            this._isAtStart = value;
            this.OnPropertyChanged(nameof(IsAtStart));
        }
    }

    private bool _isAtEnd;
    public bool IsAtEnd
    {
        get { return this._isAtEnd; }
        set
        {
            this._isAtEnd = value;
            this.OnPropertyChanged(nameof(IsAtEnd));
        }
    }
    ...
}
```

Эти два свойства будут использоваться вами для отслеживания состояния `ViewModel`. Свойство `IsAtStart` будет установлено в `true`, когда поле `currentCustomer` в `ViewModel` позиционировано на начало коллекции клиентов, а поле `IsAtEnd` будет установлено в `true`, когда это поле позиционировано на конец коллекции клиентов.

Внесите в конструктор изменения, показанные жирным шрифтом, для установки значений свойств `IsAtStart` и `IsAtEnd`:

```
public ViewModel()
{
    this.currentCustomer = 0;
    this.IsAtStart = true;
    this.IsAtEnd = false;

    this.customers = new List<Customer>
    ...
}
```

Добавьте к классу `ViewModel` после свойства `Current` закрытые методы `Next` и `Previous`, выделенные жирным шрифтом:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public Customer Current
    {
        ...
    }

    private void Next()
    {
        if (this.customers.Count - 1 > this.currentCustomer)
        {
            this.currentCustomer++;
            this.OnPropertyChanged(nameof(Current));
            this.IsAtStart = false;
            this.IsAtEnd =
                (this.customers.Count - 1 == this.currentCustomer);
        }
    }

    private void Previous()
    {
        if (this.currentCustomer > 0)
        {
            this.currentCustomer--;
            this.OnPropertyChanged(nameof(Current));
            this.IsAtEnd = false;
            this.IsAtStart = (this.currentCustomer == 0);
        }
    }
    ...
}
```



ПРИМЕЧАНИЕ Свойство Count возвращает количество элементов в коллекции, но не забудьте, что нумерация элементов коллекции идет от 0 до Count – 1.

Эти методы обновляют значение переменной `currentCustomer` для ссылки на следующего (или предыдущего) клиента в списке клиентов. Заметьте, что эти методы предоставляют значения для свойств `IsAtStart` и `IsAtEnd` и показывают, что текущий клиент изменился, инициируя для свойства `Current` событие `PropertyChanged`. Эти методы объявлены закрытыми, потому что к ним не должно быть доступа за пределами класса `ViewModel`. Внешние классы будут запускать их, используя команды, которые вы добавите на следующих этапах выполнения упражнения.

Добавьте к классу `ViewModel` автоматически создаваемые свойства `NextCustomer` и `PreviousCustomer`, показанные далее жирным шрифтом:

```
public class ViewModel : INotifyPropertyChanged
{
    private List<Customer> customers;
    private int currentCustomer;
    public Command NextCustomer { get; private set; }
    public Command PreviousCustomer { get; private set; }
    ...
}
```

Представление будет привязываться к этим `Command`-объектам, чтобы пользователь мог переходить от клиента к клиенту.

Установите в конструкторе `ViewModel` для свойств `NextCustomer` (следующий клиент) и `PreviousCustomer` (предыдущий клиент) ссылки на новые `Command`-объекты:

```
public ViewModel()
{
    this.currentCustomer = 0;
    this.IsAtStart = true;
    this.IsAtEnd = false;
    this.NextCustomer = new Command(this.Next, () =>
        { return this.customers.Count > 1 && !this.IsAtEnd; });
    this.PreviousCustomer = new Command(this.Previous, () =>
        { return this.customers.Count > 0 && !this.IsAtStart; });
    ...
}
```

Ссылка на новый `Command`-объект в свойстве `NextCustomer` указывает на метод `Next` как на операцию, проводимую в том случае, когда вызывается метод `Execute`. Лямбда-выражение `() => { return this.customers.Count > 1 && !this.IsAtEnd; }` указывается в качестве функции, вызываемой при запуске метода `CanExecute`. Это выражение возвращает `true` при условии, что в списке клиентов содержится более одного клиента, а объект типа `ViewModel` не позиционирован на последнего клиента в списке. Ссылка на новый `Command`-объект в свойстве `PreviousCustomer` действует по аналогичной схеме: она вызывает метод `Previous` для извлечения

из списка предыдущего клиента, а метод `CanExecute` ссылается на выражение `() => { return this.customers.Count > 0 && !this.IsAtStart; }`, которое возвращает `true` при условии, что в списке клиентов содержится хотя бы один клиент, и объект типа `ViewModel` не позиционирован на первого клиента в этом списке.

В меню Сборка щелкните на пункте Собрать решение и убедитесь в том, что сборка приложения прошла без ошибок.

Теперь, после добавления к `ViewModel` команд `NextCustomer` и `PreviousCustomer`, вы можете привязать их к кнопкам представления. Когда пользователь щелкнет на кнопке, запустится соответствующая команда.

Общие рекомендации и руководства по Microsoft publishes, касающиеся добавления кнопок к представлениям в UWP-приложениях, сводятся к тому, что кнопки, вызывающие команды, должны помещаться на панели команд. В UWP-приложениях предоставляются две панели команд: та, что появляется в верхней части формы, и та, что появляется в ее нижней части. Кнопки навигации по приложению или данным чаще всего помещаются в панель команд, расположенную в верхней части. Именно этим подходом вы и воспользуетесь в следующем упражнении.



ПРИМЕЧАНИЕ Рекомендации компании Microsoft относительно реализации панелей команд можно найти по адресу <http://msdn.microsoft.com/library/windows/apps/hh465302.aspx>.

Добавление к форме Customers кнопок Next и Previous

В окне конструктора откройте файл `MainPage.xaml`. Перейдите в нижнюю часть кода в панели XAML и добавьте следующую разметку, выделенную жирным шрифтом, непосредственно перед закрывающим тегом `</Page>`:

```
...
<Page.TopAppBar>
    <CommandBar>
        <AppBarButton x:Name="previousCustomer" Icon="Previous"
Label="Previous" Command="{Binding Path=PreviousCustomer}"/>
        <AppBarButton x:Name="nextCustomer" Icon="Next"
Label="Next" Command="{Binding Path=NextCustomer}"/>
    </CommandBar>
</Page.TopAppBar>
</Page>
```

В данном фрагменте XAML-разметки следует обратить внимание на ряд особенностей.

- ❑ По умолчанию панель команд появляется в верхней части экрана — видны значки содержащихся на ней кнопок. Надпись для каждой кнопки возникает только тогда, когда пользователь щелкает на кнопке `More (...)`, появляющейся

справа от панели команд. Но при разработке приложения, которое может использоваться в разных локализациях, следует для надписей вместо жестко заданных значений сохранять текст надписей в файле ресурсов с указанием локализаций, а свойство `Label` привязывать к нему в динамическом режиме при запуске приложения. Для получения дополнительных сведений следует обратиться к странице «Краткое руководство: преобразование ресурсов пользовательского интерфейса (XAML)» на веб-сайте компании Microsoft по адресу <https://msdn.microsoft.com/library/windows/apps/xaml/hh965329.aspx>.

- Элемент управления `CommandBar` может содержать лишь ограниченный набор элементов управления (тех, которые реализуют интерфейс `ICommandBarElement`). Этот набор включает элементы управления типа `AppBarButton`, `AppBarToggleButton` и `AppBarSeparator`. Они специально разработаны для работы внутри элемента `CommandBar`. При попытке добавления на панель команд такого элемента управления, как кнопка, будет получено сообщение об ошибке с объяснением, что указанное значение не может быть присвоено коллекции: «The specified value cannot be assigned to the collection».
- Шаблоны UWP-приложения включают разнообразные значки (такие, как для кнопок `Previous` и `Next`, показанных в предыдущем примере кода), которые можно отобразить в элементе управления `AppBarButton`. Просмотреть весь набор доступных значков можно с помощью окна Свойства. Щелкните в разделе Значок на пункте `Symbol Icon` — в поле списка будет выведен набор доступных значков. Можно также определить собственные значки и графические изображения.
- У каждой кнопки есть свойство `Command`, которое можно привязать к объекту, реализующему интерфейс `ICommand`. В данном приложении кнопки вами были привязаны в классе `ViewModel` к командам `PreviousCustomer` и `NextCustomer`. Когда пользователь в ходе выполнения приложения щелкает на любой из них, запускается соответствующая команда.

Щелкните в меню Отладка на пункте Начать отладку. Должна появиться форма `Customers` и показать информацию о клиенте по имени John Sharp. На рис. 26.4 показано, что в верхней части формы должна отобразиться панель команд, содержащая кнопки `Next` и `Previous`.

Заметьте, что кнопка `Previous` недоступна. Дело в том, что свойство `IsAtStart` `ViewModel`-объекта имеет значение `true`, а метод `CanExecute` объекта типа `Command`, на который ссылается кнопка `Previous`, показывает, что команду запустить нельзя.

На панели команд щелкните на кнопке с тремя точками. Должны появиться надписи для кнопок. Они будут видны до тех пор, пока не будет сделан щелчок на одной из кнопок на панели команд.

На панели команд щелкните на кнопке `Next`. Появятся сведения о клиенте № 2, Diana Sharp, и после небольшой задержки (не более секунды) кнопка `Previous`

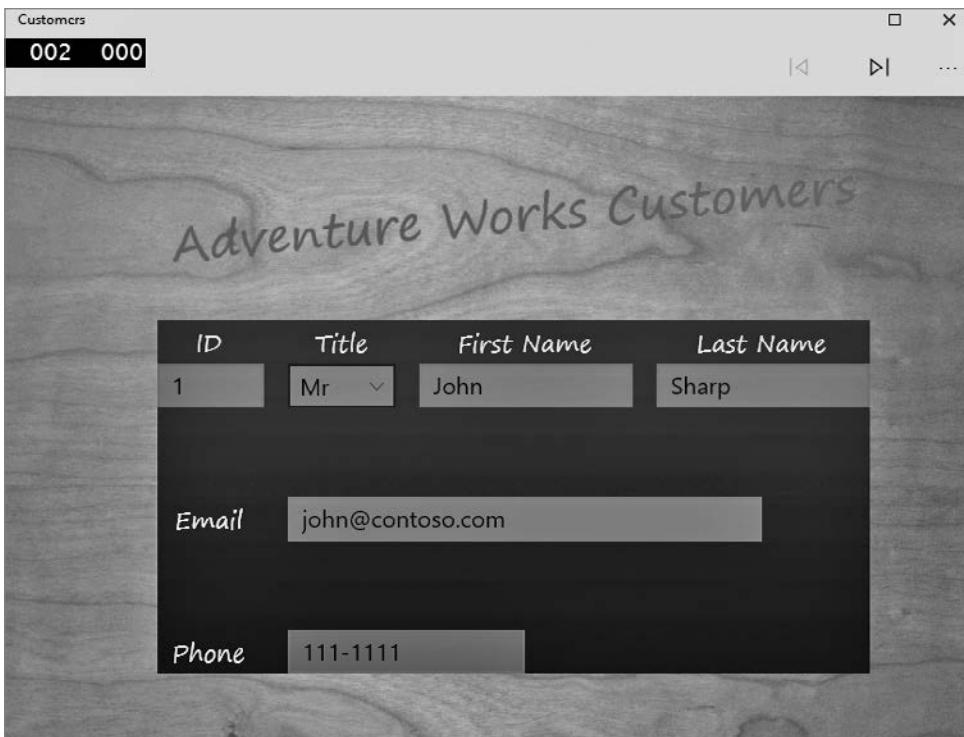


Рис. 26.4

станет доступной. Теперь свойство `IsAtStart` уже не имеет значения `true`, поэтому метод `CanExecute` этой команды возвращает `true`. Но кнопка не уведомляется об этом изменении состояния до тех пор, пока не истечет время в объекте таймера команды и не будет инициировано событие `CanExecuteChanged`, на что может уйти не более секунды.



ПРИМЕЧАНИЕ Если вам нужна более быстрая реакция на изменение состояния команд, можно настроить таймер в классе `Command` на более частое истечение времени. Но не следует сокращать время срабатывания таймера слишком сильно, поскольку излишне частое инициирование события `CanExecuteChanged` может повлиять на производительность интерфейса пользователя.

Еще раз щелкните на кнопке `Next` на панели команд.

Должны появиться сведения о клиенте № 3, *Francesca Sharp*, и после не более чем секундной задержки кнопка `Next` должна стать недоступной. На этот раз значение `true` получит свойство `IsAtEnd` объекта типа `ViewModel`, поэтому метод `CanExecute` объекта типа `Command` для кнопки `Next` вернет значение `false` и команда станет недоступной.

Измените размер окна приложения, чтобы оно перешло к отображению, предназначенному для узкой области просмотра, и убедитесь, что приложение продолжает работать подобающим образом. Кнопки **Next** и **Previous** будут вызывать переход в списке клиентов на одну позицию.

Вернитесь в среду Visual Studio и остановите отладку.

Выходы

В этой главе вы научились выводить данные в форму с помощью привязки данных. Вы увидели, как устанавливается контекст данных для формы и как реализацией интерфейса **INotifyPropertyChanged** создается источник данных, поддерживающий привязку данных. Вы также научились использовать для создания UWP-приложения шаблон **Model – View – ViewModel** и увидели, как создается модель представления (**ViewModel**), с помощью которой представление может взаимодействовать с источником данных путем использования команд.

Краткий справочник

Чтобы	Сделайте следующее
Привязать свойство элемента управления к свойству объекта	Воспользуйтесь выражением привязки данных в XAML-разметке элемента управления, например: <code><TextBox ... Text="{Binding FirstName}" ...></code>
Задействовать объект и уведомить привязку об изменении в значении данных	Реализуйте интерфейс INotifyPropertyChanged в том классе, который определяет объект и инициирует событие PropertyChanged при каждом изменении значения свойства, например: <code>class Customer : INotifyPropertyChanged { ... public event PropertyChangedEventHandler PropertyChanged; protected virtual void OnPropertyChanged(string propertyName) { if (PropertyChanged != null) { PropertyChanged(this, new PropertyChangedEventArgs(propertyName)); } } }</code>

Чтобы	Сделайте следующее
Задействовать элемент управления, использующий привязку данных, для обновления значения свойства, к которому сделана эта привязка	Настройте привязку данных на работу в обоих направлениях, например: <pre><TextBox ... Text="{Binding FirstName, Mode=TwoWay}" .../></pre>
Отделить бизнес-логику, запускаемую при щелчке пользователя на элементе управления типа Button, от пользовательского интерфейса, содержащего этот элемент управления	Воспользуйтесь моделью представления (ViewModel), которая предоставляет команды, реализованные с помощью интерфейса ICommand, и привяжите элемент управления типа Button к одной из этих команд, например: <pre><Button x:Name="nextCustomer" ... Command="{Binding Path=NextCustomer}"/></pre>

27

Доступ к удаленной базе данных из приложения универсальной платформы Windows

Прочитав эту главу, вы научитесь:

- использовать среду Entity Framework для создания модели элементов предметной области (entity-модели), способной извлекать и изменять информацию, хранящуюся в базе данных;
- создавать веб-сервис Representational State Transfer (REST), обеспечивающий удаленный доступ к базе данных через entity-модель;
- извлекать данные из удаленной базы данных путем использования веб-сервиса REST;
- производить вставку, обновление и удаление данных в удаленной базе данных путем использования веб-сервиса REST.

В главе 26 «Отображение и поиск данных в приложении универсальной платформы Windows» были показаны способы реализации шаблона Model — View — ViewModel (MVVM). В ней также объяснялось, как с использованием класса ViewModel, представляющего доступ к данным в модели и реализующего команды, которые могут использоваться пользовательским интерфейсом для вызова логики приложения, отделить бизнес-логику этого приложения от пользовательского интерфейса. В главе 26 также было показано, как привязка данных используется для отображения данных, предоставляемых классом ViewModel, и как пользовательский интерфейс может обновлять эти данные. Все эти технологии помогают создать полнофункциональное приложение универсальной платформы Windows.

В этой главе ваше внимание будет привлечено к модельному аспекту шаблона MVVM. В частности, вы увидите, как реализуется модель, которую ваше UWP-приложение может использовать для извлечения и обновления данных в удаленной базе данных.



ПРИМЕЧАНИЕ Облачная платформа Microsoft Azure предоставляет мобильные сервисы, которые можно использовать для создания веб-сервисов RESTful, предоставляющих доступ к серверной базе данных, наряду с Windows-приложением, которое может обращаться к этим сервисам с устройств Windows. Но на время написания этих строк данное предложение могло создавать приложения только для Windows 8.1, хотя в скором будущем ожидается появление поддержки Windows 10 (если она уже не появилась на тот момент, когда вы читаете эту книгу). Но даже при этом предложение может помочь понять, как можно создать подобную систему своими силами. Дополнительные сведения о мобильных сервисах для создания мобильных приложений можно найти в статье «Создание приложения Windows» по адресу <https://azure.microsoft.com/documentation/articles/app-service-mobile-dotnet-backend-windows-store-dotnet-get-started-preview/>.

Извлечение данных из базы данных

До сих пор вы использовали данные, ограничиваемые простой коллекцией, встроенной в модель представления приложения. В реальном мире данные, отображаемые и поддерживаемые приложением, чаще всего хранятся в таком источнике, как база данных.

Технологии, предоставляемые компанией Microsoft, не могут обеспечить непосредственный доступ к реляционной базе данных, хотя вам доступны решения для работы с базами данных от сторонних разработчиков. Это может восприниматься как весьма серьезное ограничение, но для него имеются довольно веские причины. Во-первых, эти технологии исключают зависимости от внешних ресурсов, которые могли бы возникнуть в UWP-приложениях, что превратило бы эти приложения в самостоятельный программный продукт, который несложно упаковать и загрузить из магазина Windows, при этом пользователям не потребуется устанавливать и настраивать на своем компьютере систему управления базами данных. Во-вторых, многие устройства, работающие под управлением Windows 10, имеют ограниченные ресурсы и не обладают достаточным объемом оперативной памяти или дискового пространства, позволяющим запустить локальную систему управления базами данных. Но многие бизнес-приложения все же выставляют требования по доступу к базе данных, и для удовлетворения потребностей при таком сценарии вы можете воспользоваться веб-сервисом.

