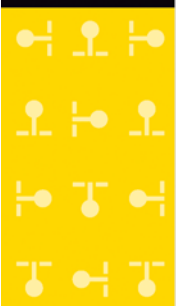




БИБЛИОТЕКА ПРОГРАММИСТА



Пол Дейтел Харви Дейтел

**КАК ПРОГРАММИРОВАТЬ НА**

# **Visual C# 2012**

Включая работу в **Windows 7** и **Windows 8**

**5-е издание**

PEARSON



 **ПИТЕР®**

**Paul Deitel**  
Deitel & Associates, Inc.

**Harvey Deitel**  
Deitel & Associates, Inc.

# Visual C# 2012 How to Program

5th Edition



**PEARSON**

Boston Columbus Indianapolis New York San Francisco Upper Saddle River Amsterdam  
Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi  
Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo





БИБЛИОТЕКА ПРОГРАММИСТА

Пол Дейтел, Харви Дейтел

**КАК ПРОГРАММИРОВАТЬ НА**

# **Visual C# 2012**

Включая работу в **Windows 7** и **Windows 8**

**5-е издание**



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

**2014**

ББК 32.973.2-018.1я7

УДК 004.43(075)

ДЗЗ

**Дейтел П., Дейтел Х.**

ДЗЗ Как программировать на Visual C# 2012. 5-е изд. — СПб.: Питер, 2014. — 864 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-00897-6

Эта книга, выходящая уже в пятом издании, является одним из самых популярных в мире учебников по программированию на платформе Microsoft .NET на языке Visual C# 2012. Здесь рассматриваются основы синтаксиса Visual C# и работа с программой Visual C# Express 2012. По ходу работы с книгой читатели изучат структуры управления, классы, объекты, методы, переменные, массивы C# и основные методы объектно-ориентированного программирования. Также рассматриваются и более сложные методы, в том числе поиск, сортировка, структуры данных, коллекции. Каждая глава содержит множество практических примеров. Пятое издание было полностью обновлено под новейшую версию Visual C# 2012.

Книга может служить учебником по Visual C#, также она будет полезна широкому кругу начинающих программистов, которые хотят научиться программировать на C#.

**12+** (Для детей старше 12 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1я7

УДК 004.43(075)

Права на издание получены по соглашению с Prentice Hall, Inc. Upper Sadle River, New Jersey 07458. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

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

ISBN 978-0133379334 англ.

© Authorized Translation from the English language edition, entitled VISUAL C# 2012 HOW TO PROGRAM, 5th Edition; ISBN 0133379337; by DEITEL, PAUL; and by DEITEL, HARVEY; published by Pearson Education, Inc., publishing as Prentice Hall. Copyright ©2014 by Pearson Education, Inc. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage retrieval system, without permission from Pearson Education, Inc. Electronic RUSSIAN language edition published by PITER PRESS Ltd. Copyright © 2015.

ISBN 978-5-496-00897-6

© Перевод на русский язык ООО Издательство «Питер», 2014

© Издание на русском языке, оформление ООО Издательство «Питер», 2014

# Краткое содержание

<b>Предисловие.....</b>	<b>20</b>
<b>Подготовка к работе.....</b>	<b>27</b>
<b>Глава 1. Компьютеры, Интернет и Visual C# .....</b>	<b>30</b>
<b>Глава 2. Visual Studio Express 2012 for Windows Desktop .....</b>	<b>58</b>
<b>Глава 3. Приложения C# .....</b>	<b>83</b>
<b>Глава 4. Классы, объекты и методы.....</b>	<b>114</b>
<b>Глава 5. Управляющие команды: часть 1 .....</b>	<b>145</b>
<b>Глава 6. Управляющие команды: часть 2 .....</b>	<b>181</b>
<b>Глава 7. Методы.....</b>	<b>216</b>
<b>Глава 8. Массивы и обработка исключений .....</b>	<b>257</b>
<b>Глава 9. LINQ и коллекция List .....</b>	<b>305</b>
<b>Глава 10. Подробнее о классах и объектах .....</b>	<b>322</b>
<b>Глава 11. Наследование.....</b>	<b>353</b>
<b>Глава 12. ООП: полиморфизм, интерфейсы и перегрузка операторов.....</b>	<b>386</b>
<b>Глава 13. Обработка исключений: следующий шаг .....</b>	<b>429</b>
<b>Глава 14. Графический интерфейс и Windows Forms: часть 1.....</b>	<b>459</b>
<b>Глава 15. Графический интерфейс и Windows Forms: часть 2.....</b>	<b>506</b>
<b>Глава 16. Строки и символы.....</b>	<b>573</b>

<b>Глава 17.</b> Файлы и потоки.....	<b>599</b>
<b>Глава 18.</b> Сортировка и поиск.....	<b>641</b>
<b>Глава 19.</b> Структуры данных .....	<b>668</b>
<b>Глава 20.</b> Обобщенные типы.....	<b>706</b>
<b>Глава 21.</b> Коллекции.....	<b>727</b>
<b>Глава 22.</b> Базы данных и LINQ .....	<b>759</b>
<b>Глава 23.</b> Разработка веб-приложений с использованием ASP.NET .....	<b>803</b>
<b>Приложение А.</b> Приоритет операторов .....	<b>853</b>
<b>Приложение Б.</b> Простые типы.....	<b>856</b>
<b>Приложение В.</b> Набор символов ASCII .....	<b>858</b>

# Содержание

<b>Предисловие .....</b>	<b>20</b>
Visual C#® 2012 и Visual Studio® 2012 IDE, .NET 4.5, Windows® 7 и Windows® 8 .....	20
Объектно-ориентированное программирование.....	21
Полные примеры кода.....	22
Другие особенности .....	22
Врезки.....	23
Программное обеспечение .....	24
Благодарности .....	24
Рецензенты.....	24
Об авторах.....	25
<b>Подготовка к работе.....</b>	<b>27</b>
Программное обеспечение .....	27
Программные и аппаратные требования.....	27
Просмотр расширений файлов .....	27
Загрузка примеров кода .....	28
Тема оформления Visual Studio.....	29
Отображение номеров строк и настройка табуляции .....	29
Замечания .....	29
<b>Глава 1. Компьютеры, Интернет и Visual C# .....</b>	<b>30</b>
1.1. Введение.....	30
1.2. Оборудование и закон Мура.....	30
1.3. Иерархия данных.....	31
1.4. Устройство компьютера.....	34
1.5. Машинный язык, ассемблер и языки высокого уровня .....	36
1.6. Объектные технологии .....	36



1.7. Интернет и Всемирная паутина.....	39
1.8. С# .....	41
1.8.1. Объектно-ориентированное программирование.....	41
1.8.2. Событийное программирование.....	41
1.8.3. Визуальное программирование .....	41
1.8.4. Международный стандарт и другие реализации С#.....	42
1.8.5. Интернет и веб-программирование.....	42
1.8.6. Знакомство с async/await.....	42
1.8.7. Другие современные языки программирования.....	42
1.9. Microsoft .NET.....	43
1.9.1. .NET Framework .....	44
1.9.2. CLR.....	44
1.9.3. Платформенная независимость.....	44
1.9.4. Языковая совместимость .....	45
1.10. Операционная система Microsoft Windows®.....	45
1.11. Windows Phone 8 для смартфонов.....	47
1.11.1. Продажа ваших приложений через Windows Phone Marketplace.....	47
1.11.2. Бесплатные и платные приложения .....	47
1.11.3. Тестирование приложений Windows Phone.....	48
1.12. Windows Azure™ и облачные вычисления.....	48
1.13. Visual Studio Express 2012 .....	49
1.14. Пробный запуск приложения Painter в Visual Studio Express 2012 for Windows Desktop .....	49
1.15. Пробный запуск приложения Painter в Visual Studio Express 2012 for Windows 8.....	53
<b>Глава 2. Visual Studio Express 2012 for Windows Desktop .....</b>	<b>58</b>
2.1. Введение.....	58
2.2. Обзор Visual Studio Express 2012 IDE.....	58
2.3. Строка меню и панель инструментов .....	64
2.4. Перемещение в Visual Studio IDE.....	66
2.4.1. Solution Explorer.....	67
2.4.2. Панель элементов.....	69
2.4.3. Окно свойств .....	70
2.5. Справочная система.....	71
2.6. Простое приложение с текстом и графикой.....	72
2.7. Итоги .....	82

<b>Глава 3. Приложения C# .....</b>	<b>83</b>
3.1. Введение.....	83
3.2. Простое приложение C#: вывод строки текста.....	83
3.3. Создание простого приложения в Visual Studio.....	89
3.4. Изменение простого приложения C#.....	95
3.5. Форматирование текста методами Console.Write и Console.WriteLine .....	97
3.6. Другое приложение C#: сложение целых чисел.....	99
3.7. Работа с памятью .....	103
3.8. Арифметика .....	104
3.9. Принятие решений: операторы сравнения и проверки равенства .....	108
3.10. Итоги.....	113
<b>Глава 4. Классы, объекты и методы.....</b>	<b>114</b>
4.1. Введение.....	114
4.2. Классы, объекты, методы, свойства и переменные экземпляров .....	114
4.3. Объявление класса с методом и создание экземпляра класса .....	116
4.4. Объявление метода с параметром.....	120
4.5. Переменные экземпляров и свойства .....	124
4.6. Диаграмма классов UML со свойством .....	129
4.7. Свойства и методы доступа.....	130
4.8. Автоматическая реализация свойств.....	131
4.9. Значимые типы и ссылочные типы .....	132
4.10. Инициализация объектов конструкторами .....	134
4.11. Числа с плавающей точкой и тип decimal.....	137
4.12. Итоги.....	144
<b>Глава 5. Управляющие команды: часть 1.....</b>	<b>145</b>
5.1. Введение.....	145
5.2. Алгоритмы.....	145
5.3. Псевдокод .....	146
5.4. Управляющие структуры .....	146
5.5. Команда одиночного выбора if.....	149
5.6. Команда двойного выбора if...else .....	150
5.7. Цикл while .....	155
5.8. Формулировка алгоритмов: цикл со счетчиком .....	157
5.9. Формулировка алгоритмов: цикл со сторожевым значением.....	162
5.10. Формулировка алгоритмов: вложенные управляющие команды .....	170

5.11. Комбинированные операторы присваивания.....	175
5.12. Операторы инкремента и декремента .....	176
5.13. Простые типы .....	179
5.14. Итоги.....	179

## **Глава 6. Управляющие команды: часть 2..... 181**

6.1. Введение.....	181
6.2. Основные принципы повторения со счетчиком .....	181
6.3. Цикл for .....	183
6.4. Примеры использования цикла for .....	187
6.5. Циклы do...while .....	191
6.6. Команда множественного выбора switch.....	193
6.7. Команды break и continue .....	201
6.8. Логические операторы.....	203
6.9. Основы структурного программирования.....	209
6.10. Итоги.....	214

## **Глава 7. Методы..... 216**

7.1. Введение.....	216
7.2. Упаковка кода в C# .....	216
7.3. Статические методы, статические переменные и класс Math.....	218
7.4. Объявление методов с несколькими параметрами.....	221
7.5. Объявление и использование методов.....	225
7.6. Стек вызовов и кадр стека .....	226
7.7. Преобразования типа аргументов.....	227
7.8. .NET Framework Class Library .....	228
7.9. Пример: генератор случайных чисел .....	230
7.9.1. Масштабирование и сдвиг случайных чисел .....	234
7.9.2. Повторение случайных чисел при тестировании и отладке.....	235
7.10. Пример: игра «крэпс» и перечисления .....	235
7.11. Область действия объявлений.....	240
7.12. Перегрузка методов .....	243
7.13. Необязательные параметры.....	246
7.14. Именованные параметры.....	248
7.15. Рекурсия .....	249
7.16. Передача аргументов по значению и по ссылке.....	252
7.17. Итоги.....	255

<b>Глава 8. Массивы и обработка исключений .....</b>	<b>257</b>
8.1. Введение .....	257
8.2. Массивы .....	257
8.3. Объявление и создание массивов .....	259
8.4. Примеры использования массивов .....	260
8.4.1. Создание и инициализация массива .....	260
8.4.2. Использование инициализатора массива .....	261
8.4.3. Вычисление значения, сохраняемого в каждом элементе массива .....	262
8.4.4. Суммирование элементов массива .....	264
8.4.5. Использование гистограмм для графического представления данных массива .....	265
8.4.6. Использование элементов массива как счетчиков .....	267
8.4.7. Основы обработки исключений .....	268
8.5. Пример: моделирование тасования колоды и сдачи карт .....	271
8.6. Команда foreach .....	276
8.7. Передача массивов и элементов массивов методам .....	277
8.8. Передача массивов по значению и по ссылке .....	279
8.9. Пример: класс GradeBook с массивом для хранения оценок .....	284
8.10. Многомерные массивы .....	289
8.11. Пример: класс GradeBook с использованием прямоугольного массива .....	295
8.12. Списки аргументов переменной длины .....	301
8.13. Аргументы командной строки .....	302
8.14. Итоги .....	304
<b>Глава 9. LINQ и коллекция List .....</b>	<b>305</b>
9.1. Введение .....	305
9.2. Выборка из массива с использованием LINQ .....	306
9.3. Запрос к массиву объектов Employee с использованием LINQ .....	311
9.4. Коллекции .....	316
9.5. Запросы к обобщенной коллекции с использованием LINQ .....	319
9.6. Итоги .....	321
9.7. Deitel LINQ Resource Center .....	321
<b>Глава 10. Подробнее о классах и объектах .....</b>	<b>322</b>
10.1. Введение .....	322
10.2. Класс Time .....	322
10.3. Управление доступом к членам .....	327

10.4. Ссылка this .....	328
10.5. Перегруженные конструкторы .....	330
10.6. Конструкторы по умолчанию и конструкторы без параметров .....	337
10.7. Композиция .....	337
10.8. Уборка мусора и деструкторы .....	341
10.9. Статические члены классов .....	342
10.10. Ключевое слово readonly .....	346
10.11. Абстракция данных и инкапсуляция .....	347
10.12. Окно Class View и Object Browser .....	349
10.13. Инициализаторы объектов .....	351
10.14. Итоги .....	351
<b>Глава 11. Наследование .....</b>	<b>353</b>
11.1. Введение .....	353
11.2. Базовые и производные классы .....	354
11.3. Защищенные члены классов .....	356
11.4. Отношения между базовыми и производными классами .....	357
11.4.1. Создание и использование класса CommissionEmployee .....	358
11.4.2. Создание класса BasePlusCommissionEmployee без применения наследования .....	364
11.4.3. Создание иерархии наследования .....	369
11.4.4. Иерархия наследования CommissionEmployee– BasePlusCommissionEmployee с использованием защищенных переменных экземпляров .....	372
11.4.5. Иерархия наследования CommissionEmployee– BasePlusCommissionEmployee с использованием закрытых переменных экземпляров .....	377
11.5. Конструкторы в производных классах .....	382
11.6. Применение наследования при проектировании программных продуктов .....	383
11.7. Класс object .....	384
11.8. Итоги .....	385
<b>Глава 12. ООП: полиморфизм, интерфейсы и перегрузка операторов .....</b>	<b>386</b>
12.1. Введение .....	386
12.2. Примеры полиморфизма .....	388
12.3. Демонстрация полиморфного поведения .....	389
12.4. Абстрактные классы и методы .....	392



12.5. Пример: система начисления заработка с использованием полиморфизма.....	396
12.5.1. Создание абстрактного базового класса Employee .....	397
12.5.2. Создание конкретного производного класса SalariedEmployee .....	399
12.5.3. Создание конкретного производного класса HourlyEmployee .....	401
12.5.4. Создание конкретного производного класса CommissionEmployee ....	403
12.5.5. Создание конкретного класса BasePlusCommissionEmployee .....	404
12.5.6. Полиморфная обработка, оператор is и понижающее преобразование .....	406
12.5.7. Сводка разрешенных присваиваний между переменными базового и производного класса.....	411
12.6. Запечатанные методы и классы.....	412
12.7. Пример: создание и использование интерфейсов.....	413
12.7.1. Разработка иерархии IPayable .....	415
12.7.2. Объявление интерфейса IPayable.....	416
12.7.3. Создание класса Invoice .....	416
12.7.4. Изменение класса Employee для реализации интерфейса IPayable ....	419
12.7.5. Изменение класса SalariedEmployee для использования с IPayable ...	420
12.7.6. Использование интерфейса IPayable для полиморфной обработки объектов Invoice и Employee.....	422
12.7.7. Часто используемые интерфейсы .NET Framework Class Library .....	423
12.8. Перегрузка операторов .....	424
12.9. Итоги.....	428
<b>Глава 13. Обработка исключений: следующий шаг .....</b>	<b>429</b>
13.1. Введение .....	429
13.2. Пример: деление на ноль без обработки исключений .....	430
13.3. Пример: обработка исключений DivideByZeroException и FormatException.....	433
13.3.1. Размещение кода в блоке try .....	435
13.3.2. Перехват исключений .....	436
13.3.3. Неперехваченные исключения .....	436
13.3.4. Модель обработки исключений.....	437
13.3.5. Передача управления при возникновении исключений.....	438
13.4. Иерархия исключений .NET.....	439
13.4.1. Класс SystemException .....	439
13.4.2. Определение исключений, инициируемых методом .....	440
13.5. Блок finally.....	441

13.6. Команда using.....	448
13.7. Свойства исключений.....	449
13.8. Пользовательские классы исключений .....	454
13.9. Итоги.....	457

## **Глава 14. Графический интерфейс и Windows Forms: часть 1..... 459**

14.1. Введение .....	459
14.2. Windows Forms .....	460
14.3. Обработка событий.....	463
14.3.1. Простой графический интерфейс.....	463
14.3.2. Автоматически генерируемый код графического интерфейса.....	465
14.3.3. Делегаты и механизм обработки событий .....	468
14.3.4. Другой способ создания обработчиков событий.....	469
14.3.5. Поиск информации о событиях.....	470
14.4. Свойства и макет элемента управления .....	472
14.5. Надписи, текстовые поля и кнопки.....	476
14.6. GroupBox и Panel.....	479
14.7. Флажки и переключатели .....	482
14.8. Графическое поле.....	491
14.9. Подсказки.....	494
14.10. Поле со счетчиком .....	495
14.11. Обработка событий мыши .....	498
14.12. Обработка событий клавиатуры .....	501
14.13. Итоги .....	505

## **Глава 15. Графический интерфейс и Windows Forms: часть 2..... 506**

15.1. Введение .....	506
15.2. Меню .....	507
15.3. Элемент управления MonthCalendar .....	516
15.4. Элемент управления DateTimePicker .....	518
15.5. Элемент управления LinkLabel .....	521
15.6. Элемент управления ListBox.....	524
15.7. Элемент управления CheckedListBox.....	530
15.8. Элемент управления ComboBox .....	533
15.9. Элемент управления TreeView .....	537
15.10. Элемент управления ListView .....	543
15.11. Элемент управления TabControl .....	550
15.12. Окна MDI.....	554

15.13. Визуальное наследование.....	562
15.14. Пользовательские элементы управления .....	567
15.15. Итоги .....	572
<b>Глава 16. Строки и символы .....</b>	<b>573</b>
16.1. Введение .....	573
16.2. Основы работы с символами и строками .....	573
16.3. Конструкторы string.....	575
16.4. Индексатор string, свойство Length и метод CopyTo.....	576
16.5. Сравнение строк.....	577
16.6. Поиск символов и подстроки.....	581
16.7. Извлечение подстрок .....	584
16.8. Конкатенация.....	584
16.9. Другие методы string.....	585
16.10. Класс StringBuilder .....	587
16.11. Свойства Length и Capacity, метод EnsureCapacity и индексатор класса StringBuilder .....	588
16.12. Методы Append и AppendFormat класса StringBuilder.....	590
16.13. Методы Insert, Remove и Replace класса StringBuilder .....	592
16.14. Методы Char .....	595
16.15. Итоги .....	597
<b>Глава 17. Файлы и потоки.....</b>	<b>599</b>
17.1. Введение .....	599
17.2. Иерархия данных.....	599
17.3. Файлы и потоки .....	602
17.4. Классы File и Directory.....	603
17.5. Создание текстового файла с последовательным доступом.....	612
17.6. Чтение данных из текстового файла с последовательным доступом.....	622
17.7. Запрос информации о счетах .....	626
17.8. Сериализация.....	632
17.9. Создание файла последовательного доступа с использованием сериализации .....	633
17.10. Чтение и десериализация данных из двоичного формата .....	638
17.11. Итоги .....	640
<b>Глава 18. Сортировка и поиск.....</b>	<b>641</b>
18.1. Введение .....	641
18.2. Алгоритмы поиска.....	642

18.2.1. Линейный поиск .....	642
18.2.2. Бинарный поиск.....	647
18.3. Алгоритмы сортировки .....	652
18.3.1. Сортировка выбором.....	652
18.3.2. Сортировка вставкой.....	657
18.3.3. Сортировка слиянием.....	661
18.4. Сводка эффективности алгоритмов поиска и сортировки .....	666
18.5. Итоги.....	667
<b>Глава 19. Структуры данных .....</b>	<b>668</b>
19.1. Введение .....	668
19.2. Структуры простых типов, упаковка и распаковка.....	668
19.3. Самоотносимые классы.....	669
19.4. Связанные списки .....	671
19.5. Стеки.....	683
19.6. Очереди.....	688
19.7. Деревья.....	691
19.7.1. Бинарное дерево поиска с целыми значениями .....	692
19.7.2. Бинарное дерево поиска для объектов IComparable .....	699
19.8. Итоги.....	705
<b>Глава 20. Обобщенные типы .....</b>	<b>706</b>
20.1. Введение .....	706
20.2. Причины использования обобщенных методов.....	707
20.3. Реализация обобщенного метода .....	709
20.4. Ограничения типов.....	712
20.5. Перегрузка обобщенных методов .....	715
20.6. Обобщенные классы.....	716
20.7. Итоги.....	726
<b>Глава 21. Коллекции .....</b>	<b>727</b>
21.1. Введение .....	727
21.2. Обзор коллекций.....	728
21.3. Класс Array и перечислители .....	730
21.4. Необобщенные коллекции.....	734
21.4.1. Класс ArrayList.....	735
21.4.2. Класс Stack.....	739
21.4.3. Класс Hashtable.....	742

21.5. Обобщенные коллекции .....	748
21.5.1. Обобщенный класс SortedDictionary .....	748
21.5.2. Обобщенный класс LinkedList.....	751
21.6. Ковариантность и контрвариантность для обобщенных типов.....	755
21.7. Итоги.....	758

## **Глава 22. Базы данных и LINQ ..... 759**

22.1. Введение .....	759
22.2. Реляционные базы данных.....	760
22.3. База данных Books .....	761
22.4. LINQ to Entities и ADO.NET Entity Framework .....	765
22.5. Запрос к базе данных с использованием LINQ.....	767
22.5.1. Создание библиотеки классов модели данных сущностей ADO.NET .....	768
22.5.2. Создание проекта Windows Forms и его настройка для использования модели данных сущностей .....	773
22.5.3. Привязка данных между элементами управления и моделью данных сущностей.....	775
22.6. Динамическая привязка результатов запроса .....	780
22.6.1. Создание графического интерфейса.....	781
22.6.2. Код приложения DisplayQueryResults.....	782
22.7. Выборка данных из нескольких таблиц с использованием LINQ.....	785
22.8. Создание приложения типа «главное/детализированное представление» .....	791
22.8.1. Создание графического интерфейса приложения MasterDetail.....	792
22.8.2. Код приложения MasterDetail .....	794
22.9. Адресная книга.....	795
22.9.1. Создание графического интерфейса.....	797
22.9.2. Код приложения адресной книги .....	798
22.10. Инструменты и сетевые ресурсы .....	802
22.11. Итоги .....	802

## **Глава 23. Разработка веб-приложений с использованием ASP.NET ..... 803**

23.1. Введение .....	803
23.2. Основы веб-программирования .....	804
23.3. Многоуровневая архитектура приложений .....	806
23.4. Первое веб-приложение .....	807



23.4.1. Построение приложения WebTime .....	809
23.4.2. Файл программной логики WebTime.aspx .....	819
23.5. Стандартные веб-элементы управления .....	820
23.6. Проверка данных .....	825
23.7. Отслеживание сеанса .....	833
23.7.1. Cookie.....	834
23.7.2. Отслеживание сеанса с использованием HttpSessionState .....	835
23.7.3. Options.aspx: выбор языка программирования .....	837
23.7.4. Recommendations.aspx: вывод рекомендаций на основании сеансовых значений .....	841
23.8. Пример: гостевая книга ASP.NET с использованием базы данных .....	843
23.8.1. Построение веб-формы для вывода информации из базы данных.....	845
23.8.2. Изменение файла программной логики приложения Guestbook.....	850
23.9. Итоги.....	852
<b>Приложение А. Приоритет операторов .....</b>	<b>853</b>
<b>Приложение Б. Простые типы .....</b>	<b>856</b>
Дополнительная информация о простых типах .....	857
<b>Приложение В. Набор символов ASCII .....</b>	<b>858</b>

*Нашей команде рецензентов: Шей Фридман, Октавио Эрнандесу,  
Стивену Хастеду, Хосе Антонио Гонсалес Секо, Шону Висфелду.*

*Мы благодарны вам за рекомендации и экспертизу.*

*Пол и Харви Дейтел*

# Предисловие

Добро пожаловать в мир программирования для Microsoft Windows и веб-программирования на платформе Microsoft .NET на языке Visual C# 2012!

Эта книга может использоваться для вводных учебных курсов, основанных на рекомендациях по образовательным программам двух ключевых профессиональных организаций — ACM и IEEE. Рассмотренные примеры доступны для студентов, изучающих компьютерные и информационные технологии, а также экономику на курсах начального и среднего уровня. Книга также может использоваться профессиональными программистами.

В основу книги заложен «фирменный» для Deitel метод *живого кода* — вместо фрагментов кода мы приводим концепции в контексте описания полноценных работоспособных программ с результатами тестовых запусков. В разделе «Подготовка к работе» приведены инструкции по настройке компьютера для выполнения многочисленных примеров кода. Исходный код примеров можно загрузить по адресам [www.deitel.com/books/vcsharp2012](http://www.deitel.com/books/vcsharp2012) и [www.pearsonhighered.com/deitel](http://www.pearsonhighered.com/deitel). Используйте этот исходный код для компиляции и запуска каждой программы в процессе ее изучения — это поможет быстрее и глубже изучить Visual C# и сопутствующие технологии Microsoft.

Мы полагаем, что читатель найдет в этой книге содержательное, нетривиальное и интересное введение в Visual C#. Если у вас возникнут вопросы, обращайтесь к нам по адресу [deitel@deitel.com](mailto:deitel@deitel.com), — мы постараемся быстро ответить на них. Чтобы получить актуальную информацию об обновлениях книги, посетите страницу [www.deitel.com/books/vcsharp2012](http://www.deitel.com/books/vcsharp2012), вступайте в наше сообщество в Facebook ([www.deitel.com/DeitelFan](http://www.deitel.com/DeitelFan)), Twitter ([@deitel](https://twitter.com/deitel)), Google+ ([gplus.to/deitel](https://plus.google.com/deitel)) и LinkedIn ([bit.ly/DeitelLinkedIn](http://bit.ly/DeitelLinkedIn)) и подписывайтесь на бюллетень Deitel® Buzz Online ([www.deitel.com/newsletter/subscribe.html](http://www.deitel.com/newsletter/subscribe.html)).

## Visual C#® 2012 и Visual Studio® 2012 IDE, .NET 4.5, Windows® 7 и Windows® 8

Выход Visual C# 2012 и сопутствующих технологий вдохновил нас на написание этой книги. Некоторые ключевые особенности нового издания:

- ❑ **Материал может использоваться как в Windows 7, так и в Windows 8.** Книга написана так, что вы можете продолжить работать в Windows 7 и постепенно переходить на Windows 8, а при желании можете сразу перейти на Windows 8. Все примеры кода были протестированы в Windows 7 и Windows 8.
- ❑ **C# и Visual C#.** Язык C# прошел международную стандартизацию в ECMA и ISO (стандарт можно бесплатно загрузить по адресу [bit.ly/ECMA334](http://bit.ly/ECMA334)). Visual C# 2012 представляет собой реализацию этого стандарта от компании Microsoft.
- ❑ **Модульный подход к построению многовариантного графического интерфейса для Windows Forms, Windows 8 и WPF.** В печатной версии книги представлен графический интерфейс Windows Forms.
- ❑ **Работа с базами данных в LINQ to Entities.** В предыдущем издании книги рассматривалась технология LINQ (Language Integrated Query) to SQL. Компания Microsoft в 2008 году прекратила дальнейшую разработку LINQ to SQL в пользу более новых и мощных технологий LINQ to Entities и ADO.NET Entity Framework, на которые мы переключились в этом издании, постаравшись сделать материал более доступным для новичков.
- ❑ **База данных SQL Server.** Для представления основных принципов программирования для баз данных используется бесплатная версия Microsoft SQL Server Express 2012 (которая устанавливается в составе бесплатного продукта Visual Studio Express 2012 for Windows Desktop). В главах 22–23 базы данных и средства LINQ используются для построения адресной книги для настольной системы, веб-приложения гостевой книги, электронного книжного магазина и системы бронирования авиабилетов.
- ❑ **ASP.NET 4.5** — серверная технология Microsoft на базе .NET — позволяет создавать мощные, масштабируемые приложения на базе веб-технологий. В главе 23 мы построим несколько приложений, включая веб-приложение гостевой книги, использующее ASP.NET и ADO .NET Entity Framework для хранения информации в базе данных и ее вывода на веб-страницах. Также в этой главе рассматривается применение веб-сервера IIS Express для тестирования веб-приложения на локальном компьютере.

## Объектно-ориентированное программирование

- ❑ **Раннее знакомство с объектами.** Основные концепции и терминология объектных технологий представлены в главе 1. В главе 2 рассматриваются визуальные операции с объектами (надписями и изображениями на формах), в главе 3 мы перейдем к написанию программного кода для работы с уже существующими объектами, а в главе 4 займемся написанием собственных классов. Раннее знакомство с объектами и классами способствует ускоренному переходу на «объектное мышление» и более глубокому усвоению этих концепций.

- ❑ **Подробное изложение фундаментальных принципов программирования.** В главах 5 и 6 представлено доступное описание управляющих конструкций и методов решения типичных задач.
- ❑ **Доступное описание классов, объектов, наследования, полиморфизма и интерфейсов с рассмотрением примеров.**
- ❑ **Три парадигмы программирования.** В книге рассматривается структурное программирование, объектно-ориентированное программирование и обобщенное программирование.

## Полные примеры кода

В книге приводятся разнообразные примеры, связанные с информатикой, экономикой, моделированием, компьютерными играми, графикой, мультимедиа и многими другими областями.

## Другие особенности

- ❑ **Использование LINQ для создания запросов к файлам, базам данных, XML и коллекциям.** Вводная глава LINQ to Objects (глава 9) намеренно была сделана простой и краткой, чтобы преподавателям было проще начать изложение основ LINQ на ранней стадии. Позднее в книге технология LINQ рассматривается более подробно на примерах LINQ to Entities (главы 22–23).
- ❑ **Локальное определение типов.** При инициализации локальной переменной в ее объявлении тип переменной можно не указывать — компилятор вычислит его по значению инициализатора.
- ❑ **Инициализаторы объектов.** Для новых объектов можно использовать синтаксис инициализаторов (сходный с синтаксисом инициализаторов массивов) для присваивания значений открытым свойствам и открытым переменным экземпляров новых объектов.
- ❑ Мы обращаем внимание читателя на **возможности использования функции IDE IntelliSense**, которая помогает быстрее писать код с меньшим количеством ошибок.
- ❑ **Файлы и строки.**
- ❑ **Серия глав, посвященных структурам данных**, с описанием алгоритмов сортировки и поиска, обобщенных типов данных и коллекций.



- ❑ **Интегрированная обработка исключений.** Мы знакомим читателя с механизмом обработки исключений довольно рано — в главе 8 он используется для контроля за выходом индекса за границу массива. В главе 10 показано, как использовать исключение при получении функцией недействительного аргумента. Полная информация об обработке исключений приводится в главе 13.

## Врезки

В текст книги включены врезки, выделяющие важные аспекты разработки. В них отражено самое полезное, что мы узнали за семь десятилетий совместного опыта практического программирования и преподавания.



### **ПРИЕМЫ ПРОГРАММИРОВАНИЯ**

Эти врезки привлекают внимание читателя к приемам, которые делают программный код более элегантным и понятным и упрощают его сопровождение.



### **ТИПИЧНАЯ ОШИБКА**

Описания часто встречающихся ошибок сокращают вероятность того, что вы допустите их в своих программах.



### **КАК ИЗБЕЖАТЬ ОШИБОК**

Рекомендации по выявлению и устранению ошибок в ваших программах; во многих врезках этого типа описываются аспекты Visual C#, препятствующие проникновению ошибок в программный код.



### **ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ**

В этих врезках подчеркиваются аспекты, которые помогают ускорить выполнение программ или сократить объем занимаемой памяти.



### **АРХИТЕКТУРНОЕ РЕШЕНИЕ**

Аспекты архитектуры и проектирования, влияющие на процесс построения программных систем (особенно крупномасштабных).



### **КРАСИВО И УДОБНО**

Эти врезки помогут создать привлекательный и удобный графический интерфейс, соответствующий современным стандартам.

## Программное обеспечение

При написании примеров кода используются бесплатные продукты компании Microsoft Visual Studio Express 2012:

- ❑ Visual Studio Express 2012 for Windows Desktop включает Visual C# и другие средства разработки Microsoft; работает в Windows 7 и 8.
- ❑ Visual Studio Express 2012 for Web (глава 23).

Все эти продукты доступны для загрузки по адресу

[www.microsoft.com/visualstudio/eng/products/visual-studio-express-products](http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products)

## Благодарности

Мы благодарны Эбби Дейтел (Abbey Deitel) и Барбаре Дейтел (Barbara Deitel) из Deitel & Associates, Inc. за многие часы, посвященные этому проекту. Эбби участвовала в написании предисловия и главы 1; они с Барбарой методично изучали новые возможности Visual C# 2012, .NET 4.5, Windows 8, Windows Phone 8, Windows Azure и других ключевых областей.

Нам повезло работать с группой профессионалов в области издательского дела из Pearson Higher Education. Мы благодарим Трейси Джонсон (Tracy Johnson), выпускающего редактора компьютерной редакции, за руководство, мудрость и энергичность. Кэрол Снайдер (Carole Snyder) провела выдающуюся работу по привлечению рецензентов книги и организации рецензирования. Боб Энглхарт (Bob Engelhardt) прекрасно справился с управлением процессом публикации.

## Рецензенты

Мы хотим поблагодарить наших рецензентов за проделанную работу. С материалами книги работали профессиональные преподаватели курсов C# и отраслевые эксперты. Они предоставили бесчисленные советы по улучшению материала. Все недостатки, оставшиеся в книге, лежат исключительно на нашей ответственности.

Рецензенты: Шей Фридман (Shay Friedman) — Microsoft Visual C# MVP, Октавио Эрнандес (Octavio Hernandez) — Microsoft Certified Solutions Developer, Стивен Хастедд (Stephen Hustedd) — колледж Саут Маунтин, Хосе Антонио Гонсалес Секо (José Antonio González Seco) — парламент Андалусии, Испания, и Шонн Вайсфилд (Shawn Weisfeld) — Microsoft MVP, президент и основатель UserGroup.tv.

Другие рецензенты последних изданий: Хуанхуэй Ху (Huanhui Hu) — Microsoft Corporation), Нагрес Касири (Narges Kasiri) — университет штата Оклахома, Чарльз Лю (Charles Liu) — Техасский университет в Сан-Антонио, доктор Хамид Р. Немати (Hamid R. Nemati) — университет Северной Каролины в Гринсборо, Джеффри П. Скотт (Jeffrey P. Scott) — технический колледж Блэкхок, Дуглас Б. Бок (Douglas B.

Bock) — MCS.D.NET, университет Южного Иллинойса в Эдвардсвилле, Дэн Кревьё (Dan Crevier) — Microsoft, Амит К. Гхош (Amit K. Ghosh) — Техасский университет в Эль Пасо, Марсело Гуэрра Хан (Marcelo Guerra Hahn) — Microsoft, Ким Хэмилтон (Kim Hamilton) — программист из компании Microsoft, соавтор книги «Learning UML 2.0», Джеймс Эдвард Кизор (James Edward Keyser) — Технологический институт штата Флорида, Хелена Котас (Helena Kotas) — Microsoft, Крис Ловетт (Chris Lovett) — специалист по архитектуре ПО из компании Microsoft, Башар Лулу (Bashar Lulu) — руководитель INETA по странам Персидского залива, Джон Макильхинни (John McIlhinney) — специалист по пространственному мышлению, Microsoft MVP 2008 Visual Developer, Visual Basic, Гед Мид (Ged Mead) — Microsoft Visual Basic MVP, DevCity.net, Ананд Мукундан (Anand Mukundan) — специалист по архитектуре, Polaris Software Lab Ltd., Тимоти Онг (Timothy Ng) — Microsoft, Акира Ониси (Akira Onishi) — Microsoft, Джо Стегнер (Joe Stagner) — старший руководитель программ, инструментарий разработчика и платформы, Эрик Томпсон (Erick Thompson) — Microsoft, Хесус Убальдо Кеведо-Торреро (Jesús Ubaldo Quevedo-Torrero) — университет штата Висконсин — Парксайд, кафедра информатики, Цзыцзян Ян (Zijiang Yang) — университет Западного Мичигана.

Мы будем искренне рады вашим замечаниям, критике и предложениям по улучшению книги. Пишите по адресу: [deitel@deitel.com](mailto:deitel@deitel.com)

Пол Дейтел  
Харви Дейтел

## От издательства

Скачать архив примеров, использованных в книге, можно по адресу [www.deitel.com/books/vcsharp2012htp](http://www.deitel.com/books/vcsharp2012htp). Вам необходимо зарегистрироваться на сайте, после чего станет доступна ссылка для скачивания примеров. Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты [comp@piter.com](mailto:comp@piter.com) (издательство «Питер», компьютерная редакция). Мы будем рады узнать ваше мнение!

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

## Об авторах

Пол Дейтел, генеральный и технический директор Deitel & Associates, Inc., — выпускник Массачусетского технологического института. Во время работы в Deitel & Associates, Inc. он провел сотни семинаров по программированию для различных организаций, среди которых были Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA (Космический центр Кеннеди), Национальная лаборатория исследования ураганов, ракетный полигон Уайт Сэндс, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys и многие другие. Он и его соавтор Харви М. Дейтел написали несколько популярных учебников по программированию и создали ряд видеокурсов.

В 2012 году Пол получил сертификат Microsoft® MVP (Most Valuable Professional) в области C# — по выражению Microsoft, «ежегодной награды для выдающихся руководителей технологических сообществ по всему миру, которые активно делятся своим ценным практическим опытом с пользователями и Microsoft».

Доктор Харви Дейтел, председатель и директор по стратегическим вопросам Deitel & Associates, Inc., обладает 50-летним опытом работы в компьютерной области. Доктор Дейтел получил степени бакалавра и магистра по электротехнике в Массачусетском технологическом институте, а также степень кандидата наук по математике в Бостонском университете. Он имеет огромный опыт преподавания, работал деканом факультета информатики в Бостонском колледже, прежде чем основать Deitel & Associates, Inc. в 1991 году вместе со своим сыном Полом Дейтелом. Труды Дейтелов пользуются международным признанием. Доктор Дейтел провел сотни учебных курсов для корпоративных, образовательных, правительственных и военных организаций.

## Программа обучения Deitel® Dive-Into® Series

Компания Deitel & Associates, Inc., основанная Полом Дейтелом (Paul Deitel) и Харви Дейтелом (Harvey Deitel), получила международное признание в области разработки авторских методик и корпоративного обучения. Компания специализируется на языках программирования, объектных технологиях, разработке мобильных приложений, интернет- и веб-технологий. Ее клиенты — крупнейшие компании, правительственные организации, силовые министерства и образовательные учреждения. Компания предлагает учебные курсы на территории клиента по основным языкам программирования и платформам: Visual C#®, Visual Basic®, Visual C++®, C++, C, Java™, XML®, Python®, объектные технологии, интернет- и веб-программирование, разработка приложений Android, разработка приложений на Objective-C для iPhone и многие другие дополнительные курсы по программированию и разработке программного обеспечения.

За 37 лет сотрудничества с издательством Prentice Hall/Pearson, компания Deitel & Associates, Inc., опубликовала множество учебников по программированию, профессиональной литературы и видеоуроков LiveLessons. С Deitel & Associates и авторами можно связаться по адресу: [deitel@deitel.com](mailto:deitel@deitel.com).

Чтобы больше узнать об программах корпоративного обучения Deitel Dive-Into® Series, обращайтесь по адресу: [www.deitel.com/training](http://www.deitel.com/training).

Чтобы получить предложение по проведению выездных учебных курсов под руководством инструкторов для вашей организации, напишите по адресу [deitel@deitel.com](mailto:deitel@deitel.com). Читатели желающие приобрести книги Deitel и видеоуроки LiveLessons, могут воспользоваться сайтом [www.deitel.com](http://www.deitel.com). Оптовые заказы от корпораций, правительственных учреждений, военных и образовательных учреждений следует размещать через компанию Pearson. За дополнительной информацией обращайтесь по адресу [www.informit.com/store/sales.aspx](http://www.informit.com/store/sales.aspx).

# Подготовка к работе

В этом разделе приведена информация, с которой следует ознакомиться до чтения книги, а также инструкции по подготовке программного обеспечения компьютера.

## Программное обеспечение

В книге используются следующие программные продукты:

- ☐ Microsoft Visual Studio Express 2012 for Windows Desktop.
- ☐ Microsoft Visual Studio Express 2012 for Web (глава 23).

Все эти продукты доступны для бесплатной загрузки по адресу [www.microsoft.com/express](http://www.microsoft.com/express). Версии Express Edition обладают полной функциональностью, а продолжительность их использования не ограничена.

## Программные и аппаратные требования

Для установки и запуска продуктов Visual Studio 2012 семейства Express Edition убедитесь в том, что ваша система соответствует минимальным требованиям, указанным по адресу [www.microsoft.com/visualstudio/eng/products/compatibility](http://www.microsoft.com/visualstudio/eng/products/compatibility).

Microsoft Visual Studio Express 2012 for Windows 8 работает только в Windows 8.

## Просмотр расширений файлов

На некоторых снимках экранов, приведенных в книге, отображаются имена файлов с расширениями (например, `.txt`, `.cs` или `.png`). Возможно, для отображения расширений потребуется внести изменения в настройки системы. Чтобы выполнить настройку на компьютере с Windows 7, выполните следующие действия:

1. В меню Пуск выберите команду Все программы, затем Стандартные и Проводник Windows.
2. Нажмите клавишу Alt, чтобы вызвать строку меню. Выберите в меню Сервис Проводника Windows команду Свойства папки.

3. В открывшемся диалоговом окне перейдите на вкладку Вид.
4. На панели Дополнительно снимите флажок слева от пункта Скрывать расширения для зарегистрированных типов файлов. [*Примечание:* если флажок уже снят, никакие действия не нужны].
5. Щелкните на кнопке ОК, чтобы применить изменения и закрыть диалоговое окно.

На компьютере с Windows 8 настройка выполняется следующим образом:

1. На стартовом экране щелкните на плитке Рабочий стол.
2. На панели задач щелкните на кнопке Проводник.
3. Перейдите на вкладку Вид и убедитесь в том, что флажок Расширения файлов установлен.

