

Евдокимов П. В.

C#

на примерах

Основы языка C#,
первые программы

Многопоточное
программирование

Клиент-серверные
приложения

Создание мобильных
приложений на C#

4-е издание



Евдокимов П. В.

C#

на примерах

4-е издание



"Наука и Техника"

Санкт-Петербург

УДК 004.438

ББК 32.973.2

ISBN 978-5-94387-782-7

Евдокимов П. В.

С# на примерах. 4-е издание (переработанное и обновленное) —
СПб.: Наука и Техника, 2019. — 320 с., ил.

Серия "На примерах"

Эта книга является превосходным учебным пособием для изучения языка программирования С# на примерах. Изложение ведется последовательно: от развертывания .NET и написания первой программы, до многопоточного программирования, создания клиент-серверных приложений и разработки программ для мобильных устройств. По ходу книги даются все необходимые пояснения и комментарии. В четвертом издании был частично переработан текст по ходу изложения всей книги, а также обновлены некоторые примеры.

Книга написана простым и доступным языком. Лучший выбор для результативного изучения С#. Начните сразу писать программы на С#!

ISBN 978-5-94387-782-7



9 78- 5- 94387- 782- 7

Контактные телефоны издательства:

(812) 412 70 26

Официальный сайт: www.nit.com.ru

© Евдокимов П. В., ПРОКДИ РГ

© Наука и Техника (оригинал-макет)

Содержание

Глава 1. ВВЕДЕНИЕ В .NET	10
1.1. ЧТО ТАКОЕ .NET	11
1.2. ИСТОРИЯ .NET	14
1.3. ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ	17
1.4. КАК ПРОГРАММИРОВАЛИ РАНЬШЕ	18
1.4.1. Язык Си и Windows API - традиционный подход	19
1.4.2. Язык C++ и библиотека базовых классов	19
1.4.3. Visual Basic 6.0	20
1.4.4. Язык Java	20
1.4.5. Модель компонентных объектов	21
1.5. ЧТО ПРЕДЛАГАЕТ НАМ .NET	22
1.6. ОСНОВНЫЕ КОМПОНЕНТЫ .NET	23
1.6.1. Три кита: CLR, CTS и CLS	23
1.6.2. Библиотека базовых классов	24
1.7. ЯЗЫК C#	24
1.8. СБОРКИ В .NET	27
1.9. ПОДРОБНО О CTS	29
1.9.1. Типы классов	29
1.9.2. Типы интерфейсов	30
1.9.3. Типы структур	30
1.9.4. Типы перечислений	31
1.9.5. Типы делегатов	31
1.9.6. Встроенные типы данных	31
1.10. ПОДРОБНО О CLS	32
1.11. ПОДРОБНО О CLR	34
1.12. ПРОСТРАНСТВА ИМЕН	34
 Глава 2. РАЗВЕРТЫВАНИЕ .NET И ПЕРВАЯ ПРОГРАММА	 37
2.1. РАЗВЕРТЫВАНИЕ У ЗАКАЗЧИКА	38

2.2. РАЗВЕРТЫВАНИЕ У ПРОГРАММИСТА. УСТАНОВКА VISUAL STUDIO COMMUNITY	43
2.3. ПЕРВАЯ ПРОГРАММА С ИСПОЛЬЗОВАНИЕМ VISUAL STUDIO	46
 Глава 3. ОСНОВНЫЕ КОНСТРУКЦИИ ЯЗЫКА С#	50
3.1. ИССЛЕДОВАНИЕ ПРОГРАММЫ HELLO, WORLD!	51
3.1.1. Пространства имен, объекты, методы	51
3.3. ТИПЫ ДАННЫХ И ПЕРЕМЕННЫЕ	54
3.3.1. Системные типы данных	54
3.3.2. Объявление переменных	55
3.3.3. Внутренние типы данных	56
3.3.4. Члены типов данных	56
3.3.5. Работа со строками	57
Члены класса System.String	58
Базовые операции	58
Сравнение строк	59
Поиск в строке	61
Конкатенация строк	63
Разделение и соединение строк	63
Заполнение и обрезка строк	65
Вставка, удаление и замена строк	65
Получение подстроки	66
Управляющие последовательности символов	66
Строки и равенство	67
Тип System.Text.StringBuilder	67
3.3.6. Области видимости переменных	68
3.3.7. Константы	70
3.4. ОПЕРАТОРЫ	70
3.4.1. Арифметические операторы	70
3.4.2. Операторы сравнения и логические операторы	72
3.4.3. Операторы присваивания	74
3.4.4. Поразрядные операторы	74
3.5. ПРЕОБРАЗОВАНИЕ ТИПОВ ДАННЫХ	75
3.6. НЕЯВНО ТИПИЗИРОВАННЫЕ ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ	79
3.7. ЦИКЛЫ	80

3.7.1. Цикл for	81
3.7.2. Цикл foreach	81
3.7.3. Циклы while и do/while.....	82
3.8. КОНСТРУКЦИИ ПРИНЯТИЯ РЕШЕНИЙ.....	83
3.9. МАССИВЫ	85
3.9.1. Одномерные массивы	85
3.9.2. Двумерные массивы.....	87
3.9.3. Ступенчатые массивы.....	88
3.9.4. Класс Array. Сортировка массивов.....	89
3.9.5. Массив - как параметр	91
3.10. КОРТЕЖИ	91
3.11. КАК ПОДСЧИТАТЬ КОЛИЧЕСТВО СЛОВ В ТЕКСТЕ	92
3.12. ВЫЧИСЛЯЕМ ЗНАЧЕНИЕ ФУНКЦИИ	93
3.13. ДЕЛАЕМ КОНСОЛЬНЫЙ КАЛЬКУЛЯТОР.....	95
3.14. ГРАФИЧЕСКИЙ КАЛЬКУЛЯТОР.....	97
3.15. "УГАДАЙ ЧИСЛО". ИГРА.....	100
 Глава 4. ФАЙЛОВЫЙ ВВОД/ВЫВОД.....	 103
4.1. ВВЕДЕНИЕ В ПРОСТРАНСТВО ИМЕН SYSTEM.IO	104
4.2. КЛАССЫ ДЛЯ МАНИПУЛЯЦИИ С ФАЙЛАМИ И КАТАЛОГАМИ	105
4.2.1. Использование класса DirectoryInfo	106
4.2.2. Классы Directory и DriveInfo. Получение списка дисков ...	108
4.2.3. Класс FileInfo	110
4.2.4. Класс File	113
4.2.5. Классы Stream и FileStream	114
4.2.6. Классы StreamWriter и StreamReader.....	116
4.2.7. Классы BinaryWriter и BinaryReader	117
4.3. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ	118
4.4. ВЫВОД СОДЕРЖИМОГО ФАЙЛА НА C#	120
4.5. РАБОТА С XML-ФАЙЛОМ	123
4.6. АРХИВАЦИЯ ФАЙЛОВ НА C#	129
4.7. ПОДСЧЕТ КОЛИЧЕСТВА СЛОВ В ФАЙЛЕ.....	131

Глава 5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ 133

5.1. ОСНОВЫ ООП 134

5.2. КЛАССЫ И ОБЪЕКТЫ 137

5.2.1. Члены класса 137

5.2.2. Ключевое слово `class` 138

5.2.3. Класс `System.Object` 141

5.2.4. Конструкторы 143

5.2.5. Деструкторы 144

5.2.6. Обращаемся сами к себе. Служебное слово `this` 145

5.2.7. Доступ к членам класса 146

5.2.8. Модификаторы параметров 147

5.2.9. Необязательные параметры 152

5.2.10. Именованные аргументы 152

5.2.11. Ключевое слово `static` 153

5.2.12. Индексаторы 155

5.2.13. Свойства 158

5.3. ПЕРЕГРУЗКА ФУНКЦИЙ ЧЛЕНОВ КЛАССА 158

5.3.1. Перегрузка методов 158

5.3.2. Перегрузка конструкторов 160

5.3.3. Перегрузка операторов 161

5.4. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ 163

5.4.1. Введение в наследование 163

5.4.2. Защищенный доступ 165

5.4.3. Запечатанные классы. Ключевое слово `sealed` 166

5.4.4. Наследование конструкторов 167

5.4.5. Соккрытие имен. Ключевое слово `base` 167

5.4.6. Виртуальные члены 169

5.4.7. Абстрактные классы 170

Глава 6. ИНТЕРФЕЙСЫ, СТРУКТУРЫ И ПЕРЕЧИСЛЕНИЯ 172

6.1. ПОНЯТИЕ ИНТЕРФЕЙСА 173

6.2. КЛЮЧЕВЫЕ СЛОВА `AS` И `IS` 175

6.3. ИНТЕРФЕЙСНЫЕ СВОЙСТВА 176

6.4. ИНТЕРФЕЙСЫ И НАСЛЕДОВАНИЕ	177
6.5. СТРУКТУРЫ	178
6.6. ПЕРЕЧИСЛЕНИЯ	181

Глава 7. ОБРАБОТКА ИСКЛЮЧЕНИЙ..... 183

7.1. ВВЕДЕНИЕ В ОБРАБОТКУ ИСКЛЮЧЕНИЙ	184
7.2. ПЕРЕХВАТ ИСКЛЮЧЕНИЙ. БЛОКИ TRY, CATCH, FINALLY	186
7.3. КЛАСС EXCEPTION	188
7.4. ИСКЛЮЧЕНИЯ УРОВНЯ СИСТЕМЫ	190
7.5. КЛЮЧЕВОЕ СЛОВО FINALLY	191
7.6. КЛЮЧЕВЫЕ СЛОВА CHECKED И UNCHECKED	192

Глава 8. КОЛЛЕКЦИИ И ИТЕРАТОРЫ 194

8.1. ВВЕДЕНИЕ В КОЛЛЕКЦИИ	195
8.2. НЕОБОБЩЕННЫЕ КОЛЛЕКЦИИ	198
8.3. ОБОБЩЕННЫЕ КОЛЛЕКЦИИ	200
8.4. КЛАСС ARRAYLIST. ДИНАМИЧЕСКИЕ МАССИВЫ	202
8.5. ХЕШ-ТАБЛИЦА. КЛАСС HASHTABLE	206
8.6. СОЗДАЕМ СТЕК. КЛАССЫ STACK И STACK<T>	209
8.7. ОЧЕРЕДЬ. КЛАССЫ QUEUE И QUEUE<T>	210
8.8. СВЯЗНЫЙ СПИСОК. КЛАСС LINKEDLIST<T>	212
8.9. СОРТИРОВАННЫЙ СПИСОК. КЛАСС SORTEDLIST<TKEY, TVALUE> ...	215
8.10. СЛОВАРЬ. КЛАСС DICTIONARY<TKEY, TVALUE>	217
8.11. СОРТИРОВАННЫЙ СЛОВАРЬ: КЛАСС SORTEDDICTIONARY<TKEY, TVALUE>	221
8.12. МНОЖЕСТВА: КЛАССЫ HASHSET<T> И SORTEDSET<T>	223
8.13. РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА ICOMPARABLE	225
8.14. ПЕРЕЧИСЛИТЕЛИ	226

8.15. РЕАЛИЗАЦИЯ ИНТЕРФЕЙСОВ IENUMERABLE И IENUMERATOR	227
8.16. ИТЕРАТОРЫ. КЛЮЧЕВОЕ СЛОВО YIELD	228

Глава 9. КОНФИГУРАЦИЯ СБОРОК .NET 230

9.1. СПЕЦИАЛЬНЫЕ ПРОСТРАНСТВА ИМЕН	231
9.2. УТОЧНЕННЫЕ ИМЕНА ИЛИ КОНФЛИКТЫ НА УРОВНЕ ИМЕН	233
9.3. ВЛОЖЕННЫЕ ПРОСТРАНСТВА ИМЕН. ПРОСТРАНСТВО ПО УМОЛЧАНИЮ	234
9.4. СБОРКИ .NET	235
9.4.1. Зачем нужны сборки?	235
9.4.2. Формат сборок	237
9.4.3. Однофайловые и многофайловые сборки	238
9.5. СОЗДАНИЕ СБОРКИ (DLL)	239
9.6. СОЗДАНИЕ ПРИЛОЖЕНИЯ, ИСПОЛЬЗУЮЩЕГО СБОРКУ	243

Глава 10. МНОГОПОТОЧНОСТЬ И ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ 246

10.1. ПАРАЛЛЕЛЬНЫЕ КОЛЛЕКЦИИ	247
10.2. БИБЛИОТЕКА РАСПАРАЛЛЕЛИВАНИЯ ЗАДАЧ	250
10.3. КЛАСС TASK	251
10.4. ОЖИДАНИЕ ЗАДАЧИ	255
10.5. КЛАСС TASKFACTORY	258
10.6. ПРОДОЛЖЕНИЕ ЗАДАЧИ	259
10.7. ВОЗВРАТ ЗНАЧЕНИЯ ИЗ ЗАДАЧИ	259

Глава 11. СЕТЕВОЕ ПРОГРАММИРОВАНИЕ..... 261

11.1. ПРОСТРАНСТВО ИМЕН SYSTEM.NET	262
11.2. КЛАСС URI	263
11.3. ЗАГРУЗКА ФАЙЛОВ (HTTP И FTP)	264
11.4. КЛАСС DNS. РАЗРЕШЕНИЕ ДОМЕННЫХ ИМЕН	268

11.5. СОКЕТЫ	269
11.5.1. Типы сокетов	269
11.5.2. Порты	270
11.5.3. Классы для работы с сокетами.....	271
11.6. КОНВЕРТЕР ВАЛЮТ	272
11.7. ПРОСТОЙ СКАНЕР ПОРТОВ	274
 Глава 12. СОЗДАНИЕ ПРИЛОЖЕНИЯ	
 КЛИЕНТ/СЕРВЕР	277
12.1. ПРИНЦИП РАБОТЫ ПРИЛОЖЕНИЯ	278
12.2. РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ.....	278
12.3. ПРИЛОЖЕНИЕ-КЛИЕНТ	281
12.4. МНОГОПОТОЧНЫЙ СЕРВЕР	285
 Глава 13. РАЗРАБОТКА ПРИЛОЖЕНИЙ	
 ДЛЯ ПЛАНШЕТА	
 ПОД УПРАВЛЕНИЕМ WINDOWS 10	293
13.1. ПОДГОТОВКА К СОЗДАНИЮ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ	294
13.2. ПРОЕКТИРОВАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА	298
13.3. НАПИСАНИЕ КОДА ПРИЛОЖЕНИЯ	302
13.4. КОМПИЛЯЦИЯ И ЗАПУСК ПРИЛОЖЕНИЯ	303
 Глава 14. РАБОТА С БАЗАМИ ДАННЫХ	305

Глава 1.

Введение в .NET

Основы языка C#,
первые программы

Клиент-серверные
приложения

Многопоточное
программирование

Создание мобильных
приложений на C#

1.1. Что такое .NET

Платформа .NET Framework — это технология, поддерживающая создание и выполнение нового поколения приложений и веб-служб.

При разработке платформы .NET Framework учитывались следующие цели:

- Обеспечение среды выполнения кода, которая бы минимизировала конфликты при развертывании программного обеспечения и управлении версиями.
- Обеспечение объектно-ориентированной среды программирования для локального сохранения и выполнения объектного кода либо для удаленного/распределенного выполнения.
- Предоставление среды выполнения кода, гарантирующей безопасное выполнение кода, включая код, созданный неизвестным или не полностью доверенным сторонним разработчиком.
- Предоставление единых принципов разработки для разных типов приложений, таких как приложения Windows и веб-приложения.
- Обеспечение среды выполнения кода, которая бы исключала проблемы с производительностью сред выполнения сценариев или интерпретируемого кода.
- Разработка взаимодействия на основе промышленных стандартов, что позволяет интегрировать код платформы .NET Framework с любым другим кодом.

Платформа .NET Framework состоит из общезыковой среды выполнения (среды CLR или Common Language Runtime) и библиотеки классов .NET Framework.

Основа платформы .NET Framework - среда CLR. Ее можно считать агентом, управляющим кодом во время выполнения и предоставляющим основные службы, такие как управление памятью, потоками и удаленное взаимодействие.

Основная задача среды выполнения - это управление кодом. Код, обращающийся к среде выполнения, называют управляемым кодом, а код, который не обращается к среде выполнения, называют неуправляемым кодом. Библиотека классов является комплексной объектно-ориентированной

коллекцией типов, применяемых для разработки приложений - начиная с консольных приложений, запускаемых из командной строки, и заканчивая приложениями, использующими последние технологические возможности ASP.NET, например, Web.Forms и веб-службы XML. Конечно, с помощью .NET можно создавать и обычные Windows-приложения с интерфейсом пользователя (GUI).

Платформа .NET Framework может размещаться неуправляемыми компонентами. Такие компоненты загружают среду CLR в собственные процессы и запускают выполнение управляемого кода, что в итоге создает программную среду, которая позволяет использовать средства управляемого и неуправляемого выполнения.

Среда CLR управляет памятью, выполнением потоков, выполнением кода, проверкой безопасности кода, компиляцией и другими системными службами. Все эти средства являются внутренними для управляемого кода, который выполняется в среде CLR. Из соображений безопасности управляемым компонентам присваивают разные степени доверия, которые зависят от многих факторов. Управляемый компонент может или не может выполнять операции доступа к файлам, операции доступа к реестру и другие важные функции.

Также среда выполнения обеспечивает управление доступом для кода. Пользователи могут доверить исполняемому приложению, которое внедрено в веб-страницу, воспроизведение звука, но при этом не разрешить ему доступ к файловой системе и личным данным.

CLR реализует инфраструктуру строгой типизации и проверки кода, которую называют системой общих типов (CTS, Common Type System). Такая система обеспечивает самоописание всего управляемого кода. В результате все это приводит к надежности кода.

CLR может размещаться в серверных приложениях, таких как Microsoft SQL Server и службы IIS (Internet Information Services).

Такая инфраструктура позволяет использовать управляемый код для разработки собственной логики программ, пользуясь при этом высочайшей производительностью лучших серверов, которые поддерживают размещение среды выполнения.

Теперь поговорим о библиотеке классов .NET Framework. Библиотека классов платформы представляет собой коллекцию типов, которые тесно интегрируются со средой CLR. Понятно, что библиотека является объектно-ориентированной.

Библиотека предоставляет типы, из которых управляемый код пользователя может наследовать функции. Это упрощает работу с типами .NET, но и уменьшает время, затрачиваемое на изучение новых средств платформы .NET. В коллекциях .NET реализуется набор интерфейсов, которые можно использовать для разработки пользовательских классов коллекций, которые могут объединяться с классами .NET.

Основные функции .NET следующие:

- **Богатая функциональность.** Платформа .NET Framework предоставляет богатый набор функционала “из коробки”. Она содержит сотни классов, которые предоставляют функциональность, готовую к использованию в ваших приложениях. Это означает, что разработчику не нужно вникать в низкоуровневые детали различных операций, таких как I/O, сетевое взаимодействие и т.д.
- **Простая разработка веб-приложений.** ASP.NET - это технология, доступная на платформе .NET для разработки динамических веб-приложений. ASP.NET предоставляет управляемую событиями модель программирования (подобную Visual Basic 6, которая упрощает разработку веб-страниц). ASP.NET предоставляет различные элементы пользовательского интерфейса (таблицы, сетки, календари и т.д.), что существенно упрощает задачу программиста.
- **Поддержка ООП.** Преимущества объектно-ориентированного программирования известны всем. Платформа .NET предоставляет полностью объектно-ориентированное окружение. Даже примитивные типы вроде целых чисел и символов теперь считаются объектами.
- **Поддержка многоязычности.** Как правило, в больших компаниях есть программисты, пишущие на разных языках. Есть те, кто предпочитает C++, Java или Visual Basic. Чтобы переучить человека, нужно потратить время и деньги. Платформа .NET позволяет человеку писать на том языке, к которому он привык.
- **Автоматическое управление памятью.** Утечки памяти - серьезная причина сбоя многих приложений. .NET позволяет программисту не заботиться об управлении памятью, а сконцентрироваться на главном - на приложении.
- **Совместимость с COM и COM+.** Ранее, до появления .NET, COM был стандартом де-факто для компонентизированной разработки приложений. .NET полностью совместим с COM и COM+.
- **Поддержка XML.** Платформа .NET предоставляет XML веб-сервисы, которые основаны на стандартах вроде HTTP, XML и SOAP.

- Простое развертывание и настройка. Развертывание Windows-приложений, использующих компоненты COM, являлось сложной задачей. Однако .NET существенно упростила ее, устранив тем самым еще одну причину головной боли любого программиста, использующего COM.
- Безопасность. Windows-платформа всегда подвергалась критике за плохую безопасность. Microsoft приложила огромные усилия и сделала платформу .NET безопасной для корпоративных приложений. Безопасность типов, безопасность доступа к коду, аутентификация на основе ролей - все это делает приложения надежными и безопасными.

1.2. История .NET

В июле 2000 года на конференции PDC (Professional Developer Conference) компания Microsoft анонсировала новый фреймворк для разработки программного обеспечения - .NET Framework. Первая же бета-версия .NET Framework SDK Beta 1 была опубликована на сайте Microsoft 12 ноября 2000 года, однако она была настолько "сырой", что Microsoft рекомендовала ее устанавливать только на компьютеры, предназначенные для тестирования. Как говорится, первый блин всегда комом. И таким комом была первая бета-версия .NET.

Наверное, вам будет интересно узнать, что изначально платформа должна была называться Microsoft.Net, однако Билл Гейтс решил переименовать ее просто в .NET. Также он заявил, что "стратегия корпорации целиком и полностью будет определяться платформой .Net" и что все продукты компании со временем будут переписаны с учетом этой платформы.

Первая версия .NET появилась лишь два года спустя - 1 мая 2002 года. В целом, таблица 1.1 содержит информацию обо всех версиях .NET, выпущенных с 2002 года.

Таблица 1.1. Версии .NET

Версия	Дата выхода	Visual Studio	По умолчанию в Windows	Заменяет версию
1.0	1 мая 2002 г.	Visual Studio .NET	-	-
1.1	1 апреля 2003 г.	Visual Studio .NET 2003	Windows Server 2003	1.0

2.0	11 июня 2005 г.	Visual Studio 2005	Windows Vista, Windows 7, Windows Server 2008 R2	-
3.0	6 ноября 2006 г.	Visual Studio 2005 + расширения	Windows Vista, Windows Server 2008, Windows 7, Windows Server 2008 R2	2.0
3.5	9 ноября 2007 г.	Visual Studio 2008	Windows 7, Windows Server 2008 R2	2.0, 3.0
4.0	12 апреля 2010 г.	Visual Studio 2010	Windows 8, Windows Server 2012	-
4.5	15 августа 2012 г.	Visual Studio 2012	Windows 8, Windows Server 2012	4.0
4.5.1	17 октября 2013 г.	Visual Studio 2013	Windows 8.1, Windows Server 2012 R2	4.0, 4.5
4.5.2	5 мая 2014 г.	-	-	4.0 - 4.5.1
4.6	20 июля 2015 г.	Visual Studio 2015	Windows 10	4.0 - 4.5.2
4.6.1	17 ноября 2015 г.	Visual Studio 2015 Update 1	Windows 10 Version 1511	4.0 - 4.6
4.6.2	20 июля 2016			4.0-4.6.1
4.7	5 апреля 2017	Visual Studio 2017	Windows 10 v1703	4.0-4.6.2
4.7.1	17 октября 2017	Visual Studio 2017 v15.5	Windows 10 v1709, Windows Server 2016	4.0-4.7
4.7.2	30 апреля 2018	Visual Studio 2017 v15.8	Windows 10 v1803	4.0-4.7.1

Первый релиз .NET Framework 1.0 предназначался для Windows 98, NT 4.0, 2000 и XP. Поддержка XP, а значит, и .NET Framework 1.0, закончилась в

2009 году. Версия 1.1 автоматически устанавливалась вместе с Windows Server 2003, для других выпусков Windows она была доступна в виде отдельного установочного файла. Обычная поддержка этой версии .NET закончилась в 2008 году, а расширенная - в 2013-м.

Версия 2.0 выпущена вместе с Visual Studio 2005. В этой версии была добавлена поддержка обобщенных (generic) классов, анонимных методов, а также полная поддержка 64-битных платформ x64 и IA-64. Поддержка этой версии закончена в 2011 году, а расширенная заканчивается 12 апреля 2016 года.

Интересно, что изначально версия .NET Framework 3.0 должна была называться WinFX, что должно было отражать ее суть, а именно добавленные в ее состав компоненты:

- Windows Presentation Foundation (WPF) — презентационная графическая подсистема, использующая XAML;
- Windows Communication Foundation (WCF) — программная модель межплатформенного взаимодействия;
- Windows Workflow Foundation (WF) — технология определения, выполнения и управления рабочими процессами;
- Windows CardSpace — технология унифицированной идентификации.

Расширенной поддержки этой версии не было, а обычная уже закончилась - еще в 2011 году.

Версия 3.5 поддерживает C# 3.0, VB.NET 9.0, в ней также расширена функциональность WF и WCF (см. выше), добавлена поддержка ASP.NET, добавлено новое пространство имен.

Версия 4.0 появилась в 2010 году вместе с Visual Studio 2010. Нововведений довольно много:

- Полная поддержка F#, IronRuby, IronPython;
- Поддержка подмножеств .NET Framework и ASP.NET в варианте Server Core;
- Библиотека параллельных задач (Task Parallel Library), предназначенная для упрощения создания распределенных систем;
- Средства моделирования Oslo;
- Язык программирования M, используемый для создания предметно-ориентированных языков;

Версия 4.5 появилась в августе 2012 года, и ее характерной особенностью является отсутствие поддержки Windows XP. Основные особенности этой версии:

- Улучшенная поддержка сжатия ZIP;
- Поддержка UTF-16 в консоли;
- Уменьшение количества перезапусков системы посредством обнаружения и закрытия приложений платформы .NET Framework версии 4 во время развертывания;
- Фоновая компиляция по требованию (JIT), которая доступна на многоядерных процессорах для повышения производительности приложения;
- Поддержка огромных массивов (размер более 2 Гб) на 64-разрядных системах;
- Улучшенная производительность при извлечении ресурсов.

И это только начало. Нововведений очень много и дополнительную информацию вы можете получить по адресу:

<http://www.codeproject.com/Articles/599756/Five-Great-NET-Framework-Features>

Предпоследняя на момент написания этих строк версия - 4.6. Она является обновлением для версии 4.0 - 4.5.2. Устанавливается при необходимости вместе с версией 3.5 SP1, поставляется вместе с Visual Studio 2015.

В версии 4.6 появились новый JIT-компилятор для 64-разрядных систем, поддержка последней версии CryptoAPI от Microsoft, а также поддерживаются TLS 1.1 и 1.2.

Самая последняя версия - 4.7.2 которая появилась относительно недавно. Является обновлением для версий 4.0 - 4.7.1 и поставляется вместе с Visual Studio 2017.

1.3. Поддерживаемые операционные системы

Практически каждая новая версия .NET по умолчанию использовалась в ближайшем следующем выпуске Windows. Однако это не означает, что данная версия .NET поддерживает только этот выпуск Windows. Информация о поддержке операционных систем приведена в таблице 1.2.

Таблица 1.2. Поддержка ОС

Версия Windows	1.0	1.1	2.0	3.0	4.0	4.5	4.5.2	4.6	4.6.1	4.7	4.7.1.
Windows 98	+										
Windows NT	+										
Windows 2000	+	+	+								
Windows XP	+	+	+	+	+						
Windows Server 2003		+	+	+	+						
Windows Server 2008			+	+	+	+	+	+			
Windows Vista			+	+	+	+	+	+			
Windows 7			+	+	+	+	+	+	+	+	+
Windows Server 2008 R2			+	+	+	+	+	+	+	+	+
Windows Server 2012			+	+	+	+	+	+	+	+	+
Windows 8			+	+	+	+	+	+	+	+	+
Windows 8.1			+	+	+	+	+	+	+	+	+
Windows Server 2012 R2			+	+	+	+	+	+	+	+	+
Windows 10			+	+	+	+	+	+	+	+	+

1.4. Как программировали раньше

Прежде чем переходить к изучению платформы .NET, нужно разобраться, как программировали до ее появления и что сподвигло Microsoft создать

новую платформу разработки. Поэтому далее ваш ждет небольшой экскурс в историю программирования. Конечно, в перфокарты мы погружаться не будем, а затронем более современные методы программирования.

1.4.1. Язык Си и Windows API - традиционный подход

Данный подход можно назвать традиционным: разработка программы ведется на Си с использованием интерфейса Windows API (Application Programming Interface — интерфейс прикладного программирования). Данный подход проверен временем, и с его использованием написано очень много приложений.

Хотя этот подход и проверен временем, процесс создания приложений с помощью одного только API-интерфейса является очень сложным занятием. Основная проблема в том, что Си сам по себе уж очень лаконичный язык. Любой Си-программист вынужден мириться с необходимостью “вручную” управлять памятью, иметь дело с указателями и ужасными синтаксическими конструкциями. В современном мире все эти вещи существенно упрощены, а язык (среда) сам управляет памятью, выделением и освобождением ресурсов и т.д. Но в традиционном Си всего этого нет.

Кроме того, поскольку Си - это структурный язык программирования, ему не хватает преимуществ, которые обеспечиваются объектно-ориентированным подходом. В современном мире программа, написанная на Си и Windows API, выглядит как динозавр - ужасно и устрашающе. Неудивительно, что раньше Windows-приложения часто “глючили”.

1.4.2. Язык C++ и библиотека базовых классов

Появление C++ - огромный шаг вперед. Многие не считают C++ полноценным языком программирования, а лишь объектно-ориентированной надстройкой над Си. Но это никак не уменьшает его преимуществ. С появлением C++ программистам стало доступно объектно-ориентированное программирование и его основные принципы - инкапсуляция, наследование и полиморфизм.

Но несмотря на поддержку ООП, программисты вынуждены мириться с утомительными деталями языка Си (выделение памяти вручную, ужасные указатели и т.д.). Поэтому было решено упростить работу самим себе - так появились платформы для программирования на C++. Одна из самых популярных называется MFC (Microsoft Foundation Classes — библиотека базовых классов Microsoft).

Платформа MFS предоставляет исходный API-интерфейс Windows в виде набора классов, макросов и множества средств для автоматической генерации программного кода (мастеры, wizards).

Хотя этот подход к программированию более удачен, чем предыдущий, процесс программирования на С++ остается очень трудным, и велика вероятность возникновения ошибок из-за его связи с языком Си.

1.4.3. Visual Basic 6.0

Первым языком программирования, с которого многие начинают свою карьеру программиста, является Basic (Beginner's All-purpose Symbolic Instruction Code). Данный язык специально был предназначен для новичков, он очень прост, и обучение программированию часто начинают именно с этого языка. Язык сам по себе довольно древний - он был разработан в 1964 году, а вторую жизнь Basic получил с появлением Visual Basic от Microsoft.

Чтобы облегчить себе жизнь, многие программисты перешли с C/C++ в мир простых и более дружелюбных языков вроде Visual Basic 6.0 (VB6). VB6 предоставляет возможность создавать сложные пользовательские интерфейсы, библиотеки программного кода (вроде COM-серверов) и логику доступа к базам данных. И все это на VB6 делается очень просто, в чем и заключается причина его популярности.

Основной недостаток языка VB6 - в том, что он не является полностью объектно-ориентированным. Можно сказать, что он просто "объектный". Так, VB6 не позволяет программисту устанавливать между классами отношения "подчиненности" (т.е. прибегать к классическому наследованию) и не обладает никакой внутренней поддержкой для создания параметризованных классов. Также VB6 не позволяет создавать многопоточные приложения, если программист не готов работать на уровне Windows API (это сложно и довольно опасно).

Однако с появлением .NET все недостатки VB6 устранены, правда, новый язык теперь имеет мало общего с VB6 и называется VB.NET. В этом современном языке поддерживается переопределение операций (перегрузка), классическое наследование, конструкторы типов, обобщения и многое другое.

1.4.4. Язык Java

Первое, что мне запомнилось в свое время, когда впервые появился Java, - это возможность написания кросс-платформенного кода. Но это не единственное преимущество Java.

Java - это объектно-ориентированный язык программирования, который по своему синтаксису похож на C++, но при этом Java не имеет многих из тех неприятных синтаксических аспектов, которые присутствуют в C++, а как платформа — предоставляет в распоряжение программистам большее количество готовых пакетов с различными определениями типов внутри. Благодаря этому программисты могут создавать на Java приложения с возможностью подключения к БД, поддержкой обмена сообщениями, веб-интерфейсами - и все это используя возможности только самого Java.

Сам по себе Java - очень элегантный язык, но у него есть одна потенциальная проблема: применение Java означает необходимость использования Java в цикле разработки и для взаимодействия клиента с сервером. Другими словами, Java очень сложно интегрировать с другими языками, поскольку он задумывался изначально как единственный язык программирования и единственная платформа для удовлетворения любой потребности. Смешать Java с другим языком будет очень сложно, поскольку ограничен доступ к Java API.

1.4.5. Модель компонентных объектов

Вот мы и добрались к пред-.NET решению. Модель COM (Component Object Model - модель компонентных объектов) была предшественницей платформой для разработки приложений, которая предлагалась Microsoft перед .NET. Впервые COM появилась в 1993 году, так что модель сама по себе уже является довольно древней, учитывая скорость развития технологий.

Модель COM позволяет строить типы в соответствии с правилами COM и получать блок многократно используемого двоичного кода. Такие двоичные блоки кода называют “серверами COM”. Одно из преимуществ сервера COM в том, что к нему можно получить доступ, используя другой язык программирования. Например, кто-то может создать COM-объект на C++, а вы можете использовать Delphi и подключаться к COM-серверу.

Понятно, что подобная независимость COM от языка является немного ограниченной. Например, нет способа породить новый COM-класс с использованием уже существующего (COM не поддерживает классическое наследие). Но это уже нюансы.

Если говорить о преимуществах, то одно из преимуществ COM - прозрачность расположения. За счет применения различных конструкций (прокси-серверы, заглушки, AppID) программисты могут избежать необходимости иметь дело с низким уровнем - сокетами, RPC-вызовами и т.д.

Модель COM можно считать успешной объектной моделью, однако ее внутреннее устройство является очень сложным для восприятия, именно поэтому программистам требуются месяцы на изучение этой модели.

Для облегчения процесса разработки двоичных COM-объектов программисты могут использовать многочисленные платформы, поддерживающие COM. Так, в ATL (Active Template Library — библиотека активных шаблонов) для упрощения процесса создания COM-серверов предоставляется набор специальных классов, шаблонов и макросов на C++. Во многих других языках сложная инфраструктура COM также скрывается из вида. Но поддержки одного только языка для сокрытия всей сложности COM — мало. Даже при выборе относительно простого языка с поддержкой COM (пусть это будет VB6) все равно программисту нужно бороться с записями о регистрации и многочисленными деталями развертывания. Все это называется адом DLL (DLL Hell).

Конечно, COM упрощает процесс создания приложений с помощью разных языков программирования. Но независимая от языка природа COM не настолько проста, как нам бы этого хотелось. Вся эта сложность — следствие того, что приложения, написанные на разных языках, получаются совершенно не связанными с точки зрения синтаксиса. Взять, например, языки Java и Си — их синтаксисы очень похожи, но вот VB6 вообще никак не похож на Си. Его корни уходят в Basic. А COM-серверы, созданные для выполнения в среде COM+ (она представляет собой компонент ОС Windows, предлагающий общие службы для библиотек специального кода, такие как транзакции, жизненный цикл объектов и т.д.), имеют совершенно другое поведение, чем ориентированные на использование в веб-сети ASP-страницы, в которых они вызываются.

В результате получается очень запутанная смесь технологий. Каждый API-интерфейс поставляется с собственной коллекцией уже готового кода, а базовые типы данных не всегда интерпретируются одинаково. У каждого языка своя собственная и уникальная система типов. Из-за этого COM-разработчикам нужно соблюдать предельную осторожность при создании общедоступных методов в общедоступных классах COM. Если вам, например, нужно создать метод на C++, который бы возвращал массив целых чисел в приложение на VB6, то вам придется полностью погрузиться в сложные вызовы API-интерфейса COM для создания структуры безопасного массива. Ведь если разработчик на C++ просто вернет собственный массив, приложение на VB6 просто не поймет, что с ним делать.

1.5. Что предлагает нам .NET

Как видите, жизнь программиста Windows-приложений раньше была очень трудной. Но с появлением .NET Framework все стало гораздо проще. Как уже отмечалось, .NET Framework представляет собой программную платформу для создания приложений на базе семейства операционных систем Windows, а также многочисленных операционных систем разработки не Microsoft, таких как Mac OS X и различные дистрибутивы Unix и Linux.

Основные функциональные возможности .NET:

- Возможность обеспечения взаимодействия с существующим программным кодом. Позволяет обеспечивать взаимодействие существующих двоичных единиц COM с более новыми двоичными единицами .NET и наоборот. С появлением .NET 4.0 данная возможность выглядит еще проще благодаря ключевому слову *dynamic* (в книге мы о нем поговорим).
- Поддержка разных языков программирования (C#, Visual Basic, F# и т.д.).
- Наличие общего исполняющего механизма, который используется всеми языками .NET. Благодаря этому есть хорошо определенный набор типов, которые способен понимать каждый поддерживающий .NET язык.
- Тотальная интеграция языков. Поддерживаются межъязыковое наследование, обработка исключений, отладка кода.
- Огромная библиотека базовых классов. Она позволяет упростить прямые вызовы к API-интерфейсу и предлагает согласованную объектную модель, которую могут использовать все языки .NET.
- Упрощенная модель развертывания. В .NET не нужно заботиться о регистрации двоичной единицы в системном реестре. В .NET можно сделать так, чтобы разные версии одной и той же сборки DLL могли без проблем одновременно существовать на одной и той же машине.

Как видите, .NET не имеет ничего общего с COM, за исключением разве что поддержки двоичных единиц COM.

1.6. Основные компоненты .NET

1.6.1. Три кита: CLR, CTS и CLS

Настало время познакомиться с тремя ключевыми компонентами .NET: CLR, CTS и CLS. С точки зрения программиста .NET представляет со-

бой исполняющую среду и обширную библиотеку базовых классов. Уровень исполняющей среды называется общезыковой исполняющей средой (Common Language Runtime) или средой CLR (такое название используется чаще).

Основная задача CLR - автоматическое обнаружение, загрузка и управление типами .NET. Теперь типами управляет .NET, а не программист. Также среда CLR заботится о ряде низкоуровневых деталей - управление памятью, обработка потоков, выполнение разных проверок, связанных с безопасностью.

Другой компонент .NET - общая система типов (Common Type System) или система CTS. Предоставляет полное описание всех возможных типов данных и программных конструкций, которые поддерживаются исполняющей средой, а также способов, как все эти сущности могут взаимодействовать друг с другом. Нужно понимать, что любая возможность CTS может не поддерживаться в отдельно взятом языке, совместимом с .NET.

Именно поэтому существует третий компонент - CLS (Common Language Specification) или спецификация CLS. В ней описано лишь то подмножество общих типов и программных конструкций, каковое способны воспринимать все .NET языки. Следовательно, если вы используете типы .NET только с функциональными возможностями, предусмотренными в CLS, можете быть уверены, что все совместимые с .NET языки могут их и использовать. Если же вы используете тип данных, которого нет в CLS, нет гарантии того, что с этим типом данных сможет работать любой поддерживаемый .NET язык. К счастью, существует способ указать компилятору С#, чтобы он проверял весь код на предмет совместимости с CLS.

1.6.2. Библиотека базовых классов

Кроме среды CLR и спецификаций CTS и CLS, в составе платформы .NET существует библиотека базовых классов. Она доступна для всех языков, поддерживающих .NET. В этой библиотеке содержатся определения примитивов (потоки, файловый I/O, системы графической визуализации, механизмы для взаимодействия с разными внешними устройствами), предоставлена поддержка целого ряда служб, которые нужны в большинстве реальных приложений.

В библиотеке базовых классов содержатся определения типов, которые могут упростить процесс доступа к базам данным, обеспечить безопасность, создание обычных, консольных и веб-интерфейсов и т.д. На рис. 1.1 показана связь между библиотекой базовых классов и компонентами .NET.

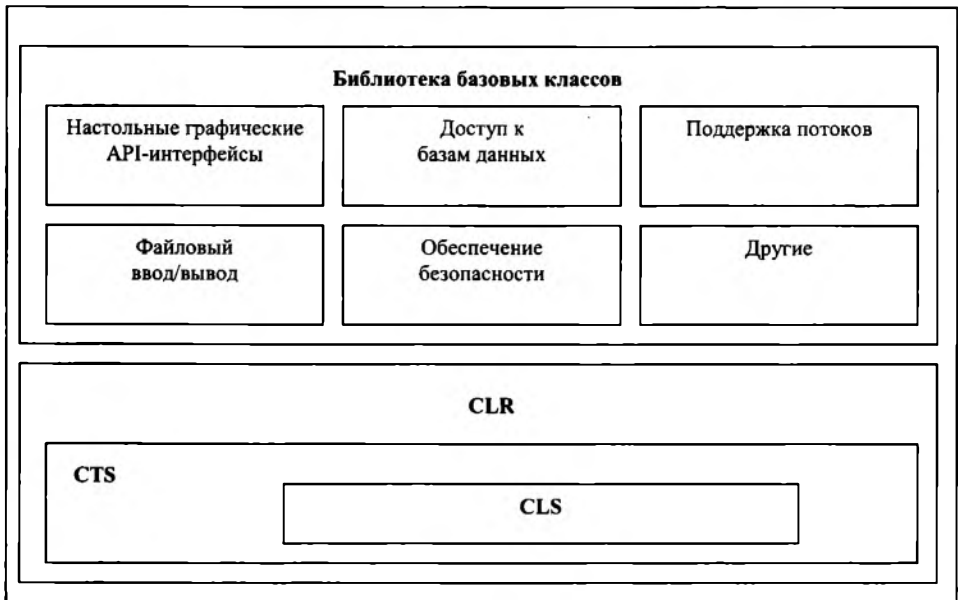


Рис. 1.1. Взаимосвязь между компонентами .NET и библиотекой базовых классов

1.7. Язык C#

Поскольку платформа .NET радикально отличается от предыдущих технологий Microsoft, корпорация разработала специально для нее новый язык программирования - C#. Синтаксис этого языка похож... на Java. Даже не на C++ (хотя язык называется C#), а именно на Java. В принципе, все эти языки - Си, Objective C, C++, C# и Java - используют похожий синтаксис. Все они являются членами семейства языков программирования Си.

Не смотря на схожесть с Java, многие синтаксические конструкции в C# моделируются согласно различным особенностям Visual Basic 6.0 и C++. Как и в VB6, в C# поддерживаются понятие формальных свойств типов (вместо традиционным методом `get` и `set`). Как и в C++, в C# допускается перезагрузка операций, создание структур, перечислений и функций обратного вызова.

В C# поддерживается целый ряд функциональных возможностей, которые обычно встречаются в разных языках программирования (LISP и подобные) - лямбда-выражения, анонимные типы и т.п.

Поскольку C# - смесь бульдога с носорогом, то есть гибридный язык, взявший лучшее из нескольких языков программирования, он является простым, мощным и гибким языком. Рассмотрим же неполный список его ключевых функциональных возможностей:

- Не нужно никаких указателей. Наконец-то программисты избавились от необходимости работы с указателями. Обычно в C# не нужно работать с указателями напрямую, но поддержка самих указателей есть, если вдруг понадобится.
- Автоматическая сборка мусора (автоматическое управление памятью). Именно поэтому ключевое слово **delete** в C# отсутствует.
- Формальные синтаксические конструкции для классов, интерфейсов, структур, перечислений и делегатов.
- Возможность перегрузки операторов для пользовательских типов (как в C++), но при этом не нужно заботиться о возврате **this* для обеспечения связывания.
- Возможность создания обобщенных типов и обобщенных элементов-членов.
- Поддержка анонимных методов, позволяющих предоставлять встраиваемую функцию везде, где требуется использовать тип делегата. С появлением NET 2.0 (с 2005 года) появились упрощения в модели "делегат-событие", в том числе появилась возможность применения ковариантности¹, контравариантности² и преобразования групп методов.
- Начиная с версии .NET 3.5, появилась поддержка для строго типизированных запросов (их также называются запросами LINQ), которые используются для взаимодействия с разными видами данных.
- Поддержка анонимных типов, позволяющих моделировать форму типа, а не его поведение.

1 Ковариантность - это сохранение иерархии наследования исходных типов в производных типах в том же порядке. Так, если класс `Toyota` наследует от класса `Cars`, то естественно полагать, что перечисление `IEnumerable<Toyota>` будет потомком перечисления `IEnumerable<Cars>`

2 Контравариантность - это обращение иерархии исходных типов на противоположную в производных типах. Так, если класс `String` наследует от класса `Object`, а делегат `Action<T>` определен как метод, принимающий объект типа `T`, то `Action<Object>` наследует от делегата `Action<String>`, а не наоборот. Действительно, если "все строки — объекты", то "всякий метод, оперирующий произвольными объектами, может выполнить операцию над строкой", но не наоборот. В таком случае говорят, что тип (в данном случае обобщенный делегат) `Action<T>` контравариантен своему параметру - типу `T`.

С появлением .NET 4.0 язык C# снова был обновлен и дополнен рядом новых функциональных возможностей, а именно:

- Поддержка необязательных параметров в методах, а также именованных аргументов.
- Поддержка динамического поиска членов во время выполнения посредством ключевого слова **dynamic**.
- Значительное упрощение обеспечение взаимодействия приложений на C# с унаследованными COM-серверами благодаря устранению зависимости от сборок взаимодействия и предоставлению поддержки необязательных аргументов **ref**.
- Работа с обобщенными типами стала гораздо понятнее благодаря появлению возможности легко отображать обобщенные данные из общих коллекций `System.Object`.

Наиболее важный момент, который вы должны знать, программируя на C#: с помощью этого языка можно создавать только такой код, который будет выполняться в исполняющей среде .NET (то есть использовать C# для построения "классического" COM-сервера или неуправляемого приложения с вызовами API-интерфейса и кодом на Си и C++ нельзя).

Код, ориентируемый на выполнение в исполняющей среде .NET, называется управляемым кодом (*managed code*). Код, который не может обслуживаться непосредственно в исполняющей среде .NET, называют неуправляемым кодом (*unmanaged code*).

Примечание. Вы должны также понимать, что C# - это не единственный язык, который можно использовать для построения .NET-приложений. При установке бесплатного комплекта разработки Microsoft .NET 4.0 Framework SDK (как и при установке Visual Studio) для выбора доступны пять языков: C#, Visual Basic, C++, JScript .Net, F#.

1.8. Сборки в .NET

Какой бы язык .NET вы бы ни выбрали для программирования, важно понимать, что двоичные .NET-единицы имеют такое же расширение файлов, как и двоичные единицы COM-серверов и неуправляемых программ Win32 (.dll и .exe), но внутри они устроены совершенно иначе.

Двоичные .NET-единицы DLL не экспортируют методы для упрощения взаимодействия с исполняющей средой COM - ведь .NET - это не COM. Они не описываются с помощью библиотек COM-типов и не регистрируются в системном реестре. Самое важное заключается в том, что они содер-

жат не специфические, а наоборот, не зависящие от платформы инструкции на промежуточном языке (Intermediate Language — IL), а также метаданные типов.

Работает это так. Исходный код на X проходит через компилятор X, который генерирует инструкции IL и метаданные. Другими словами, все компиляторы всех поддерживаемых .NET языков генерируют одинаковые инструкции IL и метаданные.

При создании DLL- или EXE-файла с помощью .NET-компилятора получаемый большой двоичный объект называется сборкой (assembly). Подробно сборки будут рассмотрены в других главах этой книги, а сейчас нужно рассказать хотя бы об основных свойствах этого формата файлов.

В сборке содержится CIL-код, который похож на байт-код Java тем, что не компилируется в ориентированные на определенную платформу инструкции до тех пор, пока это не становится действительно необходимым. Обычно этот момент наступает, когда к какому-то блоку CIL-инструкций выполняется обращение для его выполнения в среде .NET.

Кроме CIL-инструкций, в сборках есть также метаданные, описывающие особенности каждого имеющегося внутри данной двоичной .NET-единицы “типа”.

Сами сборки, кроме CIL и метаданных типов, также описываются с помощью метаданных, называемых манифестом (manifest). В каждом манифесте содержатся информация о текущей версии сборки, сведения о культуре (применяемые для локализации строковых и графических ресурсов) и перечень ссылок на все внешние сборки, которые требуются для правильного функционирования (зависимости).

Сборки бывают однофайловыми и многофайловыми. В большинстве случаев между сборками .NET и файлами двоичного кода (.dll или .exe) соблюдается простое соответствие “один к одному”. Поэтому при построении DLL-библиотеки .NET можно полагать, что файл двоичного кода и сборка — это одно и то же. Однако это не совсем так. С технической точки зрения, сборка, состоящая из одного модуля (.dll или .exe), называется однофайловой. В такой сборке все необходимые CIL-инструкции, метаданные и манифесты содержатся в одном автономном, четко определенном пакете.

Многофайловые сборки состоят из множества файлов двоичного кода .NET, каждый из которых называется модулем (module). При построении многофайловой сборки в одном из ее модулей содержится манифест всей самой сборки, а во всех остальных — манифест, CIL-инструкции и метаданные типов, охватывающие уровень только соответствующего модуля. В главном

модуле содержится описание набора требуемых дополнительных модулей внутри манифеста сборки. Сборки будут рассмотрены в главе 9.

1.9. Подробно о CTS

В каждой конкретной сборке может содержаться любое количество самых разных типов. В мире .NET "тип" - это просто общий термин, который может использоваться для обозначения любого элемента из множества (класс, интерфейс, структура, перечисление, делегат).

При построении решений .NET, скорее всего, придется взаимодействовать со многими из этих типов. Так, в сборке может содержаться один класс, реализующий определенное количество интерфейсов, метод одного из которых может принимать в качестве параметра перечисление, а возвращать - массив.

CTS представляет собой формальную спецификацию, в которой описано то, как должны быть определены типы для того, чтобы они могли обслуживаться в CLR-среде. Всем .NET-разработчикам важно уметь работать на предпочитаемом ими языке с пятью типами из CTS. Далее приводится краткий обзор этих типов.

1.9.1. Типы классов

В каждом совместимом с .NET языке поддерживается, как минимум, понятие типа класса (class type), которое играет центральную роль в объектно-ориентированном программировании (далее - ООП). Каждый класс может содержать произвольное количество членов (конструкторы, свойства, методы и события) и точек данных (полей). В C#, как и в других языках программирования, для объявления класса используется ключевое слово **class**:

```
class Car
{
    public int Run()
    { return 1; }
}
```

В таблице 1.3 приводится краткий перечень характеристик, свойственных типам классов.

Таблица 1.3. Характеристики классов CTS

Характеристика	Описание
Степень видимости	Каждый класс должен настраиваться с атрибутом видимости (visibility). По сути, данный атрибут указывает, должен ли класс быть доступным внешним сборкам или его можно использовать только внутри определенной сборки.
Абстрактные и конкретные классы	Экземпляры абстрактных классов не могут создаваться напрямую и предназначены для определения общих аспектов поведения для произвольных типов. Экземпляры конкретных классов могут создаваться напрямую.
Запечатанные	Запечатанные (sealed) классы не могут выступать в роли базовых для других классов, то есть не поддерживают наследия.
Реализующие интерфейсы	Интерфейс (interface) - это коллекция абстрактных членов, которые обеспечивают возможность взаимодействия между объектом и пользователем этого объекта. CTS позволяет реализовать в классе любое количество интерфейсов.

1.9.2. Типы интерфейсов

Интерфейсы представляют собой именованную коллекцию определений абстрактных членов, которые могут поддерживаться в данном классе или структуре. В C# типы интерфейсов определяются с помощью ключевого слова `interface`. Пример:

```
public interface IDrive
{
    void Press();
}
```

От самих интерфейсов проку мало. Однако, если они реализованы в классах или структурах, они позволяют получить доступ к дополнительным функциональным возможностям за счет добавления просто ссылки на них в полиморфной форме.

1.9.3. Типы структур

В CTS есть понятие структуры. Если вы раньше программировали на Си, то будете приятно удивлены, что в современном мире .NET нашлось место для вашего любимого типа данных. Структура может считаться "облегченной" версией класса. Обычно структуры лучше подходят для моделирования математических данных. В C# структуры определяются ключевым словом `struct`:

```
struct Nums
```

```

{
    public int xs, ys;
    public Nums(int x, int y) { xs = x; ys = y; }
    public int Add()
    {
        return xs + ys;
    }
}

```

Обратите внимание: структуры могут содержать конструкторы и методы - подобно классам.

1.9.4. Типы перечислений

Перечисления (enumeration) - удобная программная конструкция, позволяющая сгруппировать данные в пары "имя-значение". Например:

```

public enum eNums
{
    A = 1,
    B = 2,
    C = 3
}

```

По умолчанию для хранения каждого элемента выделяется блок памяти, который соответствует 32-битному целому, но при необходимости (например, если нужно экономить оперативную память при создании приложений для устройств с небольшим объемом ОЗУ) это значение можно изменить. Кроме того, в CTS сделано так, чтобы перечисляемые типы наследовались от общего базового класса System.Enum.

1.9.5. Типы делегатов

Делегаты (delegate) - являются .NET-эквивалентом безопасных в отношении типов указателей функций в стиле Си. Основное отличие заключается в том, что делегат в .NET представляет собой класс, который наследуется от System.MulticastDelegate, а не просто указатель на какой-то конкретный адрес в памяти. Объявить делегат можно с помощью ключевого слова **delegate**:

```

public delegate int AddOp(int x, int y);

```

Делегатов удобно использовать, когда нужно обеспечить одну сущность возможностью перенаправлять вызов другой сущности и образовывать основу для архитектуры обработки событий .NET.

1.9.6. Встроенные типы данных

В CTS также содержится четко определенный набор фундаментальных типов данных. В каждом отдельно взятом языке для объявления того или иного встроенного типа данных из CTS обычно предусмотрено свое уникальное ключевое слово. В таблице 1.4 показано, какие ключевые слова в разных языках соответствуют типам данных в CTS.

Таблица 1.4. Встроенные типы данных в CTS

CTS	C#	C++	Visual Basic
System.ByteByte	byte	unsigned char	Byte
System.SByteSByte	SByte	sbyte	signed char
System.Int16	short	short	Short
System.Int32	int	int или long	Integer
System.Int64	long	__int64	Long
System.UInt16	ushort	unsigned short	UShort
System.UInt32	uint	unsigned int	UInteger
System.UInt64	ulong	unsigned __int64	ULong
System.SingleSingle	float	float	Single
System.DoubleDouble	double	double	Double
System.ObjectObject	object	object^	Object
System.CharChar	char	wchar_t	Char
System.StringString	String	String^	String
System.DecimalDecimal	decimal	Decimal	Decimal
System.BooleanBoolean	bool	bool	Boolean

При этом в C# можно указать названия типов из CTS:

```
long x=0;
System.Int64 y = 0;
```

1.10. Подробно о CLS

CLS (Common Language Specification — общая спецификация для языков программирования) представляет собой набор правил, которые подробно описывают минимальный и полный набор функциональных возможностей, которые должен обязательно поддерживать каждый отдельно взятый .NET-

компилятор, чтобы генерировать такой программный код, который мог бы обслуживаться CLR.

CLS можно считать просто подмножеством всех функциональных возможностей, определенных в CTS. В конечном итоге CLS является своего рода набором правил, которых должны придерживаться создатели компиляторов, если они хотят чтобы их продукты могли без особых проблем функционировать в мире .NET.

У каждого такого правила есть простое название, и оно описывает, как именно его действие касается тех, кто создает компиляторы, и тех, кто будет с ними взаимодействовать.

Самое главное правило в CLS гласит, что правила CLS касаются только тех частей типа, которые делаются доступными за пределами сборки, в которой они определены. Отсюда следует, что все остальные правила в CLS не распространяются на логику, применяемую для построения внутренних рабочих деталей типа .NET.

Самое интересное, что в самом C# есть целый ряд программных конструкций, не соответствующих правилам CLS. Однако компилятор C# можно заставить выполнять проверку программного кода на предмет соответствия правилам CLS с помощью атрибута .NET:

```
[assembly: System.CLSCompliant(true)]
```

Атрибут [CLSCompliant] заставляет компилятор C# проверять каждую строку кода на предмет соответствия правилам CLS. Если будут обнаруже-

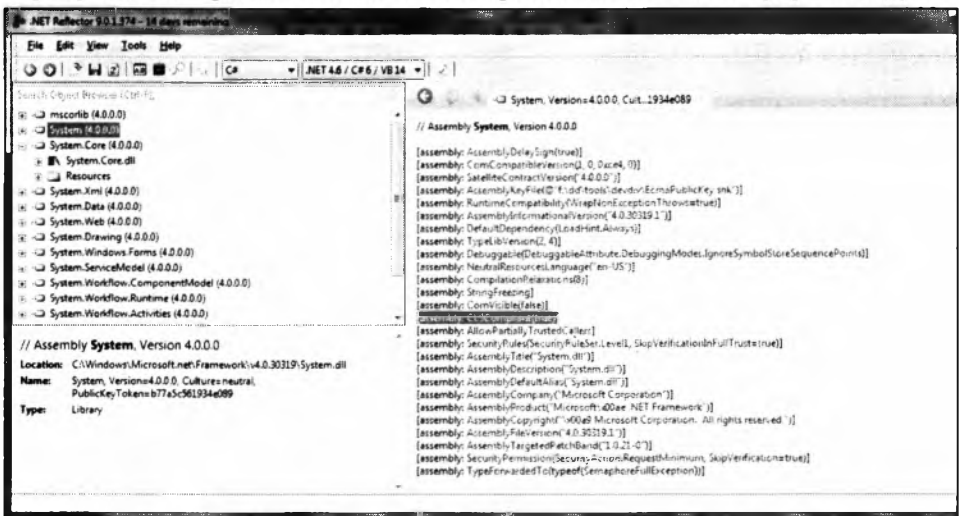


Рис. 1.2. Reflector - утилита для декомпиляции объектов .NET

ны нарушения каких-либо правил CLS, компилятор выдаст ошибку и описание вызвавшего ее кода. На рис. 1.2 показано, что атрибут [CLSCompliant] задан для сборки System.Dll. Утилиту Reflector можно скачать по адресу <http://www.red-gate.com/products/reflector>.

1.11. Подробно о CLR

CLR (Common Language Runtime) - общезыковая исполняющая среда. CLR можно расценивать как коллекцию внешних служб, которые необходимы для выполнения скомпилированной единицы программного кода. Например, при использовании платформы MFC для создания нового приложения программист осознает, что приложению понадобится библиотека времени выполнения MFC (то есть mfc40.dll).

Аналогично и с другими языками. VB-программист понимает, что должен привязываться к одному или двум модулям исполняющей среды (msvbvm60.dll), а Java-программисты привязаны к виртуальной машине Java (JVM).

В составе .NET есть еще одна исполняющая среда. Основное отличие между исполняющей средой .NET и вышеупомянутыми средами заключается в том, что исполняющая среда .NET обеспечивает единый четко определенный уровень выполнения. Этот уровень выполнения способны использовать все совместимые с .NET языки и платформы.

Физически основной механизм CLR представляет собой библиотеку под названием mscorée.dll. Название MSCOREE - это аббревиатура от Microsoft Common Object Runtime Execution Engine - общий механизм выполнения исполняемого кода объектов.

Когда добавляется ссылка на сборку, предварительно автоматически загружается библиотека mscorée.dll, а уже после этого загружается сборка в память и происходит ее выполнение.

Механизм исполняющей среды отвечает за выполнение целого ряда задач. Первым делом отвечает за определение места расположения сборки и обнаружение запрашиваемого типа в двоичном файле за счет считывания содержащихся там метаданных. Также он размещает тип в памяти, преобразует CIL-код в соответствующие платформе инструкции, производит проверки безопасности и запрашивает программный код.

1.12. Пространства имен

Все обилие типов, которое можно использовать в .NET, содержится в многочисленных пространствах (namespaces) имен .NET. Основное пространство

имен, с которого нужно начинать знакомство с пространствами, называется `System`. В нем содержится набор ключевых типов, с которыми любой разработчик .NET будет иметь дело снова и снова. Создание функционального приложения на C# невозможно без добавления ссылки хотя бы на пространство имен `System`, поскольку все основные типы данных (`System.Int32`, `System.Boolean`) определены именно в нем. В таблице 1.5 приводятся некоторые пространства имен .NET.

Таблица 1.5. Некоторые пространства имен в .NET

Пространство имен	Описание
<code>System</code>	Содержит много полезных типов, позволяющих работать с математическими вычислениями, генератором случайных чисел, переменными среды и т.д.
<code>System.Collections</code>	Содержит ряд контейнерных типов, а также несколько базовых типов и интерфейсов, позволяющих создавать специальные коллекции
<code>System.Data</code>	Используется для взаимодействия с базами данных через ADO.NET
<code>System.IO</code>	Здесь содержится много типов, предназначенных для работы с операциями файлового ввода/вывода, сжатия данных, портами
<code>System.Reflection</code>	Содержит типы, которые поддерживают обнаружение типов во время выполнения, а также динамическое создание типов
<code>System.Runtime</code>	Содержит средства, позволяющие взаимодействовать с неуправляемым кодом, точнее, эти средства находятся в <code>System.Runtime.InteropServices</code>
<code>System.Drawing</code> <code>System.Windows.Forms</code>	Содержит типы, применяемые для построения настольных приложений (Windows Forms)
<code>System.Windows</code>	Является корневым для нескольких пространств имен, предоставляющих набор графических инструментов WPF (Windows Presentation Foundation)
<code>System.Linq</code>	Содержит типы, применяемые при выполнении программирования с использованием API LINQ
<code>System.Web</code>	Позволяет создавать веб-приложения ASP.NET
<code>System.ServiceModel</code>	Позволяет создавать распределенные приложения с помощью API-интерфейса Windows Communication Foundation (WCF)
<code>System.Xml</code>	Здесь содержатся многочисленные типы, которые используются при работе с XML-данными
<code>System.Security</code>	Типы, имеющие дело с разрешениями, криптографией и т.д.

System.Threading	Средства для создания многопоточных приложений, способных разделить нагрузку среди нескольких процессоров
System.Workflow	Типы для построения поддерживающих рабочие потоки приложений с помощью API-интерфейса Windows Workflow Foundation (WWF)

Для подключения пространства имен в C# используется ключевое слово `using`, например:

```
using System;  
using System.Drawing;  
using System.Windows.Forms;
```

На этом все. В следующей главе мы поговорим о развертывании среды .NET и о создании приложений на языке C#.

Глава 2.

Развертывание .NET и первая программа



2.1. Развертывание у заказчика

Как узнать, какая версия .NET Framework установлена у заказчика? Можно воспользоваться специальными программами для определения версии .NET Framework. Одна из таких программ - ASoft .NET Version Detector, скачать которую совершенно бесплатно можно по адресу:

<http://net-framework.ru/soft/asoft-net-version-detector>

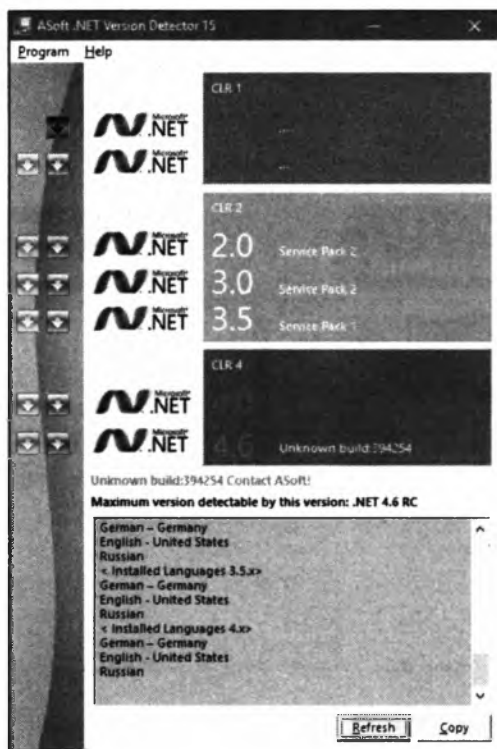


Рис. 2.1. Программа ASoft .NET Version Detector

Внимание! Загружать .NET Framework нужно только с официального сайта. Что платформа .NET Framework, что пакет разработчика Visual Studio Community Edition - совершенно бесплатны. Для их загрузки никакая регистрация не нужна. Поэтому нет никакого смысла загружать их со сторонних сайтов - очень часто таким образом распространяются вирусы и прочие вредоносные программы - вместе с загружаемым продуктом вы получаете "в нагрузку" вирус.