Веб-сервисы могут реализовывать различные функции, но одним из самых распространенных сценариев их использования является предоставление интерфейса, с помощью которого приложение может подключиться к удаленному источнику данных с целью извлечения и обновления информации. Веб-сервис может находиться практически где угодно, от компьютера, на котором запущено приложение, до веб-сервера, размещенного на компьютере на другом континенте. При условии получения возможности подключения к веб-сервису вы можете воспользоваться им для предоставления доступа к хранилищу своей

информации. Среда Microsoft Visual Studio предоставляет шаблоны и инструментальные средства, позволяющие вам очень быстро и легко создать веб-сервис. Самая простая стратегия, показанная на следующей схеме (рис. 27.1), заключается в том, чтобы положить в основу веб-сервиса entity-модель, созданную путем использования среды Entity Framework.

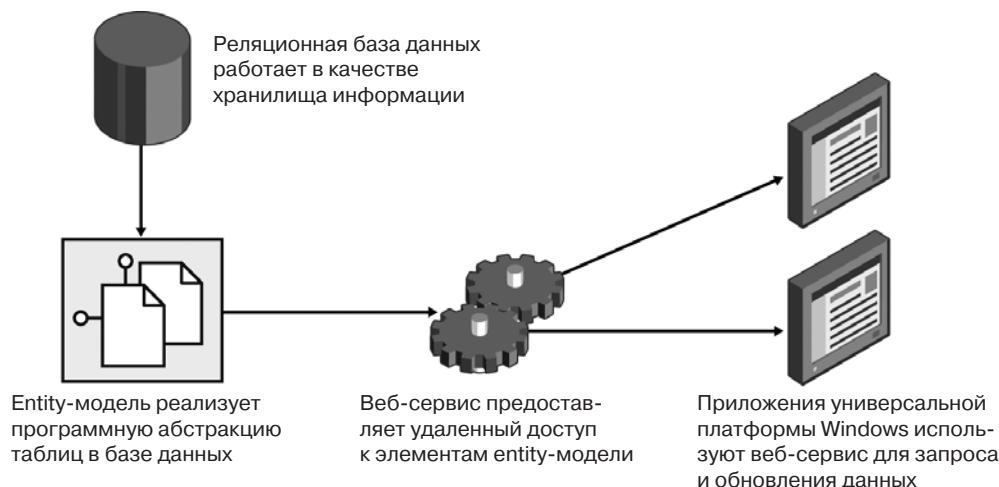


Рис. 27.1

Среда Entity Framework представляет собой эффективную технологию, позволяющую подключиться к реляционной базе данных. Она может сократить объем кода, который большинству разработчиков приходится создавать для добавления к приложениям возможности доступа к данным. С нее вы и начнете, но сначала нужно настроить базу данных AdventureWorks, содержащую информацию о клиентах компании Adventure Works.



ПРИМЕЧАНИЕ Ограниченные объемы книги не позволяют вдаваться во все тонкости использования среды Entity Framework, и упражнения в данном разделе проведут вас по самым главным действиям, позволяющим приступить к работе с этой средой. Если вам понадобятся дополнительные сведения, обратитесь к статье «Entity Framework» на веб-сайте компании Microsoft, которая находится по адресу <http://msdn.microsoft.com/data/aa937723>.

Чтобы придать сценарию более реалистичные очертания, упражнения в этой главе показывают, как создавать базу данных в облаке, используя Microsoft Azure SQL Database, и как развертывать веб-сервис в Azure. Эта архитектура используется во многих коммерческих приложениях, включая приложения электронной торговли, сервисы мобильного банкинга и даже системы потокового видео.



ПРИМЕЧАНИЕ Для выполнения упражнений потребуется создать учетную запись Azure и войти в нее. Если у вас пока нет учетной записи в Azure, можете подписать-ся на бесплатную пробную версию учетной записи, обратившись по адресу <https://azure.microsoft.com/pricing/free-trial/>. Кроме того, облачная платформа Azure требует, чтобы у вас была действующая учетная запись в Microsoft, которую она могла бы связать с вашей учетной записью в Azure. Подписаться на учетную запись Microsoft можно по адресу <https://signup.live.com/>.

Создание сервера Azure SQL Database и установка учебной базы данных AdventureWorks

Через веб-браузер подключитесь к порталу Azure, находящемуся по адресу <https://portal.azure.com>. Войдите на портал, используя свою учетную запись Microsoft.

В панели инструментов, находящейся в левой части окна портала, щелкните на пункте Создать. На странице Создать щелкните на пункте Данные + хранилище, а затем на пункте SQL Database.

На панели SQL Database наберите в поле Имя базы данных строку AdventureWorks.

Щелкните на пункте Сервер, а затем на пункте Создание нового сервера. На панели Новый сервер наберите уникальное имя для вашего сервера. (Воспользуйтесь именем вашей компании или даже своим именем, лично я использовал имя csharpstepbystep. Если введенное имя уже кем-то занято, вы получите предупреждение, после чего нужно будет выбрать другое имя.) Введите имя и пароль для учетной записи администратора (если говорить об этих элементах, то я воспользовался именем JohnSharp, а пароль раскрывать не намерен), выберите наиболее близкое место расположения, а затем щелкните на кнопке Выбрать.

Щелкните на пункте Выбрать источник, а затем на пункте Пример. Убедитесь, что выбран пункт AdventureWorksLT [V12].

Щелкните на пункте Ценовая категория. На панели Выберите ценовую категорию щелкните на пункте Basic, а затем на кнопке Выбрать. (Эта самый дешевый вариант, если вы платите за базу данных самостоятельно, и его будет вполне достаточно для работы с упражнениями в этой главе. Если создается широкомасштабное коммерческое приложение, то вам, скорее всего, понадобится использование ценовой категории Premium, предоставляющей больше места и более высокий уровень производительности, но при более высокой цене.) Эти настройки показаны на рис. 27.2.



ВНИМАНИЕ Не выбирайте какие-либо другие ценовые категории, кроме Basic, если не хотите получить в конце месяца счет на приличную сумму. Сведения о расценках на использование базы данных SQL Database можно найти по адресу <https://azure.microsoft.com/en-us/pricing/details/sql-database/>.

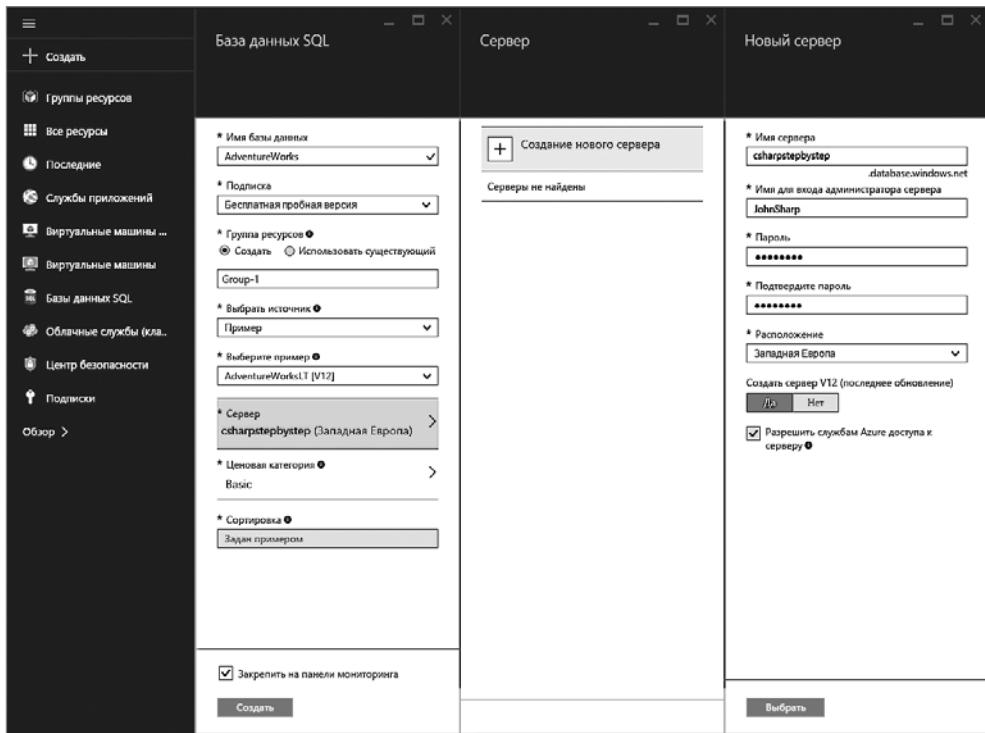


Рис. 27.2

Щелкните на кнопке **Создать** и дождитесь создания сервера базы данных и самой базы. Ход создания можно отслеживать на экране.

Щелкните на панели инструментов в левой части окна портала на кнопке **Обзор**.

Щелкните на сервере вашей базы данных, имя которого отображено на панели **Все ресурсы** страницы **Обзор** (но не на базе данных AdventureWorks).

Щелкните на пункте **Параметры**, который находится в верхней части панели сервера базы данных на панели инструментов.

На панели **Параметры** щелкните на пункте **Брандмауэр** и получите результат, показанный на рис. 27.3.

На панели инструментов **Параметры брандмауэра** щелкните на пункте **Добавить IP-адрес**. Щелкните на кнопке **Сохранить**. Убедитесь в том, что появилось сообщение «Правила брандмауэра сервера успешно обновлены», после чего щелкните на кнопке **OK**.

Закройте панель **Параметры брандмауэра**, затем панель **Параметры**, после чего закройте панель сервера базы данных.

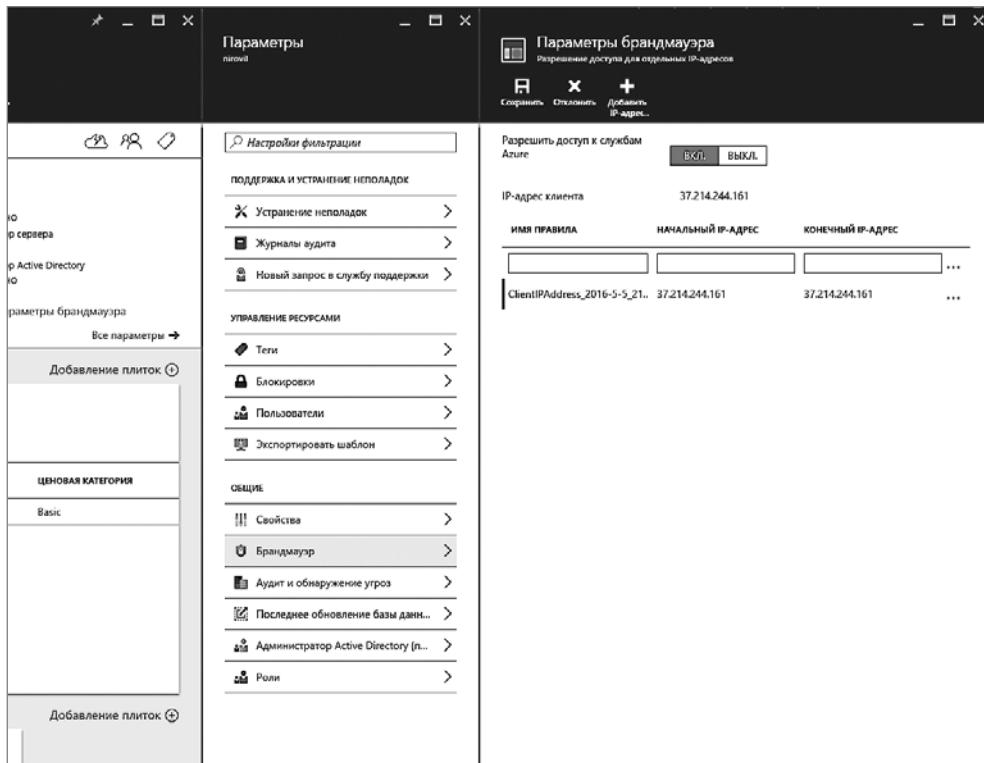


Рис. 27.3



ПРИМЕЧАНИЕ Выполнение этих действий играет важную роль. Без этого вы не сможете подключиться к базе данных из приложений, запущенных на вашем компьютере. Если нужно открыть доступ к набору компьютеров, вы также можете создать брандмауэры, пропускающие целый диапазон IP-адресов.

Используемая в качестве примера база данных AdventureWorks содержит в схеме SalesLT таблицу по имени Customer. Эта таблица включает столбцы, содержащие данные, предоставляемые UWP-приложением Customers, а также несколько других столбцов. Используя среду Entity Framework, можно игнорировать столбцы, не имеющие отношения к приложению, но если какие-либо проигнорированные столбцы не могут иметь пустых значений (null) и не имеют значений по умолчанию, вы не сможете создать записи о новых клиентах. В таблице Customer эти ограничения применяются к столбцам NameStyle, PasswordHash и PasswordSalt, используемым для шифрования паролей пользователей. Чтобы не возникали излишние сложности и вы смогли сконцентрироваться на функциональных возможностях приложения, в следующем упражнении вы удалите эти столбцы из таблицы Customer.

Удаление из базы данных AdventureWorks ненужных столбцов

В портале Azure щелкните в панели Все ресурсы на базе данных AdventureWorks.

На панели инструментов в верхней части панели AdventureWorks SQL Database щелкните на кнопке Сервис, а затем на пункте Открыть в Visual Studio. На панели Открыть в Visual Studio щелкните на пункте Открыть в Visual Studio.

Если появится сообщение «Did You Mean To Switch Applications?» («Вы имеете в виду переключение приложений?»), щелкните на кнопке Yes (Да) (рис. 27.4).

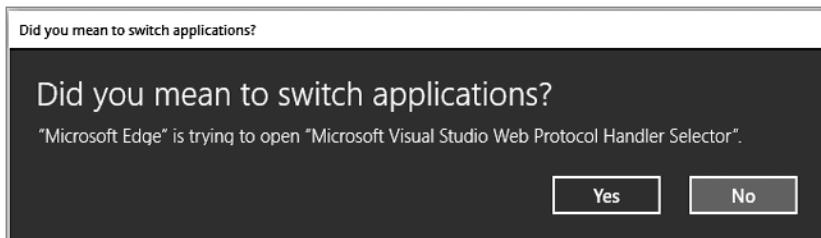


Рис. 27.4

Запустится среда Visual Studio, которая выведет приглашение на подключение к базе данных.

Если появится диалоговое окно Соединение с сервером, введите указанный ранее пароль администратора и щелкните на кнопке Соединить (рис. 27.5).

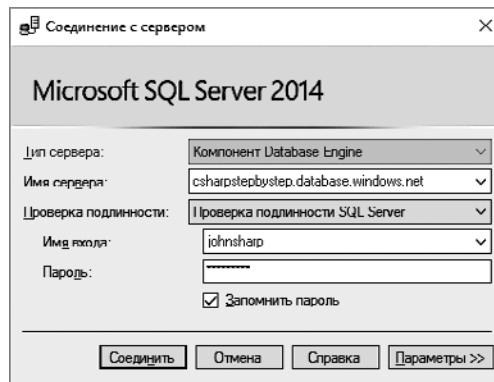


Рис. 27.5

Среда Visual Studio подключится к базе данных, которая появится на панели Обозреватель объектов SQL Server в левой части окна среды Visual Studio (рис. 27.6).

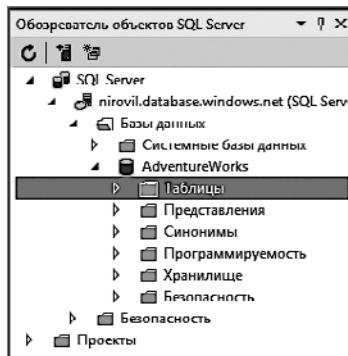


Рис. 27.6

На панели Обозреватель объектов SQL Server раскройте элемент с базой данных AdventureWorks, раскройте элемент Таблицы, элемент SalesLT.Customer, а затем элемент Столбцы.

Будет открыт список со столбцами таблицы. Должны быть удалены три столбца, которые не используются приложением и не допускают пустых значений (в противном случае приложение не сможет создать записи для новых клиентов).

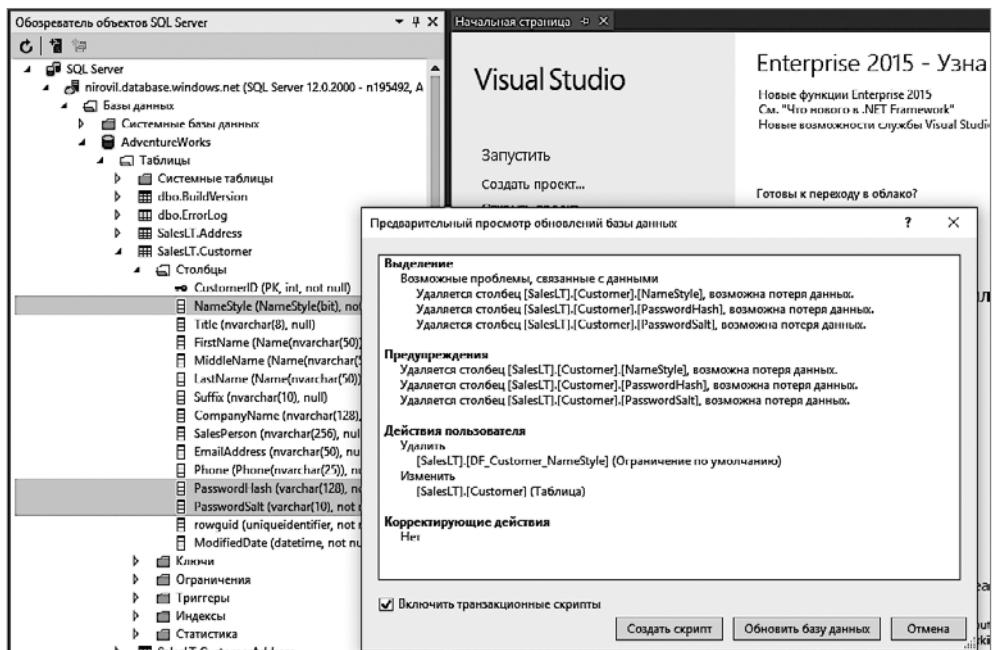


Рис. 27.7

Щелкните на столбце **NameStyle**, нажмите клавишу **Ctrl** и, удерживая ее, щелкните на столбцах **PasswordHash** и **PasswordSalt**. Щелкните правой кнопкой мыши на столбце **PasswordSalt**, а затем на пункте **Удалить**.