## Загрузка примеров кода

Примеры программ, используемых в книге, можно загрузить по адресу

[www.deitel.com/books/vcsharp2012http/](http://www.deitel.com/books/vcsharp2012http/)

Если вы еще не зарегистрированы на нашем сайте, перейдите по адресу [www.deitel.com](http://www.deitel.com) и щелкните на кнопке Register под логотипом в левом верхнем углу страницы. Введите информацию о себе. Регистрация на сайте бесплатна, и мы никому не передаем полученную информацию. На введенный адрес электронной почты будут отправляться только сообщения, связанные с управлением учетной записью, если только вы не подпишетесь отдельно на наш бесплатный бюллетень по адресу [www.deitel.com/newsletter/subscribe.html](http://www.deitel.com/newsletter/subscribe.html). При регистрации необходимо ввести действительный адрес электронной почты. После регистрации вы получите сообщение с кодом проверки. Щелкните на ссылке в сообщении, чтобы перейти на сайт [www.deitel.com](http://www.deitel.com) и подтвердить регистрацию.

Затем перейдите по адресу [www.deitel.com/books/vcsharp2012http/](http://www.deitel.com/books/vcsharp2012http/). Щелкните на ссылке Examples, чтобы загрузить ZIP-файл с архивом примеров на компьютер. Запишите папку, в которой был сохранен файл; обычно браузеры сохраняют файлы в папке Загрузки.

Когда в книге приводятся описания действий, связанных с обращением к коду на компьютере, предполагается, что примеры были извлечены из ZIP-файла и помещены в папку C:\Examples. Вы можете распаковать их в любую другую папку, но в этом случае необходимо внести соответствующие изменения в описание. Для распаковки архива в формате ZIP можно воспользоваться такими программами, как WinZip ([www.winzip.com](http://www.winzip.com)), 7-zip ([www.7-zip.org](http://www.7-zip.org)), встроенными средствами Проводника Windows в Window 7 или Проводника в Windows 8.

## Тема оформления Visual Studio

Visual Studio 2012 поддерживает темы **Dark** (используется по умолчанию) и **Light**. Снимки экранов, приведенные в книге, используют тему **Light**, которая лучше воспринимается на печати. Если вы захотите переключиться на тему **Light**, выберите в меню **TOOLS** команду **Options...**; на экране появляется диалоговое окно **Options**. Выберите в левом столбце категорию **Environment**, затем выберите тему **Light** в группе **Color theme**. Оставьте диалоговое окно **Options** открытым для следующего шага.

## Отображение номеров строк и настройка табуляции

Чтобы в среде разработки отображались номера строк, откройте узел **Text Editor** на левой панели и выберите категорию **All languages**. В правой части окна установите флажок **Line numbers**. Затем разверните узел **C#** на левой панели и выберите категорию **Tabs**. Убедитесь в том, что включен режим **Insert spaces**. Введите в полях **Tab size** и **Indent size** значение 3. После этого во всем новом коде каждый уровень отступов будет представляться тремя пробелами. Сохраните внесенные изменения кнопкой **ОК**.

## Замечания

Некоторые разработчики изменяют структуру своего рабочего пространства в среде разработки. Чтобы вернуться к структуре по умолчанию, выполните команду **Window ▸ Reset Window Layout**.

Рядом со многими командами меню, используемыми в книге, отображаются значки, которые также присутствуют на кнопках панели инструментов. Постепенно вы начнете узнавать эти значки и сможете использовать панели инструментов для ускорения процесса разработки. Также для многих команд меню определяются комбинации клавиш (также отображаемые рядом с командами) для ускоренного выполнения.

Все готово к тому, чтобы вы могли приступить к изучению **C#**. Надеемся, что эта книга вам понравится!

# 1

# Компьютеры, Интернет и Visual C#

## 1.1. Введение

Знакомьтесь: язык Visual C# 2012, который в дальнейшем мы будем называть просто C#<sup>1</sup>. Это мощный язык программирования, подходящий для создания серьезных информационных систем.

В этой книге мы займемся изучением объектно-ориентированного программирования — ключевой современной методологии программирования, которая повышает производительность труда программиста и сокращает затраты на разработку программного обеспечения. Мы создадим многочисленные программы, моделирующие как абстрактные, так и реальные объекты, а также построим приложения C# для разнообразных сред, включая настольные системы, мобильные устройства (смартфоны и планшеты) и даже «облачные» среды.

## 1.2. Оборудование и закон Мура

Компьютер состоит из различных устройств, объединяемых под общим термином «оборудование»: клавиатуры, экрана, мыши, жестких дисков, памяти, дисководов DVD и вычислительных блоков. Каждый год или два возможности компьютерного оборудования возрастали примерно вдвое без значительных затрат. Эта выдающаяся тенденция часто называется *законом Мура* по имени человека, который распознал ее, — это был Гордон Мур, соучредитель Intel, ведущего производителя процессоров для современных компьютеров и встроенных систем.

---

<sup>1</sup> Название C# (произносится «си-шарп») происходит от языка C и музыкального знака # (диез).



Закон Мура и другие наблюдения в первую очередь относятся к следующим характеристикам:

- ❑ Объем памяти, доступной компьютерам для выполнения программ и обработки данных.
- ❑ Объем вторичного пространства (например, пространства жесткого диска), используемого для долгосрочного хранения программ и данных.
- ❑ Скорость процессора — скорость, с которой компьютеры выполняют свои программы.

Аналогичный рост наблюдается в области коммуникаций. Огромный спрос на пропускную способность каналов (то есть способность передачи информации) породил жесткую конкуренцию, что привело к обвалу цен. Мы не знаем ни одной области, в которой технология развивалась бы так быстро, а цены падали так стремительно. Эти феноменальные достижения продвигают Информационную революцию и открывают значительные карьерные перспективы.

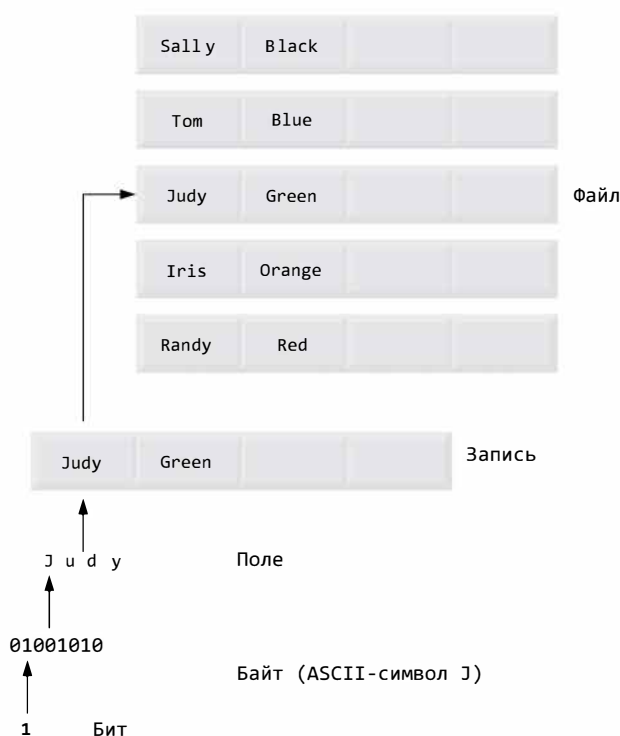
В результате непрерывного потока технологических достижений компьютеры уже способны выполнять вычисления и принимать логические решения несравнимо быстрее человека. Современные персональные компьютеры выполняют миллиарды вычислений в секунду — больше чем человек способен выполнить за всю жизнь. А производительность суперкомпьютеров уже достигает тысяч триллионов (квадриллионов) инструкций в секунду! Самый быстрый суперкомпьютер в мире — Cray Titan — выполняет свыше 17 квадриллионов вычислений в секунду (17,59 петафлопс)<sup>1</sup> — более двух миллионов вычислений в секунду для каждого человека на планете! И эти «верхние границы» постоянно расширяются!

## 1.3. Иерархия данных

Обрабатываемые компьютерами данные складываются в иерархии, которые увеличиваются и обретают все более сложную структуру по мере продвижения от простейших данных («биты») до расширенных элементов данных (символы, поля и т. д.) На ил. 1.1 изображена часть иерархии данных.

Наименьший элемент данных на компьютере — *бит* (сокращение от binary digit, то есть «двоичная цифра») может принимать всего два значения: 0 или 1. Как ни удивительно, все впечатляющие возможности компьютеров основаны на простых манипуляциях с нулями и единицами — чтением значений битов, записью и изменением (1 на 0 или наоборот).

<sup>1</sup> [top500.org/lists/2012/11/](http://top500.org/lists/2012/11/).



Ил. 1.1. Иерархия данных

## Символы

Человеку неудобно работать с данными в низкоуровневой форме, то есть с битами. Вместо этого мы предпочитаем иметь дело с десятичными цифрами (0–9), буквами (A–Z и a–z) и специальными знаками (\$, @, %, &, \*, (, ), –, +, ", :, ? и /). Цифры, буквы и специальные знаки называются *символами* (characters).

*Набор символов* складывается из всех символов, которые могут использоваться для написания программ и представления данных на устройстве. Компьютеры обрабатывают только значения 1 и 0, поэтому каждый символ с точки зрения компьютера представляется серией из 1 и 0. Набор символов *Юникод* содержит символы многих мировых языков. C# поддерживает несколько наборов символов, включая 16-разрядные символы Юникода, состоящие из двух байтов (один байт состоит из восьми битов).

## Поля

Итак, отдельные символы состоят из битов; поля состоят из символов (или байтов). Поле (field) представляет собой серию символов (или байтов), обладающую смыслом. Например, поле, состоящее из символов верхнего и нижнего регистра, может представлять имя человека, а поле, состоящее из десятичных цифр, — его возраст.

## Записи

Несколько взаимосвязанных полей могут составлять запись. Например, в системе расчета зарплаты запись работника может состоять из следующих полей (в круглых скобках указаны возможные типы этих полей):

- ☐ Табельный номер работника (целое число).
- ☐ Имя (последовательность символов).
- ☐ Адрес (последовательность символов).
- ☐ Почасовая оплата (дробное число).
- ☐ Заработок с начала года (дробное число).
- ☐ Налоговые отчисления (дробное число).

В этом примере все поля относятся к одному работнику. В компании может быть много работников, для каждого из которых создается отдельная запись.

## Файлы

Файл представляет собой группу взаимосвязанных записей. [*Примечание:* в более общем смысле файл содержит произвольные данные в произвольных форматах. В некоторых операционных системах файл рассматривается просто как последовательность байтов; любая организация байтов в файле (например, распределение данных по записям) интерпретируется как логическое представление, определяемое программистом.] В крупной организации могут использоваться тысячи и даже миллионы файлов, причем некоторые могут содержать миллиарды и даже триллионы символов. Работа с файлами рассматривается в главе 17.

## База данных

База данных представляет собой совокупность данных, упорядоченных для удобства доступа и обработки. В самой популярной модели баз данных — *реляционной* базе данных — данные хранятся в простых таблицах. Таблица состоит из записей и полей. Например, таблица с данными студентов может включать поля с именем, фамилией, отчеством, датой рождения, личным номером и средней оценкой. Данные каждого студента образуют запись, а отдельными блоками информации в каждой записи являются поля. Вы можете проводить поиск, сортировку и иные операции с данными с учетом их отношений с другими таблицами или базами данных. Например, университет может использовать информацию из базы данных студентов в сочетании с информацией из базы данных учебных курсов и т. д. Базы данных рассматриваются в главе 22.

## Большие данные

Объем данных, производимых в мировом масштабе, огромен и продолжает быстро расти. По данным IBM, ежедневно в мире создается 2,5 квинтиллиона байтов (2,5 экзабайта), и 90% мировых данных было создано за два последних года<sup>1</sup>! Результаты

<sup>1</sup> [www-01.ibm.com/software/data/bigdata/](http://www-01.ibm.com/software/data/bigdata/).

исследований IDC показывают, что в 2011 году использовалось приблизительно 1,8 зеттабайта (1,8 триллиона гигабайт) данных<sup>1</sup>. На ил. 1.2 представлены основные единицы измерения объема данных.

Единица	Байты	Приблизительно
1 килобайт	1024 байта	$10^3$ (ровно 1024 байта)
1 мегабайт	1024 килобайта	$10^6$ (1000000 байт)
1 гигабайт	1024 мегабайта	$10^9$ (1000000000 байт)
1 терабайт	1024 гигабайта	$10^{12}$ (1000000000000 байт)
1 петабайт	1024 терабайта	$10^{15}$ (1000000000000000 байт)
1 эксабайт	1024 петабайта	$10^{18}$ (1000000000000000000 байт)
1 зеттабайт	1024 эксабайта	$10^{21}$ (1000000000000000000000 байт)

**Ил. 1.2.** Единицы измерения объема данных

## 1.4. Устройство компьютера

При всем многообразии вариантов внешнего вида любой компьютер можно условно разделить на несколько логических блоков.

### Блок ввода

Блок ввода получает информацию (данные и компьютерные программы) от устройств ввода и передает их другим блокам для обработки. Большая часть информации вводится в компьютер с клавиатуры, сенсорного экрана и мыши. Также существует множество других форм ввода: голосовой ввод, сканирование изображений и штрих-кодов, чтение из внешних запоминающих устройств (жесткие диски, DVD-дисководы, дисководы Blu-ray Disc™ и USB-накопители), получение видео с веб-камеры или смартфона, загрузка информации из Интернета и т. д. Среди новейших форм ввода можно выделить получение позиционных данных с GPS-устройств или данных о перемещении и ориентации с акселерометров в смартфонах или игровых контроллеров.

### Блок вывода

Этот блок берет информацию, обработанную компьютером, и передает ее на различные устройства вывода, чтобы сделать ее доступной для использования вне компьютера. В наши дни большая часть выводимой информации отображается на экране монитора; печатается на бумаге; воспроизводится как аудио- или видеоролики на компьютерах, мультимедийных проигрывателях и гигантских экранах на стадионах; передается по Интернету или используется для управления внешними устройствами (роботами, 3D-принтерами и «умными» устройствами).

<sup>1</sup> [www.emc.com/collateral/about/news/idc-emc-digital-universe-2011-infographic.pdf](http://www.emc.com/collateral/about/news/idc-emc-digital-universe-2011-infographic.pdf).

### **Блок памяти**

В этом скоростном, относительно небольшом блоке хранится информация, полученная из блока ввода; сохраненная информация немедленно предоставляется для дальнейшей обработки. В блоке памяти также хранится обработанная информация до того момента, когда она может быть направлена на выходные устройства блоком вывода. Информация, хранимая в блоке памяти, энергозависима — обычно она теряется при выключении питания компьютера. Блок памяти часто называется просто памятью или основной памятью — на настольных компьютерах и ноутбуках его емкость обычно составляет до 16 гигабайт (Гбайт) памяти.

### **Арифметико-логический блок**

«Производственный» блок выполняет вычисления: сложение, вычитание, умножение и деление. Он также содержит механизм принятия решений, который позволяет компьютеру сравнить две ячейки памяти и проверить их на равенство содержимого. В современных системах арифметико-логический блок обычно реализуется как часть другого логического блока — центрального процессора.

### **Центральный процессор**

Этот блок осуществляет координацию и управление работой других блоков. Центральный процессор (далее для краткости «процессор») сообщает блоку ввода, когда информацию следует прочесть в блок памяти; арифметико-логическому блоку — когда информация из блока памяти должна использоваться в вычислениях; блоку вывода — когда информацию из блока памяти следует направить на различные устройства вывода. Многие современные компьютеры оснащены несколькими процессорами, а следовательно, могут выполнять несколько операций одновременно. Многоядерный процессор состоит из нескольких процессоров, размещенных на одной микросхеме, — так, двухъядерный процессор содержит два встроенных процессора, а четырехъядерный — четыре встроенных процессора. Четырехъядерные процессоры современных компьютеров способны выполнять миллиарды команд в секунду. В этой книге вы научитесь писать программы, которые задействуют все процессоры и обеспечивают их одновременную работу для ускорения вычислений.

### **Блок внешней памяти**

«Склад» большого объема, предназначенный для долгосрочного хранения информации. Программы или данные, которые не используются активно другими блоками, обычно помещаются на внешние запоминающие устройства (например, жесткий диск) до того момента, когда они снова потребуются компьютеру, — через несколько часов, дней, месяцев или даже лет. Информация на внешних запоминающих устройствах не пропадает даже при отключении питания компьютера. Обращения к данным на внешних запоминающих устройствах занимают существенно больше времени, чем обращения к основной памяти, но и стоимость единицы пространства внешнего запоминающего устройства существенно ниже, чем у основной памяти. В частности, к внешним запоминающим устройствам относятся дисководы CD-ROM, дисководы DVD и USB-накопители. Типичный жесткий диск настольного компьютера или ноутбука имеет емкость до 2 терабайт (Тбайт). В этом издании вы

увидите, что «облачное» хранилище может рассматриваться как дополнительное внешнее запоминающее устройство, доступное для ваших приложений C#.

## 1.5. Машинный язык, ассемблер и языки высокого уровня

Программисты пишут инструкции на языках программирования (таких, как C#). Одни языки «понятны» компьютеру без дополнительной обработки, другие требуют промежуточного преобразования — трансляции.

### Машинные языки

Любой компьютер непосредственно «понимает» только свой машинный язык, определяемый его аппаратной архитектурой. Машинные языки обычно состоят из чисел, которые в конечном итоге сводятся к цепочкам 0 и 1. Они неудобны для людей, предпочитающих обозначать выполняемые операции знаками и словами, поэтому числовые версии этих команд в машинном языке стали называться кодом. В наше время термин «код» используется в более широком смысле; сейчас им обозначаются программные инструкции в языках *любого* уровня.

### Ассемблеры

Работа с машинным языком была попросту слишком медленной и утомительной, и программисты стали обозначать элементарные операции сокращениями английских слов. Эти сокращения заложили основу *ассемблерных языков*. Программы-трансляторы, называемые *ассемблерами*, быстро переводили код на ассемблерном языке в машинный код. Хотя код на ассемблерном языке лучше читается человеком, он остается непонятным компьютеру до преобразования в машинный код.

### Языки высокого уровня, компиляторы и интерпретаторы

Для дальнейшего ускорения процесса программирования были разработаны языки высокого уровня, в которых одна команда могла выполнять нетривиальные операции. Инструкции на таких языках высокого уровня, как C#, Visual Basic, C++, C, Objective-C и Java, похожи на тексты, написанные на обычном английском языке, и в них используются общепринятые математические выражения. Программы-трансляторы, называемые *компиляторами*, преобразуют код на языке высокого уровня в машинный код.

Процесс компиляции программ на языке высокого уровня в машинный язык может занимать довольно много времени. Появились специальные программы-*интерпретаторы*, способные выполнять программы на языках высокого уровня напрямую, без компиляции (хотя и медленнее, чем откомпилированные программы).

## 1.6. Объектные технологии

C# является объектно-ориентированным языком программирования. В этом разделе представлены основы объектных технологий.

Быстрая, правильная и бюджетная разработка остается труднодостижимой целью, тогда как потребность в новых, более мощных программах растет. Объекты (а вернее классы, на основе которых создаются объекты) по сути представляют собой программные компоненты, пригодные для повторного использования. Объект может представлять дату, время, видеоролик, автомобиль, человека и т. д. Почти любое существительное может быть осмысленно представлено в виде программного объекта, обладающего атрибутами (имя, цвет, размер и т. д.) и поведением (вычисление, перемещение, обмен данными и т. д.). Разработчики обнаружили, что методология модульного объектно-ориентированного проектирования и реализации существенно повышает производительность труда групп разработчиков по сравнению с более ранними технологиями — объектно-ориентированные программы обычно считаются более понятными, в них проще вносить изменения и исправления.

Начнем с простой аналогии. Допустим, вы ведете машину и хотите ускорить ее, нажав педаль газа. Что должно произойти перед тем, как вы сможете это сделать? Для начала кто-то должен спроектировать вашу машину. Как правило, проектирование начинается с технических чертежей, в которых присутствует педаль газа. Педаль скрывает от водителя сложные механизмы, которые заставляют машину двигаться быстрее, — подобно тому, как педаль тормоза скрывает механизмы, замедляющие движение, а руль скрывает механизмы поворота. Все это позволяет легко управлять машиной человеку, не имеющему представления о работе двигателя, тормозов и поворотных механизмов.

Прежде чем вы сможете вести машину, необходимо построить ее по техническим чертежам. В построенной машине имеется педаль газа, которая заставляет ее двигаться быстрее, но даже этого недостаточно — машина не будет ускоряться сама по себе (хочется надеяться), поэтому водитель должен нажать на педаль.

### **Методы и классы**

Аналогия с машиной поможет представить некоторые ключевые концепции объектно-ориентированного программирования. Для выполнения некоторой операции в программу включается *метод*. В нем содержатся все команды программы, непосредственно выполняющие эту операцию. Эти команды скрыты от пользователя подобно тому, как педаль газа скрывает от водителя механизмы ускорения. В языках объектно-ориентированного программирования мы создаем программную единицу, называемую *классом*, для хранения набора методов, выполняющих операции класса. Например, класс, представляющий машину, может содержать методы для ускорения, торможения и поворота. Класс можно сравнить с техническим чертежом машины, на котором определяется устройство педали газа, руля и т. д.

### **Создание объектов на основе классов**

Чтобы водитель мог сесть за руль, машину нужно сначала построить по имеющимся чертежам. Так же дело обстоит и с классами: прежде чем программа сможет выполнить операции, определяемые методами класса, необходимо создать объект на основе класса. Созданный объект также называется *экземпляром* (instance) класса.

## Повторное использование

По одному техническому чертежу можно построить много машин. Класс тоже может использоваться многократно для создания многих объектов. Повторное использование существующих классов при построении новых классов и программ экономит время и силы. Оно способствует созданию более надежных и эффективных систем, потому что существующие классы и компоненты часто проходят тщательное тестирование (для выявления проблем), отладку (для исправления проблем) и оптимизацию. Концепция повторного использования классов стала важнейшим фактором революции программирования, вызванной внедрением объектных технологий.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 1.1

Используйте модульный принцип конструирования при создании своих программ. Не стоит заново «изобретать велосипед» — используйте готовые блоки там, где это возможно. Повторное использование программного обеспечения — важнейшее преимущество объектно-ориентированного программирования.

## Сообщения и вызовы методов

Когда вы ведете машину, нажатие педали газа передает машине сообщение о необходимости выполнить операцию — в данном случае повысить скорость. Аналогичным образом вы отправляете сообщения объектам. Каждое сообщение реализуется вызовом метода, который приказывает методу объекта выполнить свою операцию. Например, программа может вызвать метод `deposit` объекта банковского счета для начисления средств на счет.

## Атрибуты и переменные экземпляров

У машины, кроме способности к выполнению операций, также имеются атрибуты: цвет, количество дверей, текущая скорость, пробег (показания одометра) и т. д. Атрибуты машины, как и ее операции, включаются в ее технический чертеж (в котором, например, присутствует одометр и датчик топлива). Каждая машина поддерживает собственный набор атрибутов. Например, каждая машина знает, сколько бензина осталось в ее баке, но ничего не знает о наличии бензина в баках других машин.

Объект тоже обладает набором атрибутов, определенным в классе объекта. Например, у объекта банковского счета может быть атрибут, представляющий текущий баланс. Каждый объект банковского счета знает свой баланс, но не балансы других счетов в банке. Атрибуты задаются *переменными экземпляров* класса.

## Инкапсуляция

Классы *инкапсулируют* (то есть упаковывают) атрибуты и методы в объекты. Между атрибутами и операциями объекта существует тесная связь. Объекты могут взаимодействовать друг с другом, но обычно не имеют доступа к информации о том, как реализованы другие объекты — подробности реализации скрываются внутри объектов. Это сокрытие информации, как вы вскоре увидите, крайне важно для качественного проектирования программ.



## Наследование

Новые классы могут быстро и удобно создаваться посредством *наследования* (inheritance) — новый класс получает характеристики существующего класса, возможно, с их модификацией и добавлением собственных уникальных характеристик. В нашей аналогии с машинами объект класса «кабриолет» бесспорно может рассматриваться как объект более общего класса «автомобиль», но обладает специфической особенностью: его крыша может подниматься или опускаться.

## Объектно-ориентированный анализ и проектирование

Скоро вы начнете писать программы на С#. Возможно, как и многие программисты, вы будете просто включать свой компьютер и приступать к вводу кода. Такой подход работает для мелких программ (вроде тех, которые будут представлены в ранних главах книги), но что, если вам предстоит написать программную систему для управления тысячами банкоматов для крупного банка? Или вы руководите тысячами разработчиков, создающих систему управления полетами нового поколения? В таких больших и сложных проектах разработчик не может просто сесть за компьютер и взяться за написание программы.

Чтобы создать качественное решение, необходимо провести подробный анализ с определением требований проекта (то есть понять, *что* должна делать система) и разработать архитектуру, которая этим требованиям удовлетворяет (то есть решить, *как* система должна решать свои задачи). Этот процесс называется объектно-ориентированным анализом и проектированием, или ООАП. Язык С# относится к категории объектно-ориентированных языков. Программирование на таком языке, называемое *объектно-ориентированным программированием* (ООП), воплощает объектно-ориентированную архитектуру в рабочей системе.

## 1.7. Интернет и Всемирная паутина

В конце 1960-х годов Управление по перспективным исследованиям и разработкам Министерства обороны США (ARPA) выдвинуло проект по созданию сети, связывающей главные компьютерные системы десятка университетов и исследовательских институтов, находящихся на финансировании ARPA. Обмен данными между компьютерами должен был происходить на невероятной в те времена скорости 56 Кбит/с — в то время, когда большинство пользователей (из меньшинства, имевшего доступ к сети) передавало данные по телефонным линиям со скоростью 110 бит/с. Вскоре агентство ARPA перешло к реализации проекта, который получил известность под названием ARPAnet и стал предшественником современного Интернета.

Впрочем, результат несколько отличался от исходного плана. Хотя проект ARPAnet позволил исследователям объединить их компьютеры в сеть, его главным преимуществом оказалась возможность быстрого и простого общения между людьми по *электронной почте*. Это относится и к современному Интернету: электронная

почта, системы передачи мгновенных сообщений, пересылка файлов и социальные сети (такие, как Facebook и Twitter) позволяют миллиардам людей по всему миру быстро и удобно общаться друг с другом.

Протокол (набор правил) передачи данных в ARPAnet был назван *TCP* (Transmission Control Protocol). TCP гарантирует, что сообщения, состоящие из последовательно пронумерованных фрагментов, называемых *пакетами*, будут правильно переданы от отправителя к получателю, приняты без искажений и собраны в правильном порядке.

### Интернет: сеть сетей

Одновременно с ранним развитием Интернета организации по всему миру создавали собственные сети как для внутреннего, так и внешнего обмена данными. В процессе развития появилось много разных сетевых устройств и программ, которые нужно было как-то заставить работать друг с другом. В ARPA для этой цели был разработан протокол IP (Internet Protocol), который сформировал настоящую «сеть сетей» — текущую архитектуру Интернета. Объединенный набор протоколов известен под названием TCP/IP.

Коммерческие предприятия быстро поняли, что Интернет поможет повысить эффективность их работы и предоставить новые, более качественные услуги клиентам. Компании стали расходовать большие деньги на разработку и расширение своего присутствия в Интернете. Это породило ожесточенную конкуренцию между поставщиками услуг связи, оборудования и программ для удовлетворения растущей потребности в инфраструктуре. В результате суммарная пропускная способность каналов связи Интернета многократно увеличилась, а стоимость оборудования упала.

### Всемирная паутина

Всемирная паутина (World Wide Web), или просто «веб», представляет собой совокупность аппаратного и программного обеспечения в Интернете, которая позволяет пользователям находить и просматривать мультимедийные документы (документы с текстом, графикой, анимацией, аудио- и видеоданными) практически по любым темам. Развитие веб-технологий началось относительно недавно. В 1989 году Тим Бернерс-Ли из Европейской организации ядерных исследований (CERN) взялся за разработку технологии обмена информацией на основе «гипертекстовых» документов. Бернерс-Ли назвал свое изобретение «языком гипертекстовой разметки», или HTML (HyperText Markup Language). Он также написал коммуникационные протоколы (например, HTTP — HyperText Transfer Protocol), заложившие основу новой гипертекстовой системы, которую он назвал «Всемирной паутиной».

В 1994 году Бернерс-Ли основал комитет World Wide Web Consortium (W3C, [www.w3.org](http://www.w3.org)), посвященный развитию веб-технологий. Одной из основных целей W3C стало предоставление доступа к Всемирной паутине всем пользователям независимо от физических возможностей, языка и культуры. В этой книге мы будем использовать C# и другие технологии Microsoft для построения веб-приложений.

## 1.8. C#

В 2000 году компания Microsoft представила новый язык программирования C#. Своими корнями C# уходит к языкам программирования C, C++ и Java. C# обладает примерно такими же возможностями, как и Java; он подходит для самых ответственных задач разработки, включая построение современных крупномасштабных корпоративных приложений, мобильных и «облачных» приложений.

### 1.8.1. Объектно-ориентированное программирование

C# относится к категории объектно-ориентированных языков. Программы, написанные на C#, могут использовать .NET Framework Class Library — гигантскую библиотеку готовых классов, ускоряющую разработку приложений (ил. 1.3). О .NET более подробно рассказано в разделе 1.9.

Ключевые возможности .NET Framework Class Library	
Базы данных	Отладка
Построение веб-приложений	Многопоточное программирование
Графика	Операции с файлами
Ввод-вывод	Безопасность
Сетевые взаимодействия	Веб-взаимодействия
Управление разрешениями	Графический интерфейс
Мобильные приложения	Структуры данных
Обработка строк	

**Ил. 1.3.** Ключевые возможности библиотеки .NET Framework Class Library

### 1.8.2. Событийное программирование

Программирование на C# управляется событиями. Разработчик пишет программу, которая реагирует на события, инициированные пользователем: щелчки кнопкой мыши, нажатия клавиш, срабатывания таймера и (новая возможность Visual C# 2012) прикосновения к сенсорному экрану и жесты, широко используемые на смартфонах и планшетах.

### 1.8.3. Визуальное программирование

Microsoft Visual C# является *визуальным* языком программирования — помимо написания команд для построения частей приложения, разработчик может использовать графический интерфейс (GUI, Graphic User Interface) Visual Studio

для удобного перетаскивания заранее определенных объектов (таких, как кнопки и текстовые поля) в нужное место экрана, назначения их текста и изменения размеров. Visual Studio сгенерирует большую часть GUI-кода за вас.

### 1.8.4. Международный стандарт и другие реализации C#

Язык C# прошел международную стандартизацию. Это позволило создать другие реализации языка, кроме Microsoft's Visual C#, — такие как Mono ([www.mono-project.com](http://www.mono-project.com)) для систем Linux, iOS (для Apple's iPhone, iPad и iPod touch), Google Android и Windows. Документ со стандартом C# доступен по адресу

[www.ecma-international.org/publications/standards/Ecma-334.htm](http://www.ecma-international.org/publications/standards/Ecma-334.htm)

### 1.8.5. Интернет и веб-программирование

Во многих современных приложениях учитываются взаимодействия между компьютерами по всему миру. Как вы узнаете, это одна из основных тем стратегии Microsoft .NET. В главе 23 мы построим веб-приложения на базе C# и технологии Microsoft ASP.NET.

### 1.8.6. Знакомство с `async/await`

Как правило, в современных программах выполнение каждой задачи должно быть завершено до того, как начнется выполнение следующей задачи. Этот стиль, называемый *синхронным программированием*, используется в большинстве программ книги. В C# также поддерживается модель *асинхронного программирования*, в которой несколько задач могут выполняться одновременно. В частности, асинхронное программирование ускоряет отклик приложений на действия пользователя: щелчки и нажатия клавиш.

Асинхронное программирование в предыдущих версиях Visual C# было сложным и ненадежным. Новые конструкции `async` и `await` в Visual C# 2012 упрощают асинхронное программирование, так как компилятор скрывает большую часть сложности от разработчика.

### 1.8.7. Другие современные языки программирования

На ил. 1.4 приведена сводка популярных языков программирования, которые по своей функциональности близки к C#.

Язык программирования	Описание
C	Язык C был реализован в 1972 году Деннисом Ричи из Bell Laboratories. На первых порах он был известен как язык разработки операционной системы UNIX. В наши дни большая часть кода операционных систем общего назначения пишется на C или C++
C++	Язык C++, расширение C, был разработан Бьярном Страуструпом в начале 1980-х годов в Bell Laboratories. C++ предоставляет ряд функций, «украшающих» язык C, но что еще важнее — в нем реализована поддержка объектно-ориентированного программирования. C++ часто используется в приложениях с жесткими требованиями к производительности: операционных системах, системах реального времени, встроенных и коммуникационных системах. Visual C++ — версия языка C++ от Microsoft
Java	В 1990-е годы компания Sun Microsystems (теперь часть Oracle) разработала объектно-ориентированный язык на базе C++, получивший название Java. Главной целью Java была возможность написания программ, способных работать на самых разных компьютерных системах и управляющих устройствах. Java используется для разработки крупномасштабных корпоративных приложений, расширения функциональности веб-серверов, программирования приложений для потребительских устройств (смартфоны, планшеты, бытовые приборы и т. д.) и многих других целей. Компания Microsoft разработала язык C# как конкурента Java
Visual Basic	В основу Visual Basic был заложен язык BASIC, разработанный в 1960-х годах для знакомства новичков с фундаментальными принципами программирования. В конце 1980-х и начале 1990-х годов компания Microsoft разработала графический интерфейс Microsoft Windows — визуальную часть операционной системы, с которой взаимодействует пользователь. Это обстоятельство привело к естественной эволюции BASIC в язык Visual Basic, упрощавший программирование Windows-приложений. Последние версии Visual Basic по своим возможностям близки к C#
Objective-C	Objective-C — еще один объектно-ориентированный язык на базе C. Он был разработан в Stepstone в начале 1980-х годов и позднее приобретен компанией NeXT, которая, в свою очередь, была приобретена Apple. Objective-C стал ключевым языком для настольных операционных систем Mac OS X и устройств на базе iOS (iPod, iPhone, iPad)

Ил. 1.4. Популярные языки программирования

## 1.9. Microsoft .NET

В 2000 году компания Microsoft представила свою инициативу .NET ([www.microsoft.com/net](http://www.microsoft.com/net)) — концепцию использования Интернета и веб-технологий в разработке, производстве, распространении и использовании программных продуктов. Вместо того чтобы привязывать пользователя к одному конкретному языку программирования, .NET позволяет создавать приложения на любом .NET-совместимом языке (таким, как C#, Visual Basic, Visual C++ и многие другие). Одной из составляющих инициативы .NET является технология Microsoft ASP.NET.

### 1.9.1. .NET Framework

Среда .NET Framework выполняет приложения и библиотеку .NET Framework Class Library, широкие возможности которой позволяют быстро и легко строить серьезные приложения на C#. .NET Framework Class Library содержит тысячи полезных готовых классов, протестированных и оптимизированных для максимальной производительности. Вы научитесь создавать собственные классы, но там, где это возможно, старайтесь использовать классы .NET Framework; они ускоряют процесс разработки с одновременным повышением качества и производительности разрабатываемого продукта.

### 1.9.2. CLR

Среда CLR (Common Language Runtime), еще один ключевой компонент .NET Framework, исполняет программы .NET и предоставляет многие полезные средства для упрощения разработки и отладки. CLR представляет собой *виртуальную машину* (VM, Virtual Machine) — программу, которая управляет выполнением других программ и изолирует их от базовой операционной системы и оборудования. Исходный код программ, выполняемых под управлением CLR, называется *управляемым кодом*. CLR предоставляет управляемому коду различные виды сервиса — интеграцию программных компонентов, написанных на разных языках .NET, обработку ошибок между такими компонентами, повышенную безопасность, автоматическое управление памятью и т. д. Сервис CLR недоступен для программ с неуправляемым кодом, что усложняет их написание<sup>1</sup>. Управляемый код компилируется в машинные инструкции за несколько этапов:

1. Сначала код компилируется на язык MSIL (Microsoft Intermediate Language). Среда CLR может связывать воедино код, преобразованный в MSIL с других языков и источников, — это позволяет программистам работать на своих любимых языках .NET. Код MSIL компонентов приложения помещается в исполняемый файл приложения.
2. При выполнении приложения другой компилятор (называемый Just-In-Time-, или JIT-компилятором) в CLR транслирует MSIL из исполняемого файла в машинный код (для конкретной платформы).
3. Код на машинном языке выполняется на текущей платформе.

### 1.9.3. Платформенная независимость

Если версия .NET Framework существует и установлена на платформе, эта платформа может выполнить любую программу .NET. Способность программы выполняться без изменений на разных платформах называется *платформенной независимостью*.

<sup>1</sup> msdn.microsoft.com/en-us/library/8bs2ecf4.aspx.

Единожды написанный код может использоваться на другом типе компьютеров без изменений, а это экономит время и деньги. Вдобавок платформенная независимость расширяет круг пользователей. Прежде компаниям приходилось решать, оправданы ли затраты по адаптации их программ для других платформ (этот процесс называется *портированием*). С появлением .NET проблемы портирования отпали (по крайней мере после того, как версии .NET стали доступными на разных платформах).

#### 1.9.4. Языковая совместимость

.NET Framework предоставляет высокий уровень языковой совместимости. Поскольку программные компоненты, написанные на разных языках .NET (таких, как C# и Visual Basic), компилируются в код MSIL, компоненты могут объединяться для создания общей программы. Таким образом, MSIL обеспечивает независимость .NET Framework от языка.

Библиотека .NET Framework Class Library может использоваться любым языком .NET. Версия .NET 4.5, выпущенная в 2012 году, включает ряд усовершенствований и новых возможностей, ускоряющих работу и повышающих скорость отклика приложений. В нее также входит .NET for Windows Store Apps — подмножество .NET, используемое для создания приложений с интерфейсом в стиле Windows 8.

### 1.10. Операционная система Microsoft Windows®

Microsoft Windows — самая распространенная настольная операционная система в мире. Операционной системой называется программная система, упрощающая работу с компьютером для пользователей, разработчиков и системных администраторов. Операционные системы предоставляют средства, которые позволяют каждому приложению выполняться безопасно, эффективно и параллельно с другими приложениями. Среди популярных операционных систем для настольных компьютеров также можно упомянуть Linux и Mac OS X. На смартфонах и планшетах популярны такие мобильные операционные системы, как Microsoft Windows Phone, Google Android, Apple iOS (для устройств iPhone, iPad и iPod Touch) и BlackBerry OS. На ил. 1.5 приведена эволюция операционной системы Windows.

Версия	Описание
Windows 1990-х годов	В середине 1980-х компания Microsoft разработала операционную систему Windows с графическим интерфейсом, использующим кнопки, текстовые поля, меню и другие графические элементы. Различные версии, выпускавшиеся в 1990-е годы, были предназначены для персональных компьютеров. Компания Microsoft пришла на рынок корпоративных операционных систем с выпуском Windows NT в 1993 году

**Ил. 1.5.** Эволюция операционной системы Windows (продолжение ↗)

Версия	Описание
Windows XP и Windows Vista	В системе Windows XP, выпущенной в 2001 году, были объединены две линейки операционных систем Microsoft: корпоративная и потребительская. XP до сих пор сохраняет популярность; по данным исследования 2012 года Netmarketshare, она используется более чем на 40% компьютеров с системой Windows ( <a href="http://netmarketshare.com/operating-system-market-share.aspx?qprid=10&amp;qpcustomd=0">netmarketshare.com/operating-system-market-share.aspx?qprid=10&amp;qpcustomd=0</a> ). Версия Windows Vista, выпущенная в 2007 году, предлагала привлекательный новый пользовательский интерфейс Aero, много полезных усовершенствований, новые приложения и улучшенные средства безопасности. Однако версия Vista так и не прижилась — сегодня она занимает «всего» 6 % рынка настольных операционных систем (что на самом деле не так уж мало; <a href="http://netmarketshare.com/operating-system-market-share.aspx?qprid=10&amp;qpcustomd=0">netmarketshare.com/operating-system-market-share.aspx?qprid=10&amp;qpcustomd=0</a> )
Windows 7	Windows 7 сейчас является самой распространенной версией Windows. Она отличается наличием улучшенного пользовательского интерфейса Aero, уменьшенным временем запуска, дальнейшим совершенствованием средств безопасности Vista, поддержкой сенсорных экранов и т. д. Windows 7 принадлежит 44% рынка. В мировом масштабе системы Windows (включая Windows 7, Windows XP и Windows Vista) занимают более 90% рынка настольных операционных систем ( <a href="http://netmarketshare.com/operating-system-market-share.aspx?qprid=10&amp;qpcustomd=0">netmarketshare.com/operating-system-market-share.aspx?qprid=10&amp;qpcustomd=0</a> ). Материал книги в основном ориентирован на Windows 7, Visual Studio 12 и Visual C# 2012
Windows 8 для настольных систем и планшетов	Система Windows 8, выпущенная в 2012 году, предоставляет сходную платформу (систему, в которой работают приложения) и интерфейс для широкого спектра устройств, включая персональные компьютеры, смартфоны, планшеты и сетевой игровой сервис Xbox Live. В новом начальном экране каждое приложение представлено плиткой (tile), по аналогии с интерфейсом Windows Phone — операционной системой Microsoft для смартфонов. Windows 8 обеспечивает поддержку множественных касаний для сенсорных панелей и устройств с сенсорными экранами, улучшенные средства безопасности и ряд других усовершенствований
Пользовательский интерфейс Windows 8	Visual C# 2012 поддерживает новый пользовательский интерфейс Windows 8 (ранее называвшийся «Metro»). Приложения Windows 8 используют окно без дополнительного оформления — окна уже не имеют рамки с такими интерфейсными элементами, как строки заголовка и меню. Эти элементы скрыты, благодаря чему приложение может занимать весь экран; это особенно полезно на малых экранах планшетов и смартфонов. Элементы интерфейса отображаются в строке приложения, когда пользователь протаскивает указатель мыши у верхнего или нижнего края экрана; на устройствах с сенсорным экраном достаточно провести пальцем по экрану

Ил. 1.5. Эволюция операционной системы Windows (окончание)



## 1.11. Windows Phone 8 для смартфонов

Windows Phone 8 — усеченная версия Windows 8 для смартфонов. Эти устройства весьма ограничены по ресурсам; они уступают настольным компьютерам по объему памяти и вычислительной мощности процессора и обладают меньшим временем работы батареи. Windows Phone 8 использует те же базовые функции операционной системы, что и Windows 8, включая общую файловую систему, безопасность, поддержку сети и веб-браузер Internet Explorer 10. Однако система Windows Phone 8 содержит *только* функциональность, необходимую для смартфонов, что позволяет ей работать эффективно и с минимальными затратами ресурсов.

В этом издании книги вы научитесь использовать Visual C# 2012 для разработки ваших собственных приложений Windows Phone 8. Подобно тому как популярность языка Objective-C выросла из-за разработки приложений iOS, среда Visual C# 2012 наверняка станет более популярной с ростом спроса на устройства Windows Phone. IDC (International Data Corporation) предсказывает, что к 2016 году устройствам на базе Windows Phone будет принадлежать свыше 19% рынка смартфонов; эти устройства опережат Apple iPhone и будут уступать только устройствам на базе Android<sup>1</sup>.

### 1.11.1. Продажа ваших приложений через Windows Phone Marketplace

Вы можете продавать свои приложения Windows Phone через Windows Phone Marketplace ([www.windowsphone.com/marketplace](http://www.windowsphone.com/marketplace)) — аналог таких коммерческих платформ, как App Store (Apple), Google Play (ранее Android Market), Facebook App Center и Windows Store. Также можно заработать, предоставив свое приложение для бесплатной загрузки и продавая виртуальные товары (дополнительный контент, игровые уровни, виртуальные подарки и расширения) через механизм покупок из приложения.

### 1.11.2. Бесплатные и платные приложения

Недавнее исследование Gartner показало, что 89% всех мобильных приложений распространяется бесплатно, а к 2016 году их доля вырастет до 93%; к этому моменту покупки внутри приложения составят свыше 40% доходов от мобильных приложений<sup>2</sup>. Цены на платные приложения Windows Phone 8 лежат в диапазоне от \$1,49 (выше стартовой цены \$0,99 приложений Google Play и App Store) до \$999,99. Средняя цена мобильного приложения составляет от \$1,50 до \$3 в зависимости от платформы. Для приложений Windows Phone компания Microsoft удерживает

<sup>1</sup> [www.idc.com/getdoc.jsp?containerId=prUS23523812](http://www.idc.com/getdoc.jsp?containerId=prUS23523812).