Однако текущая версия этой программы сообщает, что номер версии .NET не определен, поскольку она просто не умеет определять версии выше 4.6 RC. Можно попытаться найти другую программу, а можно заглянуть в реестр.

Гораздо проще открыть редактор реестра (regedit.exe) и обратиться к ветке HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP. Подразделы этой ветки с именами, начинающимися на "v.", позволяют судить об установленных версиях платформы .NET.

Посмотрим, так ли это. На рис. 2.2 показано, что установлены версии 2.0, 2.0, 2.5, 4.0. Кстати, это установка Windows 10 Enterprise по умолчанию, безо всяких обновлений.

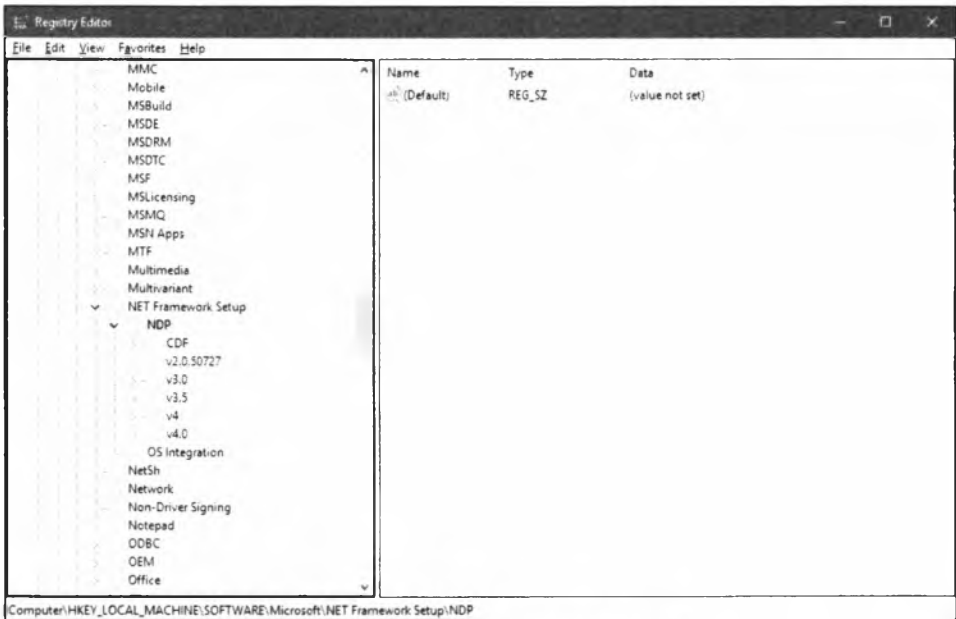


Рис. 2.2. Установленные версии .NET

Вот только судить только по названиям веток неправильно. Откройте подраздел v4\Full и посмотрите на значения параметров Release и Version. Первый параметр означает номер релиза .NET Framework - по нему можно более точно определить номер версии. Однако гораздо проще взглянуть на параметр Version - на рис. 2.3 видно, что установлена версия 4.6.01038.

Можно вообще ничего не определять, а просто скачать инсталлятор последней версии (на данный момент это 4.7.1). В случае, если установлена эта или более поздняя версия, инсталлятор сообщит вам об этом (рис. 2.4).

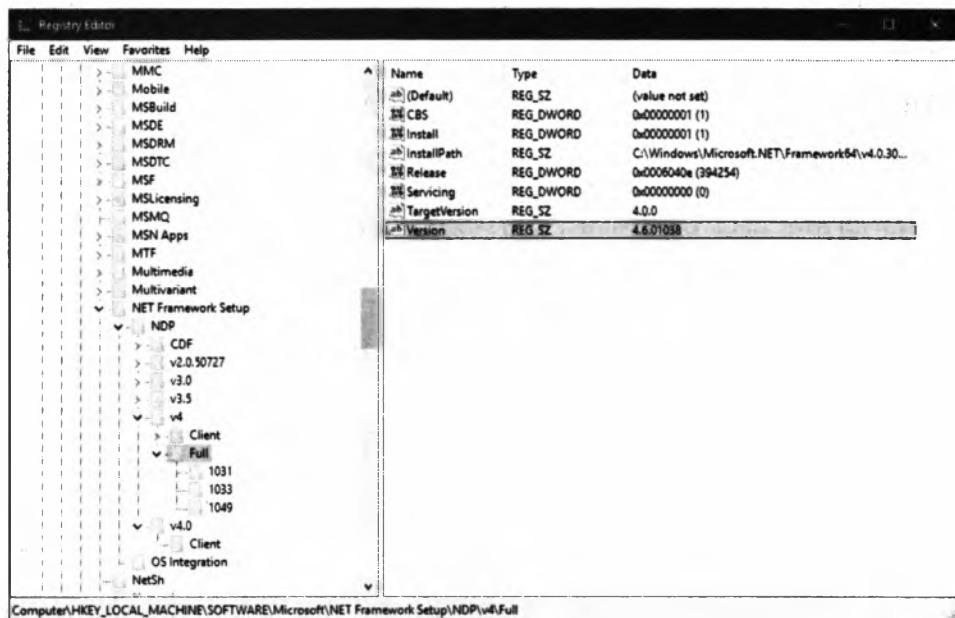


Рис. 2.3. Правильное определение версии .NET

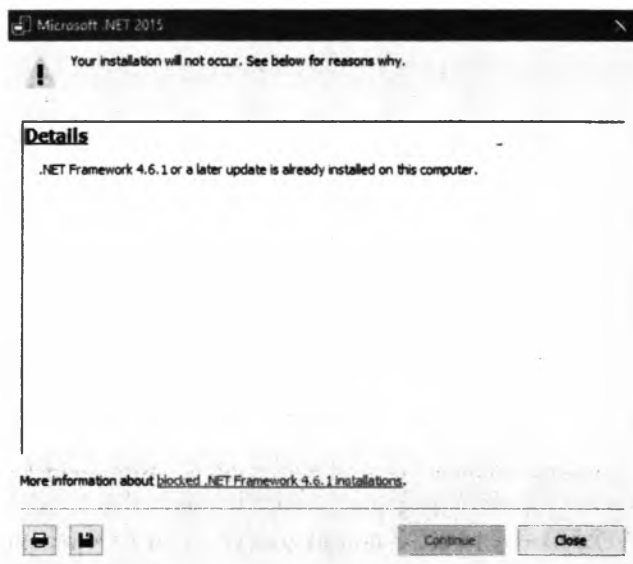


Рис. 2.4. Установщик .NET Framework 4.6.1: установлена более новая версия

В коде программы определить версию .NET можно так, как показано в листинге 2.1. Именно этот код приводится на сайте <https://msdn.microsoft.com> и рекомендуется для определения версии .NET самой Microsoft.

Листинг 2.1. Определение версии .NET Framework на C#

```
using System;
using Microsoft.Win32;
...
private static void GetVersionFromRegistry()
{
    // Открываем раздел реестра и читаем версию .NET
    using (RegistryKey ndpKey =
        RegistryKey.OpenRemoteBaseKey(RegistryHive.LocalMachine, "").
        OpenSubKey(@"SOFTWARE\Microsoft\NET Framework Setup\NDP\"))
    {
        // В качестве альтернативы, если установлена версия 4.5 или выше,
        // можно использовать следующий запрос
        // (RegistryKey ndpKey =
        // RegistryKey.OpenBaseKey(RegistryHive.LocalMachine,
        // RegistryView.Registry32).
        // OpenSubKey(@"SOFTWARE\Microsoft\NET Framework Setup\NDP\"))
        foreach (string versionKeyName in ndpKey.GetSubKeyNames())
        {
            if (versionKeyName.StartsWith("v"))
            {
                RegistryKey versionKey = ndpKey.OpenSubKey(versionKeyName);
                string name = (string)versionKey.GetValue("Version", "");
                string sp = versionKey.GetValue("SP", "").ToString();
                string install = versionKey.GetValue("Install", "").ToString();
                if (install == "") //no install info, must be later.
                    Console.WriteLine(versionKeyName + " " + name);
                else
                {
                    if (sp != "" && install == "1")
                    {
                        Console.WriteLine(versionKeyName + " " + name + " SP" + sp);
                    }
                }
                if (name != "")
                {
                    continue;
                }
                foreach (string subKeyName in versionKey.GetSubKeyNames())
                {
                    RegistryKey subKey = versionKey.OpenSubKey(subKeyName);
                    name = (string)subKey.GetValue("Version", "");
                    if (name != "")
                        sp = subKey.GetValue("SP", "").ToString();
                    install = subKey.GetValue("Install", "").ToString();
                    if (install == "") //нет инфо об установке
                        Console.WriteLine(versionKeyName + " " + name);
                }
            }
        }
    }
}
```

```

else
{
    if (sp != "" && install == "1")
    {
        Console.WriteLine(" " + subKeyName + " " + name + " SP" + sp);
    }
    else if (install == "1")
    {
        Console.WriteLine(" " + subKeyName + " " + name);
    }
}
}
}
}

```

Если у заказчика Windows 10, то, скорее всего, обновлять .NET Framework не придется, поскольку будет установлена одна из последних версий. В более старых выпусках Windows могут быть установлены старые версии платформы, поэтому вполне вероятно, что придется обновлять .NET Framework.

Как обновить .NET Framework? Для этого произведите поиск в Google по поисковой фразе ".NET Framework 4.6 download" (вы же помните, что загружать .NET Framework можно только с сайта Microsoft) или перейдите по одному из следующих адресов:

<https://www.microsoft.com/ru-ru/download/details.aspx?id=48130>

<https://www.microsoft.com/ru-ru/download/details.aspx?id=49982>

Первый адрес - это веб-инсталлятор платформы. Вы загружаете небольшой файл, запускаете его, а все необходимые файлы он получает с Интернета. Второй адрес - это offline-установщик. На рис. 2.5 изображена страница загрузки offline-инсталлятора. Будет загружен файл NDP461-KB3102436-x86-x64-AllOS-ENU.exe размером 66 Мб. Запустите его и следуйте инструкциям мастера установки. По окончании установки понадобится перезагрузка компьютера.

Ясное дело, что когда у вас будет серьезный проект, вы создадите для него инсталлятор, который будет автоматически загружать нужную версию .NET Framework. Поэтому данный раздел предназначен больше не для заказчика, а для вас - программиста. Ведь пока вы еще не создали этот самый серьезный проект, то и инсталлятор вам еще не нужен. Но вам нужно протестировать свою программу на другой машине (отличной от той, на которой вы ведете разработку), а без .NET Framework она не запустится. Поэтому вы должны знать, что и откуда можно скачать, чтобы ваша программа работала.

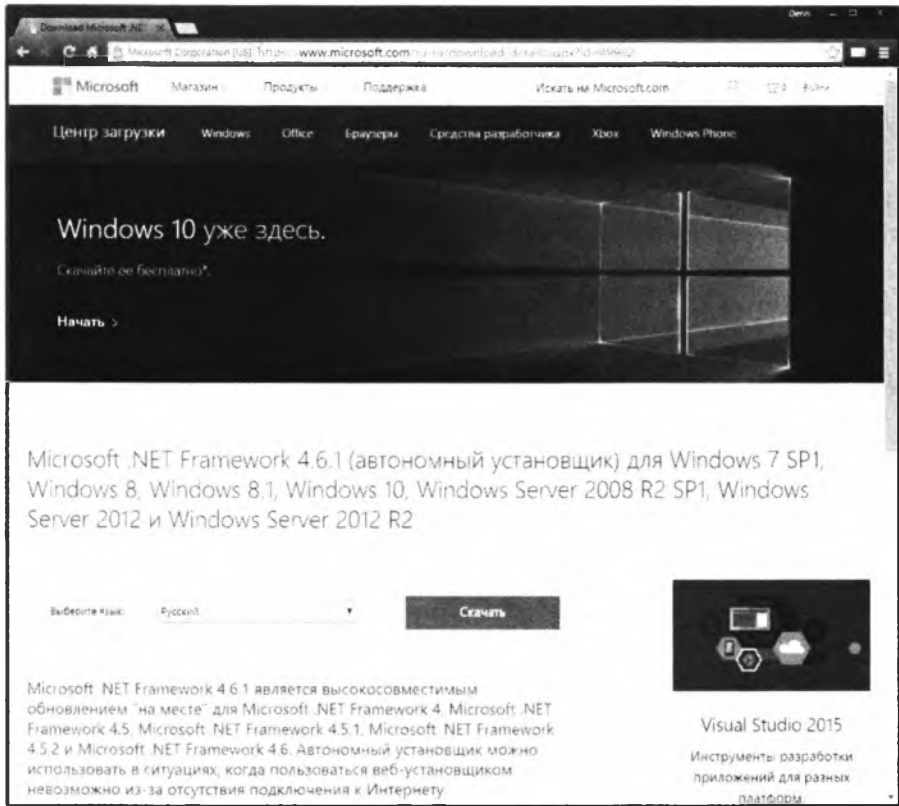


Рис. 2.5. Страница загрузки offline-инсталлятора

2.2. Развертывание у программиста. Установка Visual Studio Community

На компьютере программиста вам нужно установить .NET Framework SDK, а чтобы созданные вами приложения могли работать на компьютере заказчика, на нем нужно установить .NET Framework, желательно той же версии (или чтобы версия была новее, чем ваш SDK).

По сути, .NET Framework SDK - это единственное, что вам нужно установить для начала разработки приложений .NET. Раньше можно было скачать сам .NET Framework SDK и не устанавливать какую-либо IDE. В результате вы могли бы писать программы прямо в блокноте или любом другом текстовом редакторе и компилировать с помощью компилятора CSC.EXE.

Конечно, для работы с серьезным проектом IDE понадобится - она значительно упрощает управление файлами проекта, отладку и много других

процессов, происходящих при создании программного обеспечения. Я уже молчу о возможности быстрой разработки форм. Разработать форму графического приложения в блокноте будет очень сложно. Поэтому без среды все равно не обойтись. В Microsoft это понимают, и поэтому сейчас загрузка .NET Framework SDK невозможна без установки Visual Studio.

Но ведь раньше можно было установить .NET Framework SDK совершенно бесплатно, а как же сейчас? Дискриминация?! Нет, в Microsoft выпустили бесплатную версию Visual Studio - Community, с которой вы познакомитесь чуть позже. Если вы уже установили Visual Studio, то устанавливать .NET Framework SDK вам не нужно, так как он уже входит в ее состав.

Перейдите по адресу <http://msdn.microsoft.com/en-us/vstudio/aa496123>. На этой же страничке можно скачать не только SDK, но и установщики самой платформы разных версий (см. рис. 2.6).

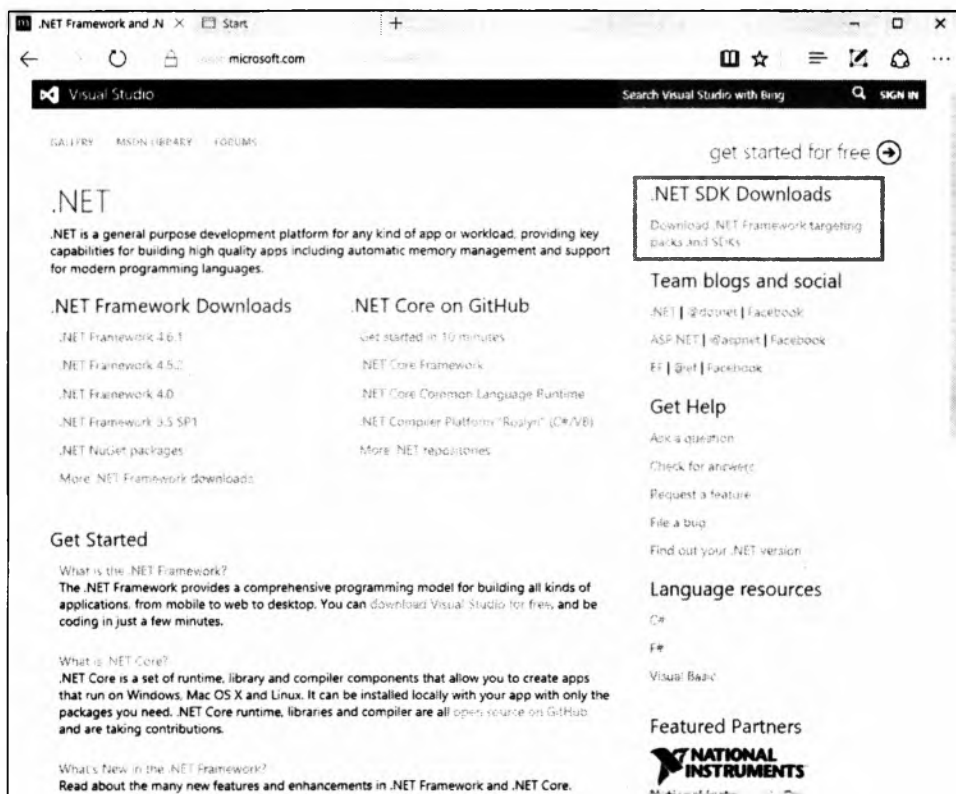


Рис. 2.6. Страница загрузки .NET Framework

Сейчас нас больше интересует раздел .NET SDK Downloads. В нем вы можете скачать бесплатную версию Visual Studio Community. В отличие от полной версии, данная версия Visual Studio совершенно бесплатна. Перейти сразу к загрузке Visual Studio Community можно по адресу:

<http://www.visualstudio.com/products/visual-studio-community-vs>

Вы скачаете файл vs_community_ENU.exe. Инсталлятор очень небольшого размера - ведь загрузка необходимых файлов осуществляется с сервера Microsoft, поэтому не отключайте Интернет во время установки Visual Studio.

Примечание. Понятное дело, что Visual Studio Community - несколько ограниченная версия. Не нужно быть гением, чтобы догадаться. Но для начинающих программистов (а вы таким и являетесь, иначе бы не читали эту книгу) и обучению программированию возможностей этой версии будет достаточно. Более того, даже в этой книге не будут раскрыты все возможности этой IDE (ведь она позволяет разрабатывать приложения не только для Windows, но и для других платформ), поэтому смело устанавливайте Visual Studio Community и не думайте ни о каких ограничениях! Когда вам понадобится платная версия, вы об этом узнаете. А пока нет смысла тратить деньги на те функции, которыми вы не будете пользоваться.

На рис. 2.7 показан инсталлятор Visual Studio Community. Установка по умолчанию занимает 8 Гб - это минимальная установка. Можно установить ее, а можно выбрать **Custom** и установить дополнительные компоненты (рис. 2.8).



Рис. 2.7. Инсталлятор Visual Studio Community



Рис. 2.8. Выбираем компоненты Visual Studio Community

В зависимости от расторопности вашего компьютера, установка Visual Studio может занимать от нескольких десятков минут до пары часов. При первом запуске среда предложит вам подключиться к сервисам для разработчиков - вы можете получить собственный Git-репозиторий (систем управления версиями), синхронизировать свои настройки и другие возможности (рис. 2.9). Вы можете зарегистрироваться прямо сейчас, а можете - при следующем запуске IDE.



Рис. 2.9. Предложение зарегистрироваться



Рис. 2.10. Выбор темы оформления

Далее Visual Studio предложит выбрать тему (ее также можно будет изменить позже) - синяя (по умолчанию), темная или светлая (рис. 2.10). Хотя по умолчанию используется синяя тема, для этой книги с целью улучшения иллюстраций будет использована светлая.

2.3. Первая программа с использованием Visual Studio

Настало время создать наше первое приложение на C#. Основное окно Visual Studio изображено на рис. 2.11. Для создания нового проекта воспользуйтесь ссылкой **New Project** или выберите команду меню **File ► New ► Project**. В окне **New Project** выберите тип приложения. Наиболее часто используемый тип **Windows Forms Application** - обычное приложение с

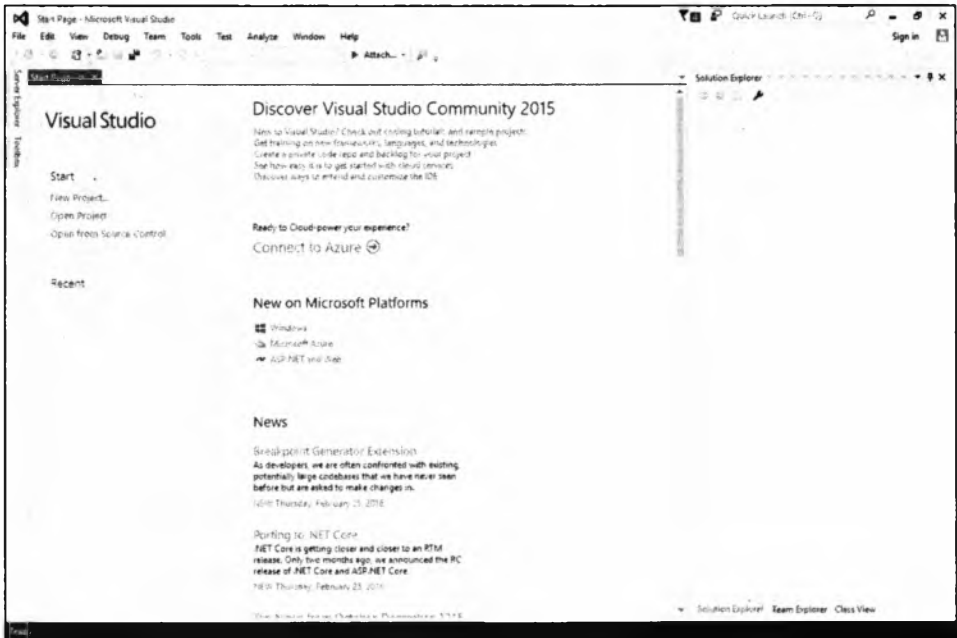


Рис. 2.11. Основное окно

графическим интерфейсом (формой). Но для первого приложения выберите **Console Application**.

Введите название проекта - **Hello** и нажмите кнопку **ОК**. Созданный проект показан на рис. 2.12. Среда подготовила основной код для нашего консоль-



Рис. 2.12. Окно создания нового проекта

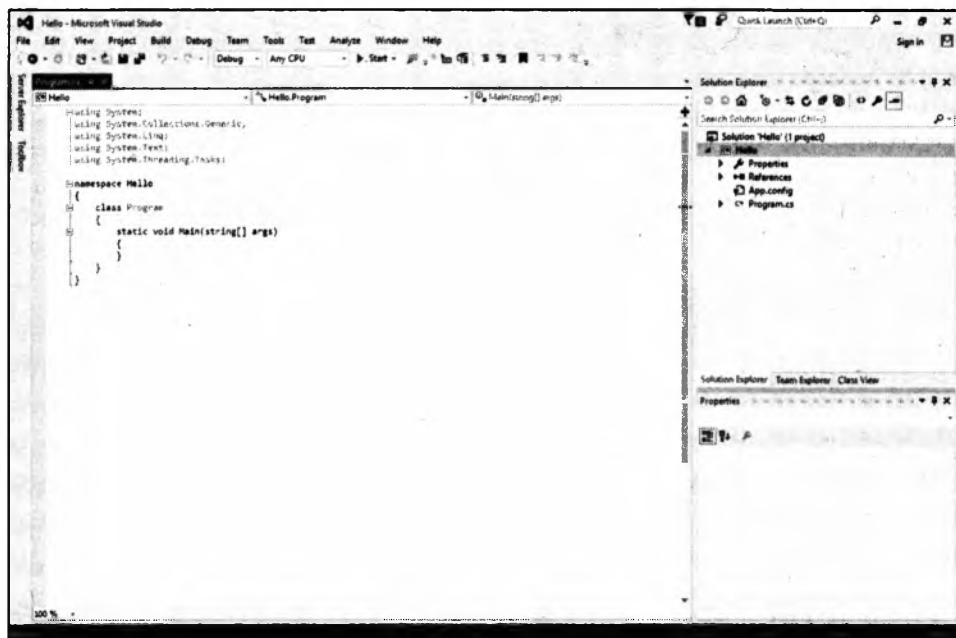


Рис. 2.13. Созданный проект

ного приложения. Изменим его так, как показано в листинге 2.2. Особо это приложение мы изменять не станем, а лишь добавим вывод строки "Hello, world!" (не будем изменять традиции) на консоль.

Листинг 2.2. Приложение Hello, world!

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

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

Примечание. По умолчанию среда подключает избыточные пространства имен. Для нашего очень "сложного" проекта хватило бы одного пространства имен - System.

Откомпилируем проект. Для этого выполните команду **Build ► Build Hello** (если вы ввели другое имя проекта, то вместо Hello будет введенная вами строка). В нижней части окна вы увидите отчет компилятора, а в нем - местоположение результирующего exe-файла (рис. 2.14).

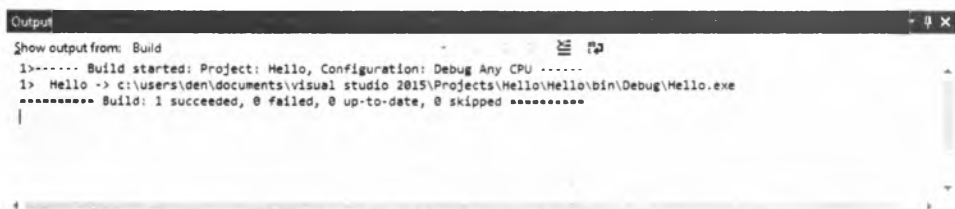


Рис. 2.14. Вывод компилятора

Для запуска проекта можно нажать кнопку **Start** на панели инструментов. Однако будет открыто окно командной строки, которое моментально закроется, как только завершится выполнение программы, а поскольку она выводит только одну строку, то вы и глазом не успеете моргнуть, как окно закроется. Поэтому откройте командную строку и перейдите в каталог проекта, содержащий исполнимый файл (в нашем случае это c:\users\<имя>\documents\visual studio 2015\Projects\Hello\Hello\bin\Debug\). Результат выполнения нашей программы приведен на рис. 2.15.



Рис. 2.15. Результат выполнения hello.exe

Вот теперь у вас есть понимание, что такое .NET Framework, и у вас есть среда для написания .NET-приложений. Значит, вы полностью готовы к дальнейшему изучению .NET-программирования.

Глава 3.

Основные конструкции языка C#



В данной главе мы рассмотрим основные конструкции языка C#, в которых вам нужно разобраться, чтобы успешно изучить платформу .NET Framework. Будет показано, как создать объект приложения, как создать структуру метода Main(), который является точкой входа (entry point) в любой исполняемой программе. Также рассматриваются основные типы данных в C#. Данная глава больше рассчитана на бывших программистов, нежели совсем на новичка. Повторюсь, что книга не является справочником по C#.

3.1. Исследование программы Hello, world!

3.1.1. Пространства имен, объекты, методы

В предыдущей главе нами была написана очень простая программа, выводящая строку Hello, world! Чтобы вы не листали книгу, привожу ее код еще раз - в листинге 3.1.

Листинг 3.1. Простая программа

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

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

Давайте разберемся, что есть что. В языке C# вся логика программы должна содержаться внутри какого-то типа. Тип - это общий термин, которым можно обозначить любой элемент множества {класс, интерфейс, структура, перечисление, делегат}. В отличие от других языков программирования, в C# невозможно создать ни глобальную функцию, ни глобальный элемент данных. Вместо этого нужно, чтобы все данные и методы находились внутри определения типа.

Посмотрите на листинг 3.1. В нем создано пространство имен **Hello**. Внутри него объявлен класс **Program**, в котором есть метод **Main()**, выводящий строку на экран. Для простых программ такой подход кажется запутанным, но все окупится, когда приложения станут отнюдь не простыми.

Важным элементом любой программы являются комментарии. В С# используются традиционные комментарии в стиле Си, которые бывают однострочные (`//`) и многострочные (`/* .. */`).

Пример:

```
// это однострочный комментарий
/* это
   многострочный
   комментарий */
```

В первом случае комментарием считаются символы, начинающиеся от `//` и до конца строки. Во втором случае игнорируются все символы между `/*` и `*/`.

Комментарии могут быть встроенными, например:

```
SomeMethod (Width, /*Height*/ 200);
```

Такие комментарии использовать не рекомендуется, так как они могут ухудшить читабельность кода.

Если символы комментария включены в строковый литерал, то они считаются обычным кодом программы, например:

```
Console.WriteLine("//эта строка не будет считаться комментарием");
```

Не стесняйтесь писать комментарии - со временем вы будете благодарны себе за это (когда уже не будете помнить, что есть что в своей программе). Другие программисты, которые будут обслуживать ваш код, также скажут вам спасибо, хотя бы мысленно.

Символ форматирования	Описание
C (или c)	Форматирование денежных значений.
D (или d)	Форматирование десятичных чисел.
E (или e)	Экспоненциальное представление. Регистр символа означает, в каком регистре будет выводиться экспоненциальная константа - в верхнем (E) или в нижнем (e).

F (или f)	Числа с фиксированной точкой. Число после символа означает количество знаков после точки, например, {0:f3} выведет 1.000, если в качестве 0 указать значение, равное 1.
G (или g)	Общий формат.
N (или n)	Базовое числовое форматирование (с запятыми)
X (или x)	Форматирование шестнадцатеричных чисел. Если указан X, то в hex-представлении символы будут в верхнем регистре, например, 1AH, а не 1ah, если указан x.

Примеры:

```
Console.WriteLine("Значение 123456 в разных форматах:");
Console.WriteLine("d7: {0:d7}", 123456);
Console.WriteLine("c: {0:c}", 123456);
Console.WriteLine("n: {0:n}", 123456);
Console.WriteLine("f3: {0:f3}", 123456);
```

Результат будет таким, как показано на рис. 3.4.

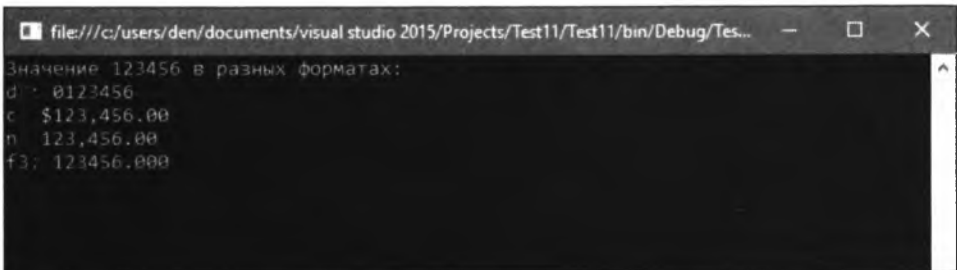


Рис. 3.4. Значение в разных форматах

Не нужно думать, что приведенные сведения, раз они предназначены для консоли, то не пригодятся вам при работе с графическим интерфейсом. Отформатировать можно любую строку, а потом вывести ее средствами графического интерфейса:

```
string Msg = string.Format("d7: {0:d7}", 123456);
Windows.Forms.MessageBox.Show(Msg);
```

Метод `string.Format()` понимает тот же формат форматирования, что и метод `WriteLine()`. Далее отформатированную строку можно использовать, как вам будет нужно. В данном случае мы выводим ее в `MessageBox`.

3.3. Типы данных и переменные

3.3.1. Системные типы данных

В любом языке программирования есть собственный набор основных (системных) типов данных. Язык C# в этом плане - не исключение. Но в отличие от Си, в C# эти ключевые слова - не просто лексемы, распознаваемые компилятором. Они представляют собой сокращенные варианты обозначения полноценных типов из пространства имен System. Например, тип **bool** - это системный тип System.Boolean. Таблица 3.2 содержит информацию о системных типах языка C#. Обратите внимание на колонку CLS: она означает, отвечает ли тип требованиям общезыковой спецификации (CLS). Как уже упоминалось ранее (см. гл. 2), если вы в программе используете типы данных, которые не соответствуют CLS, другие языки не смогут их использовать.

Таблица 3.2. Системные типы данных C#

Тип в C#	Системный тип	Диапазон	CLS	Описание
bool	Syste.Boolean	true, false	Да	Логическое (булево) значение
sbyte	System.SByte	-128...127	Нет	8-битное число со знаком
byte	System.Byte	0...255	Да	8-битное число без знака
short	System.Int16	-32 768...32 767	Да	16-битное число со знаком
ushort	System.Int16	0...65 535	Нет	16-битное число без знака
int	System.Int32	-2 147 483 648 ... 2 147 483 647	Да	32-битное число со знаком
uint	System.UInt32	0...4 294 967 295	Нет	32-битное число без знака
long	System.Int64	-9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807	Да	64-битное число со знаком
ulong	System.UInt64	0... 18 446 744 073 709 551 615	Нет	64-битное число без знака

char	System.Char	U+0000... U+FFFF	Да	Один символ (16 бит, Unicode)
float	System.Single	$+1,5 \times 10^{-45} \dots$ $3,4 \times 10^{38}$	Да	32-битное число с плавающей точкой
decimal	System.Decimal	$\pm 1,0 \times 10^{-28} \dots$ $-7,9 \times 10^{28}$	Да	96-битное число со знаком
double	System.Double	$5,0 \times 10^{-324} \dots$ $1,7 \times 10^{308}$	Да	64-битное число с плавающей точкой
string	System.String	Ограничен объемом до- ступной па- мяти	Да	Ряд символов в коди- ровке Unicode
object	System.Object	Используется для хранения любого типа в памяти	Да	Базовый класс для всех типов в .NET

3.3.2. Объявление переменных

Объявление переменной осуществляется так же, как и в Си: сначала нужно указать тип переменной, а затем - ее имя:

```
int X;
```

После объявления переменной ее нужно инициализировать (присвоить значение) - до первого использования:

```
x = 0;
```

В случае использования локальной переменной до присваивания ей начального значения компилятор сообщит об ошибке.

Инициализировать переменную можно при объявлении:

```
int x = 0;
```

При желании можно в одной строке объявить и инициализировать сразу несколько переменных:

```
int x = 1; y = 0;
```

```
int a, b, c;
```

3.3.3. Внутренние типы данных

Все внутренние типы данных поддерживают конструктор по умолчанию, что позволяет создавать переменные с помощью использования ключевого слова **new** и устанавливать для них значения, которые являются принятыми для них по умолчанию:

- Для переменных типа `bool` - значение `false`;
- Для переменных числовых типов - значение `0`;
- Для типа `string` - один пустой символ;
- Для типа `DateTime` - `1/1/0001 12:00:00`;
- Для объектных ссылок - значение `null`.

Примеры:

```
bool b = new bool();      // будет присвоено значение false
int x = new int();        // будет присвоено значение 0
```

Обычно с помощью **new** создаются объекты в классическом их понимании. Для числовых/булевых/строковых переменных обычно проще указать значения при инициализации, чем использовать **new**:

```
bool b = false;
int x = 0;
```

3.3.4. Члены типов данных

Числовые типы в .NET поддерживают свойства `MaxValue` и `MinValue`, позволяющие получить информацию о допустимом диапазоне значений типа. Кроме свойств `MaxValue` и `MinValue` тип `Double` поддерживает следующие свойства:

- `Epsilon` - эпсилон;
- `PositiveInfinity` - положительная бесконечность;
- `NegativeInfinity` - отрицательная бесконечность.

Пример:

```
int x = int.MaxValue;
double d = double.Epsilon;
```

Понятное дело, что тип данных `System.Boolean` не поддерживает свойства `MinValue` и `MaxValue`. Но зато они поддерживают свойства `TrueString` и

FalseString, которые, соответственно, содержат строки "True" и "False".
Пример:

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

Текстовые данные в C# представлены типами System.String (или просто string) или System.Char (char). Оба типа хранят данные в кодировке Unicode. Первый тип данных позволяет хранить строку, второй - только один символ.

Тип char поддерживает следующие методы:

- IsDigit() - возвращает true, если переданный символ является десятичной цифрой, в противном случае возвращается false;
- IsWhiteSpace() - возвращает true, если переданный символ является пробельным символом (пробел, табуляция и др);
- IsPunctuation() - возвращает true, если переданный символ является знаком пунктуации.

Пример:

```
Console.WriteLine("{0}", char.IsDigit('1'));
```

О типе **string** мы поговорим в следующем разделе, когда будем рассматривать работу со строками.

3.3.5. Работа со строками

Строки в любом языке программирования являются одним из самых важных типов данных. В C# для строк используется тип **string** (системный тип System.String), а для хранения одиночных символов используется тип **char**.

В других языках программирования строки являются массивами символов. В языке C# строки являются объектами, что будет продемонстрировано далее.

Объявить строку можно так:

```
string <имя переменной> [= "значение"];
```

Значение строковой переменной указывается в двойных кавычках.

В C# можно также использовать и массивы символов, например:

```
char[] carray = {'e', 'x', 'a', 'm', 'p', 'l', 'e'};
```

Превратить массив символов в тип данных string можно так:

```
string str = new string(carray);
```

Члены класса System.String

Настало время рассмотреть тип System.String, предоставляющий набор различных методов для работы с текстовыми данными (см. табл. 3.3). Далее эти методы будут рассмотрены подробно.

Таблица 3.3. Некоторые члены типа System.String

Член	Описание
Length	Свойство, содержащее длину текущей строки
Compare()	Метод, позволяющий сравнить две строки. Статический метод
Contains()	Метод, позволяющий определить, содержится ли в строке определенная подстрока
Equals()	Метод, позволяющий проверить, являются ли две строки эквивалентными
Format()	Метод, использующийся для форматирования строки. Статический метод
Insert()	Позволяет вставить строку внутрь другой строки
PadLeft(), PadRight()	Позволяют дополнить строку какими-то символами, соответственно, слева и справа
Remove()	Используется для удаления символов из строки
Replace()	Замена символов в строке
Split()	Разделение строк на подстроки
Trim()	Удаляет все вхождения определенного набора символов с начала и конца текущей строки
ToUpper(), ToLower()	Создают копию текущей строки, соответственно, в верхнем и нижнем регистре

Базовые операции

Работа с членами System.String осуществляется довольно просто - нужно объявить переменную типа **string** и получить доступ к методам (членам) класса через операцию точки. Но имейте в виду, что некоторые члены System.String представляют собой статические методы и потому должны вызываться на уровне класса, а не объекта.

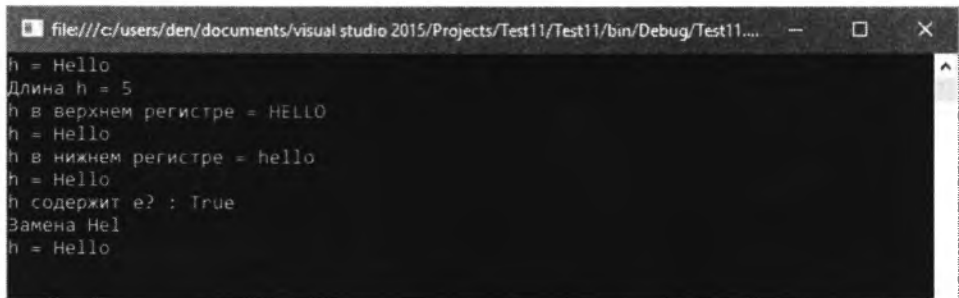
Рассмотрим некоторые примеры:

```
string h = "Hello";
```

```

Console.WriteLine("h = {0}", h);
Console.WriteLine("Длина h = {0}", h.Length);
Console.WriteLine("h в верхнем регистре = {0}", h.ToUpper());
Console.WriteLine("h = {0}", h);
Console.WriteLine("h в нижнем регистре = {0}", h.ToLower());
Console.WriteLine("h = {0}", h);
Console.WriteLine("h содержит e? : {0}", h.Contains("e"));
Console.WriteLine("Замена {0}", h.Replace("lo", ""));
Console.WriteLine("h = {0}", h);
Console.ReadLine();

```



```

file:///c:/users/den/documents/visual studio 2015/Projects/Test11/Test11/bin/Debug/Test11....
h = Hello
Длина h = 5
h в верхнем регистре = HELLO
h = Hello
h в нижнем регистре = hello
h = Hello
h содержит e? : True
Замена Hel
h = Hello

```

Рис. 3.5. Работа со строками

Обратите внимание на вывод этого кода (рис. 3.5). Методы `ToLower()`, `ToUpper()`, `Replace()` и другие не изменяют строку, а работают с ее копией. Они возвращают измененную копию строки. Если вы хотите изменить саму строку, это делается так:

```
h = h.ToUpper();
```

Методы `ToUpper()` и `ToLower()` объявлены так:

```

public string ToUpper()
public string ToLower()

```

Сравнение строк

Для сравнения строк используется метод `Compare()`. Существует много разных форм вызова этого метода. Рассмотрим формы, использующиеся для сравнения целых строк:

```

public static int Compare(string strA, string strB)
public static int Compare(string strA, string strB, bool ignoreCase)
public static int Compare(string strA, string strB, StringComparison
comparisonType)
public static int Compare(string strA, string strB, bool ignore-

```

```
Case, CultureInfo culture)
```

В данном случае метод сравнивает строку `strA` со строкой `strB`. Возвращает положительное значение, если строка `strA` больше строки `strB`; отрицательное значение, если строка `strA` меньше строки `strB`; и нуль, если строки `strA` и `strB` равны. Сравнение выполняется с учетом регистра и культурной среды.

Если параметр `ignoreCase` равен `true`, то при сравнении не учитывается регистр символов. В противном случае (`false`) эти различия учитываются.

Параметр `comparisonType` определяет конкретный способ сравнения строк. Класс `CultureInfo` определен в пространстве имен `System.Globalization`. Используется для лучшей локализации программы.

С помощью метода `Compare()` можно сравнивать фрагменты строк, а не целые строки:

```
public static int Compare(string strA, int indexA, string strB, int
indexB, int length)
public static int Compare(string strA, int indexA, string strB, int
indexB, int length, bool ignoreCase)
public static int Compare(string strA, int indexA, string strB, int
indexB, int length, StringComparison comparisonType)
public static int Compare(string strA, int indexA, string strB, int
indexB, int length, bool ignoreCase, CultureInfo culture)
public static int CompareOrdinal(string strA, string strB)
public static int CompareOrdinal(string strA, int indexA, string
strB, int indexB, int count)
```

Данная форма сравнивает фрагменты строк `strA` и `strB`. Сравнение начинается со строковых элементов `strA[indexA]` и `strB[indexB]` и включает количество символов, определяемых параметром `length`.

Метод возвращает положительное значение, если часть строки `strA` больше части строки `strB`; отрицательное значение, если часть строки `strA` меньше части строки `strB`; и нуль, если сравниваемые части строк `strA` и `strB` равны. Сравнение выполняется с учетом регистра и культурной среды. Аналогично, можно указать параметры `ignoreCase` и `CultureInfo`, как в предыдущем случае.

Кроме метода `Compare()`, есть и метод `CompareOrdinal()`, который работает так же, как и `Compare()`, но не имеет параметра `CultureInfo()` и локальных установок. Метод объявлен так:

```
public static int CompareOrdinal(string strA, string strB)
public static int CompareOrdinal(string strA, int indexA,
string strB, int indexB, int count)
```

Метод Equals() также используется для сравнения строк. Различные формы этого метода и их описания представлены в таблице 3.4.

Таблица 3.4. Формы метода Equals()

Синтаксис	Описание
<pre>public override bool Equals(object obj)</pre>	<p>Метод возвращает true, если вызывающая строка содержит ту же последовательность символов, что и строковое представление объекта obj. Выполняется порядковое сравнение с учетом регистра, но параметры локализации не учитываются</p>
<pre>public bool Equals(string value) public bool Equals(string value, StringComparison comparisonType)</pre>	<p>Возвращает true, если вызывающая строка содержит ту же последовательность символов, что и строка value. Выполняется порядковое сравнение с учетом регистра, но параметр локализации не используется. Параметр comparisonType определяет конкретный способ сравнения строк</p>
<pre>public static bool Equals(string a, string b) public static bool Equals(string a, string b, StringComparison comparisonType)</pre>	<p>Возвращает логическое значение true, если строка a содержит ту же последовательность символов, что и строка b. Выполняется порядковое сравнение с учетом регистра, но параметры локализации не используются. Параметр comparisonType определяет конкретный способ сравнения строк</p>

Поиск в строке

Для поиска в строке в C# есть множество методов. Начнем с метода Contains(), который описан так:

```
public bool Contains(string value)
```

Это самый простой метод, позволяющий определить, есть ли в строке определенная подстрока.

Метод `StartsWith()` позволяет определить, начинается ли вызывающая подстрока с подстроки **value** (если так, метод возвращает **value**, иначе метод возвращает **false**). Параметр `comparisonType` определяет конкретный способ выполнения поиска. Синтаксис следующий:

```
public bool StartsWith(string value)
public bool StartsWith(string value, StringComparison
comparisonType)
```

Аналогично, существует метод `EndsWith()`, который возвращает **true**, если вызывающая строка заканчивается подстрокой **value**:

```
public bool EndsWith(string value)
public bool EndsWith(string value, StringComparison
comparisonType)
```

Для поиска первого вхождения заданной подстроки или символа используется метод `IndexOf()`:

```
public int IndexOf(char value)
public int IndexOf(string value)
```

Если искомый символ или подстрока *не обнаружены*, то возвращается значение **-1**. В противном случае возвращает позицию, с которой начинается подстрока или позицию, где впервые встречается заданный символ.

Если нужно начать поиск с определенной позиции `startIndex`, то синтаксис вызова метода будет немного другим:

```
public int IndexOf(char value, int startIndex)
public int IndexOf(string value, int startIndex)
public int IndexOf(char value, int startIndex, int count)
public int IndexOf(string value, int startIndex, int count)
```

Поиск начинается с элемента, который указывается индексом `startIndex`, и охватывает число элементов, определяемых параметром `count` (если указан).

Последнее вхождение символа/подстроки можно найти методом `LastIndexOf()`. Параметры у этого метода такие же, как и у `IndexOf()`.

Усложним задачу. Представим, что нужно найти не просто первое вхождение какого-то символа, а первое вхождение одного из символов, например, есть строка **Hello** и нам нужно определить первое вхождение символов **l** и **o**. Можно два раза вызвать метод `IndexOf()` - для символа **l** и для символа **o**, од-

нако это неэффективно. Гораздо удобнее использовать метод `IndexOfAny()`, которому можно передать массив искомых символов:

```
public int IndexOfAny(char[] anyOf)
public int IndexOfAny(char[] anyOf, int startIndex)
public int IndexOfAny(char[] anyOf, int startIndex, int count)
```

Первый параметр - это массив искомых символов, второй - начальная позиция поиска, третий - счетчик символов. Метод возвращает индекс первого вхождения любого символа из массива `anyOf`, обнаруженного в вызывающей строке. Метод возвращает значение `-1`, если не обнаружено совпадение ни с одним из символов из массива `anyOf`.

Вернемся к нашему примеру. У нас есть строка `Hello` и мы хотим найти позицию символа `l` или символа `o` - какой встретится раньше. Обратите внимание, метод возвращает одну позицию, а не массив позиций - каждого из приведенных символов. Пример кода:

```
string s = "Hello";
char[] Ch = {'l', 'o'};
if (s.IndexOfAny(Ch) != -1)
    Console.WriteLine("Один из символов был найден в позиции {0}",
        s.IndexOfAny(Ch));
```

Аналогично методу `IndexOfAny()`, существует метод `LastIndexOfAny()`, который осуществляет поиск с конца.

Конкатенация строк

Переменные типа `string` можно соединить вместе, то есть выполнить конкатенацию. Для этого используется оператор `+`. Компилятор C# преобразует оператор `+` в вызов метода `String.Concat()`, поэтому вы можете использовать `+` или метод `Concat()` - как вам больше нравится:

```
string s1 = "s1";
string s2 = "s2";
string s3 = s1 + s2;
```

Метод `Concat()` объявлен так:

```
public static string Concat(string str0, string str1);
public static string Concat(params string[] values);
```

Разделение и соединение строк

Представим, что есть строка, содержащая какой-то разделитель (сепаратор). С помощью метода `Split()` можно разделить эту строку на подстроки.

Метод, возвращающий массив **string** с присутствующими в данном экземпляре подстроками внутри, которые отделяются друг от друга элементами из указанного массива **separator**:

```
public string[] Split(params char[] separator)
public string[] Split(params char[] separator, int count)
```

Если массив **separator** пуст или ссылается на пустую строку, то в качестве разделителя подстрок используется *пробел*. Во второй форме данного метода возвращается количество подстрок, заданное параметром **count**.

Существуют и другие формы вызова метода **Split()**:

```
public string[] Split(params char[] separator,
StringSplitOptions options)
public string[] Split(string[] separator, StringSplitOptions
options)
public string[] Split(params char[] separator, int count,
StringSplitOptions options)
public string[] Split(string[] separator, int count,
StringSplitOptions options)
```

Разница, как видите, заключается в параметре **options**. В перечислении типа **StringSplitOptions** определяются только два значения: **None** и **RemoveEmptyEntries**. Если параметр **options** принимает значение **None**, то пустые строки включаются в конечный результат разделения исходной строки. А если параметр **options** принимает значение **RemoveEmptyEntries**, то пустые строки исключаются из конечного результата разделения исходной строки.

С помощью метода **Join()** можно решить обратную задачу, а именно построить строку по массиву строк, разделив каждый элемент этого массива каким-то разделителем. Синтаксис такой:

```
public static string Join(string separator, string[] value)
public static string Join(string separator, string[] value,
int startIndex, int count)
```

В первой форме метода **Join()** возвращается строка, состоящая из соединяемых подстрок из массива **value**. Во второй форме также возвращается строка, состоящая из элементов массива **value**, но они соединяются в определенном количестве **count**, начиная с элемента массива **value[startIndex]**. В обеих формах каждая последующая строка отделяется от предыдущей разделителем, заданным **separator**.

Заполнение и обрезка строк

В PHP есть удобные функции заполнения и обрезки строк. Подобные функции есть и в C#. Например, в C# есть метод `Trim()`, позволяющий удалять все вхождения определенного набора символов с начала и конца текущей строки:

```
public string Trim()
public string Trim(params char[] trimChars)
```

Первая форма удаляет из вызывающей строки начальные и конечные пробелы. Во второй форме удаляются начальные и конечные вхождения в вызывающей строке символов из массива `trimChars`. В обеих формах возвращается получающаяся в результате строка.

Методы `PadLeft()` и `PadRight()` позволяют дополнить строку символами слева и справа соответственно. Синтаксис обоих методов одинаковый, отличаются только названия:

```
public string PadLeft(int totalWidth)
public string PadLeft(int totalWidth, char paddingChar)
```

Первая форма дополняет строку пробелами так, чтобы ее общая длина стала равной значению `totalWidth`. Вторая форма позволяет указать символ заполнения. В обеих формах возвращается получающаяся в итоге строка. Если значение параметра `totalWidth` меньше длины вызывающей строки, то возвращается копия неизменной строки.

Вставка, удаление и замена строк

Метод `Insert()` позволяет вставить строку `value` в вызывающую строку по индексу `startIndex`:

```
public string Insert(int startIndex, string value)
```

В результате будет возвращена результирующая строка.

Для удаления части строки используется метод `Remove()`:

```
public string Remove(int startIndex)
public string Remove(int startIndex, int count)
```

Первая форма метода `Remove()` позволяет удалить часть строки, начинающуюся с индекса `startIndex`, и до конца строки. Вторая форма удаляет `count` символов, начиная с позиции `startIndex`.

Для выполнения замены в строке используется метод `Replace()`, синтаксис которого приведен ниже:

```
public string Replace(char oldChar, char newChar)
public string Replace(string oldValue, string newValue)
```

Первая форма заменяет все вхождения символа `oldChar` на символ `newChar`.
Вторая форма заменяет все вхождения подстроки `oldValue` на `newValue`.

Получение подстроки

Получить подстроку можно методом `Substring()`, обладающим следующим синтаксисом:

```
public string Substring(int startIndex)
public string Substring(int startIndex, int length)
```

Первая форма возвращает подстроку, начинающуюся с позиции `startIndex` и до конца строки, а вторая - возвращает подстроку длиной `length` символов, начиная с позиции `startIndex`.

Управляющие последовательности символов

В языке Си строковые литералы могут содержать различные управляющие последовательности символов (escape characters). Язык C# - не исключение. Управляющие последовательности позволяют уточнить то, как символьные данные будут выводиться в выходном потоке.

Управляющая последовательность начинается с символа обратного слеша, после которого следует управляющий знак. В таблице 3.5 приведены наиболее популярные управляющие последовательности.

Таблица 3.5. Управляющие последовательности

Последовательность	Описание
\'	Используется для вставки символа одинарной кавычки
\"	Позволяет вставить символ двойной кавычки
\\	Вставляет в строковой литерал символ обратной черты
\a	Заставляет систему воспроизводить звуковой сигнал (beep)
\n	Символ новой строки
\r	Возврат каретки
\t	Вставляет символ табуляции

Пример:

```
// Выведем две пустых строки и звуковой сигнал после слова Hello
Console.WriteLine("Hello\n\n\a");
```

Строки и равенство

В языке C# операция равенства предусматривает посимвольную проверку строк с учетом регистра. Другими словами, строки `Hello` и `hello` не равны. Проверку на равенство двух строк можно произвести или с помощью метода `Equals()`, или с помощью оператора `==`:

```
string s1 = "s1";
string s2 = "s2";
Console.WriteLine("s1 == s2: {0}", s1 == s2);
```

В результате будет выведена строка:

```
s1 == s2: false
```

Метод `Equals()` был описан ранее.

Тип `System.Text.StringBuilder`

Тип `string` может оказаться неэффективным в ряде случаев. Именно поэтому в .NET предоставляется еще одно пространство имен - `System.Text`. Внутри этого пространства имен находится класс по имени `StringBuilder`. В нем содержатся методы, позволяющие заменять и форматировать сегменты.

Для использования класса `StringBuilder` первым делом нужно подключить пространство имен `System.Text`:

```
using System.Text;
```

При вызове членов `StringBuilder` производится непосредственное изменение внутренних символьных данных объекта, а не получение копии этих данных в измененном формате. При создании экземпляра `StringBuilder` начальные значения для объекта можно задавать с помощью не одного, а нескольких конструкторов. Рассмотрим пример использования `StringBuilder`:

```
StringBuilder sb = new StringBuilder("Операционные системы:");
sb.Append("\n");
sb.AppendLine("Windows");
sb.AppendLine("Linux");
sb.AppendLine("Mac OS x");
Console.WriteLine(sb.ToString());
Console.WriteLine("В sb {0} символов", sb.Length);
Console.ReadLine();
```

Сначала мы создаем объект `sb` и добавляем в него строку "Операционные системы:". Затем методом `Append()` мы добавляем символ перевода строки. Можно было бы достичь того же эффекта, если бы мы создали объект так:

```
StringBuilder sb = new StringBuilder("Операционные  
системы:\n")
```

После этого мы добавляем строки методом `AppendLine()`. Данный метод автоматически добавляет символ `\n`. Для вывода общей строки мы используем метод `ToString()`, а свойство `Length` содержит количество символов в `sb`.



Рис. 3.6. Пример использования `StringBuilder`

3.3.6. Области видимости переменных

Область видимости переменной (ее еще называют контекстом переменной) - это фрагмент кода, в пределах которого будет доступна данная переменная.

Область видимости в С# определяется следующими правилами:

Поле или переменная-член класса находится в области видимости до тех пор, пока в этой области находится содержащий это поле класс.

Локальная переменная находится в области видимости до тех пор, пока закрывающая фигурная скобка не укажет конец блока операторов или метода, в котором она объявлена.

Локальная переменная, объявленная в операторах цикла `for`, `while` или подобных им, видима в пределах тела цикла

Рассмотрим несколько примеров:

```
for (int i = 0; i < 5; i++)
{
    Console.Write(" {0}", i);
}
// здесь заканчивается область видимости i
int k = 5 * i; // оператор не будет выполнен
```

После закрывающей скобки цикла **for** переменная **i** больше не будет "видна", следовательно, последний оператор не будет выполнен.

Еще один пример:

```
public static int Main()
{
    int m = 10;
    for (int i = 0; i < 10; i++)
    {
        int m = 20;           // ошибка
        Console.WriteLine(m + i);
    }
    return 0;
}
```

Переменная **m** будет доступна в пределах всего метода **Main()**. Вы не можете повторно объявить переменную **m** в теле цикла **for**. Это если вкратце. Если развернуто, то переменная **m**, определенная перед началом цикла **for**, будет находиться в области видимости до тех пор, пока не завершится метод **Main()**. Вторая переменная **m** якобы объявлена в контексте цикла, но он является вложенным в контекст метода **Main()**, поэтому компилятор не может различить эти две переменные и не допустит объявления второй переменной с именем **m**.

В некоторых случаях два идентификатора с одинаковыми именами и одинаковой областью видимости можно различить, при этом компилятор допустит объявление второй переменной. Причина в том, что язык C# делает различие между переменными, объявленными на уровне поля, и переменными, объявленными в методах (локальными переменными).