Среда Visual Studio проанализирует эти столбцы. В диалоговом окне **Предварительный просмотр обновлений базы данных** она покажет список предупреждений и других вопросов, которые могут возникнуть при удалении столбцов (рис. 27.7).

В диалоговом окне **Предварительный просмотр обновлений базы данных** щелкните на кнопке **Обновить базу данных**.

Закройте панель **Обозреватель объектов SQL Server**, но оставьте открытым окно среды Visual Studio 2015.

Создание entity-модели

После создания в облаке базы данных AdventureWorks появляется возможность создать с помощью среды Entity Framework entity-модель, которой приложение сможет пользоваться для создания запросов к этой базе данных и обновления в ней информации. Если вы имеете опыт работы с базами данных, вам могут быть знакомы такие технологии, как ADO.NET, предоставляющие библиотеку классов, которой можно воспользоваться для подключения к базе данных и запуска SQL-команд. ADO.NET, конечно, полезная технология, но она требует определенного знания языка SQL, и если не проявить должной осмотрительности, она может навязать вам структурирование кода вокруг логики, необходимой для выполнения SQL-команд, вместо того чтобы сконцентрироваться на бизнес-операциях вашего приложения. Среда Entity Framework предоставляет уровень абстракции, сокращающий зависимость ваших приложений от SQL.

Фактически среда Entity Framework реализует уровень отображения между реляционной базой данных и вашим приложением: она создает entity-модель, состоящую из коллекций объектов, которую ваше приложение может использовать точно так же, как и любую другую коллекцию. Эта коллекция обычно соответствует таблице в базе данных, и каждая строка таблицы соответствует элементу коллекции. Запросы выполняются путем последовательного перебора элементов коллекции обычно с помощью интегрированного в C# языка запросов (Language-Integrated Query (LINQ)). Entity-модель скрытно преобразует ваши запросы в команды SQL **SELECT**, которые извлекают информацию. Данные в коллекции можно изменять, после чего сделать так, чтобы entity-модель генерировала и выполняла соответствующие SQL-команды **INSERT**, **UPDATE** и **DELETE** для проведения эквивалентных операций в базе данных. Короче говоря, среда Entity Framework является отличным механизмом для подключения к базе данных, извлечения из нее данных и управления данными, не требующим встраивания в ваш код SQL-команд.

В следующем упражнении вы создадите очень простую entity-модель для таблицы *Customer*, находящейся в базе данных AdventureWorks. При этом вы примените подход к entity-моделированию, названный *database-first*, при котором приоритет в формировании структуры отдается базе данных. Он предполагает создание средой Entity Framework классов на основе определений таблиц в базе данных. Среда Entity Framework также предоставляет подход, где приоритет отдается коду, и в базе данных могут создаваться наборы таблиц на основе классов, реализованных вами в приложении.



ПРИМЕЧАНИЕ Если понадобятся дополнительные сведения о подходе, при котором для создания entity-модели приоритет отдается коду, обратитесь к статье «Code First to an Existing Database» на веб-сайте компании Microsoft, находящейся по адресу <http://msdn.microsoft.com/data/jj200620>.

Создание entity-модели AdventureWorks

Откройте в среде Visual Studio проект *Customers*, находящийся в папке \Microsoft Press\VCSBS\Chapter 27\Web Service вашей папки документов.

Этот проект содержит измененную версию приложения *Customers* из главы 26. В модели представления реализованы дополнительные команды, позволяющие пользователю переходить к первому и последнему клиенту в коллекции клиентов, и панель команд, содержащая кнопки *First* и *Last*, вызывающие эти команды.

В обозревателе решений щелкните правой кнопкой мыши на решении *Customers* (но не на проекте *Customers*), укажите на пункт *Добавить*, а затем щелкните на пункте *Создать проект*.

В левой панели диалогового окна *Добавить новый проект* щелкните на узле *Веб*. В средней панели щелкните на шаблоне *Веб-приложение ASP.NET*. Убедитесь, что в раскрывающемся списке над средней панелью указывается версия .NET Framework 4.6 (или при необходимости внесите соответствующие изменения). Наберите в поле *Имя* строку *AdventureWorksService* и щелкните на кнопке *OK*.

В диалоговом окне *Новый проект ASP.NET — AdventureWorksService* щелкните под надписью *Выбор шаблона* в разделе *Шаблоны ASP.NET 4.6* на значке *Web API*, а затем на кнопке *Изменить способ проверки подлинности*.

В диалоговом окне *Изменить способ проверки подлинности* выберите пункт *без проверки подлинности* и щелкните на кнопке *OK*, чтобы вернуться в диалоговое окно *Новый проект ASP.NET*.

Убедитесь, что в диалоговом окне *Новый проект ASP.NET* снят флажок *Host In The Cloud* (сначала вы протестируете *Web API* локально, а затем развернете его в облаке), после чего щелкните на кнопке *OK* (рис. 27.8).



ПРИМЕЧАНИЕ Шаблон ASP.NET Web API может создать код для обработки регистрации пользователей, но Azure предоставляет собственную масштабируемую систему идентификации и управления доступом, которая предпочтительнее для встраивания системы управления безопасностью в ваш код, если вы развертываете сервис в облаке. Система безопасности Azure в данной книге не рассматривается, но дополнительные сведения можно получить в статье «Azure Active Directory» по адресу <http://azure.microsoft.com/services/active-directory/> и в статье «Многофакторная проверка подлинности» по адресу <http://azure.microsoft.com/services/multi-factor-authentication/>.

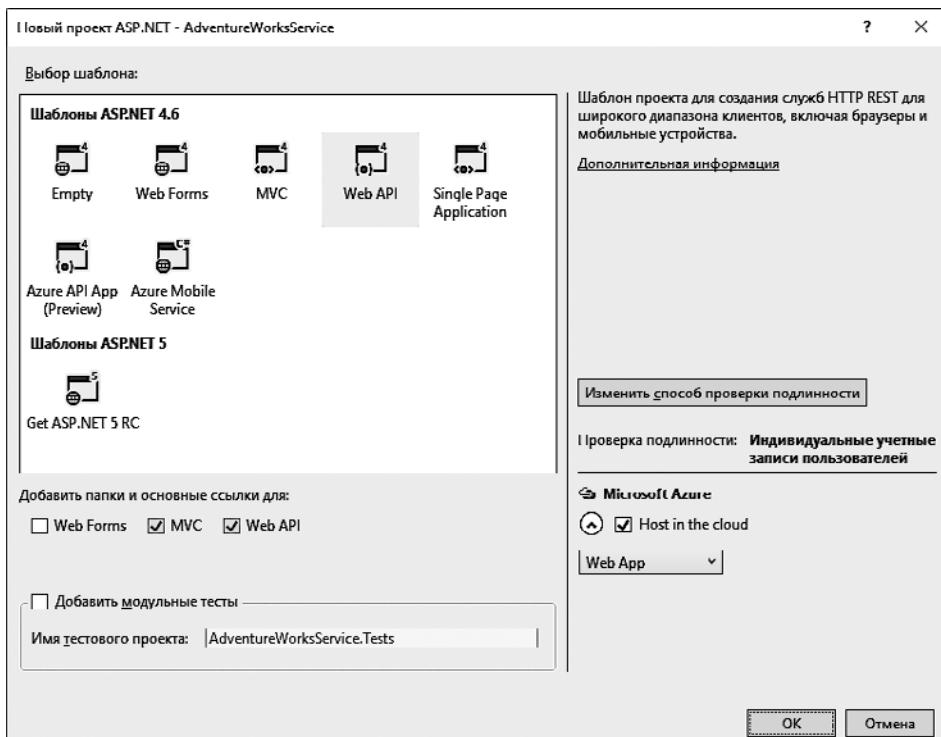


Рис. 27.8

Как упоминалось в начале главы, получить непосредственный доступ к реляционной базе данных из UWP-приложения невозможно, даже если используется среда Entity Framework. Для этого нужно создать веб-приложение (которое не является UWP-приложением) и разместить в нем создаваемую entity-модель. Шаблон Web API предоставляет мастера и инструменты, с помощью которых можно быстро создать веб-сервис, чем вы и займитесь в следующем упражнении. Этот веб-сервис предоставит удаленный доступ к entity-модели для UWP-приложения Customers.

В обозревателе решений щелкните правой кнопкой мыши на решении *Customers*, а затем на пункте **Назначить запускаемые проекты**.

В диалоговом окне **Страницы свойств Решение «Customers»** щелкните на пункте **Несколько запускаемых проектов**. Установите для проекта *AdventureWorksService* действие **Запуск без отладки**, а для проекта *Customers* — **Запуск**, после чего щелкните на кнопке **OK**.

Такая конфигурация гарантирует, что веб-приложение *AdventureWorksService* запустится, как только произойдет запуск проекта из меню **Отладка**. В обозревателе решений щелкните правой кнопкой мыши на проекте *AdventureWorksService*, а затем на пункте **Свойства**.

На странице свойств щелкните на вкладке **Веб**, находящейся в левом столбце.

На странице **Веб** щелкните на пункте **Не открывать страницу**. Дождаться запроса от внешней программы.

Обычно при запуске веб-приложения из среды Visual Studio открывается веб-браузер (Microsoft Edge), который пытается отобразить главную страницу приложения. Но у приложения *AdventureWorksService* нет главной страницы, поскольку оно предназначено для содержания веб-сервиса, к которому клиентское приложение может подключиться, чтобы извлечь данные из базы данных *AdventureWorks*.

В поле **URL-адрес проекта** измените адрес веб-приложения на **http://localhost:50000/**, а затем щелкните на кнопке **Создать виртуальный каталог**. Убедитесь в появлении окна среды Microsoft с сообщением об успешном создании виртуального каталога, после чего щелкните на кнопке **OK**.

По умолчанию шаблон проекта ASP.NET создает веб-приложение, размещенное с помощью IIS Express, и выбирает для URL-адреса произвольный порт. Данная конфигурация устанавливает для порта значение 50000, чтобы было легче дать описание следующих этапов выполнения упражнений в этой главе.

В меню **Файл** щелкните на пункте **Сохранить все**, а затем закройте страницу **Свойства**.

В обозревателе решений щелкните правой кнопкой мыши на папке *Models*, которая находится в проекте *AdventureWorksService*, укажите на пункт **Добавить**, а затем щелкните на пункте **Создать элемент**.

В левой панели диалогового окна **Добавить новый элемент** — *AdventureWorksService* щелкните на пункте **Данные**. В средней панели щелкните на шаблоне **Модель ADO.NET EDM**. В поле **Имя** наберите строку *AdventureWorksModel1*, а затем щелкните на кнопке **Добавить**. Запустится мастер моделей EDM. Этим мастером можно воспользоваться для создания entity-модели из существующей базы данных.

В разделе **Что должна содержать модель?** страницы мастера щелкните на пункте **Конструктор EF из базы данных**, а потом на кнопке **Далее**.

На странице Выбор подключения к данным щелкните на кнопке Создать соединение. Если появится диалоговое окно Выбор источника данных, выберите пункт Microsoft SQL Server, после чего щелкните на кнопке Продолжить.



ПРИМЕЧАНИЕ Диалоговое окно Выбор источника данных появляется только в том случае, если вы раньше не пользовались мастером подключения к данным и не выбирайали источник данных.

В поле Имя сервера диалогового окна Свойства подключения наберите строку `tcp:<servername>.database.windows.net,1433`, где `<servername>` является уникальным именем сервера базы данных Azure SQL, созданным вами в предыдущем упражнении. Щелкните на пункте Использовать аутентификацию SQL Server и введите имя и пароль, указанные вами для регистрации администратора в предыдущем упражнении. В поле Выберите или введите имя базы данных наберите строку `AdventureWorks`, а затем щелкните на кнопке OK (рис. 27.9).

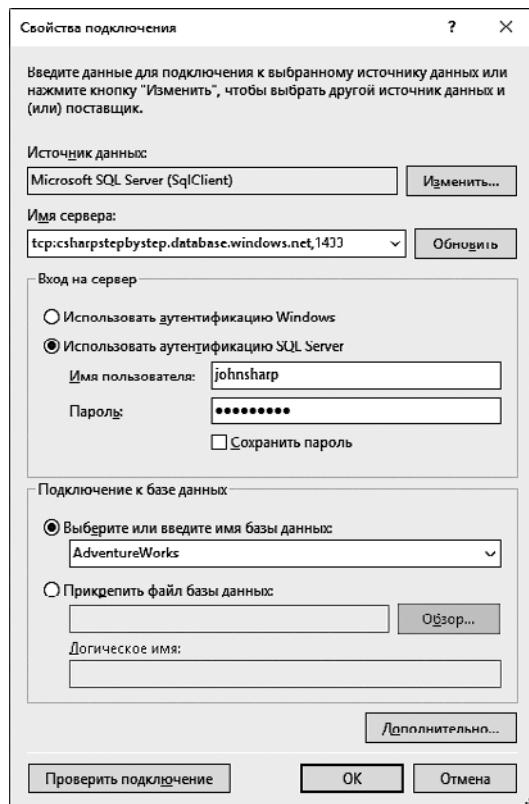


Рис. 27.9

В результате будет создано подключение к базе данных AdventureWorks, запущенной в облаке.

На странице Выбор подключения к данным щелкните на пункте Нет, исключить конфиденциальные данные из строки подключения. Они будут заданы в коде приложения. Убедитесь в том, что установлен флажок Сохранить параметры соединения в Web.Config как:, а в качестве имени строки соединения указана строка AdventureWorksEntities. Щелкните на кнопке Далее (рис. 27.10).

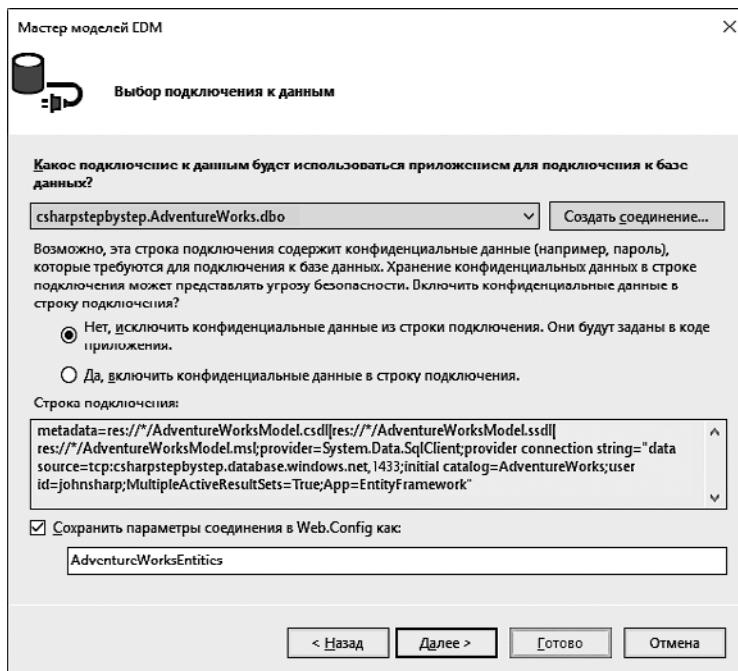


Рис. 27.10

На странице Выберите версию выберите пункт Entity Framework 6.x, а затем щелкните на кнопке Далее.

На странице Выберите параметры и объекты базы данных раскройте пункт Таблицы, далее пункт SalesLT, а затем выберите пункт Customer. Убедитесь в том, что установлен флажок Формировать имена объектов во множественном или единственном числе. (Два остальных флажка также должны быть изначально установлены.) Заметьте, что среда Entity Framework генерирует классы для entity-модели в пространстве имен AdventureWorksModel, а затем щелкните на кнопке Готово (рис. 27.11).

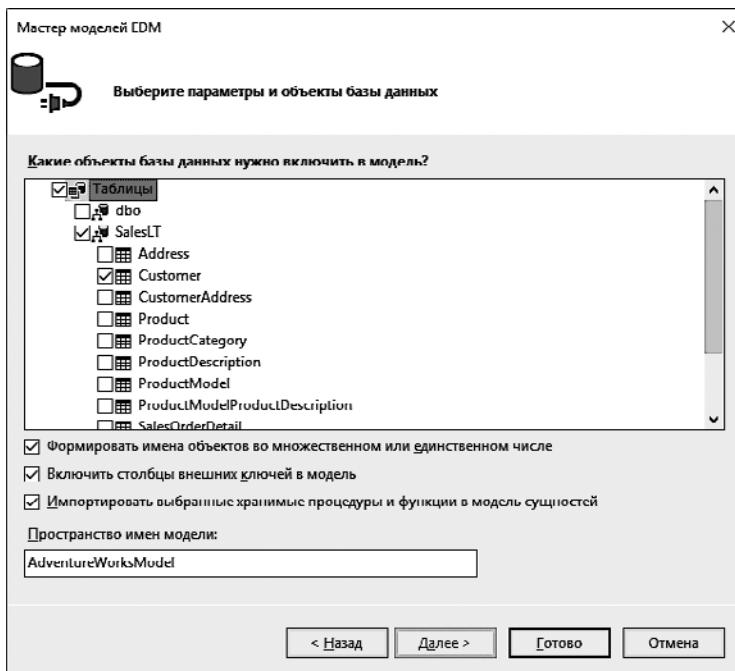


Рис. 27.11

Мастер моделей EDM создаст модель для таблицы Customer и покажет графическое представление в редакторе Entity Model (рис. 27.12).

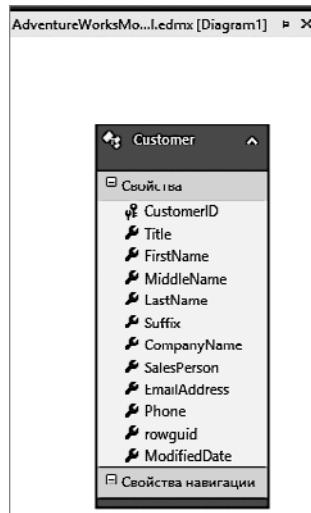


Рис. 27.12

Если появится окно Предупреждение о безопасности, установите флажок Больше не выводить это сообщение, а затем щелкните на кнопке OK. Данное окно предупреждения о безопасности появляется из-за того, что среда Entity Framework использует для создания кода вашей entity-модели технологию, известную как шаблоны T4, и загружает эти шаблоны из сети, используя NuGet. Шаблоны среди Entity Framework были проверены компанией Microsoft: пользоваться ими не опасно (рис. 27.13).