<sup>2</sup> [techcrunch.com/2012/09/11/free-apps/](http://techcrunch.com/2012/09/11/free-apps/).

30% цены продажи, а 70% достается разработчику. На момент написания книги в Windows Phone Marketplace было представлено свыше 100 000 приложений<sup>1</sup>.

### 1.11.3. Тестирование приложений Windows Phone

Для тестирования приложений Windows Phone можно воспользоваться эмулятором Windows Phone Emulator, включенным в Windows Phone 8 SDK (пакет разработчика). Чтобы протестировать приложение на телефоне или передать его в Windows Phone Marketplace для продажи/бесплатного распространения, необходимо зарегистрироваться в Windows Phone Dev Center. Для участников установлена ежегодная оплата \$99; программа бесплатна для студентов и подписчиков MSDN. На сайте доступны средства разработки, примеры кода, советы по продаже приложений, рекомендации по дизайну и много другой полезной информации. Чтобы присоединиться к программе Windows Phone Dev Center, посетите страницу *dev.windowsphone.com/en-us/downloadsdk*.

## 1.12. Windows Azure™ и облачные вычисления

Технологии облачных вычислений позволяют использовать программные продукты и данные, хранящиеся в «облаке» (то есть на удаленных компьютерах или серверах в Интернете и доступных по требованию), а не на вашем настольном компьютере, ноутбуке или мобильном устройстве. Облачные вычисления повышают гибкость, позволяя наращивать и сокращать компьютерные ресурсы в соответствии с текущими потребностями. Такие решения обладают большей экономической эффективностью, чем приобретение дорогостоящего оборудования с достаточным объемом памяти и вычислительных мощностей при периодических пиковых нагрузках.

Облачные вычисления также экономят деньги за счет того, что нагрузка по управлению этими приложениями переходит на поставщика сервиса. Microsoft Windows Azure — облачная платформа для разработки, управления и распространения приложений в облачных средах. С Windows Azure ваши приложения хранят свои данные в облаке, чтобы они были доступны в любой момент с любого из ваших настольных компьютеров или мобильных устройств. Участники программы DreamSpark могут загрузить Visual Studio 2012 Professional со встроенной поддержкой Windows 8 и Windows Azure<sup>2</sup>. Вы можете подписаться на бесплатный 90-дневный период пробного использования Windows Azure по адресу *www.windowsazure.com/en-us/pricing/free-trial/*.

<sup>1</sup> [windowsteamblog.com/windows\\_phone/b/windowsphone/archive/2012/06/20/announcing-windows-phone-8.aspx](http://windowsteamblog.com/windows_phone/b/windowsphone/archive/2012/06/20/announcing-windows-phone-8.aspx).

<sup>2</sup> [www.dreamspark.com/Product/Product.aspx?productid=44](http://www.dreamspark.com/Product/Product.aspx?productid=44).

## 1.13. Visual Studio Express 2012

Программы C# создаются в Microsoft Visual Studio — наборе программных инструментов, объединенных в интегрированную среду разработки (IDE, Integrated Development Environment). Версия Visual Studio 2012 Express позволяет быстро и удобно писать, запускать, тестировать и отлаживать программы C#. Она поддерживает языки программирования Microsoft Visual Basic, Visual C++ и F#. Большинство примеров книги было построено в версии Visual Studio Express 2012 for Windows Desktop, работающей в Windows 7 и Windows 8.

## 1.14. Пробный запуск приложения Painter в Visual Studio Express 2012 for Windows Desktop

[Примечание: для этого примера вам понадобится компьютер, работающий под управлением Windows 7 или Windows 8. Описание в тексте предназначено для Windows 7. Запуск приложений в Windows 8 рассматривается в разделе 1.15.]

Сейчас мы воспользуемся Visual Studio Express 2012 for Windows Desktop для пробного запуска готового приложения, позволяющего рисовать на экране мышью. Приложение Painter (которое мы построим в одной из последующих глав) позволяет, среди прочего, выбирать размеры и цвета кисти. Ниже приведено пошаговое описание тестового запуска приложения.

### Шаг 1. Проверка конфигурации

Проверьте правильность настройки компьютера и программного обеспечения (раздел «Подготовка к работе»).

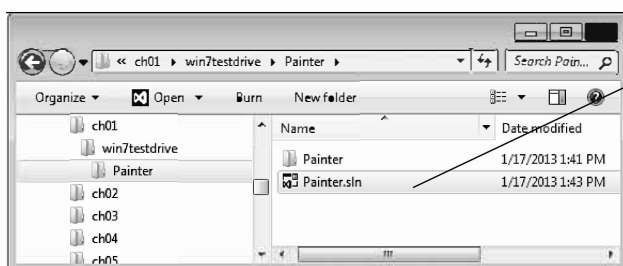
### Шаг 2. Поиск каталога приложения Painter

Откройте окно Проводника Windows и перейдите к каталогу C:\examples\ch01\win7testdrive. (Предполагается, что примеры были установлены в каталог C:\examples.) Сделайте двойной щелчок на папке Painter, чтобы просмотреть ее содержимое (ил. 1.6); сделайте двойной щелчок на файле решения Painter.sln, чтобы открыть его в Visual Studio. Файл решения содержит все файлы кода приложения, вспомогательные файлы (графика, видео, файлы данных и т. д.) и конфигурационную информацию. Содержимое решений более подробно рассматривается в следующей главе.

В зависимости от настройки системы Проводник Windows может не отображать расширения файлов (например, .sln на ил. 1.6). Режим отображения расширений включается следующим образом:

1. В Проводнике Windows нажмите Alt+t, чтобы открыть меню Сервис, и выберите команду Свойства папки.

2. Выберите вкладку Вид в диалоговом окне.
3. Сбросьте флажок Скрывать расширения для зарегистрированных типов файлов.
4. Щелкните на кнопке ОК, чтобы закрыть диалоговое окно Свойства папки.

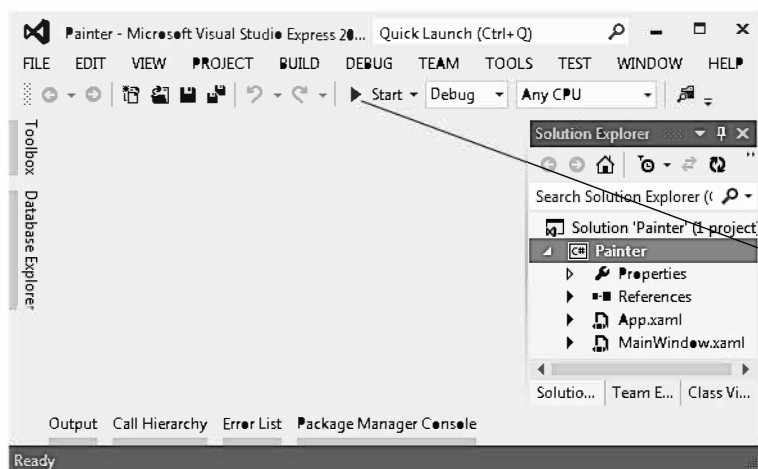


Сделайте двойной щелчок на файле решения Painter.sln, чтобы открыть проект в Visual Studio

**Ил. 1.6.** Содержимое каталога C:\examples\ch01\win7testdrive\Painter

### Шаг 3. Запуск приложения Painter

Чтобы увидеть приложение Painter в действии, щелкните на кнопке Start (▶) (ил. 1.7) или нажмите клавишу F5. На ил. 1.8 изображено окно выполняемого приложения.

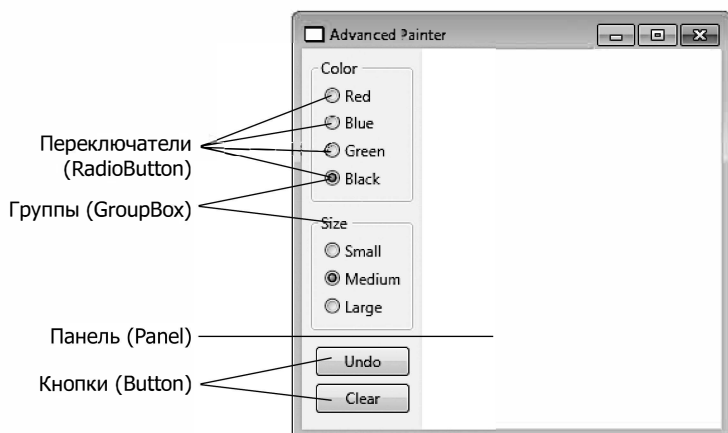


Нажмите кнопку Start, чтобы запустить приложение

**Ил. 1.7.** Запуск приложения Painter

На ил. 1.8 изображены некоторые графические элементы приложения, называемые *элементами управления* (controls). Эти элементы более подробно рассматриваются далее. Приложение позволяет рисовать черной, красной, синей или зеленой кистью трех разных размеров: малого, среднего и большого. При перетаскивании мыши по белой панели приложение рисует круги заданного размера и цвета в текущей позиции указателя. Чем медленнее перемещается мышь, тем ближе расположены круги. Таким образом, при медленном перетаскивании образуется сплошная линия (как на ил. 1.9), а при быстром — отдельные круги, разделенные промежутками. Вы также можете отменить предыдущую операцию или стереть рисунок, нажав кнопку Undo или Clear под переключателями. Готовые элементы управления (которые

представляются в программе объектами) позволяют создавать мощные приложения гораздо быстрее, чем если бы вам пришлось писать весь код самостоятельно.

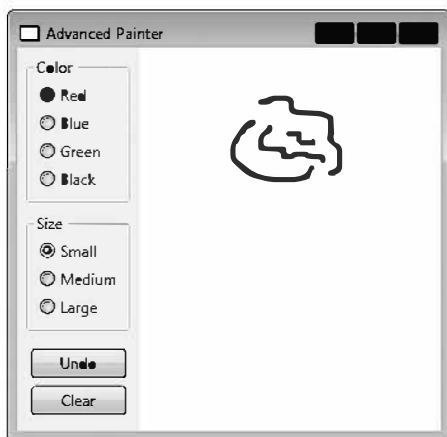


**Ил. 1.8.** Приложение Painter в Windows 7

Свойства кисти (переключатели **Black** и **Medium**) выбираются по умолчанию при запуске приложения. Программисты реализуют конфигурацию по умолчанию, чтобы в работе приложения использовались разумные значения, даже если они не были заданы пользователем. Кроме того, значения по умолчанию предоставляют визуальные подсказки, которые помогают пользователю выбрать нужные значения. Сейчас вы измените их как пользователь приложения.

#### Шаг 4. Изменение цвета кисти

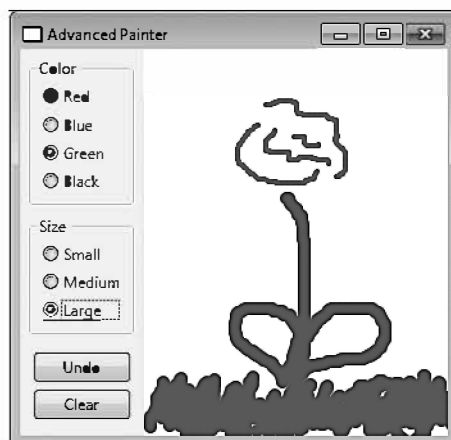
Щелкните на переключателе с надписью **Red**, чтобы изменить цвет кисти; затем щелкните на переключателе с надписью **Small**, чтобы изменить размер кисти. Наведите указатель мыши на белую панель. Перетаскивая указатель, нарисуйте лепестки цветка, как показано на ил. 1.9.



**Ил. 1.9.** Лепестки, нарисованные малой красной кистью

### Шаг 5. Изменение цвета и размера кисти

Щелкните на переключателе **Green**, чтобы изменить цвет кисти. Щелкните на переключателе **Large**, чтобы изменить размер кисти. Нарисуйте траву и стебель цветка (ил. 1.10).



**Ил. 1.10.** Стебель цветка и трава, нарисованные большой зеленой кистью

### Шаг 6. Последние штрихи

Установите переключатели **Blue** и **Medium**. Нарисуйте капли дождя (ил. 1.11), чтобы завершить рисунок.



**Ил. 1.11.** Капли дождя, нарисованные средней синей кистью

### Шаг 7. Завершение работы приложения

Приложение, запущенное из Visual Studio, можно завершить кнопкой **Stop** (■) на панели инструментов Visual Studio или же кнопкой закрытия (X) в окне выполняемого приложения.

## 1.15. Пробный запуск приложения Painter в Visual Studio Express 2012 for Windows 8

[Примечание: упражнение может быть выполнено *только* на компьютере с Windows 8.]

Сейчас мы воспользуемся Visual Studio для пробного запуска готового приложения Windows 8, позволяющего рисовать на экране мышью. Приложение Painter (которое мы построим в одной из последующих глав) позволяет, среди прочего, выбирать размеры и цвета кисти. Ниже приведено пошаговое описание тестового запуска приложения.

### Шаг 1. Проверка конфигурации

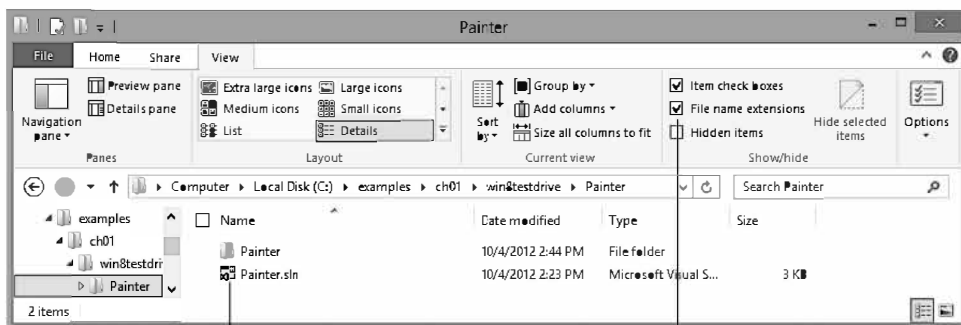
Проверьте правильность настройки компьютера и программного обеспечения (раздел «Подготовка к работе»).

### Шаг 2. Переключение на рабочий стол Windows 8

Щелкните на плитке Рабочий стол начального экрана Windows 8, чтобы перейти к рабочему столу системы.

### Шаг 3. Поиск каталога приложения Painter

Щелкните на кнопке File Explorer (📁) на панели задач, чтобы открыть окно Проводника Windows, и перейдите к каталогу C:\examples\ch01\win8testdrive. (Предполагается, что примеры были установлены в каталог C:\examples.) Сделайте двойной щелчок на папке Painter, чтобы просмотреть ее содержимое (ил. 1.12); сделайте двойной щелчок на файле решения Painter.sln, чтобы открыть его в Visual Studio. Файл решения содержит все файлы кода приложения, вспомогательные файлы (графика, видео, файлы данных и т. д.) и конфигурационную информацию. Содержимое решений более подробно рассматривается в следующей главе. [Примечание: в зависимости от настройки системы Проводник Windows может не отображать расширения файлов (например, .sln на ил. 1.12). Чтобы включить режим отображения расширений, перейдите на вкладку Вид в окне Проводника Windows и убедитесь в том, что флажок File name extensions установлен.]



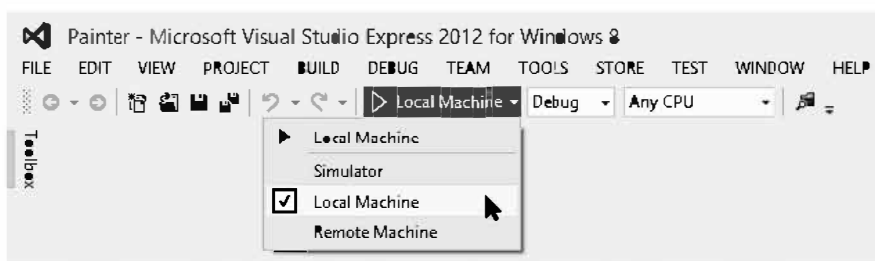
Сделайте двойной щелчок на файле Painter.sln, чтобы открыть проект в Visual Studio

Установите флажок, чтобы видеть расширения файлов

Ил. 1.12. Содержимое каталога C:\examples\ch01\win7testdrive\Painter

## Шаг 4. Запуск приложения Painter

Приложения с пользовательским интерфейсом Windows 8 обычно занимают весь экран, хотя пользователь также может размещать приложения в области шириной 320 пикселей, находящейся в левой или правой части экрана, чтобы видеть два приложения одновременно. Чтобы увидеть работающее приложение Painter, вы можете либо установить его на начальном экране Windows 8 и выполнить его, выбрав поле **Local Machine** (ил. 1.13) с последующим нажатием кнопки **Start Debugging** (▶), либо нажать клавишу F5. После того как приложение будет установлено на начальном экране, его также можно запустить щелчком на его плитке. На ил. 1.14 изображено работающее приложение. В последующих главах будут рассмотрены варианты **Simulator** и **Remote Machine**, изображенные на ил. 1.13.

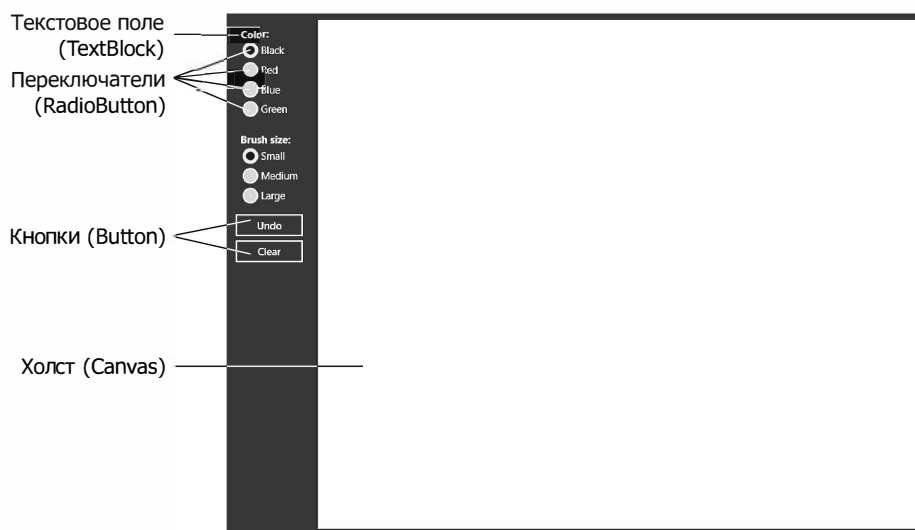


**Ил. 1.13.** Запуск приложения Painter в режиме Local Machine

На ил. 1.14 изображены некоторые графические элементы приложения, называемые *элементами управления* (controls). Эти элементы более подробно рассматриваются далее. Приложение позволяет рисовать черной, красной, синей или зеленой кистью трех разных размеров: малого, среднего и большого. При перетаскивании мыши по белой панели **Canvas** (объект, используемый для рисования) приложение рисует круги заданного размера и цвета в текущей позиции указателя. Чем медленнее перемещается мышь, тем ближе расположены круги. Таким образом, при медленном перетаскивании образуется сплошная линия (как на ил. 1.15), а при быстром — отдельные круги, разделенные промежутками (это можно заметить на некоторых каплях на ил. 1.17). Вы также можете отменить предыдущую операцию или стереть рисунок, нажав кнопку **Undo** или **Clear** под переключателями. Готовые элементы управления (которые представляются в программе объектами) позволяют создавать мощные приложения гораздо быстрее, чем если бы вам пришлось писать весь код самостоятельно.

Свойства кисти (переключатели **Black** и **Small**) выбираются по умолчанию при запуске приложения. Программисты реализуют конфигурацию по умолчанию, чтобы в работе приложения использовались разумные значения, даже если они не были заданы пользователем. Кроме того, значения по умолчанию предоставляют визуальные подсказки, которые помогают пользователю выбрать нужные значения. Сейчас вы измените их как пользователь приложения.

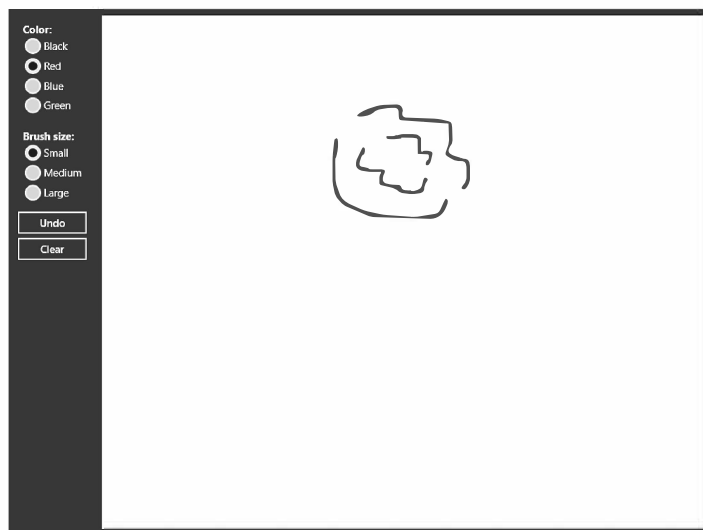




**Ил. 1.14.** Приложение Painter в Windows 8

### Шаг 5. Изменение цвета кисти

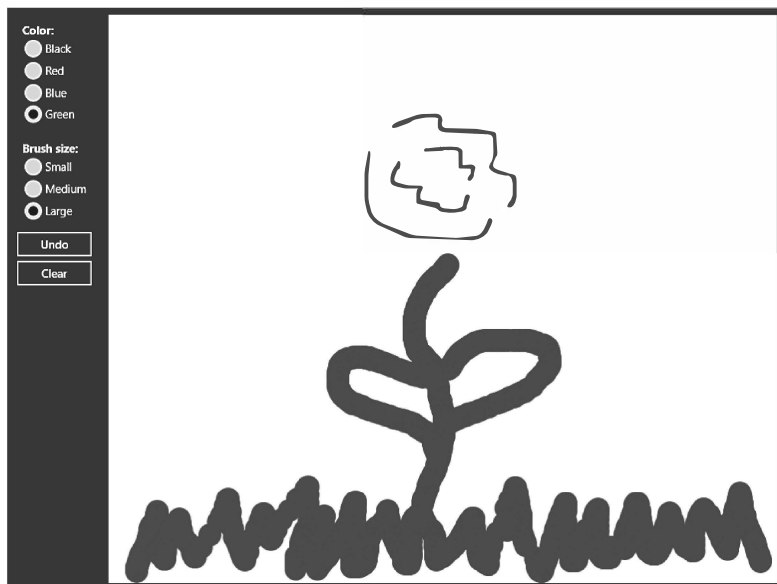
Щелкните на переключателе с надписью Red, чтобы изменить цвет кисти. Наведите указатель мыши на белую панель Canvas. Перетаскивая указатель, нарисуйте лепестки цветка, как показано на ил. 1.15.



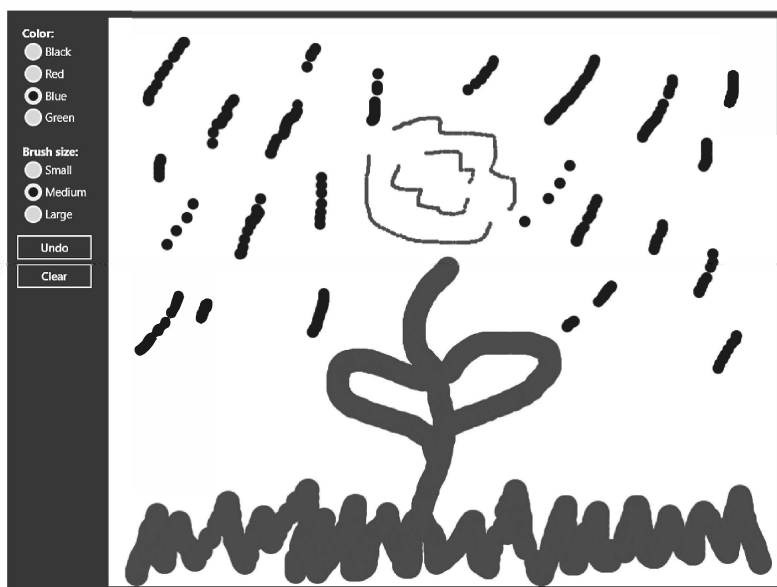
**Ил. 1.15.** Лепестки, нарисованные малой красной кистью

### Шаг 6. Изменение цвета и размера кисти

Щелкните на переключателе **Green**, чтобы изменить цвет кисти. Щелкните на переключателе **Large**, чтобы изменить размер кисти. Нарисуйте траву и стебель цветка (ил. 1.16).



**Ил. 1.16.** Стебель цветка и трава, нарисованные большой зеленой кистью



**Ил. 1.17.** Капли дождя, нарисованные средней синей кистью

### Шаг 7. Последние штрихи

Установите переключатели **Blue** и **Medium**. Нарисуйте капли дождя (ил. 1.17), чтобы завершить рисунок.

### Шаг 8. Завершение работы приложения

Приложение, запущенное из Visual Studio, можно завершить кнопкой **Stop** (■) на панели инструментов Visual Studio. Как правило, после завершения работы в приложениях с пользовательским интерфейсом Windows 8 вы не завершаете приложение, а просто запускаете другое. Windows 8 приостанавливает выполнение предыдущего приложения, но оставляет его в памяти на случай, если вы решите вернуться к нему. Windows также может решить завершить приостановленное приложение, чтобы освободить память для других приложений. Чтобы принудительно завершить приложение Windows 8, просто проведите пальцем от верхнего края экрана вниз или нажмите **Alt+F4**.

Итак, после пробного запуска можно переходить к разработке приложений C#. В главе 2 мы используем Visual Studio для создания вашей первой программы на C# средствами визуального программирования. Как вы увидите, Visual Studio автоматически генерирует код построения пользовательского интерфейса приложения. В главе 3 мы займемся написанием программ C# с традиционным кодом, который не генерируется автоматически, а набирается на клавиатуре.

# 2 Visual Studio Express 2012 for Windows Desktop

## 2.1. Введение

Visual Studio 2012 — созданная компанией Microsoft интегрированная среда разработки (IDE) для создания, запуска и отладки приложений на различных языках программирования .NET. В этой главе приведен обзор Visual Studio 2012 IDE и описан процесс создания простого приложения Visual C#, основанный на размещении готовых структурных элементов в нужных местах, — эта методология называется *визуальной разработкой приложений*.

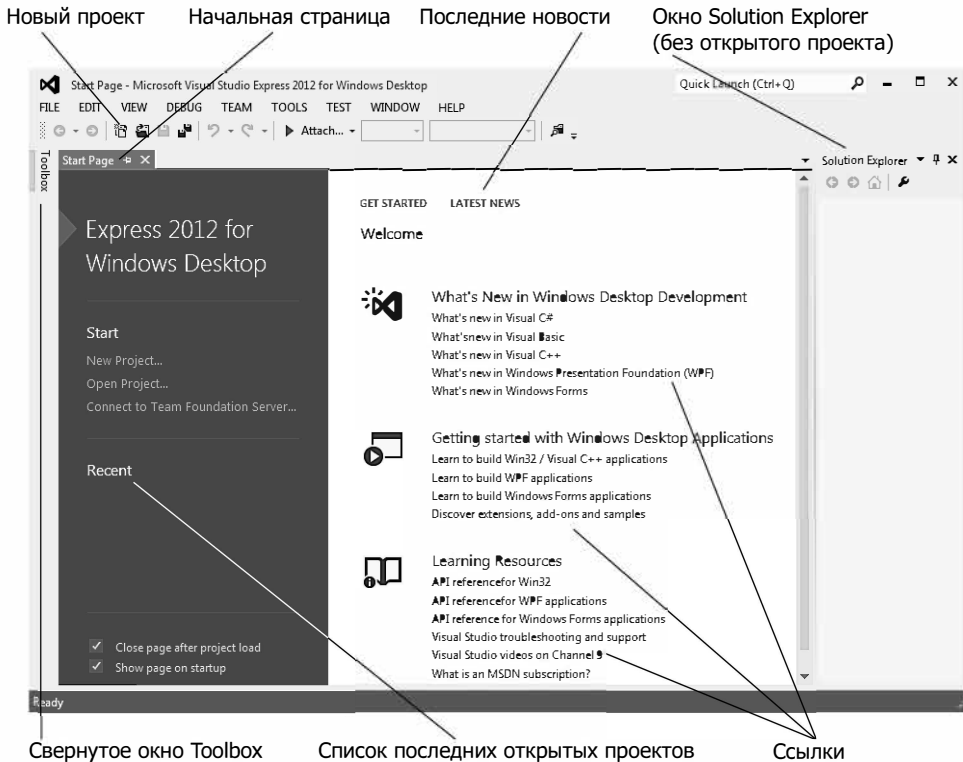
## 2.2. Обзор Visual Studio Express 2012 IDE

Visual Studio существует в нескольких версиях. В большинстве примеров книги используется Visual Studio Express 2012 for Windows Desktop. За информацией об установке программного обеспечения обращайтесь к разделу «Подготовка к работе». Наши иллюстрации и описания ориентированы на Visual Studio Express 2012 for Windows Desktop. Примеры также будут работать в полных версиях Visual Studio — хотя некоторые параметры, меню и инструкции могут выглядеть иначе. В дальнейшем мы будем называть Visual Studio Express 2012 for Windows Desktop IDE просто «Visual Studio» или «IDE». Предполагается, что читатель имеет опыт работы в Windows.

Для обозначения команд меню будет использоваться знак ►. Например, запись FILE ► Open File... означает, что вам следует выбрать команду Open File... из меню FILE.

IDE запускается командой Пуск ► Все программы ► Microsoft Visual Studio 2012 Express ► VS Express for Desktop (в Windows 8 нажмите VS for Desktop на начальном экране). В начале

выполнения Express Edition выводится начальная страница (ил. 2.1). В зависимости от вашей версии Visual Studio ваша начальная страница может выглядеть иначе. На начальной странице находится список ссылок на ресурсы Visual Studio и на ресурсы в Интернете. К начальной странице можно в любой момент вернуться командой **VIEW** ► **Start Page**. [*Примечание:* Visual Studio поддерживает как темную тему оформления (с темным фоном окна и светлым текстом), так и светлую тему (со светлым фоном и темным текстом). В книге будет использоваться светлая тема. В разделе «Подготовка к работе» рассказано, как выбрать нужную тему.]



**Ил. 2.1.** Начальная страница в Visual Studio Express 2012 for Windows Desktop

### Ссылки на начальной странице

Ссылки на начальной странице выводятся в два столбца. Ссылки из раздела **Start** левого столбца позволяют перейти к созданию новых приложений или продолжить работу над существующим проектом. В разделе **Recent** содержатся ссылки на недавно созданные или измененные проекты. Создавать новые или открывать существующие проекты также можно при помощи ссылок из раздела **Start**.

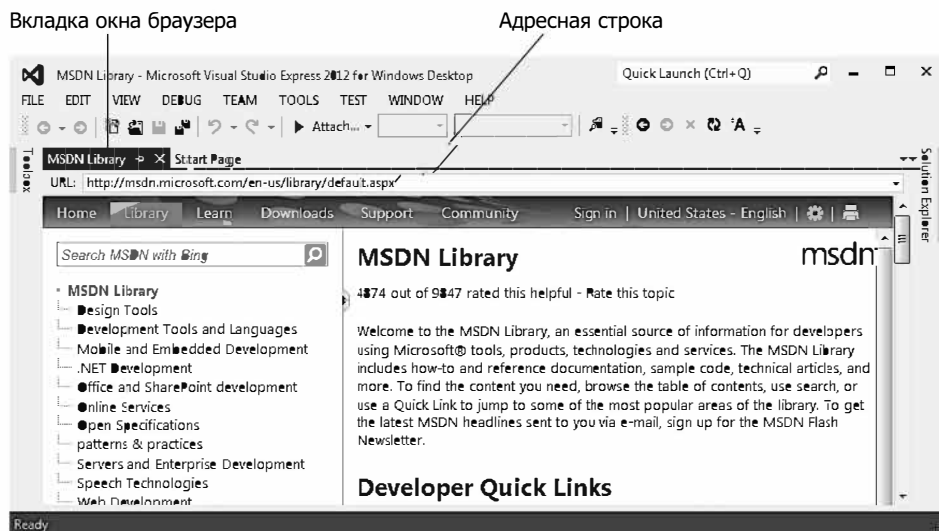
Правый столбец состоит из двух вкладок — **GET STARTED** (выбрана по умолчанию) и **LATEST NEWS**. Ссылки на вкладке **GET STARTED** предоставляют информацию о языках

программирования, поддерживаемых Visual Studio, и различных образовательных ресурсах. Для обращения к этой информации из IDE необходимо подключение к Интернету.

На вкладке **LATEST NEWS** находится кнопка **Enable RSS Feed**. Если щелкнуть на ней, IDE выводит ссылки на последние новости по поводу Visual Studio (например, обновления и исправления ошибок) и информацию по нетривиальным вопросам разработки. Более подробную информацию по Visual Studio можно найти в MSDN (Microsoft Developer Network) Library по адресу

*[msdn.microsoft.com/en-us/library/default.aspx](http://msdn.microsoft.com/en-us/library/default.aspx)*

На сайте MSDN собраны статьи, загружаемые материалы и учебники по технологиям, представляющим интерес для разработчиков Visual C#. Вы также можете просматривать веб-страницы из IDE командой **VIEW ► Other Windows ► Web Browser**. Чтобы запросить веб-страницу, введите ее URL-адрес в адресной строке (ил. 2.2) и нажмите клавишу **Enter** — естественно, ваш компьютер при этом должен быть подключен к Интернету. Просматриваемая веб-страница открывается в новой вкладке в IDE (см. ил. 2.2).



**Ил. 2.2.** Веб-страница MSDN Library в Visual Studio

## Создание нового проекта

Разработка приложения в Visual C# начинается с создания нового или открытия существующего проекта. Чтобы создать новый проект, выполните команду **FILE ► New Project...**, а чтобы открыть существующий — команду **FILE ► Open Project...** Также можно щелкнуть на ссылках **New Project...** или **Open Project...** в разделе **Start**. Проект представляет собой группу взаимосвязанных файлов (например, содержащих код

Visual C# или графику, используемую в приложении). В Visual C# приложения организуются в *проекты* (projects) и *решения* (solutions), содержащие один или несколько проектов. Решения используются при создании крупномасштабных приложений. Большинство приложений, представленных в книге, состоят из решения с одним проектом.

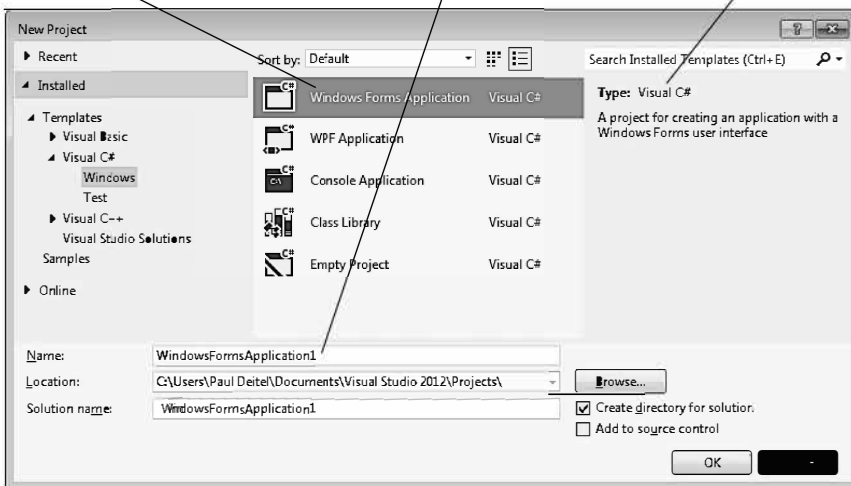
### Диалоговое окно New Project и шаблоны проектов

При выполнении команды **FILE** ► **New Project...** или щелчке на ссылке **New Project...** на начальной странице открывается диалоговое окно **New Project** (ил. 2.3).

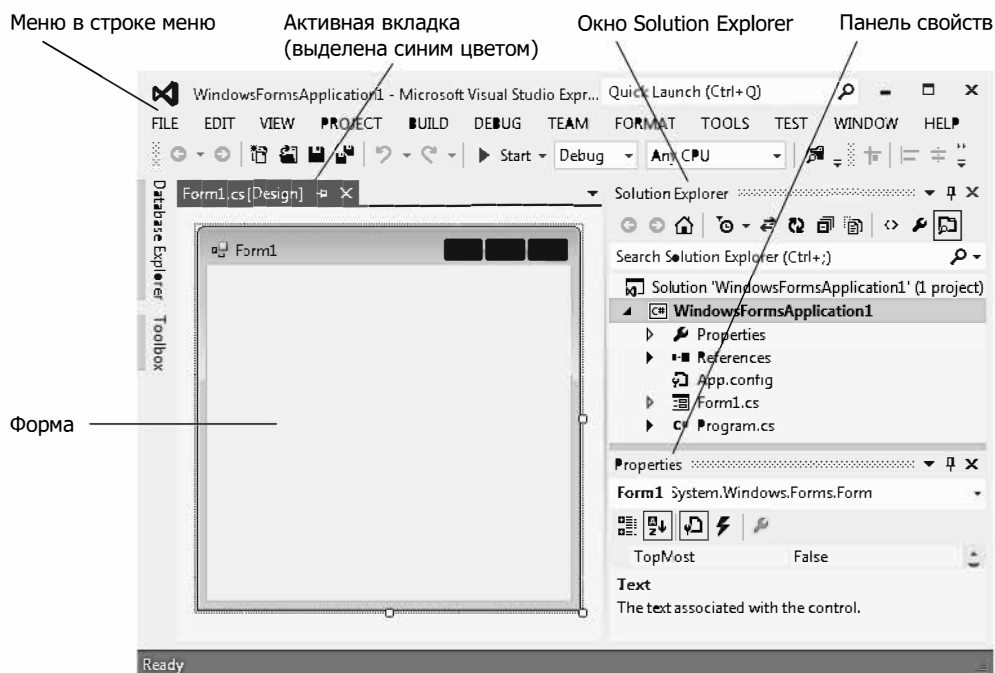
Visual Studio предоставляет разработчику несколько шаблонов (см. ил. 2.3) — типов проектов, которые могут создаваться в Visual C# и других языках. В их число входят шаблоны приложений Windows Forms, приложений WPF и многих других — полная версия Visual Studio содержит много дополнительных шаблонов. В этой главе мы построим приложение на базе шаблона Windows Forms Application. Приложение Windows Forms выполняется в операционной системе Windows (например, Windows 7 или Windows 8) и обычно обладает графическим интерфейсом (GUI), с которым взаимодействуют пользователи.

По умолчанию Visual Studio присваивает новому проекту и решению на базе шаблона Windows Forms Application имя `WindowsFormsApplication1` (см. ил. 2.3). Выберите шаблон **Windows Forms Application** и щелкните на кнопке **OK**, чтобы перевести IDE в режим конструирования (ил. 2.4). Функции этого режима предназначены для создания графического интерфейса приложения.

Шаблон Windows Forms Application (выделен в списке)      Описание выделенного проекта (предоставляется Visual Studio)      Имя проекта по умолчанию (предоставляется Visual Studio)



**Ил. 2.3.** Диалоговое окно New Project

**Ил. 2.4.** Режим конструирования (Design view) в IDE

## Формы и элементы управления

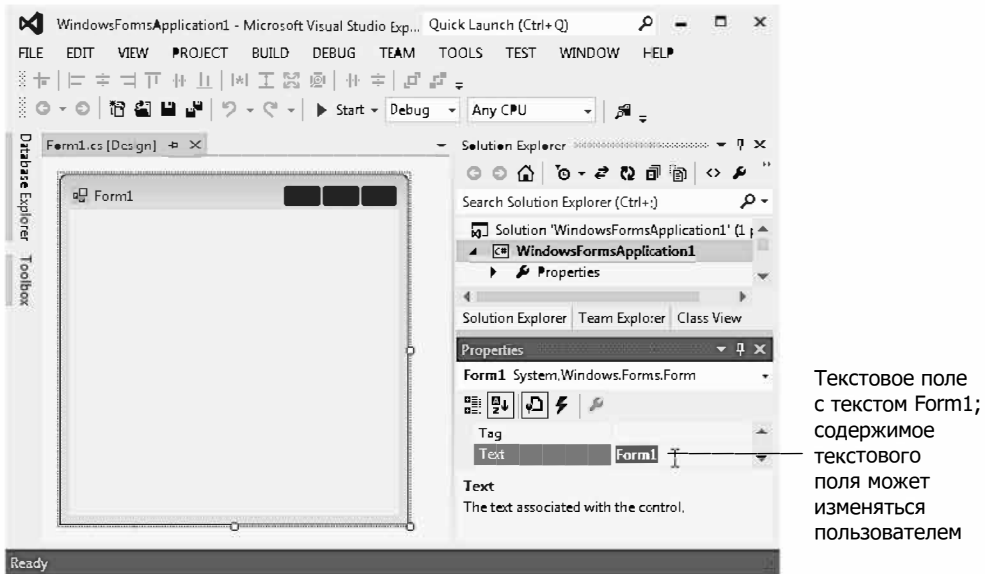
Прямоугольник с заголовком **Form1** (*форма*) представляет главное окно создаваемого приложения Windows Forms. Приложения Visual C# могут содержать несколько форм (окон); впрочем, большинство приложений, которые мы создадим в книге, будут использовать только одну форму. Вы научитесь изменять формы, добавляя к ним элементы управления, — в нашем примере на форме будут размещены надпись (**Label**) и графическое поле (**PictureBox**) (ил. 2.20). Надписи обычно содержат текст с описанием, а в графическом поле выводится изображение. Visual Studio содержит много готовых элементов управления и других компонентов, которые могут использоваться для создания и настройки поведения приложений.

В этой главе мы будем использовать готовые элементы управления из .NET Framework Class Library. Разместив элемент на форме, разработчик может изменить его свойства (см. раздел 2.4). Например, на ил. 2.5 показано, где изменяется текст заголовка формы, а на ил. 2.6 изображено диалоговое окно для выбора свойств шрифта в элементе управления.

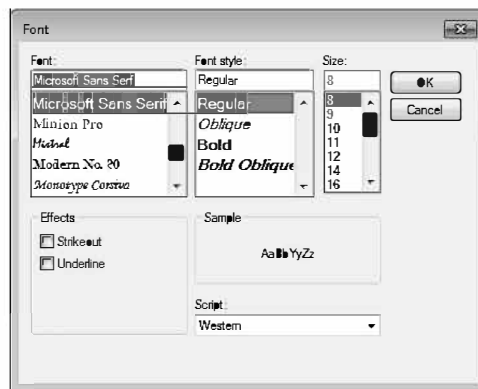
Форма и элементы управления образуют графический интерфейс приложения. Пользователь набирает данные на клавиатуре, щелкает кнопками мыши или вводит их другими способами. Графический интерфейс используется для отображения инструкций и другой информации, предназначенной для пользователя. Например,



в диалоговом окне New Project на ил. 2.3 пользователь щелкает кнопкой мыши, чтобы выбрать тип шаблона, а затем вводит название проекта с клавиатуры.



**Ил. 2.5.** Текстовое поле для изменения свойства в Visual Studio IDE



**Ил. 2.6.** Диалоговое окно для выбора свойств шрифта элемента управления

Имя каждого открытого документа выводится на корешке вкладки. Чтобы просмотреть документ при наличии нескольких открытых документов, щелкните на соответствующем корешке. Активная вкладка (вкладка с документом, отображаемым в настоящее время) выделена синим цветом (например, Form1.cs [Design] на ил. 2.4).

## 2.3. Строка меню и панель инструментов

Основные команды управления разработкой, сопровождением и выполнением приложений объединены в *меню*, которые размещаются в строке меню IDE (ил. 2.7). Набор отображаемых меню зависит от того, какая операция в настоящий момент выполняется в IDE.