В качестве примера рассмотрим следующий фрагмент кода:

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static string uname = "den";

        public static void Main()
        {
            string uname = "john";
            Console.WriteLine(uname);
            return;
        }
    }
}
```

Обратите внимание, что в данном коде есть поле (член класса Program) и локальная переменная с одинаковыми именами - `ipname` (объявлена в методе `Main()`). Когда вы запустите этот код, вы увидите строку "john". Приоритет у локальных переменных выше, чем у членов класса, поэтому вы увидите именно эту строку, а не строку "den". Об этом следует помнить.

3.3.7. Константы

Константы - это переменные, значение которых нельзя изменить во время выполнения программы. Константа объявляется с помощью служебного слова `const`, после которого следует тип константы:

```
const int j = 100;
```

Особенности констант:

- Константы должны инициализироваться при объявлении, присвоенные им значения никогда не могут быть изменены.
- Значение константы вычисляется во время компиляции. Поэтому инициализировать константу значением, взятым из другой переменной, нельзя.
- Константы всегда являются неявно статическими. Но при этом не нужно указывать модификатор `static`.

3.4. Операторы

3.4.1. Арифметические операторы

Арифметические операторы представлены в таблице 3.6. Операторы `+`, `-`, `*` и `/` работают так, как предполагает их обозначение. Их можно применять к любому встроенному числовому типу данных. Думаю, в особых комментариях данные операторы не нуждаются.

Таблица 3.6. Арифметические операторы в C#

Оператор	Действие
<code>+</code>	Сложение
<code>-</code>	Вычитание, унарный минус
<code>*</code>	Умножение
<code>/</code>	Деление
<code>%</code>	Деление по модулю
<code>--</code>	Декремент
<code>++</code>	Инкремент

Хотя эти операторы всем знакомы, нужно рассмотреть оператор `/`, а также операторы инкремента и декремента.

Когда `/` применяется к целому числу, то любой остаток от деления отбрасывается. Остаток от этого деления можно получить с помощью оператора деления по модулю (`%`), который иначе называется оператором вычисления остатка. В C# оператор `%` можно применять как к целочисленным типам данных, так и к типам с плавающей точкой. В этом отношении C# отличается от языков Си и C++, где этот оператор (деление по модулю) разрешается только для целочисленных типов данных.

Особенности есть и у операторов инкремента и декремента. Оператор инкремента (`++`) увеличивает свой операнд на 1, а оператор декремента (`--`) уменьшает операнд на 1. Фактически, оператор:

```
x++;
```

аналогичен следующему:

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

Вот только нужно иметь в виду, что при использовании операторов инкремента и декремента значение переменной `x` вычисляется только один, а не два раза. Это сделано, чтобы повысить эффективность выполнения программы.

Операторы инкремента и декремента можно указывать до операнда (в префиксной форме) или же после операнда (в постфиксной форме). Операция инкремента/декремента в префиксной форме происходит раньше, нежели в постфиксной форме. Пример:

```
int a = 0;
int x = 1;
a = x++;           // a = 1, x = 2
a = 0; x = 1;      // исходные значения
a = ++x;           // a = 2; x = 2;
```

Хочется немного отклониться от темы обсуждения и сделать небольшой перерыв. Не всегда есть возможность получить доступ к среде Visual Studio. Конкретный тому пример происходит сейчас - при написании этих строк. Не всегда есть возможность написать главу за один день и за одним компьютером. Вот и сейчас этот раздел пишется на компьютере, на котором не установлена Visual Studio. Как делать скриншот результатов выполнения? Как проверить свой собственный код на отсутствие ошибок (у человеческого мозга есть один недостаток - часто он не видит собственных ошибок)? Специально для этого есть Online-компиляторы. Их не нужно устанавливать на свой компьютер. Просто откройте браузер, введите следующий

URL, и у вас будет доступ к полноценному компилятору того или иного языка программирования (в данном случае - C#):

http://www.tutorialspoint.com/compile_csharp_online.php

Введите код, который вы хотите проверить, и нажмите кнопку **Compile**. Если ее не нажать, то сайт сообщит, что не может открыть ехе-файл - понятное дело, он еще не откомпилирован. После компиляции нажмите кнопку **Execute** и получите результат выполнения (рис. 3.7).



Рис. 3.7. Использование online-компилятора

3.4.2. Операторы сравнения и логические операторы

Данные типы операторов обычно используются в условном операторе `if/else` (см. раздел 3.8), поэтому пока только перечислим операторы, а примеры кода вы найдете далее.

Результатом операторов (как сравнения, так и логических) является значение типа `bool` - или `true`, или `false`.

Таблица 3.7. Операторы сравнения

Оператор	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Таблица 3.8. Логические операторы

Оператор	Значение
&	И
	ИЛИ
^	Исключающее ИЛИ
&&	Укороченное И
	Укороченное ИЛИ
!	НЕ

В языке C# существуют так называемые укороченные варианты логических операторов И и ИЛИ. Оба эти оператора предназначены для получения более эффективного кода.

Рассмотрим небольшой пример работы укороченных операций. Если первый операнд операции И (&&) имеет ложное значение (false), то ее результат будет иметь ложное (false) значение, независимо от значения второго операнда. Если же первый операнд логической операции ИЛИ (||) имеет истинное значение (true), то ее результат будет иметь истинное значение независимо от значения второго операнда. Поскольку значение второго операнда вычислять не нужно, экономится время и повышается эффективность кода.

Укороченные операции || и && отличаются от обычных тем, что второй операнд вычисляется только по мере необходимости. Обычно укороченные версии более эффективны. Но зачем тогда нужны обычные операции? Иногда нужно вычислить значение обоих операторов из-за неприятных побочных эффектов, которые могут возникнуть.

Пример:

```
bool t = true;
int i = 0;
// При использовании обычного оператора в данной конструкции
// i будет увеличиваться
```

```

if (t | (++i < 5))
Console.WriteLine("i равно {0}", i);    // i = 1

i = 0;
// При использовании укороченного оператора
// значение i останется прежним
if (t || (++i < 5))
Console.WriteLine("i равно {0}", i);    // i = 0

```

3.4.3. Операторы присваивания

Оператор присваивания обозначается как знак равенства (=) и обычно имеет форму:

имя_переменной = выражение

Пример:

```

x = 5;
b = true;

```

Кроме этой простой формы в C# поддерживаются так называемые составные операторы присваивания (см. табл. 3.9). Как правило, такие операторы довольно удобны и они делают код компактнее, хотя и менее понятны начинающим программистам.

Таблица 3.9. Составные операторы присваивания

Оператор	Пример кода	Полная форма оператора
+=	x += 1;	x = x + 1;
-=	x -= 1;	x = x - 1;
*=	x *= 1;	x = x * 1;
/=	x /= 1;	x = x / 1;
%=	x %= 1;	x = x % 1;
=	x = 1;	x = x 1;
^=	x ^= 1;	x = x ^ 1;

3.4.4. Поразрядные операторы

В C# поддерживаются те же поразрядные операторы, что и в Си/C++. Работают данные операторы так же, как и в языках Си/C++.

Таблица 3.10. Поразрядные операторы

Оператор	Значение
&	Поразрядное И
	Поразрядное ИЛИ

^	Поразрядное исключающее ИЛИ
<<	Сдвиг влево
>>	Сдвиг вправо
~	Дополнение до 1 (унарный оператор НЕ)

3.5. Преобразование типов данных

В языке C# допускается преобразование типов данных с их автоматическим сужением и расширением. Рассмотрим следующий пример (листинг 3.2).

Листинг 3.2. Пример расширения типов данных

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            short a = 1000, b = 2000;
            int c = Add(a, b);
            Console.WriteLine("c = {0}", c);
            Console.ReadLine();
        }

        static int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

Теперь рассмотрим, что произошло. В метод `Add()` мы передаем два значения типа **short**, хотя метод подразумевает прием параметров типа **int**. На самом деле ничего страшного не происходит - тип **short** поглощается типом **int** (диапазон **int** значительно шире, чем **short**). Ошибки компиляции не будет - просто компилятор расширит значения типа **short** до значений типа **int**. Расширение всегда происходит без потери данных, поэтому вы увидите правильный результат:

```
c = 3000
```

Теперь рассмотрим листинг 3.3, попытка скомпилировать который приведет к ошибке компиляции. Измененные строки выделены жирным.

Листинг 3.3. Ошибка при преобразовании типов

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            short a = 20000, b = 20000;
            short c = Add(a, b);
            Console.WriteLine("c = {0}", c);
            Console.ReadLine();
        }

        static int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

Во-первых, мы увеличили значения переменных **a** и **b**. Тем не менее оба значения все еще вписываются (даже с запасом) в диапазон типа **short**. Во-вторых, переменная **c** теперь типа **short**, а не **int**. Разберемся, что происходит. Переменные типа **short** передаются в качестве параметров методу **Add()**. Компилятор может выполнить расширение до типа **int**, как было отмечено ранее, но ему приходится возвращать ответ типа **int**, который нужно поместить в переменную типа **short**. Выполнить сужение типа без потери данных в этом случае невозможно. Ведь результат будет 40000, а максимальное значение для **short** - 32767. Поэтому вы увидите ошибку компилятора (рис. 3.8):

```
Cannot implicitly convert type 'int' to 'short'. An explicit
conversion exists (are you missing a cast?)
```

Чтобы заставить компилятор выполнить сужение типа с потерей данных, нужно использовать *операцию явного приведения типов*. В C# данная операция обозначается с помощью скобок **()**, внутри которых приводится имя типа, к которому нужно привести значение:

```
short c = (short)Add(a, b);
```

Однако в этом случае, хотя программа и будет выполнена, результат будет неправильным (рис. 3.9).

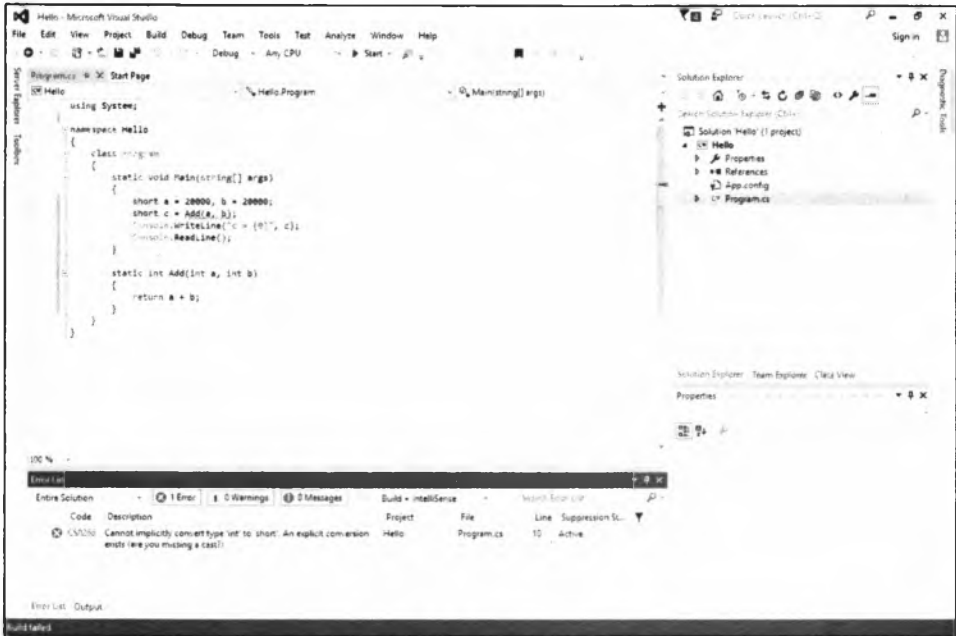


Рис. 3.8. Ошибка компиляции



Рис. 3.9. Вместо ожидаемого значения 40000 мы получили совсем другой результат

Помните, что операцию явного приведения типов нужно использовать с осторожностью, поскольку она может привести к потере данных.

Понятное дело, во многих приложениях, особенно финансовых, такие потери данных просто недопустимы. К счастью, в C# предлагаются ключевые слова `checked` и `unchecked`, гарантирующие, что потеря данных окажется незамеченной. Ведь в предыдущем примере мы просто получили непра-

вильный результат ($20000 + 20000 = -25536$) и никаких предупреждений компилятора о том, что результат может быть неправильным.

Если оператор или блок операторов заключен в контекст `checked`, то компилятор сгенерирует дополнительные CIL-инструкции, обеспечивающие проверку на предмет условий переполнения, которые могут возникать в результате сложения, умножения, вычитания или деления двух числовых типов данных. В случае возникновения условия переполнения во время выполнения будет генерироваться исключение `System.OverflowException`.

Обработать исключение можно, как обычно, с помощью `try/catch`. Если вы не знакомы с обработкой исключений, то позже в этой книге мы о них поговорим, а сейчас просто рассмотрим следующий блок кода:

```
try
{
    short c = checked((short)Add(a, b));
    Console.WriteLine("c = {0}", c);
}
catch (OverflowException ex)
{
    Console.WriteLine(ex.Message);
}
```

В случае, если возникнет исключение, мы введем его сообщение на консоль (рис. 3.10).



Рис. 3.10. При выполнении арифметической операции возникло исключение

Из-за того, что действие флага `checked` распространяется на всю арифметическую логику, в C# предусмотрено ключевое слово `unchecked`, которое позволяет отключить выдачу связанного с переполнением исключения в отдельных случаях. Применяется это ключевое слово по-хожим на `checked`

образом, поскольку может быть указано как для одного оператора, так и для целого блока:

```
unchecked
{
    short c = (short)Add(a, b);
    Console.WriteLine("c = {0}", c);
}
```

В результате арифметическое переполнение будет явно проигнорировано. По крайней мере, программист будет об этом знать. Использовать ключевое слово `unchecked` можно только в тех случаях, где переполнение является допустимым. Надеюсь, вы знаете, что делаете, раз используете его.

3.6. Неявно типизированные локальные переменные

До этого момента мы везде явно указывали тип переменной - при ее объявлении. Явное указание типа переменной считается хорошим стилем, но в C# поддерживается неявная типизация. Создать неявно типизированные локальные переменные можно с помощью ключевого слова `var`, например:

```
var A = 0;
var s = "String";
```

Ключевое слово `var` можно использовать вместо указания конкретного типа данных. При этом компилятор автоматически определяет тип переменной по типу первого значения, которое присваивается при инициализации. В нашем примере для переменной `A` будет выбран тип `int` (`System.Int32`), а для `s` - `string` (`System.String`).

На самом деле слово `var` не является ключевым словом языка C#. С его помощью можно объявлять переменные, параметры и поля и не получать никаких ошибок на этапе компиляции. При использовании этой лексемы в качестве типа данных она воспринимается компилятором как ключевое слово.

Понятное дело, использование неявно типизированных переменных несколько ограничено. Неявная типизация применима только для локальных переменных в контексте какого-то метода или свойства. Вы не можете использовать ключевое слово `var` для определения возвращаемых значений, параметров или данных полей специального типа - это приведет к выдаче сообщений об ошибках на этапе компиляции.

Также переменным, объявленным с помощью **var**, сразу должно быть присвоено значение - при самом объявлении. При этом использовать **null** в качестве значения не допускается, поскольку компилятор не сможет определить тип переменной по ее значению:

```
var variable1;           // приведет к ошибке!
```

```
// тоже приведет к ошибке
```

```
var variable2;  
variable2 = 0;
```

```
// правильно  
var variable3 = 0;
```

Что касается значения **null**, то его можно присваивать, но уже после того, как переменной было присвоено первоначальное значение определенного типа:

```
var sb = new StringBuilder("Операционные системы:\n")  
sb = null;
```

Значение неявно типизированной локальной переменной может быть присвоено другим переменным, причем как неявно, так и явно типизированным:

```
// Ошибок нет  
var X = 0;  
var Y = X;  
string str = "Hello!";  
var world = str;
```

Стоит отметить, что особой пользы от неявно типизированных переменных нет. К тому же использование **var** может даже вызвать путаницу у всех, кто будет изучать написанный вами код, - ведь другой программист сразу не сможет определить тип данных и понять, для чего используется переменная. Поэтому рекомендуется всегда явно указывать тип данных.

3.7. Циклы

В любом языке программирования имеются итерационные конструкции, использующиеся для повторения блоков кода до тех пор, пока не будет выполнено какое-то условие завершения. В языке C# поддерживаются четыре таких конструкции: **for**, **foreach**, **while** и **do/while**.

Если вы ранее программировали на других языках, все эти конструкции (может быть, за исключением **foreach**, хотя смотря на каком языке вы программировали до этого) вам должны быть знакомы. Начнем мы с классики - цикла **for**.

3.7.1. Цикл **for**

Идеальное решение, если нужно выполнить какой-то блок кода фиксированное количество раз. Оператор **for** позволяет указать, сколько раз должен повторяться блок кода, и задать условие завершения цикла. Первый параметр оператора **for** задает оператор, который будет выполнен до первой инициализации цикла. Обычно здесь инициализируется переменная-счетчик. Второй параметр задает условие выхода из цикла, а третий - оператор, который будет выполнен после каждой итерации. Но, думаю, вы все это знаете.

Небольшой пример:

```
for(int i = 0; i < 9; i++)
{
    Console.Write("{0}", i);
}
```

Будет выведена строка 012345678.

Как и в других языках, в C# можно создавать сложные конечные условия, определять бесконечные циклы и использовать ключевые слова **goto**, **continue** и **break**. Поскольку я подразумеваю, что вы знакомы с циклом **for**, подробно мы его рассматривать не будем. Если это не так, подробности вы можете получить в любом справочнике по Си/C++/C#.

Скажу только пару слов о переменной-счетчике. Она (переменная **i**) доступна только в теле цикла **for**. За пределами цикла (после выполнения последней итерации) переменная **i** будет недоступна.

3.7.2. Цикл **foreach**

Цикл **foreach** удобно использовать при проходе по всем элементам массива или коллекции без проверки верхнего предела.

Рассмотрим пример прохода по массиву целых чисел с использованием цикла **foreach**:

```
int[] digits = { 1, 2, 3, 4 };
foreach (int i in digits)
    Console.WriteLine(i);
```

Сначала в **foreach** указывается переменная, которая должна быть такого же типа, что и элементы массива. В эту переменную будет получен элемент массива для его дальнейшей обработки (в нашем случае обработка заключается в выводе на консоль). После ключевого слова **in** задается название переменной массива.

Истинная мощь этого оператора раскрывается отнюдь не при работе с массивами, а при работе с интерфейсами, поэтому рассмотрение этого оператора откладывается до момента вашего знакомства с интерфейсами. Тогда рассмотрение **foreach** будет целесообразным в случае с C#.

3.7.3. Циклы **while** и **do/while**

В некоторой мере цикл **for** очень похож на цикл с предусловием (**while**), так как сначала проверяется условие, а потом уже выполняется тело цикла. Рассмотрим цикл с предусловием:

```
while ( логическое выражение )
    оператор;
```

Сначала цикл вычисляет значение логического выражения. Если оно истинно, происходит итерация цикла (выполняется тело цикла), иначе происходит выход из цикла и выполнение следующего за циклом оператора.

Вот пример вывода строки 12345678910:

```
int i=0;
while(i++ < 10) Console.Write("{0}", i);
```

Если переменную **i** увеличивать в теле цикла после вывода предыдущего значения **i**, мы получим строку 0123456789:

```
int i=0;
while(i < 10)
{
    Console.Write(i);
    i++;
}
```

В C# есть еще одна форма цикла - **do/while**. В отличие от цикла **while**, здесь сначала выполняются операторы (тело цикла), а затем уже проверяется условие. Если условие истинно, то начинается следующая итерация. Получается, что тело цикла будет выполнено как минимум один раз. Синтаксис цикла:

```
do
```

```
{
// тело цикла
}
while (условие);
```

Пример:

```
int i = 1;
do
    Console.Write(i);
while (i++ < 10);
```

В результате будет выведена та же строка 12345678910.

3.8. Конструкции принятия решений

Как и в любом другом языке, кроме итерационных конструкций есть конструкции принятия решений. В C# есть две таких конструкции - операторы `if/else` и `switch`.

Синтаксис оператора `if/else` такой же, как в языке Си/C++. Но в отличие от Си и C++, в C# этот оператор может работать только с булевыми выражениями, но не с произвольными значениями вроде -1 и 0. Учитывая этот факт, в операторе `if/else` можно использовать следующие операции сравнения:

- `==` - возвращает `true`, если выражения одинаковые, например, `if (page == 5)`
- `!=` - возвращает `true`, если выражения не равны: `if (page != 4)`
- `< (<=)` - возвращает `true`, если выражение слева меньше или равно, чем выражение справа: `if (price < 100)`
- `> (>=)` - возвращает `true`, если выражение слева больше или равно, чем выражение справа, `if (price > 200)`

В Си/C++ можно использовать следующий код:

```
int k = 100;
if (k)
{
    // do something
}
```

В C# такой код недопустим, поскольку `k` - это целое, а не булево значение.

В операторе `if` можно использовать сложные выражения, и он может содержать операторы `else`, что позволяет создать более сложные проверки. Синтаксис похож на языки Си/C++:

- `&&` - условная операция AND (И), возвращает true, если все выражения истинны.
- `||` - условная операция OR (ИЛИ), возвращает true, если хотя бы одно из выражений истинно.
- `!` - условная операция NOT (НЕ), возвращает true, если выражение ложно, и false - если истинно.

Пример:

```
if (page == 1)
{
    Console.WriteLine("Первая страница");
}
else
{
    Console.WriteLine("Страница: {0}", page);
}

if (page == 1 && price < 100)
    Console.WriteLine("Дешевые продукты на первой странице");
```

Рассмотрим следующий не очень хороший пример кода:

```
if (page == 1) Console.WriteLine("Первая страница");
if (page == 2) Console.WriteLine("Вторая страница");
if (page == 3) Console.WriteLine("Третья страница");
if (page == 4) Console.WriteLine("Четвертая страница");
if (page == 5) Console.WriteLine("Пятая страница");
else Console.WriteLine("Страница {0}", page);
```

Данный код можно переписать с использованием оператора switch:

```
switch (page)
{
    case 1: Console.WriteLine("Первая страница");
break;
    case 2: Console.WriteLine("Вторая страница");
break;
    case 3: Console.WriteLine("Третья страница");
break;
    case 4: Console.WriteLine("Четвертая страница");
break;
    case 5: Console.WriteLine("Пятая страница");
default: Console.WriteLine("Страница {0}", page);
break;
}
```

В языке C# каждый блок **case**, в котором содержатся выполняемые операторы (default в том числе), должен завершаться оператором **break** или **goto**, во избежание сквозного прохода.

У оператора **switch** в C# есть одна прекрасная особенность: помимо числовых данных он также позволяет производить вычисления и со строковыми данными. Рассмотрим пример:

```
string OS = "Linux";
switch (OS)
{
    case "Windows": Console.WriteLine("Хороший выбор!");
    break;
    case "Linux" : Console.WriteLine("OpenSource!");
    break;
    default : Console.WriteLine("Мы не знаем такую систему!");
    break;
}
```

3.9. Массивы

3.9.1. Одномерные массивы

Массив - это набор элементов данных одного типа, доступ к которым осуществляется по числовому индексу и к общему имени. В C# массивы могут быть как одномерными, так и многомерными. Массивы служат самым разным целям, поскольку они предоставляют удобные средства для объединения связанных вместе переменных.

Массивы в C# можно использовать почти так же, как и в других языках программирования, но у них есть одна особенность - они реализованы в виде объектов.

Чтобы воспользоваться массивом в программе, необходимо сначала объявить переменную, которая может обращаться к массиву, а после - создать экземпляр массива, используя оператор **new**.

Объявить массив можно так:

```
<тип>[] <имя>;
```

Пример:

```
int[] digits;    // массив целых чисел
string[] strs;   // массив строк
bool[] bools;    // массив булевых значений
```

Обратите внимание: следующий код неправильный!

```
int ints[];    // ошибка компиляции
```

После объявления массива необходимо установить его размер с помощью ключевого слова **new**, точно так же, как в Java. Пример:

```
ints = new int[5];
```

Установить размер можно и при объявлении массива:

```
int[] ints = new int[5];
```

Инициализировать массив можно двумя способами: можно заполнить его поэлементно, а можно использовать фигурные скобки:

```
ints[0] = 12;  
ints[1] = 15;  
ints[2] = 22;  
ints[3] = 5;  
ints[4] = 122;
```

При заполнении массива вручную помните, что нумерация начинается с 0. Но гораздо удобнее использовать фигурные скобки:

```
int[] ints = new int[] {100,200,300,400,500};
```

```
// Синтаксис инициализации массива без использования  
// ключевого слова new  
string[] user = {"1001", "den", "1234"};
```

```
// Используем ключевое слово new и желаемый размер массива  
char[] symbol = new char[4] { 'A','B','C','D' };
```

Массивы могут быть неявно типизированными (тип элементов массива - **var**):

```
var arr1 = new[] { 1, 2, 3 };  
Console.WriteLine("Type of array - {0}",arr1.GetType());
```

Ранее было сказано, что массив - это набор элементов одного типа. Но в C# есть одна уловка, позволяющая помещать в массив элементы разных типов. В C# поддерживаются массивы объектов. Вы можете объявить массив типа **object[]** и поместить в него элементы самых разных типов. Вот как это работает:

```
object[] arrOfObjects = { true, 10, "Hello", 1.7 };
```

Формально определение массива соблюдается - все элементы типа object. Но поскольку все элементарные типы данных в C# представлены в виде объектов, то ими можно заполнить массив.

С каждым массивом в C# связано свойство Length, содержащее число элементов, из которых может состоять массив. Работает это так:

```
int[] ints = { 1, 2, 3, 4, 5 };

for (int i = 0; i < ints.Length; i++)
    Console.WriteLine(ints[i]);
```

3.9.2. Двумерные массивы

Многомерный массив содержит два или больше измерений, причем доступ к каждому элементу такого массива осуществляется с помощью определенной комбинации двух или более индексов. Многомерный массив индексируется двумя и более целыми числами. Наиболее часто используются двумерные массивы. Это одновременно самый простой и самый популярный тип многомерного массива. Местоположение любого элемента в двумерном массиве обозначается двумя индексами. Такой массив можно представить в виде таблицы, на строки которой указывает один индекс, а на столбцы — другой.

```
// Объявляем двумерный массив 4x5
int[,] RandomArr = new int[4, 5];

// Инициализируем генератор случайных чисел
Random ran = new Random();

// Инициализируем двумерный массив случайными числами
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 5; j++)
    {
        RandomArr[i, j] = ran.Next(1, 100); // случайное число от 1 до 100
        Console.Write("{0}\t", RandomArr[i, j]);
    }
    Console.WriteLine();
}
```

Если вам приходилось раньше программировать на Си, C++ или Java, то будьте осторожны при работе с многомерными массивами в C#. В этих языках программирования размеры массива и индексы указываются в отдельных квадратных скобках, тогда как в C# они разделяются запятой.

3.9.3. Ступенчатые массивы

Как было отмечено ранее, двумерный массив представляет собой таблицу, в которой длина каждой строки неизменна. Но в C# можно создавать так называемые ступенчатые массивы. Ступенчатый массив - это массив массивов, в котором длина каждого массива может быть разной.

Ступенчатые массивы объявляются с помощью ряда квадратных скобок, в которых указывается их размерность. Например, для объявления двумерного ступенчатого массива служит следующая общая форма:

```
тип [][] имя массива = new тип[размер] [];
```

Здесь размер обозначает число строк в массиве, а память для самих строк распределяется индивидуально, и поэтому длина строк может быть разной. Пример работы со ступенчатым массивом приведен в листинге 3.4.

Листинг 3.4. Работа со ступенчатым массивом

```
int i = 0;
// Объявляем ступенчатый массив
// В нем будет три массива длиной, соответственно,
// 3, 5 и 4 элемента
int[][] myArr = new int[3][];
myArr[0] = new int[3];
myArr[1] = new int[5];
myArr[2] = new int[4];

// Инициализируем ступенчатый массив
for (; i < 3; i++)
{
    myArr[0][i] = i;
    Console.Write("{0}\t", myArr[0][i]);
}

Console.WriteLine();
for (i = 0; i < 5; i++)
{
    myArr[1][i] = i;
    Console.Write("{0}\t", myArr[1][i]);
}

Console.WriteLine();
for (i = 0; i < 4; i++)
{
    myArr[2][i] = i;
    Console.Write("{0}\t", myArr[2][i]);
}
```

3.9.4. Класс Array. Сортировка массивов

Для создания массива можно использовать еще и класс `Array`. Класс `Array` является абстрактным, поэтому создать массив с использованием какого-либо конструктора нельзя. Но вместо применения синтаксиса C# для создания экземпляров массивов также возможно создавать их с помощью статического метода `CreateInstance()`. Это исключительно удобно; когда заранее неизвестен тип элементов массива, поскольку тип можно передать методу `CreateInstance()` в параметре как объект `Type`:

```
// Создаем массив типа string, длиной 3
Array strs = Array.CreateInstance(typeof(string), 3);

// Инициализируем первые два поля массива
strs.SetValue("Brand", 0);
strs.SetValue("Model", 1);

// Считываем данные из массива
string s = (string)strs.GetValue(1);
```

Гораздо удобнее использовать предлагаемый языком C# синтаксис, чем использовать класс `Array`, но поскольку мы изучаем C#, мы не могли его не рассмотреть. Однако существуют ситуации, когда использование `Array` является оправданным. Примеры таких ситуаций - копирование и сортировка массивов.

Массивы - это ссылочные типы, поэтому присваивание переменной типа массива другой переменной создает две переменных, ссылающихся на один и тот же массив. Для копирования массивов предусмотрена реализация массивами интерфейса `ICloneable`. Собственно, само клонирование выполняется методом `Clone()`, который определен в этом интерфейсе. Метод `Clone()` создает неглубокую копию массива. Если элементы массива относятся к типу значений, то все они копируются, если массив содержит элементы ссылочных типов, то сами эти элементы не копируются, а копируются лишь ссылки на них.

В классе `Array` также есть метод копирования - `Copy()`. Он тоже создает неглубокую копию массива. Но между `Clone()` и `Copy()` есть одно важное отличие: `Clone()` создает новый массив, а `Copy()` требует наличия существующего массива той же размерности с достаточным количеством элементов.

Пример вызовов `Clone()` и `Copy()`:

```
string[] arr1 = (string[])myArr.Clone();
Array.Copy(myArr, arr2, myArr.Length);
```

В первом примере мы клонируем массив `MyArr` в `arr1`, во втором - массив `myArr` копируется в массив `arr2`.

В класс `Array` встроен алгоритм быстрой сортировки (Quicksort) элементов массива. Простые типы вроде `System.String` и `System.Int32` реализуют интерфейс `Comparable`, поэтому вы можете сортировать массивы, содержащие элементы этих типов. Если вам нужно отсортировать элементы других типов, то они должны реализовать интерфейс `Comparable`.

С помощью разных вариантов метода `Sort()` можно отсортировать массив полностью или в заданных пределах либо отсортировать два массива, содержащих соответствующие пары “ключ-значение”. Сортировка массива необходима, чтобы можно было осуществить эффективный поиск, используя разные варианты метода `BinarySearch()`.

Пример сортировки массива приведен в листинге 3.5.

Листинг 3.5. Сортировка массива

```
int[] myArr = { -5, 7, -13, 121, 45, -1, 0, 77 };
```

```
Console.WriteLine("До сортировки: ");
```

```
foreach (int i in myArr)
```

```
Console.Write("\t{0}", i);
```

```
Array.Sort(myArr);
```

```
Console.WriteLine("\nПосле сортировки:");
```

```
foreach (int i in myArr)
```

```
Console.Write("\t{0}", i);
```

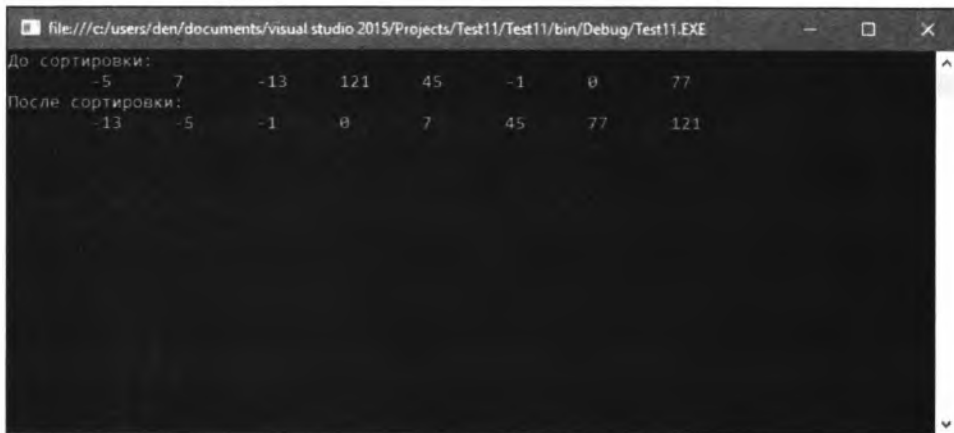


Рис. 3.11. Сортировка массива

3.9.5. Массив как параметр

Массив может выступать как параметр метода. Также метод может возвращать целый массив. Рассмотрим небольшой пример описания метода, принимающего массив в качестве параметра и возвращающего также массив:

```
static Array arrChange(Array Arr)
{
    // делаем что-то с массивом Arr
    return Arr;
}
```

3.10. Кортежи

В отличие от массивов (которые содержат объекты одного типа), кортежи (tuple) могут содержать объекты самых разных типов. Кортежи часто используются в языке F#, а с появлением .NET 4 кортежи доступны в .NET Framework для всех языков .NET.

В .NET 4 определены восемь обобщенных классов Tuple и один статический класс Tuple, который служит "фабрикой" кортежей.

Существуют разные обобщенные классы Tuple для поддержки различного количества элементов, например, Tuple<T1> содержит один элемент, Tuple<T1, T2> - два элемента и т.д. Элементы кортежа доступны через свойства Item1, Item2. Если имеется более восьми элементов, которые нужно включить в кортеж, можно использовать определение класса Tuple с восемью параметрами. Последний параметр называется TRest, в котором должен передаваться сам кортеж. Поэтому есть возможность создавать кортежи с любым количеством параметров.

Следующий метод возвращает кортеж из четырех элементов:

```
static Tuple<int, float, string, char> tup(int z, string name)
{
    int x = 4 * z;
    float y = (float) (Math.Sqrt(z));
    string s = "Привет, " + name;
    char ch = (char) (name[0]);

    return Tuple.Create<int, float, string, char>(x, y, s, ch);
}
```

Работать с этим методом можно так:

```
var t = tup(5, "Мак");
Console.WriteLine("{0} {1} {2} {3}", t.Item3, t.Item1,
t.Item2, t.Item4);
```

Далее мы рассмотрим несколько практических примеров - задачи, которые могут возникнуть у вас на практике.

3.11. Как подсчитать количество слов в тексте

Представим, что есть какой-то текст и нам нужно подсчитать количество символов. Код программы будет очень прост:

```
Console.WriteLine("Введите текст:");
string[] tArr;
string text = Console.ReadLine();
tArr = text.Split(' ');
Console.WriteLine("Количество слов:");
Console.WriteLine(tArr.Length);
Console.ReadLine();
```

Все очень просто. Мы создаем массив строк `textMass` и простую строковую переменную `text`. В переменную `text` считывается введенный пользователем текст, а в массив `tArr` добавляются элементы из строки `text`, разделенные пробелом при помощи метода `Split`. Каждый элемент данного массива – это как раз одно слово, заключенное в тексте между пробелов. Все, что осталось, - это вывести количество элементов массива.

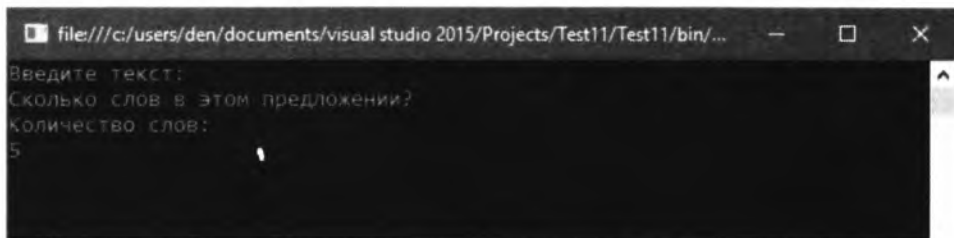


Рис. 3.12. Подсчет слов в тексте

Полный код этого приложения приведен в листинге 3.6.

Листинг 3.6. Подсчет количества слов в тексте

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

namespace Test11
```

```

{
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.InputEncoding = Encoding.GetEncoding(866);

            Console.WriteLine("Введите текст:");
            string[] tArr;
            string text = Console.ReadLine();
            tArr = text.Split(' ');
            Console.WriteLine("Количество слов:");
            Console.WriteLine(tArr.Length);
            Console.ReadLine();

            Console.ReadLine();
        }
    }
}

```

3.12. Вычисляем значение функции

Теперь задача такая: нам нужно вычислить сумму значений, возвращаемых функцией $f()$ от 1 до x . Функция будет такой:

$$3x^3 - 2x^2$$

Как мы знаем, в Си для возведения в степень используется функция `pow()`. Конечно, в нашем простом примере мы могли бы использовать вот такой оператор:

$$3 * x * x * x - 2 * x * x;$$

Но, согласитесь, это не совсем правильно. Думаю, читая эти строки, у вас созревает лишь один вопрос: а зачем приводить столь простой пример? Но этот пример не так прост, как вам кажется. Не стоит забывать, что вы программируете не на Си, а на C#, где все представлено в виде объектов. Полный код приложения приведен в листинге 3.7, а результат выполнения программы - на рис. 3.13.

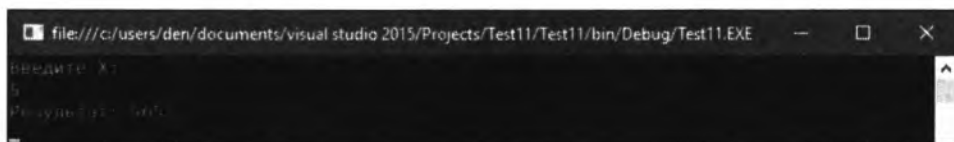


Рис. 3.13. Результат выполнения программы

Листинг 3.7. Вычисляем значение функции

```

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

namespace Test11
{
    class Program
    {
        static double f(double x)
        {
            return 3 * Math.Pow(x, 3) - 2 * Math.Pow(x, 2);
        }

        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.InputEncoding = Encoding.GetEncoding(866);

            Console.WriteLine("Введите X:");
            string t = Console.ReadLine();

            int x = Convert.ToInt32(t);
            int i;
            double sum = 0;

            for (i = 1; i <= x; i++) sum = sum + f(i);

            Console.WriteLine("Результат: {0}", sum);
            Console.ReadLine();
        }
    }
}

```

Начнем с самого начала. Мы не можем использовать функцию `pow()`. В C# она называется `Math.Pow()`. Обратите внимание на регистр символов. Затем просто так создать функцию `f()` вы не можете. Нужно создать член класса. Подробнее о них мы поговорим в главе 5, а пока мы создадим лишь статический член класса, чтобы можно было вызывать функцию `f()` непосредственно, без указания имени класса - так наш код будет больше похож на старый Си-код:

```

static double f(double x)
{

```

```

    return 3 * Math.Pow(x, 3) - 2 * Math.Pow(x, 2);
}

```

Поскольку `Math.Pow()` возвращает значение типа **double**, то и наша функция будет возвращать значение типа **double**.

Далее пользователь должен ввести *X*, мы должны прочесть ввод и преобразовать его в целое число. Для преобразования мы будем использовать метод `ToInt32()` класса `Convert`:

```

Console.WriteLine("Введите X:");
string t = Console.ReadLine();

int x = Convert.ToInt32(t);
int i;

```

Ну а дальше - дело техники. В цикле `for()` пройтись от 1 до *x* и посчитать сумму. Тем не менее в нашей программе есть один недостаток - если пользователь введет строку, которую нельзя будет преобразовать в целое число, возникнет исключение. Однако пока мы о них ничего не знаем - они будут рассмотрены в главе 7.

3.13. Делаем консольный калькулятор

Сейчас мы напишем простенький калькулятор, который будет работать в консоли. Программа будет работать так: пользователь вводит операнды и оператор, а программа сообщает результат.

Наш калькулятор будет поддерживать только базовые операторы `+`, `-`, `*`, `/`. Вместо обработки исключения деления на 0 (гл. 7) мы произведем проверку ввода - если второй оператор будет равен 0, мы просто не будем производить вычисление. Готовая программа приведена в листинге 3.8 и изображена на рис. 3.14.

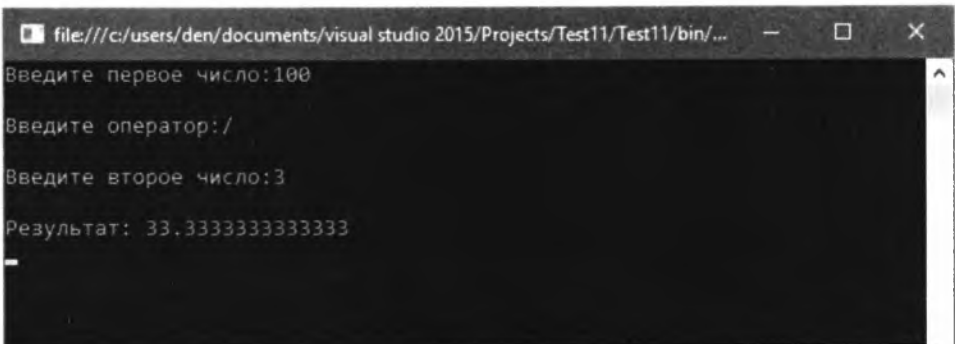


Рис. 3.14. Консольный калькулятор

Листинг 3.8. Консольный калькулятор

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

namespace Test11
{
    class Program
    {
        static double f(double x)
        {
            return 3 * Math.Pow(x, 3) - 2 * Math.Pow(x, 2);
        }

        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.InputEncoding = Encoding.GetEncoding(866);

            double a;
            double b;
            double res=0;
            char oper;

            Console.Write("Введите первое число:");
            a = Convert.ToDouble(Console.ReadLine());

            Console.Write("\nВведите оператор:");
            oper = Convert.ToChar(Console.ReadLine());

            Console.Write("\nВведите второе число:");
            b = Convert.ToDouble(Console.ReadLine());

            if (oper == '+')
            {
                res = a + b;
            }

            else if (oper == '-')
            {
                res = a - b;
            }
        }
    }
}
```

```

else if (oper == '*')
{
    res = a * b;
}

else if (oper == '/')
{
    if (b != 0)
        res = a / b;
    else Console.WriteLine("На 0 делить нельзя!");
}
else
{
    Console.WriteLine("Неизвестный оператор.");
}

Console.WriteLine("\nРезультат: {0}", res);
Console.ReadLine();
}
}
}

```

3.14. Графический калькулятор

Настало время разработать графический калькулятор. Он также довольно прост, однако нужно обратить внимание на некоторые моменты. Разработку калькулятора начнем с разработки формы приложения. Конечная форма изображена на рис. 3.15. На форму нужно поместить кнопки, текстовое поле, а также надпись. Надпись нужна, чтобы пользователь видел первый операнд. Когда пользователь введет, скажем, 5, нажмет кнопку +, надпись будет содержать значение "5+" - операнд и оператор.

Теперь посмотрим, как работает калькулятор. Мы используем четыре члена класса:

```

float a, b;
int oper;
bool znak = true;

```

Первые два - это наши операнды, **a** и **b**. Третий - это выбранный пользователем оператор. Четвертый - признак знака. Обработчик кнопки +/-, изменяющий знак числа, выглядит так:

```

if (znak==true)
{
    textBox1.Text = "-" + textBox1.Text;
}

```



Рис. 3.15. Конечная форма

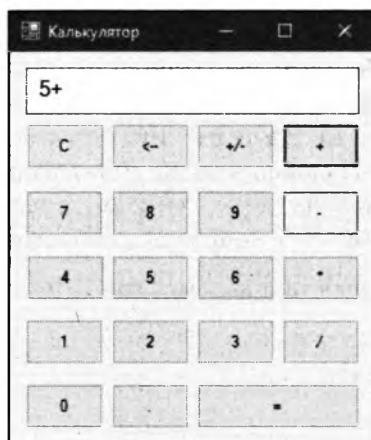


Рис. 3.16. Калькулятор в действии: введен операнд и оператор

```

    znak = false;
}
else if (znak==false)
{
    textBox1.Text=textBox1.Text.Replace("-", "");
    znak = true;
}

```

}
Как видите, если `znak = true`, мы меняем знак операнда в `textBox1`, а если `false`, то знак числа удаляется из текстового поля.

Для каждой цифровой кнопки обработчик будет такой:

```
// Обработчик нажатия кнопки 1
private void button13_Click(object sender, EventArgs e)
{
    textBox1.Text = textBox1.Text + 1;
}
```

Обработчик кнопки оператора (+, - и т.д.) выглядит так:

```
// обработчик нажатия кнопки +
private void button4_Click(object sender, EventArgs e)
{
    a = float.Parse(textBox1.Text);
    textBox1.Clear();
    oper = 1;
    label1.Text = a.ToString() + "+";
    znak = true;
}
```

Здесь мы в переменную `a` (первый операнд) записываем значение из текстового поля, попутно преобразовав его в `float` посредством вызова метода `float.Parse()`. Далее мы очищаем текстовое поле, чтобы можно было вводить второй операнд. Понятное дело, устанавливаем номер оператора (для + - это 1) и заносим в надпись информацию об операнде и операторе.

Обработчик кнопки `","` просто добавляет запятую в текстовое поле:

```
textBox1.Text = textBox1.Text + ",";
```

Обработчик кнопки `=` должен вычислить результат. Для этого он вызывает метод `calc()` и очищает текст надписи:

```
calc();
label1.Text = "";
```

Метод `calc()` выглядит так:

```
private void calculate()
{
    switch (oper)
    {
        case 1:
            b = a + float.Parse(textBox1.Text);
```

```

        textBox1.Text = b.ToString();
        break;
    case 2:
        b = a - float.Parse(textBox1.Text);
        textBox1.Text = b.ToString();
        break;
    case 3:
        b = a * float.Parse(textBox1.Text);
        textBox1.Text = b.ToString();
        break;
    case 4:
        b = a / float.Parse(textBox1.Text);
        textBox1.Text = b.ToString();
        break;

    default:
        break;
}
}

```

Здесь тоже все просто: мы определяем оператор и вычисляем значение. Обратите внимание: мы не производим проверку деления на 0. Пусть это будет вашим домашним заданием.

3.15. "Угадай число". Игра

В завершение этой главы мы разработаем простую программу - игру в угадывание чисел. Компьютер загадает число от 0 до 9, выдаст подсказку - больше ли это число 5 или нет, затем сравнит введенное пользователем число с загаданным. "Загадывание" осуществляется с помощью генератора случайных чисел:

```

Random rand = new Random();
int i = rand.Next(10);

```

В метод Next() нужно передать верхнюю границу диапазона, причем сама граница в диапазон не входит. То есть если вы вызываете rand.Next(10), то число будет сгенерировано от 0 до 9.

Полный код программы приведен в листинге 3.9.

Листинг 3.9. Игра "Угадай число"

```

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

```

```

using System.Text;

namespace GuessTheNumber
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.InputEncoding = Encoding.GetEncoding(866);

            char again = 'y';
            Random rand = new Random();

            while (again == 'y') {

                int i = rand.Next(10);

                Console.WriteLine("Компьютер загадал число от 0 до 9");

                if (i < 5) Console.WriteLine("Число меньше 5");
                else Console.WriteLine("Число больше или равно 5");

                int x = Convert.ToInt32(Console.ReadLine());

                if (i == x) Console.WriteLine("Вы победили! Поздравляем!");
                else Console.WriteLine("К сожалению, вы проиграли.
                    Компьютер загадал число {0}", i);

                Console.WriteLine("Попробовать еще? (y = Да, n = Нет)");
                again = Convert.ToChar(Console.ReadLine());
            }
        }
    }
}

```

В следующих главах будет рассмотрено объектно-ориентированное программирование в .NET.



```
File:///c:/users/den/documents/visual studio 2015/Projects/GuessTheNumber/GuessTheNumber/bin/Debug/GuessTheNumber.EXE
Компьютер загадал число от 0 до 9
число меньше 5
Вы выиграли! Поздравляем!
Попробовать еще? (y = Да, n = Нет)
Компьютер загадал число от 0 до 9
число больше или равно 5
К сожалению, вы проиграли. Компьютер загадал число 8
Попробовать еще? (y = Да, n = Нет)
```

Рис. 3.17. Игра в действии

Глава 4.

Файловый ввод/вывод



До этого мы рассматривали примеры, обрабатывающие какие-то данные. Как правило, сами данные или вводил пользователь, или же они были заданы в приложении. Но в реальном мире все немного сложнее - приложению нужно получать данные извне, а после их обработки часто нужно не просто вывести, но и сохранить результат. Именно поэтому просто необходима глава, объясняющая файловый ввод/вывод в С#. Кроме того, мы рассмотрим сериализацию объектов, то есть приведение объекта в форму, в которой можно его сохранить в обычном файле, а затем загрузить по мере необходимости.

В .NET существует очень много классов и методов, связанных с файловым вводом/выводом. Рассмотрение всех методов выходит за рамки этой книги (можно было бы написать отдельную книгу, посвященную только вводу/выводу). Мы рассмотрим только самые важные классы, тем не менее, как вы увидите далее, данная глава тоже получится не маленькой.

В этой главе, как было уже отмечено, рассматриваются две темы. Первая просто связана с файловым вводом/выводом. Соответствующие типы вы можете найти в пространстве имен System.IO. Вторая тема - это сериализация, то есть представление в символьном виде различных объектов. После того как объект сериализован, его можно сохранить в файл (или в базу данных) или передать на удаленную машину посредством технологии WCF (Windows Communication Foundation).

4.1. Введение в пространство имен System.IO

Все, что так или иначе связано с вводом/выводом, хранится в пространстве имен System.IO. В .NET это пространство имен посвящено службам файлового ввода-вывода, а также ввода-вывода из памяти. В этом пространстве определен набор классов, интерфейсов, перечислений, структур и делегатов, большинство из которых физически находятся в mscorlib.dll. В таблице 4.1 представлены основные классы из пространства имен System.IO.

Таблица 4.1. Основные классы из пространства имен System.IO.

Класс	Описание
BinaryReader, BinaryWriter	Позволяют читать и записывать элементарные типы данных (целочисленные, булевские, строковые и т.п.) в двоичном виде
BufferedStream	Предоставляет временное хранилище для потока байтов, которые могут затем быть перенесены в постоянные хранилища
Directory, DirectoryInfo	Используются для манипуляций с каталогами
DriveInfo	Предоставляет подробную информацию о дисковых устройствах
File, FileInfo	Используются для манипуляций с файлами
FileStream	Обеспечивает произвольный доступ к файлу с данными, представленными в виде потока байтов
MemoryStream	Обеспечивает произвольный доступ к данным, хранящимся в памяти, а не в физическом файле
Path	Предоставляет информацию о пути к файлу/каталогу
StreamReader, StreamWriter	Используются для чтения и записи текстовой информации из файла. Не поддерживают произвольного доступа к файлу.
StringReader, StringWriter	Также используются для чтения и записи текстовой информации из файла. Однако в качестве хранилища они используют строковый буфер, а не физический файл

4.2. Классы для манипуляции с файлами и каталогами

В System.IO определено четыре класса для манипуляции с файлами и каталогами - File, FileInfo, Directory, DirectoryInfo. Классы Directory и File используются для создания, удаления, копирования и перемещения каталогов и файлов соответственно. Классы DirectoryInfo и FileInfo также используются для манипуляций с каталогами и файлами, но предлагают свою функциональность в виде методов уровня экземпляра - они должны размещаться в памяти с помощью ключевого слова **new**.

Классы `DirectoryInfo` и `FileInfo` получили от абстрактного базового класса `FileSystemInfo` значительную часть своего функционала. Члены класса `FileSystemInfo` используются для получения общих характеристик (таких как время создания, различные атрибуты и т.д.). Давайте рассмотрим наиболее полезные свойства этого класса:

- `Attributes` - получает или устанавливает ассоциированные с текущим файлом атрибуты (только чтение, системный, скрытый, сжатый).
- `CreationTime` - получает/устанавливает время создания файла/каталога.
- `Exists` - используется для определения, существует ли данный файл/каталог.
- `Extension` - извлекает расширение файла.
- `FullName` - полный путь к файлу/каталогу.
- `LastAccessTime` - получает/устанавливает время последнего доступа к текущему файлу или каталогу.
- `LastWriteTime` - получает/устанавливает время последней записи в файл/каталог.
- `Name` - содержит имя файла/каталога.

4.2.1. Использование класса `DirectoryInfo`

Для работы с каталогами удобнее всего использовать класс `DirectoryInfo`. Рассмотрим основные члены этого класса:

- `Create()`, `CreateSubdirectory()` - создает каталог или дерево каталогов по заданному имени.
- `Delete()` - удаляет каталог вместе со всем содержимым. Обратите внимание, в отличие от некоторых системных функций, удаляющих только пустой каталог, этот метод удаляет непустой каталог, что очень удобно - не нужно извлекать обход дерева каталогов.
- `GetDirectory()` - возвращает в себе массив объектов `DirectoryInfo`, представляющий собой все подкаталоги заданного (текущего) каталога.
- `GetFiles()` - извлекает массив объектов `FileInfo`, представляющий собой все файлы из текущего каталога.
- `CopyTo()` - копирует каталог со всем содержимым.

- `MoveTo()` - перемещает весь каталог (с файлами и подкаталогами).
- `Parent` - содержит родительский каталог.
- `Root` - содержит корневой каталог.

Чтобы начать работу с `DirectoryInfo`, нужно создать новый объект этого типа, указав каталог, с которым вы будете работать:

```
DirectoryInfo d = new DirectoryInfo("C:\Files");
```

Указать можете как существующий, так и несуществующий каталог. Если вы пытаетесь привязаться к несуществующему каталогу, то будет сгенерировано исключение `System.IO.DirectoryNotFoundException`. Если нужно создать несуществующий каталог, то укажите перед его именем знак `@`, например:

```
DirectoryInfo d = new DirectoryInfo(@"C:\Files\Den");
d.Create();           // создаем новый каталог
```

Для создания подкаталогов используется метод `CreateSubdirectory()`. Пример:

```
DirectoryInfo d = new DirectoryInfo(@"C:\Files\Den");
d.CreateSubdirectory("Data");
d.CreateSubdirectory("Help\Html");
```

В результате в каталоге `C:\Files\Den` будут созданы подкаталоги `Data` и `Help\Html`. Обратите внимание на то, что данный метод удобно использовать для создания целого дерева подкаталогов.

После того как мы создали объект `DirectoryInfo`, можно исследовать его содержимое, используя любое свойство, которое было унаследовано от класса `FileSystemInfo`. Пример:

```
DirectoryInfo d = new DirectoryInfo(@"C:\Test");

Console.WriteLine("FullName: {0}", d.FullName);
Console.WriteLine("Name: {0}", d.Name);
Console.WriteLine("Parent: {0}", d.Parent);
Console.WriteLine("Creation: {0}", d.CreationTime);
Console.WriteLine("Attributes: {0}", d.Attributes);
Console.WriteLine("Root: {0}", d.Root);
```

Получить содержимое каталога (список файлов) можно с помощью метода `GetFiles()` класса `DirectoryInfo`. Данному классу нужно передать маску файлов и опции поиска. В качестве результата метод возвращает массив объектов типа `FileInfo`, каждый из которых используется для предоставления информации о конкретном файле. Пример кода:

```

Directory d = new DirectoryInfo(@"C:\test");
FileInfo files = d.GetFiles("*.doc", SearchOption.
AllDirectories);
Console.WriteLine("Found {0} documents", files.Length);
// Выводим файлы:
foreach (FileInfo f in files)
{
    Console.WriteLine("Filename: {0}", f.Name);
    Console.WriteLine("Size: {0}", f.Length);
    Console.WriteLine("Attributes: {0}", f.Attributes);
}

```

4.2.2. Классы Directory и DriveInfo. Получение списка дисков

После рассмотрения класса DirectoryInfo можно приступить к классу Directory. Члены этого класса обычно возвращают строковые данные, а не типизированные объекты типов FileInfo/DirectoryInfo - в этом основная разница между этими двумя классами.

Сейчас мы, используя этот класс, создадим более интересное приложение, состоящее из всего четырех строк рабочего кода, а именно мы выведем список дисков вашего компьютера:

```

Console.WriteLine("Ваши диски:");
string[] drives = Directory.GetLogicalDrives();
foreach (string s in drives)
    Console.WriteLine("{0}", s);

```

Во-первых, обратите внимание, что метод GetLogicalDrives() возвращает массив строк, а не массив объектов DirectoryInfo. Во-вторых, метод вызывается без предварительного создания объекта оператором new.

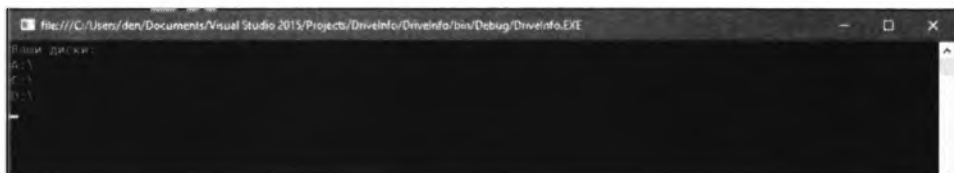


Рис. 4.1. Использование метода GetLogicalDrives()

Метод GetLogicalDrives() не информативен. Он просто сообщает буквы логических дисков, при этом невозможно понять, где какой диск. Если нужно получить больше информации о дисках, нужно использовать тип DriveInfo:

```

DriveInfo[] drvs = DriveInfo.GetDrives();
foreach (DriveInfo d in drvs)

```

```

{
    Console.WriteLine("Диск: {0} Type {1}", d.Name,
d.DriveType);
    if (d.IsReady) {
        Console.WriteLine("Свободно: {0}", d.TotalFreeSpace);
        Console.WriteLine("Файловая система: {0}", d.DriveFormat);
        Console.WriteLine("Метка: {0}", d.VolumeLabel);
    }
    Console.WriteLine();
}

```

Полный код приложения приведен в листинге 4.1.

Листинг 4.1. Информация о дисках

```

using System;
using System.IO;

namespace MyDriveInfo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Изменяем кодировку консоли для вывода текста
            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.WriteLine("Ваши диски:");
            string[] drives = Directory.GetLogicalDrives();
            foreach (string s in drives)
                Console.WriteLine("{0}", s);

            DriveInfo[] drvs = DriveInfo.GetDrives();

            foreach (DriveInfo d in drvs)
            {
                Console.WriteLine("Диск: {0} Тип {1}", d.Name, d.DriveType);
                if (d.IsReady)
                {
                    Console.WriteLine("Свободно: {0}", d.TotalFreeSpace);
                    Console.WriteLine("Файловая система: {0}", d.DriveFormat);
                    Console.WriteLine("Метка: {0}", d.VolumeLabel);
                    Console.WriteLine();
                }
            }
            Console.ReadLine();
        }
    }
}

```

```
    }  
}
```

Вывод этого приложения показан на рис. 4.2. Видно, что сначала мы просто выводим список дисков методом `GetLogicalDrives()`. Далее мы используем метод `GetDrives()` и получаем информацию о дисках - как минимум мы получаем тип. Если диск смонтирован, что проверяется свойством `IsReady`, то мы выводим свободное пространство, тип файловой системы и метку диска.



Рис. 4.2. Пример использования метода `GetDrives()`

4.2.3. Класс `FileInfo`

Подобно классу `DirectoryInfo`, класс `FileInfo` используется для манипуляции с файлами. В таблице 4.2 приведены его основные члены.

Таблица 4.2. Основные члены класса `FileInfo`

Член	Описание
<code>Create()</code>	Создает новый файл и возвращает объект <code>FileStream</code> , который используется для взаимодействия с созданным файлом
<code>CreateText()</code>	Создает объект <code>StreamWriter</code> , который используется для создания текстового файла
<code>CopyTo()</code>	Копирует существующий файл в другой файл
<code>AppendText()</code>	Создает объект <code>StreamWriter</code> для добавления текста в файл
<code>Delete()</code>	Удаляет файл, связанный с экземпляром <code>FileInfo</code>
<code>Directory</code>	Используется для получения экземпляра родительского каталога
<code>DirectoryName</code>	Содержит полный путь к родительскому каталогу
<code>Length</code>	Получает размер текущего файла или каталога
<code>MoveTo()</code>	Используется для перемещения файла
<code>Name</code>	Содержит имя файла
<code>Open()</code>	Открывает файл с различными разрешениями чтения/записи

<code>OpenRead()</code>	Создает доступный только для чтения объект <code>FileStream</code>
<code>OpenText()</code>	Создает объект <code>StreamReader</code> для чтения информации из текстового файла
<code>OpenWrite()</code>	Создает объект <code>FileStream</code> , доступный только для записи

Теперь рассмотрим примеры использования класса `FileInfo`. Начнем с примеров использования метода `Create()`:

```
FileInfo myFile = new FileInfo(@"C:\temp\file.vdd");
FileStream fs = myFile.Create();
// производим какие-либо операции с fs
// закрываем поток
fs.Close();
```

Итак, мы создали поток типа `FileStream`, позволяющий производить манипуляции с содержимым файла, например, читать из него данные, записывать в него данные. Позже этот поток будет подробно рассмотрен, пока разберемся с созданием и открытием файлов.

Для открытия файла используется метод `Open()`, позволяющий как открывать существующие файлы, так и создавать новые. Причем этот метод принимает несколько параметров, в отличие от метода `Create()`, что позволяет более гибко управлять процессом открытия/создания файлов. В результате выполнения метода `Open()` создается объект типа `FileStream`, как и в предыдущем случае:

```
FileInfo mf = new FileInfo(@"C:\temp\file.vdd");
FileStream fs = mf.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None);
```

Методу `Open()` нужно передать три параметра. Первый из них - тип запроса ввода/вывода из перечисления типа `FileMode`:

```
public enum FileMode
{
    CreateNew,
    Create,
    Open,
    OpenOrCreate,
    Truncate,
    Append
}
```

Рассмотрим члены этого перечисления:

- `CreateNew` - создать новый файл, если он существует, будет сгенерировано исключение `IOException`.
- `Create` - создать новый файл, если он существует, он будет перезаписан.
- `Open` - открыть существующий файл. Если он не существует, будет сгенерировано исключение `FileNotFoundException`.
- `OpenOrCreate` - открывает файл, если он существует. Если файл не существует, он будет создан.
- `Truncate` - открывает файл и усекает его до нулевой длины. По сути, удаляет все его содержимое.
- `Append` - открывает файл и переходит в самый его конец. Этот флаг может использоваться только для потоков, которые открыты только для чтения. Если файл не существует, он будет создан.

Второй параметр метода `Open()` - одно из значений перечисления `FileAccess`. Используется для определения операций чтения/записи:

```
public enum FileAccess
{
    Read,
    Write,
    ReadWrite
}
```

Думаю, значения членов перечисления `FileAccess` в комментариях не нужны. Третий параметр метода `Open()` - член перечисления `FileShare`, задающий тип совместного доступа к этому файлу:

```
public enum FileShare
{
    Delete,
    Inheritable,
    None,
    Read,
    ReadWrite,
    Write
}
```

Методы `OpenRead()` и `OpenWrite()` используются для создания потоков, доступных только для чтения и только для записи. Данные методы возвращают поток `FileStream`, сконфигурированный соответствующим образом. В принципе, можно было бы обойтись только методом `Open()`, но использовать эти методы немного удобнее - ведь вам не нужно применять различные значения из перечислений. Примеры:

```

FileInfo f = new FileInfo(@"C:\temp\1.dat");
using (FileStream ro = f.OpenRead())
{
    // выполняем операции чтения из файла
}
FileInfo f2 = new FileInfo(@"C:\temp\2.dat");

using (FileStream rw = f2.OpenWrite())
{
    // записываем в файл
}

```

Для работы с текстовыми файлами пригодятся методы `OpenText()`, `CreateText()` и `AppendText()`. Первый метод возвращает экземпляр типа `StreamReader` (в отличие от методов `Create()`, `Open()` и `OpenRead/Write()`, которые возвращают тип `FileStream`).

Пример открытия текстового файла:

```

FileInfo txt = new FileInfo(@"program.log");
using (StreamReader txt_reader = txt.OpenText ())
{
    // Читаем данные из текстового файла
}

```

Методы `CreateText()` и `AppendText()` возвращают объект типа `StreamWriter`. Использовать эти методы можно так:

```

FileInfo f = new FileInfo(@"C:\temp\1.txt");
using (StreamWriter sw = f.CreateText())
{
    // Записываем данные в файл...
}
FileInfo f = new FileInfo(@"C:\temp\2.txt");
using (StreamWriter swa = f.AppendText())
{
    // Добавляем данные в текстовый файл
}

```

4.2.4. Класс `File`

Тип `File` поддерживает несколько полезных методов, которые пригодятся при работе с текстовыми файлами. Например, метод `ReadAllLines()` позволяет открыть указанный файл и прочитать из него все строки - в результате будет возвращен массив строк. После того как все данные из файла прочитаны, файл будет закрыт.

Аналогично, метод `ReadAllBytes()` читает все байты из файла, возвращает массив байтов и закрывает файл.

Метод `ReadAllText()` читает все содержимое текстового файла в одну строку и возвращает ее. Как обычно, файл после чтения будет закрыт.

Существуют и аналогичные методы записи `WriteAllBytes()`, `WriteAllLines()` и `WriteAlltext()`, которые записывают в файл, соответственно, массив байтов, массив строк и строку. После записи файл закрывается.

Пример:

```
string[] myFriends = {"Jane", "Max", "John" };

// Записать все данные в файл
File.WriteAllLines(@"C:\temp\friends.txt", myFriends);

// Прочитать все обратно и вывести
foreach (string friend in File.ReadAllLines(@"C:\temp\friends.
txt"))
{
    Console.WriteLine("{0}", friend);
}
```

4.2.5. Классы `Stream` и `FileStream`

На данный момент вы знаете, как открыть и создать файл, но как прочитать из него информацию? Как сохранить информацию в файл? Методы класса `File` хороши, но они подходят только в самых простых случаях. При работе с текстовыми файлами их вполне достаточно, а вот при работе с двоичными файлами методов `ReadAllBytes()` и `WriteAllBytes()` будет маловато.

В абстрактном классе `System.IO.Stream` определен набор членов (см. табл. 4.3), которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, файлом или областью памяти).

Таблица 4.3. Члены абстрактного класса `System.IO.Stream`

Член	Описание
<code>CanRead</code> , <code>CanWrite</code> , <code>CanSeek</code>	Определяют, поддерживает ли поток чтение, запись, поиск соответственно
<code>Close()</code>	Метод закрывает поток и освобождает все ресурсы
<code>Flush()</code>	Обновляет лежащий в основе источник данных. Например, позволяет перечитать источник или же сбросить содержимое буферов ввода/вывода на диск. Если поток не реализует буфер, этот метод ничего не делает
<code>Length</code>	Возвращает длину потока в байтах
<code>Position</code>	Определяет текущую позицию в потоке

Read(), ReadByte()	Читает последовательность байтов или один байт соответственно из текущего потока и перемещает позицию потока на количество прочитанных байтов
Seek()	Устанавливает позицию в текущем потоке
SetLength()	Устанавливает длину текущего потока
Write(), WriteByte()	Записывает последовательность байтов или одиночный байт в текущий поток и перемещает текущую позицию на количество записанных байтов

Класс `FileStream` предоставляет реализацию абстрактного члена `Stream` для потоковой работы с файлами. Это элементарный поток, и он может записывать или читать только один байт или массив байтов. Однако программисты редко используют `FileStream` непосредственно, гораздо чаще они используют оболочки потоков, облегчающие работу с текстовыми данными или типами .NET. Но в целях иллюстрации мы рассмотрим возможности синхронного чтения/записи типа `FileStream` (лист. 4.2).

Листинг 4.2. Использование типа `FileStream`

```
// Получаем объект типа FileStream
using(FileStream fStream = File.Open(@"C:\temp\test.dat",
    FileMode.Create))
{
    // Кодировем строку в виде массива байтов
    string txt = "Test!";
    byte[] txtByteArray = Encoding.Default.GetBytes(txt);

    // Записываем массив в файл
    fStream.Write(txtByteArray, 0, txtByteArray.Length);

    // Сбрасываем position
    fStream.Position = 0;

    // Читаем из файла и выводим на консоль

    byte[] bytesFromFile = new byte[txtByteArray.Length];
    for (int i = 0; i < txtByteArray.Length; i++)
    {
        bytesFromFile[i] = (byte)fStream.ReadByte();
        Console.Write(bytesFromFile[i]);
    }

    Console.WriteLine();

    // Декодируем сообщение и выводим.
    Console.Write("Декодированное сообщение: " );
}
```

```
Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
}
```

Пример показывает не только наполнение файла тестовыми данными, но и демонстрирует основной недостаток при работе с типом `FileStream`: вам необходимо выполнять низкоуровневое кодирование и декодирование информации, которую вы записываете и которую вы читаете. Это очень неудобно. Но иногда ситуация этого требует. Например, когда нужно записать поток байтов в область памяти (используется класс `MemoryStream`) или в сетевое соединение (класс `NetworkStream` из пространства имен `System.Net.Sockets`).

4.2.6. Классы `StreamWriter` и `StreamReader`

Классы `StreamWriter` и `StreamReader` удобно использовать во всех случаях, когда нужно читать или записывать символьные данные (например, строки). Оба типа работают по умолчанию с символами `Unicode`, но кодировку можно изменить предоставлением правильно сконфигурированной ссылки на объект `System.Text.Encoding`.

Класс `StreamReader` унаследован от абстрактного класса `TextReader`, основные методы которого следующие:

- `Close()` - закрывает объект и освобождает ресурсы.
- `Flush()` - очищает все буферы объекта-писателя и записывает все буферизированные данные на устройство, но объект при этом не закрывается.
- `NewLine` - свойство, позволяющее задать константу перевода строки. В `Windows` используется `\r\n`.
- `Write()` - позволяет записывать данные в текстовый поток без добавления константы новой строки.
- `WriteLine()` - записывает данные в текстовый поток с добавлением константы новой строки.

Рассмотрим пример записи в файл с использованием `StreamWriter`:

```
using (StreamWriter writer = File.CreateText("friends.txt"))
{
    writer.WriteLine("Вася");
    writer.WriteLine("Ира");
    writer.WriteLine("Олег");
}
```

Для чтения информации из текстового файла используется `StreamReader`, который унаследован от абстрактного класса `TextReader`, обладающего следующими методами:

- `Peek()` - возвращает следующий доступный символ, не изменяя текущей позиции читателя. Если достигнут конец потока, возвращается -1.
- `Read()` - читает данные из входного потока.
- `ReadLine()` - читает строку символов из текущего потока. Если достигнут конец потока, возвращается `null`.
- `ReadToEnd()` - читает все символы, начиная с текущей позиции и до конца потока. Прочитанные символы возвращаются в виде одной строки.

Пример:

```
using (StreamReader reader1 = File.OpenText("friends.txt"))
{
    string s = null;
    while ((s = reader1.ReadLine()) != null)
    {
        Console.WriteLine(s);
    }
}
```

4.2.7. Классы `BinaryWriter` и `BinaryReader`

Для работы с двоичными (не текстовыми) данными используются классы `BinaryWriter` и `BinaryReader`. Оба эти класса унаследованы от `System.Object` и позволяют читать и записывать данные в потоки в двоичном формате. Для записи у `BinaryWriter` используется метод `Write()`, а для чтения - метод `Read()` у `BinaryReader`. Метод `Close` (есть у обоих классов) позволяет закрыть поток.

Метод `Flush()` у `BinaryWriter` позволяет сбросить буфер двоичного потока на носитель.

Пример использования:

```
FileInfo f = new FileInfo(@"file.dat");
using (BinaryWriter bw = new BinaryWriter(f.OpenWrite()))
{
    // Данные для записи
    double a = 1234.56;
    int b = 123456;
    string s = "123456";
    // Записать данные.
    bw.Write(a);
```

```

    bw.Write(b);
    bw.Write(s);
}
...
// читаем данные
using(BinaryReader br = new BinaryReader(f.OpenRead()))
{
    Console.WriteLine(br.ReadDouble());
    Console.WriteLine(br.ReadInt32());
    Console.WriteLine(br.ReadString());
}

```

4.3. Сериализация объектов

Настало время поговорить о сериализации объектов. Для начала нужно разобраться, что такое сериализация. Сериализация описывает процесс сохранения (и, возможно, передачи) состояния объекта в потоке. Именно в потоке. Потоком может выступать память, файл, сетевое соединение.

Последовательность сохраняемых данных содержит всю необходимую информацию, необходимую для восстановления (или десериализации) состояния объекта с целью последующего использования.

С помощью сериализации программист может сохранить большие объемы разных данных с минимальными усилиями.

Представим, что вам нужно сохранить настройки приложения в файле. Можно просто создать текстовый файл (например, program.ini), записать в него настройки и при запуске приложения обрабатывать этот файл, читая из него тот или иной конфигурационный параметр. Можно пойти по другому пути и записать в файл целый класс, отвечающий за хранение параметров пользователя. Тогда достаточно при загрузке программы выполнить десериализацию этого класса (восстановить настройки), а при завершении работы - сериализовать класс в файл (сохранить настройки).

Представим, что у нас есть класс `UserSettings`, содержащий параметры, установленные пользователем. Чтобы этот класс можно было сериализовать, его нужно снабдить атрибутом `Serializable`:

```

[Serializable]
public class UserSettings
{
    public string WorkDir = "C:\\temp";
    public int TimeOut = 2000;
}

```

Теперь давайте посмотрим, как сериализовать этот класс. Первым делом нужно создать объект класса:

```
UserSettings currentSettings = new UserSettings();
currentSettings.TimeOut = 1000; // устанавливаем конкретное значение
```

Для сериализации нужно создать объект типа `BinaryFormatter` (определен в пространстве имен `System.Runtime`):

```
using System.Runtime;
...
BinaryFormatter bf = new BinaryFormatter();
using (Stream fs = new FileStream("settings.dat", FileMode.
Create, FileAccess.Write, FileShare.None))
{
    bf.Serialize(fs, currentSettings);
}
```

Для десериализации нужно выполнить метод `Deserialize()`:

```
FileStream fs = new FileStream("settings.dat", FileMode.Open);
try
{
    BinaryFormatter bf = new BinaryFormatter();
    currentSettings = (UserSettings)bf.Deserialize(fs);
}
catch (SerializationException e)
{
    Console.WriteLine("Ошибка. Причина: " + e.Message);
    throw;
}
finally
{
    fs.Close();
}
```

Посмотрите, как мы выполняем десериализацию объекта. Первым делом мы открываем файл для чтения. Далее мы создаем новый объект класса `BinaryFormatter` и десериализируем поток `fs`. Если в результате возникнет исключение, то мы выводим об этом сообщение. Блок **finally** позволяет в любом случае закрыть поток `fs` и освободить ресурсы.

Нужно отметить, что `BinaryFormatter` - это не единственный класс для сериализации объекта. Он сохраняет объект в двоичном формате. Однако, если вам нужно сохранить объект в формате XML или в формате SOAP, то вам нужно использовать, соответственно, классы `XmlSerializer` или `SoapFormatter`.

Тип `SoapFormatter` сохраняет состояние объекта в виде сообщения SOAP (стандартный XML-формат для передачи и приема сообщений от веб-служб). Этот тип определен в пространстве имен `System.Runtime.Serialization.Formatters.Soap`, находящемся в отдельной сборке. Поэтому для преобразования объектов в формат SOAP необходимо сначала установить ссылку на `System.Runtime.Serialization.Formatters.Soap.dll`, используя диалоговое окно **Add Reference** в Visual Studio, и затем указать следующую директиву **using**:

```
// Нужна ссылка на System.Runtime.Serialization.Formatters.Soap!
using System.Runtime.Serialization.Formatters.Soap;
```

Для преобразования объекта в формат XML имеется тип `XmlSerializer`. Чтобы использовать этот тип, нужно указать директиву **using** для пространства имен `System.Xml.Serialization` и установить ссылку на сборку `System.Xml.dll`. Однако шаблоны проектов Visual Studio автоматически ссылаются на `System.Xml.dll`, поэтому достаточно просто указать соответствующее пространство имен:

```
using System.Xml.Serialization;
```

Дополнительная информация может быть получена по адресу:

<https://msdn.microsoft.com/en-us/library/system.runtime.serialization.formatters.binary.binaryformatter%28v=vs.110%29.aspx>

4.4. Вывод содержимого файла на С#

Ранее мы познакомились с классом `StreamReader`, сейчас же рассмотрим небольшой пример его практического использования, а именно - вывод на консоль содержимого текстового файла.

Первым делом нужно создать сам файл. При сохранении файла в Блокноте нужно выбрать кодировку Юникод, поскольку почему-то по умолчанию даже в Windows 10 Блокнот хочет сохранять файлы в кодировке ANSI.

Вообще Windows в плане различных кодировок проявляет высшую степень идиотизма. Это вам не Linux, где все взяли в один момент и перешли на UTF-8. В конечном итоге в UTF-8 оказались все файлы, настроена консоль и графический интерфейс, переделаны файлы локализации (вывод осуществляется тоже в UTF-8). В Windows все как-то не как у людей. С одной стороны, поддерживается Юникод, с другой стороны, консоль почему-то до сих пор живет в кодировке CP-866, которая впервые появилась в MS DOS 4.01. На секунточку: эта операционная система была выпущена в декабря 1988 года. Кодировке, которая до сих пор использу-

ется в командной строке Windows 10, скоро будет небольшой юбилей - 30 лет. При этом блокнот почему-то сохраняет файлы в кодировке ANSI (CP-1251). Зачем это все делать? Обратная совместимость? Кто будет запускать приложения 20-летней давности?

Все это - риторические вопросы, поэтому возвращаемся к нашему приложению. Само приложение очень простое, но, как говорится, есть нюансы. Первый нюанс - это установка кодировки вывода:

```
Console.OutputEncoding = Encoding.GetEncoding(866);
```

Однако после данного изменения вы все равно вместо русских букв на консоли увидите много вопросительных знаков. И это понятно: нужно указать StreamReader, в какой кодировке нужно читать файл. У нас это Юникод - именно в этой кодировке мы сохранили файл:

```
StreamReader sr = new StreamReader("c:\\users\\den\\file.txt",  
System.Text.Encoding.Default);
```

Обратите внимание: по умолчанию, значит, используется Юникод (System.Text.Encoding.Default). Так почему же эта кодировка по умолчанию не используется в приложениях? В том же Блокноте?

Третий момент, на который стоит обратить внимание, - это на экранирование символа \ при указании пути к файлу. Об этом уже говорилось, но все равно хочется акцентировать еще раз ваше внимание.

Далее все просто - мы читаем строки из StreamReader и построчно выводим их на консоль. Полный код приложения приведен в листинге 4.3.

Листинг 4.3. Чтение и вывод файла на консоль

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using System.IO;  
  
namespace StreamReaderExample  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.OutputEncoding = Encoding.GetEncoding(866);  
            StreamReader sr = new StreamReader("c:\\users\\  
den\\file.txt", System.Text.Encoding.Default);
```

```

        string s;

        while (sr.EndOfStream != true)
        {
            s = sr.ReadLine();

            Console.WriteLine(s);
        }

        sr.Close();
        Console.ReadLine();
    }
}

```

На рис. 4.4 изображена среда VisualStudio, содержимое текстового файла в блокноте и запущенное приложение.