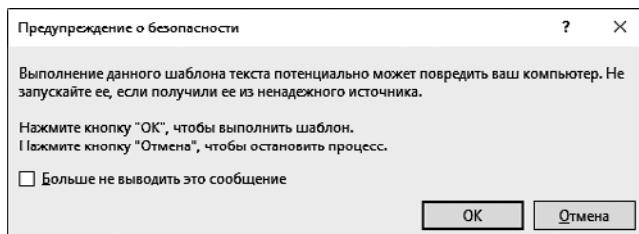


Рис. 27.13

В редакторе Entity Model щелкните правой кнопкой мыши на столбце MiddleName, после чего выберите пункт Удалить из модели. Точно так же удалите из entity-модели столбцы Suffix, CompanyName и SalesPerson.

Приложение Customers эти столбцы не использует, и в их извлечении из базы данных нет никакой необходимости. Они могут содержать null-значения, поэтому их можно, не опасаясь негативных последствий, оставить в качестве части таблицы базы данных. А вот столбцы rowguid и ModifiedDate удалять нельзя, поскольку они используются базой данных для идентификации строк в таблице Customer и для отслеживания изменений в этих строках в многопользовательской среде. Если эти столбцы удалить, вы не сможете корректно сохранять измененные данные в базе данных.

В меню Сборка щелкните на пункте Собрать решение.

В обозревателе решений в проекте AdventureWorksService откройте папку Models, раскройте запись AdventureWorksModel.edmx, затем запись AdventureWorksModel.tt и дважды щелкните на файле Customer.cs.

В этом файле содержится класс, созданный мастером моделей EDM для представления сведений о клиенте. В этом классе содержатся автоматически созданные свойства для каждого столбца, имеющегося в таблице Customer, включенного вами в entity-модель:

```
public partial class Customer
{
    public int CustomerID { get; set; }
    public string Title { get; set; }
```

```
public string FirstName { get; set; }
public string LastName { get; set; }
public string EmailAddress { get; set; }
public string Phone { get; set; }
public System.Guid rowguid { get; set; }
public System.DateTime ModifiedDate { get; set; }
}
```

В обозревателе решений ниже записи AdventureWorksModel.edmx раскроите запись AdventureWorksModel.Context.tt, а затем дважды щелкните на файле AdventureWorksModel.Context.cs. В этом файле содержится определение класса по имени AdventureWorksEntities. (Точно такое же имя использовалось вами при создании подключения к базе данных в мастере моделей EDM.)

```
public partial class AdventureWorksEntities : DbContext
{
    public AdventureWorksEntities()
        : base("name=AdventureWorksEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public DbSet<Customer> Customers { get; set; }
}
```

Класс AdventureWorksEntities является производным от класса DbContext, а этот класс предоставляет функциональные средства, используемые приложением для подключения к базе данных. Пассивный конструктор передает конструктору базового класса параметр, указывающий имя строки подключения для подключения к базе данных. Если посмотреть на содержимое файла web.config, то эту строку можно найти в разделе <ConnectionStrings>. В ней, кроме всего прочего, содержатся параметры, которые вы указываете при запуске мастера моделей EDM. Но эта строка не содержит информации о пароле, требующемся для аутентификации подключения, — вы выбрали вариант указания соответствующих данных в ходе выполнения программы. Решением данного вопроса займитесь, выполняя следующие задания упражнения.

Метод OnModelCreating в классе AdventureWorksEntities можно проигнорировать. В коллекции Customers имеется единственный элемент. Эта коллекция относится к типу DbSet<Customer>. Тип-обобщение DbSet предоставляет методы, с помощью которых вы можете добавлять, вставлять, удалять и запрашивать объекты в базе данных. Для создания соответствующих SQL-команд SELECT, необходимых для извлечения клиентской информации из базы данных и заполнения коллекции, он работает в союзе с классом DbContext. Он также используется для создания

SQL-команд `INSERT`, `UPDATE` и `DELETE`, запускаемых, если `Customer`-объекты добавляются, изменяются или удаляются из коллекции. Коллекция `DbSet` часто упоминается как entity-набор.

Щелкните в обозревателе решений правой кнопкой мыши на папке `Models`, на пункте **Добавить**, а затем на пункте **Класс**.

Убедитесь, что в диалоговом окне **Добавить новый элемент** — `AdventureWorksService` выбран шаблон **Класс**. Наберите в поле **Имя** строку `AdventureWorksEntities`, а затем щелкните на кнопке **Добавить**.

К проекту будет добавлен новый класс по имени `AdventureWorksEntities`, и его код будет выведен в окно редактора. На данный момент этот класс конфликтует с уже существующим классом с таким же именем, который был создан средой Entity Framework, но вы этот класс будете использовать в качестве расширения кода среды Entity Framework путем его преобразования в частичный класс (partial class). Частичным называется класс, в котором код разделен на несколько исходных файлов. Такой подход применяется для таких инструментальных средств, как среда Entity Framework, поскольку он позволяет вам добавлять собственный код без риска его случайной перезаписи в случае какого-либо преобразования среды Entity Framework в будущем.

Измените в окне редактора определение класса `AdventureWorksEntities`, объявив его частичным.

```
public partial class AdventureWorksEntities
{
}
```

Добавьте к классу `AdventureWorksEntities` конструктор, получающий строковый параметр по имени `password`. Конструктор должен вызвать конструктор базового класса с передачей ему имени строки подключения, ранее записанной в файл `web.config` мастером моделей EDM и указанной на странице **Выбор подключения** к данным:

```
public partial class AdventureWorksEntities
{
    public AdventureWorksEntities(string password)
        : base("name=AdventureWorksEntities")
    {
    }
}
```

Добавьте к конструктору код, показанный далее жирным шрифтом. Этот код изменяет строку подключения, используемую средой Entity Framework, с целью включения в нее пароля. Приложение `Customers` будет вызывать этот конструктор и предоставлять пароль в ходе своего выполнения:

```
public partial class AdventureWorksEntities
{
    public AdventureWorksEntities(string password)
        : base("name=AdventureWorksEntities")
    {
        this.Database.Connection.ConnectionString += $""; Password={password}";
    }
}
```

Создание и использование веб-сервиса REST

Вы создали entity-модель, предоставляющую операции для извлечения клиентской информации и ее обработки. Следующим этапом станет создание веб-сервиса, позволяющего UWP-приложению получить доступ к entity-модели.

С помощью среды Visual Studio 2015 можно создать модель веб-сервиса в веб-приложении ASP.NET, основываясь непосредственно на entity-модели, созданной средой Entity Framework. Веб-сервис использует entity-модель для извлечения данных из базы данных и обновления этой базы данных. Вы будете создавать веб-сервис, используя мастер под названием Добавление шаблона. Этот мастер может создать веб-сервис, реализующий REST-модель, которая использует схему навигации для представления бизнес-объектов и сервисов по сети, а также HTTP-протокол для передачи запросов на доступ к этим объектам и сервисам. Клиентское приложение, обращающееся к ресурсу, отправляет запрос в форме URL-адреса, который анализируется и обрабатывается веб-сервисом. Например, компания Adventure Works может опубликовать информацию о клиентах в виде единого ресурса, воспользовавшись схемой, похожей на следующую:

<http://Adventure-Works.com/DataService/Customers/1>

Обращение по этому URL-адресу заставляет веб-сервис извлечь данные для клиента № 1. Эти данные могут быть возвращены в нескольких форматах, но для обеспечения переносимости наиболее часто используемыми форматами являются XML и JavaScript Object Notation (JSON). Созданный веб-сервисом с REST-моделью типовой JSON-ответ на предыдущий запрос выглядит следующим образом:

```
{
    "CustomerID":1,
    "Title":"Mr",
    "FirstName":"Orlando",
    "LastName":"Gee",
    "EmailAddress":"orlando0@adventure-works.com",
    "Phone":"245-555-0173"
}
```

REST-модель зависит от приложения, обращающегося к данным, и от отправки им соответствующего HTTP-глагола в качестве части запроса на доступ к данным. Например, простой запрос, показанный ранее, должен вызвать отправку HTTP GET-запроса к веб-сервису. В HTTP поддерживаются и другие глаголы, например POST, PUT и DELETE, которыми можно воспользоваться для создания, изменения и удаления ресурсов соответственно. Написание кода для генерирования соответствующих HTTP-запросов и синтаксического разбора ответов, возвращенных веб-сервисом REST, представляется весьма непростой задачей. К счастью, создание основного объема этого кода мастер Добавление шаблона может взять на себя.

В следующем упражнении вы создадите для entity-модели простой REST веб-сервис. Он откроет для клиентского приложения возможность запрашивать информацию о клиентах и распоряжаться ею.

Создание веб-сервиса AdventureWorks

В среде Visual Studio, в проекте AdventureWorksService щелкните правой кнопкой мыши на папке Controllers, укажите на пункте Добавить, а затем щелкните на пункте Создать шаблонный элемент.

В средней панели мастера Добавление шаблона щелкните на шаблоне Контроллер Web API 2 с действиями, использующий Entity Framework, а затем на кнопке Добавить (рис. 27.14).

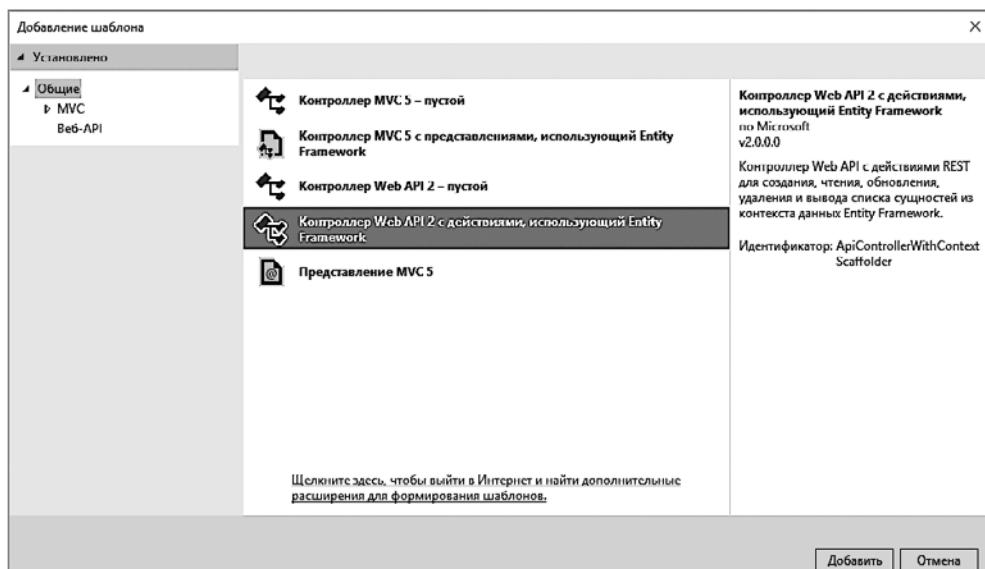


Рис. 27.14

В диалоговом окне Добавить контроллер в раскрывающемся списке Класс модели выберите элемент Customer (AdventureWorksService.Models). Выберите в раскрывающемся списке Класс контекста данных элемент AdventureWorksEntities (AdventureWorksService.Models). Установите флажок Использование асинхронных действий контроллера. Убедитесь в том, что в поле Имя контроллера показано имя CustomersController, а затем щелкните на кнопке Добавить (рис. 27.15).

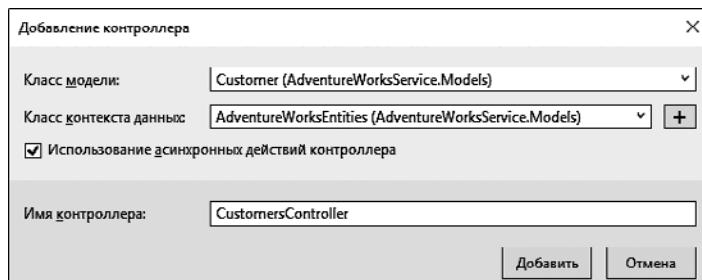


Рис. 27.15

В веб-сервисе, созданном путем использования шаблона ASP.NET Web API, все поступающие веб-запросы обрабатываются одним или несколькими классами контроллера и каждый класс контроллера предоставляет методы, отображающие различные типы REST-запросов для каждого ресурса, предоставляемого контроллером. Например, CustomersController имеет следующий вид:

```
public class CustomersController : ApiController
{
    private AdventureWorksEntities db = new AdventureWorksEntities();

    // GET: api/Customers
    public IQueryable<Customer> GetCustomers()
    {
        return db.Customers;
    }

    // GET: api/Customers/5
    [ResponseType(typeof(Customer))]
    public async Task<IHttpActionResult> GetCustomer(int id)
    {
        Customer customer = await db.Customers.FindAsync(id);
        if (customer == null)
        {
            return NotFound();
        }

        return OK(customer);
    }

    // PUT: api/Customers/5
```

```
[ResponseType(typeof(void))]
public async Task<IHttpActionResult> PutCustomer(int id, Customer customer)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    if (id != customer.CustomerID)
    {
        return BadRequest();
    }

    db.Entry(customer).State = EntityState.Modified;

    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!CustomerExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
}

return StatusCode(HttpStatusCode.NoContent);
}

// POST: api/Customers
[ResponseType(typeof(Customer))]
public async Task<IHttpActionResult> PostCustomer(Customer customer)
{
    ...
}

// DELETE: api/Customers/5
[ResponseType(typeof(Customer))]
public async Task<IHttpActionResult> DeleteCustomer(int id)
{
    ...
}
...
```

Метод `GetCustomers` обрабатывает запросы на извлечение сведений обо всех клиентах и удовлетворяет эти запросы простым возвращением всей коллекции `Customers` из ранее созданной средой Entity Framework модели данных. Сама среда Entity Framework извлекает все сведения о клиентах из базы данных

и использует эту информацию для заполнения коллекции `Customers`. Этот метод вызывается, если приложение отправляет HTTP GET-запрос к этому веб-сервису по URL-адресу `api/Customers`.

Метод `GetCustomer` (не перепутайте его с `GetCustomers`) получает целочисленный параметр. Этот параметр указывает значение идентификатора `CustomerID` конкретного клиента, и метод использует среду Entity Framework для поиска сведений об этом клиенте перед их возвращением. `GetCustomer` запускается, когда приложение отправляет HTTP GET-запрос по URL-адресу `api/Customers/n`, где `n` — идентификатор клиента, сведения о котором должны быть извлечены.

Метод `PutCustomer` запускается, когда приложение отправляет веб-сервису HTTP PUT-запрос. В запросе указываются идентификатор клиента и сведения о нем, а код в этом методе использует среду Entity Framework для обновления сведений о конкретном клиенте. Метод `PostCustomer` откликается на HTTP POST-запросы и получает в качестве своего параметра сведения о клиенте. Этот метод добавляет нового клиента с этими сведениями к базе данных (сведения в предыдущем примере кода не показаны). И наконец, метод `DeleteCustomer` обрабатывает HTTP DELETE-запросы и удаляет сведения о клиенте с конкретно указанным идентификатором.



ПРИМЕЧАНИЕ В коде, созданном шаблоном Web API, оптимистично предполагается, что у него всегда будет возможность подключиться к базе данных. В мире распределенных систем, где база данных и веб-сервис расположены на разных серверах, такая возможность может представиться не всегда. В сетях бывают ошибки переходных процессов и задержки, попытки подключения могут проваливаться из-за временных сбоев и чуть позже удаваться при повторном обращении. Сообщение клиенту о временном сбое как об ошибке может стать неприятным сюрпризом для пользователя. Лучше будет по возможности молча повторить неудавшуюся операцию при условии, что число попыток не превысило определенный порог, поскольку замораживать веб-сервис в случае реальной недоступности базы данных нежелательно. Подробные сведения об этой стратегии можно найти в статье «Cloud Service Fundamentals Data Access Layer—Transient Fault Handling» по адресу <http://social.technet.microsoft.com/wiki/contents/articles/18665.cloud-service-fundamentals-data-access-layer-transient-fault-handling.aspx>.

Шаблон ASP.NET Web API автоматически создает код, направляющий запросы к соответствующему методу в классах контроллеров, и если нужно управлять другими ресурсами, например товарами или заказами, вы можете добавить дополнительные классы контроллеров.



ПРИМЕЧАНИЕ Дополнительные сведения о реализации веб-сервисов REST с использованием шаблона ASP.NET Web API можно найти в статье «Web API» по адресу <http://www.asp.net/web-api>.

Можно также создавать классы контроллеров вручную, воспользовавшись той же самой схемой, которая была показана для класса `CustomersController`, — вам необязательно извлекать данные и сохранять их в базе данных путем использования среди Entity Framework. Шаблон ASP.NET Web API содержит в файле `ValuesController.cs` демонстрационный контроллер, который вы можете скопировать и дополнить собственным кодом.

Измените в классе `CustomersController` инструкцию, которая создает объект контекста `AdventureWorksEntities` для использования конструктора, получающего пароль в качестве своего параметра. В качестве аргумента конструктора укажите пароль администратора, который был вами задан при создании базы данных. (В следующем примере кода замените строку `YourPassword` своим паролем.)

```
public class CustomersController : ApiController
{
    private AdventureWorksEntities db = new
        AdventureWorksEntities("YourPassword");

    // GET: api/Customers
    public IQueryable<Customer> GetCustomers()
    {
        return db.Customers;
    }
    ...
}
```



ПРИМЕЧАНИЕ В реальном мире вы могли бы предложить пользователю, запустившему приложение `Customers`, ввести эту информацию или позволить приложению сохранить пароль локально на устройстве пользователя в виде части конфигурационных данных приложения. Затем приложение `Customers` передало бы эту информацию веб-сервису.