Меню содержат группы взаимосвязанных команд, при выборе которых IDE выполняет конкретные действия — например, открывает окно, сохраняет файл, выводит файл на печать или выполняет приложение. Например, новые проекты создаются командой **FILE ► New Project....** На ил. 2.8 приведено краткое описание меню, изображенных на ил. 2.7.

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM FORMAT TOOLS TEST WINDOW HELP

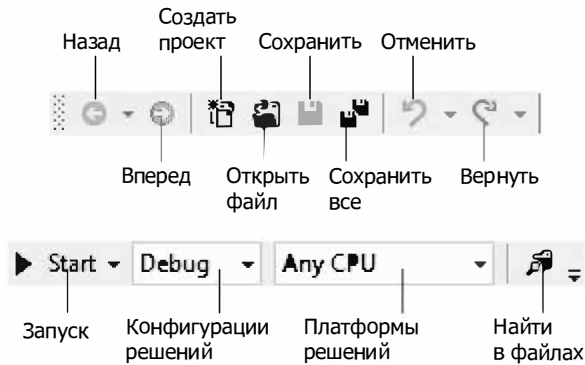
**Ил. 2.7.** Строка меню Visual Studio

Меню	Описание
FILE	Команды открытия, закрытия, добавления и сохранения проектов, а также печати данных проектов и выхода из Visual Studio
EDIT	Команды редактирования приложений: копирования, вырезания, вставки, отмены, повтора, удаления, поиска и выделения
VIEW	Команды отображения окон IDE (например, Solution Explorer, панели инструментов и окна свойств) и добавления панелей инструментов в IDE
PROJECT	Команды управления проектами и их файлами
BUILD	Команды преобразования приложения в исполняемую программу
DEBUG	Команды компиляции, отладки (то есть выявления и исправления ошибок) и запуска приложений
TEAM	Подключение к серверу Team Foundation Server для групп разработки, в которых над одним приложением работают сразу несколько участников
FORMAT	Команды, управляющие размещением и характеристиками элементов управления формы. Меню FORMAT выводится ТОЛЬКО при выделении управления в режиме конструирования
TOOLS	Команды вызова дополнительных инструментов и средств настройки IDE
TEST	Команды для выполнения различных видов автоматизированного тестирования в приложениях
WINDOW	Команды для свертывания, открытия, закрытия и отображения окон IDE
HELP	Команды для обращения к справочным средствам IDE

**Ил. 2.8.** Сводка меню Visual Studio, отображаемых в режиме конструирования

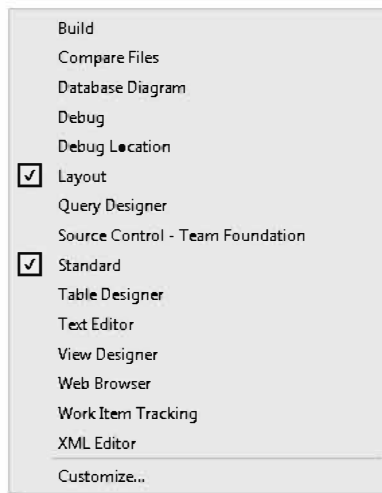
Многие часто используемые команды меню могут выполняться с панели инструментов (ил. 2.9). Кнопки, находящиеся на панели, являются графическими представлениями команд. По умолчанию при первом запуске Visual Studio отображается стандартная панель инструментов с кнопками часто используемых команд: открытия

файлов, добавления новых элементов в проект, сохранения файлов и запуска приложений (см. ил. 2.9). Состав кнопок на стандартной панели инструментов зависит от версии Visual Studio. Некоторые команды изначально заблокированы (недоступны для использования) и становятся доступными только при необходимости. Например, команда сохранения файла становится доступной тогда, когда вы начинаете редактировать файл.



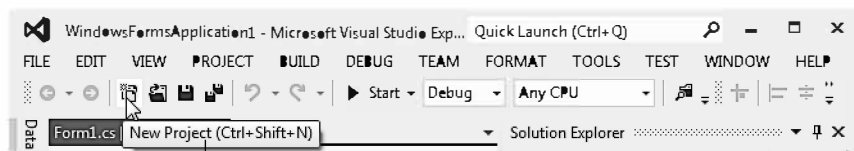
**Ил. 2.9.** Стандартная панель инструментов Visual Studio

Чтобы изменить набор отображаемых панелей инструментов, выполните команду **VIEW ► Toolbars** и выберите нужную панель в списке (ил. 2.10). Каждая выбранная вами панель инструментов отображается вместе с другими панелями в верхней части окна Visual Studio. Чтобы переместить панель инструментов, перетащите ее манипулятор (☷) в левой части панели. Чтобы выполнить команду с панели инструментов, щелкните на ее кнопке.



**Ил. 2.10.** Список панелей инструментов, которые можно добавить в IDE

Возможно, вам будет сложно запомнить, какую команду представляет та или иная кнопка на панели инструментов. Если навести указатель мыши на кнопку и задержать его, после небольшой паузы появляется подсказка с описанием (ил. 2.11). Подсказки помогают освоить возможности IDE и напоминают, что делает каждая из кнопок панелей.

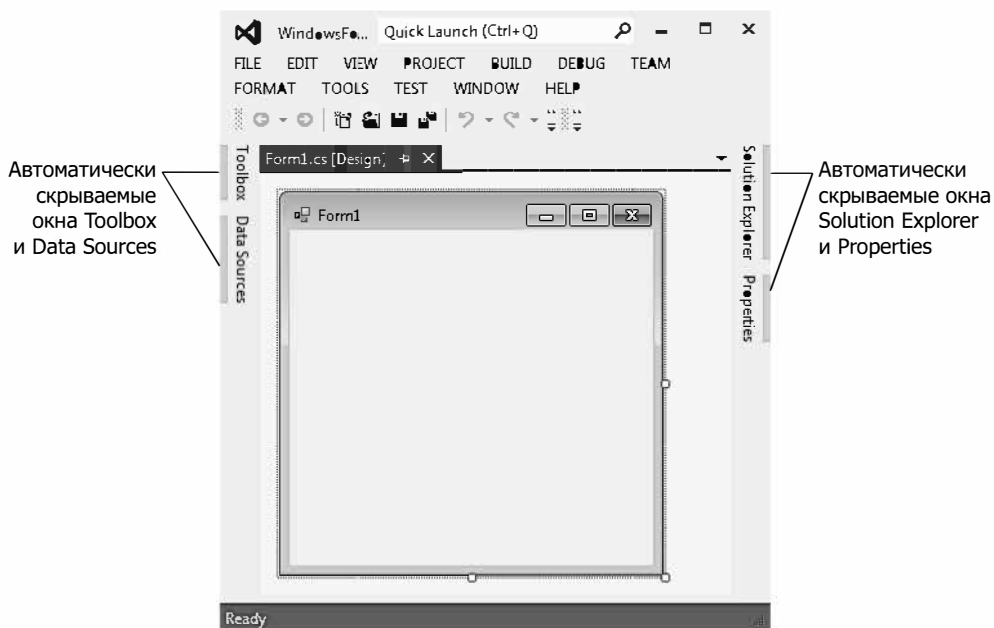


Подсказка появится, когда вы наведете указатель мыши на значок

Ил. 2.11. Подсказка

## 2.4. Перемещение в Visual Studio IDE

IDE предоставляет окна для работы с файлами проекта и настройки элементов управления. В этом разделе описаны некоторые окна, часто используемые при разработке приложений Visual C#. Чтобы обратиться к любому из этих окон, выберите его имя в меню VIEW.



Автоматически скрываемые окна Toolbox и Data Sources

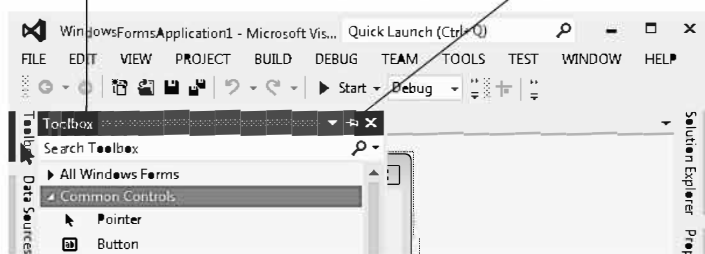
Автоматически скрываемые окна Solution Explorer и Properties

Ил. 2.12. Режим автоматически скрываемых окон

Для экономии места в Visual Studio реализована функция автоматически скрываемых окон. Когда для окна включен этот режим, у левого, правого или нижнего края окна IDE появляется корешок с именем окна (ил. 2.12). Если щелкнуть на нем, окно появляется на экране (ил. 2.13). Повторный щелчок на имени окна (или за его пределами) скрывает окно. Чтобы «закрепить» окно (то есть отключить режим автоматически скрываемых окон и оставить окно открытым), щелкните на кнопке с изображением булавки. При включенном режиме автоматически скрываемых окон булавка на кнопке расположена горизонтально (■, ил. 2.13), а когда окно «закреплено», булавка стоит вертикально (▮, ил. 2.14).

Развернутое окно Toolbox

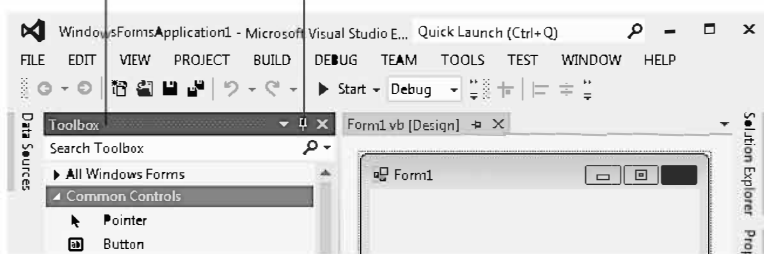
В режиме автоматически скрываемых окон булавка расположена горизонтально



**Ил. 2.13.** Отображение скрытого окна Toolbox

Окно Toolbox закреплено

В режиме закрепленного окна булавка расположена вертикально

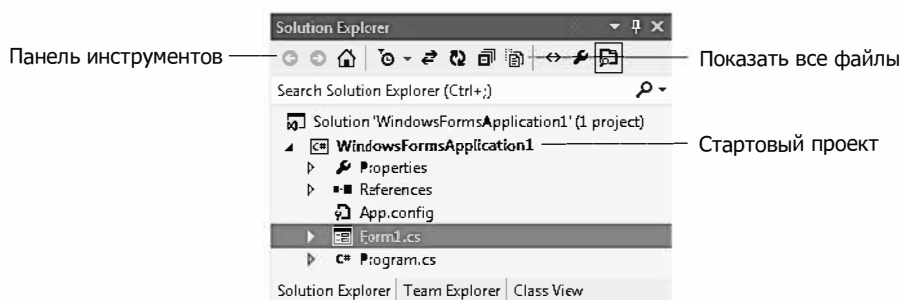


**Ил. 2.14.** Режим автоматически скрываемых окон отключен — окно «закреплено»

В следующих разделах рассматриваются три главных окна Visual Studio — Solution Explorer, окно свойств Properties и окно панели элементов Toolbox. Эти окна выводят информацию о проекте и помогают строить приложения.

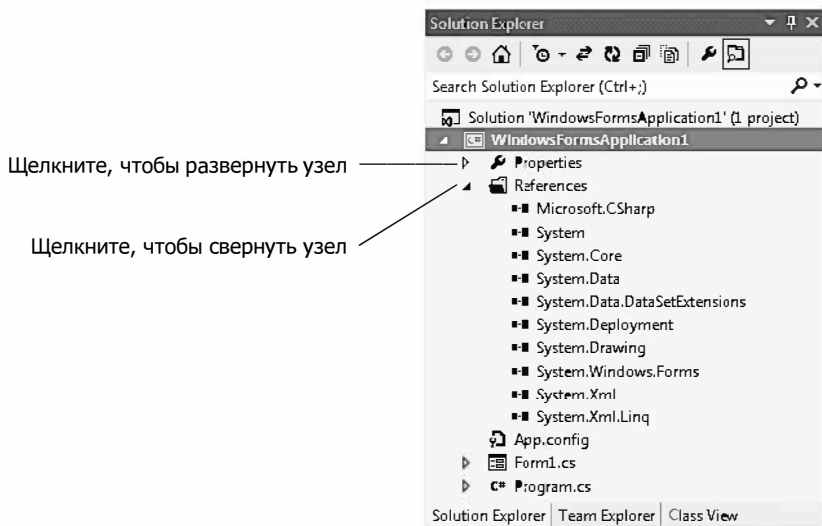
### 2.4.1. Solution Explorer

Окно Solution Explorer (ил. 2.15) предоставляет доступ ко всем файлам приложения. Если оно не отображается в IDE, выполните команду **VIEW ► Solution Explorer**. Когда вы открываете новое или существующее решение, его содержимое выводится в окне Solution Explorer.



**Ил. 2.15.** Окно Solution Explorer с открытым проектом

*Стартовым проектом* называется проект, который запускается при выборе команды **DEBUG ▶ Start Debugging** (или нажатии клавиши **F5**). Для решений с одним проектом, как в примерах книги, стартовый проект является единственным (в данном случае **WindowsFormsApplication1**). Имя стартового проекта выделяется жирным шрифтом в окне **Solution Explorer**. При первом создании приложения окно **Solution Explorer** выглядит так, как показано на ил. 2.15. Файл **Visual C#**, соответствующий форме на ил. 2.4, называется **Form1.cs** (выделен на ил. 2.15). Файлы **Visual C#** используют расширение **.cs**.



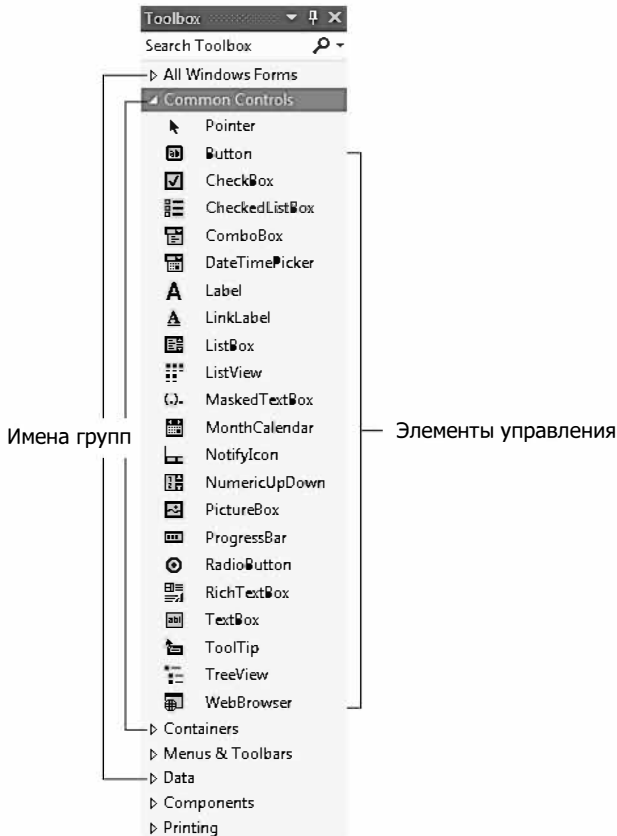
**Ил. 2.16.** Окно Solution Explorer с развернутым узлом References

По умолчанию в IDE отображаются только те файлы, которые вам, возможно, придется редактировать; другие файлы, сгенерированные IDE, остаются скрытыми. В окне **Solution Explorer** присутствует панель инструментов с несколькими кнопками. Кнопка **Show All Files** (см. ил. 2.15) отображает все файлы решения, включая сгенерированные IDE. Щелчок на стрелке слева от узла разворачивает или сворачивает этот узел.

Попробуйте щелкнуть на стрелке слева от **References**, чтобы просмотреть элементы, сгруппированные под этим заголовком (ил. 2.16). Сверните дерево повторным щелчком на стрелке. Эта схема также используется в других окнах Visual Studio.

## 2.4.2. Панель элементов

Чтобы вызвать окно панели элементов (**Toolbox**), выполните команду **VIEW ▸ Toolbox**. В нем содержатся элементы, используемые для модификации форм (ил. 2.17). В методологии визуального программирования разработчик «перетаскивает» элементы управления на форму, а IDE генерирует код создания этих элементов. Так код создается проще и быстрее, чем при самостоятельном написании. Чтобы вести машину, не обязательно знать, как устроен ее двигатель; точно так же не обязательно уметь строить элементы, чтобы использовать их. Повторное использование готовых элементов экономит время и деньги при разработке. Мы используем панель элементов позднее, при создании нашего первого приложения в этой главе.

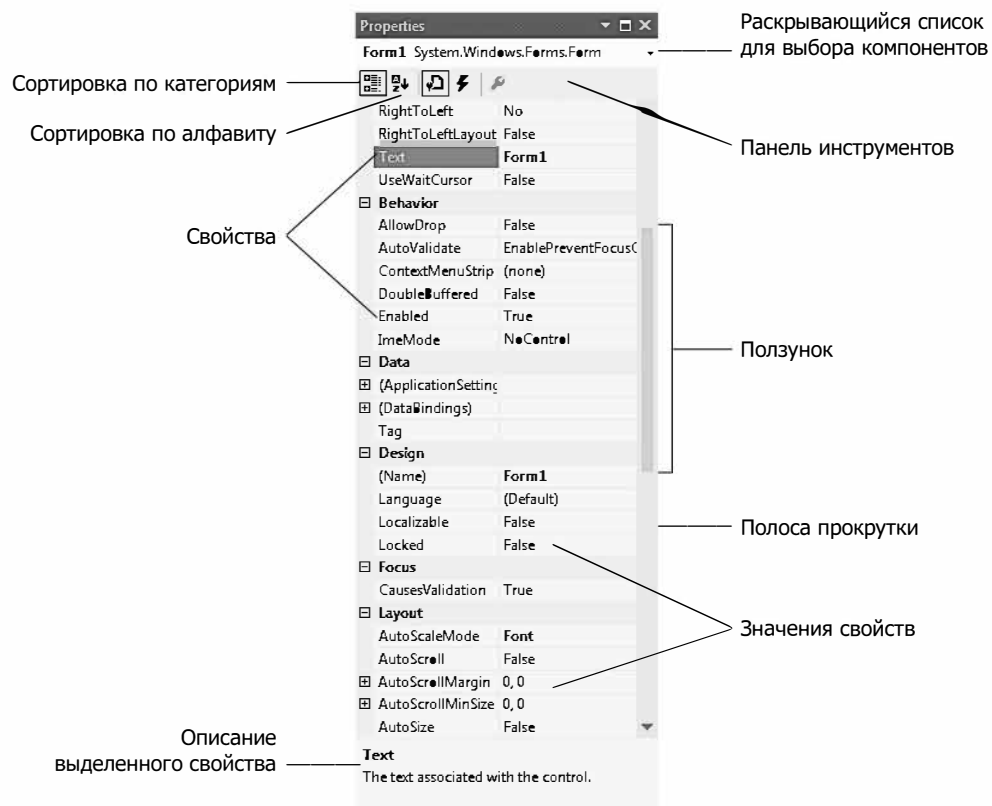


**Ил. 2.17.** Окно Toolbox с элементами управления из группы Common Controls

Готовые элементы управления на панели элементов сгруппированы по категориям (см. ил. 2.17). И снова обратите внимание на стрелки, используемые для сворачивания или разворачивания групп. Мы рассмотрим многие элементы управления панели элементов и их функциональность в следующих главах.

### 2.4.3. Окно свойств

Если окно свойств (Properties) не отображается под окном Solution Explorer, выполните команду **VIEW ► Properties Window**. В окне свойств выводятся свойства текущей выделенной формы, элемента управления или файла. Свойства описывают характеристики формы или элемента управления: размер, цвет, позицию и т. д. Разные формы и элементы управления обладают разными наборами свойств — описание свойства выводится в нижней части окна свойств при выделении этого свойства.



Ил. 2.18. Окно свойств

На ил. 2.18 изображено окно свойств формы Form1. В левом столбце перечислены свойства формы, а в правом выводятся текущие значения каждого свойства. Список свойств можно отсортировать либо по алфавиту (кнопка **Alphabetical**), либо по категориям (кнопка **Categorized**). В зависимости от размера окна свойств некоторые



свойства могут не поместиться на экране. Чтобы прокрутить список свойств, используйте полосу прокрутки или кнопки со стрелками у верхнего и нижнего края полосы. О том, как задавать значения свойств, будет рассказано позже в этой главе.

Окно свойств играет важнейшую роль в визуальной разработке — оно позволяет изменять свойства элементов управления на визуальном уровне, без написания кода. Разработчик видит, какие свойства можно изменить, а во многих случаях видит диапазон допустимых значений заданного свойства. В окне свойств выводится краткое описание выделенного свойства, помогающее понять его предназначение. Свойство можно быстро задать в этом окне, без написания какого-либо кода.

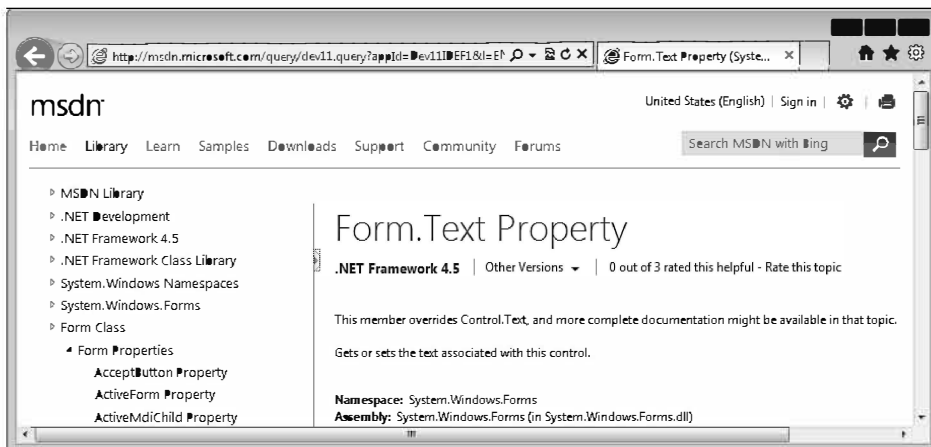
В верхней части окна свойств расположен раскрывающийся список для выбора компонентов. В нем можно выбрать форму или элемент управления, свойства которых вы хотите вывести в окне свойств, без выделения этой формы/элемента управления в графическом интерфейсе.

## 2.5. Справочная система

Компания Microsoft предоставляет обширную справочную документацию в меню **HELP**; это отличный способ быстро получить нужную информацию о Visual Studio, Visual C# и многом другом.

### Контекстная подсказка

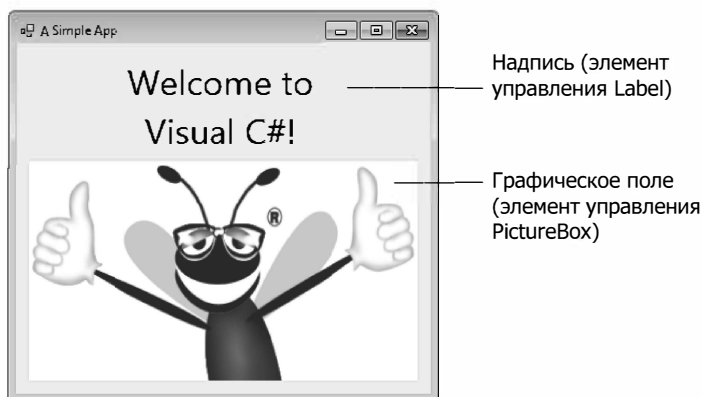
Visual Studio предоставляет контекстную справку по «текущему» объекту, находящемуся под указателем мыши. Чтобы получить контекстную справку, щелкните на объекте и нажмите клавишу **F1**. Справочная документация отображается в окне браузера. Чтобы вернуться в IDE, либо закройте окно браузера, либо выберите кнопку IDE на панели задач Windows. На ил. 2.19 изображена справочная страница для свойства **Text** объекта формы. Чтобы просмотреть ее, выделите форму, щелкните на ее свойстве **Text** в окне свойств и нажмите клавишу **F1**.



**Ил. 2.19.** Контекстная справка

## 2.6. Простое приложение с текстом и графикой

Сейчас мы создадим приложение, которое выводит приветственное сообщение и изображение жука — эмблемы Deitel & Associates. Приложение состоит из формы с двумя элементами управления: надписью (Label) и графическим полем (PictureBox). Окно выполняемого приложения изображено на ил. 2.20. Приложение и графический файл с жуком находятся в примерах этой главы. Инструкции по загрузке приведены в разделе «Подготовка к работе». Предполагается, что примеры установлены в каталог C:\examples на вашем компьютере.



**Ил. 2.20.** Простое приложение

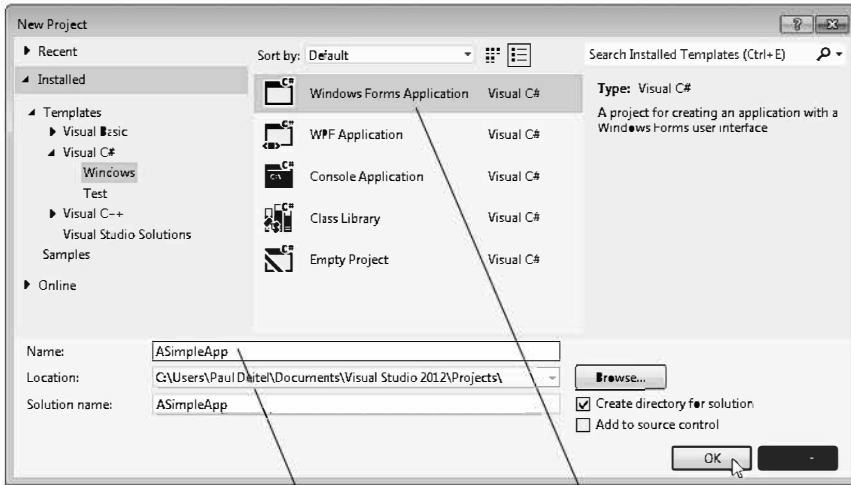
При создании этого приложения мы не напишем ни единой строки кода. Вместо этого будет использоваться методология визуальной разработки приложений: Visual Studio обрабатывает ваши действия (щелчки кнопкой мыши, перетаскивание) и генерирует код приложения. Написание кода приложения рассматривается начиная с главы 3.

В этой книге мы будем строить все более содержательные и полнофункциональные приложения, обычно состоящие из кода, написанного вами и сгенерированного Visual Studio. Возможно, недостаточно опытному разработчику будет трудно разобраться в сгенерированном коде, но вам почти не придется иметь с ним дела.

Визуальное программирование хорошо подходит для графических приложений, подразумевающих существенное взаимодействие с пользователем. Чтобы создать, сохранить, запустить и завершить свое первое приложение, выполните следующие действия:

1. Закройте открытый проект. Если проект, над которым вы работали ранее в этой главе, еще открыт, выполните команду **FILE ► Close Solution**.
2. Создайте новый проект. Чтобы создать новое приложение Windows Forms, выполните команду **FILE ► New Project...** для вызова диалогового окна **New Project**

(ил. 2.21). Выберите шаблон Windows Forms Application. Введите имя проекта ASimpleApp, укажите папку для сохранения (мы использовали вариант по умолчанию) и щелкните на кнопке ОК. Как было показано ранее в этой главе, при создании нового приложения Windows Forms среда разработки открывается в режиме конструирования (то есть приложение находится в процессе построения, а не выполнения).

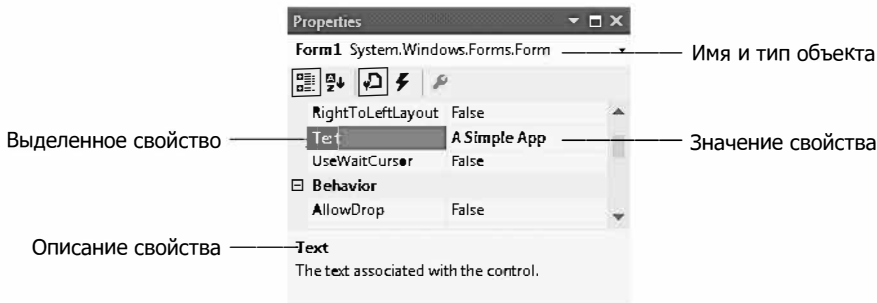


Здесь вводится имя нового проекта

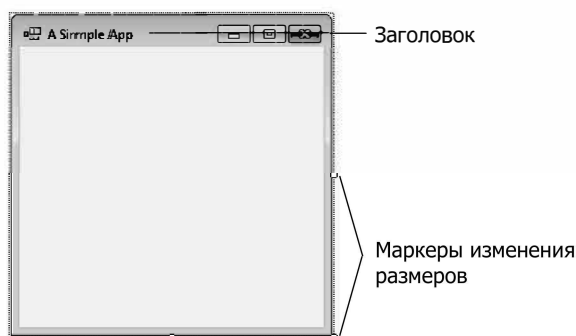
Выберите шаблон Windows Forms Application

**Ил. 2.21.** Диалоговое окно New Project

- Введите текст строки заголовка формы. Этот текст определяется свойством Text объекта Form (ил. 2.22). Если окно свойств не открыто, выполните команду **VIEW ► Properties Window**. Щелкните в любой точке формы, чтобы вызвать набор свойств формы в окне свойств. В текстовом поле справа от свойства Text введите строку A Simple App, как показано на ил. 2.22. Нажмите клавишу Enter; заголовок формы немедленно обновится (ил. 2.23).

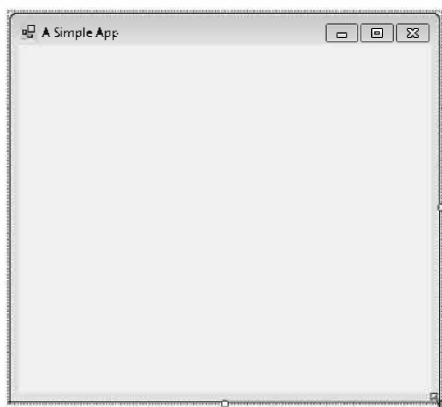


**Ил. 2.22.** Задание свойства Text в окне свойств



**Ил. 2.23.** Форма с маркерами изменения размеров

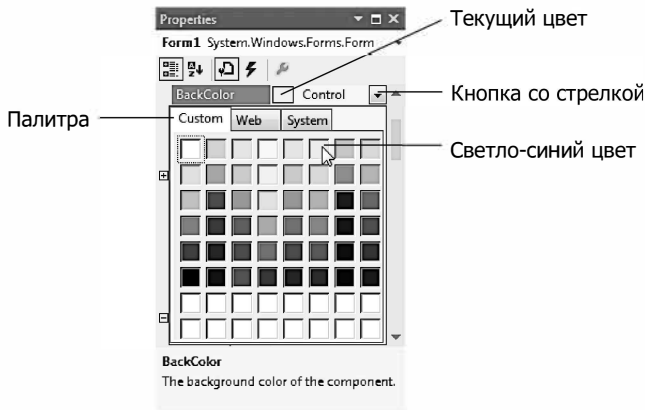
4. Измените размеры формы. Перетащите правый нижний маркер изменения размеров (маленькие белые квадратики вокруг формы на ил. 2.23), чтобы увеличить размеры формы (ил. 2.24).



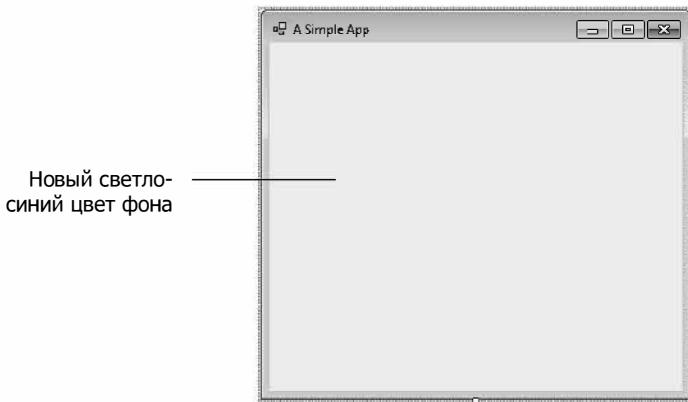
**Ил. 2.24.** Измененная форма

5. Измените цвет фона формы. Свойство `BackColor` задает цвет фона формы или элемента управления. Если щелкнуть на свойстве `BackColor` в окне свойств, рядом со значением свойства появляется стрелка (ил. 2.25). При щелчке на кнопке со стрелкой выводится набор возможных значений, зависящий от свойства. В нашем примере выводится набор вкладок `Custom`, `Web` и `System` (по умолчанию). Щелкните на кнопке `Custom`, чтобы вызвать палитру (сетку с образцами цветов). Щелкните на квадратике светло-синего цвета. При выборе цвета палитра закрывается, а фон формы окрашивается в светло-синий цвет (ил. 2.26).
6. Добавьте на форму надпись. Если окно панели элементов не отображается, выполните команду **VIEW ► Toolbox**. Для типа приложения, создаваемого в этой главе, обычно используются элементы управления из группы `All Windows Forms` или `Common Controls`. Если какая-то из этих групп свернута, разверните ее щелчком

на стрелке слева от имени группы (ил. 2.17). Затем сделайте двойной щелчок на элементе управления Label1 в окне панели элементов.

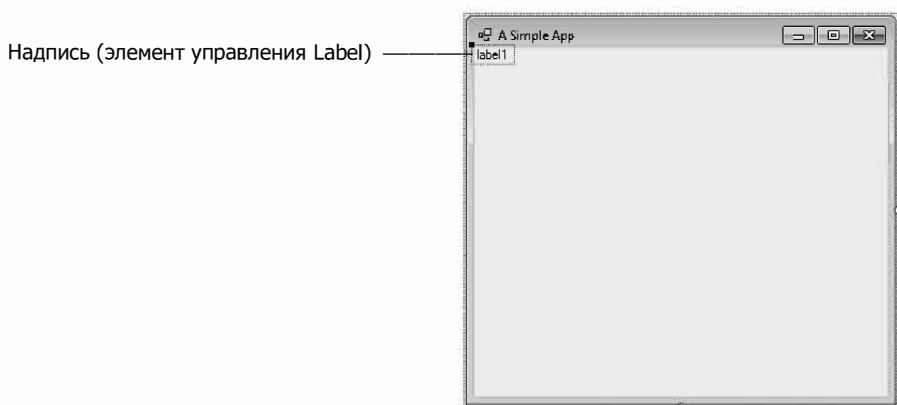


**Ил. 2.25.** Изменение свойства BackColor формы



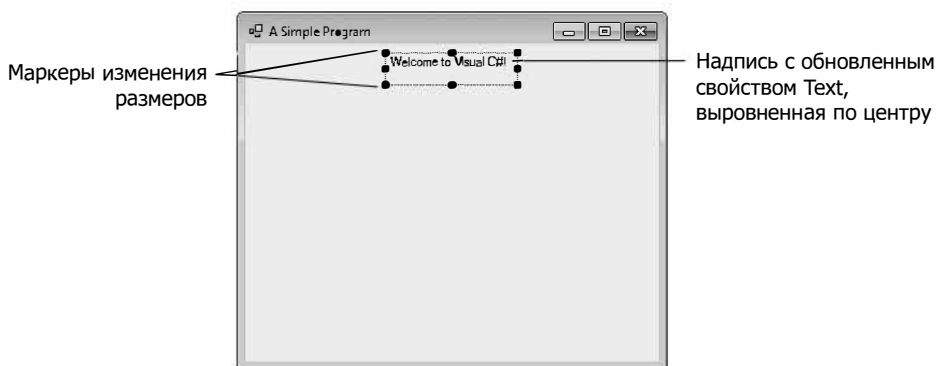
**Ил. 2.26.** Форма с измененным свойством BackColor

В левом верхнем углу формы появляется надпись (см. ил. 2.27). [Примечание: если форма находится за панелью элементов, возможно, вам придется скрыть панель, чтобы увидеть надпись.] И хотя двойной щелчок на любом элементе управления в окне панели элементов размещает его на форме, вы также можете «перетаскивать» элементы управления с панели элементов на форму — и возможно, этот способ вам покажется более удобным, потому что элемент можно разместить в любом нужном месте. В надписях по умолчанию отображается текст label1. При добавлении надписи на форму IDE задает значение свойства BackColor надписи свойству BackColor формы. Чтобы изменить цвет фона надписи, измените ее свойство BackColor.



**Ил. 2.27.** Добавление надписи на форму

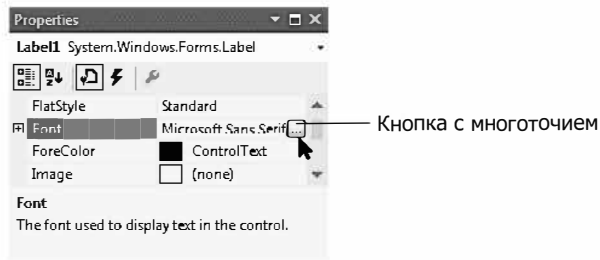
Измените параметры оформления надписи. Щелкните на надписи, чтобы выделить ее. Свойства элемента управления отображаются в окне свойств. Свойство `Text` элемента управления `Label` определяет текущий текст надписи. У каждого из элементов управления `Form` и `Label` имеется собственное свойство `Text` — формы и элементы управления могут иметь одноименные свойства (`BackColor`, `Text` и т. д.), это не приводит к конфликтам. Задайте свойству `Text` надписи значение `Welcome to Visual C#!`. Размеры элемента управления автоматически изменяются так, чтобы весь текст поместился в одной строке. Чтобы задать размеры элемента управления `Label` вручную, задайте его свойству `AutoSize` значение `True`. Измените размеры надписи (используя маркеры изменения размеров), чтобы текст помещался в элементе. Разместите надпись у верхнего края формы по центру; используйте перетаскивание мышью или клавиши `←` и `→` (ил. 2.28). Также для выравнивания надписи по центру формы можно выделить ее и выполнить команду **FORMAT** ► **CenterInForm** ► **Horizontally**.



**Ил. 2.28.** Форма после изменения свойств `Form` и `Label`

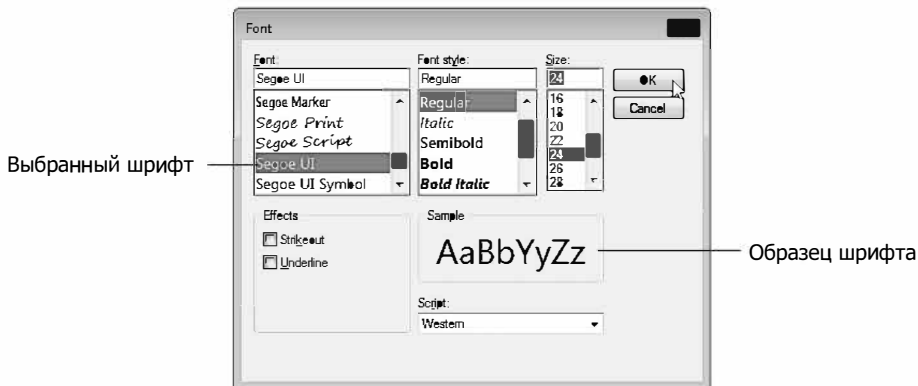
7. Задайте размер шрифта надписи. Чтобы изменить гарнитуру и начертание текста, выделите значение свойства `Font`; рядом со значением появляется кнопка с

многоточием (ил. 2.29). Если щелкнуть на ней, на экране появляется диалоговое окно для выбора дополнительных значений — в данном случае это окно Font (ил. 2.30). В этом окне можно выбрать имя шрифта (список доступных шрифтов зависит от системы), начертание (обычное, курсивное, полужирное и т. д.) и размер шрифта (16, 18, 20 и т. д.). В поле Sample выводится образец текста, оформленного заданным шрифтом.



Ил. 2.29. Окно свойств со свойством Font надписи

Выберите для свойства Font значение Segoe UI — шрифт, рекомендованный Microsoft для пользовательских интерфейсов. Задайте свойству Size значение 24 (пункта) и щелкните на кнопке ОК. Если текст надписи не помещается в одной строке, он переносится на следующую строку. Измените размеры надписи так, чтобы слова «Welcome to» выводились в первой строке, а слова «VisualC#!» — во второй. Снова выровняйте надпись по центру формы.



Ил. 2.30. Диалоговое окно Font для выбора шрифта, начертания и размера

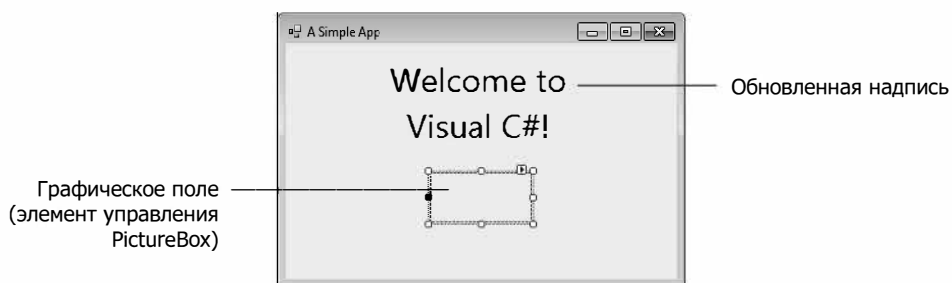
Выровняйте текст надписи. Задайте свойство TextAlign надписи, которое определяет способ выравнивания текста. На экране появляется сетка 3 × 3 из кнопок, представляющих разные способы выравнивания (ил. 2.31). Позиция каждой кнопки соответствует местонахождению текста в надписи. Для нашего приложения задайте свойству TextAlign значение MiddleCenter — в этом варианте текст надписи выравнивается по центру по горизонтали и вертикали. Другие значения TextAlign

позволяют расположить текст в нужном месте надписи. С некоторыми способами выравнивания вам придется изменить размеры надписи, чтобы текст лучше размещался в ней.



**Ил. 2.31.** Выравнивание текста по центру надписи

- Добавьте на форму графическое поле — элемент управления для вывода графики. Это делается примерно так же, как на шаге 6, когда мы добавили на форму надпись. Найдите на панели элементов (см. ил. 2.17) элемент `PictureBox` и сделайте на нем двойной щелчок. Когда графическое поле появится на форме, переместите его в нижнюю часть формы — используйте перетаскивание мышью или клавиши со стрелками (ил. 2.32).

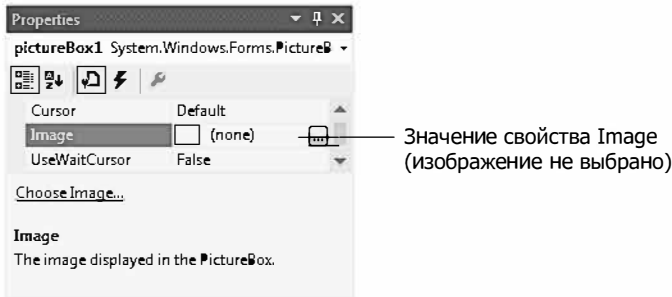


**Ил. 2.32.** Вставка и выравнивание графического поля

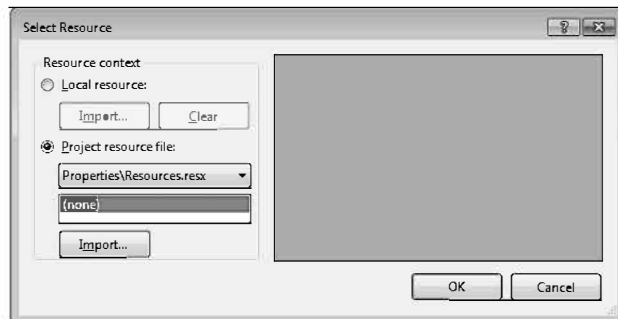
- Вставьте изображение. Щелкните на кнопке на графическом поле, чтобы вывести его свойства в окне свойств (ил. 2.33). Найдите и выделите свойство `Image`; так как изображение не выбрано, появляется строка (`none`). Щелкните на кнопке с многоточием (или ссылке `Choose Image...` над описанием свойства), чтобы вызвать диалоговое окно `Select Resource` (ил. 2.34), предназначенное для импортирования файлов, используемых приложением, — например, графики. Щелкните на кнопке `Import...`, найдите папку с изображением, выделите графический файл и щелкните на кнопке `OK`. Мы использовали файл `bug.png` из примеров этой главы. Эскиз изображения появляется в диалоговом окне `Select Resource` (ил. 2.35). Щелкните на кнопке `OK`, чтобы использовать изображение.