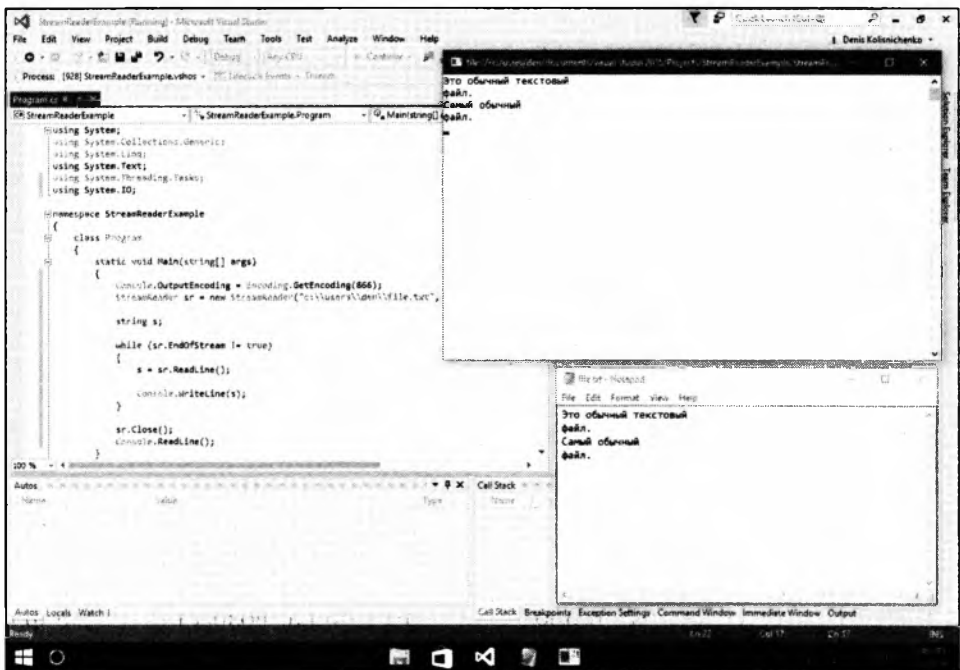


Рис. 4.4. Программа в действии

Обратите внимание, что изменен цвет консоли - черные символы на белом фоне. Если хотите изменить цвета консоли, щелкните правой кнопкой мыши на заголовке консоли, выберите команду **Свойства** и на вкладке **Цвет** установите нужные цвета.

4.5. Работа с XML-файлом

В прошлом разделе была продемонстрирована простая программа чтения обычного текстового файла. Сейчас же мы создадим приложение Windows Forms для чтения данных из XML-файла. Подробно останавливаться на формате XML мы не будем по двум причинам. Во-первых, формат XML прост и интуитивно понятен и с ним многие уже знакомы. Во-вторых, XML посвящено много информации в Интернете и найти ее не составляет труда. Лучше сконцентрируемся на главном - на чтении информации из XML.

В качестве исходного файла мы будем использовать файл с информацией о сотрудниках небольшой компании, см. лист. 4.4.

Листинг 4.4. Исходный XML-файл

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
  <Employee name="Evgeniy Orlov">
    <Age>35</Age>
    <Programmer>True</Programmer>
  </Employee>

  <Employee name="Sergey Pertov">
    <Age>25</Age>
    <Programmer>False</Programmer>
  </Employee>

  <Employee name="Mark Landau">
    <Age>30</Age>
    <Programmer>True</Programmer>
  </Employee>

  <Employee name="Vika Orlova">
    <Age>31</Age>
    <Programmer>False</Programmer>
  </Employee>
</root>
```

Наш файл содержит фамилию и имя сотрудника, возраст, а также флаг Programmer, свидетельствующий о том, что тот или иной человек является программистом.

Создать XML-файл можно или в любом текстовом редакторе, или в среде Visual Studio (команда File ► New ► File ► XML File), см. рис. 4.6. Создайте новый проект Windows Forms (рис. 4.7). Далее расположите на форме компоненты ListBox и PropertyGrid, чтобы наша форма была такой, как

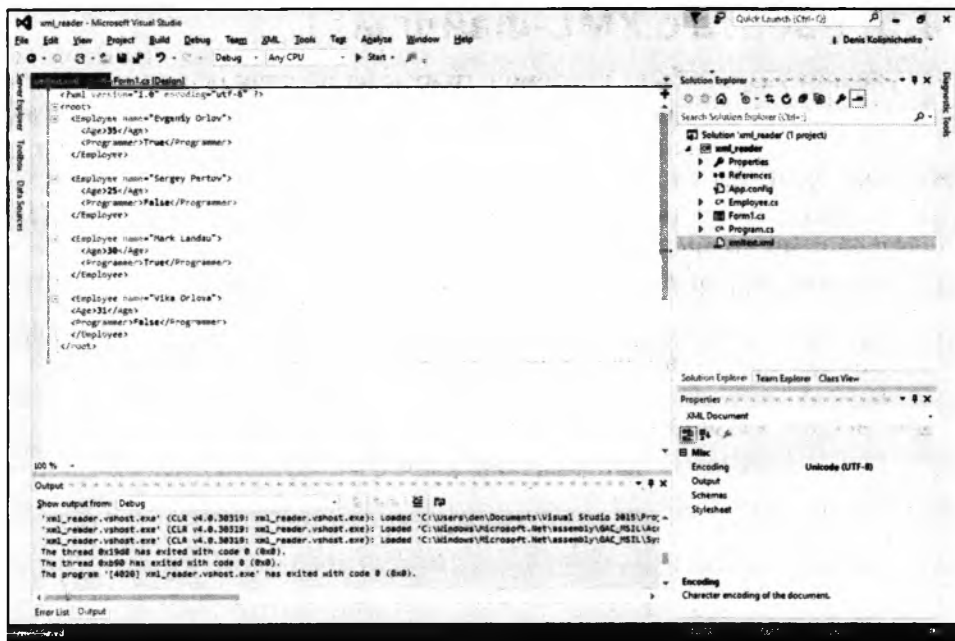


Рис. 4.5. Исходный XML-файл в среде Visual Studio



Рис. 4.6. Создание XML-файла в Visual Studio

показано на рис. 4.8. В ListBox мы будем загружать имена сотрудников, в PropertyGrid - "свойства" сотрудников.

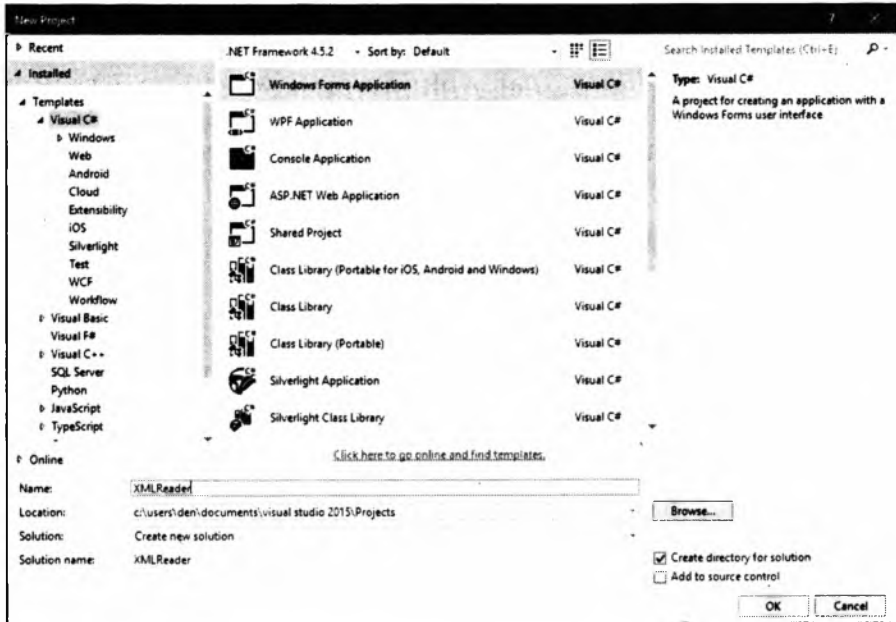


Рис. 4.7. Создание нового проекта Windows Forms Application

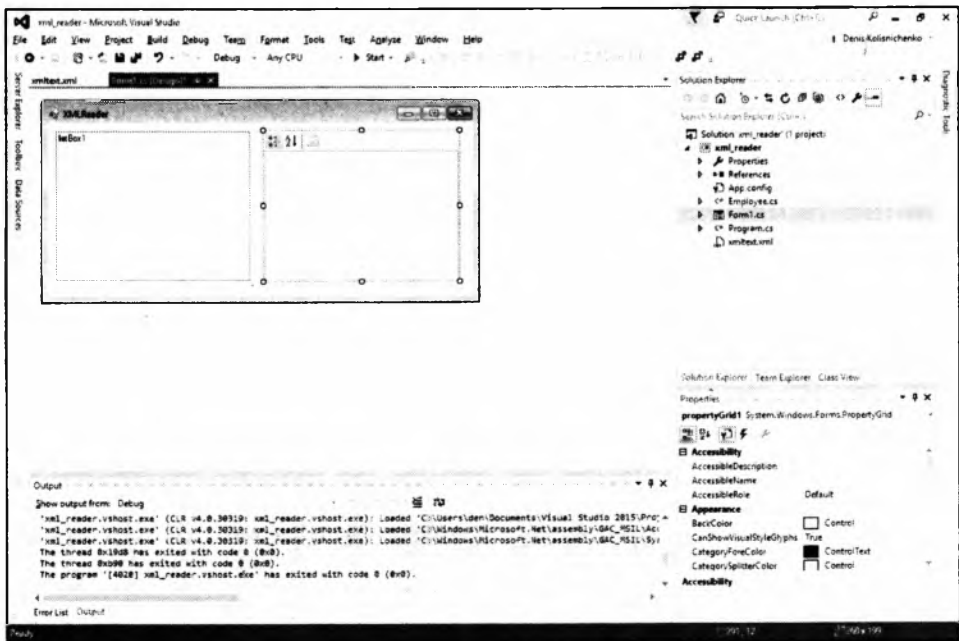


Рис. 4.8. Форма нашего приложения

Для работы с данными из XML-файла нам нужно создать вспомогательный класс. Выполните команду **Project » Add Class**. Данный класс очень зависит от формата XML-файла - данные читаемые из XML-файла должны быть помещены в этот класс, поэтому если вы создали отличный от приведенного XML-файл, вам придется изменить и этот класс. Код класса **Employee** приведен в листинге 4.5.

Листинг 4.5. Вспомогательный класс Employee

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

namespace xml_reader
{
    class Employee
    {
        public string Name { get; private set; }
        public int Age { get; private set; }
        public bool Programmer { get; private set; }

        public Employee(string name, int age, bool programmer)
        {
            Name = name;
            Age = age;
            Programmer = programmer;
        }

        public override string ToString()
        {
            return Name;
        }
    }
}
```

Как видите, класс очень простой. Однако, тем не менее, без него - никак. Конструктор класса просто инициализирует свойства класса. Также мы переопределили метод `ToString()` - в качестве строкового значения будет возвращаться значение свойства **Name**.

Теперь приступаем к редактированию файла кода формы - `Form1.cs`. Первым делом нужно добавить метод чтения из XML-файла:

```
private void LoadEmployees()
{
```

```

XmlDocument doc = new XmlDocument();
doc.Load("xmltext.xml");

foreach(XmlNode node in doc.DocumentElement)
{
    string name = node.Attributes[0].Value;
    int age = int.Parse(node["Age"].InnerText);
    bool programmer = bool.Parse(node["Programmer"].InnerText);
    listBox1.Items.Add(new Employee(name, age, programmer));
}
}

```

Метод Load() загружает XML-файл. Убедитесь, что указали правильное имя. Далее в цикле мы читаем все элементы XML-файла и загружаем в listBox1. Обратите внимание, как мы читаем имена сотрудников и их характеристики. Для доступа к элементам Age и Programmer мы используем имена элементов.

Чтобы класс XmlDocument был доступен, нужно добавить директиву:

```
using System.Xml;
```

В метод Form1() нужно добавить вызов метода чтения из XML-файла, чтобы он загружался при открытии формы:

```

public Form1()
{
    InitializeComponent();
    LoadEmployees();
}

```

Также нам надо сделать так, чтобы при выборе имени в ListBox все данные выводились в PropertyGrid. Для этого дважды щелкните левой кнопкой мыши на ListBox в форме и в появившемся блоке кода введите:

```

private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    if (listBox1.SelectedIndex != -1)
    {
        propertyGrid1.SelectedObject = listBox1.SelectedItem;
    }
}

```

Полный код приложения приведен в листинге 4.6.

Листинг 4.6. Код файла Form1.cs

```

using System;
using System.Collections.Generic;
using System.ComponentModel;

```

```

using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Xml;

namespace xml_reader
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            LoadEmployees();
        }

        private void LoadEmployees()
        {
            XmlDocument doc = new XmlDocument();
            doc.Load("xmltext.xml");

            foreach(XmlNode node in doc.DocumentElement)
            {
                string name = node.Attributes[0].Value;
                int age = int.Parse(node["Age"].InnerText);
                bool programmer = bool.Parse(node["Programmer"].InnerText);
                listBox1.Items.Add(new Employee(name, age, programmer));
            }
        }
    }
}

```

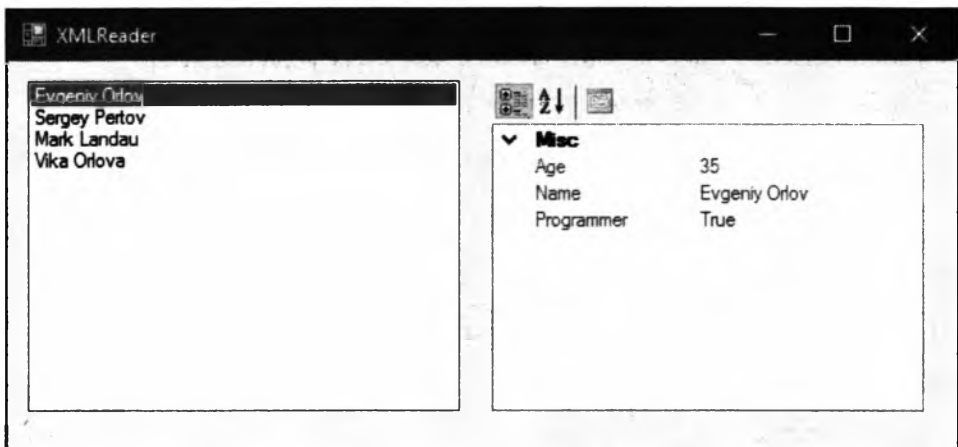


Рис. 4.9. Готовое приложение

```

    }
}
private void listBox1_SelectedIndexChanged(object
sender, EventArgs e)
{
    if (listBox1.SelectedIndex != -1)
    {
        propertyGrid1.SelectedObject = listBox1.SelectedItem;
    }
}
}
}

```

Работающая программа изображена на рис. 4.9.

4.6. Архивация файлов на C#

Для архивации файлов на C# обычно используется библиотека Ionic.Zip, скачать которую можно по адресу:

<http://dotnetzip.herokuapp.com/DNZHelp/Introduction/GettingStarted.htm>

Данная библиотека позволяет выполнять все операции с архивами в формате Zip - архивацию файлов, распаковку архивов, добавление файлов в архив и т.д. В этом разделе мы покажем, как создать приложение, архивирующее целую папку (архивация одного файла - малополезная операция).

Итак, приступим. Загрузите Ionic.Zip.dll в каталог приложения или в любой другой удобный вам. После этого создайте приложение Windows Forms и создайте форму, состоящую из одного текстового поля и двух кнопок. Первая кнопка будет вызывать диалог выбора папки, а вторая - диалог сохранения файла, в нем нужно будет ввести название архива.

После создания формы щелкните правой кнопкой мыши на элементе **Reference** в области **Solution Explorer** и выберите кнопку **Add Reference**. В появившемся окне нажмите кнопку **Browse** и укажите путь к библиотеке Ionic.Zip. Посмотрите на рис. 4.10 - на нем не только отображается созданная форма, но и путь к DLL.

После этого в файле Form1.cs нужно указать, что мы будем использовать подключенную библиотеку:

```
using Ionic.Zip;
```

Обработчик нажатия первой кнопки (Выберите папку) будет таким:

```
FolderBrowserDialog fo = new FolderBrowserDialog();
private void button1_Click(object sender, EventArgs e)

```



Рис. 4.10. В процессе создания приложения

```

{
    if (fo.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = fo.SelectedPath;
    }
}

```

Здесь мы открываем диалог выбора папки, а выбранный путь заносим в textBox1 - текстовое поле. Обработчик второй папки тоже довольно простой. Мы открываем диалог сохранения файла, и если пользователь не нажал **Отмена** (то есть DialogResult.OK), то начинается процесс архивации:

```

SaveFileDialog sfd = new SaveFileDialog();
sfd.Filter = "Zip files (*.zip)|*.zip";
if (textBox1.Text != "" && sfd.ShowDialog() == DialogResult.OK)
{
    ZipFile zf = new ZipFile(sfd.FileName);
    zf.AddDirectory(fo.SelectedPath);
    zf.Save();
    MessageBox.Show("Архивация прошла успешно.", "Выполнено");
}

```

Полный листинг Form1.cs для экономии бумаги и ваших денег приводить не буду - он очень прост. Программа в действии отображена на рис. 4.11.

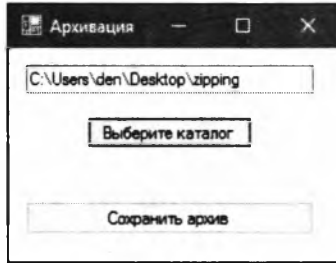


Рис. 4.11. Результат

4.7. Подсчет количества слов в файле

В заключение этой главы рассмотрим небольшой пример - программу, подсчитывающую количество слов в тексте. Исходный код этой программы приведен в листинге 4.7.

Листинг 4.7. Подсчет слов в тексте

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

namespace wordscount
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = "" ;
            string[] txt;
            StreamReader sr = new StreamReader("file.txt",
            System.Text.Encoding.Default);
            Console.OutputEncoding = Encoding.
            GetEncoding(866);

            while (sr.EndOfStream != true)
            {
                s = sr.ReadLine();
            }
            txt = s.Split(' ');
            Console.WriteLine("Количество слов:");
```

```
        Console.WriteLine(txt.Length);  
  
        sr.Close();  
        Console.ReadLine();  
    }  
}
```

Мы читаем слова из файла (словом считается все, что отделено пробелом) в массив, а потом выводим количество элементов (длину) в массиве. Алгоритм, может быть, и не очень хороший, но зато простой. Для больших файлов, возможно, его придется переделать, а именно - читать файл построчно, определять количество слов в строке и добавлять это количество к какой-то переменной, а потом выводить конечный результат.

Глава 5.

Объектно-ориентированное программирование



5.1. Основы ООП

Объектно-ориентированное программирование (ООП) — это особый подход к написанию программ. Чтобы понять, что такое ООП и зачем оно нужно, необходимо вспомнить некоторые факты из истории развития вычислительной техники. Первые программы вносились в компьютер с помощью переключателей на передней панели компьютера - в то время компьютеры занимали целые комнаты. Такой способ "написания" программы, сами понимаете, был не очень эффективным - ведь большую часть времени (несколько часов, иногда - целый рабочий день) занимало подключение кабелей и установка переключателей. А сами расчеты занимали считанные минуты. Вы только представьте, что делать, если один из программистов (такие компьютеры программировались, как правило, группами программистов) неправильно подключил кабель или установил переключатель? Да, приходилось все перепроверять - по сути, все начинать заново.

Позже появились перфокарты. Программа, то есть последовательность действий, которые должен был выполнить компьютер, наносилась на перфокарту. Пользователь вычислительной машины (так правильно было называть компьютеры в то время) писал программу, оператор "записывал" программу на перфокарту, которая передавалась оператору вычислительного отдела. Через определенное время оператор возвращал пользователю результат работы программы - рулон бумаги с результатами вычислений. Мониторов тогда не было, а все, что выводил компьютер, печаталось на бумаге. Понятно, если в расчетах была допущена ошибка (со стороны пользователя, компьютеры ведь не ошибаются - они делают с точностью то, что заложено программой), то вся цепочка действий (программист, оператор перфокарты, оператор вычислительной машины, проверка результатов) повторялась заново.

Следующий этап в программировании - это появление языка Ассемблера. Этот язык программирования позволял писать довольно длинные для того времени программы. Но Ассемблер - это язык программирования низкого уровня, все операции проводятся на уровне "железа". Если вы не знаете, то сейчас я вам поясню. Чтобы в С# выполнить простейшее действие, например сложение, достаточно записать '\$ = 2 + 2; '. На языке Ассемблера вам для выполнения этого же действия нужно было выполнить как минимум

три действия - загрузить в один из регистров первое число (команда MOV), загрузить в другой регистр второе число (опять команда MOV), выполнить сложение регистров командой ADD. Результат сложения будет помещен в третий регистр. Названия регистров я специально не указывал, поскольку они зависят от архитектуры процессора, а это еще один недостаток Ассемблера. Если вам нужно перенести программу на компьютер с другой архитектурой, вам нужно переписать программу с учетом особенностей целевой архитектуры.

Требования к программным продуктам и к срокам их разработки росли (чем быстрее будет написана программа, тем лучше), поэтому появились языки программирования высокого уровня. Язык высокого уровня позволяет писать программы, не задумываясь об архитектуре вашего процессора. Нет, это не означает, что на любом языке высокого уровня можно написать программу, которая в итоге станет работать на процессоре с любой архитектурой. Просто при написании программы знать архитектуру процессора совсем не обязательно. Вы пишете просто $A = B + C$ и не задумываетесь, в каком из регистров (или в какой ячейке оперативной памяти) сейчас хранятся значения, присвоенные переменным **B** и **C**. Вы также не задумываетесь, куда будет помещено значение переменной **A**. Вы просто знаете, что к нему можно обратиться по имени **A**. Первым языком высокого уровня стал FORTRAN (FORmula TRANslator).

Следующий шаг - это появление структурного программирования. Дело в том, что программы на языке высокого уровня очень быстро стали расти в размерах, что сделало их нечитабельными из-за отсутствия какой-нибудь четкой структуры самой программы. Структурное программирование подразумевает наличие структуры программы и программных блоков, а также отказ от инструкций безусловного перехода (GOTO, JMP).

После выделения структуры программы появилась необходимость в создании подпрограмм, которые существенно сокращали код программы. Намного проще один раз написать код вычисления какой-то формулы и оформить его в виде процедуры (функции) - затем для вычисления 10 результатов по этой формуле нужно будет 10 раз вызвать процедуру, а не повторять 10 раз один и тот же код. Новый класс программирования стал называться процедурным.

Со временем процедурное программирование постигла та же участь, что и структурное программирование - программы стали настолько большими, что их было неудобно читать. Нужен был новый подход к программированию. Таким стало объектно-ориентированное программирование (далее ООП).

ООП базируется на трех основных принципах - инкапсуляция, полиморфизм, наследование. Разберемся, что есть что.

С помощью инкапсуляции вы можете объединить воедино данные и обрабатывающий их код. Инкапсуляция защищает и код, и данные от вмешательства извне. Базовым понятием в ООП является класс. Грубо говоря, класс - это своеобразный тип переменной. Экземпляр класса (переменная типа класс) называется объектом. В свою очередь, объект - это совокупность данных (свойств) и функций (методов) для их обработки. Данные и методы обработки называются членами класса.

Получается, что объект - это результат инкапсуляции, поскольку он включает в себя и данные, и код их обработки. Чуть дальше вы поймете, как это работает, пока представьте, что объект - это эдакий рюкзак, собранный по принципу "все свое ношу с собой".

Члены класса могут быть открытыми или закрытыми. Открытые члены класса доступны для других частей программы, которые не являются частью объекта. Закрытые члены доступны только методам самого объекта.

Теперь поговорим о полиморфизме. Если вы программировали на языке Си (на обычном Си, не C++), то наверняка знакомы с функциями `abs()`, `fabs()`, `labs()`. Все они вычисляют абсолютное значение числа, но каждая из функций используется для своего типа данных. Если бы Си поддерживал полиморфизм, то можно было бы создать одну функцию `abs()`, но объявить ее трижды - для каждого типа данных, а компилятор бы уже сам выбирал нужный вариант функции, в зависимости от переданного ей типа данных. Данная практика называется перезагрузкой функций. Перезагрузка функций существенно облегчает труд программиста - вам нужно помнить в несколько раз меньше названий функций для написания программы.

Полиморфизм позволяет нам манипулировать с объектами путем создания стандартного интерфейса для схожих действий.

Осталось поговорить о наследовании. С помощью наследования один объект может приобретать свойства другого объекта. Заметьте, наследование - это не копирование объекта. При копировании создается точная копия объекта, а при наследовании эта копия дополняется уникальными свойствами (новыми членами). Наследование можно сравнить с рождением ребенка, когда новый человек наследует "свойства" своих родителей, но в то же время не является точной копией одного из родителей.

5.2. Классы и объекты

5.2.1. Члены класса

Итак, у вас уже есть представление о том, что такое ООП, класс и объект. Теперь настало время поговорить о классах подробнее. Класс можно считать шаблоном, по которому определяется форма объекта. В этом шаблоне указываются данные и код, который будет работать с этими данными - поля и методы. В С# используется спецификация класса для построения объектов, которые являются экземплярами класса (об этом уже говорилось в первом разделе этой главы).

Следовательно, класс является схематическим описанием способа построения объекта. При этом нужно помнить, что класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после создания объекта этого класса.

При определении класса нужно определить данные и код, который будет работать с этими данными. Есть и самые простые классы, которые содержат только код (или только данные). Однако большая часть настоящих классов содержит и то, и другое.

Данные содержатся в членах данных, а код - в функциях-членах (это не только методы!). В языке С# предусмотрено несколько видов членов данных и функций членов. Все члены могут быть публичными (`public`) или приватными (`private`). Данные-члены могут быть статическими (`static`). Член класса является членом экземпляра, если только он не объявлен явно как `static`. Если вы не знаете, что такое `private`, `public` и `static`, об этом мы поговорим позже, когда придет время.

Рассмотрим члены данных:

- Поля - любые переменные, связанные с классом.
- Константы - ассоциируются с классом, как и поля. Константа объявляется с помощью ключевого слова `const`. Если константа объявлена публичной (`public`), то она становится доступной извне класса.
- События - члены класса, позволяющие объектам уведомлять вызывающий код о том, что произошло какое-то событие, например, изменилось свойство класса или произошло взаимодействие с пользователем.

Функции-члены - это члены, предназначенные для манипулирования данными класса. В С# поддерживаются следующие функции-члены:

- Методы (method) - функции, связанные с определенным классом. Как и члены данных, они являются членами экземпляра. Они могут быть объявлены статическими с помощью модификатора **static**.
- Свойства (property) - представляют собой наборы функций, которые могут быть доступны так же, как и общедоступные поля класса. В C# используется специальный синтаксис для реализации чтения (Get) и записи (Set) свойств для классов, поэтому писать собственные методы с именами, начинающимися на Set и Get, не понадобится.
- Конструкторы (constructor) - функции, вызываемые автоматически при инициализации объекта. Имена конструкторов совпадают с именами классов, которым они принадлежат. Конструкторы не имеют типа возврата и обычно используются для инициализации полей.
- Финализаторы (finalizer) - вызываются, когда среда CLR решает, что объект больше не нужен. В других языках программирования финализаторы называются деструкторами. Имеют то же имя, что и класс, но с предшествующим символом тильды.
- Операторы (operator) - используются для перезагрузки простейших действий вроде + и -. В C# можно указать, как эти простейшие операции будут работать с пользовательскими классами. Такое явление называется перегрузка операции и поддерживается не только в Си, но и в других языках программирования (C++, Java, PHP и т.д.).
- Индексаторы (indexer) - используются для индексирования объектов.

5.2.2. Ключевое слово class

Для объявления класса используется ключевое слово **class**. Общая форма этого оператора выглядит так:

```
class имя_класса {
    // Объявление переменных экземпляра.
    access type var1;
    access type var2;
    //...
    access type varN;

    // Объявление методов.
    access return_type method1 ([args]) {
        // тело метода
    }
    access return_type method2 ([args]) {
        // тело метода
    }
}
```

```

    }
    //...
    access return_type methodN([args]) {
        // тело метода
    }
}

```

Сначала объявляются переменные экземпляра var1...varN. Спецификатор **access** задает тип доступа, например, **public**. После этого модификатора указывается тип переменной, а потом ее имя.

При объявлении метода указываются спецификатор доступа (access), тип возвращаемого значения (return_type) и название метода. В скобках - обязательный список параметров (args).

Спецификатор доступа определяет порядок доступа к члену класса. Член класса может быть приватным (private) и публичным (public). Приватные члены доступны только в пределах класса, а публичные доступны из других классов. Указывать спецификатор доступа необязательно. Если он не указан, то член считается приватным.

Теперь пришло время разработать наш первый класс. Вместо простого класса, выводящего что-то вроде Hello, world!, сейчас мы разработаем класс, выводящий информацию о системе (лист. 5.1).

Листинг 5.1. Пример класса, выводящего информацию о системе

```

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

namespace ConsoleApplication1
{
    class Program
    {
        class SysInfo
        {
            public string win, net, cpu;
            public string hostname, username;

            public SysInfo()
            {
                net = Environment.Version.ToString();
                win = Environment.OSVersion.ToString();
            }
        }
    }
}

```

```

        cpu = Environment.ProcessorCount.ToString();
        hostname = Environment.MachineName.ToString();
        username = Environment.UserName.ToString();
    }
}

static void Main(string[] args)
{
    string p;

    SysInfo info = new SysInfo();

    if (args.Length > 0) p = args[0];
    else p = "null";

    switch (p) {
        case "cpu":
            Console.WriteLine("CPU count: {0}", info.cpu);
            break;
        case "win":
            Console.WriteLine("Windows Version: {0}", info.win);
            break;
        case "net":
            Console.WriteLine(".NET Version: {0}", info.net);
            break;
        case "host":
            Console.WriteLine("Hostname: {0}", info.hostname);
            break;
        case "user":
            Console.WriteLine("Username: {0}", info.username);
            break;
        default:
            Console.WriteLine("Usage: sysinfo <cpu|win|net|host|user>");
            break;
    }
}
}

```

Теперь разберемся, что есть что. Сначала в методе `Main()` определяются две переменные. Первая - это строковая переменная `p`, мы ее используем для облегчения работы с параметрами программы. В принципе, можно было бы обойтись и без нее. Переменная `info` - это экземпляр класса `SysInfo()`.

Посмотрим на сам класс. Класс содержит следующие поля:

```

public string win, net, cpu;
public string hostname, username;

```

Конструкто

Далее мы а



На практике это означает то, что помимо определяемых вами методов и полей, созданные вами классы поддерживают множество общедоступных и защищенных методов-членов, определенных в классе `System.Object` (табл. 5.1).

Таблица 5.1. Методы класса `System.Object()`

Метод	Описание
<code>ToString()</code>	Возвращает символьную строку, содержащую описание объекта, для которого он вызывается. С этим методом мы уже знакомы из предыдущего раздела. Также метод <code>ToString()</code> автоматически вызывается при выводе содержимого объекта с помощью метода <code>Write()/WriteLine()</code> . В своем классе вы можете переопределить этот метод, чтобы созданные вами объекты могли быть корректно преобразованы в строку.
<code>GetHashCode()</code>	Используется при помещении объекта в структуру данных - карту (<code>map</code>), которая также называется хеш-таблицей. Используется классами, которые манипулируют картами. Если вы желаете использовать свой класс в таком контексте, то вы должны переопределить метод <code>GetHashCode()</code> . Однако это нужно довольно редко, если вам когда-то и понадобится использовать этот метод, подробности вы сможете узнать в документации по .NET, поскольку существуют строгие требования перезагрузки этого метода.
<code>Equals()</code>	Метод <code>Equals(object)</code> определяет, ссылается ли вызывающий объект на тот же самый объект, что и объект, указываемый в качестве аргумента этого метода. То есть он проверяет, являются ли оба объекта одинаковыми. Метод возвращает <code>true</code> , если сравниваемые объекты одинаковые, в противном случае - <code>false</code> .
<code>Finalize()</code>	Является деструктором и вызывается при очистке ресурсов, занятых объектом. По умолчанию этот метод ничего не делает. Переопределять этот метод нужно, если ваш объект владеет неуправляемыми ресурсами, которые нужно освободить при его уничтожении. Во всех остальных случаях переопределять этот метод не нужно.
<code>GetType()</code>	Возвращает экземпляр класса, унаследованный от <code>System.Type</code> . Может предоставить большой объем информации о классе, в том числе базовый тип, методы, поля и т.д.
<code>Clone()</code>	Создает копию объекта и возвращает ссылку на эту копию.

5.2.4. Конструкторы

С конструктором мы уже знакомы, поскольку недавно реализовали конструктор для класса `SysInfo`. Основная задача конструктора - инициализировать объект при его создании. Наш конструктор заполнял поля информацией, полученной от класса `Environment`.

С точки зрения синтаксиса определение конструктора подобно определению метода, но у конструкторов нет возвращаемого типа. Общая форма определения конструкторов приведена ниже:

```
access имя_класса(args) {  
    // тело конструктора  
}
```

Спецификатор доступа (`access`) обычно указывается **public**, поскольку конструкторы зачастую вызываются в классе, а вот список параметров **args** может быть как пустым, так и состоящим из одного или нескольких параметров. Имя конструктора, как уже было отмечено, должно совпадать с именем класса.

Каждый класс в C# по умолчанию оснащается конструктором, который вы при желании можете переопределить, что мы и сделали в нашем классе (см. лист. 5.1).

Конструктор также может принимать один или несколько параметров. В конструктор параметры передаются таким же образом, как и в метод. Для этого достаточно объявить их в скобках после имени конструктора.

В листинге 5.2 приводится пример определения конструктора класса с параметрами.

Листинг 5.2. Конструктор класса с параметрами

```
class Human  
{  
    public string Name;  
    public byte Age;  
    // Устанавливаем параметры  
    public Human(string n, byte a)  
    {  
        Name = n;  
        Age = a;  
    }  
    public void GetInfo()  
    {  
        Console.WriteLine("Name: {0}\nAge: {1}", Name, Age);  
    }  
}
```

}

В нашем классе есть два поля, конструктор и метод `GetInfo()`. Параметры конструктору передаются при создании объекта:

```
Human John = new Human("John", 33);
John.GetInfo();
```

5.2.5. Деструкторы

Обратите внимание, как создается новый объект. Перед указанием имени вы указываете оператор `new`, который выделяет оперативную память для создаваемого объекта.

Понятное дело, что оперативная память - не резиновая, поэтому свободная память рано или поздно закончится.

Именно поэтому одной из главных функций любой схемы динамического распределения памяти является освобождение памяти от неиспользуемых объектов, чтобы сделать ее доступной для последующего выделения (allocation).

В некоторых языках (например, в C++) освобождение выделенной ранее памяти осуществляется вручную. Например, в C++ для этого используется оператор `delete`. Но в C# (и не только в C#, в PHP тоже) используется процесс, названный сборкой мусора.

Благодаря автоматической “сборке мусора” в C# память освобождается от лишних объектов. Данный процесс происходит незаметно и без всякого вмешательства со стороны программиста. Если ссылки на объект отсутствуют, то такой объект считается ненужным, и занимаемая им память в итоге освобождается. В итоге освобожденная память может использоваться для других объектов.

Процесс “сборки мусора” происходит полностью в автоматическом режиме и нельзя заранее знать или предположить, когда он произойдет.

В C# программист может определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта. Такой метод называется деструктором и гарантирует четкое окончание времени жизни объекта.

Синтаксис определения деструктора такой же, как и конструктора, но перед именем класса указывается `~`:

```
access ~имя_класса(args) {
// тело деструктора
}
```

Пример деструктора приведен в листинге 5.3.

Листинг 5.3. Пример определения деструктора

```
class Human
{
    public string Name;
    public byte Age;

    // Устанавливаем параметры
    public Human(string n, byte a)
    {
        Name = n;
        Age = a;
    }

    public ~Human()
    {
        Console.WriteLine("Object was destroyed");
    }

    public void GetInfo()
    {
        Console.WriteLine("Name: {0}\nAge: {1}", Name, Age);
    }
}
```

Обычно деструкторы не нужны, поскольку все используемые объектом ресурсы автоматически освобождаются при его уничтожении. Но при желании вы можете освободить ресурсы явно, например, закрыть используемые файлы и сетевые соединения.

5.2.6. Обращаемся сами к себе. Служебное слово `this`

В C# есть ключевое слово **this**, обеспечивающее доступ к текущему экземпляру класса. Данное ключевое слово используется для разрешения неоднозначности, когда один из параметров назван так же, как поле данных. Конечно, чтобы такой неоднозначности не было, лучше придерживаться правил именования переменных, параметров, полей и т.д.

Лучше всего продемонстрировать применение ключевого слова **this** на примере, приведенном в листинге 5.4.

Листинг 5.4. Использование ключевого слова `this`

```
class Human
```

```

{
    public string Name;
    public byte Age;

    // Устанавливаем параметры
    public Human(string Name, byte Age)
    {
        // Что и чему присваивается?
        // Name = Name;
        // Age = Age;
        this.Name = Name;
        this.Age = Age;
    }
}

```

Без использования служебного слова **this** было бы непонятно, что и чему присваивается - то ли параметру присваивается значение поля, то ли наоборот:

```

Name = Name;
Age = Age;

```

При использовании ключевого слова **this** все становится на свои места - полям присваиваются значения параметров:

```

this.Name = Name;
this.Age = Age;

```

Использование **this** позволяет использовать более понятные имена параметров методов. Ранее мы использовали имена параметров **n** и **a**, а теперь в конструкторе мы можем смело указывать имена **Name** и **Age** и не беспокоиться, что компилятор не поймет, что мы имели в виду.

5.2.7. Доступ к членам класса

Что дает нам инкапсуляция? Прежде всего, она связывает данные с кодом. Но это далеко не все. Благодаря ей, класс предоставляет средства для управления доступом к его членам. Именно об управлении доступом мы сейчас и поговорим.

По существу, есть два типа членов класса - публичные (**public**) и приватные (**private**). Также их еще называют открытыми и закрытыми. Доступ к открытому члену возможен из кода, существующего за пределами класса. А вот доступ к приватному члену возможен только методам, которые опреде-

лены в самом классе. Приватные члены позволяют организовать управление доступом.

Кроме известных спецификаторов (модификаторов) доступа **public** и **private**, в С# поддерживаются еще два модификатора - **protected** и **internal**.

Как уже было отмечено, доступ к члену с модификатором **private** невозможен из других классов. Если модификатор доступа не указан, то считается, что член класса является приватным.

Спецификатор **protected** означает член класса, доступ к которому открыт в пределах иерархии классов. Спецификатор **internal** используется в основном для сборок, и начинающим программистам он не нужен.

Примеры:

```
public int a;      // Открытый член
private int b;    // Закрытый член
int c;            // Закрытый член по умолчанию
```

При разработке собственных классов вы должны придерживаться следующих правил ограничения доступа:

- Члены, которые вы планируете использовать только в классе, должны быть закрытыми (**private**).
- Если изменение члена приведет к последствиям, которые распространяются за пределы области действия самого члена, этот член должен быть закрытым.
- Методы, которые используются для получения и установки закрытых членов (полей) данных, должны быть публичными (**public**).
- Если доступ к члену допускается из производных классов, то такой член можно объявить как защищенный (**protected**).
- Если какой-то член может нанести вред объекту, если он будет использоваться неправильно, он должен быть закрытым.
- Если нет никаких оснований, чтобы та или иная переменная класса была закрытой, ее можно сделать публичной.

5.2.8. Модификаторы параметров

Как уже было отмечено, метод может не содержать параметров вообще, а может содержать один или несколько параметров. У каждого параметра может быть свой модификатор (см. табл. 5.2).

Таблица 5.2. Модификаторы параметров методов

Модификатор параметра	Описание
(отсутствует)	Если модификатор параметра не задан, считается, что он должен передаваться по значению, т.е. вызываемый метод должен получать копию исходных данных
out	Выходные параметры должны присваиваться вызываемым методом (и, следовательно, передаваться по ссылке). Если параметрам с модификатором out в вызываемом методе значения не присвоены, компилятор сообщит об ошибке
ref	Это значение первоначально присваивается вызывающим кодом и при желании может повторно присваиваться в вызываемом методе (поскольку данные также передаются по ссылке). Если параметрам с модификатором ref в вызываемом методе значения не присвоены, компилятор не будет генерировать сообщение об ошибке.
params	Этот модификатор позволяет передавать в виде одного логического параметра переменное количество аргументов. В каждом методе может присутствовать только один модификатор params , и он должен обязательно указываться последним в списке параметров. В реальности необходимость в использовании модификатора params возникает очень редко, но он применяется во многих методах внутри библиотек базовых классов

Наиболее часто используется модификатор **ref**, позволяющий изменять значение параметра внутри метода и передавать его по ссылке в вызывающий метод. Лучше всего продемонстрировать работу этого параметра на примере (лист. 5.5).

Листинг 5.5. Использование модификатора ref

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

namespace ConsoleApplication1
{
    class Program
    {
        // Метод, изменяющий свой аргумент
```

```

static void changeNum(ref int n)
{
    n = 100;
}

static void Main()
{
    int x = 0;
    Console.WriteLine("Value of x before calling changeNum: {0}", x);
    changeNum(ref x);
    Console.WriteLine("Value of x after calling changeNum: {0}", x);

    Console.ReadLine();
}
}

```

Вывод будет таким:

```

Value of x before calling changeNum: 0
Value of x after calling changeNum: 100

```

Если бы параметр был объявлен без модификатора **ref**, метод не смог бы изменить значение переменной **x**. Также обратите внимание, как переменная **x** передается в сам метод.

Модификатор параметра **out** похож на **ref**, за одним исключением: он служит только для передачи значения за пределы метода. Поэтому переменной, используемой в качестве параметра **out**, не нужно присваивать какое-то значение. Более того, в методе параметр **out** считается неинициализированным, т.е. предполагается, что у него отсутствует первоначальное значение.

Рассмотрим пример использования модификатора **out** (лист. 5.6).

Листинг 5.6. Пример использования модификатора out

```

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

namespace ConsoleApplication1
{
    class Program
    {
        static int test(double a, out double b, out double c)

```

```

    {
        int i = (int)a;
        b = a * a;
        c = a * a * a;

        return i;
    }

    static void Main()
    {
        int i;
        double c, b, a = 5.5;

        i = test(a, out b, out c);

        Console.WriteLine("Original value: {0}\n
Int : {1}\n
a ^ 2: {2}\n
a ^ 3: {3}\", a, i, b, c);

        Console.ReadLine();
    }
}

```

Вывод программы будет таким:

```

Original value: 5.5
Int : 5
a ^ 2 : 30,25
a ^ 3 : 166,375

```

Использование модификатора **out** позволяет методу возвращать сразу три числа - два через параметры с модификатором **out** и одно - через **return**.

Модификатор **params** позволяет передавать методу переменное количество аргументов одного типа в виде единственного логического параметра. Рассмотрим пример использования модификатора **params** на примере передачи массива в качестве параметра метода. Наш метод `minElement` будет искать минимальный элемент в массиве, который передан с модификатором **params**.

Листинг 5.7. Использование модификатора **params**

```

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

```

```

using System.Text;
namespace ConsoleApplication1
{
    class Program
    {
        static double minElement(params double[] arr)
        {
            double min;
            // Array is empty?
            if (arr.Length == 0)
            {
                Console.WriteLine("Empty array!");
                return Double.NegativeInfinity;
            }
            else
            {
                // Only 1 element in array
                if (arr.Length == 1)
                {
                    min = arr[0];
                    return min;
                }

                min = arr[0];
                // Searching for min
                for (int j = 1; j < arr.Length; j++)
                    if (arr[j] < min)
                        min = arr[j];
            }
            return min;
        }
        static void Main()
        {
            double result = 0;
            double[] arr1 = { 5.5, 7.6, 3.3, -11.7};

            result = minElement(arr1);
            Console.WriteLine("Min: {0}", result);

            Console.ReadLine();
        }
    }
}

```

Результатом будет число -11.7, как несложно было догадаться.

5.2.9. Необязательные параметры

Начиная с версии 4.0 языка C#, появились необязательные аргументы, позволяющие определить значение по умолчанию для параметра метода. Данное значение будет использоваться по умолчанию в том случае, если для параметра не указан соответствующий аргумент при вызове метода. Такие аргументы довольно давно существуют в других языках программирования, но в C# появились только в версии 4.0.

Основная цель, которую преследовали разработчики C#, - необходимость в упрощении взаимодействия с объектами COM, многие из которых были написаны давно и рассчитаны на использование необязательных параметров.

Определяются данные параметры так:

```
public int Prod3(int a, int b = 1, int c = 1)
{
    return a * b * c;
}
```

Нужно отметить, что все необязательные аргументы должны непременно указываться справа от обязательных. Кроме обычных методов, необязательные параметры можно применять в конструкторах, индексаторах и делегатах.

5.2.10. Именованные аргументы

Также в версии .NET 4.0 появилась поддержка так называемых именованных аргументов (named arguments). Обычно аргументы передаются в том порядке, который должен совпадать с порядком указания аргументов при объявлении метода. Вспомним наш конструктор класса Human:

```
public Human(string Name, byte Age)
{
    this.Name = Name;
    this.Age = Age;
}
```

Вызывать его можно было только так:

```
Human Den = new Human("Den", 32);
```

Вы не можете вызвать его так:

```
Human Den = new Human(32, "Den");
```

Именованные аргументы позволяют указывать параметры в произвольном порядке - в том, в котором вам хочется. К тому же вы будете точно знать,

что и какому параметру вы присваиваете, поскольку при передаче аргументов указываются имена параметров. Для указания аргументов по имени используется форма:

имя_параметра : значение

Пример:

```
Human Den = new Human(Age: 32, Name: "Den");
```

При этом вносить изменения в описание самого метода не нужно. Просто укажите компилятору, какому параметру какое значение вы передаете.

5.2.11. Ключевое слово **static**

Бывают ситуации, когда нужно определить член класса, который будет использоваться независимо от всех остальных объектов этого класса. Доступ к члену класса будет осуществляться посредством объекта этого класса, но в то же время можно создать член класса для самостоятельного применения без ссылки на конкретный экземпляр объекта. Такие члены называются статическими и для их определения используется ключевое слово **static**.

Ключевое слово **static** можно использовать как для переменных, так и для методов. Переменные, объявляемые как **static**, по существу, являются глобальными.

Самый часто используемый пример статического члена - метод **Main()**, который объявляется таковым потому, что он должен вызываться операционной системой в самом начале выполняемой программы.

Чтобы воспользоваться членом типа **static** за пределами класса, достаточно указать имя этого класса с оператором-точкой. Но создавать объект для этого не нужно. Рассмотрим наш класс **Human**, в который мы добавим статический метод **Hello()**:

Листинг 5.8. Статический метод

```
class Human
{
    public string Name;
    public byte Age;

    // Устанавливаем параметры
    public Human(string n, byte a)
    {
        Name = n;
        Age = a;
    }
}
```

```

    }

    public ~Human()
    {
        Console.WriteLine("Object was destroyed");
    }

    public void GetInfo()
    {
        Console.WriteLine("Name: {0}\nAge: {1}", Name, Age);
    }

    static void Hello()
    {
        Console.WriteLine("Hello, world!");
    }
}

```

Теперь посмотрим, как использовать данный метод:

```
Human.Hello();
```

Заметьте, вызвать таким образом метод `GetInfo()` не получится, поскольку этот метод не является статическим и нужно сначала создать объект:

```
Human Den = new Human("Den", 32);
Den.GetInfo();
```

Логика в следующем: для работы `GetInfo()` нужно, чтобы были инициализированы поля класса. Следовательно, его нельзя объявлять как статический. А вот метод `Hello()` независим от остального класса - он всегда будет выводить одну и ту же строку, поэтому его можно объявить статическим.

Конструктор можно также объявить как **static**. Статический конструктор обычно используется для инициализации компонентов, применяемых ко всему классу, а не к отдельному экземпляру объекта этого класса. Поэтому члены класса инициализируются статическим конструктором до создания каких-либо объектов этого класса. Причем в одном классе может быть как статический, так и обычный конструктор. Пример объявления двух конструкторов приведен в листинге 5.9.

Листинг 5.9. Статический и публичный конструкторы

```

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

```

```
using System.Text;
namespace ConsoleApplication1
{
    class SampleClass
    {
        public static int x;
        public int y;

        // Статический конструктор
        static SampleClass()
        {
            x = 1;
        }

        // Обычный конструктор
        public SampleClass()
        {
            y = 12;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Access to x: " + SampleClass.x);

            SampleClass obj = new SampleClass();
            Console.WriteLine("Access to y: " + obj.y);

        }
    }
}
```

Более того, статическим можно объявить не только конструктор класса, но и весь класс. Статические классы обладают двумя основными свойствами:

- Объекты статического класса создать нельзя.
- Статический класс должен содержать только статические члены.

Объявляется статический класс так:

```
static class имя класса { // ...
```

5.2.12. Индексаторы

Любой программист хорошо знаком с процессом доступа к отдельным элементам массива с помощью квадратных скобок []. В языке С# имеется возможность создавать специальные классы и структуры, которые можно

индексировать подобно обычному массиву посредством определения индексатора. Другими словами, в С# можно обращаться к классу как к массиву. Это позволяет создавать специальные типы коллекций.

Индексаторы бывают одномерными и многомерными. Последние используются редко, поэтому остановимся только на одномерных. Синтаксис определения одномерного индексатора такой:

```
тип_элемента this[int индекс] {  
    // Аксессор для получения данных,  
    get {  
        // Возвращает значение, которое определяет индекс.  
    }  
    // Аксессор для установки данных,  
    set {  
        // Устанавливает значение, которое определяет индекс.  
    }  
}
```

Рассмотрим, что есть что:

- **тип_элемента** - конкретный тип элемента индексатора. У каждого элемента, доступного с помощью индексатора, должен быть определенный тип_элемента. Этот тип соответствует типу элемента массива.
- **индекс** - параметр "индекс" получает конкретный индекс элемента, к которому осуществляется доступ. Этот параметр не обязательно должен быть типа **int**, но поскольку индексаторы часто применяются для индексирования массивов, скорее всего, вы будете использовать тип **int**.

В теле индексатора определяются два аксессора (от англ. *accessor*): **get** и **set**. Первый используется для получения значения, второй - для его установки. Аксессор похож на метод, но в нем не указываются тип возвращаемого значения или параметры.

Аксессоры вызываются автоматически при использовании индексатора, и оба получают индекс в качестве параметра. Аксессор **set** также получает неявный параметр **value**, который содержит значение, присваиваемое по указанному индексу.

Все это звучит очень сложно, пока не виден код. Листинг 5.10 демонстрирует, как изящно можно превратить класс в массив и работать с ним как с обычным массивом.

Листинг 5.10. Пример использования индексатора

```

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

namespace ConsoleApplication1
{
    class MyArr
    {
        int[] arr;
        public int Length;

        public MyArr(int Size)
        {
            arr = new int[Size];
            Length = Size;
        }

        // Простой индексатор
        public int this[int index]
        {
            set           // Устанавливаем массив элементов
            {
                arr[index] = value;
            }
            get
            {
                return arr[index];
            }
        }
    }

    class Program
    {
        static void Main()
        {
            MyArr arr1 = new MyArr(10);

            // Инициализируем каждый индекс экземпляра класса arr1
            for (int i = 0; i < arr1.Length; i++)
            {
                arr1[i] = i * 2;
                Console.Write("{0} ", arr1[i]);
            }
        }
    }
}

```

```
    }
}
```

В цикле **for** работа с переменной `agg1` осуществляется как с массивом. Однако `agg1` - это объект типа `MyAgg`, а не массив.

5.2.13. Свойства

В C# есть еще одна диковинка - свойства. В других языках программирования, поддерживающих ООП, свойством называется член класса, содержащий данные. Просто есть свойства и методы. Свойства - это переменные, а методы - это функции. Грубо говоря, конечно. Но в C# все немного иначе. Свойство сочетает в себе данные и методы доступа к ним.

Свойства очень похожи на индексаторы. В частности, свойство состоит из имени и аксессоров **get** и **set**. Аксессоры служат для получения и установки значения переменной. Разница в том, что имя свойства может быть использовано в выражениях и операторах присваивания аналогично имени обычной переменной, но в действительности при обращении к свойству автоматически вызываются методы **get** и **set**.

Синтаксис описания свойства выглядит так:

```
тип имя {
    get
    {
        // код аксессора для чтения из поля
    }

    set
    {
        // код аксессора для записи в поле
    }
}
```

Здесь тип означает конкретный тип, например, `char`. Как видите, свойства не определяют место в памяти для хранения полей, а лишь управляют доступом к полям. Это означает, что само свойство не предоставляет поле, и поэтому поле должно быть определено независимо от свойства.

5.3. Перегрузка функций членов класса

5.3.1. Перегрузка методов

В C# допускается перегрузка методов. С помощью перегрузки методов можно или переопределить ранее определенный метод (например, метод

ToString(), который неявно есть у каждого класса), чтобы он соответствовал вашему классу, или же определить несколько методов, которые будут определяться по-разному. Например, вы можете создать методы с одинаковым названием, но разными типами возврата, а компилятор будет выбирать нужный метод, в зависимости от производимых вычислений.

Для перегрузки метода достаточно объявить разные его варианты, а об остальном позаботится компилятор. Но при этом необходимо соблюсти следующее важное условие: тип или число параметров у каждого метода должны быть разными.

Недостаточно, чтобы методы отличались только типами возвращаемых значений. Также они должны отличаться типами и/или числом своих параметров. Если есть два метода с одинаковым списком параметров (совпадают количество и типы), но с разным типом возвращаемого значения, компилятор просто не сможет выбрать, какой из них использовать. Хотя о типе возвращаемого значения будет сказано чуть позже.

Пример перегруженного метода Info() приведен в листинге 5.11.

Листинг 5.11. Пример перегруженного метода

```
class Car
{
    // Перегружаем метод Info()
    public void Info()
    {
        Console.WriteLine("No brand selected\n");
    }

    public void Info(string Brand)
    {
        Console.WriteLine("Brand: {0}\nNo model selected", Brand);
    }

    public void Info(string Brand, string Model)
    {
        Console.WriteLine("Brand: {0} Model: {1}", Brand, Model);
    }
}
```

В данном примере мы сначала объявляем метод Info(), а затем перегружаем его два раза. Когда компилятор принимает решение о перегрузке метода, то также учитываются модификаторы параметров **ref** и **out**.

Перегрузка методов поддерживает полиморфизм. В C# соблюдается главный принцип полиморфизма: один интерфейс - множество методов. В языках, где не поддерживается перегрузка методов, каждому методу должно быть присвоено уникальное имя. Типичный пример - язык Си, в котором есть функции `abs()`, `fabs()`, `labs()`. Все они вычисляют абсолютное значение числа, но каждая из функций используется для своего типа данных.