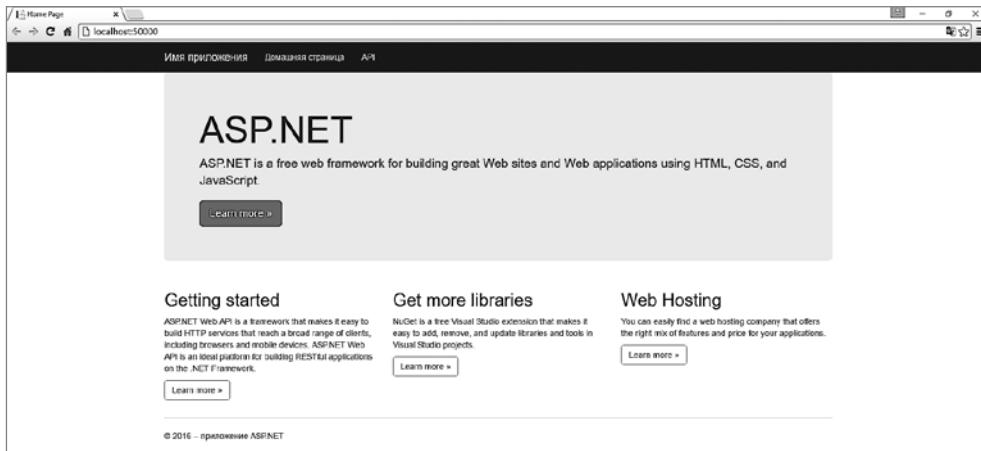
В папке `Controllers` щелкните правой кнопкой мыши на файле `ValuesController.cs`, а затем щелкните на пункте Удалить. В окне сообщения щелкните на кнопке OK, подтверждая намерение удалить этот файл.

В этом приложении пример класса `ValuesController` использоваться не будет.

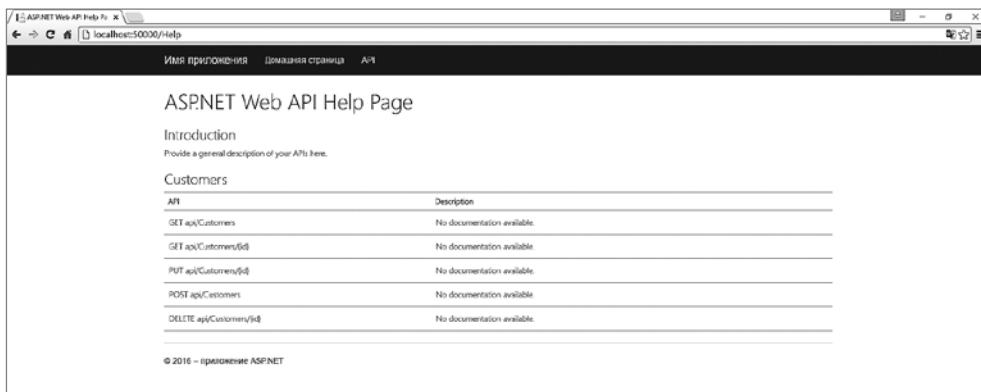


ПРИМЕЧАНИЕ Не удаляйте класс `HomeController`. Его контроллер действует в качестве точки входа в веб-приложение, размещающее веб-сервис.

Щелкните правой кнопкой мыши на проекте `AdventureWorksService`, щелкните на пункте Представление, а затем на пункте Просмотреть в браузере (Google Chrome). Если веб-сервис настроен правильно, веб-браузер запустится и покажет следующую страницу (рис. 27.16).

**Рис. 27.16**

В панели заголовков щелкните на пункте API. Появится еще одна страница, содержащая сводку REST-запросов, которые приложение может отправлять веб-сервису (рис. 27.17).

**Рис. 27.17**

Наберите в адресной строке адрес `http://localhost:50000/api/Customers/1`, а затем нажмите Ввод.

Этот запрос направлен перегруженному методу `GetCustomer` в классе `CustomersController`, а значение 1 передано этому методу в качестве параметра. Заметьте, что Web API использует маршруты, которые начинаются с адреса сервера, после которого указывается путь `api`.

Метод `GetCustomer` извлечет сведения о клиенте из базы данных и вернет их в объекте, имеющем JSON-формат, который отобразится в браузере. Данные будут выглядеть следующим образом (значение для `ModifiedDate` может отличаться от того, которое показано далее):

```
{"CustomerID":1,"Title":"Mr","FirstName":"Orlando","LastName":"Gee","EmailAddres": "orlando0@adventure-works.com","Phone":"245-555-0173","rowguid": "3f5ae95e-b87d-4aed-95b4-c3797afcb74f","ModifiedDate":"2001-08-01T00:00:00"}
```

Закройте веб-браузер и вернитесь в среду Visual Studio.

Теперь вместо того, чтобы запускать веб-сервис локально, вы развернете его в облаке Azure. Это можно сделать с помощью мастера публикации веб-сайта, доступного в Visual Studio 2015 для создания веб-приложения в облаке и выкладывания веб-сервиса в это приложение.

Развертывание веб-сервиса в облаке

В обозревателе решений щелкните правой кнопкой мыши на проекте `AdventureWorksService`, а затем щелкните на пункте Опубликовать. Запустится мастер публикации веб-сайта (рис. 27.18).

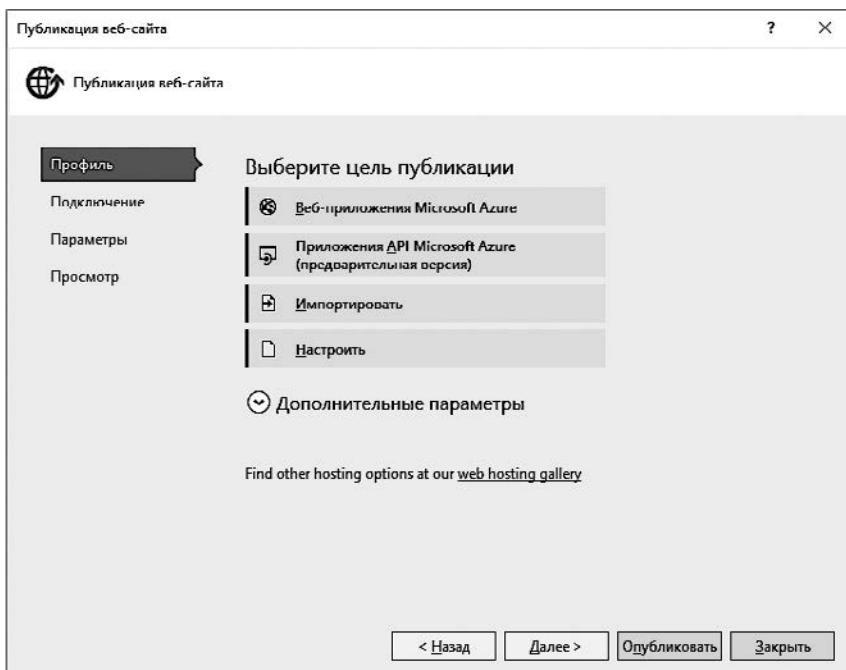


Рис. 27.18

В области Выберите цель публикации щелкните на пункте Веб-приложения Microsoft Azure.

Выберите в диалоговом окне свою учетную запись в Azure. Если вместо списка учетных записей появится надпись Добавить учетную запись, щелкните на ней, а затем на пункте Добавить учетную запись. В диалоговом окне регистрации в Visual Studio зарегистрируйтесь с использованием своей учетной записи в Microsoft.

В области Существующие веб-приложения щелкните на кнопке New.

В диалоговом окне Create Web App On Microsoft Azure дайте уникальное имя веб-приложению и укажите для его размещения ближайший к вам регион. Не указывайте сервер базы данных (о подключении к базе данных позаботится код среди Entity Framework в вашем веб-сервисе). В поле сервисного плана приложения App Service Plan щелкните на пункте создания нового сервисного плана Create New App Service Plan и дайте ему имя по своему выбору. Аналогично этому в поле группы ресурсов Resource group щелкните на пункте создания новой группы ресурсов Create New Resource Group и дайте имя этой группе. После введения всех данных щелкните на кнопке Create (Создать) (рис. 27.19).

На странице Подключение мастера публикации веб-приложения примите предлагаемые значения по умолчанию, щелкните на кнопке проверки подключения



Рис. 27.19

Validate connection, чтобы проверить, что веб-приложение, в котором размещается веб-сервис, было успешно создано, после чего щелкните на кнопке Далее.

На странице Параметры примите значения по умолчанию, а затем щелкните на кнопке Далее.

На странице Просмотр щелкните на кнопке Опубликовать.

Веб-сервис должен быть выложен в сеть. Должен открыться браузер и показать домашнюю страницу сайта, которую вы уже видели, когда выполняли локальный запуск сервиса.

Закройте веб-браузер и вернитесь в среду Visual Studio.

Следующий этап нашего путешествия предполагает подключение к веб-сервису из UWP-приложения *Customers* с использованием веб-сервиса для извлечения данных. Среда .NET Framework предоставляет класс `HttpClient`, который приложение может использовать для составления и отправки к веб-сервису запросов HTTP REST, и класс `HttpResponseMessage`, который приложение может использовать для обработки результата, полученного от веб-сервиса. Эти классы позволяют отделить от вашего кода подробности HTTP-протокола. Благодаря этому вы можете сконцентрироваться на бизнес-логике, отображающей объекты, опубликованные через веб-сервис, и манипулирующей ими. Вы воспользуетесь этими классами в следующем упражнении. Вами также будет использован JSON-парсер, реализованный в пакете `Json.NET`, поэтому придется добавить этот пакет к проекту *Customers*.



ВНИМАНИЕ В этом упражнении извлекаются сведения о каждом клиенте. Оно полезно в качестве прототипа, подтверждающего концепцию извлечения данных посредством веб-сервиса, размещенного в облаке, но в реальности вы должны подходить к извлечению данных более избирательно. Если база данных содержит большой объем данных, этот подход может стать весьма расточительным в смысле использования сетевого трафика и требований, предъявляемых к объемам памяти со стороны приложения, запущенного на устройстве пользователя. Более рациональным подходом будет разбиение на страницы, при котором сведения о клиентах извлекаются в поблочном режиме (возможно, блоками, содержащими сведения о 20 клиентах). Для поддержки такого подхода веб-сервису понадобится обновление, а модели представления (`ViewModel`) в приложении *Customers* потребуется управлять извлекаемыми блоками явным образом. Пусть разработка этого подхода станет для читателя заданием для самостоятельного выполнения.

Извлечение данных из веб-сервиса AdventureWorks

В обозревателе решений щелкните правой кнопкой мыши на файле `DataSource.cs` проекта *Customers*, а затем щелкните на пункте Удалить. Щелкните в окне сообщения на кнопке OK, чтобы подтвердить намерение удалить файл.

В этом файле содержатся демонстрационные данные, используемые приложением Customers. Вы собираетесь внести изменения в класс `ViewModel`, чтобы извлекать эти данные из веб-сервиса, следовательно, надобность в этом файле исчезла.

Щелкните правой кнопкой мыши на проекте `Customers`, а затем щелкните на пункте Управление пакетами NuGet.



ПРИМЕЧАНИЕ Если будет предложено ввести пароль для подключения к SQL Server, закройте диалоговое окно. Информация о пароле уже предоставлена веб-сервисом.

Убедитесь в том, что в окне Диспетчер пакетов NuGet: `Customers` в поле со списком Filter показано значение All, а затем наберите в поле поиска строку `Json.NET`.

Выберите в панели, показывающей результаты поиска, пакет `Newtonsoft.Json`. В правой панели настройте действие на установку пакета и щелкните на кнопке Установить.

Щелкните в окне Просмотр на кнопке OK.

Дождитесь установки пакета и закройте окно Диспетчер пакетов NuGet: `Customers`.

В обозревателе решений дважды щелкните на файле `ViewModel.cs`, чтобы его код появился в окне редактора.

Добавьте к списку в начале файла следующие директивы `using`:

```
using System.Net.Http;
using System.Net.Http.Headers;
using Newtonsoft.Json;
```

Добавьте к классу `ViewModel` следующие переменные, показанные жирным шрифтом, поставив их перед конструктором `ViewModel`. Поставьте вместо `<webappname>` имя веб-приложения, созданного вами в предыдущем упражнении, чтобы в нем содержался веб-сервис:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    private const string ServerUrl = "http://<webappname>.azurewebsites.net/";
    private HttpClient client = null;

    public ViewModel()
    {
        ...
    }
    ...
}
```

В переменной `ServerUrl` содержится базовый адрес веб-сервиса. А переменную `client` вы будете использовать для подключения к веб-сервису.

В конструкторе `ViewModel` инициализируйте список `customers` значением `null` и добавьте для настройки переменной `client` следующие инструкции, показанные жирным шрифтом:

```
public ViewModel()
{
    ...
    this.customers = null;
    this.client = new HttpClient();
    this.client.BaseAddress = new Uri(ServerUrl);
    this.client.DefaultRequestHeaders.Accept =
        Add(new MediaTypeWithQualityHeaderValue("application/json"));
}
```

В списке `customers` содержатся сведения о клиентах, которые показывает приложение, — ранее он был заполнен данными, содержащимися в файле `DataSource.cs`, который вы удалили.

Переменная `client` инициализируется адресом веб-сервера, к которому будут отправляться запросы. REST веб-сервис может получать запросы и отправлять ответы в различных форматах, но в приложении `Customers` будет использоваться формат JSON. Последняя инструкция в предыдущем фрагменте кода настраивает переменную `client` на отправку запросов именно в этом формате.

В обозревателе решений дважды щелкните на файле `Customer.cs`, хранящемся в корневой папке проекта `Customers`, чтобы он был выведен в окно редактора. Добавьте к классу `Customer` сразу же после свойства `Phone` следующие открытые свойства, показанные жирным шрифтом:

```
public class Customer : INotifyPropertyChanged
{
    ...
    public string Phone
    {
        ...
    }

    public System.Guid rowguid { get; set; }
    public System.DateTime ModifiedDate { get; set; }

    ...
}
```

Веб-сервис извлекает эти поля из базы данных, и приложение `Customers` должно быть готово к их обработке, в противном случае, если пользователь решит изменить сведения о клиенте, данные из этих полей будут утрачены.

(Возможность изменения данных будет добавлена к приложению Customers чуть позже.)

Вернитесь к классу ViewModel. Добавьте после конструктора показанный здесь открытый метод GetDataAsync:

```
public async Task GetDataAsync()
{
    try
    {
        var response = await this.client.GetAsync("api/customers");
        if (response.IsSuccessStatusCode)
        {
            var customerData =
                await response.Content.ReadAsStringAsync();
            this.customers =
                JsonConvert.DeserializeObject<List<Customer>>(customerData);
            this.currentCustomer = 0;
            this.OnPropertyChanged(nameof(Current));
            this.IsAtStart = true;
            this.IsAtEnd = (this.customers.Count == 0);
        }
        else
        {
            // TODO: Обработать сбой GET-запроса
        }
    }
    catch (Exception e)
    {
        // TODO: Обработать исключения
    }
}
```

Этот метод работает в асинхронном режиме, для вызова в веб-сервисе операции api/customers он использует метод GetAsync объекта HttpClient. С помощью этой операции из базы данных AdventureWorks извлекаются сведения о клиентах. Сам метод GetAsync также имеет асинхронную природу и возвращает объект HttpResponseMessage, заключенный в Task-объект. Объект HttpResponseMessage содержит код состояния, показывающий, увенчался ли запрос успехом, и если да, приложение для фактического извлечения данных, возвращенных веб-сервисом, использует метод ReadAsStringAsync, принадлежащий свойству Content, которое в свою очередь принадлежит объекту HttpResponseMessage. Эти данные содержат строку в JSON-формате, в которой имеются сведения о каждом клиенте, поэтому они преобразуются в список из Customer-объектов и присваиваются коллекции customers путем использования статического метода DeserializeObject, принадлежащего классу JsonConvert, который является частью пакета Json.NET. Затем в качестве значения свойства currentCustomer класса ViewModel устанавливается указатель на первого клиента в этой коллекции, а свойства IsAtStart и IsAtEnd инициализируются таким образом, чтобы показывать состояние ViewModel.



ПРИМЕЧАНИЕ Клиентские библиотеки ASP.NET Web API предполагают, что запуск всех запросов веб-службы может занимать неопределенное время. Соответственно, такие классы, как HttpClient и HttpResponseMessage, сконструированы под поддержку асинхронных операций, чтобы предотвратить блокировку приложения при ожидании ответа. Фактически, эти классы поддерживают только асинхронные операции — синхронных версий для GetAsync или ReadAsStringAsync не существует.

Заметьте, что на данный момент метод GetDataAsync не обрабатывает никаких исключений или отказов, он их просто молчаливо поглощает. Технология выдачи отчетов об исключениях в UWP-приложении будет показана чуть позже.



ПРИМЕЧАНИЕ В коммерческих приложениях из сети не должна извлекатьсяunnecessary information, и при их разработке следует проявлять избирательность в отношении извлекаемых данных. Но в данном приложении база данных AdventureWorks содержит всего лишь несколько сотен клиентов, поэтому извлекаются и кэшируются сведения обо всех клиентах, которые затем попадают в список клиентов.

Измените, как показано далее, метод доступа get свойства Current:

```
public Customer Current
{
    get
    {
        if (this.customers != null)
        {
            return this.customers[currentCustomer];
        }
        else
        {
            return null;
        }
    }
}
```

Метод GetDataAsync является асинхронным, следовательно, на момент попытки привязки к клиенту формы MainPage коллекция клиентов может быть еще не заполнена. В такой ситуации данное изменение не дает привязке данных выдавать исключения, связанные с отсутствием ссылки, при обращении к коллекции клиентов.

Обновите в конструкторе ViewModel условия, позволяющие запускаться каждой команде, внеся изменения, выделенные в следующем примере жирным шрифтом:

```
public ViewModel()
{
    ...
    this.NextCustomer = new Command(this.Next,
        () => { return this.customers != null &&
            this.customers.Count > 1 && !this.IsAtEnd; });
    this.PreviousCustomer = new Command(this.Previous,
```

```
() => { return this.customers != null &&
          this.customers.Count > 0 && !this.IsAtStart; });
this.FirstCustomer = new Command(this.First,
() => { return this.customers != null &&
          this.customers.Count > 0 && !this.IsAtStart; });
this.LastCustomer = new Command(this.Last,
() => { return this.customers != null &&
          this.customers.Count > 1 && !this.IsAtEnd; });
}
```

Эти изменения обеспечат недоступность кнопок в панели команд до тех пор, пока не появятся данные для отображения на странице.

В обозревателе решений раскройте элемент MainPage.xaml и дважды щелкните на файле MainPage.xaml.cs, чтобы он открылся в окне редактора.

Добавьте к конструктору MainPage следующую инструкцию, показанную жирным шрифтом:

```
public MainPage()
{
    ...
    ViewModel viewModel = new ViewModel();
    viewModel.GetDataAsync();
    this.DataContext = viewModel;
}
```

Эта инструкция заполнит данными ViewModel.

В меню Отладка щелкните на пункте Начать отладку, чтобы собрать и запустить приложение.

Сначала, пока будет работать метод `GetDataAsync`, появится пустая форма, но через несколько секунд будут показаны сведения о первом клиенте по имени Orlando Gee (рис. 27.20).

Воспользуйтесь кнопками навигации в панели команд для перемещения по списку клиентов, чтобы убедиться в том, что в работе формы нет никаких неожиданностей.

Вернитесь в среду Visual Studio и остановите отладку.

В качестве последнего штриха, добавляемого в этом разделе, будет полезно при начальном появлении формы оповестить пользователей о том, что, несмотря на ее пустой вид, приложение занимается извлечением данных. В UWP-приложении для создания этой обратной связи можно воспользоваться элементом управления типа `ProgressRing` (кольцевым индикатором занятости). Этот элемент управления должен выводиться на экран только в том случае, если объект типа `ViewModel` занят обменом данными с веб-сервером.



Рис. 27.20

Добавления к форме Customers индикатора занятости

В окне редактора откройте файл ViewModel.cs. Добавьте к классу ViewModel после метода GetDataAsync закрытое поле _isBusy и открытое свойство IsBusy:

```
private bool _isBusy;
public bool IsBusy
{
    get { return this._isBusy; }
    set
    {
        this._isBusy = value;
        this.OnPropertyChanged(nameof(IsBusy));
    }
}
```

Добавьте к методу GetDataAsync следующие инструкции, показанные жирным шрифтом:

```
public async Task GetDataAsync()
{
    try
    {
        this.IsBusy = true;
        var response = await this.client.GetAsync("api/customers");
        ...
    }
    catch (Exception e)
    {
        // TODO: Обработать исключения
    }
    finally
    {
        this.IsBusy = false;
    }
}
```

Перед тем как запустить запрос на извлечения сведений о клиентах, метод `GetData` устанавливает для свойства `IsBusy` значение `true`. Блок `finally` обеспечивает возвращение для свойства `IsBusy` значения `false` даже при условии выдачи исключения.

В окне конструктора откройте файл `MainPage.xaml`. Добавьте в XAML-панели элемент управления `ProgressRing`, показанный далее жирным шрифтом, сделав его первым элементом в элементе управления `Grid` верхнего уровня:

```
<Grid Style="{StaticResource GridStyle}">
    <ProgressRing HorizontalAlignment="Center"
    VerticalAlignment="Center" Foreground="AntiqueWhite"
    Height="100" Width="100" IsActive="{Binding IsBusy}"
    Canvas.ZIndex="1"/>
    <Grid x:Name="customersTabularView" Margin="40,104,0,0" ...>
        ...

```

Установите для свойства `Canvas.ZIndex` значение "1", обеспечивающее появление элемента `ProgressRing` поверх всех остальных элементов управления, отображаемых элементом управления `Grid`.

В меню Отладка щелкните на пункте Начать отладку для сборки и запуска приложения.

Заметьте, что при запуске приложения кольцевой индикатор занятости появляется на некоторое время, прежде чем на экране возникнут сведения о первом клиенте. Если окажется, что сведения о первом клиенте появляются слишком быстро, можно вставить в метод `GetDataAsync` небольшую задержку, просто чтобы удостовериться в том, что кольцевой индикатор занятости работает. Добавьте следующую инструкцию, которая поставит выполнение метода на пятисекундную паузу:

```
public async Task GetDataAsync()
{
    try
    {
        this.IsBusy = true;
        await Task.Delay(5000);
        var response = await this.client.GetAsync(...);
        ...
    }
    ...
}
```

Не забудьте удалить эту инструкцию, как только закончите тестирование кольцевого индикатора занятости.

Вернитесь в среду Visual Studio и остановите отладку.

Вставка, обновление и удаление данных через REST веб-сервис

Многие приложения наряду с возможностью запрашивать данные и выводить их на экран требуют дать пользователям возможность вставки, обновления и удаления информации. В ASP.NET Web API реализуется модель, поддерживающая эти операции посредством использования HTTP PUT-, POST- и DELETE-запросов. В соответствии с соглашениями PUT-запрос изменяет существующий ресурс в веб-сервисе, POST-запрос создает новый экземпляр ресурса, а DELETE-запрос удаляет ресурс. Код, генерируемый в шаблоне ASP.NET Web API мастером Добавление шаблона, придерживается именно этих трех соглашений.

ИДЕМПОТЕНТНОСТЬ В REST ВЕБ-СЕРВИСАХ

В REST веб-сервисе PUT-запросы должны быть идемпотентными, то есть при повторном выполнении одного и того же обновления результат всегда должен быть одним и тем же. В случае с примером AdventureWorks Service, если вы изменяете сведения о клиенте и устанавливаете для номера телефона значение «888-888-8888», то неважно, сколько раз проделана эта операция, поскольку эффект будет одним и тем же. Возможно, все это воспринимается как нечто само собой разумеющееся, тем не менее при разработке REST веб-сервиса это требование упускать из виду не следует. Благодаря этому подходу к проектированию веб-сервис сможет надежно работать в условиях одновременно отправляемых запросов или даже в случае сетевых сбоев (если клиентское приложение отключится от веб-сервиса, оно может просто попытаться восстановить подключение и выполнить тот же самый запрос еще раз, не выясняя, был ли успешным прежний

запрос). Поэтому можно считать REST веб-сервис средством для сохранения и извлечения данных и не пытаться с его помощью реализовывать операции, определяемые для решения бизнес-задач.

К примеру, если создается банковская система, может появиться соблазн ввести в обращение метод CreditAccount, добавляющий сумму к балансу счета клиента, и предоставить этот метод в виде PUT-операции. Но результатом каждого вызова этой операции будет увеличение кредитной составляющей счета. Поэтому придется отслеживать успешность вызова операции. Ваше приложение не может вызывать эту операцию повторно, если есть подозрение, что ранее сделанный вызов был неудачным или истек срок его обработки, поскольку при дублировании кредитной операции на одном и том же счете результат может стать многократно увеличенным.

Дополнительные сведения о целостности данных в облачных приложениях можно найти в статье «Data Consistency Primer» по адресу <https://msdn.microsoft.com/library/dn589800.aspx>.

В следующем упражнении вам предстоит расширить приложение Customers, дополнив его средствами, позволяющими пользователю добавлять сведения о новых клиентах и изменять сведения о существующих. Приложение будет конструировать соответствующие REST-запросы и отправлять их веб-сервису AdventureWorksService. Функциональные средства для удаления сведений о клиентах предоставлены не будут. Это ограничение станет гарантией того, что у вас имеются записи обо всех клиентах, когда-либо имевших деловые отношения с организацией Adventure Works, которые могут понадобиться для аудиторских целей. Кроме того, даже если клиент долго не проявлял активности, есть шанс, что когда-нибудь в будущем он все же разместит новый заказ.



ПРИМЕЧАНИЕ В бизнес-приложениях все шире распространяется норма, согласно которой в них никогда не удаляются данные, а в отношении них просто проводится операция обновления, каким-либо образом помечающая их как удаленные и не позволяющая им появляться на экране. Главным образом это связано с требованиями сохранения всего набора записей, зачастую в свете выполнения определенных нормативных требований.

Реализация в классе ViewModel функциональных возможностей добавления и редактирования данных

Вернитесь в среду Visual Studio. Удалите в проекте Customers файл ViewModel.cs, чтобы убрать его из проекта. Позвольте среде Visual Studio окончательно удалить этот файл.

Щелкните правой кнопкой мыши на проекте Customers, укажите на пункт Добавить, а затем щелкните на пункте Существующий элемент. Выберите файл ViewModel.

cs, который находится в папке \Microsoft Press\VCSBS\Chapter 27 вашей папки документов, после чего щелкните на кнопке **Добавить**.

Код в файле `ViewModel.cs` стал слишком длинным, поэтому он был реорганизован в области, чтобы им было легче управлять. Класс `ViewModel` был расширен за счет следующих булевых свойств, показывающих режим, в котором работает `ViewModel`: просмотра, добавления или редактирования. Эти свойства определены в области **Properties For Managing The Edit Mode** (свойства для управления режимом редактирования).

- ❑ **IsBrowsing.** Это свойство показывает, находится ли модель представления в режиме просмотра. Когда она находится в этом режиме, включены команды `FirstCustomer`, `LastCustomer`, `PreviousCustomer` и `NextCustomer` и представление может вызывать их для просмотра данных.
- ❑ **IsAdding.** Это свойство показывает, находится ли модель представления в режиме добавления. В этом режиме выключены команды `FirstCustomer`, `LastCustomer`, `PreviousCustomer` и `NextCustomer`. Вам будут определены команды `AddCustomer`, `SaveChanges` и `DiscardChanges`, которые будут включены в этом режиме.
- ❑ **IsEditing.** Это свойство показывает, находится ли модель представления в режиме редактирования. Как и в режиме добавления, в этом режиме выключены команды `FirstCustomer`, `LastCustomer`, `PreviousCustomer` и `NextCustomer`. Вам также будет определена команда `EditCustomer`, включенная в этом режиме. Кроме этого, будут включены команды `SaveChanges` и `DiscardChanges`, но команда `AddCustomer` будет выключена. В режиме добавления команда `EditCustomer` будет выключена.
- ❑ **IsAddingOrEditing.** Это свойство показывает, находится ли модель представления в режиме добавления или редактирования. Оно будет использоваться в методах, которые будут определены в данном упражнении.
- ❑ **CanBrowse.** Это свойство возвращает `true`, если модель представления находится в режиме просмотра и есть открытое подключение к веб-сервису. Код в конструкторе, который создает команды `FirstCustomer`, `LastCustomer`, `PreviousCustomer` и `NextCustomer`, был обновлен с целью использования этого свойства и получения возможности определения того, какими должны быть эти команды, включенными или выключенными:

```
public ViewModel()
{
    ...
    this.NextCustomer = new Command(this.Next,
        () => { return this.CanBrowse &&
            this.customers != null && !this.IsAtEnd; });
    this.PreviousCustomer = new Command(this.Previous,
        () => { return this.CanBrowse &&
```

```
        this.customers != null && !this.IsAtStart; });
this.FirstCustomer = new Command(this.First,
    () => { return this.CanBrowse &&
        this.customers != null && !this.IsAtStart; });
this.LastCustomer = new Command(this.Last,
    () => { return this.CanBrowse &&
        this.customers != null && !this.IsAtEnd; });
}
```

- ❑ **CanSaveOrDiscardChanges.** Это свойство возвращает `true`, если модель представления находится в режиме добавления или редактирования и имеет открытое подключение к веб-сервису.

Область `Methods For Fetching And Updating Data` (методы для извлечения и обновления данных) содержит следующие методы.

- ❑ **GetDataAsync.** Этот тот самый метод, который был создан ранее. Он осуществляет подключение к веб-сервису и извлекает сведения о каждом клиенте.
- ❑ **ValidateCustomer.** Этот метод получает `Customer`-объект и проверяет свойства `FirstName` и `LastName`, чтобы убедиться, что они не пусты. Он также проверяет свойства `EmailAddress` и `Phone`, чтобы убедиться, что они содержат информацию, имеющую допустимый формат. Метод возвращает `true`, если данные приемлемы, и `false` — в противном случае. Этот метод будет использоваться в данном упражнении чуть позже, при создании команды `SaveChanges`.



ПРИМЕЧАНИЕ Код, проверяющий свойства `EmailAddress` и `Phone`, определяет соответствие содержащихся в них значений регулярному выражению путем использования класса `Regex`, определенного в пространстве имен `System.Text.RegularExpressions`. Чтобы воспользоваться этим классом, нужно в `Regex`-объекте определить регулярное выражение, указывающее шаблон, которому должны соответствовать данные, а затем вызвать метод `IsMatch`, принадлежащий `Regex`-объекту, с данными, подлежащими проверке. Дополнительные сведения о регулярных выражениях и классе `Regex` можно найти в статье «Объектная модель регулярных выражений» на веб-сайте Microsoft по адресу <http://msdn.microsoft.com/library/30wbz966>.

- ❑ **CopyCustomer.** Задача этого метода заключается в создании поверхностной копии `Customer`-объекта. Метод будет использоваться при создании команды `EditCustomer` для создания копии исходных данных клиента перед их изменением. Если пользователь решит отменить изменения, исходные данные могут быть просто скопированы обратно из копии, созданной этим методом.

В обозревателе решений раскройте проект `Customers` и дважды щелкните на файле `ViewModel.cs`, чтобы открыть его в окне редактора.

Объявление строковой переменной `ServerUrl` ближе к началу класса `ViewModel` имеет следующий вид:

```
private const string ServerUrl = "http://<webappname>.azurewebsites.net/";
```

Измените строку и замените текст <webappname> именем веб-сервиса, созданного вами ранее при изучении этой главы.

В файле `ViewModel.cs` найдите область `Methods For Fetching And Updating Data`, раскрыв ее, если нужно. В этой области выше метода `ValidateCustomer` создайте показанный здесь метод `Add`:

```
// Создает новые (пустые) сведения о клиенте и переводит форму в режим добавления
private void Add()
{
    Customer newCustomer = new Customer { CustomerID = 0 };
    this.customers.Insert(currentCustomer, newCustomer);
    this.IsAnyAdding = true;
    this.OnPropertyChanged(nameof(Current));
}
```

Этот метод создает новый `Customer`-объект. За исключением свойства `CustomerID`, для которого в целях отображения на экране временно установлено значение `0`, он совершенно пуст. Как уже упоминалось, настоящее значение для этого свойства генерируется при сохранении сведений о клиенте в базе данных. Клиент добавляется к списку клиентов (для отображения данных в этом списке представление использует привязку данных), модель представления переводится в режим добавления, и инициируется событие `PropertyChanged`, показывающее, что `Current`-клиент подвергся изменениям.

Добавьте к списку в начале класса `ViewModel` следующую переменную типа `Command`, выделенную жирным шрифтом:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public Command LastCustomer { get; private set; }
    public Command AddCustomer { get; private set; }
    ...
}
```

В конструкторе `ViewModel` создайте экземпляр команды `AddCustomer`, выделенный далее жирным шрифтом:

```
public ViewModel()
{
    ...
    this.LastCustomer = new Command(this.Last, ...);
    this.AddCustomer = new Command(this.Add,
        () => { return this.CanBrowse; });
    ...
}
```

Этот код ссылается на только что созданный вами метод `Add`. Команда включается, если у модели представления имеется подключение к веб-сервису и текущим

является режим просмотра (команда `AddCustomer` не будет включена, если модель представления уже находится в режиме добавления).

Создайте после метода `Add` в области `Methods For Fetching And Updating Data` открытую переменную типа `Customer` по имени `oldCustomer` и определите еще один метод по имени `Edit`:

```
// Редактирует сведения о текущем клиенте
// сохраняет имеющиеся сведения о клиенте
// и переводит форму в режим редактирования
private Customer oldCustomer;

private void Edit ()
{
    this.oldCustomer = new Customer();
    this.CopyCustomer(this.Current, this.oldCustomer);
    this.IsEditing = true;
}
```

Этот метод копирует сведения о `Current`-клиенте в переменную `oldCustomer` и переводит модель представления в режим редактирования. В этом режиме пользователь может изменять сведения о текущем клиенте. Если пользователь сразу после этого решит отменить эти изменения, исходные данные могут быть скопированы обратно из переменной `oldCustomer`.