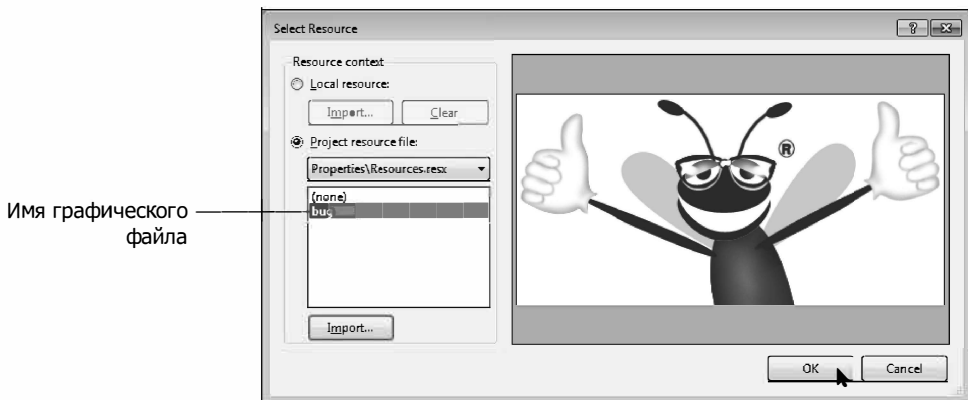
Поддерживаются такие графические форматы, как PNG (Portable Network Graphics), GIF (Graphic Interchange Format), JPEG (Joint Photographic Experts Group) и BMP (Windows bitmap). Чтобы масштабировать изображение по размерам PictureBox, задайте свойству SizeMode значение StretchImage (ил. 2.36). Увеличьте размеры графического поля (ил. 2.37).



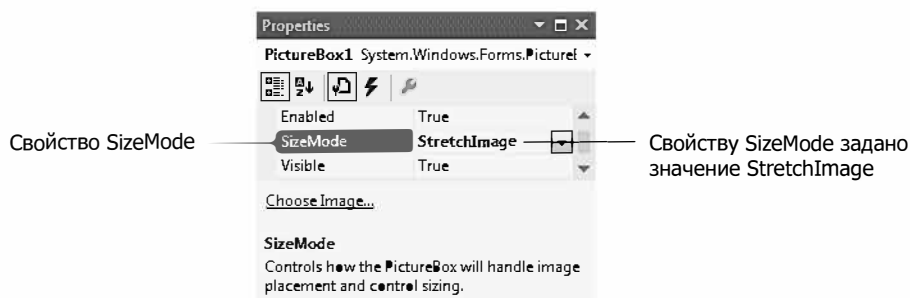
**Ил. 2.33.** Свойство Image элемента управления PictureBox



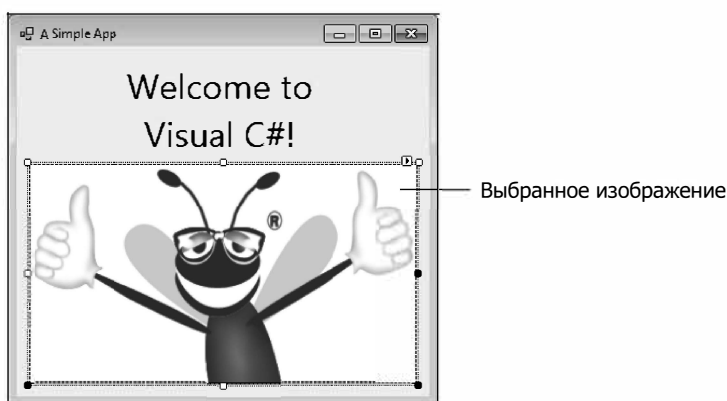
**Ил. 2.34.** Диалоговое окно Select Resource для выбора изображения



**Ил. 2.35.** Диалоговое окно Select Resource с эскизом выбранного изображения



**Ил. 2.36.** Масштабирование изображения по размерам PictureBox



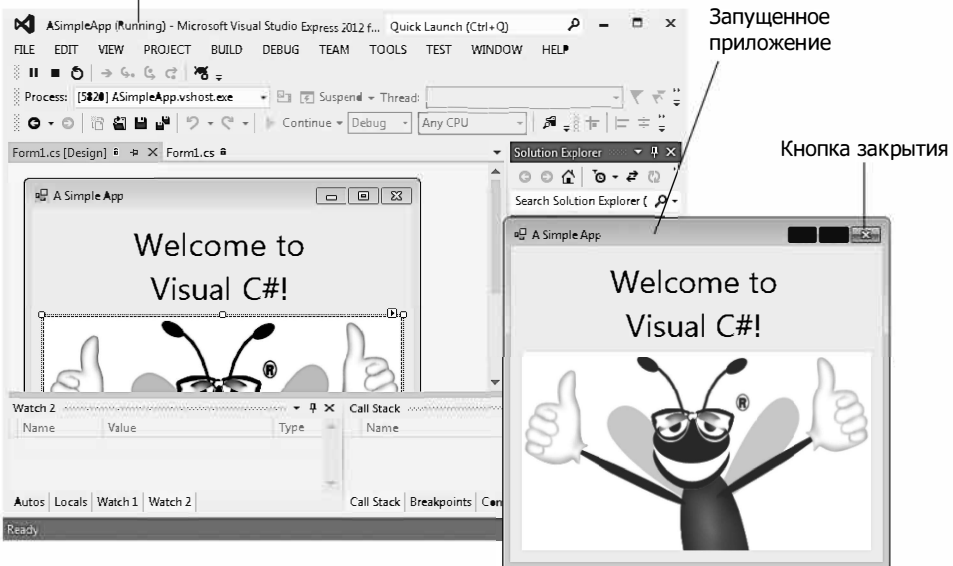
**Ил. 2.37.** Графическое поле с выбранным изображением

10. Сохраните проект. Выполните команду **FILE** ► **Save All**, чтобы сохранить всё решение. В файле решения (расширение `.sln`) хранится имя и местонахождение проекта, а файл проекта (расширение `.csproject`) содержит имена и местонахождения всех файлов проекта. Если вы захотите снова открыть свой проект в будущем, просто откройте его файл `.sln`.
11. Запустите проект. Вспомните, что до настоящего момента мы работали в режиме конструирования IDE (то есть приложение создавалось, а не выполнялось). В режиме выполнения многие функции IDE недоступны — в интерфейсе такие функции выделяются серым цветом. Текст `Form1.cs [Design]` на вкладке проекта (ил. 2.38) означает, что форма строится визуальным, а не программным способом. Если бы мы писали программный код, то вкладка содержала бы только текст `Form1.cs`. Если текст на корешке заканчивается звездочкой (\*), значит, файл содержит несохраненные изменения. Выполните команду **DEBUG** ► **Start Debugging** (или нажмите клавишу **F5**), чтобы запустить приложение. На ил. 2.39 изображена среда IDE в режиме выполнения. Многие кнопки панелей инструментов и меню во время выполнения становятся недоступными. Выполняемые приложения появляются в отдельном окне за пределами IDE, как показано в правой нижней части ил. 2.39.



**Ил. 2.38.** Отладка решения

Слово Running означает, что приложение в данный момент выполняется



**Ил. 2.39.** IDE в режиме выполнения с запущенным приложением на переднем плане

12. Завершите приложение. Щелкните на кнопке закрытия (в правом верхнем углу выполняемого приложения). Это действие прекращает выполнение приложения и возвращает IDE в режим проектирования. Также приложение можно завершить командой `DEBUG ► Stop Debugging`.

## 2.7. Итоги

В этой главе были представлены ключевые возможности Visual Studio IDE. Мы создали работоспособное приложение Visual C#, не написав ни единой строки кода. Разработка приложений Visual C# использует сочетание двух стилей: визуальная разработка позволяет легко создавать графические интерфейсы без написания рутинного кода, а «традиционное» программирование (о котором речь пойдет в главе 3) определяет поведение приложения.

В следующей главе будет рассмотрено «невизуальное», или «традиционное», программирование. Мы создадим свои первые приложения с самостоятельно написанным кодом Visual C# (вместо использования кода, сгенерированного Visual Studio). Также будут представлены основные концепции работы с памятью, математические операции и принятие решений.

# 3 Приложения C#

## 3.1. Введение

Пришло время заняться программированием традиционных приложений C#. Большинство приложений, представленных в книге, обрабатывают информацию и выводят результаты. В этой главе будут представлены *консольные приложения*, которые вводят и выводят текстовую информацию в консольном окне (в Windows оно называется окном командной строки).

Мы начнем с примеров, которые просто выводят сообщения на экран. Затем будет продемонстрировано приложение, которое получает два числа от пользователя, вычисляет их сумму и выводит результат. Мы выполним разные вычисления и сохраним результаты для использования в будущем. Во многих приложениях задействована логика принятия решений — последний пример этой главы демонстрирует основные принципы принятия решений. Например, приложение выводит сообщение о равенстве двух чисел только в том случае, если они имеют одинаковые значения. Код каждого примера будет подробно проанализирован, строка за строкой.

## 3.2. Простое приложение C#: вывод строки текста

Рассмотрим простое приложение, которое выводит строку текста. Приложение и результат его работы представлены на ил. 3.1, демонстрирующей некоторые важные возможности C#. Во всех программах, представленных в книге, присутствуют номера строк, не являющиеся частью кода C#. В разделе «Подготовка к работе» рассказано, как вывести номера строк в коде C#. Вскоре вы увидите, что вся настоящая работа приложения (а именно вывод приветствия «Welcome to C# Programming!») выполняется в строке 10. Но сначала мы подробно разберем каждую строку приложения.

Строка 1

```
// Ил. 3.1: welcome1.cs
```

начинается с символов //, которые указывают, что остаток строки содержит комментарий. Комментарии вставляются в документ для документирования кода

и удобства чтения. Компилятор C# игнорирует комментарии, поэтому при запуске приложения компьютер не выполняет для таких строк никаких действий. В этой книге каждое приложение начинается с комментария, в котором указан номер рисунка и имя файла с кодом.

Комментарий, начинающийся с `//`, называется *однострочным комментарием*, потому что он завершается в конце строки. Комментарий `//` также может начинаться в середине строки и продолжаться до конца этой строки (как в строках 7, 11 и 12).

```
1 // Ил. 3.1: Welcome1.cs
2 // Приложение для вывода текстового сообщения
3 using System;
4
5 public class Welcome1
6 {
7     // Метод Main начинает выполнение приложения C#
8     public static void Main( string[] args )
9     {
10         Console.WriteLine( "Welcome to C# Programming!" );
11     } // Конец Main
12 } // Конец класса Welcome1
```

```
Welcome to C# Programming!
```

**Ил. 3.1.** Приложение для вывода текста

Комментарии в ограничителях:

```
/* Этот комментарий может распространяться
на несколько строк */
```

могут быть разбиты на несколько строк. Такие комментарии начинаются с ограничителя `/*` и завершаются `*/`. Весь текст, заключенный в ограничители, игнорируется компилятором.



### ТИПИЧНАЯ ОШИБКА 3.1

Отсутствие одного из ограничителей в комментарии является синтаксической ошибкой. Синтаксис языка программирования определяет правила создания приложений на этом языке. Когда компилятор обнаруживает код, нарушающий правила языка C#, он не создает исполняемый файл, а выдает одно или несколько сообщений, помогающих найти и исправить ошибку в коде. Синтаксические ошибки также называются «ошибками компиляции», потому что они обнаруживаются во время компиляции. Пока в приложении не будут исправлены все синтаксические ошибки, его выполнение невозможно.

Строка 2

```
// Приложение для вывода текстового сообщения
```

содержит однострочный комментарий с описанием предназначения его приложения.

## Директива using

В строке 3

```
using System;
```

содержится директива `using`, которая сообщает компилятору, где следует искать класс, используемый в приложении. Одной из сильных сторон Visual C# является обширный набор заранее определенных классов, которые можно использовать повторно вместо того, чтобы заново «изобретать велосипед». Такие классы собираются в *пространства имен* — именованные наборы взаимосвязанных классов. Пространства имен .NET объединяются общим термином .NET Framework Class Library. Каждая директива `using` определяет переменную с заранее определенными классами, которые могут использоваться в приложении C#. Директива `using` в строке 3 указывает, что наш пример собирается использовать классы из пространства имен `System`, содержащего готовый класс `Console` (см. далее), используемый в строке 10, и много других полезных классов.



### КАК ИЗБЕЖАТЬ ОШИБОК 3.1

Если вы забудете включить директиву `using` для пространства имен, к которому принадлежит используемый в приложении класс, обычно это приводит к ошибке вида «The name 'Console' does not exist in the current context». Получив такое сообщение, проверьте наличие необходимых директив `using` и правильность написания имен (включая использование прописных и строчных букв).

Для каждого используемого класса .NET мы указываем пространство имен, в котором он находится. Эта информация важна, потому что она помогает найти описания классов в документации .NET. Веб-версия документации доступна по адресу [msdn.microsoft.com/en-us/library/ms229335.aspx](http://msdn.microsoft.com/en-us/library/ms229335.aspx).

Документация также доступна из меню **Help**. Щелкните на имени любого класса или метода .NET, нажмите клавишу F1 для получения дополнительной информации. Наконец, справку по содержимому конкретного пространства имен можно получить по адресу [msdn.microsoft.com/имя](http://msdn.microsoft.com/имя) — так, документация пространства имен `System` доступна по адресу [msdn.microsoft.com/System](http://msdn.microsoft.com/System).

## Пустые строки и пробелы

Строка 4 оставлена пустой. Пустые строки, пробелы и табуляции упрощают чтение кода; компилятор их игнорирует.

## Объявление класса

Строка 5

```
public class Welcome1
```

начинает объявление класса `Welcome1`. Каждое приложение состоит из минимум одного объявления класса, определяемого программистом. Ключевое слово `class` обозначает объявление класса, а после него следует имя класса (`Welcome1`). Ключевые слова зарезервированы для использования C#; они всегда записываются строчными буквами. Полный список ключевых слов C# приведен на ил. 3.2.

Ключевые слова				
abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			
Контекстные ключевые слова				
add	alias	ascending	async	await
by	descending	dynamic	equals	from
get	global	group	into	join
let	on	orderby	partial	remove
select	set	value	var	where
yield				

Ил. 3.2. Ключевые слова C#

### Имена классов

По общепринятой схеме первые буквы всех слов в имени класса начинаются с прописной буквы, а остальные буквы являются строчными (например, `SampleClassName`). Имя класса представляет собой идентификатор — последовательность символов, состоящую из букв, цифр и символов подчеркивания (`_`), которая не начинается с цифры и не содержит пробелов. Так, имена `Welcome1`, `identifier`, `_value` и `m_inputField1` являются допустимыми идентификаторами, а имя `7button` — не является, потому что оно начинается с цифры. Имя `input field` тоже не является допустимым идентификатором, потому что оно содержит пробел. Обычно идентификатор, не начинающийся с прописной буквы, не является именем класса. Язык C# различает регистр символов, так что `a1` и `A1` — разные идентификаторы (хотя оба являются действительными)<sup>1</sup>.

<sup>1</sup> Перед идентификатором также может стоять префикс `@`. Он означает, что слово должно интерпретироваться как идентификатор, даже если оно является ключевым словом (например, `@int`). Это позволяет использовать в C# код, написанный на других языках программирования .NET, даже если его идентификаторы совпадают с ключевыми словами C#. Контекстные ключевые слова на ил. 3.2 формально могут использоваться как идентификаторы вне контекста, в котором они являются ключевыми словами, но поступать так не рекомендуется.





### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.1

Всегда начинайте с прописной буквы каждое слово, входящее в идентификатор имени класса.



### ТИПИЧНАЯ ОШИБКА 3.2

В C# различается регистр символов. Несоблюдение регистра символов в идентификаторах обычно приводит к ошибке компиляции.

## Ключевое слово `public`

В главах 3–9 каждый определяемый нами класс начинается с ключевого слова `public`, то есть определяется как *открытый* класс. Классы более подробно рассматриваются начиная с главы 10. При сохранении объявления открытого класса в файле имя файла обычно состоит из имени класса и расширения `.cs`. В нашем приложении объявление класса хранится в файле с именем `Welcome1.cs`.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.2

По общепринятой схеме имя файла, содержащего один открытый класс, состоит из имени класса и расширения `.cs`, с соблюдением регистра символов.

## Тело объявления класса

Левая фигурная скобка (строка 6 на ил. 3.1) открывает *тело* объявления каждого класса. Объявление класса должно завершаться соответствующей правой фигурной скобкой (строка 12). Строки 7–11 снабжены отступами в соответствии с общепринятыми правилами оформления кода (см. далее врезки «Приемы программирования»).



### КАК ИЗБЕЖАТЬ ОШИБОК 3.2

Когда вы включаете в свое приложение открывающую фигурную скобку `{`, немедленно введите закрывающую фигурную скобку `}`, установите курсор между скобками и начинайте вводить код. Это поможет предотвратить ошибки из-за отсутствующих скобок.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.3

Снабдите все тело каждого объявления класса одним «уровнем» отступа между левой и правой фигурными скобками, ограничивающими тело класса. Этот формат подчеркивает структуру объявления класса и упрощает его чтение. Форматирование кода также можно поручить IDE: выполните команду `Edit ▸ Advanced ▸ Format Document`.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.4

Выберите размер отступа и соблюдайте его в будущем. Для создания отступов можно воспользоваться клавишей `Tab`, но ширина табуляции зависит от текстового редактора. Мы рекомендуем установить ширину каждого уровня отступа равной трем пробелам (см. раздел «Подготовка к работе»).



### ТИПИЧНАЯ ОШИБКА 3.3

Отсутствие парной фигурной скобки является синтаксической ошибкой.

#### Метод Main

Строка 7

```
// Метод Main начинает выполнение приложения C#
```

является комментарием, описывающим назначение строк 8–11 приложения.

Строка 8

```
public static void Main( string[] args )
```

определяет точку входа каждого приложения. Круглые скобки после идентификатора `Main` указывают на то, что перед нами основной структурный элемент любого приложения, называемый *методом*. Объявление класса обычно содержит один или несколько методов. Имена методов обычно строятся по той же схеме назначения регистра символов, что и имена классов. В каждом приложении один из методов класса должен называться `Main` (обычно определяемый так, как показано в главе 8); в противном случае приложение выполняться не будет. Методы выполняют операции и могут возвращать информацию после их завершения. Ключевое слово `void` (строка 8) означает, что метод не возвращает никакой информации после выполнения своей операции. О том, как вернуть информацию из метода, будет рассказано позже. Методы более подробно рассматриваются в главах 4 и 7. Содержимое круглых скобок метода `Main` рассматривается в главе 8, а пока просто скопируйте первую строку `Main` в своих приложениях.

#### Тело объявления метода

Левая фигурная скобка в строке 9 начинает тело объявления метода. Объявление должно завершаться соответствующей правой фигурной скобкой (строка 11). Строка 10 в теле метода снабжена отступом.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.5

Все тело метода в его объявлении следует снабдить отступом на один «уровень» между левой и правой фигурными скобками (как в объявлениях классов).

#### Вывод строки текста

Строка 10

```
Console.WriteLine( "Welcome to C# Programming!" );
```

приказывает компьютеру выполнить действие, а именно вывести последовательность символов, заключенную в кавычки. Такая последовательность символов иногда называется сообщением, строковым литералом или *строкой* (мы будем использовать именно этот термин). Пробелы и табуляции в строках не игнорируются компилятором.

Класс `Console` предоставляет стандартные средства ввода-вывода текста в консольном окне, в котором выполняется приложение. Метод `Console.WriteLine` выводит строку текста в консольном окне. В круглых скобках (строка 10) указан аргумент метода. Метод `Console.WriteLine` выводит в консольном окне свой аргумент. После выполнения этой операции `Console.WriteLine` переводит курсор (мигающая черта, обозначающая позицию вывода следующего символа) в начало следующей строки консольного окна. Перемещение курсора напоминает то, что происходит при нажатии пользователем клавиши `Enter` при вводе данных в текстовом редакторе — курсор перемещается к началу следующей строки в файле.

### Команды

Вся строка 10, включая вызов `Console.WriteLine`, круглые скобки, аргумент `"Welcome to C# Programming!"` в круглых скобках и точку с запятой (`;`), называется *командой*. Большинство команд завершается точкой с запятой. При выполнении команды в строке 10 в консольном окне выводится сообщение `Welcome to C# Programming!`. Метод обычно состоит из одной или нескольких команд, выполняющих операцию метода.



### КАК ИЗБЕЖАТЬ ОШИБОК 3.3

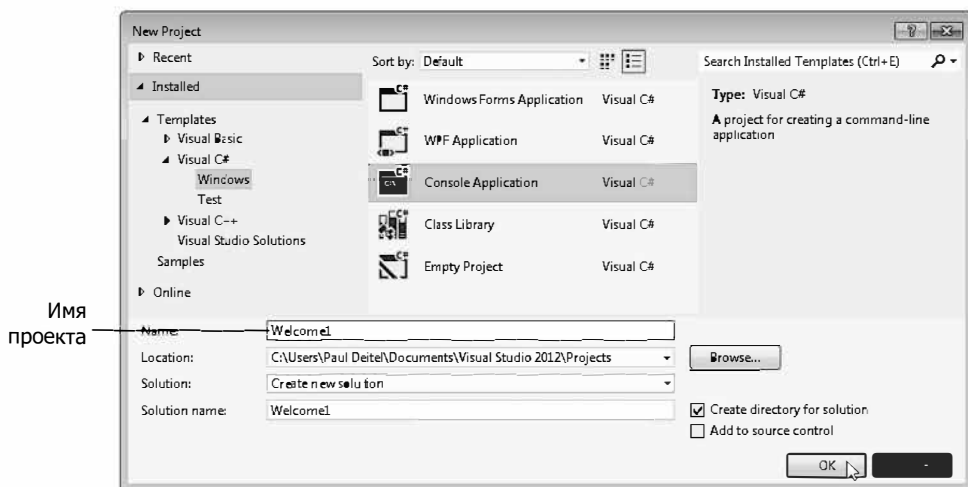
В сообщении о синтаксической ошибке, выданном компилятором, ошибка не всегда находится в указанной строке. Сначала проверьте приведенную строку. Если она не содержит синтаксических ошибок, проверьте несколько предшествующих строк.

## 3.3. Создание простого приложения в Visual Studio

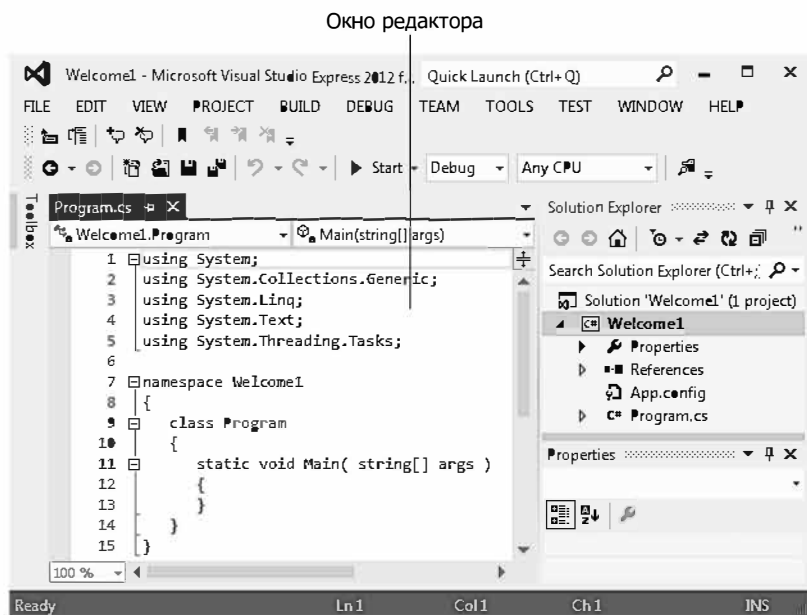
От подробного рассмотрения кода первого консольного приложения (см. ил. 3.1) мы переходим к подробному описанию его создания, компиляции и выполнения в Visual Studio 2012 Express for Windows Desktop (далее для краткости Visual Studio).

### Создание консольного приложения

Запустите Visual Studio и выполните команду `FILE ► New Project....` На экране появляется диалоговое окно `New Project` (ил. 3.3). В левой части окна, в категории `Installed ► Templates ► Visual C#` выберите подкатегорию `Windows`, затем в средней части диалогового окна выберите шаблон `Console Application`. В поле `Name` введите имя `Welcome1` и щелкните на кнопке `OK`, чтобы создать проект. По умолчанию папка проекта помещается в папку `Documents` вашей учетной записи в папке `VisualStudio 2012\Projects`. В IDE появляется открытое консольное приложение (ил. 3.4). В окне редактора уже присутствует код, сгенерированный IDE. Часть этого кода напоминает код на ил. 3.1, но есть и отличия. IDE вставляет дополнительный код для организации приложения и предоставления доступа к некоторым общим классам библиотеки `.NET Framework Class Library`. На данный момент этот код не актуален для нашего приложения; просто удалите его.



Ил. 3.3. Создание консольного приложения в диалоговом окне New Project



Ил. 3.4. IDE с открытым консольным приложением

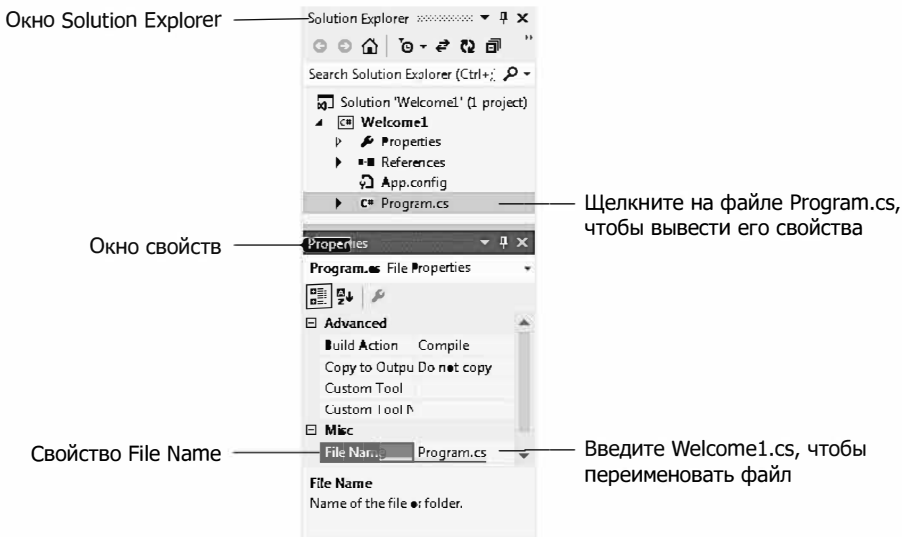
IDE использует цветное выделение синтаксиса, которое помогает различать разные элементы приложения. Например, ключевые слова выводятся синим цветом, а комментарии — зеленым. Чтобы настроить цветковые обозначения в редакторе кода, выполните команду **Tools** ► **Options...**. На экране появляется диалоговое окно **Options**. Разверните узел **Environment** и выберите категорию **Fonts and Colors**. Здесь можно изменить цвета различных элементов кода.

## Настройка окна редактора

Visual C# предоставляет широкие возможности настройки параметров процесса работы. В разделе «Подготовка к работе» мы показывали, как настроить IDE для вывода номеров строк в левой части окна редактора и как выбрать размер отступов, чтобы он соответствовал приводимым примерам.

## Переименование файла приложения

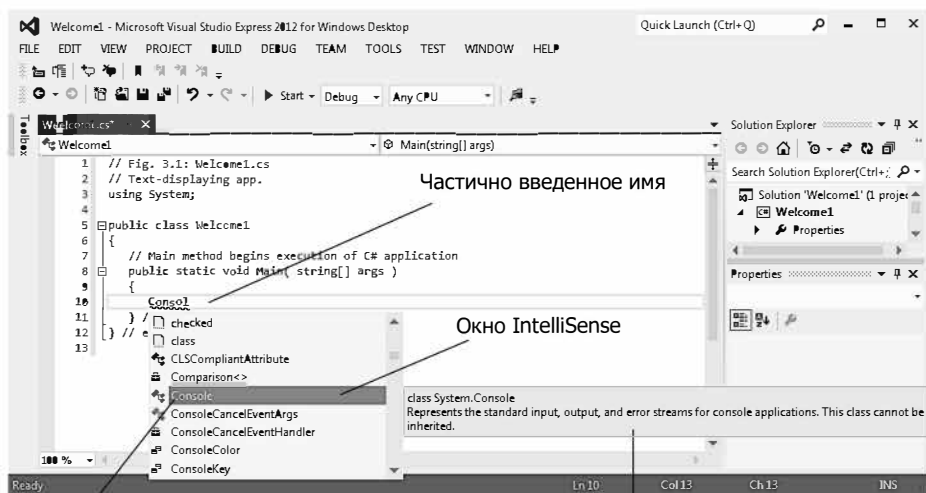
В приложениях, создаваемых в книге, мы заменяем имя файла исходного кода по умолчанию (например, `Program.cs`) более содержательным именем. Чтобы переименовать файл, щелкните на строке `Program.cs` в окне `Solution Explorer`. В окне свойств выводятся свойства файла приложения (ил. 3.5). Измените значение свойства `File Name` на `Welcome1.cs` и нажмите клавишу `Enter`.



**Ил. 3.5.** Переименование файла программы в окне свойств

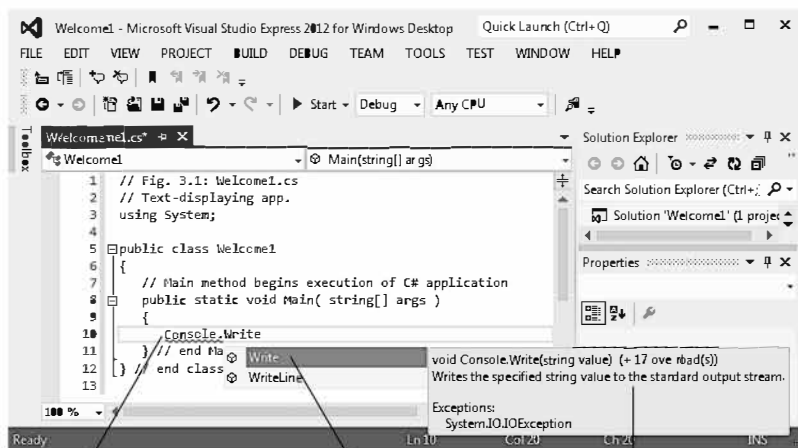
## Написание кода и использование IntelliSense

В окне редактора (см. ил. 3.4) замените код, сгенерированный IDE, кодом на ил. 3.1. Когда вы начнете вводить имя класса `Console` (строка 10), на экране появляется окно `IntelliSense` (ил. 3.6, а). В процессе ввода `IntelliSense` выводит список различных элементов, начинающихся с введенных вами букв (или содержащих их). `IntelliSense` также выводит подсказку с описанием первого подходящего элемента. Вы можете либо ввести остальные символы имени (например, `Console`), сделать двойной щелчок на имени в списке или завершить имя нажатием клавиши `Tab`. После ввода полного имени окно `IntelliSense` закрывается. Пока окно `IntelliSense` остается на экране, при нажатии клавиши `Ctrl` оно становится прозрачным, чтобы вы могли видеть находящуюся за ним часть окна.

**а) Окно IntelliSense появляется при вводе имени**

Наиболее подходящее совпадение выделяется в списке

Подсказка с описанием выделенного элемента

**б) Окно IntelliSense с именами методов, начинающимися с Write**

Частично введенное имя

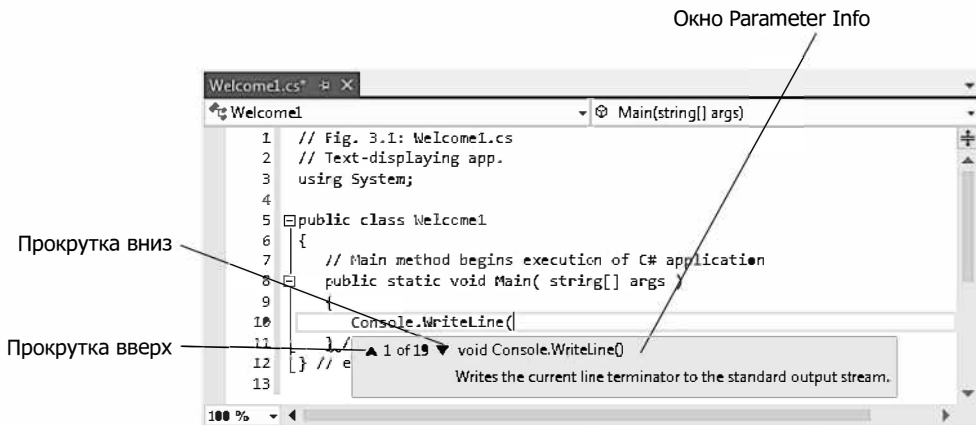
Выделенный элемент списка

Подсказка с описанием выделенного элемента

**Ил. 3.6. IntelliSense**

Когда вы вводите точку (.) после Console, на экране снова появляется окно IntelliSense. На этот раз в нем перечисляются только те члены класса Console, которые могут использоваться справа от точки (ил. 3.6, б). При вводе открывающей скобки ( после Console.WriteLine отображается окно Parameter Info (ил. 3.7) с информацией о параметрах метода. Как вы узнаете в главе 7, метод может существовать в нескольких

версиях. Иначе говоря, класс может определять несколько методов с совпадающими именами при условии, что эти методы различаются количеством и/или типами параметров (так называемая *перегрузка* методов). Одноименные методы обычно выполняют похожие операции. Окно **Parameter Info** сообщает, сколько версий существует у выделенного метода, а также предоставляет кнопки со стрелками для прокрутки. Например, метод `WriteLine` существует в 19 версиях, одна из которых используется в нашем приложении. Окно **Parameter Info** относится к числу вспомогательных инструментов IDE для упрощения разработки. В нескольких последующих главах информация, выводимая в этих окнах, будет рассмотрена более подробно. Окно **Parameter Info** особенно полезно тем, что оно позволяет увидеть разные варианты использования этого метода. Из кода на ил. 3.1 мы уже знаем, что вызов `WriteLine` будет выводить одну строку; следовательно, поскольку уже известно, какая версия `WriteLine` будет использоваться в программе, можно просто закрыть окно **Parameter Info** нажатием клавиши `Esc`.



Ил. 3.7. Окно **Parameter Info**

### Сохранение приложения

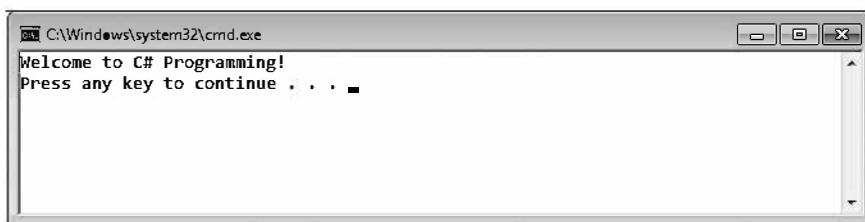
Завершив ввод кода приложения, сохраните проект командой **FILE ► SaveAll**.

### Компиляция и запуск приложения

Теперь вы готовы откомпилировать и выполнить свое приложение. В зависимости от типа проекта код может быть откомпилирован в исполняемые файлы с расширением `.exe`, библиотеки динамической компоновки с расширением `.dll` или в файлы с другими расширениями. Такие файлы называются *сборками* (assemblies) и являются блоками упаковки откомпилированного кода C#. Сборки содержат код приложения на языке MSIL (Microsoft Intermediate Language).

Чтобы откомпилировать приложение, выполните команду **BUILD ► Build Solution**. Если приложение не содержит синтаксических ошибок, то после успешной компиляции будет построен исполняемый файл (с именем `Welcome1.exe` в одном

из подкаталогов проекта). Чтобы выполнить его, нажмите клавиши Ctrl+F5; при этом управление передается методу Main (см. ил. 3.1). Если вы попытаетесь запустить приложение до его построения, то IDE сначала построит приложение, а потом запустит его только в том случае, если компиляция прошла без ошибок. Команда в строке 10 функции Main выводит сообщение `Welcome to C# Programming!`. На ил. 3.8 изображен результат выполнения этого приложения в консольном окне (командная строка). Оставьте проект приложения открытым в Visual Studio; мы вернемся к нему позже в этом разделе. [*Примечание:* окно консоли обычно имеет темный фон с белым текстом. Мы перенастроили его так, чтобы в нем использовался белый фон с черным текстом, чтобы оно лучше смотрелось в книге. Если вы предпочитаете этот вариант, щелкните на кнопке в левом верхнем углу консольного окна, выберите команду **Properties** и измените цвета на вкладке **Colors** открывшегося диалогового окна.]



**Ил. 3.8.** Выполнение консольного приложения

### Синтаксические ошибки, сообщения об ошибках и окно Error List

Вернемся к приложению в Visual Studio. В процессе ввода кода IDE реагирует либо применением цветового выделения элементов синтаксиса, либо выдачей сообщений об ошибках. Синтаксические ошибки могут происходить по разным причинам — например, из-за пропущенных круглых скобок и неправильно записанных ключевых слов.

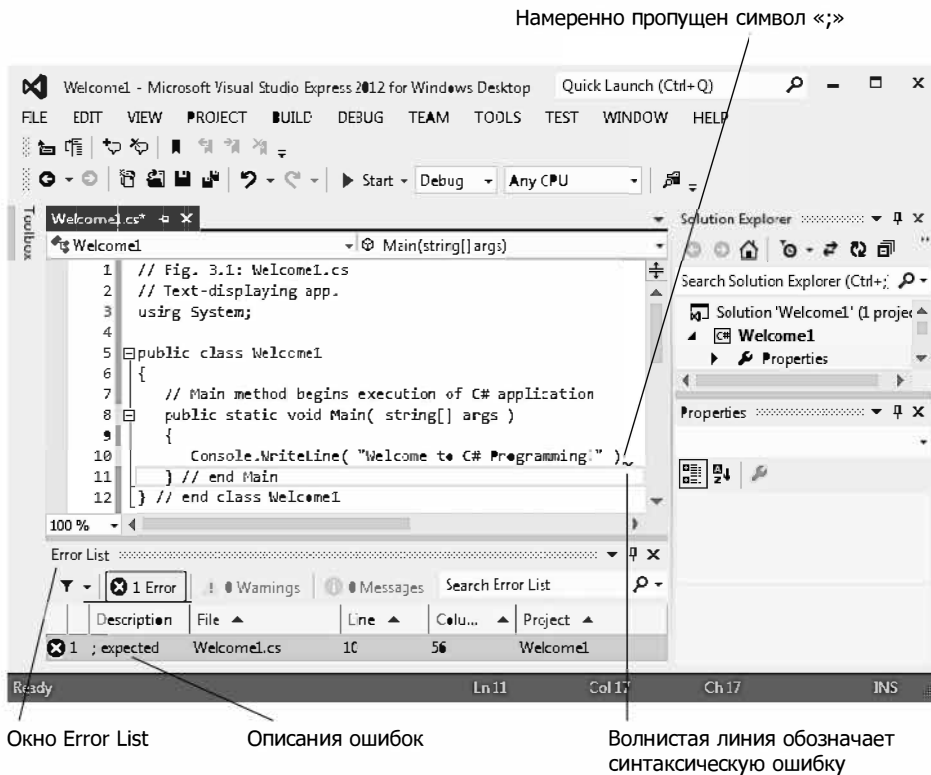
IDE выделяет местонахождение ошибки красной волнистой линией и выводит описание в окне **Error List** (ил. 3.9). Если окно **Error List** не отображается в IDE, выведите его командой **VIEW ► Error List**. На ил. 3.9 мы намеренно пропустили точку с запятой в конце команды в строке 10. Сообщение об ошибке указывает на отсутствие точки с запятой. Двойной щелчок на сообщении об ошибке в окне **Error List** осуществляет переход к позиции ошибки в коде.



#### КАК ИЗБЕЖАТЬ ОШИБОК 3.4

Одна синтаксическая ошибка может породить несколько записей в окне **Error List**. Если вы видите ошибку и знаете, как ее исправить, — исправьте и перекомпилируйте приложение; возможно, при этом исчезнут и другие ошибки.





Ил. 3.9. Сообщение о синтаксической ошибке в IDE

## 3.4. Изменение простого приложения C#

В этом разделе наше знакомство с программированием C# продолжится на паре примеров, которые представляют собой измененные версии примера на ил. 3.1.

### Вывод одной строки текста несколькими командами

Класс `Welcome2`, приведенный на ил. 3.10, использует две команды для получения такого же результата, как у кода на ил. 3.1. В дальнейшем мы будем выделять новые и самые важные места во всех листингах, как это сделано в строках 10–11 на ил. 3.10.

```

1 // Fig. 3.10: Welcome2.cs
2 // Вывод одной строки текста несколькими командами.
3 using System;
4
5 public class Welcome2
6 {
7     // Метод Main начинает выполнение приложения C#
8     public static void Main( string[] args )

```

Ил. 3.10. Приложение для вывода текста (продолжение 3.1)

```

9      {
10         Console.Write( "Welcome to " );
11         Console.WriteLine( "C# Programming!" );
12     } // Конец Main
13 } // Конец класса Welcome2

```

```
Welcome to C# Programming!
```

### Ил. 3.10. Приложение для вывода текста (окончание)

Приложение очень похоже на ил. 3.1, но пара различий все же имеется. В строке 2 указана цель приложения:

```
// Вывод одной строки текста несколькими командами.
```

В строке 5 начинается объявление класса `Welcome2`.

Строки 10–11 метода `Main`

```

Console.Write( "Welcome to " );
Console.WriteLine( "C# Programming!" );

```

выводят одну строку текста в консольном окне. Первая команда использует метод `Write` класса `Console` для вывода строки. В отличие от метода `WriteLine`, после вывода аргумента `Write` не переводит курсор в начало следующей строки в консольном окне — следующий выводимый символ следует сразу же за последним символом, введенным `Write`. Таким образом, строка 11 позиционирует первый символ в аргументе (буква «С») немедленно за последним символом, выводимым строкой 10 (пробел перед закрывающей кавычкой). Каждая команда `Write` продолжает выводить символы от позиции, в которой предыдущая команда `Write` вывела свой последний символ.

### Вывод нескольких строк текста одной командой

Одна команда также может вывести многострочный текст при помощи символов новой строки, которые сообщают методам `Write` и `WriteLine` класса `Console`, когда курсор следует перевести в начало следующей строки консольного окна. Как и пробелы с символами табуляции, символы новой строки относятся к категории *пропусков* (whitespace). Приложение на ил. 3.11 выводит четыре строки текста, при этом каждый переход на следующую строку обозначается символом новой строки.

```

1 // Ил. 3.11: Welcome3.cs
2 // Вывод нескольких строк текста одной командой.
3 using System;
4
5 public class Welcome2
6 {
7     // Метод Main начинает выполнение приложения C#
8     public static void Main( string[] args )
9     {
10         Console.Write( "Welcome\nto\nC#\nProgramming!" );
11     } // Конец Main
12 } // Конец класса Welcome2

```

```

Welcome
to
C#
Programming!

```

### Ил. 3.11. Вывод нескольких строк одной командой

Большая часть кода также не отличается от кода приложений на ил. 3.1 и 3.10, так что мы ограничимся рассмотрением изменений. В строке 2 указана цель приложения:

```
// Вывод нескольких строк текста одной командой.
```

В строке 5 начинается объявление класса `Welcome3`.