В C# есть понятие сигнатуры. Сигнатура обозначает имя и список параметров. Поэтому в классе не должно быть двух методов не с одинаковыми именами, а с одинаковыми сигнатурами. В сигнатуру не входит тип возвращаемого значения, поскольку он не учитывается, когда компилятор C# принимает решение о перегрузке метода. В сигнатуру не входит также модификатор `params`.

5.3.2. Перегрузка конструкторов

Язык C# позволяет перезагружать конструкторы. Это позволяет создавать объекты самыми разными способами. Пример перегрузки конструктора класса `Car` приведен в листинге 5.12.

Листинг 5.12. Перегрузка конструкторов

```
class Car
{
    // Перегружаем конструктор
    public void Car()
    {
        Console.WriteLine("No brand selected\n");
    }

    public void Car(string Brand)
    {
        Console.WriteLine("Brand: {0}\nNo model selected", Brand);
    }

    public void Car(string Brand, string Model)
    {
        Console.WriteLine("Brand: {0} Model: {1}", Brand, Model);
    }
}
```

Самая распространенная причина перегрузки конструкторов - необходимость инициализации объекта разными способами. При работе с перезагружаемыми конструкторами есть возможность вызвать другой конструктор. Делается это с помощью ключевого слова `this`, пример:

```
public Info() : this("None", "None")
{
}
public Info(Car obj)
    : this(obj.Brand, obj.Model)
{
}
public Info(string Brand, string Model)
{
    this.Brand = Brand;
    this.Model = Model;
}
```

5.3.3. Перегрузка операторов

В С# перегружать можно не только методы и конструкторы, но и операторы. Например, вы можете перегрузить оператор + и определить, как будет выполняться операция сложения нескольких ваших объектов.

По умолчанию перегруженными являются все операторы, поскольку они позволяют работать с данными разных типов. Например, тот же + можно использовать для сложения чисел и строк:

```
int a = 1, b = 2;
int c = a + b;
string n = "user";
string s = "hello" + " " + user;
```

Сначала, как видите, оператор + используется для сложения двух целых чисел, затем - для конкатенации трех строк.

Не все операторы могут быть перегружены. Таблица 5.3 содержит сведения о том, какие операторы могут быть перегружены, а какие - нет.

Таблица 5.3. Возможность перегрузки операторов в С#

Оператор(ы)	Возможность перегрузки
+, -, !, ++, --, true, false	Могут быть перегружены
+, -, *, /, %, &, , ^, <<, >>	Могут быть перегружены
==, !=, <, >, <=, >=	Могут быть перегружены, но нужно перегрузить все эти операторы
[]	Не может быть перегружен. Но подобный функционал предлагают индексаторы
()	Не может быть перегружен. Подобный функционал обеспечивают специальные методы преобразования
+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=	Не перезагружаются

Синтаксис перегрузки оператора отличается для унарных и бинарных операторов:

```
// Для унарных операторов
public static возвращаемый_тип operator op(тип_параметра
операнд)
{
// тело
}
// Для бинарных операторов
public static возвращаемый_тип operator op(тип_параметра1
операнд1,
тип_параметра2 операнд2)
{
// тело
}
```

Разберемся, что есть что. Возвращаемый тип обозначает конкретный тип значения, которое будет возвращаться операцией. Обычно возвращаемый тип такой же, как и у класса, для которого перегружают оператор. Вместо `op` подставляется оператор, который нужно перегрузить, например, `+` или `-`.

Рассмотрим пример перегрузки операторов `+` и `-` для произвольного класса (лист. 5.13).

Листинг 5.13. Перегрузка операторов + и -

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

namespace ConsoleApplication1
{
    class SampleClass
    {
        // Переменные
        public int x, y;

        public SampleClass(int x = 0, int y = 0)
        {
            this.x = x;
            this.y = y;
        }

        // Перегрузка +
        public static SampleClass operator +(SampleClass o1, SampleClass o2)
        {
```

```

        SampleClass res = new SampleClass();
        res.x = o1.x + o2.x;
        res.y = o1.y + o2.y;
        return res;
    }
    // Перепрызка -
    public static SampleClass operator -(SampleClass o1, SampleClass o2)
    {
        SampleClass res = new SampleClass();
        res.x = o1.x - o2.x;
        res.y = o1.y - o2.y;
        return res;
    }
}
class Program
{
    static void Main(string[] args)
    {
        SampleClass obj1 = new SampleClass(100, 64);
        SampleClass obj2 = new SampleClass(-74, 28);
        Console.WriteLine("First object: " +
            obj1.x + " " + obj1.y);
        Console.WriteLine("Second object: " +
            obj2.x + " " + obj2.y);

        SampleClass obj3 = obj1 + obj2;
        Console.WriteLine("obj1 + obj2: "
            + obj3.x + " " + obj3.y);

        obj3 = obj1 - obj2;
        Console.WriteLine("obj1 - obj2: "
            + obj3.x + " " + obj3.y);
    }
}

```

5.4. Наследование и полиморфизм

5.4.1. Введение в наследование

Наследование - один из трех основных принципов объектно-ориентированного программирования. Наследование допускает создание иерархических классификаций. Благодаря ему можно создать общий класс, в котором определены характерные особенности, которые свойственны множеству связанных элементов. От этого класса могут создаваться производные классы (могут наследовать базовый), добавляя в общий класс свои особенности.

Класс, который наследуется, называется базовым, а класс, который наследует, называется производным. В других языках программирования может использоваться другая терминология, например, базовый класс может называться родительским, а производный - дочерним.

Для объявления производного класса используется следующий синтаксис:

```
class Имя_производного_класса : Имя_базового_класса
{
...
}
```

Рассмотрим пример наследования классов (листинг 5.14). Мы определим два класса - Parent (базовый) и Child (производный от базового). В классе Parent объявлены два поля *x* и *y*. В классе Child объявлено только поле *z*.

Листинг 5.14. Наследование классов

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

```
namespace ParentChild
{
    class Parent
    {
        public int x, y;

        public Parent()
        {
            x = 10;
            y = 20;
        }
    }
    class Child : Parent
    {
        public int z;
        public Child()
        {
            y = 30;
            z = 50;
        }
    }

    class Program
    {
```

```

static void Main(string[] args)
{
    Parent p = new Parent();
    Child c = new Child();

    Console.WriteLine("Parent x, y: {0} {1}", p.x, p.y);
    Console.WriteLine("Child x, y, z: {0} {1} {2}", c.x, c.y, c.z);
    Console.ReadLine();
}
}

```

Класс `Child` наследует все члены базового класса, следовательно, он унаследует поля `x` и `y`. Посмотрите, как конструктор класса `Child` изменяет значение поля `y`, которое не было у него определено. Далее в программе мы обращаемся к полям `x` и `y` класса `Child`, которые не были изначально определены.

5.4.2. Защищенный доступ

В листинге 5.14 все члены классов были публичными (объявлены с модификатором доступа `public`) - так было нужно для простоты изложения. Но, как мы знаем, члены могут быть приватными (`private`).

Приватный член базового класса недоступен для всего внешнего кода, в том числе и для производного класса. Но иногда возникают ситуации, когда нужно предоставить доступ к определенным членам только производным классам, а чтобы все остальные классы (которые не являются производными) использовать эти члены не могли. Такие члены называются защищенными (`protected`). Защищенный член является открытым в пределах иерархии классов, но закрытым за пределами этой иерархии.

Для объявления защищенного члена используется модификатор доступа **`protected`**. Используя **`protected`**, можно создать члены класса, являющиеся закрытыми для своего класса, но все же наследуемыми и доступными для производного класса.

Пример:

```

class Parent
{
    public int x;    // доступен для наследования и для внешнего кода
    protected int y; // дост. для наследования и производных классов
    private int z;   // не доступен для наследования и произв. класса
    int a;           // то же, что и private
}

```

```

public Parent()
{
    x = 10;
    y = 20;
    z = 100;
    a = x;
}
}

```

5.4.3. Запечатанные классы. Ключевое слово sealed

В C# можно создавать классы, недоступные для наследования. Такие классы называются запечатанными (sealed). Для объявления запечатанного класса используется ключевое слово `sealed`:

```
sealed class Parent
```

Если в листинге 5.14 объявить класс `Parent` как запечатанный, то компилятор сообщит об ошибке (рис. 5.3):

'Child': cannot derive from sealed type 'Parent'

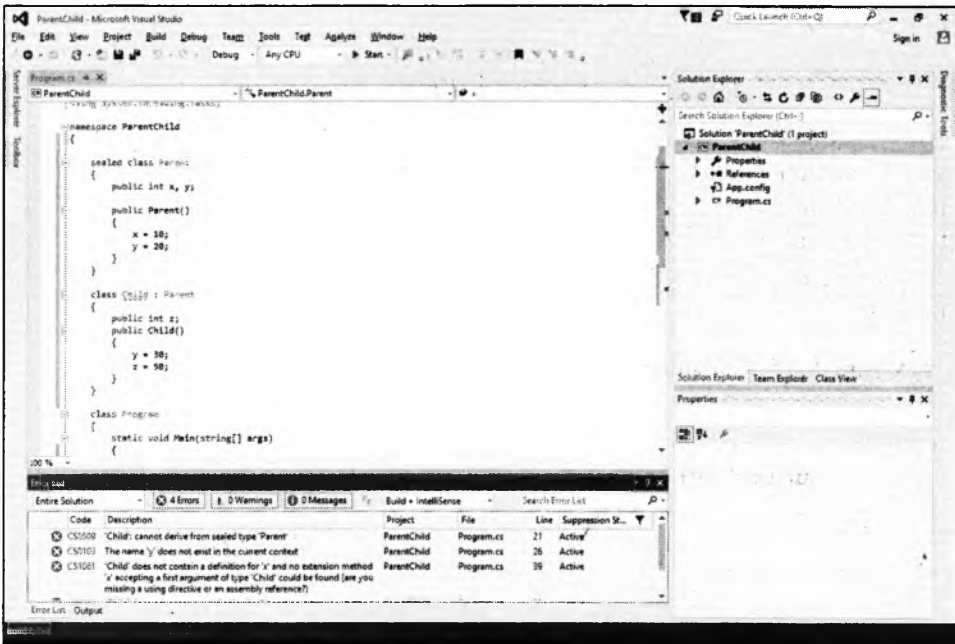


Рис. 5.3. Попытка наследования от запечатанного класса

5.4.4. Наследование конструкторов

В предыдущем примере у каждого из классов был свой конструктор. Как было отмечено, при наследовании наследуются все члены, в том числе и конструкторы. Но в иерархии классов допускается, чтобы у базовых и производных классов были свои собственные конструкторы. Как понять, какой конструктор отвечает за производный класс - конструктор базового, производного класса или же оба?

Конструктор базового класса создает базовую часть объекта. В случае с примером из листинга 5.14 конструктор класса Parent() задает начальные значения полей *x* и *y* базового класса Parent. Конструктор производного класса создает производную часть объекта. В нашем случае - задает значение поля *z* и переопределяет значение *y*, определенное в базовом классе.

Логика следующая - базовому классу неизвестны (да и недоступны) элементы производного класса, следовательно, их создание должно происходить раздельно.

Если же конструктор определен только в производном классе, то конструируется производный класс, а базовая часть объекта создается конструктором по умолчанию.

5.4.5. Соккрытие имен. Ключевое слово `base`

В производном классе могут быть определены члены с таким же именем, как в производном классе. В этом случае член базового класса будет скрыт в производном классе. Формально это не считается ошибкой, но все же компилятор выдаст предупреждение (рис. 5.4). Рассмотрим пример кода, в котором в классе Child определяется поле *y* - с таким же именем, как в базовом классе.

Листинг 5.15. Соккрытие имен

```
class Parent
{
    public int x, y;
    public Parent()
    {
        x = 10;
        y = 20;
    }
}
class Child : Parent
{
    public int z;
```

```

public int y;
public Child()
{
    y = 30;
    z = 50;
}
}

```

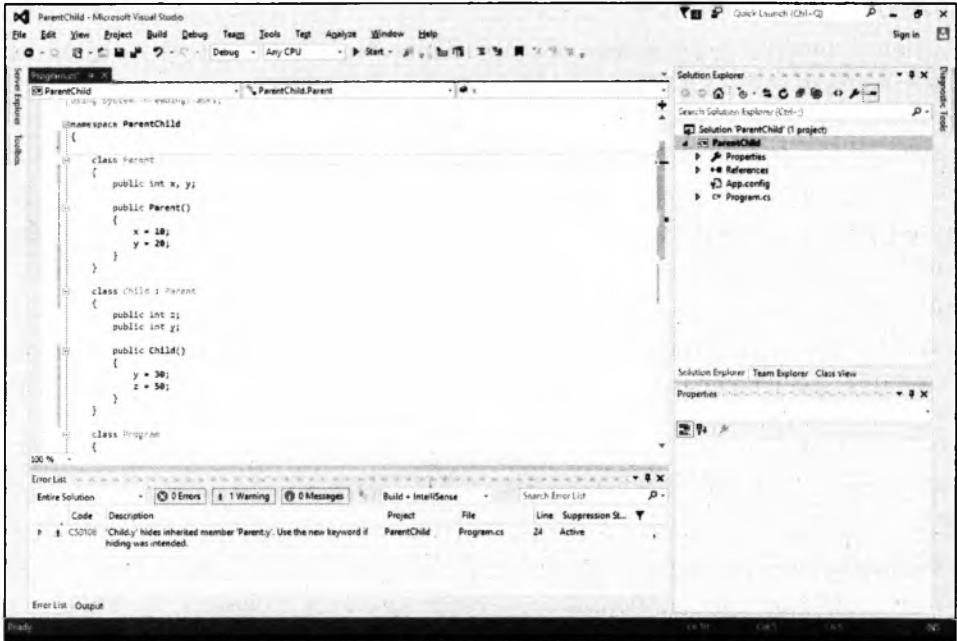


Рис. 5.4. Предупреждение компилятора

Как показано на рис. 5.4, компилятор сообщает о том, что `Child.y` скрывает унаследованный член `Parent.y`. Если член базового класса требуется скрыть намеренно, то перед его именем следует указать ключевое слово `new`, чтобы избежать появления подобного предупреждающего сообщения (рис. 5.5):

```
public new int y;
```

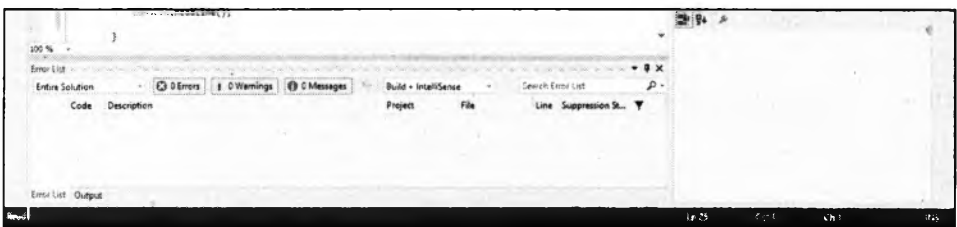


Рис. 5.5. Никаких предупреждений компилятора

Если так нужно будет обратиться к полю `y` из базового класса, нужно использовать ключевое слово **base**:

```
base.y = 5;
```

5.4.6. Виртуальные члены

В данном разделе мы поговорим о виртуальных членах класса. С помощью полиморфизма и переопределения методов подкласс может определить собственную версию метода, определенного в базовом классе. О переопределении членов мы уже говорили, но нерассмотренными остались ключевые слова **virtual** и **override**.

В C# существует специальный тип методов - виртуальные. Метод, определенный, как **virtual**, в базовом классе. Виртуальный метод может быть переопределен в одном или нескольких производных классах. Следовательно, у каждого производного класса может быть свой вариант виртуального метода.

Виртуальные методы представляют интерес тем, что происходит при их вызове по ссылке на базовый класс. Среда .NET определяет именно тот вариант виртуального метода, который нужно вызвать, в зависимости от типа объекта, к которому происходит обращение по ссылке. Обратите внимание: именно среда .NET, а не компилятор C#, поэтому выбор виртуального метода происходит при выполнении, а не при компиляции.

Если базовый класс содержит виртуальный метод, и он наследуется производными классами, при обращении к разным типам объектов по ссылке на базовый класс выполняются разные варианты этого виртуального метода.

Объявить виртуальный метод можно с помощью ключевого слова **virtual**, которое указывается перед его именем. Для переопределения виртуального метода в производном классе используется модификатор **override**. Виртуальный метод не может быть объявлен как **static** или **abstract** (см. след. раздел).

Процесс повторного определения виртуального метода в производном классе называется переопределением метода. При переопределении метода имя, возвращаемый тип и сигнатура переопределяющего метода должны быть точно такими же, как и у того виртуального метода, который переопределяется.

Пример виртуального метода в базовом классе:

```
public virtual string MyString(string s)
{
    return "MyString " + s;
```

```
}
```

Пример переопределения метода в производном классе:

```
public override string MyString(string s)
{
    return "Overrided string " + s;
}
```

5.4.7. Абстрактные классы

Иногда нужно создать базовый класс, где определяется самая общая форма для всех его производных классов. Предполагается, что наполнять эту форму деталями будут производные классы. В таком общем классе определяется лишь характер методов, грубо говоря, в нем определяются функции-заглушки. Реализовывать данные методы будут производные классы.

При создании собственных библиотек классов вы можете убедиться, что у метода часто отсутствует содержательное определение в контексте его базового класса. Данную ситуацию можно решить двумя способами. Первый из них заключается в выводе предупреждающего сообщения. При отладке данный способ пригодится, но в production-версии кода он недопустим. Нужен способ, гарантирующий, что в производном классе действительно будут переопределены все необходимые методы. Такой способ заключается в использовании абстрактного метода.

Абстрактный метод создается с помощью модификатора **abstract**. У абстрактного метода отсутствует тело и он не реализуется в базовом классе. Данный метод должен быть реализован (переопределен) в производном классе, поскольку его вариант из базового класса просто непригоден для использования. Абстрактный метод автоматически становится виртуальным, поэтому он не требует указания модификатора **virtual**. Более того, совместное использование модификаторов **virtual** и **abstract** приведет к ошибке.

Синтаксис определения абстрактного метода следующий:

```
abstract тип имя(список_параметров);
```

Пример определения абстрактного класса:

```
class Parent
{
    public int x, y;

    public Parent()
    {
```

```
        x = 10;  
        y = 20;  
    }  
  
    public abstract int sum();  
}
```

В производном классе абстрактный метод переопределяется так:

```
public override string sum()  
{  
    return x + y + z;  
}
```

Следующая глава будет не менее интересной. В ней мы поговорим об интерфейсах, структурах и перечислениях.

Глава 6.

Интерфейсы, структуры и перечисления



6.1. Понятие интерфейса

Первым делом нужно познакомиться с понятием интерфейса (interface). Интерфейс представляет собой именованный набор абстрактных членов. Абстрактные методы, с которыми мы познакомились в прошлой главе, не имеют никакой стандартной реализации. Конкретные члены, определяемые интерфейсом, зависят от того, какое поведение моделируется с его помощью. Интерфейс выражает поведение класса или структуры.

В библиотеках базовых классов .NET поставляются сотни предопределенных типов интерфейсов, которые реализуются в разных классах.

В интерфейсе тело ни одного из методов не определяется. Другими словами, в интерфейсе вообще нет никакой реализации. В нем только указано, что следует сделать, но не написано, как это сделать. После определения интерфейс может быть реализован в любом количестве классов. А в одном классе можно реализовать любое количество интерфейсов. Благодаря поддержке интерфейсов в С# полностью реализован основной принцип полиморфизма: один интерфейс - множество методов.

Чтобы реализовать интерфейс в классе, нужно указать тела методов, описанных в этом интерфейсе. Каждый класс может как угодно определять свою собственную реализацию интерфейса. А это означает, что в двух разных классах интерфейс может быть реализован по-разному. Но в каждом из них должен поддерживаться один и тот же набор методов данного интерфейса.

Интерфейс объявляется с помощью ключевого слова `interface`. Рассмотрим синтаксис объявления:

```
interface имя{
    return_type method_name_1 (args);
    return_type method_name_2 (args);
    // ...
    return_type method_name_N (args);
}
```

Здесь `имя` - имя конкретного интерфейса, `return_type` - возвращаемый тип, `method_name_N` - имя метода, `args` - список аргументов. По существу все эти методы являются абстрактными.

Как видите, в интерфейсе нет никакой реализации. Просто список методов. В интерфейсе все методы неявно считаются публичными, поэтому **public** явно указывать не нужно.

Кроме методов в интерфейсах могут быть указаны свойства, индексаторы и события. Но интерфейсы не могут содержать члены данных. Также в интерфейсах нельзя определить конструкторы и деструкторы. Ни один из членов интерфейса не может быть статическим.

После определения интерфейса его можно использовать в одном или нескольких классах. Чтобы реализовать интерфейс, нужно указать его имя после имени класса, аналогично тому, как вы это делали при указании базового класса:

```
class имя_класса : имя_интерфейса {
    // тело класса
}
```

В классе можно реализовать несколько интерфейсов. Для этого нужно указать список интерфейсов через запятую. В классе можно наследовать базовый класс и реализовать один или более интерфейсов. В этом случае имя базового класса должно быть указано перед списком интерфейсов, разделяемых запятой.

Методы, реализующие интерфейс, должны быть объявлены как **public**. Возвращаемый тип и сигнатура реализуемого метода должны точно соответствовать возвращаемому типу и сигнатуре, которые указаны при определении интерфейса.

Понимаю, что сейчас совсем ничего не понятно, но все прояснится, как только мы рассмотрим пример, приведенный в листинге 6.1.

Листинг 6.1. Пример определения интерфейсов

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

namespace ConsoleApplication1
{
    // Описываем интерфейс
    public interface ITest
    {
        // Определяем набор абстрактных методов
        int Test1();
        int Test2();
    }
}
```

```

    }
    // Данный класс реализует интерфейс ITest
    class MyClass : ITest
    {
        int My_x;

        // Пример get-тера и set-тера закрытого поля x
        public int x
        {
            set { My_x = value; }
            get { return My_x; }
        }

        public MyClass() { }

        // Реализуем методы интерфейса ITest
        public virtual int Test1()
        {
            return 1;
        }

        public int Test2()
        {
            return 2;
        }
    }

    class Program
    {
        static void Main()
        {
            MyClass obj1 = new MyClass();
            int t1 = obj1.Test1();
            int t2 = obj1.Test2();

            Console.WriteLine("{0} {1}", t1, t2);

            Console.ReadLine();
        }
    }
}

```

6.2. Ключевые слова as и is

Ключевое слово as позволяет определить, поддерживает ли данный тип тот или иной интерфейс. Если объект удастся интерпретировать как указанный

интерфейс, то возвращается ссылка на интересующий интерфейс, а если нет, то ссылка `null`. Следовательно, перед продолжением в коде необходимо предусмотреть проверку на `null`. Рассмотрим небольшой пример:

```
ITest obj = obj1 as ITest;
if (obj != null)
    Console.WriteLine("Есть поддержка интерфейса ITest");
else
    Console.WriteLine("ITest не поддерживается");
```

Проверить, был ли реализован нужный интерфейс, можно с помощью ключевого слова `is`. Если запрашиваемый объект не совместим с указанным интерфейсом, возвращается значение `false`, а если совместим, то можно спокойно вызывать члены этого интерфейса:

```
if (obj1 is ITest)
    Console.WriteLine("Да");
else
    Console.WriteLine("Нет");
```

6.3. Интерфейсные свойства

Подобно методам, свойства указываются в интерфейсе вообще без тела. Синтаксис объявления интерфейсного свойства выглядит так:

```
// Интерфейсное свойство
тип имя{
    get;
    set;
}
```

В определении интерфейсных свойств, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: `get` или `set` соответственно.

Пример определения интерфейсных свойств приведен в листинге 6.2.

Листинг 6.2. Пример определения интерфейсных свойств

```
using System;
namespace ConsoleApplication1
{
    interface ITest
    {
        string Str
        {
            get;
```

```

        set;
    }
}
class MyClass : ITest
{
    string myStr;
    public string Str
    {
        set
        {
            myStr = value;
        }
        get
        {
            return myStr;
        }
    }
}
class Program
{
    static void Main()
    {
        MyClass obj1 = new MyClass();
        user1.Str = "Hello";
        Console.ReadLine();
    }
}
}

```

6.4. Интерфейсы и наследование

Подобно классам, один интерфейс может наследовать другой, при этом синтаксис наследования интерфейсов такой же, как у классов. Если в классе реализуется один интерфейс, наследующий другой, в нем должны быть реализованы все члены, определенные в цепочке наследования интерфейсов. Интерфейсы могут быть организованы в иерархии. Если какой-то интерфейс расширяет существующий, он наследует все абстрактные члены своих родителей. В отличие от классов, производные интерфейсы никогда не наследуют саму реализацию. Вместо этого они просто расширяют собственное определение за счет добавления дополнительных абстрактных членов. Пример наследования интерфейсов приведен в листинге 6.3.

Листинг 6.3. Пример наследования интерфейсов

```
using System;
```

```

namespace ConsoleApplication1
{
    public interface A
    {
        int TestA();
    }
    // Интерфейс B наследует интерфейс A
    public interface B : A
    {
        int TestB();
    }
    // Класс MyClass наследует интерфейс B
    // и реализует методы интерфейсов A и B
    class MyClass : B
    {
        public int TestA()
        {
            return 1;
        }

        public int TestB()
        {
            return 2;
        }
    }
    class Program
    {
        static void Main()
        {
        }
    }
}

```

6.5. Структуры

Объекты конкретного класса доступны по ссылке (поскольку классы относятся к ссылочным типам данных), в отличие от значений обычных типов, которые доступны непосредственно. В некоторых ситуациях для повышения эффективности программы полезно иметь прямой доступ к объектам как к значениям простых типов. Ведь каждый доступ к объекту (даже к самому небольшому) требует дополнительные системные ресурсы (оперативную память и процессорное время).

Для решения подобных задач используются структуры. Структуры подобны классам, но они относятся не к ссылочным типам данных. Структуры отличаются от классов способом хранения в памяти и способом доступа к

ним. Классы размещаются в куче, а структуры - в стеке. Это и есть самая большая разница. Из соображений эффективности программы структуры полезно использовать для небольших типов данных.

Синтаксически описание структуры похоже на классы. Основное отличие в том, что класс определяется с помощью служебного слова **class**; а структура - с помощью служебного слова **struct**. Синтаксис следующий:

```
struct имя : интерфейсы {
// объявления членов
}
```

Как и у класса, у структуры могут быть методы, поля, индексаторы, свойства, методы и события. В структурах даже можно объявлять конструкторы, но не деструкторы. Однако в структуре нельзя определить конструктор по умолчанию (конструктор без параметров).

Поскольку структуры не поддерживают наследование, их члены нельзя указывать как **abstract**, **virtual** или **protected**.

Объект структуры можно создать с помощью оператора **new** - как вы это делали для класса. Но обычно в этом нет необходимости, поскольку при использовании оператора **new** вызывается конструктор по умолчанию, а если вы не используете оператор **new**, то объект все равно создается, но не инициализируется. Пример объявления структуры приведен в листинге 6.4.

Листинг 6.4. Пример структуры

```
using System;

namespace ConsoleApplication1
{
    // Создадим структуру
    struct CarInfo
    {
        public string Brand;
        public string Model;

        public CarInfo(string Brand, string Model)
        {
            this.Brand = Brand;
            this.Model = Model;
        }

        public void GetCarInfo()
```

```

        {
            Console.WriteLine("Бренд: {0}, Модель: {1}",
Brand, Model);
        }
    }

    class Program
    {
        static void Main()
        {
            CarInfo car1 = new CarInfo("Audi", "A6");
            CarInfo car2 = new CarInfo("BMW", "X5");
            Console.Write("car 1: ");
            car1.GetCarInfo();
            Console.Write("car 2: ");
            car2.GetCarInfo();

            car1 = car2;
            car2.Brand = "Toyota";
            car2.Model = "Camry";
            Console.Write("car1: ");
            car1.GetCarInfo();
            Console.Write("car2: ");
            car2.GetCarInfo();

            Console.ReadLine();
        }
    }
}

```

Вывод программы будет таким:

```

car1: Бренд Audi Модель A6
car2: Бренд BMW Модель X5
car1: Бренд BMW Модель X5
car2: Бренд Toyota Модель Camry

```

Если одна структура присваивается другой, то при этом создается копия ее объекта. Это главное отличие структуры от класса. Когда ссылка на один класс присваивается ссылке на другой класс, в итоге ссылка в левой части оператора присваивания указывает на тот же самый объект, что и ссылка в правой его части. А когда переменная одной структуры присваивается переменной другой структуры, создается полная копия объекта структуры из

правой части оператора присваивания. Если бы вместо структуры CarInfo мы бы использовали класс с таким же именем, то вывод программы был бы таким:

```
car1: Бренд Audi Модель A6
car2: Бренд BMW Модель X5
car1: Бренд Toyota Модель Camry
car2: Бренд Toyota Модель Camry
```

6.6. Перечисления

Перечисление (enumeration) — это определяемый пользователем целочисленный тип. Когда вы объявляете перечисление, то указываете набор допустимых значений, которые могут принимать экземпляры перечислений. Этим значениям можно присвоить имена, понятные человеку. Если присвоить экземпляру перечисления значение, не входящее в список ранее определенных, компилятор выдаст ошибку.

Синтаксис определения перечисления такой:

```
enum имя {список_перечисления};
```

Листинг 6.5 содержит пример определения и использования перечисления.

Листинг 6.5. Использование перечислений

```
using System;
namespace ConsoleApplication1
{
    // Создать перечисление
    enum car : long { Brand, Model, Year, Engine }

    class Program
    {
        static void Main()
        {
            car car1;
            for (car1 = car.Brand; car1 <= car.Engine; car1++)
                Console.WriteLine("Ключ: \"{0}\"", значение {1}",
                                   car1, (int)car1);

            Console.ReadLine();
        }
    }
}
```

Каждая символически обозначаемая константа в перечислении имеет целое значение. Однако неявные преобразования перечислимого типа во встроенные целочисленные типы и обратно в C# не определены, а значит, в подобных случаях требуется явное приведение типов, что можно наблюдать в нашей программе. Приведение типов требуется при преобразовании двух перечислимых типов. Но поскольку перечисления обозначают целые значения, то их можно, например, использовать для управления оператором выбора **switch** или же оператором цикла **for**.

При желании можно самостоятельно назначить значения элементам перечисления

```
enum car { Brand = 1, Model = 2, Year = 3, Engine = 4 }
```

Глава 7.

Обработка исключений



7.1. Введение в обработку исключений

От ошибок никто не застрахован. Но не всегда ошибки происходят по вине разработчика. В некоторых случаях ошибку могут сгенерировать действия пользователя, например, недопустимый ввод. Также ошибка может произойти из-за отсутствия того или иного драйвера, который использует приложение и который или отсутствует, или работает некорректно на конечной системе. Самый простой пример - калькулятор. Все знают, что на 0 делить нельзя, но пользователь по тем или иным причинам может указать 0 в качестве второго операнда (или же 0 будет сгенерирован в результате вычисления какой-то формулы, если вы разрабатываете сложный математический калькулятор). В этом случае произойдет ошибка деления на 0 и будет сгенерировано исключение `System.DivideByZeroException`. Программа будет остановлена, а пользователь увидит окно с ошибкой.

Несмотря на то, что ошибка произошла по вине пользователя, ответственность за нее все равно несет программист. Ведь программа не должна завершаться, она должна обработать это исключение, сообщить пользователю о недопустимости такой операции и продолжить выполнение. Программист, разумеется, все это должен закодировать.

Существует две тактики обработки исключений. Первая заключается в том, чтобы не допустить появление исключения как такового. В случае с калькулятором приложение должно проверить второй операнд до выполнения самого деления и сообщить пользователю о том, что он собирается выполнить недопустимую операцию. Аналогично, при работе с файлами приложение сначала должно проверить существование файла, а уже потом открывать его для чтения.

Вторая тактика основана на предположении того, что какой-то код может вызвать исключение. Этот код заключается в блок `try/catch`, и производится обработка ожидаемого исключения.

Первая тактика считается предпочтительной, но она возможна не всегда. В .NET предусмотрена развитая система обработки ошибок. Программист может написать код, производящий обработку для каждого типа ошибки, а также отделить код, потенциально порождающий ошибки, от кода, обрабатывающего их.

Прежде чем мы перейдем к самим исключениям, нужно поговорить о типах ошибок. Всего существует три типа ошибок - ошибки программиста, ошибки пользователя и исключения. К первому типу ошибок относятся ошибки в логике программы. Например, ошибка неучтенной единицы (off-by-one error). Встречается, когда, например, количество итераций в цикле на единицу больше или меньше количества обрабатываемого массива. Программист может забыть, что нумерация массива начинается с 0, и начать с единицы. Если в цикле используется условие остановки при достижении значения, определенного в свойстве Length, то цикл просто не обработает один элемент - первый (с индексом 0). Если же количество элементов в цикле задано жестко, например, 10, то произойдет нарушение границ массива. Это только один из примеров ошибок программиста. Их существуют сотни, если не тысячи.

Ко второму типу ошибок относятся ошибочные действия пользователей. Самой распространенной ошибкой является неправильный ввод пользователя. Если приложение не обрабатывает ввод пользователя, то это может привести к очень серьезным проблемам. Нет, это не завершение программы при исключении. Это, скажем так, мелочи. Представьте, что вы разрабатываете веб-страницу, предназначенную для ввода информации, которая будет сохранена в базу данных. Если вы не обрабатываете ввод, то пользователь вместо обычной информации может указать в поле ввода SQL-код, который удалит всю базу данных. Такая угроза называется SQL Injection.

Исключение (exception) или исключительная ситуация - аномалия, которая может возникнуть во время выполнения (runtime) и которые трудно предусмотреть во время программирования. Примеры исключительных ситуаций - открытие поврежденного файла, попытка чтения из таблицы базы данных, которая уже не существует или же ее структура изменена. Если с ошибкой деления на 0 или с открытием файла все просто: достаточно или проверить второй операнд, или существование самого файла, то с такими ошибками все не так однозначно. Приложение открывает файл и ожидает, что он будет в определенном формате, например, в XML. Проверить принадлежность к тому или иному формату можно при открытии файла - если приложение откроет не XML, файл, а обычный текстовый файл, то сразу сообщит об этом пользователю. Но если приложение таки открывает XML-файл, оно начинает чтение. По мере чтения оказывается, что файл поврежден. Сразу проверить, поврежден ли файл или нет, практически невозможно. Например, размер файла может быть довольно большим, и если проверять корректность файла перед каждой обработкой, то работать программа будет неэффективно. Ведь ошибки происходят довольно редко, а проверять корректность придется каждый раз при открытии файла.

Однако программист может предусмотреть, что такая исключительная ситуация может произойти. Он должен перехватить ее и обработать.

До появления .NET обработка ошибок в Windows была довольно сложной. Многим программистам приходилось включать собственную логику обработки ошибок в приложение. Кроме этих приемов, которые были разработаны самими разработчиками, в API-интерфейсе Windows определены сотни кодов ошибок с помощью `#define` и `HRESULT`, а также множество вариаций простых булевских значений (`bool`, `BOOL`, `VARIANT_BOOL` и т.д.). В случае с COM-приложениями, написанными на языках VB6 и C++, все еще сложнее.

Основная проблема со всеми этими подходами - отсутствие симметрии. Каждая из них довольно эффективна в рамках какого-то одного приложения или одной технологии, может быть, даже в рамках одного языка программирования.

В .NET поддерживается стандартная методика для генерации и выявления ошибок в исполняющей среде. Данная методика позволяет разработчикам использовать в области обработки ошибок унифицированный подход, являющийся общим для всех .NET-языков. Благодаря этой методике, C#-программист может обрабатывать ошибки точно таким же образом, как и VB-программист.

7.2. Перехват исключений. Блоки `try`, `catch`, `finally`

Для обработки исключений в C# используются следующие блоки:

- `try` - содержат код, который потенциально может столкнуться с исключительной ситуацией.
- `catch` - содержит код, обрабатывающий ошибочные ситуации, происходящие в коде блока `try`.
- `finally` - содержит код, очищающий любые ресурсы или выполняющий другие действия, которые обычно нужно выполнить в конце блоков `try` или `catch`. Данный код выполняется независимо от того, сгенерировано исключение или нет.

В основном используются блоки `try/catch`. Синтаксис следующий:

```

try {
// Блок кода, проверяемый на наличие ошибок.
}

catch (Exception exOb) {
// Обработчик исключения Exception
}
catch (Exception exOb) {
// Обработчик исключения Exception
}
...

```

Здесь `Exception` - это тип исключения. Если исключение сгенерировано оператором `try`, оно перехватывается составляющим ему пару оператором `catch`, который и производит обработку этого исключения. В зависимости от типа исключения выполняется и соответствующий оператор `catch`, если их задано несколько.

При перехвате исключения переменная `exOb` получает свое значение. Вообще указывать переменную `exOb` необязательно, если обработчику исключений не требуется доступ к объекту исключения, что на самом деле бывает очень часто - ведь достаточно просто знать тип исключения.

Если исключение не генерируется, то блок оператора `try` завершает как обычно, а все блоки `catch` пропускаются. Выполнение программы возобновляется с первого оператора, который следует после завершающего оператора `catch`. Другими словами, содержимое `catch` выполняется только, если генерируется исключение.

Посмотрим, как можно обработать деление на 0, о котором мы так много говорили:

```

int x = 10, y = 0, z;
try
{
    z = x / y;
}
catch (DivideByZeroException)
{
    Console.WriteLine("Деление на 0");
}

```

Преимущество обработки исключений в том, что ваша программа не будет завершаться аварийно. В данном случае программа отобразит сообщение и продолжит обычное выполнение. Правда, что произойдет далее, - зависит от самой программы.

При необходимости можно указать несколько блоков **catch**:

```
try
{
    // Некоторые вычисления
}
catch (OverflowException)
{
    Console.Write("Переполнение");
}
catch (DivideByZeroException)
{
    Console.WriteLine("Деление на 0");
}
catch (IndexOutOfRangeException)
{
    Console.WriteLine("Выход за пределы диапазона");
}
```

Каждый блок **catch** будет реагировать только на свой класс исключения.

7.3. Класс Exception

Все определяемые исключения всегда наследуются от базового класса `System.Exception`, который, в свою очередь, наследуется от класса `System.Object`. Класс `System.Exception` содержит свойства и методы. Полное описание этого класса представлено по ссылке:

<https://msdn.microsoft.com/ru-ru/library/system.exception%28v=vs.110%29.aspx>

Мы же рассмотрим основные свойства этого класса, которые используются чаще всего (табл. 7.1).

Таблица 7.1. Некоторые свойства класса `System.Exception`

Свойство	Описание
Data	Позволяет извлекать коллекцию пар "ключ/значение" (представленную объектом, реализующим интерфейс <code>IDictionary</code>), которая предоставляет дополнительную информацию об исключении. По умолчанию данная коллекция пуста и должна определяться программистом. Свойство доступно только для чтения.
HelpLink	Позволяет получать или устанавливать URL-адрес, по которому доступен справочный файл или веб-сайт с подробным описанием ошибки.

InnerException	Используется для получения информации о предыдущем исключении или исключениях, которые послужили причиной возникновения текущего исключения. Свойство доступно только для чтения.
Message	Содержит сообщение об ошибке. Доступно только для чтения.
Source	Позволяет получать или устанавливать имя сборки или объекта, который привел к выдаче исключения.
StackTrace	Содержит строку с описанием последовательности вызовов, которая привела к возникновению исключения. Доступно только для чтения. Полезно использовать во время отладки. Содержимое этого поля можно сохранять в журнале ошибок.
TargetSite	Возвращает объект <code>MethodBase</code> с описанием многочисленных деталей метода, который привел к выдаче исключения. Доступно только для чтения.

Рассмотрим пример использования объекта `Exception`:

```
int x = 10, y = 0, z;
try
{
    z = x / y;
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Деление на 0");
    Console.WriteLine(ex.Message);
}
```

Отличия от предыдущего примера выделены жирным. В данном примере мы выводим дополнительную информацию из свойства `Message`.

Самостоятельно породить исключение можно с помощью оператора **throw**. Иногда это полезно для отладки. Пример:

```
Exception excep = new Exception();
excep.HelpLink = "dkws.org.ua";
excep.Data.Add("Time: ", DateTime.Now);
excep.Data.Add("Reason: ", "Wrong input");
throw excep;
```

Рассмотрим пример блока **catch**, который выводит очень подробную информацию об ошибке:

```
catch (Exception ex)
{
    Console.WriteLine("Ошибка: ");
}
```

```

Console.Write(ex.Message + "\n\n");
Console.Write("Метод: ");
Console.Write(ex.TargetSite + "\n\n");
Console.Write("Стек: ");
Console.Write(ex.StackTrace + "\n\n");
Console.Write("Подробности: ");
Console.Write(ex.HelpLink + "\n\n");
if (ex.Data != null)
{
    Console.WriteLine("Подробности: \n");
    foreach (DictionaryEntry d in ex.Data)
        Console.WriteLine("-> {0} {1}", d.Key, d.Value);
}
}

```

7.4. Исключения уровня системы

В библиотеке классов .NET содержится множество классов, которые наследуются от `System.Exception`. Так, в пространстве имен `System` определены ключевые классы исключений - `StackOverflowException` (исключение переполнения стека), `IndexOutOfRangeException` (исключение выхода за диапазон) и др. В других пространствах имен также определены исключения - например, исключения, возникающие при вводе/выводе, исключения, которые связаны с базами данных и т.д.

Исключения, генерируемые самой платформой .NET, называются исключениями уровня системы. Такие исключения считаются также фатальными ошибками. Они наследуются прямо от базового класса `System.SystemException`, который, в свою очередь, наследуется от `System.Exception`.

Кроме исключений уровня системы есть еще и исключения уровня приложений (класс `System.ApplicationException`). Если вам нужно создать собственные исключения, предназначенные для конкретного приложения, их нужно наследовать от `System.ApplicationException`.

В классе `System.ApplicationException` никаких членов, кроме набора конструкторов, не предлагается. Единственная цель этого класса - указание на источник ошибки. Если произошло исключение, унаследованное от `System.ApplicationException`, программист может смело полагать, что исключение было вызвано кодом функционирующего приложения, а не библиотекой базовых классов .NET.

Если вы планируете создавать свои классы исключений, то вы должны придерживаться рекомендаций .NET, а именно, пользовательские классы должны:

- наследоваться от `ApplicationException`;
- содержать атрибут `[System.Serializable]`;
- иметь конструктор по умолчанию;
- иметь конструктор, который устанавливает значение унаследованного свойства `Message`;
- иметь конструктор для обработки "внутренних исключений";
- иметь конструктор для обработки сериализации типа.

Рассмотрим "болванку" такого класса (лист. 7.1).

Листинг 7.1. Болванка класса пользовательского исключения

```
[System.Serializable]
public class MyExc : ApplicationException
{
    public MyExc () { }
    public MyExc (string message) : base(message) { }
    public MyExc (string message, Exception ex) : base(message)
{ }
    protected MyExc (System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }
}
```

Данный класс, хотя ничего и не делает, но он полностью соответствует рекомендациям .NET.

7.5. Ключевое слово `finally`

Осталось нерассмотренным ключевое слово **`finally`**. Если вам нужно определить код, который будет выполняться после выхода из блока `try/catch`, вам нужно использовать блок **`finally`**. Использование этого блока гарантирует, что некоторый набор операторов будет выполняться всегда, независимо от того, возникло исключение (любого типа) или нет.

Синтаксис следующий:

```
try {
    // Блок кода, проверяемый на наличие ошибок.
}

catch (Exception exOb) {
    // Обработчик исключения Exception
}
```

```

}
catch (Ехсер2 exOb) {
// Обработчик исключения Ехсер2
}
finally {
// Этот код будет выполнен после обработки исключений
}

```

Блок **finally** выполняется каждый раз, когда происходит выход из блока **try/catch**, независимо от причин, которые привели к этому блоку. Если блок **try** завершается нормально или по причине исключения, то последним выполняется код, определяемый в блоке **finally**. Блок **finally** выполняется и в том случае, если любой код в блоке **try** или в связанных с ним блоках **catch** приводит к возврату из метода.

Пример:

```

int x = 10, y = 0, z;
try
{
    z = x / y;
}
catch (DivideByZeroException)
{
    Console.WriteLine("Деление на 0");
}
finally
{
    Console.WriteLine("Конец программы");
}

```

7.6. Ключевые слова **checked** и **unchecked**

Ранее я обещал, что в этой главе мы рассмотрим ключевые слова **checked** и **unchecked**. В языке С# можно указывать, будет ли в коде сгенерировано исключение при переполнении. Для этого используются эти ключевые слова. Если нужно указать, что выражение будет проверяться на переполнение, нужно использовать ключевое слово **checked**. Если нужно проигнорировать переполнение - ключевое слово **unchecked**.

Синтаксис такой:

```

checked {
// операторы
}

```

Если вычисление проверяемого выражения приводит к переполнению, то будет сгенерировано исключение `OverflowException`. Синтаксис `unchecked` такой же:

```
unchecked {
// операторы
}
```

Рассмотрим пример кода:

```
byte x, y, res;
try
{
    Console.Write("Введите x:");
    x = unchecked((byte)int.Parse(Console.ReadLine()));
    Console.Write("Введите y:");
    y = unchecked((byte)int.Parse(Console.ReadLine()));

    checked
    {
        res = (byte)(x + y);
        Console.WriteLine("res = {0}", res);
    }
}
catch (OverflowException)
{
    Console.WriteLine("Переполнение");
}
```

Здесь специально используется тип **byte** (диапазон 0..255), чтобы проще было вызвать переполнение. Потенциально опасный код заключен в `checked`. Обратите внимание: код, получающий значения **x** и **y**, заключен в `unchecked`. Для одиночного выражения допускается использование круглых скобок вместо фигурных.

Глава 8.

Коллекции и итераторы

Основы языка C#,
первые программы

Клиент-серверные
приложения

Многопоточное
программирование

Создание мобильных
приложений на C#

8.1. Введение в коллекции

Коллекция - это совокупность объектов. В среде .NET существует множество интерфейсов и классов, в которых определяются и реализуются различные типы коллекций.

С помощью коллекций существенно упрощается программирование многих задач, поскольку они предлагают готовые решения для создания целого ряда типичных, но иногда очень трудных для разработки структур данных. Так, в среду .NET встроены коллекции для поддержки хеш-таблиц, динамических массивов, очередей, стеков, связанных списков. Коллекции заслуживают внимания всех C#-программистов. Зачем изобретать колесо заново?

Ранее существовали только классы необобщенных коллекций. Но позже появились обобщенные классы и интерфейсы. Благодаря этому общее количество классов удвоилось. Вместе с библиотекой TPL (используется для распараллеливания задач) появился ряд новых классов коллекций, предназначенных для применения в тех случаях, когда доступ к коллекции осуществляется из нескольких потоков. О многопоточности мы поговорим в последней главе этой книги.

Интерфейс Collections API настолько важен, что он занимает огромную часть всей среды .NET.

Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе ArrayList, Hashtable, Stack и Queue, могут применяться в исходном виде - без изменений. При желании программист может создать собственную коллекцию, но учитывая богатый набор коллекций, такая необходимость возникает редко.

Примечание. Если вы ранее программировали на C++, то вам будет интересно знать, что классы коллекций по своей сути подобны классам стандартной библиотеки шаблонов (Standard Template Library — STL), определенной в C++. Коллекция в терминологии C++ - это не что иное, как контейнер.

В .NET поддерживаются пять типов коллекций:

- **Необобщенные** - коллекции, реализующие ряд основных структур данных, включая динамический массив, стек, очередь, а также словари. Об этом типе коллекций нужно помнить следующее: все они

работают с типом данных **object**. Поэтому необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен **System.Collections**.

- **Специальные** - работают с данными конкретного типа. Имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Такие коллекции объявляются в пространстве имен **System.Collections.Specialized**.
- **Поразрядная коллекция** - такая коллекция одна, но она не попадает ни под один другой тип коллекций. Она определена в **System.Collections** и называется **BitArray**. Коллекция поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например **И**, **ИЛИ**, **исключающее ИЛИ**.
- **Обобщенные** - такие коллекции обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связанные списки, стеки, очереди и словари. В силу своего обобщенного характера такие коллекции являются типизированными. Объявляются в пространстве имен **System.Collections.Generic**.
- **Параллельные** - поддерживают многопоточный доступ к коллекции. Определены в пространстве имен **System.Collections.Concurrent**.

Основным для всех коллекций является понятие перечислителя, который поддерживается в необобщенных интерфейсах **IEnumerator** и **IEnumerable**, а также в обобщенных интерфейсах **IEnumerator<T>** и **IEnumerable<T>**.

Перечислитель предоставляет стандартный способ поочередного доступа к элементам коллекции. Другими словами, он перечисляет содержимое коллекции.

В каждой коллекции реализована обобщенная или необобщенная форма интерфейса **IEnumerable**, элементы любого класса коллекции должны быть доступны с помощью методов, которые определены в интерфейсе **IEnumerator** или **IEnumerator<T>**. Для поочередного обращения к содержимому коллекции в цикле **foreach** используется перечислитель.

С перечислителями тесно связаны итераторы. Итератор упрощает процесс создания классов коллекций, например специальных, поочередное обращение к которым организуется в цикле **foreach**.

В таблице 8.1 приведены интерфейсы, реализуемые в коллекциях С#.

Таблица 8.1. Интерфейсы, реализуемые в коллекциях C#

Интерфейс	Описание
<code>ICollection<T></code>	Интерфейс, реализованный классами обобщенных коллекций. Позволяет получить количество элементов в коллекции (свойство <code>Count</code>), скопировать коллекцию в массив (метод <code>CopyTo()</code>). Также позволяет добавлять и удалять элементы из коллекции (методы <code>Add()</code> , <code>Remove()</code> , <code>Clear()</code>).
<code>IEnumerable<T></code>	Необходим, когда с коллекцией используется оператор <code>foreach</code> . Этот интерфейс определяет метод <code>GetEnumerator()</code> , возвращающий перечислитель, который реализует <code>IEnumerator</code> .
<code>ISet<T></code>	Впервые появился в версии .NET 4. Реализуется множествами. Позволяет комбинировать различные множества в объединения, а также проверять, не пересекаются ли два множества. <code>ISet<T></code> унаследован от <code>ICollection<T></code> .
<code>IList<T></code>	Предназначен для создания списков, элементы которых доступны по своим позициям. Определяет индексатор, а также способы вставки и удаления элементов в определенные позиции (методы <code>Insert()</code> и <code>Remove()</code>). <code>IList<T></code> унаследован от <code>ICollection<T></code> .
<code>IComparer<T></code>	Реализован компаратором и используется для сортировки элементов внутри коллекции с помощью метода <code>Compare()</code> .
<code>IDictionary<TKey, TValue></code>	Реализуется обобщенными классами коллекций, элементы которых состоят из ключа и значения. С помощью этого интерфейса можно получать доступ ко всем ключам и значениям, извлекать элементы по индексатору типа ключа, а также добавлять и удалять элементы.
<code>ILookup<TKey, TValue></code>	Похож на <code>IDictionary<TKey, TValue></code> и поддерживает ключи и значения. Однако в этом случае коллекция может содержать множественные значения для одного ключа.
<code>IProducerConsumerCollection<T></code>	Появился в .NET 4. Используется для поддержки новых, безопасных в отношении потоков классов коллекций.

`IEqualityComparer<T>`

Реализован компаратором, который может быть применен к ключам словаря. Через этот интерфейс объекты могут быть проверены на предмет эквивалентности друг другу. Поддерживается массивами и кортежами.

8.2. Необобщенные коллекции

Необобщенные коллекции вошли в состав среды первыми. Они появились в самой первой версии .NET, поэтому считаются самыми давними. Необобщенные коллекции определяются в пространстве имен `System.Collections`.

Такие коллекции представляют собой структуры данных общего назначения, оперирующие ссылками на объекты. Позволяют манипулировать объектом любого типа, хотя и не типизированным способом.

В способе их манипулирования объектами заключается их основное преимущество и их основной недостаток. Необобщенные коллекции оперируют ссылками на объекты, в них можно хранить разнотипные данные. В некоторых ситуациях это удобно. Например, если вам нужно манипулировать совокупностью разнотипных объектов или же когда типы хранящихся в коллекции объектов заранее неизвестны. Однако, если коллекция предназначена для хранения объекта конкретного типа, то здесь вам придется столкнуться с одним неприятным нюансом: необобщенные коллекции не обеспечивают типовую безопасность, которая обеспечивается обобщенными коллекциями.

Необобщенные коллекции определены во многих интерфейсах и классах, которые реализуют эти интерфейсы.

Множество необобщенных коллекций определено в пространстве имен `System.Collections`. Интерфейсы, являющиеся фундаментом для необобщенных коллекций, представлены в таблице 8.2.

Таблица 8.2. Интерфейсы, используемые в необобщенных коллекциях

Интерфейс	Описание
<code>ICollection</code>	Определяет элементы, которые должны иметь все необобщенные коллекции
<code>IComparer</code>	Определяет метод <code>Compare()</code> для сравнения объектов, хранящихся в коллекции
<code>IDictionary</code>	Определяет коллекцию, состоящую из пар "ключ-значение" (словарь)

IDictionaryEnumerator	Определяет перечислитель для коллекции, реализующей интерфейс IDictionary
IEnumerable	Определяет метод GetEnumerator(), предоставляющий перечислитель для любого класса коллекции
IEnumerator	Предоставляет методы, позволяющие получать содержимое коллекции по очереди
IEqualityComparer	Сравнивает два объекта на предмет равенства
IHashCodeProvider	Устарел. Используйте интерфейс IEqualityComparer
ICollection	Определяет коллекцию, доступ к которой можно получить с помощью индексатора
IStructuralComparable	Содержит метод CompareTo(), применяемый для структурного сравнения
IStructuralEquatable	Содержит метод Equals(), применяемый для выяснения структурного, а не ссылочного равенства. Кроме того, определяет метод GetHashCode()

Особое место в программировании занимают словари. Интерфейс IDictionary определяет коллекцию, которая состоит из пар “ключ-значение”. В пространстве имен System.Collections определена структура DictionaryEntry. Необобщенные коллекции пар “ключ-значение” сохраняют эти пары в объекте типа DictionaryEntry. В структуре DictionaryEntry определяются два следующих свойства:

```
public object Key { get; set; }
public object Value { get; set; }
```

Данные свойства используются для доступа к ключу или значению, связанному с элементом коллекции. Построить объект типа DictionaryEntry можно с помощью следующего конструктора:

```
public DictionaryEntry(object key, object value)
```

Параметр key - это ключ, value - это значение.

Таблица 8.3 содержит классы необобщенных коллекций.

Таблица 8.3. Классы необобщенных коллекций

Класс	Описание
ArrayList	Определяет динамический массив. Динамические массивы при необходимости могут увеличивать свой размер
Hashtable	Используется для работы с хеш-таблицами для пар “ключ-значение”

Queue	Очередь или список, построенный по принципу FIFO - первым зашел, первым вышел
SortedList	Отсортированный список пар "ключ-значение"
Stack	Стек или список, построенный по принципу LIFO - последним зашел, первым вышел

8.3. Обобщенные коллекции

Как уже было отмечено, благодаря введению обобщений, количество коллекций существенно увеличилось. Все обобщенные коллекции объявляются в пространстве имен `System.Collections.Generic`.

Обычно классы обобщенных коллекций являются не более чем обобщенными эквивалентами классов необобщенных коллекций. В некоторых случаях одни и те же функции существуют параллельно в классах обобщенных и необобщенных коллекций, хотя и под разными именами. Примером может послужить обобщенный вариант класса `HashTable`, который называется `Dictionary`. Аналогично, обобщенный класс `List` - это аналог необобщенного класса `ArrayList`. Вообще, практически все, что вы знаете о необобщенных коллекциях, применимо и к обобщенным коллекциям.

Обобщенные коллекции из пространства имен `System.Collections.Generic` определены в таблице 8.4.

Таблица 8.4. Интерфейсы обобщенных коллекций

Интерфейс	Описание
<code>ICollection<T></code>	Здесь определены основные свойства обобщенных коллекций
<code>IComparer<T></code>	Определяет обобщенный метод <code>Compare()</code> для сравнения объектов, хранящихся в коллекции
<code>IDictionary<Tkey, TValue></code>	Определяет словарь, то есть обобщенную коллекцию, состоящую из пар "ключ-значение"
<code>IEnumerable<T></code>	Содержит обобщенный метод <code>GetEnumerator()</code> , предоставляющий перечислитель для любого класса коллекции
<code>IEnumerator<T></code>	Предоставляет методы, позволяющие получать содержимое коллекции по очереди
<code>IEqualityComparer<T></code>	Компаратор. Сравнивает два объекта на предмет равенства
<code>IList<T></code>	Определяет обобщенную коллекцию, доступ к которой осуществляется с помощью индексатора

Аналогично, для работы со словарем в пространстве имен System.Collections.Generic определена структура `KeyValuePair<TKey, TValue>`. В этой структуре определяются два следующих свойства:

```
public TKey Key { get; };
public TValue Value { get; };
```

Для создания объекта типа `KeyValuePair<TKey, TValue>` используется конструктор:

```
public KeyValuePair(TKey key, TValue value)
```

В таблице 8.5 приведены основные классы обобщенных коллекций, определенные в пространстве имен System.Collections.Generic.

Таблица 8.5. Основные классы обобщенных коллекций

Класс	Описание
<code>Dictionary<Tkey, TValue></code>	Словарь. Используется для хранения пары "ключ-значение". Функции такие же, как и у необобщенного класса <code>HashTable</code>
<code>HashSet<T></code>	Используется для хранения уникальных значений с использованием хэш-таблицы
<code>LinkedList<T></code>	Сохраняет элементы в двунаправленном списке
<code>List<T></code>	Создает динамический массив. Функционал такой же, как и у необобщенного класса <code>ArrayList</code>
<code>Queue<T></code>	Очередь. Функционал такой же, как у необобщенного класса <code>Queue</code>
<code>SortedDictionary<TKey, TValue></code>	Используется для создания отсортированного списка из пар "ключ-значение"
<code>SortedList<TKey, TValue></code>	Создает отсортированный список. Функционал такой же, как у необобщенного класса <code>SortedList</code>
<code>SortedSet<T></code>	Создает отсортированное множество
<code>Stack<T></code>	Стек. Функционал такой же, как у необобщенного класса <code>Stack</code>

В пространстве имен System.Collections.Generic определены и другие классы, но они используются реже. Далее мы рассмотрим описанные ранее классы. Не все, а только лишь те, которые, на мой взгляд, являются наиболее

интересными. С остальными вы сможете познакомиться в документации по С# на официальном сайте Microsoft.

8.4. Класс ArrayList. Динамические массивы

В С# поддерживаются динамические массивы, то есть такие массивы, которые расширяются и сокращаются по мере необходимости. Стандартные массивы имеют фиксированную длину, которая не может изменяться во время выполнения программы. Другими словами, при использовании стандартных массивов программист должен задать длину массива заранее. Но количество элементов иногда неизвестно до момента выполнения программы. Например, вам нужно создать массив строк, загрузив его из файла. Вы не знаете, сколько строк будет в файле. Именно для таких ситуаций и предназначен класс `ArrayList`. В нем определяется массив переменной длины, состоящий из ссылок на объекты и способный динамически увеличивать и уменьшать свой размер.

Динамический массив `ArrayList` создается с первоначальным размером. В случае превышения размера массив будет автоматически расширен. А при удалении объектов из такого массива он автоматически сокращается. В С# коллекции класса `ArrayList` широко используются. В классе `ArrayList` реализуются интерфейсы `ICollection`, `IList`, `IEnumerable` и `ICloneable`.

Рассмотрим конструкторы класса `ArrayList`:

```
public ArrayList()  
public ArrayList(ICollection c)  
public ArrayList(int capacity)
```

Первый конструктор используется для создания пустой коллекции класса `ArrayList`. Емкость такой коллекции равна 0. При добавлении в динамический элемент массивов он будет автоматически расширяться.

Второй конструктор создает коллекцию типа `ArrayList` с количеством инициализируемых элементов, которое определяется параметром `c` и равно первоначальной емкости массива. Третий конструктор позволяет указать емкость массива целым числом. Емкость коллекции типа `ArrayList` увеличивается автоматически по мере добавления в нее элементов.

Как вы уже догадались, в классе `ArrayList` определены собственные методы. Например, произвести двоичный поиск в коллекции можно с помощью метода `BinarySearch()`. Но перед этим коллекцию желательно отсортировать методом `Sort()`. В таблице 8.6 приведены некоторые методы класса `ArrayList`.

Таблица 8.6. Некоторые методы класса ArrayList

Метод	Описание
AddRange ()	Добавляет диапазон значений из одной коллекции в конец вызывающей коллекции типа ArrayList.
BinarySearch ()	Двоичный поиск значения в вызывающей коллекции. Возвращает индекс найденного элемента или отрицательное значение, если искомое значение не найдено. Перед использованием этого метода нужно отсортировать список методом Sort().
CopyTo ()	Копирует содержимое вызывающей коллекции в массив. Массив должен быть одномерным и совместимым по типу с элементами коллекции.
FixedSize ()	Заключает коллекцию в оболочку типа ArrayList с фиксированным размером и возвращает результат. Возвращает часть вызывающей коллекции типа ArrayList. Методу нужно передать индекс элемента (параметр index), с которого начинается часть возвращаемой коллекции, и количество элементов (параметр count). Возвращаемый объект ссылается на те же элементы, что и вызывающий объект.
IndexOf ()	Возвращает индекс первого вхождения объекта в вызывающей коллекции или -1, если объект не найден.
InsertRange ()	Вставляет элементы коллекции в вызывающую коллекцию, начиная с элемента, указываемого по индексу.
ReadOnly ()	Создает коллекцию, доступную только для чтения, и возвращает результат.
RemoveRange ()	Удаляет часть вызывающей коллекции, начиная с элемента, заданного параметром index. Количество элементов задается параметром count.
Sort ()	Сортирует вызывающую коллекцию по возрастанию.

Огромное значение в классе ArrayList имеет свойство Capacity, позволяющее получать и устанавливать емкость вызывающей коллекции типа ArrayList. Свойство определено так:

```
public virtual int Capacity { get; set; }
```

Свойство Capacity позволяет получать и устанавливать емкость вызывающей коллекции типа ArrayList. Свойство содержит количество элементов, которое может содержать коллекция до очередного изменения размера. Поскольку ArrayList расширяется автоматически, задавать емкость вручную не нужно. Но это можно сделать, если количество элементов коллекции известно заранее. Благодаря этому исключаются издержки на выделение дополнительной памяти.

Иногда требуется уменьшить размер коллекции. Для этого достаточно установить меньшее значение свойства `Capacity`. Обратите внимание: оно должно быть не меньше значения свойства `Count`. Свойство `Count` определено в интерфейсе `ICollection` и содержит количество объектов, хранящихся в коллекции на данный момент.

Если вы попытаетесь установить значение свойства `Capacity` меньше значения свойства `Count`, будет сгенерировано исключение `ArgumentOutOfRangeException`. Чтобы получить такое количество элементов коллекции типа `ArrayList`, которое содержится в ней на данный момент, нужно установить значение свойства `Capacity` равным значению свойства `Count`. Также вы можете использовать метод `TrimToSize()`.

Настало время рассмотреть практический пример (листинг 8.1).