Добавьте к списку в начале класса `ViewModel` следующую переменную типа `Command`, выделенную жирным шрифтом:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public Command AddCustomer { get; private set; }
    public Command EditCustomer { get; private set; }
    ...
}
```

В конструкторе `ViewModel` создайте экземпляр команды `EditCustomer`, выделенный далее жирным шрифтом:

```
public ViewModel()
{
    ...
    this.AddCustomer = new Command(this.Add, ...);
    this.EditCustomer = new Command(this.Edit,
        () => { return this.CanBrowse; });
    ...
}
```

Этот код аналогичен инструкции для команды `AddCustomer`, за исключением того, что он ссылается на метод `Edit`.

В области `Methods For Fetching And Updating Data` добавьте к находящемуся в ней классу `ViewModel` после метода `Edit` показанный далее метод `Discard`:

```

// Отменяет изменения, сделанные в режиме добавления или редактирования,
// и возвращает форму в режим просмотра
private void Discard ()
{
    // Если пользователь добавил нового клиента, этот клиент удаляется
    if (this.IsAdding)
    {
        this.customers.Remove(this.Current);
        this.OnPropertyChanged(nameof(Current));
    }

    // Если пользователь отредактировал сведения о существующем клиенте,
    // восстановление сохраненных сведений
    if (this.IsEditing)
    {
        this.CopyCustomer(this.oldCustomer, this.Current);
    }

    this.IsBrowsing = true;
}

```

Задача этого метода заключается в том, чтобы позволить пользователю отменить любые изменения, внесенные тогда, когда модель представления находится в режиме добавления или редактирования. Если модель представления находится в режиме добавления, текущий клиент удаляется из списка (речь идет о новом клиенте, который был создан методом `Add`) и инициируется событие `PropertyChanged`, чтобы показать, что сведения о текущем клиенте в списке клиентов подверглись изменениям. Если модель представления находится в режиме редактирования, исходные сведения, находящиеся в переменной `oldCustomer`, копируются обратно, в запись текущего, отображаемого на экране клиента. В заключение модель представления возвращается в режим просмотра.

Добавьте переменную `DiscardChanges` типа `Command` к списку в начале класса `ViewModel` и обновите конструктор для создания экземпляра этой команды (изменения и добавления показаны далее жирным шрифтом):

```

public class ViewModel : INotifyPropertyChanged
{
    ...
    public Command EditCustomer { get; private set; }
    public Command DiscardChanges { get; private set; }
    ...
    public ViewModel()
    {
        ...
        this.EditCustomer = new Command(this.Edit, ...);
        this.DiscardChanges = new Command(this.Discard,
            () => { return this.CanSaveOrDiscardChanges; });
    }
    ...
}

```

Заметьте, что команда `DiscardChanges` включается, только если свойство `CanSaveOrDiscardChanges` имеет значение `true`, у модели представления есть подключение к веб-сервису и она находится в режиме добавления или редактирования.

Добавьте в области `Methods For Fetching And Updating Data` после метода `Discard` еще один метод по имени `SaveAsync`, показанный в следующем примере кода. Этот метод должен быть помечен модификатором `async`:

```
// Сохраняет новые или обновленные сведения о клиенте, отправляя их
// веб-сервису, и возвращает форму в режим просмотра
private async void SaveAsync()
{
    // Проверка сведений о клиенте
    if (this.ValidateCustomer(this.Current))
    {
        // Продолжение только в том случае, если сведения о клиенте приемлемы
        this.IsBusy = true;
        try
        {
            // Преобразование сведений о текущем клиенте в формат HTTP-запроса
            // с полезной нагрузкой в формате JSON
            var serializedData = JsonConvert.SerializeObject(this.Current);
            StringContent content =
                new StringContent(serializedData, Encoding.UTF8, "text/json");

            // Если пользователь добавляет сведения о новом клиенте,
            // отправка веб-сервису HTTP POST-запроса со сведениями
            if (this.IsAdding)
            {
                var response =
                    await client.PostAsync("api/customers", content);
                if (response.IsSuccessStatusCode)
                {
                    // TODO: Отобразить сведения о новом клиенте
                }
                // TODO: Обработать сбой POST-запроса
            }
            // Пользователь должен редактировать сведения об уже существующем
            // клиенте, поэтому отправка сведений с использованием PUT-запроса
            else
            {
                string path = $"api/customers/{this.Current.CustomerID}";

                var response = await client.PutAsync(path, content);
                if (response.IsSuccessStatusCode)
                {
                    this.IsEditing = false;
                    this.IsBrowsing = true;
                }
                // TODO: Обработать сбой PUT-запроса
            }
        }
        catch (Exception e)
```

```
        {
            // TODO: Обработать исключения
        }
    finally
    {
        this.IsBusy = false;
    }
}
```

Этот метод еще не приобрел завершенный вид. Только что введенный код проверяет приемлемость сведений о клиенте. При положительном исходе сведения могут быть сохранены, после чего свойство `IsBusy` класса `ViewModel` устанавливается в `true`, чтобы показать, что операция сохранения может занять некоторое время, пока информация отправляется по сети веб-сервису. (Вспомним, что к этому свойству привязано свойство `IsActive` элемента управления `ProgressRing`, принадлежащего форме `Customers`, и пока данные не будут сохранены, на экране будет находиться кольцевой индикатор занятости.)

Код в блоке `try` определяет, добавляет ли пользователь сведения о новом клиенте, или же он редактирует сведения, принадлежащие уже существующему клиенту. Если пользователь добавляет сведения о новом клиенте, код использует метод `PostAsync` объекта типа `HttpClient`, чтобы отправить веб-сервису POST-запрос. Следует иметь в виду, что POST-запрос отправляется методу `PostCustomer` в классе `CustomersController` веб-сервиса, а этот метод ожидает получения в качестве своего параметра объекта типа `Customer`. Сведения передаются в формате JSON.

Если пользователь редактирует сведения о существующем клиенте, приложение вызывает метод `PutAsync` объекта типа `HttpClient`. Этот метод генерирует PUT-запрос, передаваемый методу `PutCustomer` в классе `CustomersController` веб-сервиса. Метод `PutCustomer` обновляет сведения о клиенте в базе данных и ожидает в качестве параметров идентификатор клиента и сведения о нем. Эти данные передаются в адрес веб-сервиса также в JSON-формате.

Когда данные отправлены, значение свойства `IsBusy` устанавливается в `false`, что заставляет элемент управления `ProgressRing` исчезнуть с экрана.

Замените в методе `SaveAsync` комментарий `// TODO: Отобразить сведения о новом клиенте` следующим кодом, выделенным жирным шрифтом:

```
if (response.IsSuccessStatusCode)
{
    // Получение ID для нового клиента и отображение сведений о нем
    Uri customerUri = response.Headers.Location;
    var newCust = await this.client.GetAsync(customerUri);
    if (newCust.IsSuccessStatusCode)
    {
        var customerData = await newCust.Content.ReadAsStringAsync();
```

```
        this.CopyCustomer(
            JsonConvert.DeserializeObject<Customer>(customerData), this.Current);
        this.OnPropertyChanged(nameof(Current));
        this.IsAdding = false;
        this.IsBrowsing = true;
    }
    else
    {
        // TODO: обработать сбой GET-запроса
    }
}
```

В столбце `CustomerID` таблицы `Customer`, принадлежащей базе данных AdventureWorks, содержатся автоматически генерируемые значения. Когда создаются сведения о новом клиенте, данные для этих значений предоставляются не пользователем, а самой базой данных, когда в нее добавляются сведения о новом клиенте. Тем самым база данных обеспечивает наличие у каждого клиента уникального идентификатора. Следовательно, после отправки веб-сервису POST-запроса вы должны отправить GET-запрос, чтобы получить идентификатор пользователя. К счастью, `HttpResponseMessage`-объект передает обратно через веб-сервис в качестве результата POST-запроса URL-адрес, который приложение может использовать для запроса новых данных. Этот URL-адрес доступен в свойстве ответа `Headers.Location`, и он будет иметь формат `api/Customers/n`, где *n* — это идентификатор клиента. Только что добавленный вами код отправляет GET-запрос по этому URL-адресу, используя метод `GetAsync` объекта типа `HttpClient`. Он считывает обратно сведения о новом пользователе с помощью принадлежащего ответу метода `ReadAsStringAsync`. Затем код обновляет этими данными сведения о клиенте, хранящиеся в коллекции клиентов.



ПРИМЕЧАНИЕ Может создаться впечатление, что этот код выполняет совершенно ненужный обмен данными с веб-сервисом, чтобы извлечь идентификатор клиента, который доступен в свойстве `Headers.Location` сообщения, полученного в ответ на POST-запрос. Но этим действием проверяется, что данные были сохранены правильно, ведь там могут быть и другие поля, преобразуемые веб-сервисом при сохранении данных, следовательно, этот процесс гарантирует приложению, что оно отображает данные в том виде, в котором они сохраняются в базе данных.

Добавьте показанную далее переменную `SaveChanges` типа `Command` к списку в начале класса `ViewModel` и обновите конструктор для создания экземпляра этой команды:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public Command DiscardChanges { get; private set; }
    public Command SaveChanges { get; private set; }
    ...
    public ViewModel()
```

```

{
    ...
    this.DiscardChanges = new Command(this.Discard, ...);
    this.SaveChanges = new Command(this.SaveAsync,
        () => { return this.CanSaveOrDiscardChanges; });
    ...
}
...
}

```

Щелкните в меню Сборка на пункте Собрать решение и убедитесь, что ваше приложение прошло компиляцию без ошибок.

Веб-сервис нуждается в обновлении с целью поддержки функциональных возможностей редактирования. В частности, при добавлении или редактировании сведений о клиенте для свойства клиента `ModifiedDate` нужно устанавливать значение, отражающее дату внесения изменения. Кроме того, если создаются сведения о новом клиенте, то перед их сохранением свойство `rowguid` объекта типа `Customer` должно быть заполнено новым значением GUID. (Это обязательный столбец в таблице `Customer`, другие приложения, используемые в организации Adventure Works, используют этот столбец для отслеживания информации о клиентах.)



ПРИМЕЧАНИЕ GUID означает глобально уникальный идентификатор. GUID представляет собой строку, сгенерированную Windows, которая практически гарантирует ее уникальность (существует весьма призрачная возможность, что Windows может сгенерировать не уникальный GUID, но она настолько мала, что ею можно пренебречь). GUID-идентификаторы часто используются базами данных в качестве значений для ключей, идентифицирующих отдельные строки, в данном случае таблицы `Customer`, имеющейся в базе данных AdventureWorks.

Обновление веб-сервиса для поддержки функциональных возможностей добавления и редактирования

В обозревателе решений в проекте AdventureWorksService раскройте папку Controllers и откройте файл `CustomersController.cs`, чтобы он отобразился в окне редактора.

Добавьте к методу `PostCustomer` перед инструкциями, сохраняющими сведения о новом клиенте в базе данных, следующий код, выделенный жирным шрифтом:

```

// POST api/Customers
[ResponseType(typeof(Customer))]
public async Task<IHttpActionResult> PostCustomer(Customer customer)
{
    if (!ModelState.IsValid)
    {
        ...
    }
    customer.ModifiedDate = DateTime.Now;
}

```

```
customer.rowguid = Guid.NewGuid();
db.Customers.Add(customer);
await db.SaveChangesAsync();
...
}
```

Внесите в принадлежащее методу `PutCustomer` свойство клиента `ModifiedDate`, которое находится перед инструкцией, показывающей, что сведения о клиенте были обновлены, изменения, выделенные жирным шрифтом:

```
// PUT api/Customers/5
[ResponseType(typeof(void))]
public async Task<IHttpActionResult> PutCustomer(int id, Customer customer)
{
    ...
    customer.ModifiedDate = DateTime.Now;
    db.Entry(customer).State = EntityState.Modified;
    ...
}
```

Разверните веб-службу в облаке, следуя процедурам, рассмотренным ранее в упражнении «Развертывание веб-сервиса в облаке», но выложите веб-сервис не в новое веб-приложение, а в то, что уже существует в облаке.

Выдача отчета об ошибках и обновление пользовательского интерфейса

Вы уже добавили команды, с помощью которых пользователь может извлечь, добавить, редактировать и сохранить сведения о клиентах. Но если произойдет какой-либо сбой и случится ошибка, пользователь не будет знать, что случилось, потому что в класс `ViewModel` не включена возможность выдачи отчета об ошибках. Одним из способов добавления этой функции является перехват сообщений о выданных исключениях и предоставление их в виде свойства класса `ViewModel`. Для связи с этим свойством и отображения на экране сообщения об ошибках представление может использовать привязку данных.

Добавление к классу `ViewModel` возможности выдачи отчета об ошибках

Вернитесь в проект `Customers` и выведите в окно редактора файл `ViewModel.cs`. Найдите и раскройте, если необходимо, область по имени `Properties For "Busy" And Error Message Handling` (свойства для обработки сообщений о занятости и об ошибках).

Добавьте после свойства `IsBusy` закрытую строковую переменную `_lastError` и открытое строковое свойство `LastError`:

```
private string _lastError = null;
public string LastError
{
    get { return this._lastError; }
    private set
    {
        this._lastError = value;
        this.OnPropertyChanged(nameof(LastError));
    }
}
```

Найдите в области **Methods For Fetching And Updating Data** метод `GetDataAsync`. Этот метод содержит следующий обработчик исключений:

```
catch (Exception e)
{
    // TODO: Обработать исключения
}
```

Замените комментарий `// TODO: Обработать исключения` следующим кодом, выделенным жирным шрифтом:

```
catch (Exception e)
{
    this.LastError = e.Message;
}
```

Замените в блоке `else`, непосредственно предшествующем обработчику исключений, комментарий `// TODO: Обработать сбой GET-запроса` следующим кодом, показанным жирным шрифтом:

```
else
{
    this.LastError = response.ReasonPhrase;
}
```

Свойство `ReasonPhrase` объекта типа `HttpResponseMessage` содержит строку, показывающую причину сбоя, о которой сообщил веб-сервис.

Добавьте в конец блока `if`, непосредственно предшествующего блоку `else`, следующую инструкцию, показанную жирным шрифтом:

```
if
{
    ...
    this.IsAtEnd = (this.customers.Count == 0);
    this.LastError = String.Empty;
}
else
{
    this.LastError = response.ReasonPhrase;
}
```

Эта инструкция удаляет из свойства `LastError` любые сообщения об ошибках.

Найдите метод `ValidateCustomer` и добавьте непосредственно перед инструкцией `return` следующую инструкцию, показанную жирным шрифтом:

```
private bool ValidateCustomer(Customer customer)
{
    ...
    this.LastError = validationErrors;
    return !hasErrors;
}
```

Метод `ValidateCustomer` наполняет переменную `validationErrors` информацией обо всех свойствах в объекте типа `Customer`, содержащих неприемлемые данные. Только что добавленная вами инструкция копирует эту информацию в свойство `LastError`.

Найдите метод `SaveAsync`. Добавьте к нему код, показанный жирным шрифтом, чтобы перехватить любые ошибки и сведения о сбоях HTTP веб-сервиса:

```
private async void SaveAsync()
{
    // Проверка сведений о клиенте
    if (this.ValidateCustomer(this.Current))
    {
        ...
        try
        {
            ...
            // Если пользователь добавляет сведения о новом клиенте,
            // отправка веб-сервису HTTP POST-запроса со сведениями
            if (this.IsAdding)
            {
                ...
                if (response.IsSuccessStatusCode)
                {
                    ...
                    if (newCust.IsSuccessStatusCode)
                    {
                        ...
                        this.IsBrowsing = true;
                        this.LastError = String.Empty;
                    }
                    else
                    {
                        // TODO: Обработать сбой GET-запроса
                        this.LastError = response.ReasonPhrase;
                    }
                }
                // TODO: Обработать сбой POST-запроса
                else
                {
                    this.LastError = response.ReasonPhrase;
                }
            }
            // Пользователь должен редактировать сведения об уже существующем
        }
```

```
// клиенте, поэтому отправка сведений с использованием PUT-запроса
else
{
    ...
    if (response.IsSuccessStatusCode)
    {
        this.IsEditing = false;
        this.IsBrowsing = true;
        this.LastError = String.Empty;
    }
    // TODO: Обработать сбой PUT-запроса
    else
    {
        this.LastError = response.ReasonPhrase;
    }
}
catch (Exception e)
{
    // TODO: Обработать исключения
    this.LastError = e.Message;
}
finally
{
    this.IsBusy = false;
}
}
```

Найдите метод *Discard*, а затем добавьте в конец этого метода инструкцию, выделенную здесь жирным шрифтом:

```
private void Discard()
{
    ...
    this.LastError = String.Empty;
}
```

В меню Сборка щелкните на пункте Собрать решение и убедитесь в том, что приложение было собрано без ошибок.

Теперь работа над моделью представления завершена. В заключение следует доработать представление, предоставляемое формой *Customers*, включив в него новые команды, информацию о состоянии и функции отчета об ошибках.

Включение в форму *Customers* функциональных возможностей по добавлению и редактированию сведений о клиентах

В окне конструктора откройте файл MainPage.xaml. XAML-разметка для формы MainPage уже подвергалась изменениям, и к элементам управления Grid были добавлены следующие отображающие данные элементы управления типа TextBlock:

```

<Page
  x:Class="Customers.MainPage"
  ...
  <Grid Style="{StaticResource GridStyle}">
    ...
    <Grid x:Name="customersTabularView" ...>
      ...
      <Grid Grid.Row="2">
        ...
        <TextBlock Grid.Row="6" Grid.Column="1"
          Grid.ColumnSpan="7" Style="{StaticResource ErrorMessageStyle}"/>
        </Grid>
      </Grid>
      <Grid x:Name="customersColumnarView" Margin="20,10,20,110" ...>
        ...
        <Grid Grid.Row="1">
          ...
          <TextBlock Grid.Row="6" Grid.Column="0"
            Grid.ColumnSpan="2" Style="{StaticResource ErrorMessageStyle}"/>
        </Grid>
      </Grid>
    ...
  </Grid>
  ...
</Page>
```

Ресурс `ErrorTextStyle`, на который ссылаются эти элементы управления типа `TextBlock`, определены в файле `AppStyles.xaml`.

Установите для свойства `Text` обоих элементов управления типа `TextBlock` выделенную далее жирным шрифтом привязку к свойству `LastError` класса `ViewModel`:

```

...
<TextBlock Grid.Row="6" Grid.Column="1" Grid.ColumnSpan="7"
  Style="{StaticResource ErrorMessageStyle}" Text="{Binding LastError}"/>
...
<TextBlock Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="2"
  Style="{StaticResource ErrorMessageStyle}" Text="{Binding LastError}"/>
```

Элементы управления типа `TextBox` и `ComboBox`, в которых в форме показываются сведения о клиенте, должны позволять пользователю изменять эти сведения только в том случае, если модель представления находится в режиме добавления или редактирования, в противном случае изменение сведений должно быть недоступно. Добавьте к каждому из этих элементов управления свойство `IsEnabled` и привяжите его к свойству `IsAddingOrEditing` класса `ViewModel`:

```

...
<TextBox Grid.Row="1" Grid.Column="1" x:Name="id"
  IsEnabled="{Binding IsAddingOrEditing}" .../>
<TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName"
  IsEnabled="{Binding IsAddingOrEditing}" .../>
<TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName"
  IsEnabled="{Binding IsAddingOrEditing}" .../>
```

```

<ComboBox Grid.Row="1" Grid.Column="3" x:Name="title"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
<TextBox Grid.Row="3" Grid.Column="3" ... x:Name="email"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
<TextBox Grid.Row="5" Grid.Column="3" ... x:Name="phone"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
...
<TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" />
.IsEnabled="{Binding IsAddingOrEditing}" .../>
<TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
<TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
<ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
<TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail"
.IsEnabled="{Binding IsAddingOrEditing}" .../>
...
<TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone"
.IsEnabled="{Binding IsAddingOrEditing}" .../>

```

Добавьте к нижней части страницы панель команд, воспользовавшись для этого элементом `<Page.BottomAppBar>` и поместив эту панель сразу же после верхней панели команд. На этой панели команд должны находиться кнопки для команд добавления сведений о клиенте — `AddCustomer`, редактирования сведений о клиенте — `EditCustomer`, сохранения изменений — `SaveChanges` и отмены изменений — `DiscardChanges`:

```

<Page ...>
...
<Page.TopAppBar >
...
</Page.TopAppBar>
<Page.BottomAppBar>
    <CommandBar>
        <AppBarButton x:Name="addCustomer" Icon="Add"
Label="New Customer" Command="{Binding Path=AddCustomer}"/>
        <AppBarButton x:Name="editCustomer" Icon="Edit"
Label="Edit Customer" Command="{Binding Path>EditCustomer}"/>
        <AppBarButton x:Name="saveChanges" Icon="Save"
Label="Save Changes" Command="{Binding Path=SaveChanges}"/>
        <AppBarButton x:Name="discardChanges" Icon="Undo"
Label="Undo Changes" Command="{Binding Path=DiscardChanges}"/>
    </CommandBar>
</Page.BottomAppBar>
</Page>

```

Заметьте, что значки, на которые ссылаются кнопки, являются стандартными изображениями, предоставляемыми шаблоном Пустое приложение.