Строка 10 выводит четыре строки текста в консольном окне:

```
Console.WriteLine( "Welcome\nto\nC#\nProgramming!" );
```

Обычно символы строк выводятся точно в том виде, в котором они записаны в кавычках. Однако комбинация символов `\` и `n` (трижды встречающаяся в команде) *не отображается* на экране. Символ `\` (обратная косая черта) является *служебным префиксом*; он указывает `C#` на присутствие в строке «специального символа». Обнаружив в строке символ `\`, компилятор объединяет его со следующим символом для образования *служебной последовательности*. Комбинация `\n` представляет символ новой строки. Если она присутствует в текстовых данных, выводимых методами класса `Console`, то в этой позиции курсор переводится в начало следующей строки в консольном окне. На ил. 3.12 перечислены некоторые наиболее распространенные служебные последовательности и их влияние на вывод символов в консольном окне.

Служебная последовательность	Описание
<code>\n</code>	Символ новой строки; переводит курсор в начало следующей строки
<code>\t</code>	Горизонтальная табуляция; перемещает курсор к следующей позиции табуляции
<code>\"</code>	Кавычка; используется для включения в строку символа <code>"</code> — например, команда <code>Console.Write( "\"in quotes\"" );</code> выводит строку <code>"in quotes"</code>
<code>\r</code>	Возврат курсора; устанавливает курсор в начало текущей строки (без продвижения к следующей строке). Все символы, выведенные после символа возврата курсора, выводятся поверх символов, выведенных ранее в этой строке
<code>\\</code>	Обратная косая черта; используется для включения символа <code>\</code> в строку

**Ил. 3.12.** Наиболее популярные служебные последовательности

## 3.5. Форматирование текста методами Console.Write и Console.WriteLine

Консольные методы `Write` и `WriteLine` также позволяют выводить отформатированные данные. На ил. 3.13 метод `WriteLine` используется для вывода строк `"Welcome to"` и `"C#Programming!"`.

```
1 // Ил. 3.13: Welcome4.cs
2 // Вывод нескольких строк текста с форматированием.
3 using System;
4
5 public class Welcome4
```

**Ил. 3.13.** Вывод текста в виде строк (продолжение ➞)

```

6 {
7     // Метод Main начинает выполнение приложения C#
8     public static void Main( string[] args )
9     {
10         Console.WriteLine( "{0}\n{1}", "Welcome to", "C# Programming!" );
11     } // Конец Main
12 } // Конец класса Welcome4

```

```

Welcome to
C# Programming!

```

### Ил. 3.13. Вывод текста в виде строк (окончание)

В строке 10

```
Console.WriteLine( "{0}\n{1}", "Welcome to", "C# Programming!" );
```

вызывается метод `Console.WriteLine` для отображения выходных данных приложения. При вызове метода передаются три аргумента. Если методу передаются несколько аргументов, они разделяются запятыми (,).



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.6

Ставьте пробел после каждой запятой в списке аргументов, чтобы приложение лучше читалось.

Большинство команд завершается символом «точка с запятой» (;). Таким образом, строка 10 содержит всего одну команду. Длинные команды могут разбиваться на несколько строк, но при этом действуют некоторые ограничения.



#### ТИПИЧНАЯ ОШИБКА 3.4

Разбиение команды в середине идентификатора или строкового значения является синтаксической ошибкой.

### Форматные строки и форматные элементы

В первом аргументе метода `WriteLine` передается форматная строка, состоящая из фиксированного текста и форматных элементов. Фиксированный текст выводится методом `WriteLine` так же, как в программе на ил. 3.1. Каждый форматный элемент является «заместителем» для некоторого значения. Форматные элементы также могут содержать дополнительную информацию о форматировании. Они заключаются в фигурные скобки и содержат символы, которые указывают методу, какой аргумент здесь должен находиться и как он должен быть отформатирован. Например, форматный элемент `{0}` обозначает первый дополнительный аргумент (нумерация аргументов в C# начинается с 0), `{1}` обозначает второй аргумент, и т. д. Форматная строка в команде из строки 10 указывает, что метод `WriteLine` должен вывести два аргумента, причем после первого следует символ новой строки. Таким образом, в этом примере форматный элемент `{0}` заменяется на "Welcome to", а форматный элемент `{1}` — на "C# Programming!". Из приведенного результата видно, что выводимый текст разбивается на две строки. Так как фигурными

скобками в отформатированной строке обычно обозначаются заместители для подстановки текста, для вставки одиночной левой или правой фигурной скобки в отформатированную строку следует использовать удвоенную фигурную скобку, левую ({}) или правую (}) соответственно. Дополнительные возможности форматирования будут рассматриваться по мере того, как они будут задействованы в наших примерах.

## 3.6. Другое приложение C#: сложение целых чисел

Наше следующее приложение читает два целых числа, введенных пользователем с клавиатуры, вычисляет сумму введенных значений и выводит результат. Приложение должно запомнить числа, введенные пользователем, для последующего использования в приложении. Приложение хранит числа и другие данные в памяти компьютера и обращается к ним при помощи специальных конструкций, называемых *переменными* (variables). Эти концепции представлены в приложении на ил. 3.14.

```
1 // Ил. 3.14: Addition.cs
2 // Вывод суммы двух чисел, введенных с клавиатуры.
3 using System;
4
5 public class Addition
6 {
7     // Метод Main начинает выполнение приложения C#
8     public static void Main( string[] args )
9     {
10         int number1; // Объявление первого слагаемого
11         int number2; // Объявление второго слагаемого
12         int sum; // Объявление суммы number1 и number2
13
14         Console.Write( "Enter first integer: " ); // Запрос данных
15         // Получение первого числа от пользователя
16         number1 = Convert.ToInt32( Console.ReadLine() );
17
18         Console.Write( "Enter second integer: " ); // Запрос данных
19         // Получение второго числа от пользователя
20         number2 = Convert.ToInt32( Console.ReadLine() );
21
22         sum = number1 + number2; // Сложение чисел
23
24         Console.WriteLine( "Sum is {0}", sum ); // Вывод суммы
25     } // Конец Main
26 } // Конец класса Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

**Ил. 3.14.** Вывод суммы двух чисел, введенных с клавиатуры

## Строки 1–2

```
// Ил. 3.14: Addition.cs  
// Вывод суммы двух чисел, введенных с клавиатуры.
```

содержат комментарий с номером рисунка, именем файла и кратким описанием приложения.

## Класс Addition

В строке 5

```
public class Addition
```

начинается объявление класса `Addition`. Помните, что тело объявления каждого класса начинается с левой фигурной скобки (строка 6) и завершается правой фигурной скобкой (строка 26).

## Метод Main

Выполнение приложения начинается с точки входа `Main` (строки 8–25). Левая фигурная скобка (строка 9) отмечает начало тела `Main`, а соответствующая правая фигурная скобка (строка 25) отмечает конец тела `Main`. Метод `Main` снабжен одноуровневым отступом в теле класса `Addition`, а код тела `Main` имеет дополнительный отступ для удобства чтения.

## Объявление переменной `number1`

Строка 10

```
int number1; // Объявление первого слагаемого
```

содержит *объявление переменной*, в котором указывается имя (`number1`) и тип (`int`) переменной, используемой в приложении. Переменная представляет собой ячейку памяти, в которой хранится значение, используемое в приложении. Перед использованием переменные обычно объявляются с указанием имени и типа. Имя переменной используется приложением для обращения к значению в памяти — имя может быть любым действительным идентификатором (о требованиях к выбору идентификаторов рассказано в разделе 3.2). Тип переменной определяет, какого рода информация хранится в переменной и сколько памяти следует зарезервировать для хранения значения. Объявления, как и другие команды, завершаются символом «точка с запятой» (`;`).

## Тип `int`

В строке 10 переменная `number1` объявляется `int` — в таких переменных могут храниться целочисленные значения в диапазоне от `-2 147 483 648 (int.MinValue)` до `+2 147 483 647 (int.MaxValue)`. Вскоре мы рассмотрим типы `float`, `double` и `decimal` для представления вещественных чисел и тип `char` для представления символов. Вещественные числа имеют дробную часть (примеры: 3,4, 0,0 и `-11.19`). Переменные типа `float` и `double` используются для хранения приближенных значений вещественных чисел в памяти. Переменные типа `decimal` хранят вещественные числа с большой точностью (до 28–29 значащих цифр), поэтому переменные `decimal` часто

используются в финансовых вычислениях. Переменные типа `char` представляют отдельные символы: буквы (например, `A`), цифры (например, `7`), специальные символы (например, `*` или `%`) или служебные последовательности (например, символ новой строки `\n`). Такие типы, как `int`, `float`, `double`, `decimal` и `char`, часто называются *простыми типами*. Имена простых типов являются ключевыми словами и должны записываться строчными буквами.

### Объявление переменных `number2` и `sum`

Команды объявления переменных в строках 11–12

```
int number2; // Объявление второго слагаемого
int sum; // Объявление суммы number1 и number2
```

объявляют переменные с именами `number2` и `sum`, относящиеся к типу `int`.

Объявления переменных могут быть разбиты на несколько строк, при этом имена переменных отделяются друг от друга запятыми. Несколько однотипных переменных можно объявлять в одном или в разных объявлениях. Например, строки 10–12 также могут быть записаны следующим образом:

```
int number1; // Объявление первого слагаемого
int number2; // Объявление второго слагаемого
int sum; // Объявление суммы number1 и number2
```



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.7

Объявляйте каждую переменную в отдельной строке. Такой формат объявления позволяет легко добавить комментарий рядом с каждым объявлением.



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.8

Содержательные имена переменных способствуют самодокументированию кода (то есть код можно понять непосредственно из его чтения, без обращения к документации или пространных комментариев).



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.9

По распространенной схеме выбора имен идентификаторы переменных начинаются со строчной буквы, а каждое слово (кроме первого) начинается с прописной буквы.

### Вывод сообщения для пользователя

В строке 14

```
Console.Write( "Enter first integer: " ); // Запрос данных
```

метод `Console.Write` выводит сообщение `"Enter first integer:"`, предлагающее пользователю выполнить конкретное действие.

### Чтение значения `number1`

Строка 16

```
number1 = Convert.ToInt32( Console.ReadLine() );
```

работает в два этапа. Сначала она вызывает метод `ReadLine` класса `Console`; этот метод ожидает, пока пользователь введет с клавиатуры последовательность символов и нажмет клавишу `Enter`. Как упоминалось ранее, некоторые методы выполняют операцию, а затем возвращают ее результат. В данном случае `ReadLine` возвращает текст, введенный пользователем. Полученная строка передается в аргументе метода `ToInt32` класса `Convert`, который преобразует последовательность символов в данные типа `int`. В нашем случае метод `ToInt32` возвращает представление данных, введенных пользователем, в формате `int`.

### Возможные ошибки при вводе

В принципе, пользователь может ввести любую последовательность символов. Метод `ReadLine` примет ее и передаст методу `ToInt32`. Метод полагает, что строка содержит действительное целое число. Если пользователь введет что-то другое, в нашем приложении произойдет логическая ошибка с выдачей исключения, и приложение аварийно завершится. В C# поддерживается технология *обработки исключений*, которая повышает устойчивость приложений, позволяя им обработать исключение и продолжить выполнение. Обработка исключений рассматривается в разделе 8.4, потом мы снова вернемся к ней в главе 10. Более подробно обработка исключений рассматривается в главе 13.

### Присваивание значения переменной

В строке 16 результат вызова метода `ToInt32` (значение `int`) помещается в переменную `number1` с использованием оператора присваивания `=`. Команду можно прочитать следующим образом: «Переменной `number1` присваивается значение, возвращаемое `Convert.ToInt32`». Оператор `=` относится к категории *бинарных* операторов, потому что он работает с двумя исходными значениями. Эти значения называются *операндами* — в данном случае это переменная `number1` и результат вызова метода `Convert.ToInt32`. Все вычисления в правой части команды присваивания всегда выполняются *до* присваивания.



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.10

Ставьте пробелы по обе стороны оператора присваивания, чтобы код лучше читался.

### Вывод сообщения для пользователя и чтение значения `number2`

Строка 18

```
Console.Write( "Enter second integer: " ); // Запрос данных
```

предлагает пользователю ввести второе целое число.

Строка 20

```
number2 = Convert.ToInt32( Console.ReadLine() );
```

читает второе число и присваивает его переменной `number2`.

### Суммирование `number1` и `number2`

Строка 22

```
sum = number1 + number2; // Сложение чисел
```



вычисляет сумму чисел `number1` и `number2` и присваивает результат переменной `sum` оператором присваивания `=`. Большинство вычислений выполняется в командах присваивания. Когда компилятор встречает конструкцию `number1+number2`, хранящиеся в переменных значения используются в вычислениях. Оператор сложения является бинарным оператором — он имеет два операнда, `number1` и `number2`. Части команд, содержащие вычисления, называются *выражениями*. Строго говоря, выражением является любая часть команды, с которой связано некоторое значение. Например, значением выражения `number1+number2` является сумма двух чисел. Аналогичным образом значением выражения `Console.ReadLine()` является последовательность символов, введенная пользователем.

### Вывод суммы

После того как суммирование будет выполнено, в строке 24

```
Console.WriteLine( "Sum is {0}", sum ); // Вывод суммы
```

метод `Console.WriteLine` используется для вывода суммы. Форматный элемент `{0}` является заместителем для первого аргумента в форматной строке. Остальные символы форматной строки, за исключением форматного элемента `{0}`, представляют собой фиксированный текст. Таким образом, метод `WriteLine` выводит сообщение "Sum is ", затем значение суммы (в позиции форматного элемента `{0}`) и символ новой строки.

### Вычисления в командах вывода

Вычисления также могут выполняться в командах вывода. Например, команды в строках 22 и 24 можно объединить в команду

```
Console.WriteLine( "Sum is {0}", ( number1 + number2 ) );
```

Круглые скобки, в которые заключено выражение `number1+number2`, не обязательны — они всего лишь наглядно показывают, что значение выражения `number1+number2` выводится в позиции форматного элемента `{0}`.

## 3.7. Работа с памятью

Имена переменных (такие, как `number1`, `number2` и `sum`) в действительности представляют ячейки памяти компьютера. Каждая переменная обладает именем, типом, размером (который определяется типом) и значением.

В приложении на ил. 3.14 при выполнении следующей команды (строка 16):

```
number1 = Convert.ToInt32( Console.ReadLine() );
```

число, введенное пользователем, помещается в ячейку памяти, которую компилятор связал с именем `number1`. Допустим, пользователь ввел число 45. Компьютер помещает это целочисленное значение в ячейку `number1` (ил. 3.15). Каждый раз, когда в ячейку памяти записывается новое значение, ее предыдущее содержимое теряется.

number1	45
---------	----

**Ил. 3.15.** Ячейка памяти с именем и значением переменной number1

Допустим, при выполнении команды (строка 20)

```
number2 = Convert.ToInt32( Console.ReadLine() );
```

пользователь вводит 72. Компьютер помещает это число в ячейку number2. Теперь память выглядит так, как показано на ил. 3.16.

number1	45
number2	72

**Ил. 3.16.** Состояние памяти после сохранения значений number1 и number2

Получив значения number1 и number2, приложение суммирует их и помещает результат в переменную sum. Команда (строка 22)

```
sum = number1 + number2; // Сложение чисел
```

выполняет сложение и записывает результат на место предыдущего значения sum. Структура памяти после суммирования показана на ил. 3.17. Значения number1 и number2 остаются там же, где они находились до вычисления sum. Эти значения используются, но не уничтожаются при вычислениях — чтение из ячеек памяти является *неразрушающей* операцией.

number1	45
number2	72
sum	117

**Ил. 3.17.** Состояние памяти после вычисления и сохранения суммы number1 и number2

## 3.8. Арифметика

В большинстве приложений выполняются какие-либо арифметические вычисления. На ил. 3.18 приведена сводка арифметических операторов. Обратите внимание на специальные символы, не встречающиеся в традиционной алгебре: звездочка (\*) обозначает умножение, а знак процента (%) представляет оператор вычисления остатка (см. далее). Все арифметические операторы на ил. 3.18 являются

бинарными — например, выражение  $f + 7$  состоит из бинарного оператора  $+$  и двух операндов,  $f$  и  $7$ .

Оператор C#	Арифметический оператор	Алгебраическое выражение	Выражение C#
Сложение	$+$	$f + 7$	$f + 7$
Вычитание	$-$	$p - c$	$p - c$
Умножение	$*$	$b \cdot m$	$b * m$
Деление	$/$	$x / y$ или $x \div y$	$x / y$
Вычисление остатка	$\%$	$r \bmod s$	$r \% s$

**Ил. 3.18.** Арифметические операторы

Если оба операнда оператора деления ( $/$ ) являются целыми числами, то выполняется целочисленное деление, а результат является целым числом — например, результат выражения  $7 / 4$  равен 1, а результат выражения  $17 / 5$  равен 3. Дробная часть при целочисленном делении просто отбрасывается. C# поддерживает оператор вычисления остатка  $\%$ , который возвращает остаток от деления. Результат выражения  $x\%y$  равен остатку от деления  $x$  на  $y$ . Таким образом, результат  $7 \% 4$  равен 3, а результат  $17 \% 5$  равен 2. Этот оператор чаще всего используется с целочисленными операндами, но он также может использоваться с данными `float`, `double` и `decimal`. В следующих главах мы рассмотрим несколько интересных применений оператора вычисления остатка — например, для проверки кратности.

### Круглые скобки и группировка подвыражений

Круглые скобки используются для группировки частей выражений C# по аналогии с алгебраическими выражениями. Например, для умножения  $a$  на величину  $b+c$  используется запись

```
a * ( b + c )
```

Если выражение содержит вложенные круглые скобки:

```
( ( a + b ) * c )
```

то вычисление начинается с круглых скобок с максимальным уровнем вложенности.

### Приоритеты операторов

C# применяет операторы в арифметических выражениях в строгой последовательности, определяемой правилами приоритета операторов, которые обычно совпадают с правилами, применяемыми в алгебре (ил. 3.19)<sup>1</sup>.

<sup>1</sup> Сейчас мы рассмотрим несколько простых примеров, поясняющих порядок вычисления выражений. В более сложных выражениях могут встречаться некоторые нетривиальные ситуации с порядком вычисления. Дополнительную информацию можно найти в сообщениях в блоге Эрика Линнепта (Eric Lippert): [blogs.msdn.com/ericlippert/archive/2008/05/23/precedence-vs-associativity-vs-order.aspx](http://blogs.msdn.com/ericlippert/archive/2008/05/23/precedence-vs-associativity-vs-order.aspx) и [blogs.msdn.com/oldnewthing/archive/2007/08/14/4374222.aspx](http://blogs.msdn.com/oldnewthing/archive/2007/08/14/4374222.aspx).

Операторы	Операции	Порядок вычисления
Выполняются сначала		
*	Умножение	Если операторов из этой категории несколько, они выполняются слева направо
/	Деление	
%	Вычисление остатка	
Выполняются потом		
+	Сложение	Если операторов из этой категории несколько, они выполняются слева направо
-	Вычитание	

**Ил. 3.19.** Приоритеты арифметических операторов

На ил. 3.19 приведена сводка правил приоритета этих операторов. Таблица будет дополняться по мере знакомства с дополнительными операторами.

**Примеры алгебраических выражений и выражений C#**

Рассмотрим несколько примеров выражений в свете правил приоритета операторов. В каждом примере приводится алгебраическое выражение и его эквивалент в C#. Следующее выражение вычисляет среднее арифметическое пяти чисел:

Алгебраическое выражение:

$$m = \frac{a + b + c + d + e}{5}$$

Выражение C#:

$$m = (a + b + c + d + e) / 5;$$

Круглые скобки необходимы из-за того, что деление обладает более высоким приоритетом, чем сложение. На 5 необходимо разделить всю сумму (a + b + c + d + e). Без скобок получится выражение a + b + c + d + e / 5, которое вычисляется по формуле

$$a + b + c + d + \frac{e}{5}.$$

Следующий пример представляет собой формулу прямой линии:

Алгебраическое выражение:

$$y = mx + b$$

Выражение C#:

$$y = m * x + b;$$

В этом случае круглые скобки не обязательны. Сначала применяется оператор умножения, потому что эта операция обладает более высоким приоритетом, чем операция сложения. Присваивание выполняется в последнюю очередь, потому что оно имеет более низкий приоритет, чем умножение и сложение.

В следующем примере выполняются операции вычисления остатка (%), умножения, деления, сложения и вычитания:

Алгебраическое выражение:

$$z = pr\%q + w/x - y;$$

Выражение C#:

$$z = p * r \% q + w / x - y;$$

6

1

2

4

3

5

Цифры в кружках обозначают порядок применения операторов в C#. Сначала слева направо выполняются операции умножения, вычисления остатка и деления, потому что они обладают более высоким приоритетом, чем сложение и вычитание. Затем выполняются операции сложения и вычитания. Они также применяются слева направо.

### Вычисление квадратного многочлена

Чтобы вы лучше поняли, как работают правила приоритета операторов, рассмотрим процесс вычисления квадратного многочлена ( $y = ax^2 + bx + c$ ):



Цифры в кружках обозначают порядок применения операторов в C#. Сначала слева направо выполняются операции умножения, потому что они обладают более высоким приоритетом, чем сложение и вычитание. Затем выполняются операции сложения и вычитания. В C# не существует арифметического оператора возведения в степень, так что операция  $x^2$  представляется в виде  $x*x$ . В разделе 6.4 представлен альтернативный способ реализации возведения в степень в C#.

Допустим,  $a$ ,  $b$ ,  $c$  и  $x$  в приведенном квадратном многочлене инициализируются следующим образом:  $a=2$ ,  $b=3$ ,  $c=7$  и  $x=5$ . Порядок применения операторов представлен на ил. 3.20.



**Ил. 3.20.** Порядок вычисления квадратного многочлена

### Избыточные круглые скобки

Как и в алгебре, в выражение можно добавить круглые скобки, которые не являются необходимыми, но делают выражение более понятным. Например, в приведенной команде присваивания можно выделить слагаемые следующим образом:

```
y = ( a * x * x ) + ( b * x ) + c;
```

## 3.9. Принятие решений: операторы сравнения и проверки равенства

*Условием* называется выражение, которое может быть истинным или ложным. В этом разделе представлена простая версия команды C# `if`, которая позволяет приложению принять решение в зависимости от условия. Например, условие «оценка больше либо равна 60» определяет, набрал ли студент необходимое количество баллов. Если условие в команде `if` истинно, то выполняется тело команды `if`. Если условие ложно, то тело не выполняется. Вскоре мы рассмотрим конкретный пример.

Условия в командах `if` могут строиться с использованием операторов проверки равенства (`==` и `!=`) и операторов сравнения (`>`, `<`, `>=` и `<=`), перечисленных на ил. 3.21. Операторы проверки равенства (`==` и `!=`) обладают одинаковым приоритетом; операторы сравнения (`>`, `<`, `>=` and `<=`) тоже имеют одинаковый приоритет, но более высокий, чем у операторов проверки равенства. Операторы применяются слева направо.



### ТИПИЧНАЯ ОШИБКА 3.5

Не путайте оператор проверки равенства `==` с оператором присваивания `=`; это может привести к логическим или синтаксическим ошибкам.

Стандартные алгебраические операторы сравнения и проверки равенства	Операторы сравнения и проверки равенства C#	Пример условия C#	Смысл условия C#
Операторы сравнения			
<code>&gt;</code>	<code>&gt;</code>	<code>x &gt; y</code>	<code>x</code> больше <code>y</code>
<code>&lt;</code>	<code>&lt;</code>	<code>x &lt; y</code>	<code>I</code> меньше <code>y</code>
<code>≥</code>	<code>&gt;=</code>	<code>x &gt;= y</code>	<code>x</code> больше либо равно <code>I</code>
<code>≤</code>	<code>&lt;=</code>	<code>x &lt;= y</code>	<code>x</code> меньше либо равно <code>y</code>
Операторы проверки равенства			
<code>=</code>	<code>==</code>	<code>x == y</code>	<code>x</code> равно <code>y</code>
<code>≠</code>	<code>!=</code>	<code>x != y</code>	<code>x</code> не равно <code>y</code>

**Ил. 3.21.** Операторы сравнения и проверки равенства

## Команда if

Приложение на ил. 3.22 использует шесть команд if для сравнения двух целых чисел, введенных пользователем. Если условие в какой-либо из этих команд оказывается истинным, то выполняется команда присваивания в теле этой команды if. Приложение использует класс Console для вывода запроса и получения двух строк текста от пользователя, преобразует введенный текст в целые числа методом ToInt32 класса Convert, после чего сохраняет их в переменных number1 и number2. Далее класс сравнивает числа и выводит результаты истинных сравнений.

```
1 // Ил. 3.22: Comparison.cs
2 // Сравнение целых чисел с использованием команд if, операторов
3 // проверки равенства и сравнения.
4 using System;
5
6 public class Comparison
7 {
8     // Метод Main начинает выполнение приложения C#
9     public static void Main( string[] args )
10    {
11        int number1; // Объявление первого сравниваемого числа
12        int number2; // Объявление второго сравниваемого числа
13
14        // Запрос и ввод первого числа
15        Console.Write( "Enter first integer: " );
16        number1 = Convert.ToInt32( Console.ReadLine() );
17
18        // Запрос и ввод второго числа
19        Console.Write( "Enter second integer: " );
20        number2 = Convert.ToInt32( Console.ReadLine() );
21
22        if ( number1 == number2 )
23            Console.WriteLine( "{0} == {1}", number1, number2 );
24
25        if ( number1 != number2 )
26            Console.WriteLine( "{0} != {1}", number1, number2 );
27
28        if ( number1 < number2 )
29            Console.WriteLine( "{0} < {1}", number1, number2 );
30
31        if ( number1 > number2 )
32            Console.WriteLine( "{0} > {1}", number1, number2 );
33
34        if ( number1 <= number2 )
35            Console.WriteLine( "{0} <= {1}", number1, number2 );
36
37        if ( number1 >= number2 )
38            Console.WriteLine( "{0} >= {1}", number1, number2 );
39    } // Конец Main
40 } // Конец класса Comparison
```

```
Enter first integer: 42
Enter second integer: 42
42 == 42
42 <= 42
42 >= 42
```

**Ил. 3.22.** Сравнение целых чисел с использованием команд if, операторов проверки равенства и сравнения (продолжение ↗)

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

**Ил. 3.22.** Сравнение целых чисел с использованием команд `if`, операторов проверки равенства и сравнения (окончание)

### Класс `Comparison`

Объявление класса `Comparison` начинается в строке 6

```
public class Comparison
```

Метод `Main` класса (строки 9–39) начинает выполнение приложения.

### Объявления переменных

В строках 11–12

```
int number1; // Объявление первого сравниваемого числа
int number2; // Объявление второго сравниваемого числа
```

объявляются переменные типа `int` для хранения значений, введенных пользователем.

### Получение данных от пользователя

В строках 14–16

```
// Запрос и ввод первого числа
Console.Write( "Enter first integer: " );
number1 = Convert.ToInt32( Console.ReadLine() );
```

приложение запрашивает и получает от пользователя первое число. Введенное значение сохраняется в переменной `number1`. Строки 18–20

```
// Запрос и ввод второго числа
Console.Write( "Enter second integer: " );
number2 = Convert.ToInt32( Console.ReadLine() );
```

делают то же самое, но входное значение сохраняется в переменной `number2`.

### Сравнение чисел

Строки 22–23

```
if ( number1 == number2 )
    Console.WriteLine( "{0} == {1}", number1, number2 );
```

проверяют, равны ли значения переменных `number1` и `number2`. Команда `if` всегда начинается с ключевого слова `if`, за которым следует условие в круглых скобках.



Команда `if` предполагает, что тело состоит из одной команды. Приведенный в листинге отступ тела не обязателен, но он упрощает чтение кода: сразу видно, что команда в строке 23 является частью команды `if`, начинающейся в строке 22. Строка 23 выполняется только в том случае, если числа, хранящиеся в переменных `number1` и `number2`, равны (то есть условия истинны). Команды `if` в строках 25–26, 28–29, 31–32, 34–35 и 37–38 сравнивают `number1` и `number2` с применением операторов `!=`, `<`, `>`, `<=` и `>=` соответственно. Если условие любой команды `if` истинно, выполняется соответствующее тело `if`.



### ТИПИЧНАЯ ОШИБКА 3.6

Отсутствие левой и/или правой круглой скобки в команде `if` является синтаксической ошибкой — обе скобки обязательны.



### ТИПИЧНАЯ ОШИБКА 3.7

Перестановка знаков в операторах `!=`, `>=` и `<=` (соответственно `=!`, `=>` и `=<`) приводит к синтаксическим и логическим ошибкам.



### ТИПИЧНАЯ ОШИБКА 3.8

Пробелы между знаками в операторах `==`, `!=`, `>=` и `<=` (соответственно `=`, `!`, `>`, `=` и `<`, `=`) являются синтаксической ошибкой.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.11

Снабдите тело команды `if` отступом, чтобы оно визуально отделялось от самой команды.

Обратите внимание на отсутствие символа «;» в конце первой строки каждой команды `if`. Наличие этого символа привело бы к логической ошибке во время выполнения. Например, фрагмент

```
if ( number1 == number2 ); // Логическая ошибка
    Console.WriteLine( "{0} == {1}", number1, number2 );
```

будет интерпретирован C# в следующем виде:

```
if ( number1 == number2 )
    ; // пустая команда
    Console.WriteLine( "{0} == {1}", number1, number2 );
```

Точка с запятой, стоящая в строке сама по себе, называется *пустой командой*. При выполнении пустой команды приложение не выполняет никаких действий. Далее работа приложения будет продолжена командой вывода, которая выполняется всегда независимо от истинности или ложности условия, потому что команда вывода не является частью команды `if`.



### ТИПИЧНАЯ ОШИБКА 3.9

Символ «;», следующий непосредственно за правой круглой скобкой условия в команде `if`, обычно приводит к логической ошибке.

## Пропуски

Обратите внимание на пропуски на ил. 3.22. Как говорилось ранее, пропуски (табуляции, новые строки и пробелы) обычно игнорируются компилятором. Это позволяет разбивать команды на несколько строк и расставлять отступы так, как вы считаете нужным, без изменения смысла кода. Разбивать идентификаторы, строковые данные и состоящие из нескольких символов операторы (например, `>=`) запрещается. В идеале команды должны быть короткими, но это не всегда возможно.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.12

Не размещайте в одной строке приложения более одной команды, чтобы не усложнять чтение кода.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.13

Длинные команды можно разбить на несколько строк. Осмысленно выбирайте точки разбиения — например, после запятой в списке аргументов или после оператора в длинном выражении. Если команда разбивается на две и более строки, включите отступы во все последующие строки до конца приложения.

## Приоритеты и порядок выполнения операторов

На ил. 3.23 приведена сводка приоритетов операторов, описанных в этой главе. Операторы перечисляются в порядке убывания приоритетов. Все операторы, кроме оператора присваивания `=`, применяются слева направо. Например, выражение `x + y + z` вычисляется так, как если бы оно было записано в виде `(x + y) + z`. Оператор `=` применяется справа налево, то есть выражение вида `x = y = 0` вычисляется в порядке `x = (y = 0)`, то есть, как вы вскоре увидите, сначала значение 0 присваивается переменной `y`, а затем результат этого присваивания (0) присваивается переменной `x`.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 3.14

При написании выражений с множеством операторов следует руководствоваться списком приоритетов. Убедитесь в том, что операции в выражении выполняются в том порядке, который вам нужен. Если вы сомневаетесь в порядке выполнения операций в сложном выражении, используйте круглые скобки для принудительного соблюдения нужного порядка, как это делается в алгебраических выражениях. Помните, что некоторые операторы (такие, как оператор присваивания `=`) выполняются справа налево, а не слева направо.

Операторы	Порядок применения	Тип
<code>* / %</code>	Слева направо	Умножение/деление
<code>+ -</code>	Слева направо	Сложение/вычитание
<code>&lt; &lt;= &gt; &gt;=</code>	Слева направо	Сравнение
<code>== !=</code>	Слева направо	Проверка равенства
<code>=</code>	Справа налево	Присваивание

## 3.10. Итоги

В этой главе были представлены многие важные особенности C#. Сначала вы научились выводить данные на экран в окне командной строки с использованием классов `Write` и `WriteLine` класса `Console`. Затем было показано, как применять форматные строки и форматные элементы для форматирования выходных строк. Также вы узнали, как вводить данные с клавиатуры с использованием метода `ReadLine` класса `Console`. Далее рассматривалось выполнение вычислений с использованием арифметических операторов C#. Глава завершается описанием принятия решений с использованием команды `if`, операторов сравнения и проверки равенства. Приведенные в главе примеры приложений демонстрируют многие основополагающие концепции программирования. Как будет показано в главе 4, в типичном приложении C# метод `Main` обычно содержит всего несколько строк кода — эти команды создают объекты, выполняющие основную работу приложения. Вы научитесь создавать реализации собственных классов и использовать объекты этих классов в приложениях.

# 4 Классы, объекты и методы

## 4.1. Введение

Эта глава начинается с объяснения концепции классов на реальном примере. Затем будут рассмотрены пять приложений, которые демонстрируют принципы создания и использования классов в приложениях. Первые четыре приложения открывают учебный проект по разработке класса для хранения информации об оценках. Последний пример познакомит читателя с типом `decimal` на примере класса, предназначенного для представления баланса банковского счета.

## 4.2. Классы, объекты, методы, свойства и переменные экземпляров

Начнем с простой аналогии, которая поможет вам понять суть классов. Допустим, вы ведете машину и хотите увеличить скорость, нажав педаль газа. Что должно произойти перед тем, как вы сможете это сделать? Для начала кто-то должен спроектировать вашу машину. Как правило, проектирование начинается с технических чертежей, в которых присутствует педаль газа. Педаль скрывает от водителя сложные механизмы, которые заставляют машину двигаться быстрее, — подобно тому, как педаль тормоза скрывает механизмы, замедляющие движение, а руль скрывает механизмы поворота. Все это позволяет легко управлять машиной человеку, не имеющему представления о работе двигателя, тормозов и поворотных механизмов.

К сожалению, вести технический чертеж невозможно. Прежде чем вы сможете сесть за руль, машину необходимо построить по техническому чертежу. В готовой машине будет присутствовать педаль, ускоряющая ее движение, но даже этого недостаточно — машина не будет ускоряться сама по себе, на педаль газа должен нажать водитель.

### Методы

Пример с машиной поможет нам представить некоторые ключевые концепции программирования. Все операции в приложениях выполняются методами. По сути

*метод* описывает механизмы, которые непосредственно реализуют нужные операции. Метод скрывает от пользователя сложные низкоуровневые подробности — подобно тому, как педаль газа скрывает от водителя сложные механизмы, которые заставляют машину двигаться быстрее.

## **Классы**

В языке C# все начинается с создания структурных элементов приложения, называемых *классами*. Классы (среди прочего) содержат методы подобно тому, как чертежи машины содержат (среди прочего) техническое описание педали газа. В класс включаются методы (один или несколько), предназначенные для выполнения операций класса. Например, класс, представляющий банковский счет, может содержать один метод для внесения денег на счет, другой метод для снятия денег со счета и третий метод для запроса текущего баланса.

## **Объекты**

Класс — это всего лишь описание, и непосредственно использовать его невозможно (подобно тому, как невозможно вести технический чертеж машины). Прежде чем приложение сможет выполнять операции, описанные классом, необходимо сначала создать объект этого класса. Это одна из причин, по которым C# называется объектно-ориентированным языком программирования.

## **Вызовы методов**

Когда вы управляете машиной, нажатие педали газа отправляет машине сообщение, приказывающее выполнить операцию — ускорить движение. Аналогичным образом отправляются сообщения объекту; каждое сообщение, называемое *вызовом метода*, приказывает вызванному методу объекта выполнить свою операцию.

## **Атрибуты**

Кроме поддерживаемых операций, машина обладает многочисленными атрибутами: цвет, количество дверей, количество бензина в баке, текущая скорость и пробег. Эти атрибуты представляются в технических чертежах машины наряду с ее основными функциями. Во время ведения машины эти атрибуты всегда связываются с машиной. Каждая машина поддерживает собственный набор атрибутов. Например, каждая машина знает, сколько бензина находится в ее топливном баке, но ничего не знает об уровне бензина в других машинах. Аналогичным образом с каждым объектом связывается набор атрибутов, который сопровождает объект в процессе его использования приложением. Атрибуты определяются как часть класса объекта. Например, объект банковского счета может обладать атрибутом, представляющим текущий баланс. Каждый объект знает баланс того счета, который он представляет, но не балансы других счетов в банке. Атрибуты определяются в виде *переменных экземпляров* класса.

## **Свойства, get- и set-методы**

Следует учитывать, что атрибуты не всегда доступны напрямую. В конце концов, никакой банк не захочет пускать клиентов в хранилище, чтобы те могли пересчитать

деньги на своем счету. Вместо этого клиент обращается к кассиру или получает информацию на своей странице в Интернете. Этот принцип распространяется и на объекты: чтобы использовать переменные экземпляра, не обязательно иметь прямой доступ к ним — можно воспользоваться *свойствами* объекта. Свойства содержат get-методы для чтения значений переменных и set-методы для присваивания значений.

## 4.3. Объявление класса с методом и создание экземпляра класса

Начнем с примера, состоящего из классов `GradeBook` (ил. 4.1) и `GradeBookTest` (ил. 4.2). Класс `GradeBook` (объявленный в файле `GradeBook.cs`) выводит на экран приветствие, а класс `GradeBookTest` (объявленный в файле `GradeBookTest.cs`) используется для создания и выполнения операций с объектом класса `GradeBook`. Как принято, мы объявляем классы `GradeBook` и `GradeBookTest` в разных файлах, чтобы имя каждого файла соответствовало имени содержащегося в нем класса.

Выполните команду `File ► New Project...`, чтобы открыть диалоговое окно `New Project`, и выберите шаблон `GradeBook Console Application`. Переименуйте файл `Program.cs` в `GradeBook.cs`. Удалите весь код, сгенерированный IDE, и замените его кодом на ил. 4.1.

```
1 // Ил. 4.1: GradeBook.cs
2 // Объявление класса с одним методом
3 using System;
4
5 public class GradeBook
6 {
7     // Вывод приветствия для пользователя GradeBook
8     public void DisplayMessage()
9     {
10         Console.WriteLine( "Welcome to the Grade Book!" );
11     } // Конец метода DisplayMessage
12 } // Конец класса GradeBook
```

**Ил. 4.1.** Объявление класса с одним методом

### Класс `GradeBook`

Объявление класса `GradeBook` (см. ил. 4.1) содержит метод `DisplayMessage` (строки 8–11) для вывода сообщения на экран. Сообщение выводится в строке 10. Вспомните, что класс ранее сравнивался с техническим чертежом — чтобы вывести сообщение, необходимо создать объект этого класса и вызвать его метод; это делается в листинге на ил. 4.2.

Объявление класса начинается в строке 5. Ключевое слово `public` является модификатором доступа. *Модификаторы доступа* определяют доступность свойств и методов объекта для других методов приложения. Пока мы будем просто объявлять все классы *открытыми*, то есть общедоступными. Каждое объявление класса

содержит ключевое слово `class`, за которым следует имя класса. Тело каждого класса заключается в пару фигурных скобок (`{` и `}`), как в строках 6 и 12 класса `GradeBook`.

### Объявление метода `DisplayMessage`

В главе 3 каждый объявляемый класс содержит единственный метод с именем `Main`. Класс `GradeBook` также содержит один метод — `DisplayMessage` (строки 8–11). Напомним, что специальный метод `Main` всегда вызывается автоматически при запуске приложения. Большинство методов автоматически не вызывается. Как вы вскоре увидите, чтобы метод `DisplayMessage` выполнил свою операцию, его необходимо явно вызвать в программе.

Объявление метода начинается с ключевого слова `public`, которое указывает, что метод является открытым — то есть может вызываться за пределами тела объявления класса методами других классов. Ключевое слово `void` (тип возвращаемого значения) означает, что метод после завершения не возвращает никакой информации тому методу, из которого он был вызван. Если в методе указан тип возвращаемого значения, отличный от `void`, то после вызова метода и выполнения операции он возвращает результат заданного типа. Например, когда вы подходите к банкомату и запрашиваете баланс своего счета, банкомат возвращает числовое значение. Если метод `Square` возвращает квадрат своего аргумента, то в команде

```
int result = Square( 2 );
```

вызов `Square` должен вернуть значение 4, которое будет присвоено переменной `result`.

Мы уже использовали методы, возвращающие информацию, — например, в главе 3 метод `ReadLine` класса `Console` применялся для получения строки, введенной пользователем с клавиатуры. Метод `ReadLine` возвращает введенное значение для последующего использования в приложении.

### Имя метода

За типом возвращаемого значения указывается имя метода `DisplayMessage` (строка 8). Как правило, именами методов являются глаголы или глагольные конструкции, а именами классов — существительные. По общепринятой схеме имена методов начинаются с прописной буквы, а все последующие слова в имени также начинаются с прописных букв. Круглые скобки после имени метода указывают, что определяется именно метод, а не что-то другое. Пустая пара круглых скобок (строка 8) означает, что метод не требует дополнительной информации для выполнения своей операции. Строка 8 обычно называется *заголовком метода*. Тело каждого метода заключается в фигурные скобки, как показано в строках 9 и 11.

### Тело метода

Тело метода содержит команды, выполняющие операцию метода. В нашем случае метод содержит всего одну команду (строка 10), которая выводит в консольном окне приветствие `"Welcome to the Grade Book!"` и символ новой строки. После выполнения команды операция метода считается выполненной.

## Использование класса `GradeBook`

Следующий шаг — использование класса `GradeBook` в приложении. Как вы узнали из главы 3, выполнение каждого приложения начинается с метода `Main`. Класс `GradeBook` не может быть основным классом приложения, потому что в нем нет метода с именем `Main`. В главе 3 такой проблемы не было, потому что каждый объявляемый класс содержал метод `Main`. Следовательно, нам придется либо объявить отдельный класс, содержащий метод `Main`, либо добавить метод `Main` в класс `GradeBook`.

Чтобы подготовить вас к более масштабным приложениям, которые встретятся вам в этой книге и в реальной работе, для тестирования классов этой главы будет использоваться отдельный класс (`GradeBookTest` в данном случае), содержащий метод `Main`.

## Добавление класса в проект `Visual C#`

Для каждого примера этой главы мы добавим в консольное приложение новый класс. Щелкните правой кнопкой мыши на имени проекта в окне `Solution Explorer` и выберите в контекстном меню команду `Add ► New Item....` В открывшемся диалоговом окне `Add New Item` выберите вариант `Code File`, введите имя нового файла (`GradeBookTest.cs`) и щелкните на кнопке `Add`. В проект добавляется новый пустой файл. Включите в него код с ил. 4.2.