Листинг 8.1. Работа с коллекциями

```
using System;
using System.Collections;

namespace WorkingWithCollection
{
    class IntCollection
    {
        public static ArrayList NewCollection(int i)
        {
            Random ran = new Random();
            ArrayList arr = new ArrayList();

            for (int j = 0; j < i; j++)
                arr.Add(ran.Next(1, 100));
            return arr;
        }

        public static void RemoveElement(int i, int j, ref ArrayList arr)
        {
            arr.RemoveRange(i, j);
        }

        public static void AddElement(int i, ref ArrayList arr)
        {
            Random ran = new Random();
            for (int j = 0; j < i; j++)
                arr.Add(ran.Next(1, 100));
        }
    }
}
```

```

public static void Write(ArrayList arr)
{
    foreach (int a in arr)
        Console.Write("{0}\t", a);
    Console.WriteLine("");
}

}

class Program
{
    static void Main()
    {
        // Создадим новую коллекцию чисел длиной 4
        ArrayList MyCol = IntCollection.NewCollection(4);
        Console.OutputEncoding = Encoding.GetEncoding(866);
        Console.WriteLine("Моя коллекция: ");
        IntCollection.Write(MyCol);

        // Добавим еще несколько элементов
        IntCollection.AddElement(4, ref MyCol);
        Console.WriteLine("После добавления элементов: ");
        IntCollection.Write(MyCol);

        // Удалим пару элементов
        IntCollection.RemoveElement(3, 2, ref MyCol);
        Console.WriteLine("После удаления элементов: ");
        IntCollection.Write(MyCol);

        // Отсортируем теперь коллекцию
        MyCol.Sort();
        Console.WriteLine("После сортировки: ");
        IntCollection.Write(MyCol);
    }
}

```

Вывод этой программы будет примерно таким (примерно - потому что мы используем случайные элементы в диапазоне от 1 до 100 для формирования нашей коллекции, поэтому каждый запуск программы будет давать другие результаты):

Моя коллекция:

97 11 26 96
После добавления элементов:

97 11 26 96 72 33 94 31
 После удаления элементов:

97 11 26 33 94 31
 После сортировки:

11 26 31 33 94 97

Изначально была создана коллекция с числами 97, 11, 26 и 96. Для создания коллекции мы использовали метод `NewCollection()`, которому передали число 4 - количество элементов в новой коллекции. Сами же элементы коллекции создаются генератором случайных чисел:

```
Random ran = new Random();
ArrayList arr = new ArrayList();

for (int j = 0; j < i; j++)
    arr.Add(ran.Next(1, 100));
return arr;
```

Добавление элементов осуществляется методом `Add()`. За один раз в коллекцию можно добавить один элемент. Мы же создали метод `AddElement()`, который добавляет в динамический массив указанное количество случайных элементов. Мы добавили 4 элемента.

Далее методом `RemoveRange()` мы удаляем 2 элемента, начиная с третьего. Элементы 96 и 72 удалены.

В заключение этого примера мы сортируем массив по возрастанию. Результат сортировки приведен выше.

8.5. Хеш-таблица. Класс `HashTable`

Информация в хеш-таблице хранится с помощью механизма, называемого хешированием. Для создания хеш-таблицы используется класс `HashTable`.

При хешировании для определения уникального значения, называемого хеш-кодом, используется информационное содержимое специального ключа. В результате хеш-код служит в качестве индекса, по которому в таблице хранятся искомые данные, соответствующие заданному ключу.

Сам хеш-код недоступен программисту, а преобразование ключа в хеш-код осуществляется автоматически. Преимущество данного способа (хеширования) в том, что оно обеспечивает постоянство времени выполнения операций поиска, извлечения и установки значений независимо от размера массива данных.

В классе `HashTable` реализованы следующие интерфейсы:

- `ICloneable`;
- `ICollection`;
- `IDictionary`;
- `IDeserializationCallback`;
- `IEnumerable`.

Конструкторы класса `HashTable` выглядят так:

```
public Hashtable()
public Hashtable(IDictionary d)
public Hashtable(int capacity)
public Hashtable(int capacity, float loadFactor)
```

Первая форма создает объект класса `HashTable` по умолчанию. Во второй форме объект типа `HashTable` инициализируется элементами из коллекции `d`. Третья форма создает объект, инициализируемый с учетом емкости коллекции, заданной параметром **capacity**. Четвертая форма создает объект типа `Hashtable`, который инициализируется с учетом емкости **capacity** и коэффициента заполнения **loadFactor**.

Параметр **loadFactor** может принимать значения от 0.1 до 1.0. Он определяет степень заполнения хеш-таблицы до увеличения ее размера. В частности, таблица расширяется, если количество элементов оказывается больше емкости таблицы, умноженной на коэффициент заполнения. Конструкторы, не принимающие параметр **loadFactor**, считают, что этот параметр равен 1.0.

В классе `Hashtable` определяется ряд собственных методов, помимо тех, что уже объявлены в интерфейсах, которые в нем реализуются. Рассмотрим некоторые часто используемые методы (табл. 8.7).

Таблица 8.7. Некоторые часто используемые методы класса `Hashtable`

Метод	Описание
<code>ContainsKey()</code>	Если в вызывающей коллекции типа <code>HashTable</code> содержится ключ, метод возвращает <code>true</code> , в противном случае - <code>false</code> .
<code>ContainsValue()</code>	Если в вызывающей коллекции типа <code>HashTable</code> содержится значение, метод возвращает <code>true</code> , в противном случае - <code>false</code> .
<code>GetEnumerator()</code>	Возвращает для вызывающей коллекции типа <code>Hashtable</code> перечислитель типа <code>IDictionaryEnumerator</code>
<code>Synchronized()</code>	Возвращает синхронизированный вариант коллекции типа <code>Hashtable</code> , которая передана как параметр

Особую роль в классе `HashTable` играют свойства `Keys` и `Values`, содержащие ключи и значения, соответственно:

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

Теперь давайте рассмотрим пример создания и использования `Hash-таблицы` (лист. 8.2).

Листинг 8.2. Создание и использование Hash-таблицы

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            // Создаем хеш-таблицу
            Hashtable ht = new Hashtable();

            // Добавим несколько записей
            ht.Add("den", "98765456546");
            ht.Add("user", "45837685768");
            ht.Add("root", "ddfdf3545");

            // Получаем коллекцию ключей
            ICollection keys = ht.Keys;

            foreach (string s in keys)
                Console.WriteLine(s + ": " + ht[s]);

            Console.ReadLine();
        }
    }
}
```

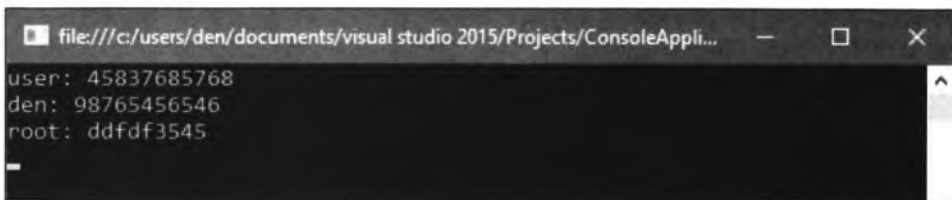


Рис. 8.1. Пример работы с хеш-таблицей

8.6. Создаем стек. Классы Stack и Stack<T>

Наверное, нет ни одного программиста, который не был бы знаком со стеком. Стек - это контейнер, работающий по принципу LIFO (Last In First Out), то есть последним зашел, первым вышел.

Добавление элемента в стек осуществляется методом `Push()`, а извлечение последнего добавленного элемента - методом `Pop()`.

В C# определен класс коллекции с именем `Stack`. Он-то и реализует стек. Конструкторы этого класса определены так:

```
public Stack()
public Stack(int initialCapacity)
public Stack(ICollection col)
```

Первая форма создает пустой стек, вторая форма - тоже создает пустой стек, но задает его первоначальный размер (параметр `initialCapacity`). Третья форма создает стек, содержащий элементы коллекции `col`. Емкость этого стека равна количеству элементов в коллекции `col`.

В классе `Stack` определены различные методы. С методами `Pop()` и `Push()` вы уже знакомы. Но поговорим о них подробнее. Метод `Push()` помещает элемент на вершину стека. А для того чтобы извлечь и удалить объект из вершины стека, вызывается метод `Pop()`. Если же объект требуется только извлечь, но не удалить из вершины стека, то вызывается метод `Peek()`. А если вызвать метод `Pop()` или `Peek()`, когда вызывающий стек пуст, то сгенерируется исключение `InvalidOperationException`.

Кроме описанных методов класс `Stack` содержит свойство `Count` и метод `Contains()`. Свойство `Count` возвращает количество элементов в стеке, а метод `Contains()` проверяет наличие элемента в стеке и возвращает `true`, если элемент находится в стеке.

Класс `Stack<T>` является обобщенным вариантом класса `Stack`. В этом классе реализуются интерфейсы `Collection`, `IEnumerable` и `IEnumerable<T>`. Кроме того, в классе `Stack<T>` непосредственно реализуются методы `Clear()`, `Contains()` и `CopyTo()`, определенные в интерфейсе `ICollection<T>`. А методы `Add()` и `Remove()` в этом классе не поддерживаются, как, впрочем, и свойство `IsReadOnly`. Коллекция класса `Stack<T>` имеет динамический характер, расширяясь по мере необходимости, чтобы вместить все элементы, которые должны в ней храниться.

В листинге 8.3 содержится пример работы со стеком.

Листинг 8.3. Пример использования класса Stack

```

using System;
using System.Collections.Generic;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            var MyStack = new Stack<char>();
            MyStack.Push('A');
            MyStack.Push('B');
            MyStack.Push('C');

            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.WriteLine("Содержимое стека: ");

            foreach (char s in MyStack)
                Console.Write(s);
            Console.WriteLine("\n");

            while (MyStack.Count > 0)
            {
                Console.WriteLine(MyStack.Pop());
            }

            if (MyStack.Count == 0)
                Console.WriteLine("Стек пуст!");
        }
    }
}

```

Вывод программы будет таким:

My Stack contains:

CBA

C

B

A

Stack is empty

8.7. Очередь. Классы Queue и Queue<T>

Очередь - это контейнер, работающий по принципу FIFO (First In First Out), то есть первым вошел, первым вышел. Элемент, вставленный в очередь первым, первым же и читается. Примером очереди в программировании может

послужить любая очередь в реальном мире. Если вы пришли первым и заняли очередь, то первым и будете обслужены.

Очередь реализуется с помощью классов Queue из пространства имен System.Collections и Queue<T> из пространства имен System.Collections.Generic.

Конструкторы класса Queue выглядят так:

```
public Queue()
public Queue(int capacity)
public Queue(int capacity, float growFactor)
public Queue(ICollection col)
```

Как и в случае со стеком, первая форма создает пустую очередь, вторая - тоже пустую, но задает ее первоначальный размер. Третья форма позволяет указать начальную емкость очереди и фактор роста (допустимые значения от 1.0 до 10.0). Четвертая форма создает очередь из элементов коллекции col. Все конструкторы, не позволяющие задать параметр growFactor, считают, что фактор роста равен 2.0.

Класс Queue<T> содержит такие конструкторы:

```
public Queue()
public Queue(int capacity)
public Queue(IEnumerable<T> collection)
```

Первая форма создает пустую очередь, емкость очереди выбирается по умолчанию. Вторая форма позволяет задать емкость создаваемой очереди. Третья форма создает очередь, содержащую элементы заданной коллекции collection.

Члены класса Queue представлены в таблице 8.8.

Таблица 8.8. Члены класса Queue

Член класса	Описание
Count	Свойство Count возвращает количество элементов очереди
Enqueue()	Метод добавляет элемент в конец очереди
Dequeue()	Читает и удаляет элемент из головы очереди. Если на момент вызова метода очередь пуста, генерируется исключение InvalidOperationException.
Peek()	Читает элемент из головы очереди, но не удаляет его
TrimExcess()	Изменяет емкость очереди. Метод Dequeue() удаляет элемент из очереди, но не изменяет ее емкости. TrimExcess() позволяет избавиться от пустых элементов в начале очереди.

Пример работы с очередью приведен в листинге 8.4.

Листинг 8.4. Работаем с очередью

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            Queue<int> MyQueue = new Queue<int>();
            Random ran = new Random();

            for (int i = 0; i < 20; i++)
                MyQueue.Enqueue(ran.Next(1, 100));

            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.WriteLine("Моя очередь:");
            foreach (int i in MyQueue)
                Console.Write("{0} ", i);

            Console.ReadLine();
        }
    }
}
```

8.8. Связный список. Класс LinkedList<T>

Рассмотрим рис. 8.2, на котором изображен типичный двухсвязный список - структура, часто используемая в программировании. У каждого элемента списка есть два указателя - на следующий (Next) и предыдущий элемент (Prev). Если нет следующего (или предыдущего) элемента, то указатель содержит значение null. Кроме указателей каждый элемент содержит значение (value).

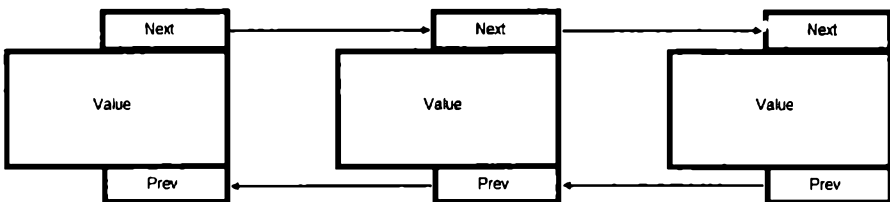


Рис. 8.2. Связный список

Ранее, в языке Си, нужно было создавать структуру связанного списка вручную. Сейчас же достаточно использовать уже готовый класс `LinkedList` и его методы.

Преимущество связанных списков в том, что операция вставки элемента в середину выполняется очень быстро. При этом только ссылки `Next` (следующий) предыдущего элемента и `Previous` (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент. В классе `List<T>` при вставке нового элемента все последующие должны быть сдвинуты.

К недостаткам связанных списков можно отнести довольно медленный поиск элементов, находящихся в центре списка.

Класс `LinkedList<T>` содержит члены `First` (используется для доступа к первому элементу списка), `Last` (доступ к последнему элементу), а также методы для работы с элементами. В классе `LinkedList<T>` реализуются интерфейсы `ICollection`, `ICollection<T>`, `IEnumerable`, `IEnumerable<T>`, `ISerializable` и `IDeserializationCallback`. В двух последних интерфейсах поддерживается сериализация списка. В классе `LinkedList<T>` определяются два конструктора:

```
public LinkedList()
public LinkedList(IEnumerable<T> collection)
```

Первый конструктор создает пустой связный список, второй - список, который инициализируется элементами из заданной коллекции.

Методы класса `LinkedList<T>` представлены в таблице 8.9.

Таблица 8.9. Методы класса `LinkedList<T>`

Метод	Описание
<code>AddFirst()</code> , <code>AddLast()</code>	Добавляют элемент, соответственно, в начало и конец списка
<code>AddAfter()</code>	Добавляет в список узел непосредственно после указанного узла. Указываемый узел не должен быть пустым (<code>null</code>). Метод возвращает ссылку на узел, содержащий значение.
<code>AddBefore()</code>	Аналогичен <code>AddAfter()</code> , но добавляет узел до указанного узла.
<code>Find()</code>	Возвращает ссылку на первый узел в списке, имеющий передаваемое значение. Если искомое значение отсутствует в списке, то возвращается <code>null</code> .

Remove ()	Используется для удаления из списка первого узла, который содержит переданное методу значение value. Возвращает true, если узел со значением был обнаружен и удален, в противном случае возвращается значение false.
-----------	--

Как обычно, настало время для примера. В листинге 8.5 приводится пример создания связанного списка и вывода его в двух направлениях - в прямом и обратном. Обратите внимание, как построены циклы `for`. В качестве начального значения при прямом обходе мы используем `First`, а затем присваиваем узлу значение `Next` (идем вперед). При обратном обходе мы начинаем с `Last` и узлу при каждой итерации присваиваем значение `Previous`, то есть идем назад.

Листинг 8.5. Пример прямого и обратного обхода связанного списка

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            // Создадим связанный список
            LinkedList<string> l = new LinkedList<string>();

            // Добавим несколько элементов
            l.AddFirst("Apple");
            l.AddFirst("Banana");
            l.AddFirst("Pear");
            l.AddFirst("Orange");
            l.AddFirst("Peach");

            // Элементы в прямом порядке
            LinkedListNode<string> node;
            Console.WriteLine("Direct Order: ");
            for (node = l.First; node != null; node = node.Next)
                Console.Write(node.Value + "\t");

            Console.WriteLine();

            // Элементы в обратном порядке
            Console.WriteLine("Reverse order: ");
            for (node = l.Last; node != null; node = node.Previous)
```

```

        Console.WriteLine(node.Value + "\t");
        Console.ReadLine();
    }
}

```

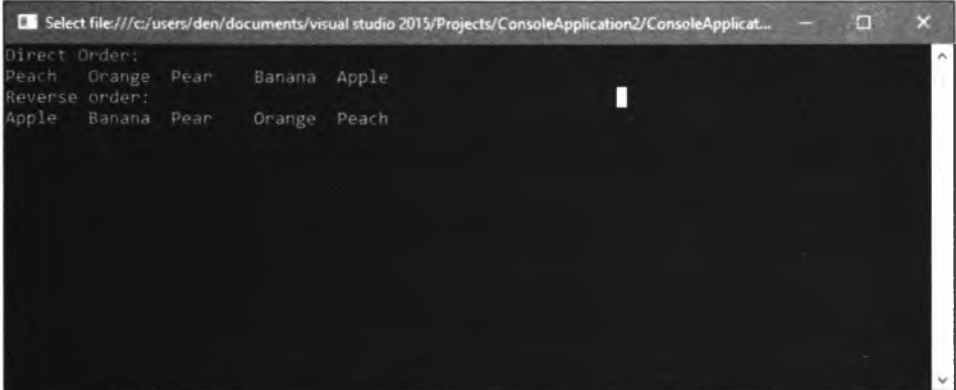


Рис. 8.3. Пример прямого и обратного обхода связного списка

8.9. Сортированный список. Класс `SortedList<TKey, TValue>`

Еще один вариант списка - сортированный по ключу список. Для его создания можно воспользоваться классом `SortedList<TKey, TValue>`. Данный класс сортирует элементы на основе значения ключа. При этом вы можете использовать любые типы значения и ключа.

В классе `SortedList<TKey, TValue>` реализуются следующие интерфейсы:

- `IDictionary`;
- `IDictionary<TKey, TValue>`;
- `ICollection`;
- `ICollection<KeyValuePair<TKey, TValue>>`;
- `IEnumerable`;
- `IEnumerable<KeyValuePair<TKey, TValue>`.

Размер сортированного списка увеличивается автоматически - по мере необходимости. Класс `SortedList<TKey, TValue>` похож на класс `SortedListDictionary<TKey, TValue>`, но он более эффективно расходует память.

Рассмотрим конструкторы класса `SortedList<TKey, TValue>`:

```
public SortedList()
public SortedList(IDictionary<TKey, TValue> dictionary)
public SortedList(int capacity)
public SortedList(IComparer<TK> comparer)
```

Первая форма создает пустой список. Вторая - создает отсортированный список, элементы берутся из словаря **dictionary**. Третья форма с помощью параметра **capacity** задает емкость отсортированного списка. Последняя форма позволяет указать способ сравнения объектов, которые находятся в списке (для этого используется параметр **comparer**).

Методы и прочие члены класса `SortedList<TKey, TValue>` приведены в таблице 8.10.

Таблица 8.10. Методы класса `SortedList<TKey, TValue>`

Член класса	Описание
<code>Add()</code>	Добавляет в список пару "ключ-значение". Если ключ уже находится в списке, то генерируется исключение <code>ArgumentException</code>
<code>ContainsKey()</code>	Возвращает значение <code>true</code> , если вызывающий список содержит объект <code>key</code> в качестве ключа, в противном случае - <code>false</code>
<code>ContainsValue()</code>	Возвращает значение <code>true</code> , если вызывающий список содержит значение <code>value</code> ; в противном случае - <code>false</code>
<code>GetEnumerator()</code>	Возвращает перечислитель для вызывающего словаря
<code>IndexOfKey()</code> , <code>IndexOfValue()</code>	Возвращает индекс ключа или первого вхождения значения в вызывающем списке. Если ключ или значение не найдены, возвращается значение -1
<code>Remove()</code>	Удаляет из списка пару "ключ-значение" по указанному ключу <code>key</code> . Если удаление успешно, возвращается <code>true</code> , если нет - <code>false</code>
<code>TrimExcess()</code>	Сокращает избыточную емкость вызывающей коллекции в виде отсортированного списка.
<code>Capacity</code>	Получает или устанавливает емкость списка
<code>Comparer</code>	Задаёт метод сравнения для вызывающего списка
<code>Keys</code>	Коллекция ключей
<code>Values</code>	Коллекция значений

Пример программы, работающей с отсортированным списком, приведен в листинге 8.6.

Листинг 8.6. Работа с отсортированным списком

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            // Создадим коллекцию сортированного списка

            SortedList<string, string> CarInfo = new
            SortedList<string, string>();

            // Добавление элементов
            CarInfo.Add("Audi", "2015");
            CarInfo.Add("Toyota", "2016");
            CarInfo.Add("BMW", "2015");
            CarInfo.Add("Renault", "2014");
            CarInfo.Add("Lexus", "2016");

            // Коллекция ключей
            ICollection<string> keys = CarInfo.Keys;

            Console.OutputEncoding = Encoding.GetEncoding(866);

            // Теперь используем ключи для получения значений
            foreach (string s in keys)
            Console.WriteLine("Марка: {0}, Год: {1}", s, CarInfo[s]);
            Console.ReadLine();
        }
    }
}
```

8.10. Словарь. Класс Dictionary<TKey, TValue>

Словарь (dictionary) - это сложная структура данных, обеспечивающая доступ к элементам по ключу. Основная особенность словаря - быстрый поиск элемента по ключу. Также вы можете быстро добавлять и удалять элементы,

подобно тому, как вы это делаете в списке `List<T>`, но гораздо эффективнее, поскольку нет необходимости смещения последующих элементов в памяти.

Тип, используемый в качестве ключа словаря, должен переопределять метод `GetHashCode()` класса `Object`. Всякий раз, когда класс словаря должен найти местоположение элемента, он вызывает метод `GetHashCode()`.

К методу `GetHashCode()` есть следующие требования:

- Не должен генерировать исключений.
- Должен выполняться быстро, не требуя значительных вычислительных затрат.
- Разные объекты могут возвращать одно и то же значение.
- Один и тот же объект должен возвращать всегда одно и то же значение.
- Должен использовать как минимум одно поле экземпляра.
- Хеш-код не должен изменяться на протяжении времени существования объекта.

Помимо реализации `GetHashCode()` тип ключа также должен реализовывать метод `IEquatable<T>.Equals()` либо переопределять метод `Equals()` класса `Object`. Поскольку разные объекты ключа могут возвращать один и тот же хеш-код, метод `Equals()` используется при сравнении ключей словаря.

Конструкторы класса `Dictionary<TKey, TValue>` выглядят так:

```
public Dictionary()  
public Dictionary(IDictionary<TKey, TValue> dictionary)  
public Dictionary(int capacity)
```

Первый конструктор создает пустой словарь, его емкость выбирается по умолчанию. Второй конструктор создает словарь с указанным количеством элементов **dictionary**. Третий конструктор позволяет указать емкость словаря (параметр **capacity**).

В классе `Dictionary<TKey, TValue>` реализуются следующие интерфейсы:

- `IDictionary`;
- `IDictionary<TKey, TValue>`;
- `ICollection`;
- `ICollection<KeyValuePair<TKey, TValue>>`;
- `IEnumerable`;

- `IEnumerable<KeyValuePair<TKey, TValue>>;`
- `ISerializable;`
- `IDeserializationCallback.`

Члены класса `Dictionary<TKey, TValue>` приведены в таблице 8.11.

Таблица 8.11. Члены класса `Dictionary<TKey, TValue>`

Член класса	Описание
<code>Add()</code>	Добавляет в словарь пару "ключ-значение". Пара определяется параметрами <code>key</code> и <code>value</code> . Если ключ <code>key</code> уже находится в словаре, то генерируется исключение <code>ArgumentException</code> . Значение самого же ключа не изменяется.
<code>ContainsKey()</code>	Возвращает <code>true</code> , если вызывающий словарь содержит объект <code>key</code> в качестве ключа, иначе - возвращает <code>false</code> .
<code>ContainsValue()</code>	Возвращает логическое значение <code>true</code> , если вызывающий словарь содержит значение <code>value</code> ; в противном случае — <code>false</code>
<code>Remove()</code>	Удаляет ключ <code>key</code> из словаря. В случае успеха возвращается <code>true</code> , иначе - <code>false</code> (если ключ отсутствует в словаре)
<code>Comparer</code>	Получает метод сравнения для вызывающего словаря
<code>Keys</code>	Получает коллекцию ключей
<code>Values</code>	Получает коллекцию значений

В классе `Dictionary<TKey, TValue>` реализуется приведенный ниже индексатор, определенный в интерфейсе `IDictionary<TKey, TValue>`:

```
public TValue this[TKey key] { get; set; }
```

Данный индексатор служит для получения и установки значения элемента коллекции, а также для добавления в коллекцию нового элемента. В качестве индекса в данном случае служит ключ элемента, а не сам индекс. При перечислении коллекции типа `Dictionary<TKey, TValue>` из нее возвращаются пары "ключ-значение" в форме структуры `KeyValuePair<TKey, TValue>`. В этой структуре определяются два поля:

```
public TKey Key;
public TValue Value;
```

Пример программы, использующей словарь, приведен в листинге 8.7.

Листинг 8.7. Работа со словарем

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

```
namespace ConsoleApplication1
{
    class CarInfo
    {
        // Реализуем словарь
        public static Dictionary<int, string> MyDic(int i)
        {
            Dictionary<int, string> dic = new Dictionary<int,string>();

            Console.WriteLine("Введите марку машины: \n");

            string s;

            for (int j = 0; j < i; j++)
            {
                Console.Write("Марка{0} --> ",j);
                s = Console.ReadLine();
                dic.Add(j, s);
            }
            return dic;
        }
    }

    class Program
    {
        static void Main()
        {
            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.Write("Сколько машин добавить? ");
            try
            {
                int i = int.Parse(Console.ReadLine());
                Dictionary<int, string> dic = CarInfo.MyDic(i);

                // Получаем ключи
                ICollection<int> keys = dic.Keys;

                Console.WriteLine("Словарь: ");
                foreach (int j in keys)
                    Console.WriteLine("ID -> {0}  Марка -> {1}",j, dic[j]);
            }
            catch (FormatException)
            {
                Console.WriteLine("Ошибка! Исключение формата");
            }
        }
    }
}
```

```

        Console.ReadLine();
    }
}

```

8.11. Сортированный словарь: класс `SortedDictionary<TKey, TValue>`

Отсортированный вариант словаря представляет собой дерево бинарного поиска, в котором все элементы отсортированы по ключу. Класс называется `SortedDictionary<TKey, TValue>`.

Ключ, точнее, его тип, должен реализовать интерфейс `IComparable<TKey>`. Если тип ключа не относится к сортируемым, вы можете реализовать `IComparer<TKey>` и указать его в качестве аргумента конструктора сортированного словаря.

Классы `SortedDictionary<TKey, TValue>` и `SortedList<TKey, TValue>` довольно похожи. Но `SortedList<TKey, TValue>` реализован в виде списка, основанного на массиве, а `SortedDictionary<TKey, TValue>` реализован как словарь. Именно поэтому эти классы обладают разными характеристиками, а именно `SortedList<TKey, TValue>` использует меньше памяти, чем `SortedDictionary<TKey, TValue>`. Также `SortedDictionary<TKey, TValue>` быстрее вставляет и удаляет элементы.

В классе `SortedDictionary<TKey, TValue>` предоставляются также следующие конструкторы:

```

public SortedDictionary()
public SortedDictionary(IDictionary<TKey, TValue> dictionary)
public SortedDictionary(IComparer<TKey> comparer)
public SortedDictionary(IDictionary<TKey, TValue> dictionary,
    IComparer<TKey> comparer)

```

Первый конструктор создает пустой словарь, второй - словарь с указанным количеством элементов **dictionary**. Третий конструктор позволяет указывать с помощью параметра **comparer** типа `IComparer` способ сравнения, используемый для сортировки. Четвертая форма конструктора позволяет инициализировать словарь, помимо указания способа сравнения.

В классе `SortedDictionary<TKey, TValue>` реализуются следующие интерфейсы:

- `IDictionary`;
- `IDictionary<TKey, TValue>`;

- ICollection;
- ICollection<KeyValuePair<TKey, TValue>>;
- IEnumerable;
- IEnumerable<KeyValuePair<TKey, TValue>>.

Методы отсортированного словаря аналогичны методам обычного, а именно поддерживаются методы Add(), ContainsKey(), ContainsValue(), Remove().

В классе SortedDictionary<TKey, TValue> реализуется приведенный ниже индекатор, определенный в интерфейсе IDictionary<TKey, TValue>:

```
public TValue this[TKey key] { get; set; }
```

Пример использования отсортированного словаря приведен в листинге 8.8.

Листинг 8.8. Использование отсортированного словаря

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

namespace ConsoleApplication1
{
    class CarInfo
    {
        public static SortedDictionary<string, string> MyDic(int i)
        {
            SortedDictionary<string, string> dic = new SortedD
            ictionary<string, string>();

            string s2, s1;

            for (int j = 0; j < i; j++)
            {
                Console.Write("Ключ: ");
                s1 = Console.ReadLine();
                Console.WriteLine("Карта: ");
                Console.Write("Карта{0} --> ", j);
                s2 = Console.ReadLine();
                dic.Add(s1, s2);
            }
            return dic;
        }
    }

    class Program
```

```

{
    static void Main()
    {
        Console.OutputEncoding = Encoding.GetEncoding(866);
        Console.Write("Сколько машин добавить? ");
        try
        {
            int i = int.Parse(Console.ReadLine());

            SortedDictionary<string, string> d = CarInfo.MyDic(i);

            // Получить коллекцию ключей
            ICollection<string> keys = d.Keys;

            Console.WriteLine("Отсортированный словарь: ");
            foreach (string j in keys)
                Console.WriteLine("ID -> {0} Марка -> {1}", j, d[j]);
        }
        catch (FormatException)
        {
            Console.WriteLine("Ошибка!");
        }

        Console.ReadLine();
    }
}

```

8.12. Множества: классы HashSet<T> и SortedSet<T>

Настало время поговорить о множествах (set). В составе .NET имеются два класса - HashSet<T> и SortedSet<T>. Оба они реализуют интерфейс ISet<T>. Класс HashSet<T> содержит неупорядоченный список различающихся элементов, а в SortedSet<T> элементы упорядочены. То есть первый класс - это просто множество, а второй - отсортированное множество.

Интерфейс ISet<T> предоставляет методы, используемые для выполнения основных операций над множеством.

В классе HashSet<T> определены следующие конструкторы:

```

public HashSet ()
public HashSet(IEnumerable<T> collection)
public HashSet(IEqualityCompare comparer)
public HashSet(IEnumerable<T> collection, IEqualityCompare
comparer)

```

Первая форма создает пустое множество, вторая форма - создает множество, состоящее из элементов коллекции **collection**. Третий конструктор позволяет указывать способ сравнения, указав параметр **comparer**. Последняя форма создает множество, состоящее из элементов заданной коллекции **collection**, и использует метод сравнения **comparer**.

Конструкторы класса `SortedSet<T>` такие же:

```
public SortedSet()
public SortedSet(IEnumerable<T> collection)
public SortedSet(IComparer comparer)
public SortedSet(IEnumerable<T> collection, IComparer
comparer)
```

В `SortedSet<T>`, помимо прочих свойств, определены дополнительные свойства:

```
public IComparer<T> Comparer { get; }
public T Max { get; }
public T Min { get; }
```

Пример использования коллекции:

```
SortedSet<int> ss = new SortedSet<int>();
ss.Add(7);
ss.Add(6);
ss.Add(9);
ss.Add(1);
```

```
foreach (int j in ss)
    Console.Write(j + " ");
```

Для объединения множеств используется метод `UnionWith()`, в качестве параметра ему нужно передать имя второго множества. Для вычитания множеств используется метод `ExceptWith()`. Оставить в двух множествах только уникальные элементы (чтобы не было пересечения этих множеств) можно методом `SymmetricExceptWith()`. Рассмотрим, как работают эти методы:

```
// создадим второе множество
SortedSet<int> sb = new SortedSet<int>();
```

```
sb.Add(7);
sb.Add(2);
sb.Add(3);
sb.Add(1);
```

```
// Пересечение
```

```

ss.ExceptsWith(sb);
foreach (int j in ss)
    Console.Write(j + " ");

// Объединение
ss.UnionWith(sb);
foreach (int j in ss)
    Console.Write(j + " ");

// Исключаем одинаковые элементы
ss.SymmetricExceptWith(sb);
foreach (int j in ss)
    Console.Write(j + " ");

```

8.13. Реализация интерфейса **IComparable**

Представим, что вы создали некий собственный класс и создали коллекцию, содержащую объекты этого класса. Что, если вам понадобилось отсортировать эту коллекцию? Для сортировки коллекции, заполненной объектами пользовательского типа, нужно реализовать интерфейс **IComparable**, то есть указать компилятору, как сравнивать объекты.

Если требуется отсортировать объекты, хранящиеся в необобщенной коллекции, то для этой цели придется реализовать необобщенный вариант интерфейса **IComparable**. В этом варианте данного интерфейса определяется только один метод, **CompareTo()**, который определяет порядок выполнения самого сравнения. Ниже приведена общая форма объявления метода **CompareTo()**:

```
int CompareTo(object obj)
```

Данный метод должен возвращать положительное значение, если значение вызывающего объекта больше, чем у объекта, с которым производится сравнение. В противном случае, если значение вызывающего объекта меньше - возвращается отрицательное значение. Если объекты равны, то возвращается 0. Представим, что в классе есть поле **price**, по которому мы и будем сравнивать объекты:

```

public int CompareTo(SomeClass obj)
{
    if (this.price > obj.price)
        return 1;
    if (this.price < obj.price)
        return -1;
    else

```

```
        return 0;
    }
}
```

Если нужно отсортировать объекты, хранящиеся в обобщенной коллекции, то для этой цели придется реализовать обобщенный вариант интерфейса `Comparable<T>`. В этом варианте интерфейса `Comparable` определяется приведенная ниже обобщенная форма метода `CompareTo()`:

```
int CompareTo(T other)
```

8.14. Перечислители

К элементам коллекции иногда приходится обращаться циклически - ранее приводились примеры цикла **foreach** как один из способов циклически обратиться ко всем элементам коллекции. Второй способ - использование перечислителя. Перечислитель - это объект, реализующий необобщенный интерфейс `IEnumerator` или обобщенный `IEnumerator<T>`.

В интерфейсе `IEnumerator` определяется одно свойство, `Current`. Такое же свойство есть и в обобщенной версии интерфейса. Свойства `Current` определяются так:

```
object Current { get; }
object Current { get; }
```

В обеих формах свойства `Current` получается текущий перечисляемый элемент коллекции. Свойство `Current` доступно только для чтения.

Доступные методы (как в необобщенной, так и в обобщенной версиях):

- `MoveNext()` - смещает текущее положение перечислителя к следующему элементу коллекции. Возвращает `true`, если следующий элемент коллекции доступен, или `false`, если был достигнут конец.
- `Reset()` - после вызова метода перечислитель перемещается в начало коллекции.

Прежде чем получить доступ к коллекции с помощью перечислителя, необходимо получить его. В каждом классе коллекции для этой цели предоставляется метод `GetEnumerator()`, возвращающий перечислитель в начало коллекции. Используя этот перечислитель, можно получить доступ к любому элементу коллекции по очереди.

Рассмотрим, как использовать перечислитель:

```
List<int> MyList = new List<int>();
Random ran = new Random();
```

```
// Заполняем список случайными значениями от 1 до 100
for (int i = 0; i < 20; i++)
    MyList.Add(ran.Next(1, 100));

// Используем перечислитель для вывода элементов списка
IEnumerator<int> enum = MyList.GetEnumerator();
while (enum.MoveNext())
    Console.Write(num.Current + " ");
```

Если нужно вывести список повторно, нужно сбросить перечислитель методом `Reset` и снова вывести список циклом `while`:

```
enum.Reset();
```

8.15. Реализация интерфейсов `IEnumerable` и `IEnumerator`

Представим, что вам нужно создать класс, объекты которого будут перебираться в цикле `foreach`, в классе нужно реализовать интерфейсы `IEnumerator` и `IEnumerable`. То есть для того чтобы обратиться к объекту определяемого пользователем класса в цикле `foreach`, необходимо реализовать интерфейсы `IEnumerator` и `IEnumerable` в их обобщенной или необобщенной форме.

Реализовать данные интерфейсы довольно просто, что и показано в листинге 8.9.

Листинг 8.9. Реализация интерфейсов `IEnumerator` и `IEnumerable`

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
    class MyBytes : IEnumerable, IEnumerator
    {
        byte[] bytes = { 1, 3, 5, 7 };
        byte index = -1;

        // Интерфейс IEnumerable
        public IEnumerator GetEnumerator()
        {
            return this;
        }

        // Интерфейс IEnumerator
        public bool MoveNext()
```

```

        {
            if (index == bytes.Length - 1)
            {
                Reset();
                return false;
            }
            index++;
            return true;
        }
// Метод, сбрасывающий перечислитель
public void Reset()
{
    index = -1;
}
public object Current
{
    get
    {
        return bytes[index];
    }
}
}
class Program
{
    static void Main()
    {
        MyBytes mb = new MyBytes();

        foreach (int j in mb)
            Console.Write(j+" ");

    }
}

```

8.16. Итераторы. Ключевое слово **yield**

В прошлом разделе мы реализовали интерфейсы `IEnumerator` и `IEnumerable`. Как было показано, это совсем несложно. Но еще проще воспользоваться итератором. Итератор - это метод, оператор или аксессор, который по очереди возвращает члены последовательности объектов - с ее начала до конца. После того как вы реализуете итератор, вы сможете обращаться к объектам определяемого пользователем класса в цикле **foreach**.

Создать итератор в C# можно с помощью ключевого слова **yield**. Данное ключевое слово является контекстным, то есть оно имеет специальное зна-

чение только в блоке итератора. Вне этого блока оно может быть использовано аналогично любому другому идентификатору. Следует особо подчеркнуть, что итератор не обязательно должен опираться на массив или коллекцию другого типа. Он должен просто возвращать следующий элемент из совокупности элементов.

Синтаксис итератора следующий:

```
public IEnumerable имя_итератора(список_параметров) {
    // ...
    yield return obj;
}
```

Здесь `имя_итератора` - конкретное имя метода, `список_параметров` - параметры, передаваемые методу итератора, `obj` - следующий объект, возвращаемый итератором. Как только именованный итератор будет создан, его можно будет использовать везде, где он нужен, например, в цикле **foreach**.

Пример итератора:

```
class MyClass {
    int limit = 0;
    public MyClass(int limit) { this.limit = limit; }

    public IEnumerable<int> CountFrom(int start)
    {
        for (int i = start; i <= limit; i++)
            yield return i;
    }
}
```

Для выхода из итератора по какому-нибудь условию может использоваться такая конструкция:

```
if (условие) yield break;
```

Глава 9.

Конфигурация сборок .NET



Ранее мы создавали только “автономные” приложения, где вся логика размещалась внутри одного исполняемого exe-файла. Однако платформа .NET предоставляет возможность повторного использования двоичного кода. А это означает, что код вашего приложения может размещаться внутри нескольких внешних сборок кода, называемых библиотеками кода. В этой главе мы поговорим о создании и развертывании сборок в .NET.

9.1. Специальные пространства имен

До этого момента мы разрабатывали простые демонстрационные программы, в которых использовались существующие пространства имен в мире .NET (чаще всего мы использовали пространство имен System). Однако при создании сложных собственных приложений вам будет удобно сгруппировать свои типы в специальные пространства имен. В C# для этого используется ключевое слово **namespace**, с которым вы уже сталкивались и не один раз.

Создание собственных пространств имен очень важно, поскольку у других разработчиков, которые будут использовать написанный вами код, должна быть возможность импортировать эти пространства имен для использования содержащихся в них типов.

Что ж, приступим к практике. Создайте новый проект типа ConsoleApplication с именем MyNamespaces.

Далее представим, что мы хотим разработать пространство имен MyPets, содержащее классы Cat и Dog. Эти классы мы сгруппируем вместе и разместим внутри сборки MyNamespaces.exe в уникальном пространстве имен MyPets.

Осуществить задуманное можно двумя способами. Первый способ заключается в том, что мы определим все наши классы в единственном файле C# (PetsLib.cs), например:

```
// Файл PetsLib.cs
using System;
namespace MyPets
{
    public class Cat
    {
        public string Name;
```

```

        public byte Age;
        public byte Weight;
    }
    public class Dog
    {
        public string Name;
        public byte Age;
        public byte Weight;
    }
}

```

Второй способ заключается в том, что одно пространство имен мы разделим на несколько файлов кода C#. Дабы обеспечить размещение типов в одной и той же логической группе, нужно просто упаковать определения всех классов в контекст одного и того же пространства имен:

```

// Файл Cat.cs
using System;
namespace MyPets
{
    public class Cat
    {
        public string Name;
        public byte Age;
        public byte Weight;
    }
}
// Файл Dog.cs
namespace MyPets
{
    public class Dog
    {
        public string Name;
        public byte Age;
        public byte Weight;
    }
}

```

Как видите, пространство имен играет роль контейнера для наших классов. Если в другом пространстве имен, а именно в `MyNamespaces`, возникает необходимость импортировать классы, определенные внутри данного пространства имен, нужно использовать ключевое слово **using**:

```

using System;
using MyPets;

namespace MyNamespaces

```

```

{
    static void Main(string[] args)
    {
        Cat Bagira = new Cat();
        Dog Rex = new Dog();
    }
}

```

В этом примере мы предполагаем, что файлы с кодом (PetsLib.cs или отдельно два файла Cat.cs и Dog.cs) являются частью проекта MyNamespaces. Все они будут использоваться для компиляции единственной сборки .NET.

Если же пространство имен MyPets определено внутри внешней сборки, то для успешной компиляции нужно добавить ссылку на эту библиотеку (сборку). Далее будет показано, как это сделать.

9.2. Уточненные имена или конфликты на уровне имен

Язык C# позволяет не использовать ключевое слово **using**. Чтобы обратиться тогда к типам, определенным во внешнем пространстве имен, нужно использовать полностью уточненные (или просто уточненные) имена:

```
MyPets.Cat Bagira = new MyPets.Cat();
```

Под уточненным именем понимается следующее имя:

```
<пространство имен>.<тип>
```

Обычно используется ключевое слово **using**, чтобы избежать лишнего ввода с клавиатуры - вы только подумайте, если отказаться от **using**, то везде придется “таскать” префикс “MyPets.”. В CIL-коде всегда используются уточненные имена типов, но программисту гораздо удобнее использовать ключевое слово **using**.

Тем не менее использование уточненных имен может понадобиться, а именно при возникновении конфликтов имен - когда в двух разных пространствах определены типы с одинаковыми именами. Например, у вас может быть два пространства имен с одинаковыми типами внутри - MyPets и JohnPets (животные Джона). Когда вы вызываете конструктор Cat(), нужно уточнить, какое пространство имен нужно использовать:

```
MyPets.Cat Bagira = new MyPets.Cat();
JohnPets.Dog Bobby = new JohnPets.Dog();
```

Есть и еще один способ разрешения конфликтов имен - использование псевдонимов. Определить псевдоним можно так:

```
using <Псевдоним> = <Пространство.Тип>;
```

Пример:

```
using JohnDog = JohnPets.Dog;
...
JohnDog Bobby = new JohnDog();
```

Внимание! Не злоупотребляйте использованием псевдонимов! Злоупотребление приведет к очень запутанному коду, особенно, если этот код придется поддерживать другим программистам.

9.3. Вложенные пространства имен. Пространство по умолчанию

Язык C# позволяет определять пространства имен внутри других пространств, то есть создавать вложенные пространства. Вспомните, внутри пространства System находятся пространства имен Text (операции с текстом), IO (ввод/вывод), Threading (создание многопоточных приложений).

Определить вложенное пространство имен очень просто - нужно определить еще один блок namespace:

```
namespace MyPets {
    namespace Cats {
        namespace Siamese {
            public class Cat
            {
                public string Name;
                public byte Age;
                public byte Weight;
            }
        }
    }
}
```

Создадим объект типа Cat:

```
MyPets.Cats.Siamese.Cat Bagira = new MyPets.Cats.Siamese.Cat();
```

При использовании **using** создание объекта будет компактнее:

```
using MyPets.Cats.Siamese;
...
Cat Bagira = new Cat();
```

При создании нового проекта C# по умолчанию создается пространство имен с таким же именем, как у проекта. При вставке в проект новых файлов

кода командой **Project** ► **Add New Item** все соответствующие типы автоматически помещаются в это пространство имен.

Если вы после создания проекта желаете изменить название пространства имен, то откройте окно свойств проекта (**Project** ► **<имя> Properties**) и на вкладке **Application** измените значение поля **Default namespace** (рис. 9.1).

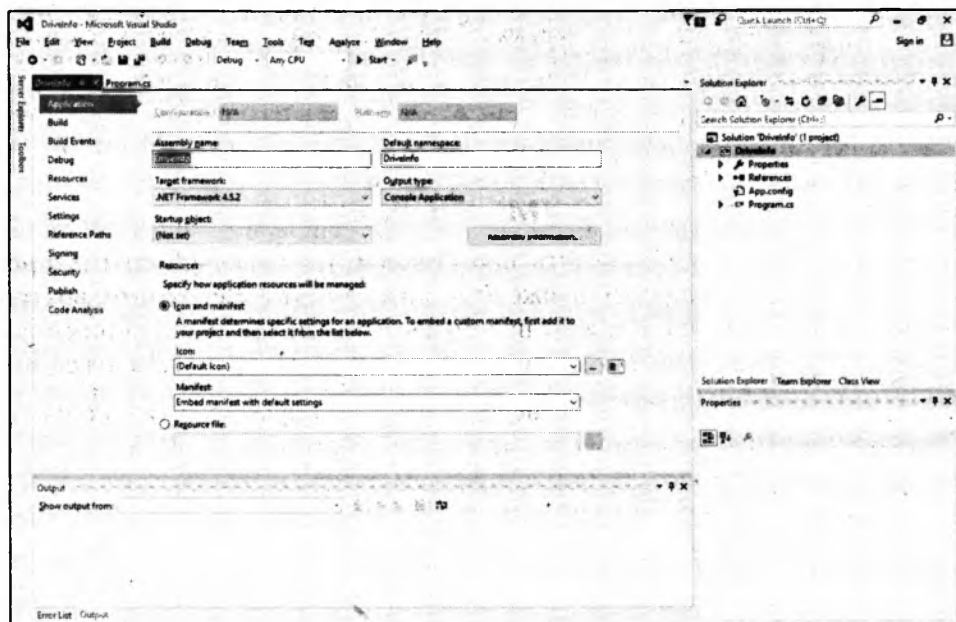


Рис. 9.1. Окно свойств проекта

9.4. Сборки .NET

9.4.1. Зачем нужны сборки?

Первым делом нужно разобраться, зачем нужны сборки .NET и что это вообще такое. Сборка - это двоичный файл, обслуживаемый CLR (Common Language Runtime). Несмотря на то, что у сборок такие же расширения, как у обычных Windows-приложений (.exe, .dll), внутри они устроены иначе. Прежде чем двигаться дальше, давайте рассмотрим преимущества, предлагаемые сборками:

- Сборки повышают удобство повторного использования кода
- Сборки являются единицами, поддерживающими версии

- Сборки определяют границы типов
- Сборки можно настраивать

Теперь давайте рассмотрим эти основные преимущества подробнее и начнем с самого важного - с первого. Всю эту книгу мы создавали простые консольные приложения и читателю могло показаться, что это приложение построено по принципу "все свое ношу с собой". Однако попробуйте запустить одно из разрабатываемых приложений на компьютере, где не установлена платформа .NET (если, конечно, такой найдете) - оно не запустится, поскольку на самом деле приложение обращается к различным DLL. Простые консольные приложения будут требовать наличия `mscorlib.dll`. Более сложные графические - `System.Windows.Forms.dll`.

Библиотека кода представляет собой файл с расширением `.dll` (хотя это не всегда так, но как правило), в котором содержатся типы, которые можно использовать во внешних приложениях. Данные типы можно использовать независимым от языка образом. То есть кто-то может использовать Visual Basic для создания библиотеки кода, которую вы будете использовать в своем C#-приложении (и наоборот!).

Важно понять, что вы можете разбить монолитный код вашего исполняемого файла на несколько сборок, которые вы можете использовать в других своих проектах, или кто-то еще может использовать их в собственных проектах. При этом не важно, какой язык использует другой программист.

Теперь переходим ко второму преимуществу. Сборкам среда .NET присваивает версии в формате <старший номер>.<младший номер>.<номер сборки>.<номер редакции>. По умолчанию используется номер 1.0.0.0, если вы не измените его. Изменить номер сборки можно в свойствах проекта. Для этого на вкладке **Application** нужно нажать кнопку **Assembly Information** и указать номер версии (рис. 9.2).

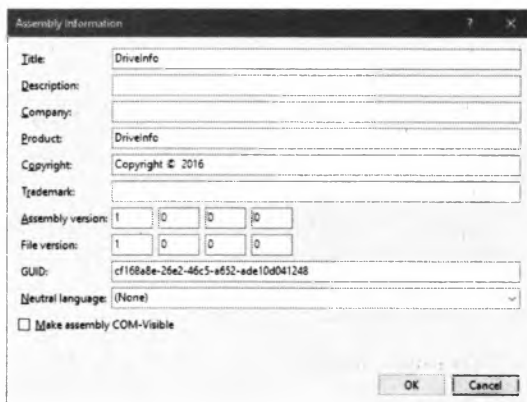


Рис. 9.2. Редактирование номера версии сборки

Сборки определяют границы типов. Мы только что говорили об уточненных именах. Такие имена получаются за счет добавления префикса с названием пространства имен, к которому относится тип. Сборка, содержащая тип, определяет его идентичность. Так, сборки с разными именами, например, `MyPets.dll` и `JohnPets.dll`, в которых определены классы `Cats`, будут выглядеть совершенно разными типами - `MyPets.Cats` и `JohnPets.Cats`. Это еще одно преимущество сборок.

Сборки можно настраивать. Сборки могут быть приватными (`private`) и совместно используемыми (`shared`). Приватные сборки находятся в том же каталоге, что и ваше приложение, которое их использует.

Совместно используемые сборки представляют собой библиотеки, предназначенные для использования в разных приложениях на одной машине, поэтому они помещаются в каталог, который называется GAC (Global Assembly Cache) или глобальный кэш сборки.

Независимо от типа развертывания (`private` или `shared`), для сборок можно создавать специальные конфигурационные файлы в формате XML. Цель этих файлов - указать CLR-среде, какую версию той или иной сборки загружать для определенного приложения, где искать сборки и т.д.

9.4.2. Формат сборки

Любая сборка .NET содержит в себе следующие элементы:

- Заголовок Windows;
- Заголовок CLR;
- CIL-код;
- Метаданные типов;
- Манифест сборки;
- Дополнительные встроенные ресурсы, например, значки.

Вместе с Visual Studio поставляется утилита `dumpbin.exe`, позволяющая просмотреть те или иные заголовки. Так, параметр `/headers` позволяет просмотреть заголовок Windows:

```
dumpbin /headers Library.dll
```

Просмотреть CLR-заголовок можно с помощью параметра `clrheader`:

```
dumpbin /clrheader Library.dll
```

Стоит отметить, что заголовки генерируются автоматически компилятором, и программисту в 99.99% случаев не нужно беспокоиться о тонкостях деталей заголовков. Главное понимать, что находится внутри сборки.

В основе любой сборки находится CIL-код, который представляет собой промежуточный код, не зависящий ни от платформы, ни от языка. Во время выполнения внутренний CIL-код компилируется в инструкции, соответствующие определенной архитектуре (процессору) и операционной системе. Благодаря такому подходу сборки .NET могут выполняться в рамках разных архитектур, устройств и операционных систем.

В любой сборке содержатся метаданные, описывающие формат находящихся внутри нее типов, а также внешних типов, которые она использует. Кроме того, в любой сборке также находится связанный с ней манифест (метаданные сборки). В нем описаны каждый входящий в состав сборки модуль, версия сборки, а также любые внешние сборки, на которые ссылается текущая сборка. Манифест также применяется во время определения места, на которое указывают представляющие внешние ссылки.

Наконец, в любой сборке может содержаться любое количество дополнительных ресурсов, например, значков приложения, звуковых файлов и т.д. Платформа .NET также поддерживает создание так называемых подчиненных сборок, которые содержат только локализованные ресурсы (например, звуковые файлы на болгарском, английском, французском языках). Построение таких сборок выходит за рамки этой книги.

9.4.3. Однофайловые и многофайловые сборки

Сборка может состоять из нескольких модулей. Модуль - просто общий термин, применяемый для обозначения двоичного файла .NET. В большинстве случаев сборка состоит из одного файла (модуля). Такие сборки называются однофайловыми. В этом случае между логической сборкой и лежащим в ее основе физическим двоичным файлом существует соответствие "один к одному".

В однофайловых сборках все элементы (заголовки, метаданные типов, манифест, ресурсы) находятся в одном-единственном файле (.dll или .exe).

Многофайловые сборки состоят из набора модулей .NET, развернутых в виде одной логической единицы. Один из этих модулей называется главным модулем и содержит манифест сборки и все необходимые CIL-инструкции, метаданные и, возможно, дополнительные ресурсы.

В манифесте главного модуля "прописаны" дополнительные модули. Основное преимущество многофайловых сборок в том, что они предоставля-

ют высокоэффективный способ выполнения загрузки содержимого. Представим, что есть машина, которая ссылается на удаленную многофайловую сборку, состоящую из трех модулей, главный из которых установлен на клиенте. Если клиенту будет нужен какой-то тип из второстепенного удаленного модуля *.netmodule, CLR-среда по его требованию немедленно загрузит на локальную машину (в каталог, который называется кэшем загрузки или download cache) только соответствующий двоичный файл. Если размер каждого модуля составляет 10 Мб, то преимущество сразу будет заметно - нужно будет загрузить один модуль размером 10 Мб вместо загрузки одного большого модуля размером 30 Мб.

Это было первое существенное преимущество многофайловых сборок. Второе - то, что разные модули могут быть написаны на разных языках. Например, вам удобно писать приложение на языке C#, но в вашей команде есть другие программисты, для которых более удобно использовать языки VB или Java. Получается, что главное приложение может быть написано на C#, а модули - на языках VB и Java.

Поскольку книга ориентирована на начинающих программистов, многофайловые сборки мы рассматривать не будем. Однако мы рассмотрим создание однофайловой сборки, а именно сначала мы создадим библиотеку PetsLibrary, а потом покажем, как подключить ее к приложению и вызвать метод, описанный в одном из классов этой библиотеки.

9.5. Создание сборки (DLL)

Настало время приступить к практике. Сейчас будет рассмотрен процесс создания однофайловой сборки .dll с именем PetsLibrary, в которой будет небольшой набор общедоступных типов (классов). Для создания библиотеки кода нужно выполнить команду **File » New Project** и создать проект типа **Class Library** (рис. 9.3).

Добавим в нашу библиотеку абстрактный класс с именем Cat. Определим в нем различные свойства кошки. Также определим один абстрактный метод Meow() (см. лист. 9.1).

Листинг 9.1. Библиотека классов PetsLibrary

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace PetsLibrary
{
```



Рис. 9.3. Создание библиотеки кода

```
public abstract class Cat
{
    public string Name;
    public byte Age;
    public byte Weight;

    public abstract string Meow();
    public Cat() { }
    public Cat(string n, byte age, byte w)
    {
        Name = n;
        Age = age;
        Weight = w;
    }
}
```

Теперь создадим непосредственного потомка класса Cat, в котором мы переопределим метод Meow(). Мяукать, конечно, метод не будет, но зато он будет отображать диалог с надписью “Мяу!”. Для отображения диалога мы будем использовать класс MessageBox, который находится в сборке System.Windows.Forms.dll. Чтобы использовать эту сборку, нужно в наш проект PetsLibrary добавить ссылку на System.Windows.Forms.dll. Для это-

го выполните команду меню **Project, Add Reference**. В появившемся окне (рис. 9.4) установите галку напротив **System.Windows.Forms.dll** и нажмите кнопку **OK**.

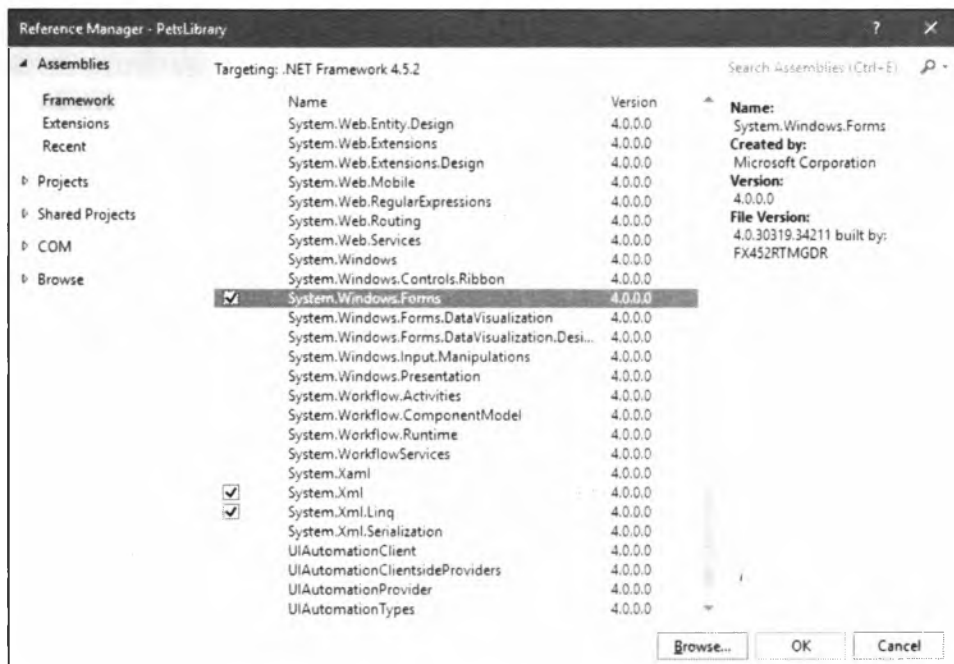


Рис. 9.4. Добавление ссылки на сборку

Поскольку мы создаем консольное приложение, то можно было бы обойтись и без класса **MessageBox**, но мы его добавили, чтобы продемонстрировать процесс добавления ссылки на внешнюю сборку.

Внимание! В окне выбора сборки будут далеко не все зарегистрированные в вашей системе сборки. Если нужной вам сборки нет в списке, используйте кнопку **Browse** для выбора **.dll** или **.exe** файла.

Теперь выполните команду **Project » Add New Item** и создайте новый файл классов с именем **MyCat.cs** (рис. 9.5). Обратите внимание на болванку класса, которую создала среда Visual Studio (рис. 9.6), а именно на пространство имен - **PetsLibrary**.

Созданный файл классов приведен в листинге 9.2.

Листинг 9.2. Файл MyCat.cs

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



Рис. 9.5. Добавление нового файла классов



Рис. 9.6. Создан новый файл класса

```

using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PetsLibrary
{
    public class MyCat : Cat
    {
        public MyCat()
        {
            Name = "Bagira";
        }
        public override string Meow()
        {
            MessageBox.Show("Мяу!");
            return Name;
        }
    }
}

```

9.6. Создание приложения, использующего сборку

Теперь создадим приложение, которое будет использовать созданную ранее сборку. Первым делом, если вы этого еще не сделали, выполните команду **Project » Build** для компиляции нашей сборки. Как обычно, после компиляции среда сообщит вам каталог, в который была помещена созданная сборка `PetsLibrary.dll` (рис. 9.7).

Создайте приложение-с названием `PetsClient` (рис. 9.8). После создания проекта выполните команду **Project » Add Reference** и в появившемся окне (рис. 9.9) выберите созданную нами библиотеку. Путь к ней показан на предыдущем рисунке (если вы делаете все, как по книге, то единственное, что у вас будет отличаться, - это имя пользователя).

Примечание. Созданная ранее библиотека может быть найдена в каталоге `C:\Users\<имя>\Documents\Visual Studio 2015\Projects\PetsLibrary\PetsLibrary\bin\Debug`.

Код нашего приложения приведен в листинге 9.3.

Листинг 9.3. Код приложения, вызывающего DLL

```

using System;
using PetsLibrary;           // вызываем нашу библиотеку

```

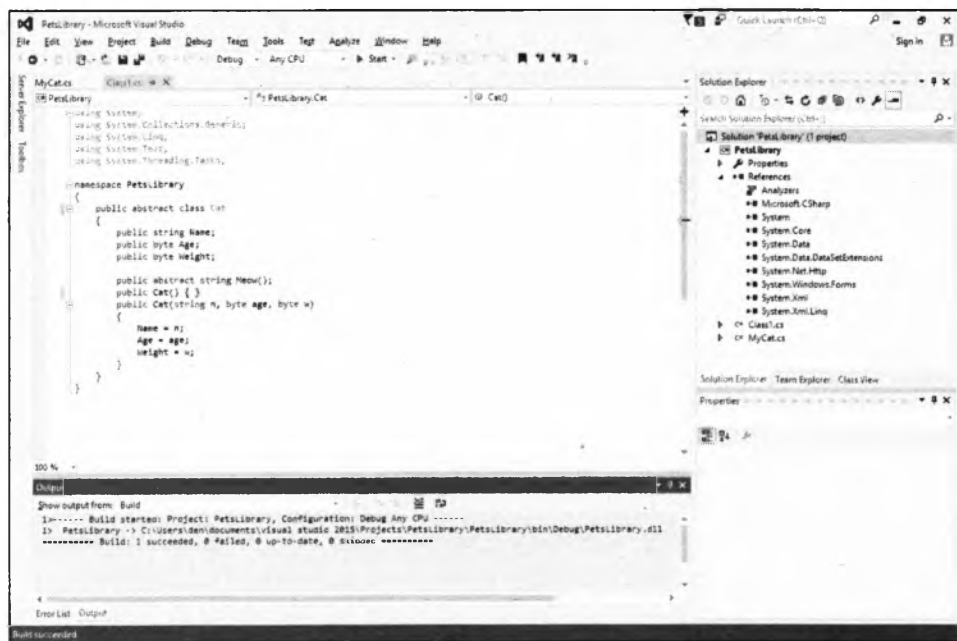


Рис. 9.7. Библиотека PetsLibrary откомпилирована

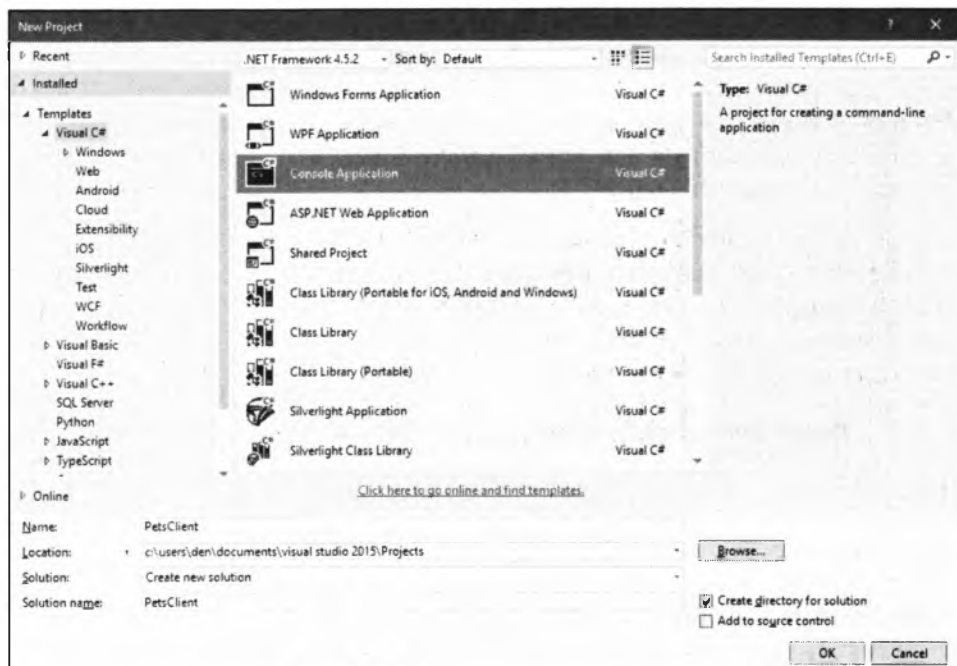


Рис. 9.8. Создание нового проекта

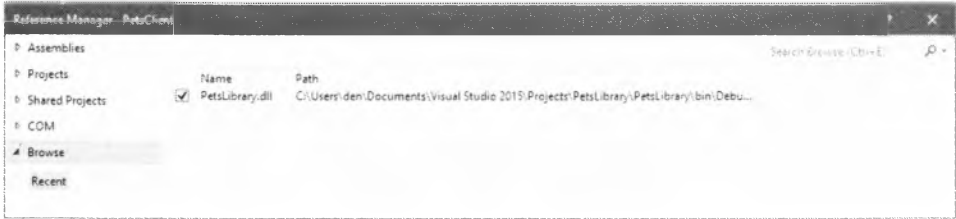


Рис. 9.9. Добавление ссылки на библиотеку PetsLibrary.dll

```
namespace PetsClient
{
    class Program
    {
        static void Main(string[] args)
        {
            MyCat Bagira = new MyCat();
            Bagira.Meow();           // Метод Meow() из DLL
            Console.ReadLine();
        }
    }
}
```

Результат выполнения нашей очень сложной программы приведен на рис. 9.10. Метод Meow(), как и класс MyCat, физически находятся в библиотеке PetsLibrary, которую мы разработали ранее.



Рис. 9.10. Программа в действии

Глава 10.

Многопоточность и параллельное программирование

Основы языка C#,
первые программы

Клиент-серверные
приложения

Многопоточное
программирование

Создание мобильных
приложений на C#

Многопоточность и параллельное программирование - довольно серьезная тема и, несмотря на то, что книга предназначена для новичков, хотелось бы ее осветить. Начнем мы с рассмотрения параллельных коллекций из пространства имен `System.Collections.Concurrent`. Параллельные коллекции являются потокобезопасными и специально предназначены для параллельного программирования. А это означает, что их можно безопасно использовать в многопоточной программе, где возможен доступ к коллекции со стороны двух или больше параллельно работающих потоков.

10.1. Параллельные коллекции

Для безопасного доступа к коллекциям предназначен интерфейс `IProducerConsumerCollection<T>`. У него много методов, но самые важные - это `TryAdd()` и `TryTake()`.

Первый метод пытается (именно поэтому есть "try" в его названии) добавить элемент в коллекцию. Однако добавление может не получиться, если коллекция заблокирована от добавления элементов. Метод возвращает булевское значение, сообщающее об успехе или неудаче операции. Метод `TryTake()` пытается вернуть элемент из коллекции.

В таблице 10.1 перечислены классы из `System.Collections.Concurrent` с кратким описанием их функциональности.

Таблица 10.1. Параллельные коллекции

Класс	Описание
<code>ConcurrentQueue<T></code>	Параллельная версия очереди. Для доступа к элементам очереди применяются методы <code>Enqueue()</code> , <code>TryDequeue()</code> и <code>TryPeek()</code> . Имена этих методов похожи на уже известные методы <code>Queue<T></code> , но с добавлением префикса <code>Try</code> к некоторым из них (к тем, которые могут вызвать ошибку в параллельной среде).
<code>ConcurrentStack<T></code>	Параллельная версия стека. Данный класс определяет методы <code>Push()</code> , <code>PushRange()</code> , <code>TryPeek()</code> , <code>TryPop()</code> и <code>TryPopRange()</code> . Внутри этот класс использует связный список для хранения элементов.

<code>ConcurrentBag<T></code>	Не определяет никакого порядка для добавления или извлечения элементов. Реализует концепцию отображения потоков на используемые внутренние массивы, при этом он старается избежать блокировок. Для доступа к элементам применяются методы <code>Add()</code> , <code>TryPeek()</code> и <code>TryTake()</code> .
<code>ConcurrentDictionary<TKey, TValue></code>	Параллельная версия словаря. Для доступа к членам в неблокирующем режиме служат методы <code>TryAdd()</code> , <code>TryGetValue()</code> , <code>TryRemove()</code> и <code>TryUpdate()</code> .
<code>BlockingCollection<T></code>	Коллекция, которая осуществляет блокировку и ожидает, пока не появится возможность выполнить действие по добавлению или извлечению элемента. Предлагает интерфейс для добавления и извлечения элементов методами <code>Add()</code> и <code>Take()</code> . Данные методы блокируют поток и затем ожидают, пока не появится возможность выполнить задачу.

Далее мы рассмотрим пример использования класса `BlockingCollection`. Это реальный пример многопоточного приложения. В качестве структуры данных была выбрана `BlockingCollection`. Также пример реального многопоточного приложения будет приведен в главе 12 - мы разработаем многопоточный сервер, позволяющий обработать запросы нескольких клиентов. В листинге 10.2 будет приведен пример использования двух потоков - основного и дополнительного.

Листинг 10.1. Многопоточное приложение. Использование `BlockingCollection`

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static BlockingCollection<int> bc;

        static void producer()
        {
            for (int i = 0; i < 100; i++)
```

```

    {
        bc.Add(i * i);
        Console.WriteLine("Producer " + i*i);
    }
    bc.CompleteAdding();
}

static void consumer()
{
    int i;
    while (!bc.IsCompleted)
    {
        if (bc.TryTake(out i))
            Console.WriteLine("Consumer: " + i);
    }
}

static void Main()
{
    bc = new BlockingCollection<int>(4);

    // Задачи поставщика и потребителя
    Task ProducerTask = new Task(producer);
    Task ConsumerTask = new Task(consumer);

    // Запустим задачи
    ProducerTask.Start();
    ConsumerTask.Start();

    try
    {
        Task.WaitAll(ConsumerTask, ProducerTask);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
    }
    finally
    {
        ConsumerTask.Dispose();
        ProducerTask.Dispose();
        bc.Dispose();
    }

    Console.ReadLine();
}
}

```

Сначала мы создаем задачи поставщика и потребителя, а потом в блоке try мы ждем завершения этих задач.