Тестирование приложения Customers

В меню Отладка щелкните на пункте Начать отладку, чтобы собрать и запустить приложение. Когда появится форма *Customers*, обратите внимание на то, что элементы типа *TextBox* и *ComboBox* недоступны, поскольку представление работает в режиме просмотра.

Щелкните правой кнопкой мыши на форме и убедитесь в появлении верхней и нижней панелей команд.

Вы, как и прежде, можете пользоваться кнопками *First*, *Next*, *Previous* и *Last* на верхней панели команд (не забудьте, что кнопки *First* и *Previous* не будут доступны, пока вы не перейдете со страницы со сведениями о первом клиенте на следующие страницы). На нижней панели команд должны быть доступны кнопки *Add* и *Edit*, а кнопки *Save* и *Discard* — недоступны, поскольку команды *AddCustomer* и *EditCustomer* разрешены, когда модель представления находится в режиме просмотра, а команды *SaveChanges* и *DiscardChanges* разрешены, только когда модель представления находится в режиме добавления или редактирования (рис. 27.21).

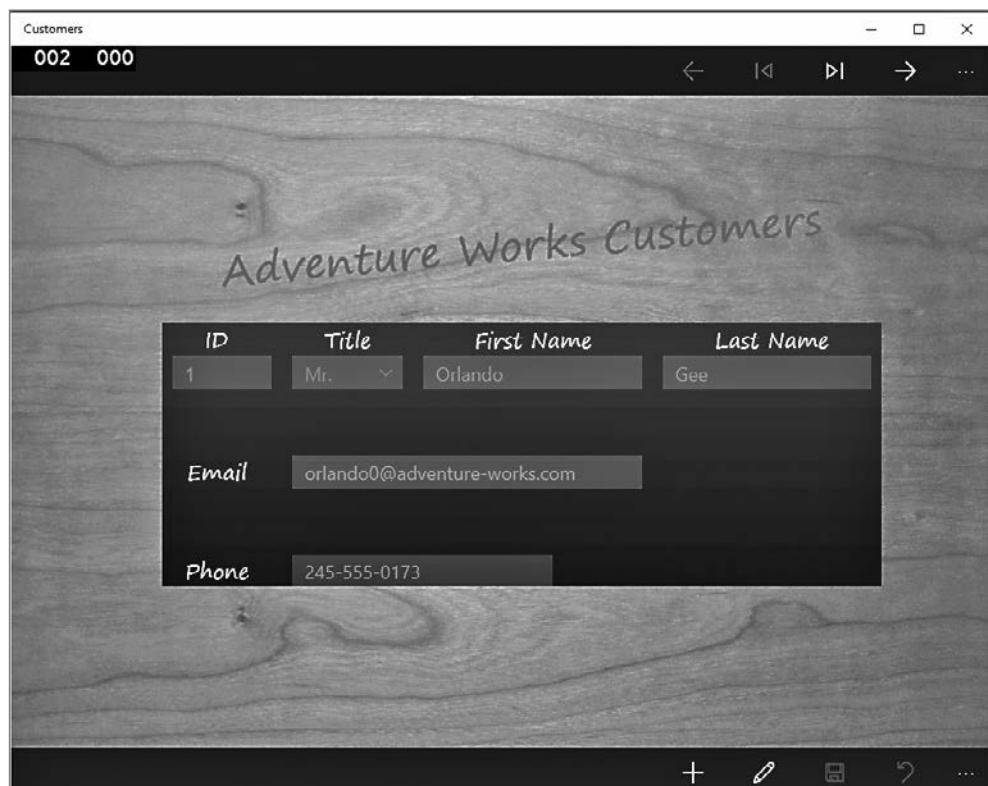


Рис. 27.21

Щелкните на нижней панели команд на кнопке редактирования сведений о клиенте.

Кнопки на верхней панели команд станут недоступны, поскольку модель представления теперь находится в режиме редактирования. Кроме того, теперь также станут недоступны кнопки *Add* и *Edit*, но должны стать доступны кнопки *Save* и *Discard*. К тому же в форме должны стать доступны поля ввода данных, и пользователь сможет изменить сведения о клиенте (рис. 27.22).

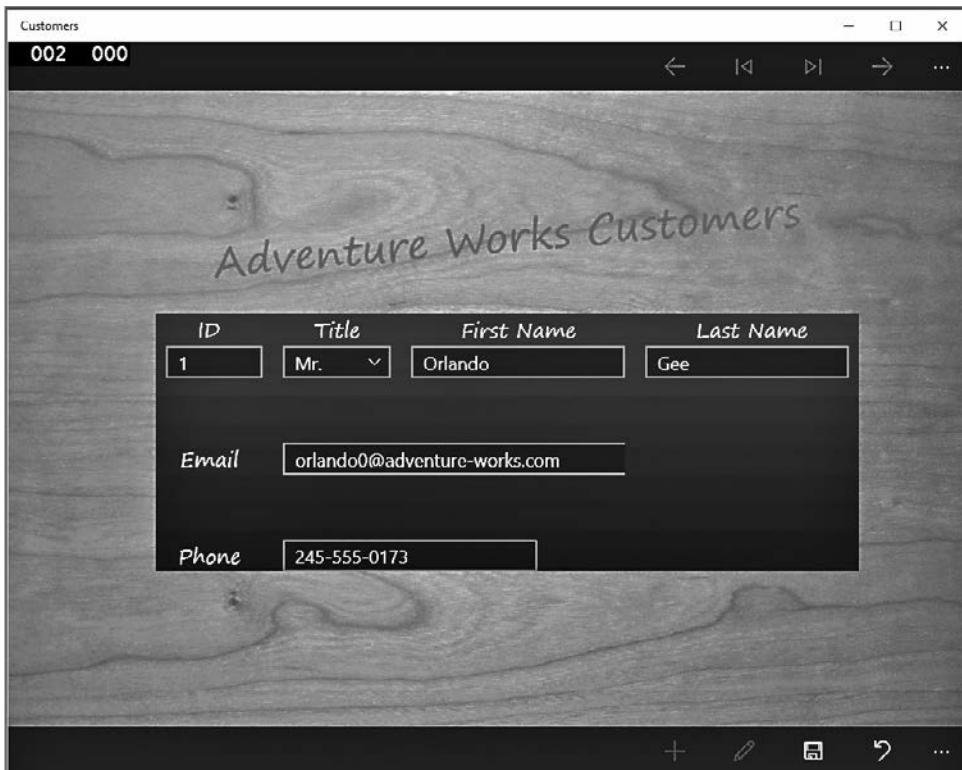


Рис. 27.22

Отредактируйте сведения о клиенте: сотрите его фамилию, наберите *Test* для адреса электронной почты, наберите *Test 2* для номера телефона, а затем щелкните на кнопке *Save*.

Эти изменения нарушают правила, используемые при проверке данных и реализованные в методе *ValidateCustomer*. Этот метод заполняет свойство *LastError* объекта типа *ViewModel* сообщением о результатах проверки, которое выводится в форме в элементе типа *TextBlock*, привязанном к свойству *LastError* (рис. 27.23).

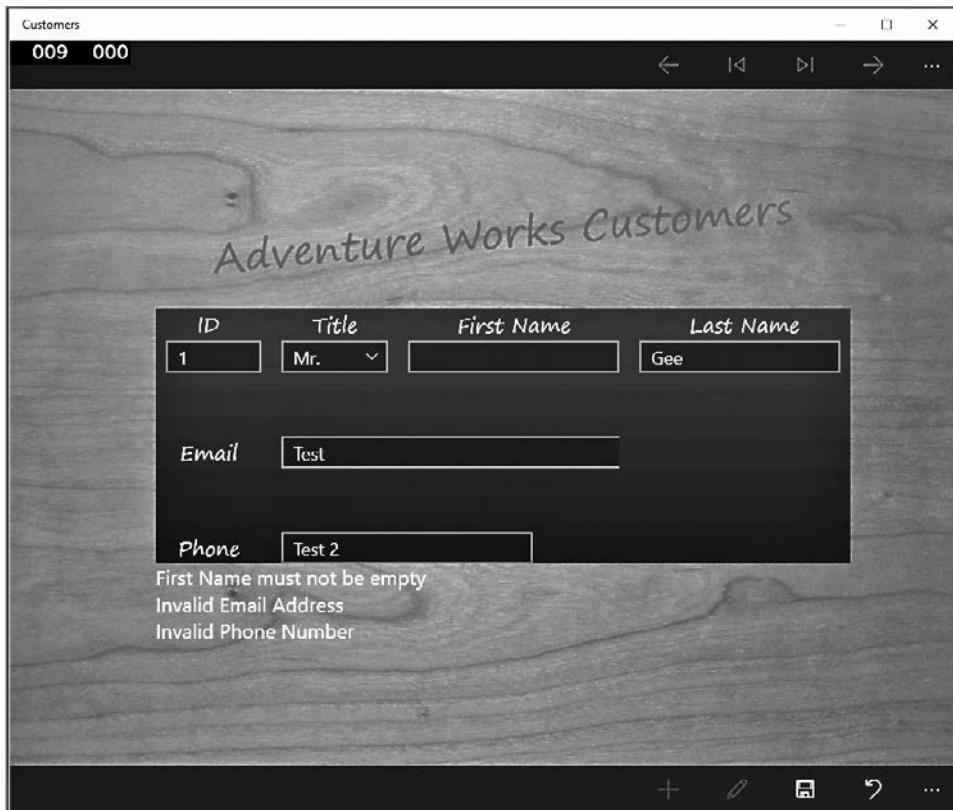


Рис. 27.23

Щелкните на кнопке Discard и убедитесь, что в форме восстановились исходные данные. Сообщение о результатах проверки исчезло, а модель представления вернулась в режим просмотра.

Щелкните на кнопке Add. Поля формы должны очиститься (кроме поля ID, которое показывает значение 0). Введите сведения о новом клиенте. Не забудьте ввести фамилию и имя, правильный адрес электронной почты в формате name@organization.com и цифровой номер телефона (кроме цифр в него можно включать круглые скобки, дефисы и пробелы).

Щелкните на кнопке Save. Если данные приемлемы (проверка не выявила ошибок), они должны быть сохранены в базе данных. В поле ID вы должны увидеть идентификатор, сгенерированный для нового клиента, а модель представления должна переключиться обратно в режим просмотра.

Поэкспериментируйте с приложением, многократно добавляя и редактируя сведения о клиентах. Заметьте, что вы можете изменить размер области

просмотра, чтобы вывести столбцовую разметку, но при этом форма все равно будет работать.

Завершив эксперименты, вернитесь в среду Visual Studio и остановите отладку.

Выводы

В этой главе вы узнали, как среда Entity Framework используется для создания entity-модели, которой можно воспользоваться для подключения к базе данных SQL Server. База данных может быть запущена локально или в облаке. Вы также увидели, как создается REST веб-сервис, которым UWP-приложение может воспользоваться для запроса и обновления данных, хранящихся в базе данных, посредством entity-модели, и узнали, как включать код, вызывающий веб-сервис, в модель представления (в класс `ViewModel`).

Вы выполнили все упражнения данной книги. Я надеюсь, что ваше знакомство с языком C# состоялось успешно и вы усвоили порядок использования среды Visual Studio 2015 для создания профессиональных приложений, работающих под управлением Windows 10. Но жизнь продолжается. Вы преодолели лишь первый барьер на своем пути, а лучшие программисты, работающие на C#, извлекают уроки из непрерывно приобретаемого опыта, который может быть получен вами только в процессе создания приложений на C#. По мере этого вы откроете для себя новые приемы использования языка C# и множество функциональных возможностей среды Visual Studio 2015, на описание которых мне в этой книге не хватило места. Также следует учесть, что язык C# не стоит на месте. В далеком 2001 году, когда мною было написано первое издание этой книги, C# предлагал синтаксис и семантику, необходимую для создания приложений, использующих среду Microsoft .NET Framework 1.0. Затем последовали усовершенствования, добавленные в Visual Studio и .NET Framework 1.1 в 2003-м и 2005-м, появился C# 2.0 с поддержкой обобщений и среды .NET Framework 2.0. В C# 3.0 было добавлено множество новых функций, таких как безымянные типы, лямбда-выражения и, что наиболее существенно, LINQ-расширение. В C# 4.0 произошло дальнейшее расширение языка, появились поддержка именованных аргументов, необязательных параметров, контравариантных и ковариантных интерфейсов, интеграция с динамическими языками. В C# 5.0 была добавлена полная поддержка асинхронной обработки посредством ключевого слова `async` и оператора `await`. В C# 6.0 были представлены дальнейшие усовершенствования языка, такие как методы с телом в виде выражения, строковая интерполяция, оператор `nameof`, фильтры исключений и многое другое.

Параллельно с развитием языка программирования C# со времени выпуска первого издания книги подверглась существенным изменениям и операционная система Windows. Пожалуй, наиболее радикальными за это время стали

изменения, появившиеся с выходом Windows 8 и следующих за ней версий, и перед разработчиками, знакомыми с ранними выпусками Windows, теперь стоят новые увлекательные задачи по созданию приложений для современных, основанных на сенсорных технологиях, мобильных платформ, предоставляемых Windows 10. Несомненно, среда Visual Studio 2015 и язык C# будут играть весьма важную роль в оказании вам помощи в решении этих задач.

Что принесет с собой новая версия C# и Visual Studio? Ждите новостей!

Краткий справочник

Чтобы	Сделайте следующее
Создать entity-модель с помощью среды Entity Framework	<p>Добавьте новый элемент к своему проекту, воспользовавшись шаблоном Модель ADO.NET EDM. Используя мастер моделей EDM, создайте код подключения к базе данных, содержащей таблицы, которые нужно смоделировать, и выберите таблицы, необходимые для вашего приложения.</p> <p>Удалите в модели данных все столбцы, не используемые вашим приложением (при условии, что в них есть значения по умолчанию и если ваше приложение вставляет в базу данных новые элементы)</p>
Создать REST веб-сервис, предоставляющий удаленный доступ к базе данных через entity-модель	<p>Создайте проект ASP.NET, воспользовавшись шаблоном Web API. Запустите мастер Добавление шаблона и выберите шаблон Контроллер Web API2 с действиями, использующий Entity Framework. Укажите в качестве класса модели имя соответствующего entity-класса из entity-модели, а в качестве класса контекста данных — класс, подходящий entity-модели</p>
Воспользоваться REST веб-сервисом в UWP-приложении	<p>Добавьте к проекту библиотеки веб-клиента ASP.NET и воспользуйтесь для подключения к базе данных объектом HttpClient. Установите в качестве значения свойства BaseAddress объекта типа HttpClient ссылку на адрес веб-сервиса, например:</p> <pre>string ServerUrl = "http://localhost:50000/"; HttpClient client = new HttpClient(); client.BaseAddress = new Uri(ServerUrl);</pre>
Извлечь данные из REST веб-сервиса в UWP-приложение	<p>Вызовите метод GetAsync объекта типа HttpClient и укажите URI ресурса, к которому идет обращение. Если метод GetAsync отработает успешно, извлеките данные, возвращенные методом GetAsync, воспользовавшись методом ReadAsStringAsync объекта типа HttpResponseMessage, например:</p>

Чтобы	Сделайте следующее
	<pre data-bbox="450 253 891 600"> HttpClient client = ...; var response = await client.GetAsync("api/customers"); if (response.IsSuccessStatusCode) { var customerData = await response.Content. ReadAsStringAsync(); ... } else { // Сбой GET-запроса }</pre>
Добавить новый элемент данных в REST веб-сервис из UWP-приложения	<p>Воспользуйтесь методом PostAsync объекта типа HttpClient и укажите в качестве параметра новый создаваемый элемент и URI коллекции, содержащей этот элемент. Проверьте статус объекта HttpResponseMessage, возвращенного методом PostAsync, чтобы убедиться, что POST-операция прошла успешно, например:</p> <pre data-bbox="450 817 903 1018"> HttpClient client = ...; StringContent newCustomer = ...; var response = await client.PostAsync("api/customers", newCustomer); if (!response.IsSuccessStatusCode) { // Сбой POST-запроса }</pre>
Обновить существующий элемент в REST веб-сервисе из UWP-приложения	<p>Воспользуйтесь методом PutAsync объекта типа HttpClient и укажите в качестве параметров обновляемый элемент и URL этого элемента. Проверьте статус объекта HttpResponseMessage, возвращенного методом PutAsync, чтобы убедиться, что PUT-операция прошла успешно, например:</p> <pre data-bbox="450 1228 1045 1481"> HttpClient client = ...; StringContent updatedCustomer = ...; string path = \$"api/customers/{updatedCustomer.CustomerID}"; var response = await client.PutAsync(path, updatedCustomer); if (!response.IsSuccessStatusCode) { // Сбой PUT-запроса }</pre>

Джон Шарп

Microsoft Visual C#. Подробное руководство
8-е издание

Перевел с английского Н. Вильчинский

Заведующая редакцией
Ведущий редактор
Литературный редактор
Корректоры
Художественный редактор
Верстка

*Ю. Сергиенко
Н. Римицан
Н. Роцина
С. Беляева, Н. Викторова, В. Сайко
С. Заматевская
Л. Егорова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 19.10.16. Формат 70×100/16. Бумага писчая. Усл. п. л. 68,370. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87



**ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает
профессиональную, популярную и детскую развивающую литературу**

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com
Подробная информация здесь: <http://www.piter.com/page/avtoru>

**Издательский дом «Питер» приглашает к сотрудничеству зарубежных
торговых партнеров или посредников, имеющих выход на зарубежный
рынок:** тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216;
e-mail: books@piter.com

Вопросы по продаже электронных книг: тел.: (812) 703-73-74, доб. 6217;
e-mail: kuznetsov@piter.com