```
1 // ил. 4.2: GradeBookTest.cs
2 // Создание объекта GradeBook и вызов его метода DisplayMessage.
3 public class GradeBookTest
4 {
5     // Метод Main начинает выполнение программы
6     public static void Main( string[] args )
7     {
8         // Создание объекта GradeBook и присваивание его myGradeBook
9         GradeBook myGradeBook = new GradeBook();
10
11         // Вызов метода DisplayMessage объекта myGradeBook
12         myGradeBook.DisplayMessage();
13     } // Конец Main
14 } // Конец класса GradeBookTest
```

```
Welcome to the Grade Book!
```

**Ил. 4.2.** Создание объекта `GradeBook` и вызов его метода `DisplayMessage`

## Класс `GradeBookTest`

Объявление класса `GradeBookTest` (см. ил. 4.2) содержит метод `Main`, управляющий работой приложения. Любой класс, содержащий метод `Main` (строка 6), может использоваться для выполнения приложения. Объявление класса начинается в строке 3 и заканчивается в строке 14. Класс содержит только метод `Main`; это типично для многих классов, которые просто начинают выполнение приложения.

## Метод `Main`

В строках 6–13 объявляется метод `Main`. Чтобы метод `Main` мог начать выполнение приложения, в объявлении должно присутствовать ключевое слово `static` (строка 6),



которое указывает, что метод `Main` является статическим. Статические методы отличаются тем, что они могут вызываться без предварительного создания объекта класса (в нашем случае `GradeBookTest`), в котором объявлен метод. Статические методы более подробно рассматриваются в главе 7.

### Создание объекта `GradeBook`

В нашем приложении требуется вызвать метод `DisplayMessage` класса `GradeBook` для вывода приветствия в консольном окне. В общем случае вы не сможете вызвать метод, принадлежащий другому классу, до создания объекта этого класса (строка 9). Мы начинаем с объявления переменной `myGradeBook`. Переменная объявляется с типом `GradeBook` — класс, объявленный в листинге на ил. 4.1. Каждый новый класс, созданный вами, становится новым типом в C#, который может использоваться для объявления переменных и создания объектов. Новые классы становятся доступными для всех классов того же проекта.

Переменная `myGradeBook` (строка 9) инициализируется результатом выражения создания объекта `new GradeBook()`. Оператор `new` создает новый объект класса, указанного справа от ключевого слова (то есть `GradeBook`). Круглые скобки справа от `GradeBook` обязательны. Как будет показано в разделе 4.10, круглые скобки в сочетании с именем класса представляют вызов конструктора — метода, используемого только в момент создания объекта для инициализации его данных. Далее вы увидите, что в круглые скобки можно заключить данные, определяющие исходные значения данных объекта. Пока мы оставим круглые скобки пустыми.

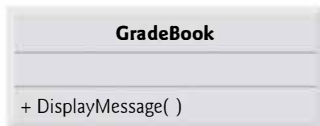
### Вызов метода `DisplayMessage` объекта `GradeBook`

Теперь переменная `myGradeBook` может использоваться для вызова метода `DisplayMessage`. В строке 12 вызывается метод `DisplayMessage` (строки 8–11 на ил. 4.1): вызов состоит из имени переменной `myGradeBook`, оператора «точка» (`.`), имени метода `DisplayMessage` и пустой пары круглых скобок. Этот вызов метода отличается от вызовов методов из главы 3, выводивших информацию в консольном окне, — тогда во всех случаях передавались аргументы с выводимыми данными. В начале строки 12 (ил. 4.2) префикс «`myGradeBook.`» указывает, что метод `Main` должен использовать объект `GradeBook`, созданный в строке 9. Пустые круглые скобки в строке 8 на ил. 4.1 означают, что методу `DisplayMessage` для выполнения его операции не требуется никакой дополнительной информации. По этой причине при вызове метода (строка 12 на ил. 4.2) за именем метода вставляется пустая пара круглых скобок, которая означает, что методу `DisplayMessage` не передаются аргументы. Когда метод `DisplayMessage` завершит свою операцию, метод `Main` продолжает выполняться в строке 13. Здесь метод `Main` заканчивается, а выполнение приложения завершается.

### Диаграмма классов UML для класса `GradeBook`

На ил. 4.3 представлена диаграмма классов UML для класса `GradeBook` на ил. 4.1. Графический язык UML используется программистами для стандартизированного представления объектно-ориентированных систем. В UML каждый класс на

диаграмме классов представляется прямоугольником, состоящим из трех отделений. Верхнее отделение содержит имя класса, выровненное по центру и выводимое жирным шрифтом. Среднее отделение содержит атрибуты класса, соответствующие переменным экземпляров и свойствам C#. На ил. 4.3 среднее отделение пусто, потому что версия класса `GradeBook` на ил. 4.1 не имеет атрибутов. В нижнем отделении перечисляются операции класса, соответствующие методам в C#. В UML за именем операции ставится пара круглых скобок. Класс `GradeBook` содержит всего один метод `DisplayMessage`, поэтому в нижнем отделении ил. 4.3 указана одна операция с этим именем. Методу `DisplayMessage` для выполнения его операции не нужна никакая дополнительная информация, поэтому за именем `DisplayMessage` на диаграмме классов следуют пустые круглые скобки (как и в объявлении метода в строке 8 на ил. 4.1). Знак «плюс» (+) перед именем операции сообщает, что `DisplayMessage` является открытой операцией в UML (то есть открытым методом в C#). Диаграммы классов UML часто используются для представления краткой информации об атрибутах и операциях классов.



**Ил. 4.3.** Диаграмма классов UML показывает, что класс `GradeBook` содержит открытую операцию `DisplayMessage`

## 4.4. Объявление метода с параметром

В нашей аналогии с машинами из раздела 4.2 упоминалось, что нажатие педали газа отправляет машине сообщение о выполнении операции — повышении скорости. Но насколько должна повыситься скорость? Чем дольше вы удерживаете педаль, тем больше прирост скорости. Таким образом, сообщение должно включать как выполняемую операцию, так и дополнительную информацию, используемую для выполнения этой операции. Дополнительная информация называется *параметром* — значение параметра помогает машине определить величину ускорения. Аналогичным образом метод может потребовать передачи одного или нескольких параметров, представляющих дополнительную информацию для выполнения его операции. При вызове метода для каждого из параметров передаются значения, называемые *аргументами*. Например, методу `Console.WriteLine` должен передаваться аргумент с данными, выводимыми в консольном окне. Аналогичным образом для внесения средств на банковский счет метод `Deposit` определяет параметр, представляющий вносимую сумму. При вызове метода `Deposit` параметру метода присваивается значение переданного аргумента. Метод зачисляет эту сумму на счет, увеличивая баланс счета.

В нашем следующем примере объявляется класс `GradeBook` (ил. 4.4) с методом `DisplayMessage`, который выводит в приветствии название учебного курса (пример

выполнения приведен на ил. 4.5). Название курса передается в параметре нового метода `DisplayMessage`.

```

1 // Ил. 4.4: GradeBook.cs
2 // Объявление класса с методом, получающим параметр.
3 using System;
4
5 public class GradeBook
6 {
7     // Вывод приветствия для пользователя GradeBook
8     public void DisplayMessage( string courseName )
9     {
10         Console.WriteLine( "Welcome to the grade book for\n{0}!",
11                             courseName );
12     } // Конец метода DisplayMessage
13 } // Конец класса GradeBook

```

**Ил. 4.4.** Объявление класса с методом, получающим параметр

```

1 // Ил. 4.5: GradeBookTest.cs
2 // Создание объекта GradeBook и передача строки
3 // его методу DisplayMessage.
4 using System;
5
6 public class GradeBookTest
7 {
8     // Метод Main начинает выполнение программы
9     public static void Main( string[] args )
10    {
11        // Создание объекта GradeBook и присваивание его myGradeBook
12        GradeBook myGradeBook = new GradeBook();
13
14        // Запрос названия учебного курса
15        Console.WriteLine( "Please enter the course name:" );
16        string nameOfCourse = Console.ReadLine(); // Чтение строки текста
17        Console.WriteLine(); // Вывод пустой строки
18
19        // Вызов метода DisplayMessage объекта myGradeBook
20        // и передача nameOfCourse в аргументе
21        myGradeBook.DisplayMessage( nameOfCourse );
22    } // Конец Main
23 } // Конец класса GradeBookTest

```

```

Please enter the course name:
CS101 Introduction to C# Programming

Welcome to the grade book for
CS101 Introduction to C# Programming!

```

**Ил. 4.5.** Создание объекта `GradeBook` и вызов его метода `DisplayMessage` с передачей строки

Прежде чем браться за рассмотрение новых возможностей класса `GradeBook`, давайте посмотрим, как этот класс используется из метода `Main` класса `GradeBookTest` (см. ил. 4.5). Строка 12 создает объект класса `GradeBook` и присваивает его переменной

`myGradeBook`. Строка 15 предлагает пользователю ввести название учебного курса. Строка 16 читает название, введенное пользователем, и присваивает его переменной `nameOfCourse`; для ввода данных используется метод `ReadLine` класса `Console`. Пользователь вводит название курса и нажимает клавишу `Enter`, чтобы передать название приложению. Клавиша `Enter` вставляет в конец введенной последовательности символ новой строки. Метод `ReadLine` читает введенные символы до символа новой строки и возвращает строку с предшествующими символами (не включая символ новой строки, который отбрасывается).

В строке 21 вызывается метод `DisplayMessage` объекта `myGradeBook`. Имя `nameOfCourse` в круглых скобках — аргумент, передаваемый методу `DisplayMessage` для выполнения его операции. Значение переменной `nameOfCourse` в `Main` становится значением метода параметра `courseName` метода `DisplayMessage` (строка 8 на ил. 4.4). При выполнении приложения метод `DisplayMessage` включает в приветствие введенное название курса (см. ил. 4.5).



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 4.1

Обычно объекты создаются оператором `new`. Исключение составляют строковые литералы, заключенные в кавычки, — например, `"hello"`. Строковые литералы неявно создаются C# тогда, когда они впервые встречаются в коде.

#### Подробнее об аргументах и параметрах

При объявлении метода необходимо указать в его объявлении, нужны ли методу дополнительные данные для выполнения его операции. Дополнительная информация включается в список параметров в круглых скобках за именем метода. Список может содержать любое количество параметров (или не содержать ни одного). Каждый параметр объявляется как переменная, то есть с указанием типа и идентификатора. В нашем случае тип `string` и идентификатор `courseName` сообщают, что методу `DisplayMessage` для выполнения его операции требуется строковое значение. В момент вызова значение аргумента, указанное при вызове, присваивается соответствующему параметру (`courseName` в данном случае) в заголовке метода. Затем тело метода использует параметр `courseName` для обращения к значению. В строках 10–11 на ил. 4.4 значение параметра `courseName` выводится с использованием форматного элемента `{0}` в первом аргументе `WriteLine`. Имя переменной-параметра (см. ил. 4.4, строка 8) может совпадать с именем переменной-аргумента или отличаться от него (см. ил. 4.5, строка 21).

Метод может объявить несколько параметров; в этом случае каждое объявление параметра отделяется от следующего запятой. Количество аргументов при вызове метода должно соответствовать количеству обязательных параметров в списке параметров из объявления вызываемого метода. Кроме того, типы аргументов в вызове метода должны быть совместимы с типами соответствующих параметров в объявлении метода. (Как вы узнаете в следующих главах, тип аргумента и тип соответствующего параметра не обязаны точно совпадать.) В нашем примере вызов метода передает один аргумент типа `string` (`nameOfCourse` объявляется с типом

`string` в строке 16 на ил. 4.5); в объявлении метода указан один параметр типа `string` (строка 8 на ил. 4.4). Таким образом, тип аргумента при вызове метода в данном случае точно совпадает с типом параметра в заголовке метода.



#### ТИПИЧНАЯ ОШИБКА 4.1

Если количество аргументов при вызове метода не соответствует количеству обязательных параметров в объявлении метода, происходит ошибка компиляции.

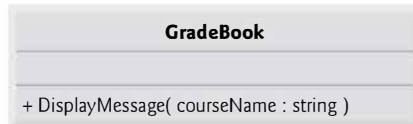


#### ТИПИЧНАЯ ОШИБКА 4.2

Если типы аргументов при вызове метода несовместимы с типами соответствующих параметров в объявлении метода, происходит ошибка компиляции.

### Обновленная диаграмма классов UML для класса `GradeBook`

Диаграмма классов UML на ил. 4.6 моделирует класс `GradeBook` на ил. 4.4. Как и в листинге на ил. 4.4, этот класс `GradeBook` содержит открытую операцию `DisplayMessage`, однако данная версия `DisplayMessage` имеет параметр. В UML моделирование параметров несколько отличается от C#: в круглых скобках за именем операции сначала указывается имя параметра, за ним следует двоеточие и тип параметра. В UML поддерживаются некоторые типы данных, сходные с типами C#. Например, типы UML `String` и `Integer` соответствуют типам C# `string` и `int`. К сожалению, не все типы C# имеют аналоги в UML. По этой причине, а также для предотвращения путаницы между типами UML и типами C# мы используем на диаграммах классов UML только типы C#. Метод `DisplayMessage` класса `Gradebook` (см. ил. 4.4) имеет строковый параметр с именем `courseName`, поэтому на ил. 4.6 в круглых скобках после имени `DisplayMessage` указывается параметр `courseName : string`.



**Ил. 4.6.** Диаграмма классов UML показывает, что класс `GradeBook` содержит открытую операцию `DisplayMessage` с параметром `courseName` типа `string`

### Директивы `using`

Обратите внимание на директиву `using` на ил. 4.5 (строка 4). Она сообщает компилятору, что приложение использует классы из пространства имен `System` — такие, как класс `Console`. Почему директива `using` необходима для использования класса `Console`, но не для класса `GradeBook`? Классы, откомпилированные в одном проекте (такие, как `GradeBook` и `GradeBookTest`), связаны особыми отношениями. По умолчанию считается, что такие классы принадлежат одному пространству имен. Директива `using` не обязательна, когда класс использует другой класс из того же пространства имен, — например, когда класс `GradeBookTest` использует класс

GradeBook. Для простоты в наших примерах из этой книги пространства имен не объявляются. Любые классы, не включенные в конкретное пространство имен, неявно помещаются в так называемое *глобальное пространство имен*.

Вообще говоря, директива `using` в строке 4 не обязательна, если мы всегда будем ссылаться на класс `Console` в виде `System.Console`, то есть с указанием пространства имен и имени класса. Такие имена классов называются *полными* (fully qualified). Например, строка 15 может быть записана в виде

```
System.Console.WriteLine( "Please enter the course name:" );
```

Большинство программистов C# считают, что полные имена неудобны, и предпочитают использовать директивы `using`.

## 4.5. Переменные экземпляров и свойства

В главе 3 все переменные приложения объявлялись в методе `Main`. Переменные, объявленные в теле метода, называются *локальными переменными* и могут использоваться только в этом методе. После завершения метода значения его локальных переменных теряются. Вспомните, о чем говорилось в разделе 4.2: объект обладает атрибутами, которые сопровождают его при использовании в приложении. Такие атрибуты существуют до вызова метода объекта и продолжают существовать после того, как метод будет выполнен.

В объявлениях классов атрибуты представляются переменными. Такие переменные называются *полями* и объявляются в объявлении класса, но за пределами объявлений методов этого класса. Когда каждый объект класса поддерживает собственную копию атрибута, поле, представляющее атрибут, также называется *переменной экземпляра* — каждый объект (экземпляр) класса имеет собственную копию переменной. В главе 10 будет рассмотрена другая разновидность полей: статические переменные, совместно используемые всеми объектами одного класса.

Класс обычно содержит одно или несколько свойств для работы с атрибутами, принадлежащими конкретному объекту класса. В примере, приведенном в этом разделе, представлен класс `GradeBook` с переменной экземпляра `courseName`, представляющей название учебного курса конкретного объекта `GradeBook`, а также свойством `CourseName` для работы с `courseName`.

### Класс `GradeBook` с переменной экземпляра и свойством

В следующем приложении (ил. 4.7–4.8) класс `GradeBook` (см. ил. 4.7) хранит название курса в переменной экземпляра, чтобы его можно было использовать в любой момент во время выполнения приложения. Класс также содержит один метод `DisplayMessage` (строки 24–30) и одно свойство `CourseName` (строки 11–21). Вспомните, о чем говорилось в главе 2: свойства предназначены для манипуляций с атрибутами объекта. Например, в этой главе мы использовали свойство `Text` элемента управления `Label` для определения текста надписи. На этот раз свойство используется в коде, а не в окне

свойств IDE. Для этого мы сначала объявляем свойство членом класса `GradeBook`. Как вы вскоре увидите, свойство `CourseName` класса `GradeBook` может использоваться для хранения названия курса в `GradeBook` (в переменной экземпляра `courseName`) или чтения названия курса `GradeBook` (из переменной экземпляра `courseName`).

Метод `DisplayMessage`, вызываемый без параметров, выводит приветствие с названием курса. Однако в этой реализации метод читает название курса из переменной экземпляра `courseName` с использованием свойства `CourseName`.

```

1  // Ил. 4.7: GradeBook.cs
2  // Класс GradeBook с закрытой переменной экземпляра courseName
3  // и открытым свойством для чтения и записи его значения.
4  using System;
5
6  public class GradeBook
7  {
8      private string courseName; // Название курса GradeBook
9
10     // Свойство для чтения и записи названия курса
11     public string CourseName
12     {
13         get
14         {
15             return courseName;
16         } // Конец get
17         set
18         {
19             courseName = value;
20         }
21     } // Конец свойства CourseName
22
23     // Вывод приветствия для пользователя GradeBook
24     public void DisplayMessage()
25     {
26         // Использование свойства CourseName для чтения
27         // названия курса из объекта GradeBook
28         Console.WriteLine( "Welcome to the grade book for\n{0}!",
29             CourseName ); // Вывод свойства CourseName
30     } // Конец метода DisplayMessage
31 } // Конец класса GradeBook

```

**Ил. 4.7.** Класс `GradeBook` содержит закрытую переменную `courseName` и открытое свойство для чтения и записи ее значения

Преподаватель обычно ведет несколько учебных курсов с разными названиями. В строке 8 объявляется переменная `courseName` с типом `string`. Строка 8 объявляет переменную экземпляра, потому что переменная объявляется в теле класса (строки 7–31), но вне тел метода класса (строки 24–30) и свойства (строки 11–21). Каждый экземпляр (то есть объект) класса `GradeBook` содержит одну копию каждой переменной экземпляра. Если создать два объекта `GradeBook`, то каждый объект будет содержать свою копию `courseName`. Все методы и свойства класса `GradeBook` могут напрямую обращаться к переменной экземпляра `courseName`, но для работы с переменными экземпляров лучше использовать свойства (как это делается в строке 29

метода `DisplayMessage`). Причины, на которых базируются такие рекомендации, вскоре станут понятны.

### Модификаторы доступа `public` и `private`

Большинство объявлений переменных экземпляров начинается с ключевого слова `private` (как в строке 8). Ключевое слово `private`, как и `public`, является модификатором доступа. Переменные, свойства и методы, объявленные с модификатором доступа `private`, доступны только для членов класса, в котором они объявлены (например, для свойств и методов). Таким образом, переменная `courseName` может использоваться только в свойстве `CourseName` и методе `DisplayMessage` класса `GradeBook`.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 4.2

Снабжайте каждое объявление поля и метода модификатором доступа. Как правило, переменные экземпляров должны объявляться закрытыми, а методы и свойства — открытыми. Если модификатор доступа перед членом класса отсутствует, по умолчанию подразумевается закрытый уровень доступа. Как вы вскоре увидите, некоторые методы можно объявлять закрытыми, если они будут вызываться только из других методов этого класса.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 4.3

Объявляя переменные экземпляров класса закрытыми, а методы и свойства класса — открытыми, вы упрощаете отладку, поскольку проблемы с обработкой данных локализуются в методах и свойствах класса (закрытые переменные экземпляров доступны только для этих методов и свойств).

Объявление переменных экземпляров с модификатором доступа `private` называется сокрытием информации (или *инкапсуляцией*). Когда приложение создает объект класса `GradeBook`, переменная `courseName` инкапсулируется (скрывается) в объекте, а обращаться к ней могут только члены класса объекта.

### Чтение и запись закрытых переменных экземпляров

Как разрешить программе работать с закрытыми переменными экземпляров класса, но проследить за тем, чтобы они имели корректные значения? Нужно предоставить в распоряжение программиста контролируемые механизмы чтения и записи значения переменной. И хотя вы можете определить методы вида `GetCourseName` и `SetCourseName`, свойства C# предоставляют более элегантное решение. Давайте посмотрим, как объявлять и использовать свойства.

### Класс `GradeBook` со свойством

Объявление свойства `CourseName` класса `GradeBook` расположено в строках 11–21 на ил. 4.7. Свойство начинается в строке 11 с модификатора доступа (в данном случае `public`), за которым следует тип, представляемый свойством (`string`) и имя свойства (`CourseName`). Имена свойств формируются по тем же правилам, что и имена методов и классов.

Свойства содержат методы доступа, которые реализуют чтение и запись данных. Объявление свойства может содержать `get`-метод доступа и/или `set`-метод.



Get-метод (строки 13–16) читает значение закрытой переменной экземпляра `courseName`; set-метод (строки 17–20) позволяет клиенту изменить `courseName`.

После того как вы определите свойство, оно используется в коде как переменная. Например, свойству можно присвоить значение оператором `=`. Эта команда выполняет set-метод свойства для задания значения соответствующей переменной экземпляра. Аналогичным образом при обращении к свойству за его значением (например, для вывода на экран) выполняется код get-метода свойства. Вскоре мы рассмотрим примеры использования свойств. По действующим правилам имя свойства представляет собой имя переменной экземпляра, записанное с прописной буквы (например, `CourseName` — свойство, представляющее переменную экземпляра `courseName`); язык C# различает регистр символов, поэтому эти идентификаторы считаются разными.

### Get- и set-методы доступа

Рассмотрим повнимательнее get- и set-методы свойства `CourseName` (см. ил. 4.7). Get-метод (строки 13–16) начинается с ключевого слова `get`, а его тело заключено в фигурные скобки. Тело метода доступа содержит команду `return`, которая состоит из ключевого слова `return` и выражения. Значение выражения возвращается клиентскому коду, использующему свойство. В нашем примере при обращении к свойству `CourseName` возвращается значение `courseName`. Например, в следующей команде:

```
string theCourseName = gradeBook.CourseName;
```

выражение `gradeBook.CourseName` (где `gradeBook` — объект класса `GradeBook`) выполняет get-метод свойства `CourseName`, который возвращает значение переменной экземпляра `courseName`. Значение сохраняется в переменной `theCourseName`. Свойство `CourseName` используется так же просто, как если бы оно было переменной экземпляра. Еще раз подчеркнем, что клиент не может напрямую работать с переменной экземпляра `courseName`, потому что она объявлена закрытой.

Set-метод доступа (строки 17–20) начинается с ключевого слова `set`, а его тело заключено в фигурные скобки. Если свойство `CourseName` включается в команду присваивания, как в примере:

```
gradeBook.CourseName = "CS100 Introduction to Computers";
```

текст `"CS100 Introduction to Computers"` присваивается контекстному ключевому слову `value`, и выполняется set-метод доступа. Учтите, что `value` неявно объявляется и инициализируется в set-методе — объявление локальной переменной `value` в теле метода приводит к ошибке компиляции. В строке 19 содержимое `value` сохраняется в переменной экземпляра `courseName`. Set-метод не возвращает никаких данных при завершении своей операции.

Команды в строках 15 и 19 тела свойства (см. ил. 4.7) обращаются к переменной `courseName` даже несмотря на то, что она объявлена за пределами свойства. Переменная экземпляра `courseName` может использоваться в методах и свойствах класса `GradeBook`, потому что `courseName` является переменной экземпляра класса.

### Использование свойства `CourseName` в методе `DisplayMessage`

Метод `DisplayMessage` (строки 24–30 на ил. 4.7) не получает параметров. Строки 28–29 выводят приветствие, включающее значение переменной экземпляра `courseName`. При этом мы не обращаемся к `courseName` напрямую, а используем свойство `CourseName` (строка 29), которое выполняет get-метод свойства, возвращающий значение `courseName`.

### Класс `GradeBookTest`, демонстрирующий использование класса `GradeBook`

Класс `GradeBookTest` (ил. 4.8) создает объект `GradeBook` и демонстрирует использование свойства `CourseName`. В строке 11 создается объект `GradeBook`, который присваивается локальной переменной `myGradeBook`. Строки 14–15 выводят исходное название учебного курса, используя свойство `CourseName` объекта, — это приводит к выполнению get-метода свойства, который возвращает значение `courseName`.

В первой строке вывода отображается пустое имя (заключенное в пару апострофов `' '`). В отличие от локальных переменных, которые не инициализируются автоматически, у каждого поля есть исходное значение по умолчанию — значение, которое предоставляет C# при отсутствии явно заданного исходного значения. Таким образом, поля не обязательно явно инициализировать перед их использованием в приложении — если только они должны быть инициализированы значениями, отличными от значений по умолчанию. Переменные экземпляра типа `string` (как `courseName`) инициализируются значением по умолчанию `null`. При выводе строковой переменной, содержащей `null`, текст на экране не выводится.

```
1 // Ил. 4.8: GradeBookTest.cs
2 // Create and manipulate a GradeBook object.
3 using System;
4
5 public class GradeBookTest
6 {
7     // Метод Main начинает выполнение программы
8     public static void Main( string[] args )
9     {
10         // Создание объекта GradeBook и присваивание его myGradeBook
11         GradeBook myGradeBook = new GradeBook();
12
13         // Вывод исходного значения CourseName
14         Console.WriteLine( "Initial course name is: '{0}'\n",
15             myGradeBook.CourseName );
16
17         // Запрос и чтение названия учебного курса
18         Console.WriteLine( "Please enter the course name:" );
19         myGradeBook.CourseName = Console.ReadLine(); // Задает CourseName
20         Console.WriteLine(); // Вывод пустой строки
21
22         // Вывод приветствия после ввода названия курса
23         myGradeBook.DisplayMessage();
24     } // Конец Main
25 } // Конец класса GradeBookTest
```

**Ил. 4.8.** Создание и манипуляция объекта `GradeBook` (продолжение ↗)

```
Initial course name is: ''

Please enter the course name:
CS101 Introduction to C# Programming

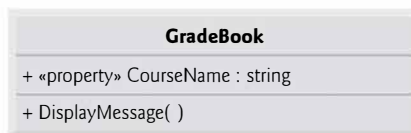
Welcome to the grade book for
CS101 Introduction to C# Programming!
```

#### Ил. 4.8. Создание и манипуляция объекта GradeBook (окончание)

В строке 18 пользователю предлагается ввести название учебного курса. В строке 19 введенное название присваивается свойству `CourseName` объекта `myGradeBook`. При присваивании `CourseName` заданное значение (которое в данном случае возвращается методом `ReadLine`) присваивается неявному параметру `value` set-метода `CourseName` (строки 17–20, ил. 4.7). Затем set-метод присваивает параметр `value` переменной экземпляра `courseName` (строка 19 на ил. 4.7). Строка 20 (см. ил. 4.8) выводит пустую строку, затем строка 23 вызывает метод `DisplayMessage` объекта `myGradeBook` для вывода приветствия с названием учебного курса.

## 4.6. Диаграмма классов UML со свойством

На ил. 4.9 представлена обновленная диаграмма классов UML для версии класса `GradeBook`, приведенной в листинге на ил. 4.7. Свойства UML моделируются атрибутами — свойство (в данном случае `CourseName`) включается в список как открытый атрибут (на что указывает знак «+»), которому предшествует слово `property` в кавычках-«елочках» (« и »). Описатели в угловых кавычках (в UML они называются *стереотипами*) помогают отличать свойства от других атрибутов и операций. Тип свойства в UML указывается через двоеточие после имени свойства. Get- и set-методы свойства подразумеваются; они не указываются явно на диаграмме классов UML. Класс `GradeBook` также содержит один открытый метод `DisplayMessage`, поэтому эта операция указывается в третьем отделении. Вспомните, что знак «+» является признаком открытого уровня видимости.



**Ил. 4.9.** Диаграмма классов UML показывает, что класс `GradeBook` содержит открытое свойство `CourseName` типа `string` и один открытый метод

Диаграмма классов помогает в проектировании класса, поэтому показывать на ней все подробности реализации не обязательно. Так как переменная экземпляра `courseName`, с которой работает свойство, относится к подробностям реализации этого свойства, на диаграмме классов она не показывается. Программист, реализующий

класс `GradeBook` на базе этой диаграммы, создает переменную экземпляра `courseName` как часть процесса реализации (как было сделано на ил. 4.7).

В некоторых ситуациях может возникнуть необходимость в моделировании закрытых переменных экземпляров класса. Переменные экземпляров, как и свойства, являются атрибутами класса и моделируются в среднем отделении. В UML переменные экземпляров представляются как атрибуты, для чего указывается имя атрибута, двоеточие и тип атрибута. Чтобы указать, что атрибут является закрытым, на диаграмме классов перед именем атрибута размещается признак закрытого уровня видимости — знак «минус» (-). Например, переменная экземпляра `courseName` на ил. 4.7 моделируется строкой «-courseName : string», обозначающей закрытый атрибут типа `string`.

## 4.7. Свойства и методы доступа

Может показаться, что использование свойств, представленное ранее в этой главе, *нарушает* принцип закрытости данных. И хотя предоставление свойства с `get`- и `set`-методами на первый взгляд не отличается от объявления открытой переменной экземпляра, это впечатление обманчиво. Значение открытой переменной экземпляра доступно для чтения и записи для любого свойства или метода в программе. Если переменная экземпляра объявлена закрытой, то клиентский код может обращаться к ней только через незакрытые свойства или методы класса. Это позволяет классу контролировать процесс чтения или записи данных. Например, `get`- и `set`-методы могут выполнять преобразование между форматом данных, хранящихся в закрытой переменной экземпляра, и форматом, удобным для клиента.

### Проверка данных

Допустим, в классе `Clock` время суток представлено закрытой переменной экземпляра `time` типа `int`. В переменной хранится количество секунд, прошедшее с полуночи. Для работы с этой переменной экземпляра класс предоставляет свойство `Time` типа `string`. И хотя методы доступа обычно возвращают данные в том виде, в каком они хранятся в объекте, «низкоуровневый» формат не является обязательным. Когда клиент обращается к свойству `Time` объекта `Clock`, `get`-метод свойства может использовать переменную экземпляра `time` для вычисления часов, минут и секунд, прошедших с полуночи, и вернуть время в строковом виде в формате "`чч:мм:сс`". Или допустим, что свойству `Time` объекта `Clock` присваивается строка в формате "`чч:мм:сс`". Используя средства работы со строками, представленные в главе 16, и метод `Convert.ToInt32` из раздела 3.6, `set`-метод свойства `Time` может преобразовать эту строку в число секунд с полуночи и сохранить полученный результат в закрытой переменной экземпляра `time` объекта `Clock`. `Set`-метод свойства `Time` также может проверять попытки изменения переменной экземпляра и следить за тем, чтобы полученное значение представляло действительное время (например, строка "`12:30:45`" представляет действительное время, а строка "`42:85:70`" — нет). Проверка данных продемонстрирована в разделе 4.11. Итак, хотя методы доступа

свойства позволяют клиентам работать с закрытыми данными, они тщательно контролируют выполняемые операции, и закрытые данные клиента остаются надежно инкапсулированными (то есть скрытыми) в объекте. Такой контроль невозможен с открытыми переменными экземпляров, которым клиент легко может присвоить недействительные значения.

### Работа с данными классов через свойства класса

Свойства класса также должны использоваться собственными методами класса для работы с закрытыми переменными экземпляров несмотря на то, что методы могут напрямую обращаться к закрытым переменным экземпляров. Обращение к переменной экземпляра через методы доступа свойства (как в теле метода `DisplayMessage` на ил. 4.7, строки 28–29) повышает надежность класса, упрощает сопровождение кода и снижает вероятность неполадок. Если мы решим изменить представление переменной экземпляра `courseName`, изменять объявление метода `DisplayMessage` не придется — достаточно изменить методы доступа, напрямую взаимодействующие с переменной экземпляра.

Допустим, вы решили представить название учебного курса двумя переменными экземпляров — `courseNumber` (например, "CS101") и `courseTitle` (например, "Introduction to C# Programming"). Метод `DisplayMessage` по-прежнему может использовать `get`-метод свойства `CourseName` для получения полного названия учебного курса, выводимого в приветствии. В этом случае `get`-метод должен построить и вернуть строку, состоящую из `courseNumber` и `courseTitle`. Метод `DisplayMessage` по-прежнему будет выводить полное название «CS101 Introduction to C# Programming», потому что изменение переменных экземпляров класса на нем никак не отразилось.

## 4.8. Автоматическая реализация свойств

На ил. 4.7 мы создали класс `GradeBook` с закрытой переменной экземпляра `courseName` и открытым свойством `CourseName`, через которое клиентский код работает с `courseName`. Обратившись к определению свойства `CourseName` (см. ил. 4.7, строки 11–21), мы видим, что `get`-метод просто возвращает значение закрытой переменной экземпляра `courseName`, а `set`-метод просто присваивает ей новое значение — никакой другой логики в методах доступа нет. Для таких случаев C# предоставляет механизм *автоматической реализации свойств*. Компилятор C# создает закрытую переменную экземпляра, `get`- и `set`-методы для чтения и изменения закрытой переменной. В отличие от свойств, определяемых пользователем, автоматически реализуемое свойство должно иметь как `get`-, так и `set`-метод. Это делает возможной тривиальную реализацию свойства при исходном проектировании класса. Если позднее вы решите добавить в `get`- или `set`-метод дополнительную логику, достаточно изменить реализацию свойства. Например, чтобы использовать автоматическую реализацию свойства в классе `GradeBook` на ил. 4.7, достаточно заменить закрытую переменную экземпляра в строке 8 и свойство в строках 11–21 следующим кодом:

```
public string CourseName { get; set; }
```

### Фрагменты кода в автоматически реализуемых свойствах

В IDE существует механизм *фрагментов кода* (code snippets), позволяющий вставлять в исходный код заранее определенные шаблоны. Один из таких фрагментов дает возможность вставить открытое, автоматически реализованное свойство: введите слово «prop» в окне кода и дважды нажмите клавишу Tab. Некоторые части вставленного кода выделяются, чтобы вы могли легко изменить тип и имя свойства. Переход от одной выделенной части к другой во вставленном коде осуществляется клавишей Tab. По умолчанию новое свойство имеет тип `int` и имя `MyProperty`. Чтобы получить список всех доступных фрагментов кода, нажмите клавиши `Ctrl+k`, `Ctrl+x`. При этом в редакторе кода открывается окно `Insert Snippet`. Для перемещения между папками фрагментов Visual C# используется мышь. Эту функцию также можно вызвать другим способом: щелкните правой кнопкой мыши в редакторе исходного кода и выберите команду `Insert Snippet...`

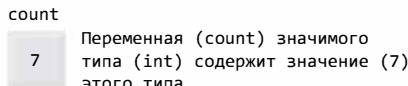
## 4.9. Значимые типы и ссылочные типы

Типы в языке C# делятся на две категории: значимые типы (value types) и ссылочные типы (reference types).

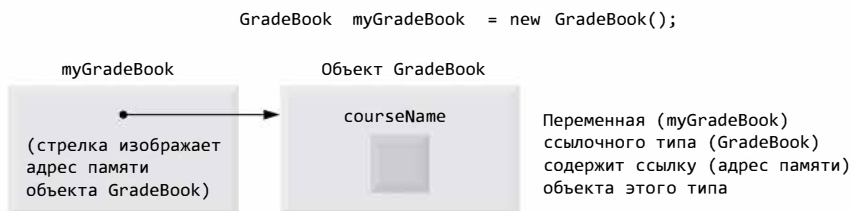
### Значимые типы

Простые типы C# (такие, как `int` и `double`) относятся к категории значимых. Переменная значимого типа просто содержит значение некоторого типа. Например, на ил. 4.10 изображена переменная типа `int` с именем `count`, содержащая значение 7.

```
int count = 7;
```



**Ил. 4.10.** Переменная значимого типа



**Ил. 4.11.** Переменная ссылочного типа

## Ссылочные типы

С другой стороны, переменная ссылочного типа (иногда такие переменные называются *ссылками* — references) содержит *адрес* ячейки памяти, в которой хранятся данные, ассоциированные с переменной. Говорят, что такая переменная ссылается на объект в программе. Строка 11 на ил. 4.8 создает объект `GradeBook`, размещает его в памяти и сохраняет ссылку на него в переменной `myGradeBook` типа `GradeBook` (ил. 4.11). Объект `GradeBook` изображен со своей переменной экземпляра `courseName`.

## Инициализация ссылочных переменных значением null

Переменные экземпляров ссылочного типа (такие, как `myGradeBook` на ил. 4.11) по умолчанию инициализируются значением `null`. Тип `string` является ссылочным типом, поэтому переменная `courseName` изображена на ил. 4.11 с пустым блоком, изображающим переменную со значением `null`. Строковая переменная со значением `null` не является пустой строкой, которая представляется литералом `""` или конструкцией `string.Empty`. Значение `null` представляет ссылку, которая не указывает на объект, — тогда как пустая строка представляет строковый объект, не содержащий символов.

## Использование ссылки для отправки сообщений объекту

Для вызова методов и обращения к свойствам объекта клиент должен использовать переменную, которая содержит ссылку на объект. На ил. 4.8 команды `Main` используют переменную `myGradeBook`, которая содержит ссылку на объект `GradeBook`, для отправки сообщений объекту `GradeBook`. Эти сообщения представляют собой вызовы методов (например, `DisplayMessage`) или обращения к свойствам (например, `CourseName`), позволяющие программе взаимодействовать с объектами `GradeBook`. Например, в команде (строка 19 на ил. 4.8)

```
myGradeBook.CourseName = Console.ReadLine(); // Задаёт CourseName
```

ссылка `myGradeBook` используется для задания названия учебного курса посредством присваивания значения свойству `CourseName`. При этом объекту `GradeBook` отправляется сообщение о вызове `set`-метода свойства `CourseName`. В аргументе сообщения передается значение, полученное из пользовательского ввода (в нашем случае `"CS101 Introduction to C# Programming"`), необходимое `set`-методу `CourseName` для выполнения его операции. `Set`-метод использует эту информацию для присваивания переменной экземпляра `courseName`. В разделе 7.16 значимые и ссылочные типы будут рассмотрены более подробно.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 4.4

Объявленный тип переменной (например, `int`, `double` или `GradeBook`) определяет, к какой категории относится переменная — значимой или ссылочной. Если тип переменной не принадлежит к числу простых типов, не является перечислимым или структурным типом (см. раздел 7.10 и главу 16 соответственно), то это ссылочный тип. Например, запись `Account account1` означает, что `account1` — переменная, которая может ссылаться на объект `Account`.

## 4.10. Инициализация объектов конструкторами

Как упоминалось в разделе 4.5, при создании объекта `GradeBook` (см. ил. 4.7) происходит создание объекта, а его переменная экземпляра `courseName` по умолчанию инициализируется значением `null`. Это также относится и к закрытым переменным экземпляров, которые создаются компилятором для автоматически реализуемого свойства `CourseName` (раздел 4.8). А если вы хотите задать название учебного курса при создании объекта `GradeBook`? Каждый класс может предоставить конструктор, который будет использоваться для инициализации объектов класса при их создании. Оператор `new` вызывает конструктор класса для проведения инициализации. Вызов конструктора обозначается именем класса, за которым следуют круглые скобки. Например, строка 11 на ил. 4.8 сначала использует `new` для создания объекта `GradeBook`. Пустые круглые скобки в `newGradeBook()` обозначают вызов конструктора класса без аргументов.

Компилятор предоставляет открытый конструктор по умолчанию без параметров для любого класса, не содержащего явного определения конструктора, так что конструктор имеется у каждого класса. Конструктор по умолчанию не изменяет значения переменных экземпляров, назначаемые по умолчанию.

### Нестандартная инициализация в конструкторах

При объявлении класса разработчик может предоставить собственный конструктор (или несколько конструкторов, как вы узнаете в главе 10) для проведения нестандартной инициализации объектов класса. Допустим, вы хотите, чтобы при создании объекта `GradeBook` можно было сразу задать название учебного курса:

```
GradeBook myGradeBook =  
    new GradeBook( "CS101 Introduction to C# Programming" );
```

Аргумент `"CS101IntroductiontoC#Programming"` передается конструктору объекта `GradeBook` и используется для инициализации `CourseName`. Каждый раз, когда в программе создается новый объект `GradeBook`, вы можете задавать новое название. Чтобы приведенная выше команда работала, класс должен иметь конструктор с параметром `string`. На ил. 4.12 представлен измененный класс `GradeBook` с таким конструктором.

```
1  // Ил. 4.12: GradeBook.cs  
2  // Класс GradeBook с конструктором для инициализации названия курса.  
3  using System;  
4  
5  public class GradeBook  
6  {  
7      // Автоматически реализованное свойство CourseName неявно  
8      // создает переменную экземпляра для названия учебного курса.  
9      public string CourseName { get; set; }  
10  
11     // Конструктор инициализирует автоматически реализованное свойство
```

**Ил. 4.12.** Класс `GradeBook` с конструктором, инициализирующим название учебного курса (продолжение ↗)



```
12 // CourseName строкой name, переданной в аргументе
13 public GradeBook( string name )
14 {
15     CourseName = name; // Инициализация CourseName
16 } // Конец конструктора
17
18 // Вывод приветствия для пользователя GradeBook
19 public void DisplayMessage()
20 {
21     // Автоматически реализованное свойство CourseName используется
22     // для получения названия курса текущего объекта GradeBook
23     Console.WriteLine( "Welcome to the grade book for\n{0}!",
24         CourseName );
25 } // Конец метода DisplayMessage
26 } // Конец класса GradeBook
```

**Ил. 4.12.** Класс `GradeBook` с конструктором, инициализирующим название учебного курса (окончание)

### Объявление конструктора

В строках 13–16 объявляется конструктор класса `GradeBook`. Имя конструктора должно совпадать с именем класса. В списке параметров конструктора, как и у любого другого метода, указываются данные, необходимые для выполнения его операции. При создании объекта оператором `new` эти данные передаются в круглых скобках после имени класса. Однако в отличие от других методов, для конструктора не указывается тип возвращаемого значения (даже `void`). В строке 13 указано, что конструктор класса `GradeBook` имеет параметр с именем `name` и типом `string`. В строке 15 значение `name`, переданное конструктору, используется для инициализации автоматически реализованного свойства `CourseName` через его метод доступа.

### Инициализация объектов `GradeBook` нестандартным конструктором

На ил. 4.13 продемонстрирована инициализация объектов `GradeBook` с использованием этого конструктора. В строках 12–13 создается и инициализируется объект `GradeBook`. Конструктор класса `GradeBook` вызывается с аргументом `"CS 101 Introduction to C# Programming"` для инициализации названия курса. Создающее объект выражение справа от `=` в строках 12–13 возвращает ссылку на новый объект, которая присваивается переменной `gradeBook1`. В строках 14–15 процесс повторяется для другого объекта `GradeBook`, на этот раз с аргументом `"CS102 DataStructures in C#"` для инициализации названия курса `gradeBook2`. В строках 18–21 свойство `CourseName` обоих объектов используется для чтения названий курсов и проверки правильности их инициализации при создании. В разделе 4.5 мы говорили о том, что каждый экземпляр (то есть объект) класса содержит собственную копию переменных экземпляров этого класса. Результат выполнения подтверждает, что каждый экземпляр `GradeBook` хранит собственное название учебного курса.