10.2. Библиотека распараллеливания задач

Современные компьютеры имеют многоядерные процессоры (самые слабые из них обладают двумя ядрами). Не запуская нескольких потоков, вы не сможете использовать на полную мощь современные процессоры. Поэтому параллельное программирование так важно и рассматривается в этой книге.

Главным нововведением в .NET Framework 4.0 является библиотека (TPL, Task Parallel Library). Данная библиотека упрощает создание и применение многих потоков. Да, потоки и задачи можно было создавать и в более ранних версиях .NET, но с TPL создание многопоточных приложений существенно упрощается. К тому же она позволяет автоматически использовать несколько процессоров. Именно поэтому библиотека TPL рекомендуется в большинстве случаев к применению для организации многопоточной обработки.

Также, начиная с версии 4.0, .NET поддерживает параллельный язык интегрированных запросов (PLINQ). Язык PLINQ дает возможность составлять запросы, для обработки которых автоматически используется несколько процессоров.

Язык PLINQ довольно простой, с его помощью очень просто запросить параллельную обработку запроса. Следовательно, используя PLINQ, вы без особых проблем сможете внедрить параллелизм в свою программу.

Библиотека TPL определена в пространстве имен System.Threading.Tasks. Однако для работы с ней также нужно подключить класс System.Threading, поскольку он поддерживает синхронизацию и другие средства многопоточной обработки, в том числе и те, что входят в класс Interlocked.

Хотя есть библиотека TPL и язык PLINQ, организовать многопоточную обработку можно и средствами класса Thread, хотя этот подход считается устаревшим и вместо него рекомендуется использовать как раз TPL и PLINQ.

Применяя TPL, добавить параллелизм в программу можно двумя основными способами. Первый способ - это параллелизм данных: одна операция над совокупностью данных разбивается на два (или больше) параллельно выполняемых потока, в каждом из которых обрабатывается часть данных.

Такой подход может привести к значительному ускорению обработки данных по сравнению с последовательным подходом.

Параллелизм данных был доступен и раньше с помощью класса `Thread`, но его использование требовало немалых усилий и времени. Но с появлением TPL все изменилось, и теперь вы можете создавать масштабируемые решения без особого труда.

Второй способ - параллелизм задач: две (или более) операции выполняются параллельно. Параллелизм задач представляет собой разновидность параллелизма, который достигался в прошлом средствами класса `Thread`. Благодаря TPL у программиста есть возможность автоматически масштабировать исполнение кода на несколько процессоров. Библиотека TPL позволяет автоматически распределять нагрузку приложений между доступными процессорами в динамическом режиме, используя пул потоков CLR. Также она занимается распределением работы, планированием потоков, управлением состоянием и прочими низкоуровневыми деталями.

10.3. Класс `Task`

В основе TPL лежит класс `Task`, то есть при использовании TPL нужно создавать объекты типа `Task`, а не `Thread`. Ранее, в листинге 10.1, уже был показан пример использования `Task`, правда, без объяснения подробностей.

Класс `Task` отличается от `Thread` тем, что он является абстракцией, представляющей асинхронную операцию. А в классе `Thread` инкапсулируется поток исполнения. Понятно, что на системном уровне поток по-прежнему остается элементарной единицей исполнения, которую можно планировать средствами операционной системы. К тому же - исполнением задач занимается планировщик задач, работающий с пулом потоков. Это означает, что несколько задач могут разделять один и тот же поток. Класс `Task` определен в пространстве имен `System.Threading.Tasks`.

Для создания новой задачи нужно просто создать новый объект класса `Task` и начать его выполнение. Для создания объекта используется конструктор `Task()`, а для запуска - метод `Start()`. Чаще всего используется следующая форма конструктора:

```
public Task(Action действие)
```

Здесь действие - это точка входа в код, который будет представлять задачу. Форма делегата `Action` следующая:

```
public delegate void Action()
```

Как видите, точкой входа служит метод, который не принимает параметров и не возвращает никаких значений. Однако позже будет показано, как передать делегату данные.

После создания задачу можно запустить методом `Start()`, что и было показано в листинге 10.1. После вызова этого метода планировщик задач планирует выполнение задачи. Обратите внимание: задача будет запущена не сразу, она будет добавлена в очередь планировщика заданий. Она будет выполнена, как только до нее дойдет очередь. В простых демонстрационных приложениях задача будет выполнена мгновенно, но в сложных больших приложениях возможна определенная задержка между вызовом `Start()` и началом выполнения задачи. В листинге 10.2 приводится еще один пример основного и дополнительного потока.

Листинг 10.2. Основной и дополнительный поток

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        // Метод, который будет запущен в качестве задачи
        static void TaskAction()
        {
            Console.WriteLine("TaskAction started");

            for (int count = 0; count < 5; count++)
            {
                // Ждем 1 секунду (1000 мс)
                // Ожидание имитирует бурную деятельность потока
                Thread.Sleep(1000);

                Console.WriteLine("Count: " + count);
            }
        }

        static void Main()
        {
            Console.WriteLine("Main thread started");

            Task task = new Task(TaskAction);

            // Запустить задачу
```

```

        task.Start();

        for (int i = 0; i < 20; i++)
        {
            Console.Write(".");
            Thread.Sleep(500);
        }

        Console.WriteLine("Main thread shutdown");
        Console.ReadLine();
    }
}

```

На рис. 10.1 приводится результат выполнения этой программы. Как видно из вывода программы, главный и дополнительный поток выполняются одновременно. Основной поток выводит точки, а дополнительный - значение счетчика.

Нужно помнить, что как только завершается основной поток, то завершаются и все дополнительные (созданные классом Task задачи). Поэтому в основном потоке мы используем метод Thread.Sleep(), чтобы сохранять основной поток активным, пока не завершится выполнение дополнительного потока.

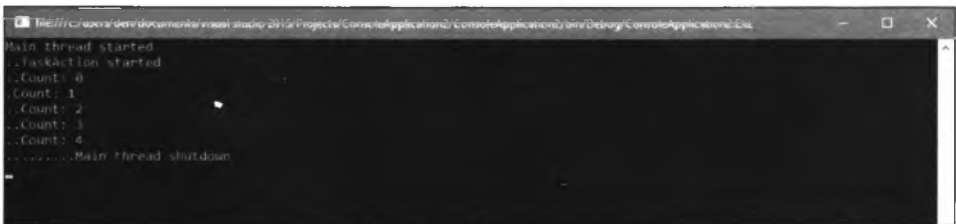


Рис. 10.1. Результат выполнения программы из листинга 10.2

Если вы ранее использовали класс Thread, то наверняка заметили, что в классе Task нет свойства Name, которое задавало имя задачи. Зато в нем есть свойство Id - идентификатор задачи, но оно типа int и доступно только для чтения.

При запуске каждой задаче назначается собственный идентификатор. Значения Id уникальны, но не упорядочены. Поэтому если вы запустили две задачи, то значение Id второй задачи не обязательно будет больше значения Id первой задачи.

Получить идентификатор текущей задачи можно с помощью свойства CurrentId. Данное свойство тоже доступно только для чтения:

Изменим немного программу из листинга 10.2. Во-первых, мы добавим вторую задачу, а во-вторых, сделаем вывод Id задачи в процессе ее выполнения. Также мы немного изменим счетчики - для большей наглядности. Измененный вариант приведен в листинге 10.3.

Листинг 10.3. Пример использования свойства CurrentId

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        // Метод, который будет запущен в качестве задачи
        static void TaskAction()
        {
            Console.WriteLine("TaskAction started, task {0}", Task.CurrentId);

            for (int count = 0; count < 10; count++)
            {
                // Ждем 1 секунду (1000 мс)
                // Ожидание имитирует бурную деятельность потока
                Thread.Sleep(1000);

                Console.WriteLine("Task {0}, Count {1} ", Task.CurrentId, count);
            }
        }

        static void Main()
        {
            Console.WriteLine("Main thread started");

            Task task1 = new Task(TaskAction);
            Task task2 = new Task(TaskAction);

            // Запустить задачу
            task1.Start();
            task2.Start();

            for (int i = 0; i < 25; i++)
            {
                Console.Write(".");
                Thread.Sleep(500);
            }
        }
    }
}
```

```

    }
    Console.WriteLine("Main thread shutdown");
    Console.ReadLine();
}
}
}

```

Проанализируем вывод программы (см. рис. 10.2). Было запущено две задачи. Первая, как видите, получила идентификатор 2, вторая 1. Первая задача (с Id 2) завершилась также первой.



```

file:///c:/users/den/documents/visual studio 2015/Projects/ConsoleApplication2/ConsoleApplication2/bin/Debug/ConsoleApplication2.exe
Main thread started
TaskAction started, task 2
TaskAction started, task 1
..Task 2, Count 0
Task 1, Count 0
..Task 2, Count 1
Task 1, Count 1
..Task 2, Count 2
Task 1, Count 2
..Task 2, Count 3
Task 1, Count 3
..Task 2, Count 4
Task 1, Count 4
..Task 2, Count 5
Task 1, Count 5
..Task 2, Count 6
Task 1, Count 6
..Task 2, Count 7
Task 1, Count 7
..Task 2, Count 8
Task 1, Count 8
..Task 2, Count 9
Task 1, Count 9
....Main thread shutdown

```

Рис. 10.2. Две задачи, демонстрация свойства `CurrentId`

10.4. Ожидание задачи

В двух предыдущих примерах основной поток выполнения (то есть метод `Main()`) мог завершиться раньше, чем выполнение запущенных задач. Чтобы этого не произошло, мы использовали метод `Thread.Sleep()`, но это, мягко говоря, неправильно по двум причинам:

Выполнение метода `Main()` все равно может завершиться раньше, если программист не угадает задержку для `Thread.Sleep()` и укажет ее меньше, чем нужно для выполнения всех запущенных задач.

Как видно из рис. 10.2, основной поток продолжает выполнение, хотя задача уже выполнена - в этом случае программист указал задержку больше, чем нужно. Выходит, программа должна уже завершиться, но она все еще продолжает работу.

Далеко не всегда можно спрогнозировать выполнение каждой запущенной задачи, поэтому использование метода `Thread.Sleep()` является неправильным.

В классе `Task` есть метод `Wait()`, позволяющий организовать завершение задач более совершенным образом. При выполнении этого метода могут быть сгенерированы два исключения:

- `ObjectDisposedException` - генерируется, если задача освобождена посредством вызова метода `Dispose()`.
- `AggregateException` - генерируется, если задача сама генерирует исключение или же отменяется.

Перепишем наш пример с использованием метода `Wait()`. Данный метод мы будем вызывать из метода `Main()`, чтобы дождаться выполнения задач `task1` и `task2`.

Листинг 10.4. Использование метода Wait()

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        // Метод, который будет запущен в качестве задачи
        static void TaskAction()
        {
            Console.WriteLine("TaskAction started, task {0}", Task.CurrentId);

            for (int count = 0; count < 10; count++)
            {
                // Ждем 1 секунду (1000 мс)
                // Ожидание имитирует бурную деятельность потока
                Thread.Sleep(1000);

                Console.WriteLine("Task {0}, Count {1} ", Task.CurrentId, count);
            }

            Console.WriteLine("End of task {0}", Task.CurrentId);
        }

        static void Main()
        {

```

```

        Console.WriteLine("Main thread started");

        Task task1 = new Task(TaskAction);
        Task task2 = new Task(TaskAction);

        // Запустить задачу
        task1.Start();
        task2.Start();

        Console.WriteLine("Waiting...");
        task1.Wait();
        task2.Wait();

        Console.WriteLine("Main thread shutdown");
        Console.ReadLine();
    }
}

```



Рис. 10.3. Использование метода Wait()

Как видите (рис. 10.3), теперь основной поток выполняется ровно столько, сколько нужно - он завершается сразу после завершения последней задачи. Теперь нет необходимости вызывать метод Thread.Sleep() и нет этих ужасных точек, свидетельствующих о выполнении главного потока.

Иногда вместо использования метода Wait() удобнее использовать метод WaitAll(), которому передаются задачи, выполнения которых нужно дождаться. Пример применения этого метода был приведен в листинге 10.1:

```

try
{
    Task.WaitAll(ConsumerTask, ProducerTask);
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
finally
{
    ConsumerTask.Dispose();
    ProducerTask.Dispose();
    bc.Dispose();
}

```

В блоке `try/catch/finally` происходит ожидание выполнения задач `ConsumerTask` и `ProducerTask`.

Метод `Dispose()` реализуется в классе `Task`, освобождая ресурсы, используемые этим классом. Как правило, ресурсы, связанные с классом `Task`, освобождаются автоматически во время "сборки мусора" (или по завершении программы). Но если эти ресурсы требуется освободить еще раньше, то для этой цели служит метод `Dispose()`. Это особенно важно в тех программах, где создается большое число задач, оставляемых на произвол судьбы. В листинге 10.1 мы использовали вызов этого метода в блоке **finally**, хотя именно в том конкретном случае особой нужды использовать данный метод не было.

10.5. Класс `TaskFactory`

Ранее мы сначала создавали задачу, а потом запускали ее выполнение. При желании, используя класс `TaskFactory`, можно сразу создать и запустить на выполнение задачу.

По умолчанию объект класса `TaskFactory` может быть получен из свойства `Factory`, доступного только для чтения в классе `Task`. Используя это свойство, можно вызвать любые методы класса `TaskFactory`. Метод `StartNew()` существует во множестве форм. Самая простая форма метода `StartNew()` выглядит так:

```
public Task StartNew(Action action)
```

В нашем случае можно было бы переписать код так (с использованием `TaskFactory`):

```
// Использование фабрики задач
TaskFactory tf = new TaskFactory();
```

```
Task t1 = tf.StartNew(TaskAction);
```

```
/* А можно создать и запустить задачу так:  
Task t2 = Task.Factory.StartNew(TaskAction);
```

Как видите, можно использовать один из двух способов создания и запуска задач с помощью TaskFactory.

10.6. Продолжение задачи

Иногда бывают ситуации, когда нужно запустить следующую задачу сразу по завершению текущей. Для этого в TPL используется метод ContinueWith(), определенный в классе Task. Ниже приведена простейшая форма его объявления:

```
public Task ContinueWith(Action<Task> действие_продолжения)
```

10.7. Возврат значения из задачи

Задача может возвращать значение. Ранее мы не рассматривали такие задачи, но вы должны знать, что такая возможность имеется. Возвращение значения из задачи удобно по нескольким причинам. Прежде всего, задача может вычислять некоторый результат и возвращать в основной поток вычисленное значение. Таким образом, в C# можно организовать параллельные вычисления. Также вызывающий процесс окажется заблокированным до тех пор, пока не будет получен результат. Это означает, что для организации ожидания результата не требуется никакой особой синхронизации, в том числе и метода Wait().

Чтобы вернуть результат из задачи, нужно создать эту задачу, используя обобщенную форму Task<TResult> класса Task. Ниже приведены два конструктора этой формы класса Task:

```
public Task(Func<TResult> функция)  
public Task(Func<Object, TResult> функция, Object состояние)
```

Здесь функция обозначает выполняемый делегат. Обратите внимание на то, что в этом случае он должен быть типа Func, а не Action. Тип Func используется именно в тех случаях, когда задача возвращает результат. В первой форме конструктора создается задача без аргументов, а во втором - задача, принимающая аргумент типа Object, передаваемый как состояние. В документации по C# вы также найдете другие формы конструктора класса Task.

Также имеются и другие формы метода StartNew(), доступные в обобщенной форме класса TaskFactory<TResult> и поддерживающие возврат

результата из задачи. Рассмотрим те из них, которые применяются параллельно с только что рассмотренными конструкторами класса `Task`:

```
public Task<TResult> StartNew(Func<TResult> функция)
public Task<TResult> StartNew(Func<Object, TResult> функция,
Object состояние)
```

Значение, возвращаемое задачей, может быть получено из свойства `Result` в классе `Task`:

```
public TResult Result { get; internal set; }
```

Аксессор `set` является внутренним для данного свойства, и поэтому во внешнем коде оно доступно только для чтения. Следовательно, задача получения результата блокирует вызывающий код до тех пор, пока результат не будет вычислен.

Мы только что рассмотрели основные возможности по созданию параллельных задач в C#. Если этого вам оказалось мало, вы заинтересовались и готовы продолжить изучение параллельного вычисления, тогда обратите свое внимание на класс `Parallel`, который поддерживает набор методов, которые позволяют выполнять итерации по коллекции данных (точнее, по объектам, реализующим `IEnumerable<T>`) в параллельном режиме. Описание этого класса вы найдете в документации по .NET Framework на сайте Microsoft:

<https://msdn.microsoft.com/ru-ru/library/system.threading.tasks.parallel%28v=vs.110%29.aspx>

Глава 11.

Сетевое программирование

Основы языка C#,
первые программы

Клиент-серверные
приложения

Многопоточное
программирование

Создание мобильных
приложений на C#

Современный компьютер сложно представить без доступа к Сети - будь то локальной или же к Интернету. Поэтому и книгу по программированию сложно представить без описания хотя бы основ сетевого программирования. В этой главе будут представлены такие основы, и начнем работу с сетью мы с пространства имен System.Net.

11.1. Пространство имен System.Net

Пространство имен System.Net содержит сетевые классы для поиска IP-адресов, сетевой аутентификации, разрешений, отправки и получения данных. Для большего удобства все классы из System.Net мы отсортируем по группам и рассмотрим.

Начнем с *поиска имен*. Для преобразования IP-адреса в символьное имя и обратно используется система доменных имен (DNS, Domain Name System). Платформа .NET содержит следующие классы для разрешения имен/IP-адресов:

- `Dns` - используется для разрешения символьных имен в IP-адреса и обратно.
- `DnsPermission` - представляет разрешение, необходимое для поиска имени.
- `DnsPermissionAttribute` - позволяет отмечать сборки, классы и методы, нуждающиеся в полномочиях, определяемых классом `DnsPermission`.

Обработка IP-адресов производится в классе `IPAddress`. Сейчас некоторые узлы содержат более одного IP-адреса и более одного имени (используются псевдонимы). Информация о дополнительных IP-адресах и псевдонимах находится в классе `IPHostEntry`. Когда мы ищем имя, класс `Dns` возвращает объект типа `IPHostEntry`.

Для *аутентификации и авторизации* используется класс `AuthenticationManager`. Данный класс обращается к этим модулям, чтобы идентифицировать пользователя. Модули аутентификации получают информацию запроса и данные о личности пользователя с помощью интерфейса `ICredentials` и возвращают объект `Authorization` для авторизованных пользователей, которые могут использовать тот или иной ресурс. Прило-

жение-клиент может использовать класс `NetworkCredential`, передающий данные о пользователе на сервер.

Различные *сетевые запросы* можно осуществить посредством абстрактных классов `WebRequest` и `WebResponse`. В `System.Net` имеется несколько специальных реализаций этих классов для HTTP и доступа к файлам: `HttpWebRequest`, `HttpWebResponse`, `FileWebRequest`, `FileWebResponse`. Класс компонентов `WebClient` облегчает использование `WebRequest` и `WebResponse` из `Visual Studio .NET`.

Для *управления соединениями* используются классы `ServicePoint` и `ServicePointManager`. Если же вы хотите использовать *сокеты*, то вам придется пространство имен `System.Net.Sockets`. Для программирования с использованием сокетов вам нужно знать тот или иной сетевой протокол. Однако использование сокетов - это хоть и более низкоуровневый способ взаимодействия, он позволяет наиболее гибко организовать связь.

Далее будут рассмотрены самые важные классы из пространства `System.Net`.

11.2. Класс Uri

Класс `Uri` предназначен для работы с универсальными идентификаторами ресурса (URI, Uniform Resource Identifier). Мы каждый день используем URI и даже не задумываемся над этим, - когда обращаемся к Web-страницам, FTP-серверам и т.д. URI можно использовать не только для обращения к ресурсам Web и FTP, но и к сетевым и даже локальным файлам.

Рассмотрим пример URI и разберемся, из чего он состоит:

`http://www.example.com:3128/public/index.php?page=about&lang=en`

Первая часть, содержащая название протокола (`http://`), называется схемой (scheme). После ограничителя схемы (`://`) указывается имя или IP-адрес узла, в нашем случае `www.example.com`.

После имени узла через двоеточие указывается номер порта. Если номер порта не указывается, то и двоеточие тоже не указывается.

Далее следует путь, определяющий каталог и страницу нужного нам ресурса. В нашем случае - это `/public/index.php`.

После символа `?` указываются параметры, передаваемые ресурсу. В терминологии URI эта часть называется запросом (query). В нашем случае запрос выглядит так: `index.php?page=about&lang=en`.

Для работы с URI используется, как уже было отмечено, класс `Uri`:

```
Uri site = new Uri("http://www.example.com:3128/public/index.  
php?page=about&lang=en");
```

Далее, используя свойства класса, можно получить доступ к компонентам Uri:

- `Scheme` - содержит первую часть URI - схему. Для протокола HTTP это будет значение `http`.
- `Host` - имя хоста.
- `Port` - номер порта, если он задан.
- `AbsolutePath` - абсолютный путь к ресурсу, в примере выше это `/public/index.php`.
- `Query` - содержит строку после пути, в нашем случае это `?page=about&lang=en`.
- `PathAndQuery` - путь и запрос, в нашем случае `/public/index.php?page=about&lang=en`.
- `Fragment` - если после пути следует фрагмент, он возвращается в свойстве `Fragment`. За путем могут следовать только строка запроса или фрагмент. Фрагмент идентифицируется символом `#`.
- `UserInfo` - если указано имя пользователя, например, `ftp://user@ftp.example.com`, то это свойство будет содержать строку `user`.

В этом классе есть и другие свойства, но они используются гораздо реже. Важно помнить, что после создания конструктором экземпляра класса `Uri` не может больше изменяться. Свойства класса `Uri` доступны только на чтение. Для динамического изменения URI можно использовать класс `UriBuilder`:

```
UriBuilder myURI = new UriBuilder("http", "www.example.com",  
3128,  
"/public/index.php?page=about&lang=en", "#team");  
Uri site = myURI.Uri;
```

11.3. Загрузка файлов (HTTP и FTP)

Одна из довольно распространенных задач - загрузка файлов, например, приложение закачивает обновление с сервера разработчика или же обрабатывает какую-то веб-страницу.

Для загрузки файла может использоваться класс `WebClient`. Вот как легко и непринужденно можно загрузить файл (лист. 11.1):

Листинг 11.1. Загрузка файлов. Пример использования WebClient

```
using System;
using System.Collections.Generic;
using System.Net;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Downloading...Please wait...");

            WebClient wc = new WebClient();
            wc.DownloadFile("http://dkws.org.ua/proxy/proxy.
zip", "c:\\temp\\proxy.zip");

            Console.WriteLine("Download complete!");
            Console.ReadLine();
        }
    }
}
```

Метод `DownloadFile()` принимает два параметра: первый - это путь к файлу, который нужно скачать, а второй - локальное имя файла. Обратите внимание: я указал не только имя, но и путь, папка `C:\temp`. Символ `\` требует экранирования, поэтому он указан как `\\`. Также папка `C:\temp` должна существовать на момент запуска программы. Результат работы программы изображен на рис. 11.1 - видно, что программа завершила загрузку файла в папку `C:\temp`.

Теперь представим другую ситуацию. Допустим, нам нужно загрузить не файл, а нам нужно получить какую-то веб-страницу в строку для последующей ее обработки. Связываться для этого с файлами не хочется. Во-первых, в реальном приложении придется производить обработку корректности записи файла, иметь дело с правами доступа и т.д. Обычно программа устанавливается в `C:\Program Files`, а права записи к этому каталогу есть не у всех пользователей. Значит, записывать придется в домашний каталог пользователя. Во-вторых, сама процедура неэффективная - сначала мы загружаем страницу в файл, потом читаем текст из этого файла. Гораздо удобнее использовать метод `OpenRead()`:

```
WebClient wc = new WebClient();
Stream str = wc.OpenRead("http://dkws.org.ua/");
```

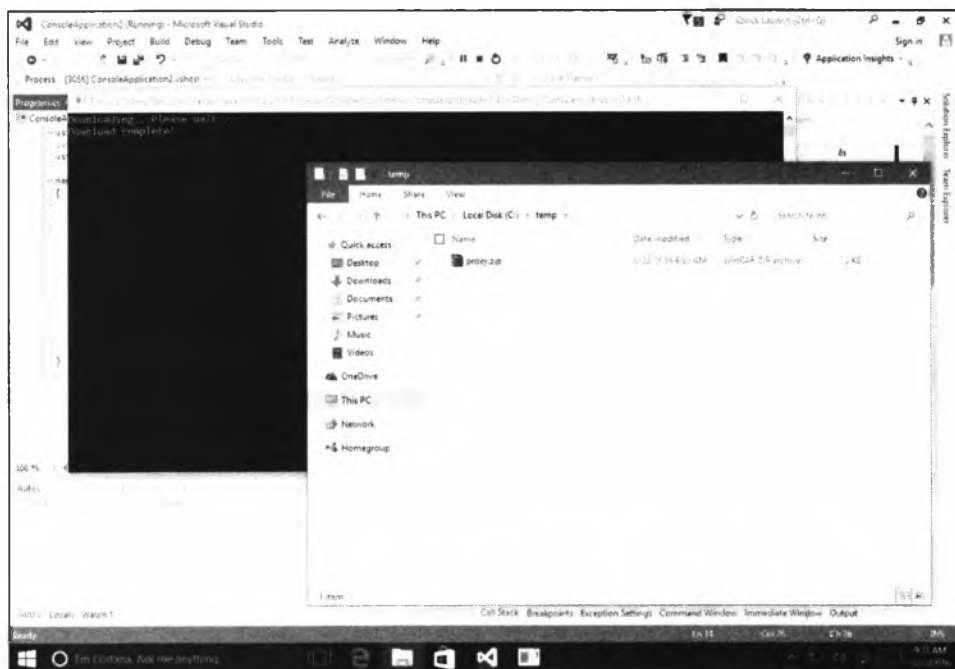


Рис. 11.1. Файл успешно загружен

Содержимое страницы будет загружено в переменную `str`. После этого можно использовать класс `StreamReader` для обработки потока `str`

```
StreamReader sr = new StreamReader(str);
string s;
```

```
while ((s = sr.ReadLine()) != null)
    Console.WriteLine(s);
```

В классе `WebClient` также имеются методы `UploadFile()` и `UploadData()`. Первый используется, чтобы отправить на сервер файл, а второй - данные HTML-формы. Стоит отметить, что второй метод выгружает на сервер двоичные данные, представленные в виде массива байт, по указанному URI (есть и метод `DownloadData()`, предназначенный для извлечения массива байтов из URI).

Нужно отметить, что выгрузка (upload) файлов по протоколу HTTP осуществляется довольно редко, поскольку такое действие на реальных серверах запрещено настройками безопасности. Как правило, загрузка файлов на сервер осуществляется по протоколу FTP, а для этого `WebClient` не подходит, поскольку его возможности весьма ограничены. Причина в том,

что WebClient — класс общего назначения, предназначенный для работы с любым протоколом, позволяющим отправлять запросы и получать ответы (вроде HTTP и FTP). Он не может обработать никаких средств, специфичных для какого-то одного протокола, например, сокет, которые специфичны для HTTP. Чтобы написать более сложное сетевое приложение, нужно использовать другие классы, а именно WebRequest и WebResponse. В листинге 11.2 приведен пример использования этих классов для загрузки файла с FTP-сервера, требующего аутентификации.

Листинг 11.2. Загрузка файла с FTP-сервера

```
using System;
using System.IO;
using System.Net;
using System.Text;

namespace FTPDownload
{
    public class WebRequestGetExample
    {
        public static void Main ()
        {
            // Получаем объект, который будем использовать для
            // связи с сервером
            FtpWebRequest request = (FtpWebRequest)WebRequest.
            Create("ftp://example.com/report.txt");

            // Выбираем метод загрузки файла
            request.Method = WebRequestMethods.Ftp.
            DownloadFile;

            // Задаем имя пользователя и пароль
            request.Credentials = new NetworkCredential
            ("den", "1234567");

            // Формируем объект для ответа
            FtpWebResponse response = (FtpWebResponse)request.
            GetResponse();

            // Создаем поток ответа
            Stream responseStream = response.
            GetResponseStream();

            // Используем StreamReader для чтения ответа
            StreamReader reader = new
            StreamReader(responseStream);
```

```

// Поскольку файл текстовый, просто выводим его на консоль
// для других типов файлов нужно сохранить поток в файл
    Console.WriteLine(reader.ReadToEnd());

    Console.WriteLine("Загрузка завершена. Статус
{0}", response.StatusDescription);

    reader.Close();           // закрываем reader
    response.Close();         // закрываем ответ
}
}
}

```

11.4. Класс Dns. Разрешение доменных имен

В этом разделе мы поговорим о преобразовании IP-адресов в символьные имена и обратно. Начнем с класса `IPAddress`, который представляет IP-адрес. Сам адрес доступен в виде свойства `GetAddressBytes` и может быть преобразован в десятичный формат с разделителями-точками с помощью метода `ToString()`.

```

IPAddress ip = IPAddress.Parse("192.168.1.9");
byte[] adress = ip.GetAddressBytes();
string ipString = ip.ToString();

```

Для разрешения имен в IP-адреса используется класс **Dns**. Рассмотрим небольшой пример:

```

using System.Net;
...
string hostname = "www.mail.ru", ips="", aliases="";

IPHostEntry entry = Dns.GetHostEntry(hostname);

foreach (IPAddress a in entry.AddressList)
    ips += a.ToString() + "\n";

foreach (string aliasName in entry.Aliases)
    aliases += aliasName + "\n";

Console.WriteLine("Псевдонимы:\n" + aliases);
Console.WriteLine("\nСписок IP:\n" + ips);

```

Как видите, у `www.mail.ru` нет псевдонимов, зато есть два IP-адреса.



Рис. 11.2. Разрешение доменного имени

11.5. Сокеты

11.5.1. Типы сокетов

Изначально сокеты появились в операционной системе UNIX. Сокет - это один конец двустороннего канала связи между двумя программами, работающими в сети. Используя два сокета, можно передавать данные между разными процессами (как локальными, так и удаленными). На локальной машине сокет может использоваться как средство межпроцессного взаимодействия. В сети сокет может использоваться для передачи данных между двумя программами, запущенными на разных компьютерах.

Прежде чем использовать сокет, его нужно открыть, задав надлежащие разрешения. Как только ресурс открыт, из него можно считывать данные и/или в него можно записывать данные. После использования сокета нужно вызвать метод `Close()` для его закрытия.

Существует два типа типа сокетов — потоковые сокеты (stream sockets) и дейтаграммные (datagram socket).

Потоковый сокет — это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, то есть через такой сокет можно передавать и принимать данные.

Потоковый сокет гарантирует доставку передаваемых данных в целостности и сохранности, он сохраняет последовательность данных. Если вам нужно не просто доставить данные, а доставить их в правильной последовательности, то нужно использовать потоковый сокет. Потоковые сокеты используют в качестве транспортного протокола TCP (Transmission Control Protocol), который контролирует доставку данных.

Дейтаграммные сокеты не подразумевают установку соединения. Сообщение отправляется указанному сокету, и на этом все. Никакая доставка или хотя бы ее контроль не гарантируется. Если все хорошо, сообщение будет доставлено получателю, если нет, вы даже не узнаете об этом. Дейтаграммные сокеты использует для доставки сообщений протокол UDP (User Datagram Protocol).

Когда нужна гарантированная доставка данных, нужно использовать потоковые сокеты. Если же надежное соединение с сервером не требуется, можно использовать дейтаграммные сокеты. Ведь на установление надежного соединения с сервером требуется время, которое просто вносит задержки в обслуживание - иногда проще и быстрее отправить несколько UDP-пакетов.

Есть еще и сырые сокеты (raw sockets). Задача таких сокетов заключается в обходе механизма, с помощью которого компьютер обрабатывает TCP/IP. Сырой сокет — это сокет, который принимает пакеты, обходит уровни TCP и UDP в стеке TCP/IP и отправляет их непосредственно приложению. Подходит, если вы хотите реализовать собственный транспортный протокол, в противном случае проще использовать TCP или UDP.

11.5.2. Порты

Представим себе взаимодействие двух компьютеров - Web-сервера, пусть у него будет IP-адрес 192.168.1.1, и компьютера пользователя с IP-адресом 192.168.1.100. Компьютер пользователя отправляет серверу запрос на получение какой-то страницы. Сервер получает запрос, обрабатывает его и отправляет пользовательскому компьютеру ответ. Но на сервере может быть запущено несколько программ для обработки запросов, например, веб-сервер Apache, почтовый сервер Exim, FTP-сервер ProFTPD и др. Как система определит, какому именно приложению нужно передать запрос от пользовательского компьютера?

В заголовке TCP-пакета передается номер порта. По сути, номер порта задает номер приложения на сервере, которому будут отправлены данные. Стандартные номера портов определены организацией IANA (Internet Assigned Numbers Authority). Так, порт с номером 80 используется для веб-сервера, 22 - для SSH, 110 - для POP3-протокола, 25 - для SMTP (отправка почты), 21 - для FTP и т.д.

Получив TCP-пакет, система “считывает” номер порта и передает этот пакет соответствующему приложению. Аналогично приложение веб-сервер, получив запрос, отправляет клиентскому компьютеру ответ. Клиентский компьютер может отправить несколько запросов - один к FTP-серверу, дру-

гой к веб-серверу и т.д. На клиентском компьютере также открывается порт для соединения, и этот порт сообщается серверу, поэтому сервер уже знает порт клиента и отправляет пакет, в котором данные адресуются указанному порту. Номер клиентского порта не стандартизирован и может быть произвольным (понятно, что в качестве клиентского порта система не позволит выбрать зарезервированный порт).

11.5.3. Классы для работы с сокетами

В таблице 11.1 перечислены классы, предназначенные для работы с сокетами.

Таблица 11.1. Классы для работы с сокетами

Класс	Описание
<code>MulticastOption</code>	Устанавливает значение IP-адреса для присоединения к IP-группе или для выхода из нее
<code>NetworkStream</code>	Реализует базовый класс сетевого потока, использующегося для отправки и получения данных
<code>Socket</code>	Обеспечивает базовую функциональность приложения сокета
<code>SocketException</code>	Задаёт исключения при работе с сокетами
<code>TcpClient</code>	Класс <code>TcpClient</code> построен на классе <code>Socket</code> , чтобы обеспечить TCP-обслуживание на более высоком уровне. <code>TcpClient</code> предоставляет несколько методов для отправки и получения данных через сеть
<code>TcpListener</code>	Построен на низкоуровневом классе <code>Socket</code> . Его основное назначение — серверные приложения. Он ожидает входящие запросы на соединения от клиентов и уведомляет приложение о любых соединениях
<code>UdpClient</code>	Содержит методы для отправки и получения данных посредством протокола UDP

Основной класс — это класс `Socket`. Он обеспечивает базовую функциональность для работы с сокетами. На его базе построены некоторые другие классы. Члены класса `Socket` приведены в таблице 11.2.

Таблица 11.2. Члены класса `Socket`

Член	Описание
<code>Accept()</code>	Создаёт новый сокет для обработки входящего запроса на соединение
<code>AddressFamily</code>	Предоставляет семейство адресов сокета — значение из перечисления <code>Socket.AddressFamily</code>

Available	Возвращает объем доступных для чтения данных
Bind()	Связывает сокет с локальной конечной точкой для ожидания входящих запросов на соединение
Blocking	Дает или устанавливает значение, показывающее, находится ли сокет в блокирующем режиме
Close()	Закрывает сокет
Connect()	Устанавливает соединение с удаленным узлом
Connected	Возвращает значение, показывающее, соединен ли сокет с удаленным узлом
GetSocketOption()	Возвращает значение SocketOption.
IOControl()	Устанавливает для сокета низкоуровневые режимы работы
Listen()	Помещает сокет в режим прослушивания (ожидания). Используется только на сервере
LocalEndPoint	Сообщает локальную конечную точку
Poll()	Определяет состояние сокета
ProtocolType	Содержит тип протокола сокета
Receive()	Получает данные из сокета
RemoteEndPoint	Сообщает удаленную конечную точку сокета
Select()	Проверяет состояние одного или нескольких сокетов
Send()	Отправляет данные соединенному сокету
SetSocketOption()	Устанавливает опцию сокета
Shutdown()	Запрещает операции отправки и получения данных на соquete
SocketType	Содержит тип сокета

11.6. Конвертер валют

В качестве примера реального приложения мы сейчас попробуем создать конвертер валют, который будет получать актуальные курсы валют у Google. Теоретически, можно, немного изменив его код, настроить на работу с любым другим сервисом, но Google, в отличие от других сервисов, будет всегда. Уже готовая программа изображена на рис. 11.3.

Создайте форму, расположив на ней компоненты, как показано на рис. 11.3. В качестве списков используйте listBox, заполнив их элементами, как показано на рис. 11.4.



Рис. 11.3. Конвертер валют в действии

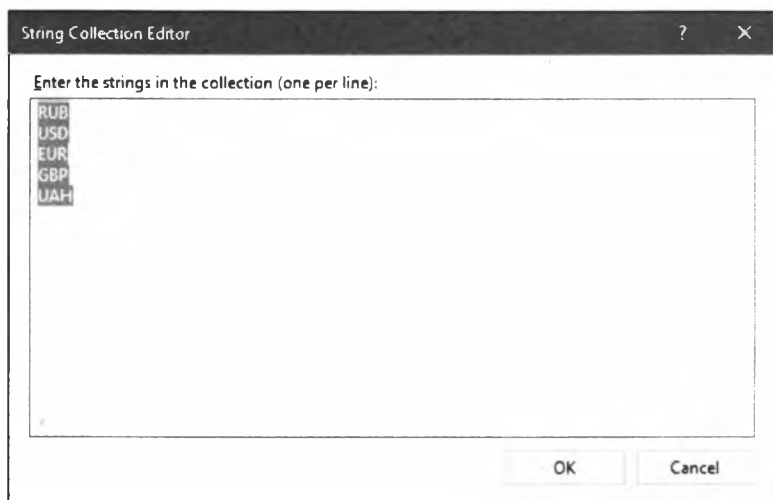


Рис. 11.4. Элементы списков валют

Компонент внизу формы - это webBrowser. Данный компонент позволяет получить содержимое любой веб-страницы. По сути, это уже готовый веб-браузер. Используя его метод Navigate(), можно загрузить любую страницу:

```
webBrowser1.Navigate("https://www.google.ru/search?q=" +
textBox1.Text + " " + from + " %D0%B2 " + to );
```

Мы пытаемся осуществить поиск Google по следующему запросу:

```
<содержимое текстового поля> исходная_валюта %D0%B2
результатирующая_валюта
```

from - это выбранная валюта в первом списке, а **to** - выбранная валюта во втором списке. Обработчик кнопки **Перевести** будет таким:

```
private void button1_Click(object sender, EventArgs e)
{
    string from, to;

    from = listBox1.SelectedItem.ToString();
    to = listBox2.SelectedItem.ToString();

    if (from==to)
    {
        MessageBox.Show("Это одна и та же валюта!", "Внимание!");
    }
    else if (textBox1.Text=="")
    {
        MessageBox.Show("Введите количество валюты!", "Внимание!");
    }
    else
    {
        webBrowser1.Navigate("https://www.google.ru/search?q=" +
        textBox1.Text + " " + from + " %D0%B2 " + to );
    }
}
```

Как видите, кроме перевода валюты, мы предусмотрели "защиту от дурака" - мы проверяем ввод пользователя и выбор валюты.

11.7. Простой сканер портов

Для закрепления материала о сокетах разработаем простейший сканер портов. Приложение будет очень простым и "сырым" - никакой обработки неправильного ввода, никакой обработки исключений. При необходимости вы всегда сможете сами "модернизировать" этот пример. Работающее приложение изображено на рис. 11.5.

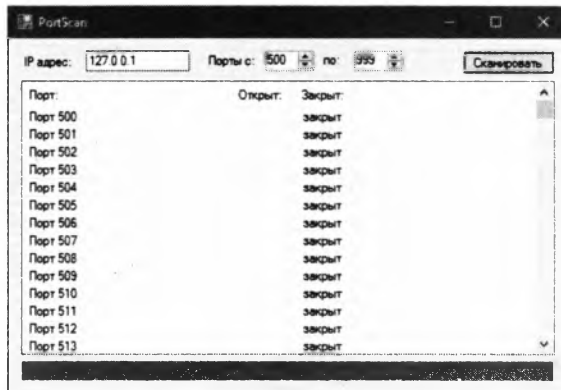


Рис. 11.5. Сканер портов

Принцип работы сканера портов прост: вы указываете IP-адрес, который нужно просканировать, вводите диапазон портов и нажимаете кнопку **Сканировать**. Сканер поочередно подключается к каждому из портов диапазона. Если соединение удалось, программа сообщает, что порт открыт, в противном случае - что закрыт.

Поместите на форме текстовое поле (имя `tIPAddress`), два компонента `NumericUpDown` с номерами `nBeginPort` и `nEndPort`. Основной рабочий компонент называется `listView1` - в него будут помещены результаты сканирования. В процессе сканирования индикатор прогресса (`progressBar1`) будет сообщать о ходе сканирования.

Обработчик кнопки **Сканировать** выглядит, как показано в листинге 11.3.

Листинг 11.3. Обработчик нажатия кнопки Сканировать

```
void Button1Click(object sender, EventArgs e)
{
    int BeginPort = Convert.ToInt32(nBeginPort.Value);
    EndPort = Convert.ToInt32(nEndPort.Value);

    int i;

    progressBar1.Maximum = EndPort - BeginPort + 1;

    progressBar1.Value = 0;
    listView1.Items.Clear();

    IPAddress addr = IPAddress.Parse(tIPAddress.Text);

    for (i = BeginPort; i <= EndPort; i++)
    {
        //Создаем и инициализируем сокет
        IPEndPoint ep = new IPEndPoint(addr, i);
        Socket soc = new Socket(AddressFamily.InterNetwork,
                                SocketType.Stream,
                                ProtocolType.Tcp);
        //Пытаемся соединиться с сервером
        IAsyncResult asyncResult = soc.BeginConnect(ep,
                                                    new AsyncCallback(ConnectCallback), soc);

        if (!asyncResult.AsyncWaitHandle.WaitOne(30, false))
        {
            soc.Close();
            listView1.Items.Add("Порт " + i.ToString());
            listView1.Items[i-BeginPort].SubItems.Add("");
        }
    }
}
```

```

        listView1.Items[i-BeginPort].SubItems.Add("закрыт");
        listView1.Refresh();
        progressBar1.Value += 1;

        }
        else
        {
            soc.Close();
            listView1.Items.Add("Порт " + i.ToString());
            listView1.Items[i-BeginPort].SubItems.Add("открыт");
            progressBar1.Value += 1;
        }
    }
}

```

Метод `BeginConnect()` асинхронно пытается подключиться к удаленному хосту. Ему нужно передать метод обратного вызова, который должен вызывать метод `EndConnect()`. Метод `EndConnect()` завершает запрос на соединение и вернет соединенный сокет.

Методу `BeginConnect()` нужно передать три параметра:

1. Удаленный хост
2. `AsyncCallback` - используется для передачи указателя на функцию
3. Сокет

Метод `CallBack` у нас будет называться `ConnectCallback()` и выглядеть так:

```

private static void ConnectCallback(IAsyncResult ar)
{
    try
    {
        Socket client = (Socket) ar.AsyncState;
        client.EndConnect(ar);
        connectDone.Set();
    }
    catch (Exception e)
    {
    }
}

```

В следующей главе мы рассмотрим еще более сложный пример - многопоточную программу-сервер.

Глава 12.

Создание приложения клиент/сервер

Основы языка C#,
первые программы

Клиент-серверные
приложения

Многопоточное
программирование

Создание мобильных
приложений на C#

12.1. Принцип работы приложения

Сейчас мы рассмотрим довольно сложное клиент/сервер приложение. Оно будет состоять из двух отдельных exe-файлов: сервера и клиента. Принцип работы сервера такой:

1. Открывает и назначает сокет
2. Прослушивает входящее соединение
3. Если есть запрос на входящее соединение от клиента, принимает его
4. Отправляет и получает данные
5. По окончании передачи/отправки данных возвращается к действию 2

Клиент действует иначе:

1. Открывает сокет
2. Подключается к серверу
3. Отправляет/получает данные
4. Закрывает сокет

12.2. Разработка серверной части

Первым делом нам нужно назначить конечную точку для сокета, то есть задать имя/IP-адрес узла (мы будем использовать localhost) и номер порта:

```
IPHostEntry ipHost = Dns.GetHostEntry("localhost");  
IPAddress ipAddr = ipHost.AddressList[0];  
IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 8888);
```

Как видите, с помощью `Dns.GetHostEntry()` мы получаем IP-адреса узла. Можно было бы указать просто IP-адрес 127.0.0.1, но это не совсем корректно. Затем мы создаем объект класса `EndPoint`, конструктору которого мы передаем запись типа `IPAddress` и номер порта 8888.

Пока все просто. Далее нам нужно создать сокет:

```
Socket sock = new Socket(ipAddr.AddressFamily, SocketType.  
Stream, ProtocolType.Tcp);
```

Конструктору класса `Socket` мы передаем информацию о семействе адресов, о типе сокета (мы используем потоковый сокет - `SocketType.Stream`), а также о типе протокола (мы используем TCP - `ProtocolType.Tcp`).

Далее нам нужно связать наш сокет с конечной точки и запустить “прослушку” сокета:

```
sock.Bind(ipEndPoint);  
sock.Listen(10);
```

Единственный параметр метода `Listen` задает максимальную длину очереди ожидающих подключений. После того как сокет настроен на прослушку, можно запускать цикл прослушки. Для этого мы используем бесконечный цикл `while`. В нем выполняется метод `Accept()`, который приостанавливает выполнение программы, ожидая входящее подключение:

```
Socket s = sock.Accept();
```

Как только будут получены данные от клиента, выполнение программы продолжится и мы можем прочитать эти данные с помощью метода `Receive()`:

```
s.Receive(bytes);
```

Данные, полученные от клиента, нужно будет декодировать, для этого мы используем метод `GetString` из `Encoding.UTF8`:

```
data += Encoding.UTF8.GetString(bytes, 0, bytesCount);
```

После этого мы выводим прочитанные данные на консоль и отправляем клиенту ответ. В ответе мы передаем количество полученных от клиента байтов. Отправка сообщения клиенту осуществляется методом `Send()`:

```
s.Send(msg);
```

Полный код серверного приложения приведен в листинге 12.1.

Листинг 12.1. Серверная часть приложения

```

using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace OneThreadServer
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.GetEncoding(866);
            Console.WriteLine("Однопоточный сервер запущен");
            // Подготавливаем конечную точку для сокета
            IPEndPoint ipHost = Dns.GetHostEntry("localhost");
            IPAddress ipAddr = ipHost.AddressList[0];
            IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 8888);

            // Создаем потоковый сокет, протокол TCP/IP
            Socket sock = new Socket(ipAddr.AddressFamily,
SocketType.Stream, ProtocolType.Tcp);
            try
            {
                // связываем сокет с конечной точкой
                sock.Bind(ipEndPoint);
                // начинаем прослушку сокета
                sock.Listen(10);

                // Начинаем слушать соединения
                while (true)
                {
                    Console.WriteLine("Слушаем, порт {0}", ipEndPoint);

                    // Программа приостанавливается,
                    // ожидая входящее соединение
                    // сокет для обмена данными с клиентом
                    Socket s = sock.Accept();

                    // сюда будем записывать полученные от клиента данные
                    string data = null;

                    // Клиент есть, начинаем читать от него запрос
                    // массив полученных данных
                    byte[] bytes = new byte[1024];

```

```
// длина полученных данных
int byteCount = s.Receive(bytes);

// Декодируем строку
data += Encoding.UTF8.GetString(bytes, 0, byteCount);

// Показываем данные на консоли
Console.WriteLine("Данные от клиента: " + data + "\n\n");

// Отправляем ответ клиенту
string reply = "Query size: " + data.Length.ToString()
              + " chars";
// кодируем ответ сервера
byte[] msg = Encoding.UTF8.GetBytes(reply);

// отправляем ответ сервера
s.Send(msg);

    if (data.IndexOf("<TheEnd>") > -1)
    {
        Console.WriteLine("Соединение завершено.");
        break;
    }

    s.Shutdown(SocketShutdown.Both);
    s.Close();
}
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
finally
{
    Console.ReadLine();
}
}
}
```

12.3. Приложение-клиент

Метод Main() будет очень прост:

```
try
{
```

```

        Communicate("localhost", 8888);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
    finally
    {
        Console.ReadLine();
    }
}

```

Мы просто запустим метод `Communicate`, передав ему имя сервера и номер порта, с которым нужно взаимодействовать. Сам метод `Communicate()` будет более сложным. В нем нужно первым делом подключиться к удаленному серверу:

```

IPHostEntry ipHost = Dns.GetHostEntry(hostname);
IPAddress ipAddr = ipHost.AddressList[0];
IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, port);

Socket sock = new Socket(ipAddr.AddressFamily, SocketType.
Stream, ProtocolType.Tcp);

// Подключаемся к серверу
sock.Connect(ipEndPoint);

```

Код должен быть вам уже знаком. Первым делом мы подготавливаем конечную точку для сокета, затем создаем сокет (параметры такие же, как в случае с сервером, иначе клиент и сервер просто не поймут друг друга), а после вместо метода `Listen()` мы вызываем метод `Connect()` для подключения к серверу.

Далее все просто: получаем данные от пользователя (которые нужно передать серверу), кодируем их и отправляем:

```
int bytesSent = sock.Send(data);
```

Мы также рекурсивно вызываем метод `Communicate()`, чтобы передать серверу следующее сообщение:

```

if (message.IndexOf("<TheEnd>") == -1)
    Communicate(hostname, port);

```

В реальном мире, скорее всего, вы не будете использовать рекурсию. Скорее всего, вы будете создавать графическое приложение, в котором в ней

не будет необходимости. В нем будет поле ввода и кнопка **Send**. Для отправки следующего сообщения серверу нужно будет ввести его в поле ввода и нажать кнопку **Send**. Здесь рекурсия нужна, чтобы пользователь смог отправить несколько сообщений серверу без перезапуска приложения клиента - если хотите, из соображений отладки кода. Чтобы завершить работу клиента и сервера, просто закройте их окна.

Полный код клиентского приложения приведен в листинге 12.2.

Листинг 12.2. Код приложения клиента

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace client
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.GetEncoding(866);
            try
            {
                Communicate("localhost", 8888);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.ToString());
            }
            finally
            {
                Console.ReadLine();
            }
        }

        static void Communicate(string hostname, int port)
        {
            // Буфер для входящих данных
            byte[] bytes = new byte[1024];
            // Соединяемся с удаленным сервером
            // Устанавливаем удаленную точку (сервер) для сокета
            IPHostEntry ipHost = Dns.GetHostEntry(hostname);
            IPAddress ipAddr = ipHost.AddressList[0];
```

```

        IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, port);

        Socket sock = new Socket(ipAddr.AddressFamily,
SocketType.Stream, ProtocolType.Tcp);

        // Подключаемся к серверу
        sock.Connect(ipEndPoint);

        Console.Write("Введите сообщение: ");
        string message = Console.ReadLine();

        Console.WriteLine("Подключаемся к порту {0} ",
sock.RemoteEndPoint.ToString());

        byte[] data = Encoding.UTF8.GetBytes(message);

        // Получаем к-во отправленных байтов
        int bytesSent = sock.Send(data);

        // Получаем ответ от сервера, bytesRec - к-во принятых байтов
        int bytesRec = sock.Receive(bytes);

        Console.WriteLine("\nОтвет сервера: {0}\n\n",
Encoding.UTF8.GetString(bytes, 0, bytesRec));

        // Вызываем Communicate() еще
        if (message.IndexOf("<TheEnd>") == -1)
            Communicate(hostname, port);

        // Освобождаем сокет
        sock.Shutdown(SocketShutdown.Both);
        sock.Close();
    }
}

```

Теперь посмотрим, что у нас получилось (рис. 12.1). Запустите сначала сервер, а затем - клиент. Введите строку, которую вы хотите передать серверу и посмотрите, какой ответ вы получите от сервера. Сервер отправляет клиенту количество символов в сообщении, полученном от клиента. Если клиент передает сообщение <TheEnd>, то это означает конец сеанса и сервер разрывает соединение с клиентом.

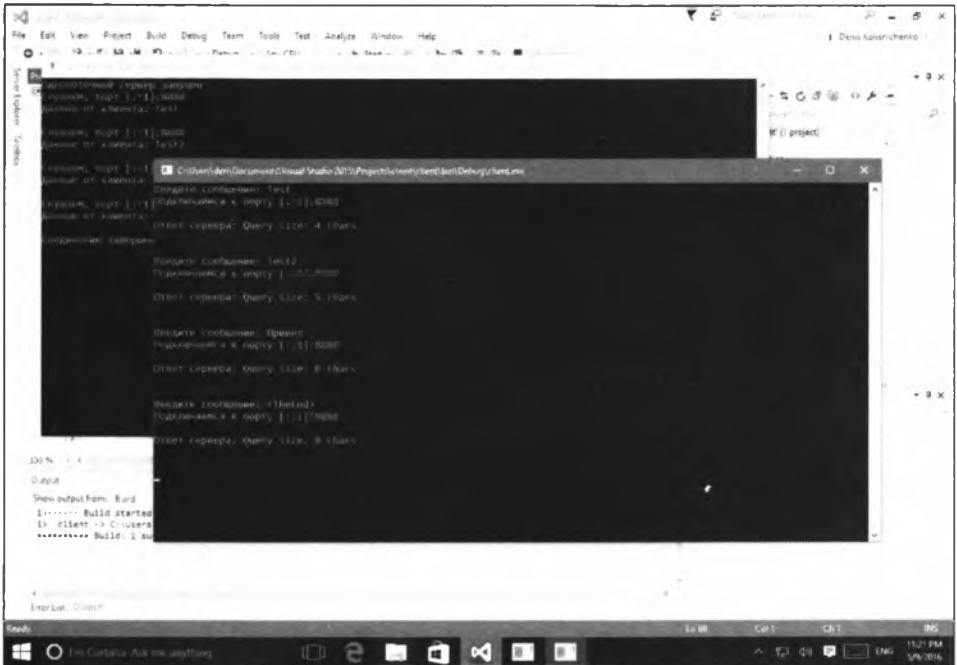


Рис. 12.1. Процесс взаимодействия клиента и сервера

12.4. Многопоточный сервер

Наше предыдущее приложение хорошо работает, но только не в реальном мире. Попробуйте запустить второго клиента и ввести сообщение. Программа не сможет подключиться к серверу, поскольку он уже занят первым клиентом (рис. 12.2).



Рис. 12.2. Неудачная попытка подключения второго клиента

В реальном мире к одному серверу может подключаться несколько клиентов и он должен успевать обслуживать все из них.

В этом разделе мы напишем многопоточный сервер. Многопоточность будет реализована через `ThreadPool`. Первым делом нужно настроить `ThreadPool`:

```
// Максимальное количество потоков - по 4 на процессор
int MaxThreadsCount = Environment.ProcessorCount * 4;
// Установим максимальное количество рабочих потоков
ThreadPool.SetMaxThreads(MaxThreadsCount, MaxThreadsCount);
// Установим минимальное количество рабочих потоков
ThreadPool.SetMinThreads(2, 2);
```

Принимать новых клиентов будем в бесконечном цикле. После того как клиент был принят, он передается в новый поток (`ClientThread`) с использованием пула потоков. Бесконечный цикл будет выглядеть так:

```
while (true)
{
    Console.WriteLine("\nОжидание соединения... ");
    ThreadPool.QueueUserWorkItem(ClientProcessing,
        server.AcceptTcpClient());
    counter++;
    Console.WriteLine("\nСоединение №" + counter.ToString() + "!");
}
```

Сам же сервер создается так:

```
Int32 port = 9595;
IPAddress localAddr = IPAddress.Parse("127.0.0.1");
server = new TcpListener(localAddr, port);
server.Start();
```

Метод `ClientProcessing()` получает запрос клиента и возвращает его в верхнем регистре. В отличие от предыдущего варианта, мы не будем выводить запрос клиента на консоль, чтобы не захламлять вывод сервера - ведь клиентов на этот раз будет много. Зато клиент, который мы немного изменим для этого случая, будет выводить, как запрос, так и ответ сервера.

```
// Буфер для принимаемых данных.
Byte[] bytes = new Byte[256];
String data = null;

TcpClient client = client_obj as TcpClient;

data = null;
```

```
// Получаем информацию от клиента
NetworkStream stream = client.GetStream();

int i;

// Принимаем данные от клиента
while ((i = stream.Read(bytes, 0, bytes.Length)) != 0)
{
    // Преобразуем данные в ASCII string.
    data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);

    // Преобразуем строку в верхний регистр
    data = data.ToUpper();

    // Преобразуем полученную строку в массив байт
    byte[] msg = System.Text.Encoding.ASCII.GetBytes(data);

    // Отправляем ответ клиенту
    stream.Write(msg, 0, msg.Length);
}

// Закрываем соединение.
client.Close();
```

Полный исходный код приложения-клиента приведен в листинге 12.3.

Листинг 12.3. Многопоточный сервер

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace MultiThreadServer
{
    class ExampleTcpListener
    {
        static void Main(string[] args)
        {
            TcpListener server = null;
            try
            {
                int MaxThreadsCount = Environment.ProcessorCount * 4;
                // Установим максимальное количество рабочих потоков
```

```

ThreadPool.SetMaxThreads(MaxThreadsCount, MaxThreadsCount);
// Установим минимальное количество рабочих потоков
ThreadPool.SetMinThreads(2, 2);

// Устанавливаем порт для TcpListener = 9595.
Int32 port = 9595;
IPAddress localAddr = IPAddress.Parse("127.0.0.1");
int counter = 0;
server = new TcpListener(localAddr, port);

Console.OutputEncoding = Encoding.GetEncoding(866);
Console.WriteLine("Конфигурация многопоточного сервера:");
Console.WriteLine("  IP-адрес   : 127.0.0.1");
Console.WriteLine("  Порт      : " + port.ToString());
Console.WriteLine("  Потоки    : " +
    MaxThreadsCount.ToString());
Console.WriteLine("\nСервер запущен\n");
// Запускаем TcpListener и начинаем слушать клиентов.
server.Start();
// Принимаем клиентов в бесконечном цикле.
while (true)
{
    Console.WriteLine("\nОжидание соединения... ");

    ThreadPool.QueueUserWorkItem(ClientProcessing,
        server.AcceptTcpClient());
    // Выводим информацию о подключении.
    counter++;
    Console.WriteLine("\nСоединение №" + counter.ToString() + "!");

}
}
catch (SocketException e)
{
    Console.WriteLine("SocketException: {0}", e);
}
finally
{
    // Останавливаем сервер
    server.Stop();
}

Console.WriteLine("\nНажмите Enter...");
Console.Read();
}

```

```

static void ClientProcessing(object client_obj)
{
    // Буфер для принимаемых данных.
    Byte[] bytes = new Byte[256];
    String data = null;
    TcpClient client = client_obj as TcpClient;

    data = null;

    // Получаем информацию от клиента
    NetworkStream stream = client.GetStream();

    int i;

    // Принимаем данные от клиента в цикле, пока не дойдем до конца.
    while ((i = stream.Read(bytes, 0, bytes.Length)) != 0)
    {
        // Преобразуем данные в ASCII string.
        data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);

        // Преобразуем строку к верхнему регистру.
        data = data.ToUpper();

        // Преобразуем полученную строку в массив байт.
        byte[] msg = System.Text.Encoding.ASCII.GetBytes(data);

        // Отправляем данные обратно клиенту (ответ).
        stream.Write(msg, 0, msg.Length);
    }

    // Закрываем соединение.
    client.Close();
}
}

```

Теперь немного переделаем наш клиент. Что такое клиент? Это приложение, вызвавшее конструктор класса `TcpListener`. Мы переделаем наш клиент так, чтобы в нем создавалось несколько объектов класса `TcpListener`, то есть устанавливалось сразу несколько подключений. В результате вам не придется вручную запускать несколько клиентов. В цикле вы будете регулировать количество клиентов и смотреть, как их обслуживает сервер. Ведь взять тот же браузер - в его окне можно открыть несколько вкладок и подключиться к веб-серверу. Каждая вкладка создаст новое соединение к серверу и для сервера будет выглядеть, по сути, как отдельный клиент. То же самое мы воссоздадим и в нашем случае. В листинге 12.4 представлена

модифицированная версия клиента, создающая несколько подключений к нашему серверу.

Листинг 12.4. Клиент для многопоточного сервера

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;

namespace NewClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.GetEncoding(866);
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("\n Соединение # "+i.ToString()+"\n");
                Connect("127.0.0.1", "Hello World! #"+i.ToString());
            }
            Console.WriteLine("\n Нажмите Enter...");
            Console.Read();
        }

        static void Connect(String server, String message)
        {
            try
            {
                // Создаём TcpClient.
                // Для созданного в предыдущем проекте TcpListener
                // Настраиваем его на IP нашего сервера и тот же порт.

                Int32 port = 9595;
                TcpClient client = new TcpClient(server, port);

                // Переводим наше сообщение в ASCII, а затем в массив Byte.
                Byte[] data = System.Text.Encoding.ASCII.GetBytes(message);

                // Получаем поток для чтения и записи данных.
                NetworkStream stream = client.GetStream();
```

```

        // Отправляем сообщение нашему серверу.
        stream.Write(data, 0, data.Length);
        Console.WriteLine("Отправлено: {0}", message);

        // Получаем ответ от сервера.

        // Буфер для хранения принятого массива bytes.
        data = new Byte[256];

        // Строка для хранения полученных ASCII данных.
        String responseData = String.Empty;

        // Читаем первый пакет ответа сервера.
        // Можно читать всё сообщение.
        // Для этого надо организовать чтение в цикле как на сервере.
        Int32 bytes = stream.Read(data, 0, data.Length);
        responseData = System.Text.Encoding.ASCII.
GetString(data, 0, bytes);
        Console.WriteLine("Получено: {0}", responseData);

        // Закрываем всё.
        stream.Close();
        client.Close();
    }
    catch (ArgumentNullException e)
    {
        Console.WriteLine("ArgumentNullException: {0}", e);
    }
    catch (SocketException e)
    {
        Console.WriteLine("SocketException: {0}", e);
    }
}
}
}

```

В цикле вы можете установить другое количество клиентов, по умолчанию у нас будет 5 клиентов. Метод Connect() занимается всей обработкой процесса отправки запроса и получения ответа от сервера. Он подключается к серверу, отправляет запрос, получает ответ и выводит его. Код довольно простой и уже должен быть понятен вам.

Теперь посмотрим, что же у нас получилось. Запустите сервер и клиент. Результат изображен на рис. 12.3. Как видите, к серверу поступило 5 соединений, все они были обработаны.

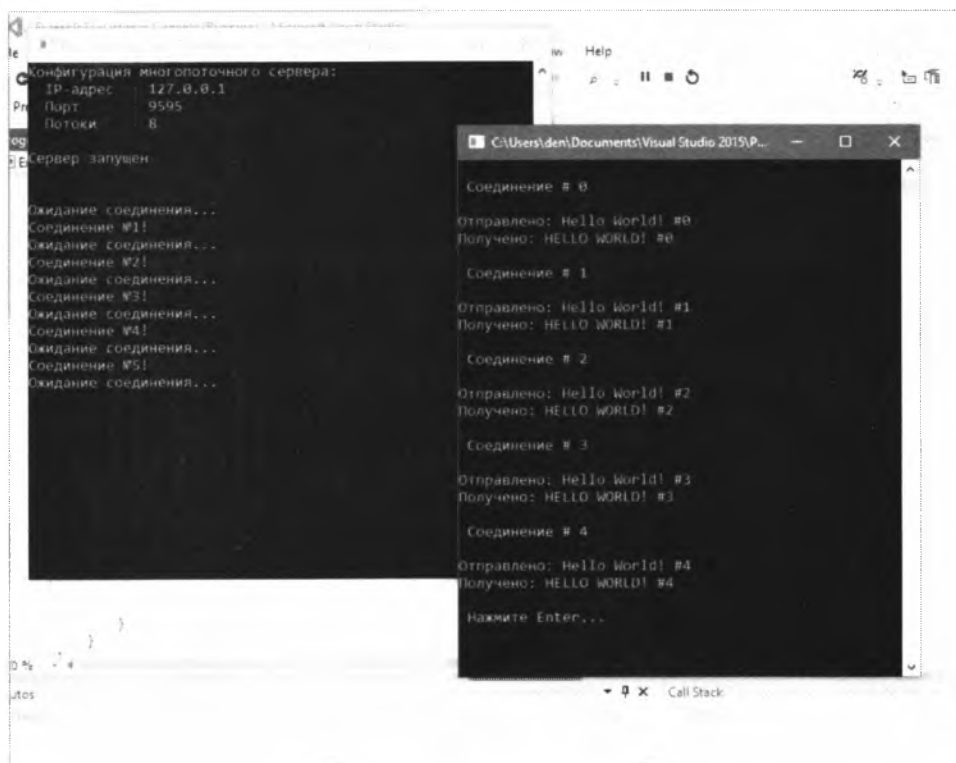


Рис. 12.3. Многопоточный сервер в работе

Глава 13.

Разработка приложений для планшета под управлением Windows 10

Основы языка C#,
первые программы

Клиент-серверные
приложения

Многопоточное
программирование

Создание мобильных
приложений на C#

13.1. Подготовка к созданию мобильных приложений

С помощью Visual Studio можно разработать самые разнообразные приложения - консольные приложения, классические Windows-приложения (Windows Forms), Android-приложения и др. Конечно же, с помощью Visual Studio можно разработать приложения, работающие на планшетах под управлением Windows 10. Создание такого приложения и будет рассмотрено в этой главе.

Забегая наперед, хочется рассказать о разработке приложений под Windows 10. Мои впечатления можно описать одним словом: **Восторг!** У меня есть опыт разработки Android-приложений в Android Studio. Совсем недавно я для собственных нужд разработал простенькое приложение в Android Studio, вычисляющее расход топлива автомобиля. Приложение очень простое, но, учитывая торможение Android Studio на моем не очень слабом компьютере, на его разработку ушел час. А вот в Visual Studio подобное приложение было разработано за 20 минут, причем я даже успевал делать



Рис. 13.1. Средства разработчика мобильных приложений не установлены

скриншоты для этой главы. Расположение элементов на форме, удобная разметка, простой код. В общем, я серьезно задумался, что следующее мое Android-приложение будет написано на C#.

Однако перед началом разработки мобильных приложений нужно подготовить ваш компьютер к этому процессу. Откройте окно создания нового проекта (**File » New » Project/Solution**) и выберите шаблон **Visual C# » Windows » Universal** (рис. 13.1). Вы увидите, что средства разработки не установлены. Выберите **Install Universal Windows Tools** и нажмите кнопку **OK**. В появившемся окне нажмите кнопку **Install** (рис. 13.2)



Рис. 13.2. Нажмите кнопку Install

Появится окно установщика Visual Studio. Вы уже видели его при установке IDE (рис. 13.3). Нажмите кнопку **Update**. Если вы не хотите по окончании установки перезагружать компьютер, завершите работу Visual Studio (рис. 13.4) и нажмите кнопку **Retry**.



Рис. 13.3. Нажмите кнопку Update



Рис. 13.4. Завершите работу Visual Studio и нажмите кнопку Retry

Далее вам будет предложено выбрать необходимые компоненты. Нам понадобятся все. Просто нажмите кнопку **Next** (рис. 13.5). Кстати, на вашем жестком диске понадобится примерно 9 Гб пространства - довольно много, но ничего с этим не поделаешь. В следующем окне нажмите кнопку **Update**.



Рис. 13.5. Выбор устанавливаемых компонентов



Рис. 13.6. Нажмите кнопку Update

Начнется мучительный процесс загрузки и установки программного обеспечения. Ожидание всегда мучительно. Время зависит от скорости вашего интернет-соединения (рис. 13.7).



Рис. 13.7. Загрузка и установка ПО



Рис. 13.8. Установка завершена

Рано или поздно вы увидите заветное сообщение (рис. 13.8).

Запустите Visual Studio и откройте окно создания нового проекта. Теперь в разделе **Visual C# » Windows » Universal** будут новые шаблоны (рис. 13.9).



Рис. 13.9. Окно создания нового проекта

Далее вы увидите сообщение, что нужно перевести ваше устройство в режим разработчика (рис. 13.10). Для этого нажмите ссылку **settings for developers**.

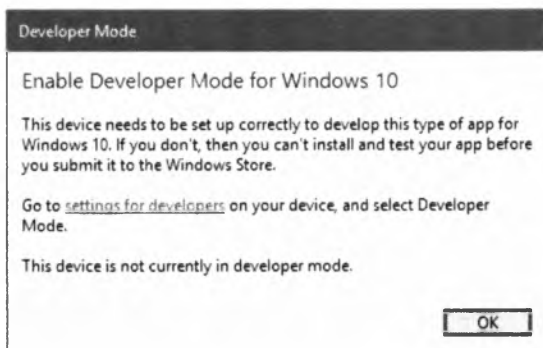


Рис. 13.10. Нужно перевести ваше устройство в режим разработчика

Откроется окно **Settings**. Выберите режим **Developer mode**. В появившемся окне нужно нажать **Yes**, чтобы подтвердить включение режима разработчика.

Убедитесь, что режим разработчика включен, и закройте окно **Settings**. Вы вернетесь в окно, изображенное на рис. 13.10. Нажмите кнопку **OK**.

Будет создан новый проект. Вот теперь можно приступить к разработке вашего первого мобильного приложения.

13.2. Проектирование графического интерфейса

Поскольку приложение у нас будет графическим, то и разработку его следует начать с разработки графического интерфейса. В области **Solution Explorer** дважды щелкните на **MainPage.xaml**. Откроется экран проектирования интерфейса (рис. 13.11). Обратите внимание, что вы можете редактировать разметку приложения, как графически, располагая на нем элементы, так и с помощью редактирования XAML-файла разметки, подобно тому, как это принято делать в **Android Studio** (там тоже есть два режима разработки интерфейса). Вот только если в **Android Studio** лично мне удобнее работать с XML-файлом разметки, то в **Visual Studio** есть превосходный графический редактор формы.

В верхнем левом углу выберите устройство, для которого вы проектируете приложение, в данный момент выбран планшет с размером экрана 8" (рис.

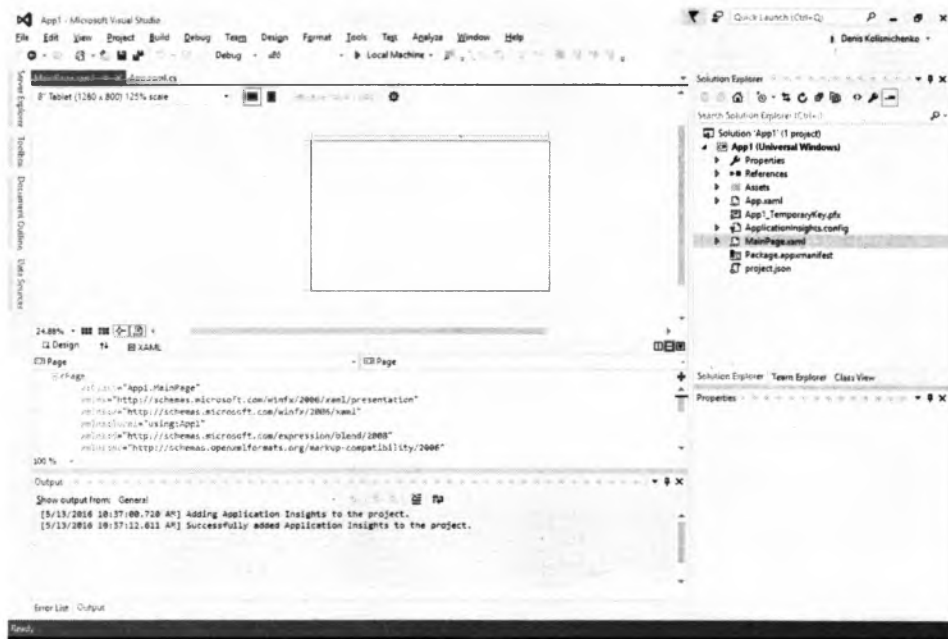


Рис. 13.11. Режим проектирования страницы приложения



Рис. 13.12. Панель Toolbox

13.11). Далее откройте панель Toolbox для выбора элементов графического интерфейса (рис. 13.12).

Чтобы вам было проще и удобнее, увеличьте масштаб страницы (Ctrl + колесико мыши). Расположите на странице компонент textBlock (рис. 13.13). Вместо тривиального приложения “Hello, world” мы разработаем программу вычисления расхода автомобиля.



Рис. 13.13. Первый компонент размещен на странице

Справа, как и при проектировании приложения Windows Forms, находится панель **Properties**, где вы можете изменять свойства компонентов. Измените свойство **Text** нашей надписи - "Километраж". На вкладке **Text** (панель **Properties**) установите размер шрифта - 16 pt (рис. 13.14).

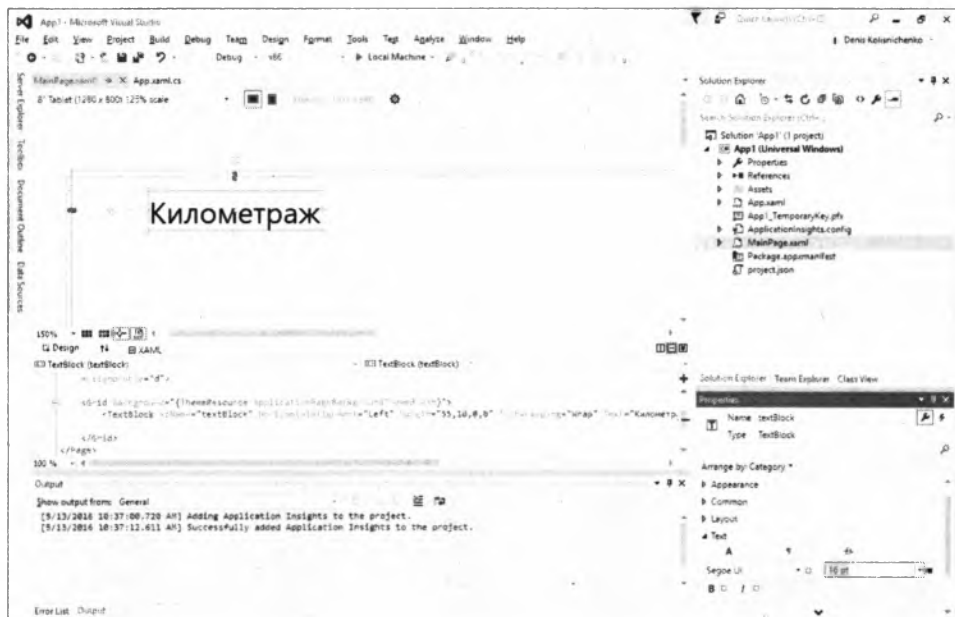


Рис. 13.14. Панель свойств компонента



Рис. 13.15. Расположение компонентов на странице

Расположите компоненты `textBlock` (надпись) и `textBox` (поле ввода) так, как показано на рис. 13.15. Также добавьте кнопку (компонент **Button**) с текстом **Вычислить** (свойство **Content**). Обратите внимание, как удобно организовано выравнивание компонентов на страницу. Android Studio далеко до такого удобства. Расположение каждого компонента существенно упрощается направляющими, которые напоминают направляющие Adobe Photoshop.

Теперь задайте значения по умолчанию для полей ввода - 100, 10 и 10, как показано на рис. 13.16. Для первого поля ввода установите имя (свойство **Name**) - *probeg*, для второго - *litres*, для третьего - *rashod*.

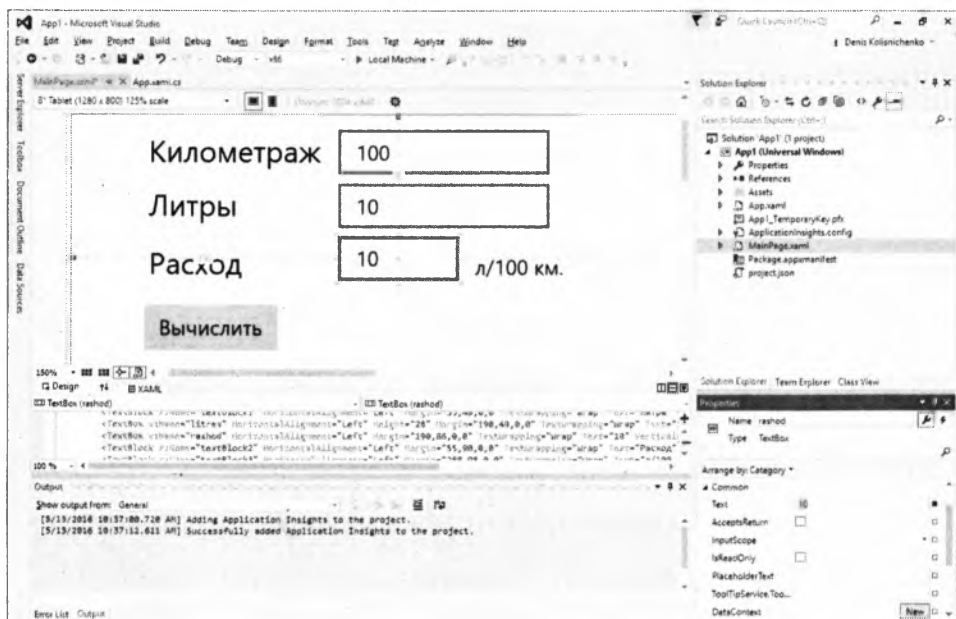


Рис. 13.16. Значения полей по умолчанию

В листинге 13.1 приведен код файла разметки `MainPage.xaml`.

Листинг 13.1. Файл `MainPage.xaml`.

```
<Page
  x:Class="Appl.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Appl"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
```

```

<Grid Background="{ThemeResource ApplicationPageBackground
ThemeBrush}">
    <TextBlock x:Name="textBlock"
HorizontalAlignment="Left" Margin="55,10,0,0"
TextWrapping="Wrap" Text="Километраж" VerticalAlignment="Top"
FontSize="21.333"/>
    <TextBox x:Name="probeg" HorizontalAlignment="Left"
Height="28" Margin="190,10,0,0" TextWrapping="Wrap" Text="100"
VerticalAlignment="Top" Width="151"/>
    <TextBlock x:Name="textBlock1"
HorizontalAlignment="Left" Margin="55,49,0,0"
TextWrapping="Wrap" Text="Литры" VerticalAlignment="Top"
FontSize="21.333"/>
    <TextBox x:Name="litres" HorizontalAlignment="Left"
Height="28" Margin="190,49,0,0" TextWrapping="Wrap" Text="10"
VerticalAlignment="Top" Width="151"/>
    <TextBox x:Name="rashod" HorizontalAlignment="Left"
Margin="190,86,0,0" TextWrapping="Wrap" Text="10"
VerticalAlignment="Top" Width="86"/>
    <TextBlock x:Name="textBlock2"
HorizontalAlignment="Left" Margin="55,90,0,0"
TextWrapping="Wrap" Text="Расход" VerticalAlignment="Top"
FontSize="21.333"/>
    <TextBlock x:Name="textBlock3"
HorizontalAlignment="Left" Margin="288,98,0,0"
TextWrapping="Wrap" Text="л/100 км." VerticalAlignment="Top"/>
    <Button x:Name="button" Content="Вычислить"
HorizontalAlignment="Left" Margin="52,135,0,0"
VerticalAlignment="Top" Click="button_Click"/>

</Grid>
</Page>

```

13.3. Написание кода приложения

Осталось написать обработчик для единственной кнопки. Дважды щелкните на кнопке и введите следующий код:

```

double dProbeg = Double.Parse(probeg.Text);
double dLitres = Double.Parse(litres.Text);
double dRes = 100 * dLitres / dProbeg;
rashod.Text = dRes.ToString("###.##");

```

Сначала мы получаем значения километража и литров. Сразу используем тип **double**, чтобы избежать лишнего преобразования типов. Затем мы вычисляем расход топлива и отображаем его в поле *rashod*, попутно преобразовав его в тип **string**. Полный код приложения приведен в листинге 13.2.

Листинг 13.2. Полный код приложения

```

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;

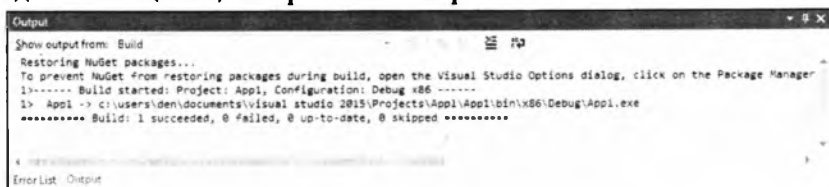
namespace Appl
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }

        private void button_Click(object sender, RoutedEventArgs e)
        {
            double dProbeg = Double.Parse(probeg.Text);
            double dLitres = Double.Parse(litres.Text);
            double dRes = 100 * dLitres / dProbeg;
            rashod.Text = dRes.ToString("##.##");
        }
    }
}

```

13.4. Компиляция и запуск приложения

Осталось откомпилировать и запустить приложение. Сначала откомпилируем приложение (меню **Build**). В результате успешной компиляции будет выведено сообщение, изображенное на рис. 13.17.

**Рис. 13.17. Успешная компиляция проекта**

Далее нажмите кнопку **Local Machine** (также на ней изображена кнопка Play зеленого цвета) для запуска приложения. На рис. 13.18 изображено наше готовое приложение. В верхнем левом углу находятся координаты указателя, при запуске на планшете их не будет.

Поздравляю! Вы только что создали свое первое приложение для планшета, работающего под управлением Windows 10!

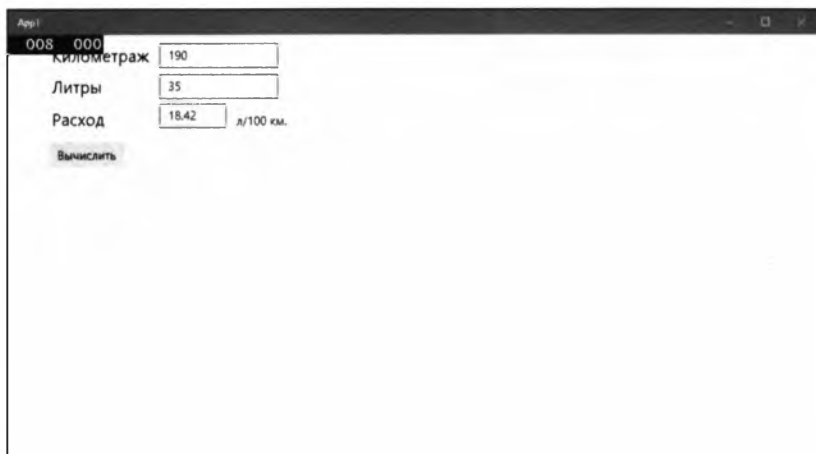


Рис. 13.18. Готовое приложение

Глава 14. Работаем с базой данных

Visual Studio содержит компоненты для работы с базами данных Microsoft Access и Microsoft SQL Server. В этой главе будет показано, как подключиться к базе данных Microsoft Access.

Прежде чем продолжить, нужно отметить, что Visual Studio умеет подключаться к базам данных Access как в новом формате (ACCDB-файл), так и в старом формате (MDB-файл).

14.1. Подключение источника данных

Для работы с базой данных первым делом нужно подключить источник данных. Мы считаем, что приложение Windows Forms у вас уже создано (рис. 14.1).



Рис. 14.1. Создание приложения Windows Forms

Поместите на форму компонент **DataGridView**. Вы сразу увидите область **DataGridView Задачи** (рис. 14.2). Откройте список **Выберите источник данных** и нажмите ссылку **Добавить источник данных проекта** (рис 14.3). Есть и альтернативный способ добавления источника данных: нужно выбрать команду **Вид > Другие окна > Источники данных**. Появится ссылка **Добавить новый источник данных**. При ее нажатии откроется мастер добавления источника данных, который можно будет потом выбрать в области **DataGridView > Задачи**.

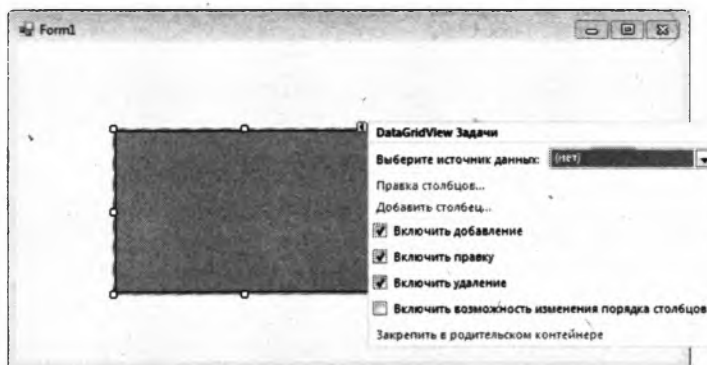


Рис. 14.2. Задачи DataGridView

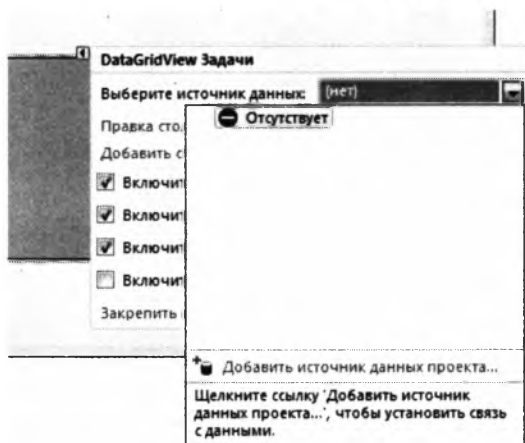


Рис. 14.3. Ссылка Добавить источник данных проекта

Итак, щелкните по ссылке **Добавить источник данных** проекта. Откроется окно **Мастер настройки источника данных**, на странице **Выбор типа источника данных** которого нужно выбрать **База данных** (рис. 14.4).

Далее выберите **Набор данных** и нажмите кнопку **Далее** (рис. 14.5). На странице **Выбор подключения к базе данных** нажмите кнопку **Создать подключение** (рис. 14.6). Далее нужно выбрать **<другое>** в качестве **Источник данных**, а в списке **Поставщик данных** - выбрать

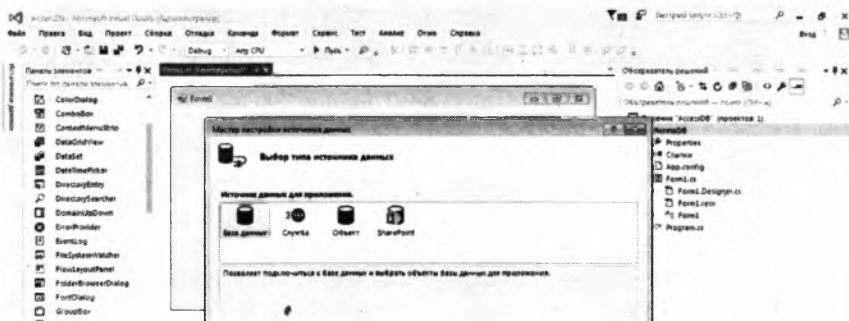


Рис. 14.4. Мастер настройки источника данных

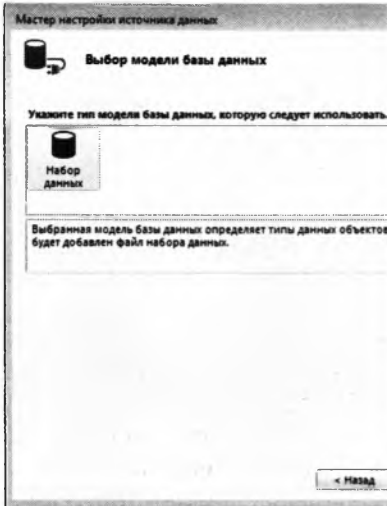


Рис. 14.5. Выберите Набор данных

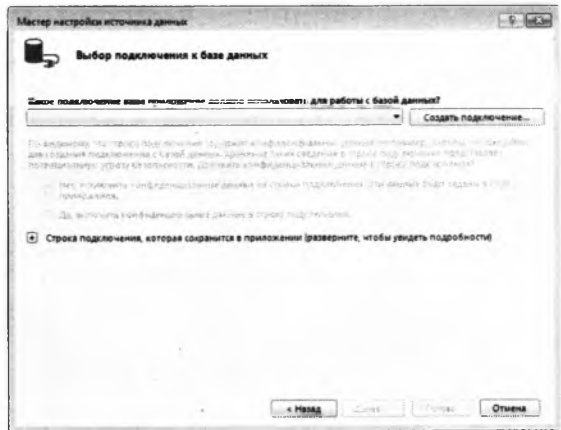


Рис. 14.6. Нажмите кнопку Создать подключение

Поставщик данных .NET Framework. Обратите внимание: здесь можно выбрать и другой источник данных, например, **Microsoft SQL Server**, **База данных Oracle** и др. Мы же рассматриваем пример подключения к БД Access, поэтому нам нужно выбрать <другое>.

После нажатия кнопки **Продолжить** нужно из списка **Поставщик OLE DB** выбрать **Microsoft Office 15.0 Access Data Provider** (версия MS Office у вас может отличаться), а в поле **Имя сервера или файла** введите имя файла базы данных MS Access (рис. 14.8). Вы можете нажать кнопку **Проверить**

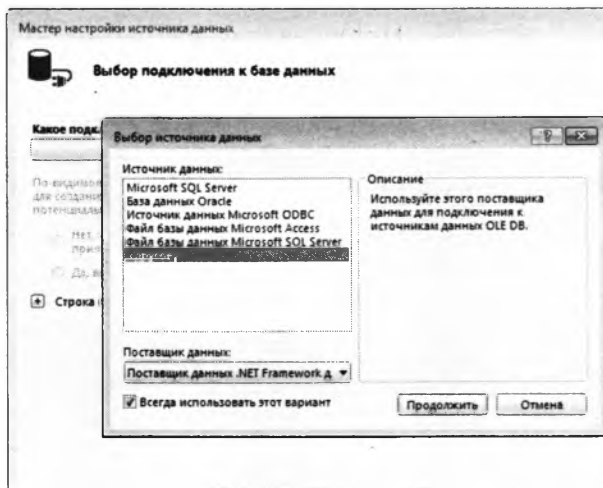


Рис. 14.7. Выбор поставщика данных



Рис. 14.8. Имя файла базы данных

подключение, чтобы убедиться, что имя файла указано правильно (или что вы правильно указали параметры для удаленного подключения к серверу).

Затем вы увидите имя базы данных, а также название провайдера данных в строке подключения (рис. 14.9). Нажмите кнопку **Далее**.

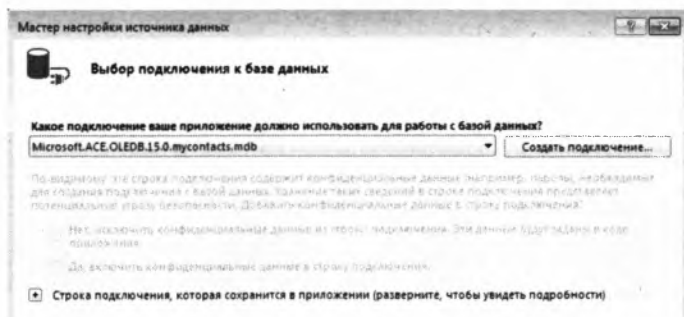


Рис. 14.9. Имя базы данных и название провайдера данных

Если база данных находится не в каталоге проекта, Visual Studio спросит вас, хотите ли вы скопировать базу данных в каталог проекта (рис. 14.10). Здесь решение принимать только вам. Удобнее, конечно, когда база данных находится в одном каталоге с проектом, особенно это хорошо, когда вы будете создавать инсталлятор - все файлы проекта будут в одном месте, и вы ничего не забудете.

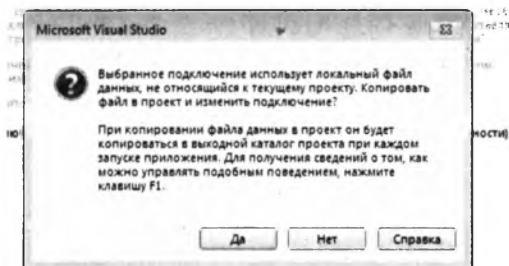


Рис. 14.10. Добавить базу данных в каталог проекта?

Введите название подключения - под этим названием оно будет доступно в вашем проекте - и нажмите кнопку **Далее**. Вам осталось выбрать таблицы и представления, которые вы будете использовать в своей программе. Выберите таблицу **contacts** и все ее поля (столбцы). Нажмите кнопку **Готово**.

Все, компонент DataGridView будет успешно добавлен в ваш проект. Запустите программу. Да, прямо сейчас. Вы увидите, что компонент DataGridView автоматически заполнен данными из базы данных (рис. 14.14)!

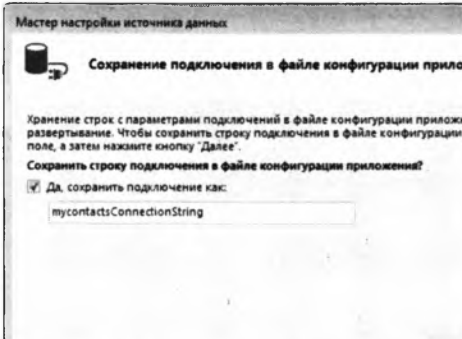


Рис. 14.11. Название подключения

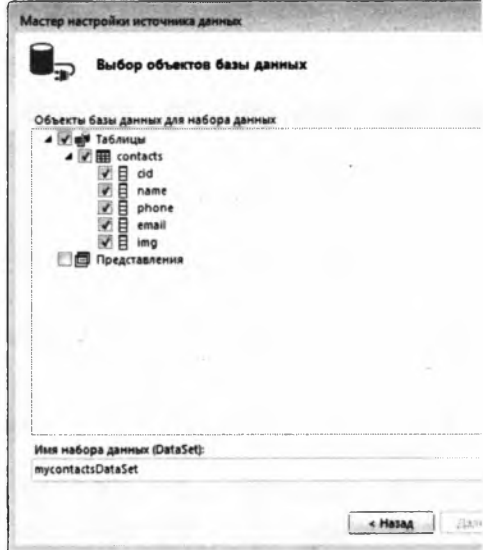


Рис. 14.12. Выбор таблиц и представлений базы данных. Обратите внимание на имя набора данных! Оно понадобится далее

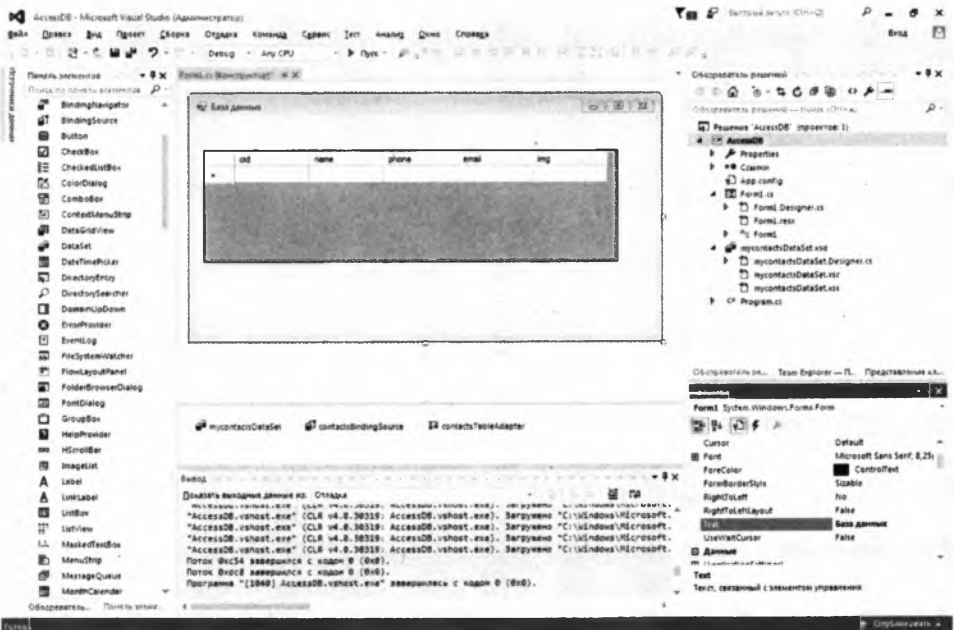


Рис. 14.13. Компонент DataGridView настроен

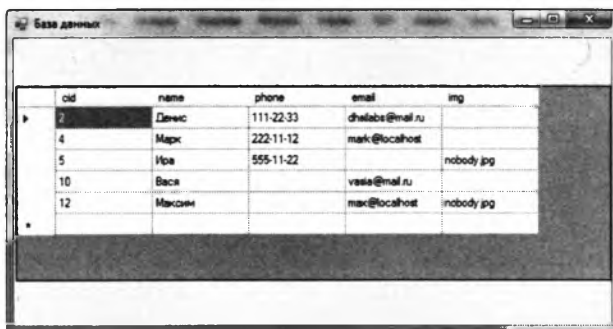


Рис. 14.14. Программа запущена

На самом деле автоматическая загрузка данных из БД осуществляется благодаря вызову метода `Fill()` из события `Load` формы:

```
private void Form1_Load(object sender, EventArgs e)
{
    // TODO: данная строка кода позволяет загрузить данные в таблицу
    «mycontactsDataSet.contacts». При необходимости она может быть
    перемещена или удалена.
    this.contactsTableAdapter.Fill(this.mycontactsDataSet.contacts);
}
```

Но данные строки за нас написал VisualStudio.

14.2. Сохранение данных

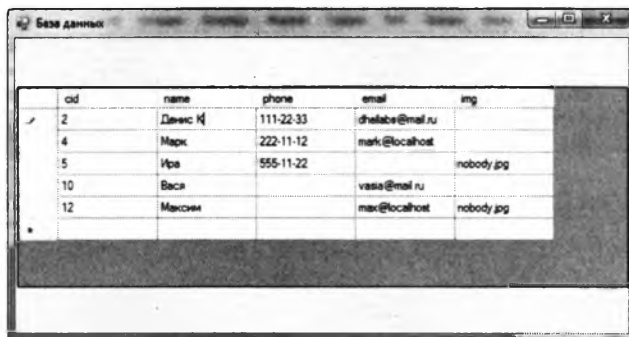


Рис. 14.15. Пользователь редактирует строку

Программа запущена и работает. Вы можете не только просматривать, но и редактировать строки (рис. 14.15). И заметьте: мы еще не написали ни одной строки кода.

Все хорошо в нашей программе, но она не сохраняет измененные данные. Редактировать данные вы можете, а вот сохранять - нет. Давайте добавим на форму кнопку с надписью **Сохранить**. Посмотрите на рис. 14.16. Кнопка

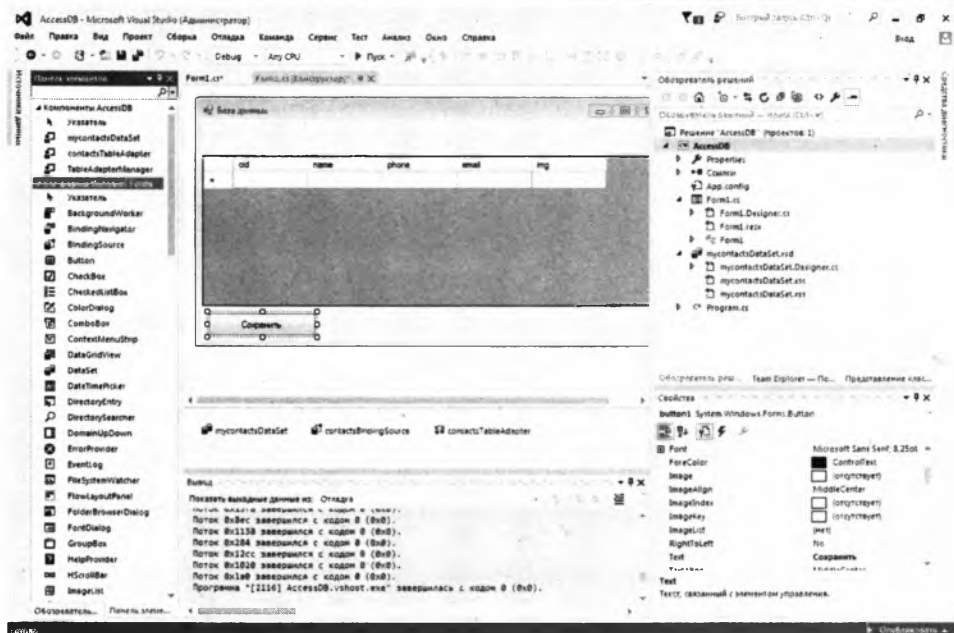


Рис. 14.16. Кнопка и служебные компоненты

уже добавлена. Ниже отображаются служебные компоненты, которые были добавлены автоматически при добавлении источника данных:

- mycontactsDataSet - источник данных;
- contactsBindingSource - используется для привязки к источнику данных;
- contactsTableAdapter - адаптер таблицы, создается для каждой таблицы базы данных. Если вы добавили базу данных с несколькими таблицами, то компонент TableAdapter будет создан для каждой из таблиц.

Установите имя кнопки SaveButton. Дважды щелкните по ней, чтобы открыть редактор кода и установить обработчик нажатия кнопки:

```
private void button1_Click(object sender, EventArgs e)
{
    contactsTableAdapter.Update(mycontactsDataSet);
}
```

Запустите программу и попробуйте сохранить данные, закройте программу и запустите ее снова. Убедитесь, что данные сохранены (рис. 14.17).

14.3. Изменение заголовков столбцов таблицы

Наша программа запускается, позволяет редактировать и сохранять данные. Уже неплохо. Особенно, если учесть, что мы написали всего одну строчку

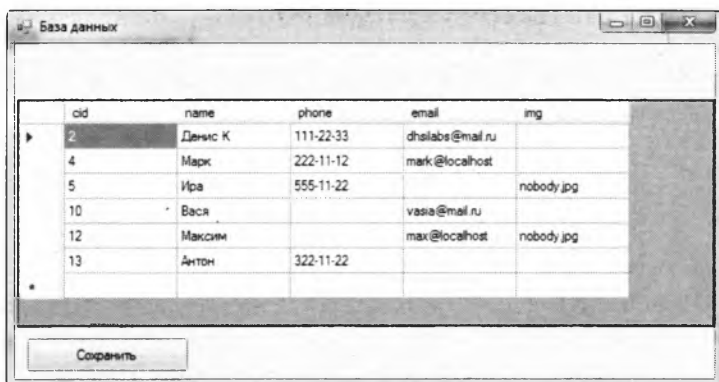


Рис. 14.17. Изменена одна строка, добавлена новая строка

кода. Но по-прежнему режут глаз названия полей таблицы. Для их редактирования щелкните по компоненту DataGridView и откройте свойство **Columns** (щелкните на кнопку ... напротив названия свойства). Откроется окно редактирования столбцов таблицы (рис. 14.18).

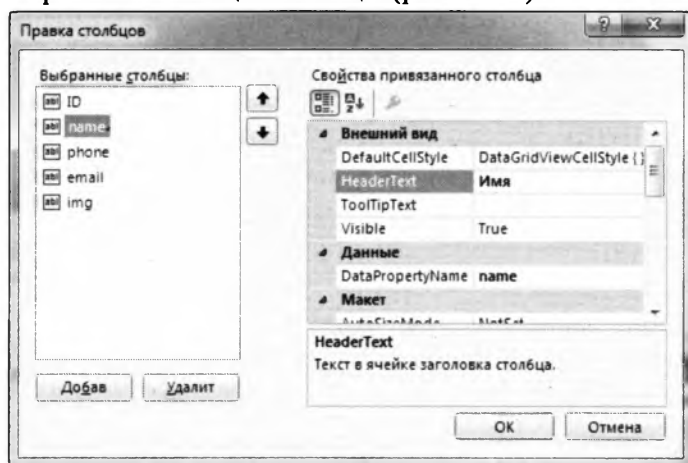


Рис. 14.18. Редактирование столбцов таблицы

Выделите столбец и установите его свойства. Вот некоторые самые полезные из них:

- **HeaderText** - задает название столбца.
- **Visible** - будет ли заголовок виден или нет.
- **Width** - ширина столбца.
- **Resizable** - можно ли изменять ширину столбца.

- **SortMode** - режим сортировки:
 - **NotSortable** - без сортировки;
 - **Automatic** - автоматически;
 - **Programmatic** - программная сортировка (определяется программистом).

На рис. 14.19 показано, что заголовки столбцов теперь на русском языке, а столбец **img** скрыт (**Visible = false**).

14.4. Защита от случайного удаления

Для удаления записи пользователю нужно выделить запись (строку) и нажать **Delete**. При этом произойдет событие **UserDeletingRow** компонента **DataGridView**, а после удаления - **UserDeletedRow**. В обработчике события **UserDeletingRow** можно вызвать диалог подтверждения удаления записи. В этом случае мы защитим данные от случайного удаления пользователем.

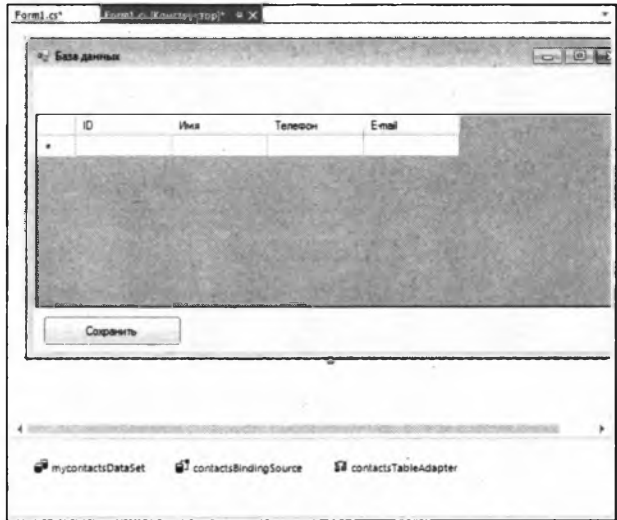


Рис. 14.19. Заголовки столбцов на русском

Перейдите к компоненту **DataGridView**, откройте список событий и установите обработчик для события **UserDeletingRow**:

```
private void dataGridView1_UserDeletingRow(object sender,
DataGridViewRowCancelEventArgs e)
{
    DialogResult dr = MessageBox.Show("Удалить запись?",
        "Удаление",
        MessageBoxButtons.OKCancel,
        MessageBoxIcon.Warning,
        MessageBoxDefaultButton.Button2);
    if (dr == DialogResult.Cancel)
    {

```

```

        e.Cancel = true;
    }
}

```

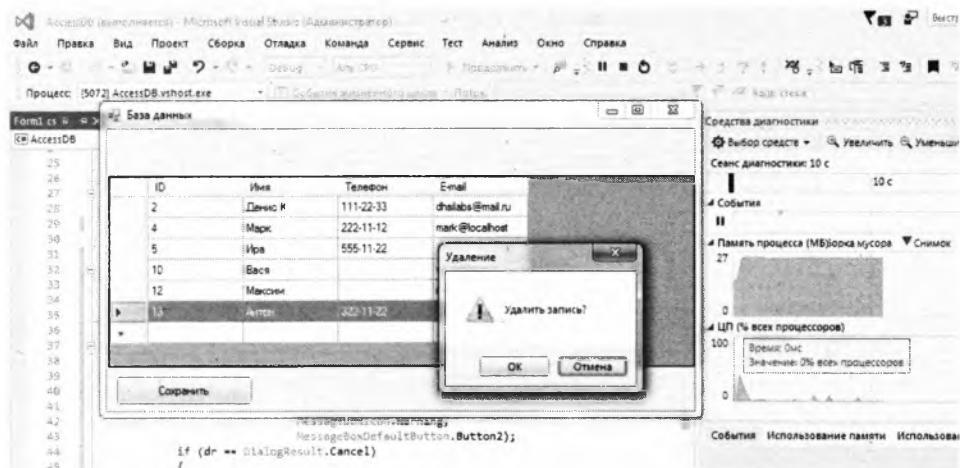


Рис. 14.20. Диалог при попытке удаления строки

14.5. Добавление данных

Теперь напишем код добавления данных в нашу таблицу. Сейчас добавление осуществляется средствами самого DataGridView, но это несколько неправильно, нужно знать, как запрограммировать добавление данных. Заранее рассмотрим взаимодействие между формами в C#.

Добавьте новую форму в проект командой **Проект » Добавить форму Windows** (рис. 14.21). Выберите **Форма Windows Forms** и установите имя **AddForm.cs**.

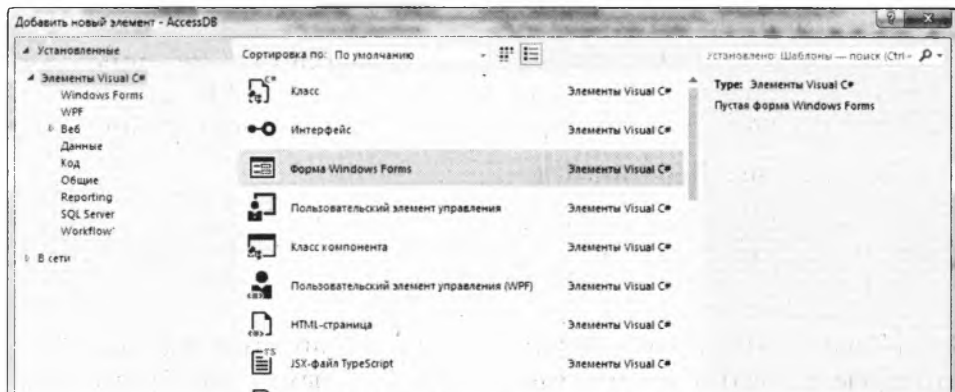


Рис. 14.21. Добавление новой формы

На новой форме разместите компоненты согласно таблице 14.1.

Таблица 14.1. Свойства компонентов

Описание	Имя компонента	Название свойства	Значение
Форма	AddForm	StartPosition	CenterParent
		Text	Форма добавления записи
Фаска	groupBox1	Text	Добавление записи
Надписи	label1 - label4	Text	Имя, Телефон, E-mail и Фото соответственно
Поля ввода		(Name)	tbName, tbPhone, tbMail, tbPhoto соответственно
Кнопка добавления	AddBtn	Text	Добавить
Кнопка закрытия	CloseBtn	Text	Заккрыть

В результате у вас должна получиться форма, изображенная на рис. 14.22.

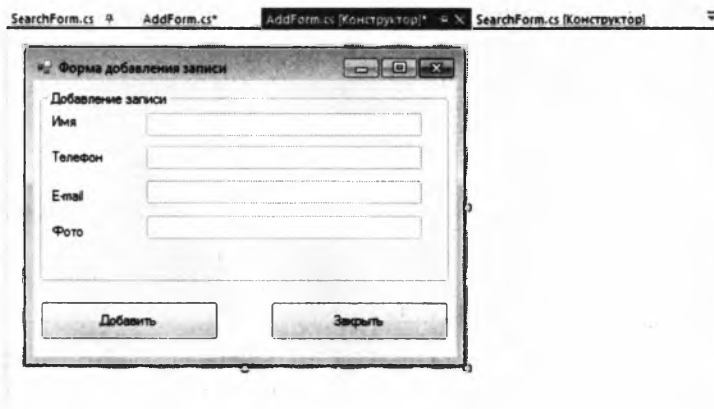


Рис. 14.22. Форма добавления записи

Теперь на главную форму добавьте две кнопки - **Найти** и **Добавить**. Первая будет использоваться для поиска записи (см. след. пункт), а вторая - для добавления новой записи программным путем, а не через форму.

Обработчик кнопки **Добавить** будет выглядеть так:

```
AddForm af = new AddForm();
af.Owner = this;
af.Show();
```

Вызов формы с установкой свойства Owner позволит нам в будущем обращаться к компонентам родительской формы - ведь нам нужно обращаться к набору данных, сетке, адаптеру таблицы и т.д.

Обратите внимание, что используется метод `Show()`, а не `ShowDialog()`, благодаря чему родительская форма не будет заблокирована и можно будет производить в ней какие-то операции. Метод `ShowDialog()` блокирует родительскую форму. Используя `Show()`, мы можем добавлять записи и наблюдать, как они появляются в сетке данных.

Теперь настало время написать обработчики для кнопок `CloseBtn` и `AddBtn`. Начнем с `CloseBtn` - ее обработчик предельно прост:

```
private void CloseBtn_Click(object sender, EventArgs e)
{
    Close();
}
```

Обработчик нажатия кнопки `AddBtn` будет немного сложнее. Чтобы код работал, как и ожидается, установите модификаторы доступа компонентов `mycontactsDataSet`, `contactsBindingSource`, `contactsTableAdapter` и `dataGridView1` в **Public**.

Весь код, который обращается к родительской форме, нужно поместить во внутрь следующего оператора **if**:

```
Form1 main = this.Owner as Form1;
if (main != null)
{
    // Обращаемся к род. форме
}
```

Преимущества следующего подхода в следующем:

- Он не противоречит канонам ООП, если какой-то зануда будет перечитывать ваш код, он не сможет придаться.
- Предоставляется доступ ко всем открытым (`public`) полям и методам первой формы.
- Передача данных возможна в обе стороны.

Если наша сетка данных привязана к источнику данных (в нашем случае - к БД), мы не можем просто добавить строчку методом `Rows.Add()`:

```
main.dataGridView1.Rows.Add(1, "Имя", "111-22-33", "email");
```

Мы получим исключение. Вместо этого нам нужно добавить данные непосредственно в источник данных, а затем обновить сетку данных. Сформируем добавляемую строку (запись):

```
DataRow nRow = main.mycontactsDataSet.Tables[0].NewRow();
int rc = main.dataGridView1.RowCount + 1;
```

```
nRow[0] = rc;
nRow[1] = tbName.Text;
nRow[2] = tbPhone.Text;
nRow[3] = tbMail.Text;
nRow[4] = tbPhoto.Text;
main.mycontactsDataSet.Tables[0].Rows.Add(nRow);
```

Сначала мы вызываем метод `NewRow()`. Здесь `mycontactsDataSet` - наш набор данных, таблица `contacts` у нас первая (нумерация с 0), поэтому ее идентификатор будет выглядеть как `Tables[0]`.

Нумерация полей записи тоже начинается с 0, поэтому элементы массива `nRow[]` нумеруются с 0. Мы вносим номер последней записи, увеличенный на 1 (`rc`), имя, телефон, электронную почту и название файла с фото - последние четыре значения получаются из полей ввода текста (`TextBox`).

После этого нам нужно вызвать метод `Rows.Add()` и передать ему сформированную нами запись. Осталось только обновить все и вся - адаптер таблицы, вызвать метод `AcceptChanges()` для самой таблицы и вызвать метод `Refresh()` для сетки данных:

```
main.contactsTableAdapter.Update(main.mycontactsDataSet.contacts);
main.mycontactsDataSet.Tables[0].AcceptChanges();
main.dataGridView1.Refresh();
```

Работающая программа изображена на рис. 14.23. Видно, что мы только что добавили строку в таблицу. В реальной программе вам придется очищать поля ввода перед вставкой следующей записи. Делается это так:

```
tbName.Text = "";
```

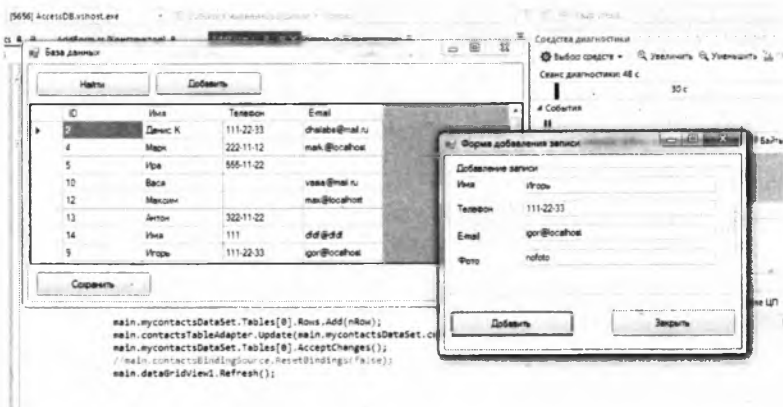


Рис. 14.23. Работа программы

Полный код обработчика кнопки **Добавить** приведен ниже:

```

private void AddBtn_Click(object sender, EventArgs e)
{
    Form1 main = this.Owner as Form1;
    if (main != null)
    {
        DataRow nRow = main.mycontactsDataSet.Tables[0].NewRow();
        int rc = main.dataGridView1.RowCount + 1;
        nRow[0] = rc;
        nRow[1] = tbName.Text;
        nRow[2] = tbPhone.Text;
        nRow[3] = tbMail.Text;
        nRow[4] = tbPhoto.Text;

        main.mycontactsDataSet.Tables[0].Rows.Add(nRow);
        main.contactsTableAdapter.Update(main.mycontactsDataSet.contacts);
        main.mycontactsDataSet.Tables[0].AcceptChanges();
        main.dataGridView1.Refresh();
        tbName.Text = «»;
        tbPhone.Text = «»;
        tbMail.Text = «»;
        tbPhoto.Text = «»;
    }
}

```

14.6. Поиск данных

Рассмотрим, как можно найти данные в таблице. Есть два способа. Первый - использовать SQL-операторы. Он подходит для очень больших таблиц, поскольку выполнение SQL-операторов обычно осуществляется на сервере и не будет нагружать компьютер клиента (пользователя). Второй - поиск по dataGridView. Ведь данные уже записаны в ячейки сетки данных и мы можем в цикле пройти по всем ячейкам и найти запись. Такой способ тоже имеет право на существование и подойдет для не очень больших таблиц. Мы рассмотрим второй способ для демонстрации использования сетки данных.

Создайте форму с именем SearchForm и расположите компоненты на ней, так, как показано на рис. 14.24. Название текстового поля - tbStr. Обработчик кнопки **Заккрыть** будет таким же, как и в предыдущем случае:

```
Close();
```

Прежде чем приступить к написанию обработчика кнопки **Найти**, нужно вернуться к обработчику кнопки **Найти** на главной форме и написать следующий код:

```

SearchForm sf = new SearchForm();
sf.Owner = this;
sf.Show();

```

Как и в предыдущем случае, данный код отображает дочернюю форму. Можно было бы сделать поле поиска, но форма поиска может предоставить больший функционал - вы можете добавить различные параметры поиска, которые бы позволяли производить поиск только по определенным столбцам или же производить поиск по целому слову. Мы используем метод `Contains()`, благодаря чему можно производить поиск по части слова. Например, если в базе есть имя **Антон** и вы введете **Ант**, то программа найдет строку, содержащую имя **Антон**.

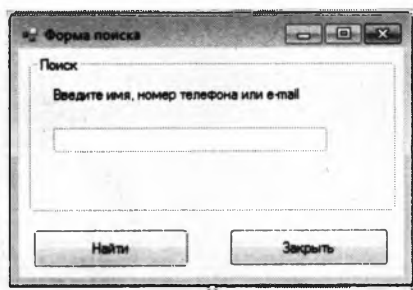


Рис. 14.24. Создание формы поиска

Обработчик кнопки поиска приведен ниже:

```
private void Find_Click(object sender, EventArgs e)
{
    Form1 main = this.Owner as Form1;
    if (main != null)
    {
        for (int i = 0; i < main.dataGridView1.RowCount; i++)
        {
            main.dataGridView1.Rows[i].Selected = false;
            for (int j = 0; j < main.dataGridView1.ColumnCount; j++)
            if (main.dataGridView1.Rows[i].Cells[j].Value != null)
            if (main.dataGridView1.Rows[i].Cells[j].Value.
ToString().Contains(tbStr.Text))
            {
                main.dataGridView1.Rows[i].Selected = true;
                break;
            }
        }
    }
}
```

Код, в принципе, очень простой. Мы проходимся по всем ячейкам, и если было найдено совпадение, мы выделяем соответствующую строку и прерываем цикл.

В завершение этой главы вам нужно модифицировать форму поиска и добавить переключатели и списки, позволяющие уточнить критерии поиска, а именно:

- Добавить возможность выбора поля (столбца), по которому осуществляется поиск.
- Добавить возможность поиска не первой встреченной записи, а продолжить поиск по принципу функции «Найти далее» в текстовом редакторе.

ЕВДОКИМОВ ПЕТР ВАЛЕНТИНОВИЧ

С#

на примерах

4-е издание

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

12+

ООО "Наука и Техника"

Лицензия №000350 от 23 декабря 1999 года.

192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107.

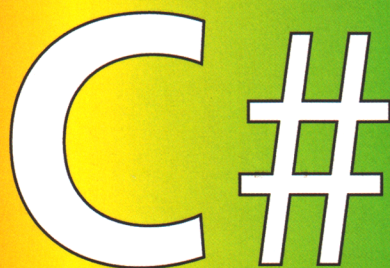
Подписано в печать 01.04.2019. Формат 70х100 1/16.

Бумага газетная. Печать офсетная. Объем 20 п. л.

Тираж 1400. Заказ 3944.

Отпечатано с готовых файлов заказчика
в АО "Первая Образцовая типография"
филиал "УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ"
432980, г. Ульяновск, ул. Гончарова, 14.

Евдокимов П. В.



на примерах

4-е издание

Эта книга является превосходным учебным пособием для изучения языка программирования C# на примерах. Изложение ведется последовательно: от развертывания .NET и написания первой программы, до многопоточного программирования, создания клиент-серверных приложений и разработки программ для мобильных устройств. По ходу даются все необходимые пояснения и комментарии. В четвертом издании был частично переработан текст по ходу изложения всей книги, а также обновлены некоторые примеры.

Книга написана простым и доступным языком. Лучший выбор для результативного изучения C#. Начните сразу писать программы на C#!

www.nit.com.ru



ISBN 978-5-94387-782-7



9 78- 5- 94387- 782- 7

Издательство "Наука и Техника"
г. Санкт-Петербург

Для заказа книг:
(812) 412-70-26
e-mail: nitmail@nit.com.ru
www.nit.com.ru