```
1 // Ил. 4.13: GradeBookTest.cs
2 // Конструктор GradeBook используется для инициализации названия
3 // учебного курса при создании объекта GradeBook.
4 using System;
5
6 public class GradeBookTest
7 {
8     // Метод Main начинает выполнение программы
9     public static void Main( string[] args )
10    {
11        // Создание объекта GradeBook
12        GradeBook gradeBook1 = new GradeBook( // Вызов конструктора
13            "CS101 Introduction to C# Programming" );
14        GradeBook gradeBook2 = new GradeBook( // Вызов конструктора
15            "CS102 Data Structures in C#" );
16
17        // Вывод исходного значения courseName для каждого объекта
18        Console.WriteLine( "gradeBook1 course name is: {0}",
19            gradeBook1.CourseName );
20        Console.WriteLine( "gradeBook2 course name is: {0}",
21            gradeBook2.CourseName );
22    } // Конец Main
23 } // Конец класса GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to C# Programming
gradeBook2 course name is: CS102 Data Structures in C#
```

**Ил. 4.13.** Конструктор GradeBook используется для задания названия учебного курса при создании объекта GradeBook

Обычно конструкторы объявляются открытыми. Если класс не определяет конструктор явно, переменные экземпляров класса инициализируются значениями по умолчанию — 0 для числовых типов, false для типа bool, null для ссылочных типов. Если в классе объявляются какие-либо конструкторы, C# не создает для этого класса конструктор по умолчанию.

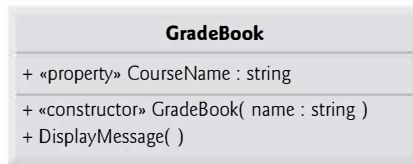
### Добавление конструктора на диаграмму классов UML класса GradeBook

Диаграмма классов UML на ил. 4.14 моделирует класс GradeBook из листинга на ил. 4.12, имеющий конструктор с параметром name типа string. В UML конструкторы, как и операции, моделируются в третьем отделении класса на диаграмме. Чтобы конструктор отличался от других операций класса, перед его именем ставится слово **constructor**, заключенное в кавычки-«елочки» (« и »). Обычно конструкторы перечисляются в третьем отделении перед другими операциями.



#### КАК ИЗБЕЖАТЬ ОШИБОК 4.1

Если инициализация переменных экземпляров, используемая по умолчанию, вам не подходит, предоставьте конструктор, который будет инициализировать эти переменные осмысленными значениями при создании нового объекта класса.



**Ил. 4.14.** Диаграмма классов UML показывает, что класс `GradeBook` содержит конструктор с параметром `name` типа `string`

## 4.11. Числа с плавающей точкой и тип `decimal`

В следующем приложении мы ненадолго отвлечемся от темы `GradeBook` и займемся классом `Account`, представляющим баланс банковского счета. Как правило, денежные суммы являются дробными величинами вида 99,99 или –20,15. По этой причине в классе `Account` баланс представляется вещественным числом. В C# для хранения вещественных чисел существуют три простых типа — `float`, `double` и `decimal`. Типы `float` и `double` называются типами с плавающей точкой. Главное отличие их от `decimal` заключается в том, что переменные `decimal` позволяют точно хранить вещественные числа из ограниченного набора, тогда как переменные с плавающей точкой хранят приближенные значения вещественных чисел, но с гораздо более широким набором представляемых значений. Кроме того, переменные типа `double` способны хранить числа с большей точностью (то есть с большим количеством цифр в дробной части), чем вещественные переменные. Важнейшим применением типа `decimal` является представление денежных сумм при финансовых расчетах.

### Точность и затраты памяти

Переменные типа `float` представляют числа одинарной точности с плавающей точкой и имеют семь значащих цифр. Переменные типа `double` представляют числа двойной точности с плавающей точкой. Они занимают вдвое больше памяти, чем переменные `float`, и обеспечивают представление до 15–16 значащих цифр — примерно вдвое больше, чем у переменных `float`. Переменные типа `decimal` занимают вдвое больше памяти, чем `double`, и обеспечивают представление до 28–29 значащих цифр. В некоторых приложениях даже точности типов `double` и `decimal` оказывается недостаточно — впрочем, такие приложения выходят за рамки книги.

Как правило, программисты используют для представления чисел с плавающей точкой тип `double`. В языке C# все вещественные числа в исходном коде приложения (такие, как 7.33 и 0.0975) по умолчанию рассматриваются как относящиеся к типу `double`. Такие числа называются *литералами с плавающей точкой*. Чтобы определить литерал типа `decimal`, следует поставить в конце вещественного числа суффикс «M» или «m»; например, 7.33M интерпретируется как литерал типа `decimal`, а не `double`. Целочисленные литералы неявно преобразуются к типу `float`, `double` или `decimal` при присваивании переменной одного из этих типов.

Хотя числа с плавающей точкой не всегда обеспечивают 100 % точность представления, они очень часто применяются на практике. Например, когда речь заходит о «нормальной» температуре тела 36,6, нас не всегда интересует точное значение. Если на градуснике отображается температура 36,6, на самом деле она может быть равна 36,5999473210643. Округление этого числа до 36,6 хорошо работает в большинстве ситуаций. Из-за неточности чисел с плавающей точкой тип `decimal` обычно используется там, где вычисления должны быть точными, — например, в финансовых приложениях. Если точности приближенного представления достаточно, то обычно типу `double` отдается предпочтение перед `float`, потому что переменные `double` точнее представляют числа с плавающей точкой. По этой причине в книге для представления денежных сумм будет использоваться тип `decimal`, а для других вещественных чисел — тип `double`.

Вещественные числа также образуются в результате деления. Например, в традиционной арифметике при делении 10 на 3 результат равен 3,3333333...; компьютер выделяет для хранения величины фиксированный объем памяти, поэтому очевидно, что хранимое значение с плавающей точкой может быть только приближенным.



#### ТИПИЧНАЯ ОШИБКА 4.3

Использование чисел с плавающей точкой, которое подразумевает абсолютную точность представления, может привести к логическим ошибкам.

#### Класс `Account` с переменной экземпляра типа `decimal`

Наше приложение (см. ил. 4.15–4.16) содержит простой класс с именем `Account` (см. ил. 4.15), представляющий баланс банковского счета. Типичный банк обслуживает множество счетов, каждый из которых имеет собственный баланс, поэтому в строке 7 объявляется переменная экземпляра с именем `balance` типа `decimal`. Переменная `balance` является переменной экземпляра, потому что она объявляется в теле класса (строки 6–36), но вне объявлений методов и свойств (строки 10–13, 16–19 и 22–35). Каждый экземпляр (то есть объект) класса `Account` содержит собственную копию `balance`.

```
1 // Ил. 4.15: Account.cs
2 // Класс class с конструктором для инициализации
3 // переменной экземпляра balance.
4
5 public class Account
6 {
7     private decimal balance; // Переменная для хранения баланса
8
9     // Конструктор
10    public Account( decimal initialBalance )
11    {
12        balance = initialBalance; // Баланс задается через свойство
13    } // Конец конструктора Account
14
15    // Внесение суммы amount на счет
```

**Ил. 4.15.** Класс `Account` с конструктором, инициализирующим переменную экземпляра `balance` (продолжение ↗)

```
16 public void Credit( decimal amount )
17 {
18     Balance = Balance + amount; // Внесение средств на счет
19 } // Конец метода Credit
20
21 // Свойство для чтения и записи баланса счета
22 public decimal Balance
23 {
24     get
25     {
26         return balance;
27     } // Конец get
28     set
29     {
30         // Убедиться в том, что значение value не отрицательно;
31         // в противном случае баланс не изменяется
32         if ( value >= 0 )
33             balance = value;
34     } // Конец set
35 } // Конец свойства Balance
36 } // Конец класса Account
```

**Ил. 4.15.** Класс Account с конструктором, инициализирующим переменную экземпляра balance (окончание)

### Конструктор класса Account

Класс Account содержит конструктор, метод и свойство. Так как счета часто открываются для немедленного внесения средств, конструктор (строки 10–13) получает параметр `initialBalance` типа `decimal`, представляющий начальный баланс счета. В строке 12 значение `initialBalance` задается свойству `Balance`, с вызовом `set`-метода `Balance` для инициализации переменной `balance`.

### Метод Credit

Метод `Credit` (строки 16–19) не возвращает данные при завершении своей операции, поэтому вместо возвращаемого типа указано ключевое слово `void`. Метод получает один параметр с именем `amount` — значение типа `decimal`, прибавляемое к текущему свойству `Balance`. В строке 18 используются оба метода доступа `Balance`. Выражение `Balance + amount` вызывает `get`-метод свойства `Balance` для получения текущего значения переменной экземпляра `balance` и прибавляет к нему `amount`. Затем результат присваивается переменной экземпляра `balance`, для чего вызывается `set`-метод свойства `Balance` (с заменой предыдущего значения `balance`).

### Свойство Balance

Свойство `Balance` (строки 22–35) предоставляет `get`-метод, при помощи которого клиенты класса (то есть другие классы, использующие этот класс) получают значение `balance` конкретного объекта `Account`. Свойство имеет тип `decimal` (строка 22). Свойство `Balance` также содержит `set`-метод с расширенной функциональностью.

В разделе 4.5 были представлены свойства, `set`-методы которых позволяли клиентам класса изменять значение закрытой переменной экземпляра. На ил. 4.7 класс `GradeBook`


определяет set-метод свойства `CourseName`, присваивающий значение параметра `value` переменной экземпляра `courseName` (строка 19). Свойство `CourseName` не гарантирует, что переменная `courseName` будет содержать только действительные данные.

В приложении на ил. 4.15–4.16 set-метод свойства `Balance` класса `Account` дополняется проверкой данных. Строка 32 (см. ил. 4.15) проверяет, что значение не отрицательно. Если значение больше либо равно 0, то сумма, хранящаяся в `value`, присваивается переменной экземпляра `balance` в строке 33. В противном случае значение `balance` остается неизменным.

### Класс `AccountTest`

Класс `AccountTest` (см. ил. 4.16) создает два объекта `Account` (строки 10–11) и инициализирует значениями `50.00M` и `-7.53M` (литералы типа `decimal`, представляющие вещественные числа 50,00 и –7,53). Конструктор `Account` (строки 10–13 на ил. 4.15) инициализирует `balance` через свойство `Balance`. В предыдущих примерах преимущества от обращения к свойству в конструкторе не были очевидны, но теперь конструктор использует проверку данных, обеспечиваемую set-методом свойства `Balance`. Конструктор просто присваивает значение `Balance` вместо того, чтобы дублировать код проверки из set-метода. В строке 11 на ил. 4.16 конструктору `Account` передается исходный баланс –7.53, конструктор передает это значение set-методу свойства `Balance`, где происходит непосредственная инициализация. Это значение меньше 0, так что set-метод *не* изменяет `balance`, а переменная экземпляра сохраняет значение по умолчанию 0.

```
1 // Ил. 4.16: AccountTest.cs
2 // Создание и использование объектов Account.
3 using System;
4
5 public class AccountTest
6 {
7     // Метод Main начинает выполнение приложения C#
8     public static void Main( string[] args )
9     {
10         Account account1 = new Account( 50.00M ); // Создание Account
11         Account account2 = new Account( -7.53M ); // Создание Account
12
13         // Вывод исходного баланса с использованием свойства
14         Console.WriteLine( "account1 balance: {0:C}",
15             account1.Balance ); // Вывод свойства Balance
16         Console.WriteLine( "account2 balance: {0:C}\n",
17             account2.Balance ); // Вывод свойства Balance
18
19         decimal depositAmount; // Зачисление введенной суммы
20
21         // Запрос и получение пользовательского ввода
22         Console.Write( "Enter deposit amount for account1: " );
23         depositAmount = Convert.ToDecimal( Console.ReadLine() );
24         Console.WriteLine( "adding {0:C} to account1 balance\n",
25             depositAmount );
```

**Ил. 4.16.** Создание и использование объектов `Account` (продолжение )

```
26     account1.Credit( depositAmount ); // Зачисление на баланс account1
27
28     // Вывод балансов
29     Console.WriteLine( "account1 balance: {0:C}",
30         account1.Balance );
31     Console.WriteLine( "account2 balance: {0:C}\n",
32         account2.Balance );
33
34     // Запрос и получение пользовательского ввода
35     Console.Write( "Enter deposit amount for account2: " );
36     depositAmount = Convert.ToDecimal( Console.ReadLine() );
37     Console.WriteLine( "adding {0:C} to account2 balance\n",
38         depositAmount );
39     account2.Credit( depositAmount ); // Зачисление на баланс account2
40
41     // display balances
42     Console.WriteLine( "account1 balance: {0:C}", account1.Balance );
43     Console.WriteLine( "account2 balance: {0:C}", account2.Balance );
44 } // Конец Main
45 } // Конец класса AccountTest
```

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 49.99
adding $49.99 to account1 balance

account1 balance: $99.99
account2 balance: $0.00

Enter deposit amount for account2: 123.21
adding $123.21 to account2 balance

account1 balance: $99.99
account2 balance: $123.21
```

#### Ил. 4.16. Создание и использование объектов Account (окончание)

В строках 14–17 на ил. 4.16 баланс каждого объекта `Account` выводится с использованием свойства `Balance`. При использовании `Balance` для `account1` (строка 15) значение баланса объекта `account1` возвращается `get`-методом в строке 26 на ил. 4.15 и выводится командой `Console.WriteLine` (см. ил. 4.16, строки 14–15). Аналогичным образом при вызове свойства `Balance` для объекта `account2` в строке 17 значение баланса `account2` возвращается в строке 26 на ил. 4.15 и выводится командой `Console.WriteLine` (см. ил. 4.16, строки 16–17). Баланс `account2` равен 0, потому что конструктор следит за тем, чтобы счет не начал свое существование с отрицательным балансом. Значение выводится методом `WriteLine` с форматным элементом `{0:C}`, который форматирует баланс в виде денежной суммы. Знак «:» после 0 указывает на то, что следующий символ представляет форматный спецификатор, а спецификатор `C` после двоеточия задает денежную сумму. Формат вывода денежной суммы определяется культурным контекстом на машине пользователя. Например, в США значение 50 выводится в формате `$50.00`, а в Германии оно будет выведено



в формате 50,00 €. [*Примечание:* чтобы символ € выводился правильно, выберите для окна командной строки шрифт Lucida Console.]

На ил. 4.17 перечислены другие форматные спецификаторы.

Форматный спецификатор	Описание
C или c	Строка форматируется как денежная сумма; рядом с числом выводится соответствующий знак денежной единицы. Группы разрядов разделяются соответствующим символом, а дробная часть по умолчанию состоит из двух цифр
D или d	Строка форматируется и выводится как целое число (только для целочисленных типов)
N или n	Строка форматируется с разделителями групп разрядов и двумя цифрами в дробной части
E или e	Число форматируется в экспоненциальной записи с шестью цифрами в дробной части
F или f	Строка форматируется с фиксированным количеством цифр в дробной части (два по умолчанию)
G или g	Строка форматируется с дробной частью или в экспоненциальной записи (в зависимости от контекста). Если форматный элемент не содержит спецификатора формата, неявно подразумевается формат G
X или x	Строка форматируется как шестнадцатеричное значение

**Ил. 4.17.** Форматные спецификаторы string

В строке 19 объявляется локальная переменная `depositAmount` для хранения зачисляемой суммы, введенной пользователем. В отличие от переменной экземпляра `balance` в классе `Account`, локальная переменная `depositAmount` в `Main` не инициализируется 0 по умолчанию. Кроме того, локальная переменная может использоваться только в том методе, в котором она была объявлена. Однако в данном случае переменную инициализировать не обязательно, потому что значение определяется пользовательским вводом. Компилятор не позволяет прочитать значение локальной переменной до того, как она будет инициализирована.

Строка 22 предлагает пользователю зачисляемую сумму для `account1`. Строка 23 получает данные от пользователя, вызывая метод `ReadLine` класса `Console`; введенная строка передается методу `ToDecimal` класса `Convert`, который возвращает значение типа `decimal`, представленное строкой. Строки 24–25 выводят зачисляемую сумму. Строка 26 вызывает метод `Credit` объекта `account1` и передает `depositAmount` в аргументе метода. При вызове метода значение аргумента присваивается параметру `amount` метода `Credit` (строки 16–19 на ил. 4.15), после чего метод `Credit` прибавляет это значение к `balance` (строка 18 на ил. 4.15). Строки 29–32 (см. ил. 4.16) снова выводят балансы обоих объектов `Account`, чтобы показать, что изменился только баланс `account1`.

Строка 35 предлагает пользователю ввести зачисляемую сумму для `account2`. Строка 36 получает данные у пользователя вызовом метода `Console.ReadLine`;



возвращаемое значение передается методу `ToDecimal` класса `Convert`. Строки 37–38 выводят зачисляемую сумму. Строка 39 вызывает метод `Credit` объекта `account2` с передачей `depositAmount` в аргументе метода, который прибавляется методом `Credit` к балансу. Наконец, строки 42–43 выводят балансы обоих объектов `Account`, чтобы показать, что изменился только баланс `account2`.

### Set- и get-методы с разными модификаторами доступа

По умолчанию `get-` и `set-`методы свойства обладают таким же уровнем доступа, как и само свойство, — например, для открытого свойства методы доступа также являются открытыми. Впрочем, `get-` и `set-`методы могут быть объявлены с другими модификаторами доступа. В этом случае один из методов доступа должен неявно обладать таким же уровнем доступа, как и свойство, а другой должен объявляться с более ограниченным модификатором доступа, чем у свойства. Например, в открытом свойстве `get-`метод может быть открытым, а `set-`метод — закрытым. Эта возможность будет продемонстрирована в разделе 10.7.



#### КАК ИЗБЕЖАТЬ ОШИБОК 4.2

Объявление закрытых переменных экземпляров еще не обеспечивает автоматических преимуществ целостности данных — вы должны реализовать проверку данных и сообщить об ошибках.

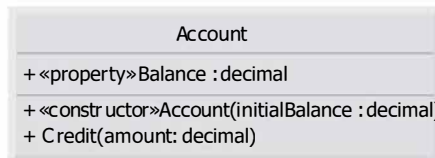


#### КАК ИЗБЕЖАТЬ ОШИБОК 4.3

`Set-`методы, задающие значения закрытых данных, должны проверить правильность новых значений; если значения неверны, `set-`методы должны оставить переменные экземпляров без изменений и сообщить об ошибке. Способы информирования об ошибках рассмотрены в главе 10.

### Диаграмма классов UML для класса Account

Диаграмма классов UML на ил. 4.18 моделирует класс `Account` из листинга на ил. 4.15. Свойство `Balance` моделируется атрибутом UML типа `decimal` (потому что соответствующее свойство `C#` относится к типу `decimal`). Конструктор класса `Account` с параметром `initialBalance` типа `decimal` моделируется в третьем отделении класса. Операция `Credit` с параметром `amount` типа `decimal` (поскольку соответствующий метод имеет параметр `amount` типа `C# decimal`) также моделируется в третьем отделении.



**Ил. 4.18.** Диаграмма классов UML показывает, что класс `Account` содержит открытое свойство `Balance` типа `decimal`, конструктор и метод

## 4.12. Итоги

В этой главе были представлены основные концепции объектно-ориентированного программирования: классы, объекты, методы, переменные экземпляров, свойства и конструкторы — все они используются в каждом сколько-нибудь содержательном приложении C#. Вы узнали, как объявлять переменные экземпляров для хранения данных каждого объекта класса, как объявлять методы для выполнения операций с данными и как объявлять свойства для чтения и записи данных. Мы показали, как выполнять операции класса посредством вызова методов и как передать методу информацию в аргументах. Мы рассмотрели различия между локальными переменными методов и переменными экземпляров (в частности, только переменные экземпляров инициализируются автоматически). Далее рассматривались темы автоматической реализации свойств и использования конструктора класса для инициализации переменных экземпляров. После обсуждения различий между значимыми и ссылочными типами были рассмотрены значимые типы `float`, `double` и `decimal` для хранения вещественных чисел.

Вы узнали, что язык UML может использоваться для создания диаграмм, моделирующих конструкторы, методы, свойства и атрибуты классов. Мы изучили, почему стоит объявлять переменные экземпляров закрытыми и использовать для работы с ними открытые свойства. Например, `set`-методы свойств могут использоваться для проверки данных объектов и обеспечения их логической целостности.

# 5

# Управляющие команды: часть 1

## 5.1. Введение

Прежде чем писать программу для решения задачи, необходимо четко понять суть задачи и тщательно спланировать ее решение. Также необходимо хорошо знать существующие структурные элементы и применять опробованные средства построения программ. В этой и в следующей главе мы рассмотрим эти вопросы в ходе знакомства с теорией и принципами структурного программирования. Представленные концепции чрезвычайно важны для эффективного построения классов и работы с объектами.

В этой главе рассматриваются команды C# `if`, `if...else` и `while` — три основные конструкции для определения логики, задействованной в выполнении операций классов. Часть этой главы (а также глав 6 и 8) будет посвящена дальнейшей разработке класса `GradeBook`. В частности, мы добавим в класс `GradeBook` функцию для вычисления средней оценки. Другой пример демонстрирует дополнительные способы объединения управляющих команд. Также будут представлены операторы инкремента и декремента — эти операторы C# сокращают и упрощают многие команды.

## 5.2. Алгоритмы

Любая вычислительная задача может быть решена выполнением последовательности действий в определенном порядке. Процедура решения задачи с определением:

- 1) выполняемых действий;
- 2) порядка их выполнения

называется *алгоритмом*. Для управления порядком выполнения в C# используются *управляющие команды*.

## 5.3. Псевдокод

*Псевдокодом* называется неформальный язык, который упрощает разработку приложений, позволяя не отвлекаться на технические подробности синтаксиса C#. Псевдокод хорошо подходит для разработки алгоритмов, которые преобразуются в структурированные части приложений C#. Псевдокод близок к естественным языкам — он не является языком программирования.

Псевдокод не выполняется на компьютерах. Скорее, он помогает «продумать» приложение, прежде чем пытаться записывать его на C#. В этой главе вы увидите, как использовать псевдокод при разработке приложений C#.

Псевдокод можно создавать в любом текстовом редакторе. Хорошо подготовленный псевдокод легко преобразуется в соответствующее приложение C#. Во многих случаях для этого достаточно заменить команды псевдокода эквивалентными конструкциями C#.

Псевдокод обычно описывает только действия: ввод, вывод, вычисления. Мы не включаем в свой псевдокод объявления переменных, хотя некоторые программисты это делают.

## 5.4. Управляющие структуры

Обычно команды выполняются одна за другой в порядке их записи. Этот процесс называется последовательным выполнением. В C# имеются команды, позволяющие выбрать для выполнения другую команду вместо следующей команды в последовательности. Это называется *передачей управления*.

В 1960-е годы стало очевидно, что беспорядочная передача управления стала причиной многих сложностей в группах разработки. Основная вина возлагалась на команду `goto` (существовавшую в большинстве языков программирования того времени), позволявшую передать управление в одну из многих возможных точек приложения; так возникал печально известный «спагетти-код». Понятие структурного программирования стало почти равнозначно «отказу от `goto`». Мы не рекомендуем использовать команду `goto` в C#.

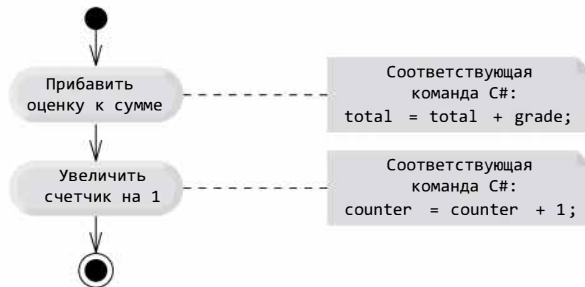
Исследования<sup>1</sup> показали, что приложения можно писать и без команд `goto`. Одним из главных испытаний для программистов того времени стал переход на «программирование без `goto`». Только в 1970-х годах программисты стали серьезно воспринимать структурное программирование. Результаты оказались впечатляющими: структурированные приложения были более понятными, простыми в отладке и сопровождении и изначально содержали меньше ошибок.

<sup>1</sup> Böhm, C., and G. Jacopini, «Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules», Communications of the ACM, Vol. 9, No. 5, May 1966, p. 366–371.

Работа Бема и Якопини продемонстрировала, что любое приложение может быть написано с использованием всего трех управляющих структур — структуры последовательности, структуры выбора и структуры повторения. Говоря о реализации этих управляющих структур в C#, мы будем использовать термин «управляющие структуры» из спецификации языка C#.

### Структура последовательности в C#

Структура последовательности встроена в язык C#. Если компьютер не получит иных распоряжений, он выполняет команды C# одну за другой в порядке их записи — то есть последовательно. Диаграмма активности UML на ил. 5.1 показывает типичную структуру последовательности с выполнением двух вычислений в заданном порядке. C# позволяет создавать последовательности из произвольного количества действий.



**Ил. 5.1.** Диаграмма структуры последовательности

*Диаграмма активности* моделирует *поток операций* (также называемый активностью) части программной системы. Такие потоки операций могут включать часть алгоритма — например, структуру последовательности на ил. 5.1. Диаграммы активности строятся из специальных условных обозначений — блоков состояния действий (прямоугольники, у которых левая и правая сторона заменены дугами), ромбов и кружков. Эти знаки соединяются стрелками переходов, представляющими порядок выполнения действий.

Диаграммы активности, как и псевдокод, упрощают разработку и представление алгоритмов, хотя большинство программистов предпочитает работать с псевдокодом. По диаграммам активности хорошо видно, как работают управляющие структуры.

Возьмем диаграмму активности для структуры последовательности на ил. 5.1. Она содержит два *состояния действий*, представляющих выполняемые действия. Каждое состояние содержит выражение действия (например, «прибавить оценку к сумме» или «увеличить счетчик на 1»), определяющее выполняемое действие. Другие действия могут включать вычисления или операции ввода-вывода. Стрелки на диаграмме активности представляют переходы, которые обозначают порядок выполнения действий. Часть приложения, реализующая действия на диаграмме, сначала прибавляет `grade` к `total`, а затем увеличивает `counter` на 1.

Черный кружок в верхней части диаграммы представляет исходное состояние приложения — начало потока операций до выполнения приложением моделируемых действий. Черный кружок в белом кружке в нижней части диаграммы представляет конечное состояние потока операций после того, как приложение выполнит свои действия.

На ил. 5.1 также изображены прямоугольники с загнутым правым верхним углом. Это примечания UML (аналоги комментариев C#), описывающие смысл условных обозначений на диаграмме. На ил. 5.1 в примечаниях UML приводится код C#, связанный с каждым состоянием действия на диаграмме активности. Каждое примечание соединяется пунктирной линией с элементом, описываемым этим примечанием. Обычно на диаграммах активности код реализации C# не выводится. Мы делаем это только для того, чтобы показать, как диаграмма связана с кодом C#.

### Структуры выбора в C#

В C# существуют три разновидности структур выбора, которые в дальнейшем мы будем называть «командами выбора». Команда `if` либо выполняет (выбирает) действие при истинности заданного условия, либо пропускает действие, если условие ложно. Команда `if...else` либо выполняет действие при истинности заданного условия, либо выполняет другое действие, если условие ложно. Команда `switch` (см. главу 6) выполняет одно из нескольких действий в зависимости от значения выражения.

Команда `if` называется *командой одиночного выбора*, потому что она выбирает или игнорирует одно действие (или, как мы вскоре увидим, одну группу действий). Команда `if...else` называется *командой двойного выбора*, потому что она выбирает между двумя разными действиями (или группами действий). Команда `switch` называется *командой множественного выбора*, потому что она выбирает одно из многих различных действий (или групп действий).

### Структуры повторения в C#

В C# поддерживаются четыре структуры повторения, которые в дальнейшем будут называться *командами повторения* (а также *итеративными командами* или *циклами*). Команды повторения позволяют приложениям многократно выполнять другие команды (в зависимости от значения условия продолжения цикла). C# предоставляет четыре команды повторения: `while`, `do...while`, `for` и `foreach`. (Команды `do...while` и `for` описаны в главе 6, а команда `foreach` — в главе 8.) Команды `while`, `for` и `foreach` выполняют действие (или группу действий), содержащееся в их теле, нуль и более раз — если условие продолжения цикла изначально ложно, то действие (или группа действий) не выполняется. Команда `do...while` выполняет действие (или группу действий), содержащееся в их теле, один и более раз. Слова `if`, `else`, `switch`, `while`, `do`, `for` и `foreach` являются ключевыми словами C#.

### Сводка управляющих команд в C#

В C# существуют всего три разновидности структурных управляющих команд: команда последовательности, команда выбора (три типа) и команда повторения

(четыре типа). Объединяя нужное количество команд каждого типа, мы заставляем команды своего приложения выполняться в нужном порядке. Каждая управляющая команда может быть смоделирована диаграммой активности, как на ил. 5.1. Каждая диаграмма состоит из одного исходного состояния и одного конечного состояния, которые представляют соответственно точку входа и точку выхода управляющей команды. Управляющие команды с одним входом и одним выходом упрощают построение приложений — они «сцепляются» друг с другом так, что точка выхода одной команды становится точкой входа другой. Кроме сцепления, управляющие команды также могут объединяться посредством *вложения*, то есть одна управляющая команда размещается внутри другой. Таким образом, алгоритмы в C# строятся всего из трех разновидностей структурных управляющих команд, объединяемых всего двумя способами. Проще не придумаешь!

## 5.5. Команда одиночного выбора if

Приложения используют команды выбора для принятия решений из нескольких альтернативных вариантов. Предположим, экзамен считается сданным, если студент наберет 60 баллов. Следующий фрагмент на псевдокоде определяет истинность или ложность условия «количество баллов больше либо равно 60»:

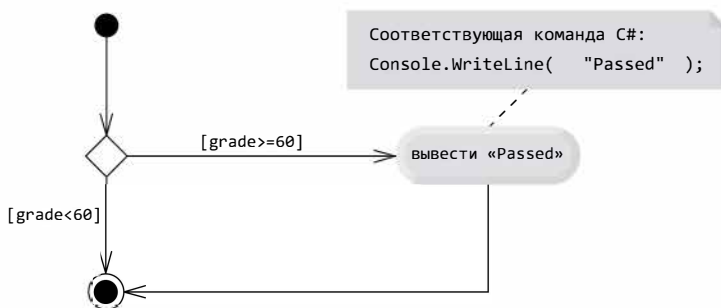
```
if количество_баллов больше либо равно 60
    вывести "Passed"
```

Если условие истинно, выводится строка "Passed", и «выполняется» следующая по порядку команда псевдокода. (Помните, что псевдокод не является настоящим языком программирования.) Если условие ложно, то команда вывода игнорируется, и выполняется следующая по порядку команда псевдокода. Отступ второй строки в команде выбора не обязателен, но желателен, поскольку он подчеркивает структуру кода.

Приведенная команда if может быть записана на C# в следующем виде:

```
if ( grade >= 60 )
    Console.WriteLine( "Passed" );
```

На ил. 5.2 изображена команда одиночного выбора if. На этой диаграмме активности присутствует, пожалуй, самое важное условное обозначение на диаграммах активности — ромб, который означает, что в этой точке принимается решение. Выполнение приложения продолжается по пути, определяемому одним из двух условий — одно условие должно быть истинным, а другое ложным. Каждая стрелка перехода, выходящая из ромба, помечается условием (оно задается в квадратных скобках рядом со стрелкой). Если условие истинно, то поток операций входит в состояние, на которое указывает эта стрелка. На ил. 5.2, если значение `grade` больше или равно 60, приложение выводит строку "Passed" и переходит в конечное состояние этой активности. Если значение `grade` меньше 60, то приложение немедленно переходит в конечное состояние без вывода сообщения.



**Ил. 5.2.** Диаграмма активности UML для команды одиночного выбора if

Команда `if` относится к категории управляющих команд с одним входом и одним выходом. Вскоре вы увидите, что диаграммы активности для других управляющих команд также содержат исходные состояния, стрелки переходов, выполняемые действия, блоки принятия решений (с соответствующими сторожевыми состояниями) и конечные состояния. Все это соответствует модели программирования, основанной на действиях и принятии решений, на которую мы ориентируемся.

Представьте восемь коробок, в каждой из которых хранится один тип управляющих команд C#. Ваша задача — собрать приложение из управляющих команд, необходимых для реализации алгоритма, с объединением их только двумя возможными способами (вложение или сцепление) и последующим заполнением состояний приложения и блоков принятия решений выражениями и условиями, соответствующими алгоритму. Разнообразные способы записи действий и решений будут подробно рассмотрены далее.

## 5.6. Команда двойного выбора `if...else`

Команда одиночного выбора `if` выполняет указанное действие только в том случае, если условие истинно; в противном случае действие пропускается. Команда двойного выбора `if...else` позволяет задать два действия: одно выполняется, если условие истинно, а другое — если оно ложно. Например, следующий фрагмент на псевдокоде выводит сообщение «Passed», если количество баллов больше или равно 60, и сообщение «Failed» в противном случае. В любом случае после вывода сообщения «выполняется» следующая команда псевдокода.

```
if количество_баллов больше либо равно 60
    вывести "Passed"
else
    вывести "Failed"
```

Приведенный фрагмент псевдокода `if...else` может быть записан на C# в следующем виде:



```

if ( grade >= 60 )
    Console.WriteLine( "Passed" );
else
    Console.WriteLine( "Failed" );

```

Тело секции else также снабжено отступом. Какую бы схему обозначений вы ни выбрали, последовательно применяйте ее в своих приложениях. Код, не соблюдающий единые правила форматирования, трудно читать.



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 5.1

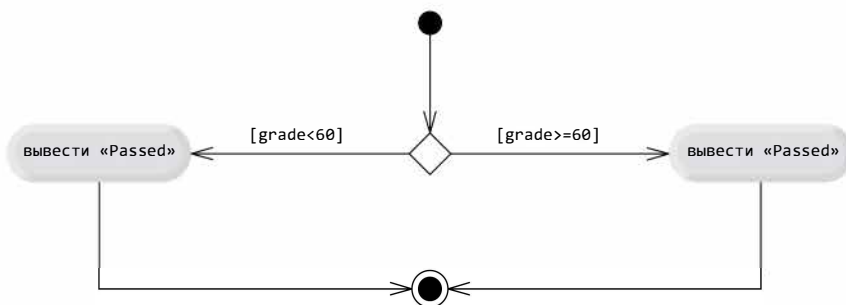
Снабдите отступами оба тела команды if...else.



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 5.2

Если в программе используются многоуровневые отступы, величины отступов на всех уровнях должны быть обязательными.

На ил. 5.3 представлена логика команды if...else. Снова представьте большую коробку с пустыми командами if...else. Ваша задача — объединить эти команды if...else (посредством вложения и сцепления) с другими управляющими командами, необходимыми для алгоритма. Вы заполняете состояния действий и блоки принятия решений выражениями и условиями, соответствующими алгоритму.



Ил. 5.3. Диаграмма активности UML для команды двойного выбора if...else

### Условный оператор (?)

В С# также существует условный оператор (? :), который может использоваться вместо команды if...else. Это единственный *тернарный* оператор С# (то есть оператор, получающий три операнда). Операнды совместно со знаками ? : образуют условное выражение. Первый операнд (слева от ?) представляет собой логическое выражение (то есть выражение, результат которого относится к логическому типу — true или false), второй (между ? и :) определяет значение условного выражения, если логическое выражение истинно, а третий операнд (справа от :) определяет значение условного выражения, если логическое выражение ложно. Например, команда

```

Console.WriteLine( grade >= 60 ? "Passed" : "Failed" );

```

выводит значение условного выражения, передаваемого в аргументе `writeLine`. Если логическое выражение `grade >= 60` истинно, то результат вычисления выражения равен "Passed", а если ложно — "Failed". Таким образом, команда с условным оператором выполняет практически те же функции, что и команда `if...else`, приведенная ранее в этом разделе. Вычисление условия каждой управляющей команды должно давать логический результат `true` или `false`. Как вы вскоре увидите, условные выражения могут использоваться в некоторых ситуациях, недоступных для `if...else`.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 5.3

Когда условное выражение используется внутри большего выражения, его стоит заключить в круглые скобки для ясности. Добавление круглых скобок также может предотвратить проблемы с приоритетом, которые становятся причиной синтаксических ошибок.

### Вложенные команды `if...else`

Приложение может проверить несколько условий подряд; для этого команды `if...else` вкладываются друг в друга, образуя вложенные конструкции `if...else`. Например, следующий псевдокод представляет вложенные команды `if...else`, которые выводят A, если количество баллов больше или равно 90, B — для количества баллов в диапазоне от 80 до 89, C — для диапазона от 70 до 79, D — для диапазона от 60 до 69 и F для всех остальных значений:

```
if grade больше либо равно 90
    вывести "A"
else
    if grade больше либо равно 80
        вывести "B"
    else
        if grade больше либо равно 70
            вывести "C"
        else
            if grade больше либо равно 60
                вывести "D"
            else
                вывести "F"
```

Этот псевдокод записывается на C# в следующем виде:

```
if ( grade >= 90 )
    Console.WriteLine( "A" );
else
    if ( grade >= 80 )
        Console.WriteLine( "B" );
    else
        if ( grade >= 70 )
            Console.WriteLine( "C" );
        else
            if ( grade >= 60 )
                Console.WriteLine( "D" );
            else
                Console.WriteLine( "F" );
```

Если значение `grade` больше или равно 90, первые четыре условия будут истинны, но выполнится только команда в части `if` первой команды `if...else`. После выполнения этой команды часть `else` «внешней» команды `if...else` пропускается. Обычно программисты C# предпочитают записывать приведенную команду `if...else` в следующем виде:

```
if ( grade >= 90 )
    Console.WriteLine( "A" );
else if ( grade >= 80 )
    Console.WriteLine( "B" );
else if ( grade >= 70 )
    Console.WriteLine( "C" );
else if ( grade >= 60 )
    Console.WriteLine( "D" );
else
    Console.WriteLine( "F" );
```

Эти две формы идентичны, не считая отступов и пробелов, которые игнорируются компилятором. Вторая форма более популярна, потому что в ней нет больших отступов со сдвигом вправо — с такими отступами в строке часто не остается свободного места, из-за чего строки приходится разбивать, а код становится менее понятным.

### Проблема «неопределенного else»

Компилятор C# всегда связывает ключевое слово `else` с непосредственно предшествующей командой `if`, если только ему явно не приказать обратное при помощи фигурных скобок (`{}` и `}`). Такое поведение может привести к проблеме «неопределенного else». Например:

```
if ( x > 5 )
    if ( y > 5 )
        Console.WriteLine( "x and y are > 5" );
else
    Console.WriteLine( "x is <= 5" );
```

Казалось бы, этот фрагмент означает следующее: если `x` больше 5, вложенная команда `if` проверяет, что значение `y` также больше 5. Если это условие выполняется, то выводится строка `"x and y are > 5"`. В противном случае, если `x` не больше 5, часть `else` команды `if...else` выводит строку `"x is <= 5"`.

Осторожно! Вложенная команда `if...else` работает не так, как кажется на первый взгляд. Компилятор в действительности интерпретирует команду следующим образом:

```
if ( x > 5 )
    if ( y > 5 )
        Console.WriteLine( "x and y are > 5" );
else
    Console.WriteLine( "x is <= 5" );
```

То есть тело первой команды `if` интерпретируется как вложенная команда `if...else`. Внешняя команда `if` проверяет, что значение `x` больше 5. В этом случае выполнение

продолжается проверкой того, что `y` тоже больше 5. Если второе условие истинно, выводится правильная строка `"x and y are > 5"`. Но если второе условие ложно, выводится строка `"x is <= 5"`, хотя мы знаем, что значение `x` больше 5.

Чтобы вложенная команда `if...else` выполнялась так, как положено, ее необходимо записать следующим образом:

```
if ( x > 5 )
{
    if ( y > 5 )
        Console.WriteLine( "x and y are > 5" );
}
else
    Console.WriteLine( "x is <= 5" );
```

Фигурные скобки (`{}`) сообщают компилятору, что вторая команда `if` находится в теле первой команды `if`, а `else` относится к первой команде `if`.

### Блоки

Команда `if` предполагает, что ее тело состоит только из одной команды. Чтобы включить в тело `if` (или тело `else` в команде `if...else`) несколько команд, следует заключить их в фигурные скобки. Набор команд, заключенных в фигурные скобки, называется блоком. Блок может размещаться в любом месте приложения, где может находиться одна команда.

Следующий пример включает блок в секции `else` команды `if...else`:

```
if ( grade >= 60 )
    Console.WriteLine( "Passed" );
else
{
    Console.WriteLine( "Failed" );
    Console.WriteLine( "You must take this course again." );
}
```

Если `grade` меньше 60, приложение выполняет обе команды в теле `else` и выводит следующий текст:

```
Failed.
You must take this course again.
```

Обратите внимание на фигурные скобки, в которые заключены две команды в секции `else`. Эти фигурные скобки важны; без них команда

```
Console.WriteLine( "You must take this course again." );
```

окажется за пределами тела `else` команды `if...else` и будет выполняться независимо от того, меньше ли `grade` или нет.

Компилятор обнаруживает синтаксические ошибки. Логическая ошибка (например, если пропущены обе фигурные скобки в блоке) проявляется во время выполнения. Фатальная логическая ошибка может привести к сбою и преждевременному завершению приложения. При нефатальной логической ошибке выполнение приложения продолжается, но результаты выполнения могут оказаться неправильными.



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 5.4

Некоторые программисты всегда используют фигурные скобки в `if...else` (и других командах), чтобы случайно не забыть их — особенно если в будущем в часть `if` или в часть `else` будут добавляться команды. Некоторые программисты вводят начальную и конечную фигурные скобки еще до того, как вводить содержащиеся в скобках команды.

Блок также может содержать *пустую команду*. Как говорилось в разделе 3.9, для представления пустой команды следует поставить точку с запятой (;) там, где должна находиться команда.



#### ТИПИЧНАЯ ОШИБКА 5.1

Точка с запятой после условия в команде `if` или `if...else` приводит к логической ошибке в команде одиночного выбора `if` и к синтаксической ошибке в команде двойного выбора `if...else`.

## 5.7. Цикл while

Команда повторения указывает, что действие должно повторяться до тех пор, пока некоторое условие остается истинным. Команда на псевдокоде

```
while в списке остаются покупки
    купить следующий товар и вычеркнуть из списка
```

описывает повторение, происходящее во время похода в магазин. Условие «в списке остаются покупки» может быть истинным или ложным. Если условие истинно, то выполняется действие «купить следующий товар и вычеркнуть из списка». Это действие выполняется многократно, пока условие остается истинным. Тело цикла `while` может содержать одну команду или блок. В какой-то момент условие становится ложным (когда последняя покупка из списка будет оформлена и вычеркнута из списка). При этом повторение прекращается и выполняется первая команда после команды повторения.

В качестве примера использования команды C# `while` рассмотрим фрагмент кода для вычисления первой степени 3, превышающей 100. После завершения следующей команды `while` переменная `product` содержит результат:

```
int product = 3;
while ( product <= 100 )
    product = 3 * product;
```

Когда эта команда `while` начинает выполняться, значение переменной `product` равно 3. При каждом повторении тела `while` значение `product` умножается на 3, так что `product` последовательно принимает значения 9, 27, 81 и 243. Когда переменная `product` становится равной 243, условие `while (product <= 100)` становится ложным. На этом повторение завершается, так что итоговое значение `product` равно 243. В этой точке управление передается следующей команде после `while`.

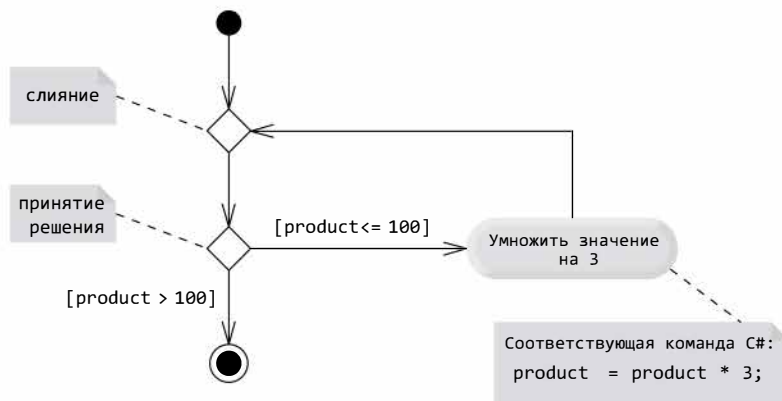


### ТИПИЧНАЯ ОШИБКА 5.2

Если в теле команды `while` не выполняется действие, из-за которого условие цикла в какой-то момент станет ложным, это приводит к логической ошибке, называемой «зацикливанием» (то есть цикл выполняется бесконечно и никогда не завершается).

### Диаграмма активности команды `while`

Диаграмма активности на ил. 5.4 демонстрирует поток операций для приведенной команды `while`. На ней также представлено условное обозначение *слияния*. В UML и принятие решений, и слияние обозначаются ромбами. В точке слияния два потока активности объединяются в один поток. На диаграмме условное обозначение слияния объединяет переходы из исходного состояния и состояния действия, так что оба переходят в принятие решения о продолжении (или начале) цикла. Условные обозначения принятия решений и слияния можно отличить по количеству «входящих» и «исходящих» стрелок переходов. У принятия решения одна стрелка перехода входит в ромб, а две и более стрелки выходят из него; с каждой выходящей стрелкой связывается условие. При слиянии две и более стрелки входят в ромб, и только одна стрелка выходит из него. Ни одна из стрелок переходов, связанных со слиянием, условия не имеет.



**Ил. 5.4.** Диаграмма активности UML для команды повторения `while`

На ил. 5.4 четко обозначена логика команды `while`, рассмотренной в этом разделе. Стрелка перехода от состояния действия возвращается к точке слияния, из которой происходит переход обратно к принятию решения в начале каждого повторения цикла. Цикл продолжает выполняться до тех пор, пока условие `product > 100` не станет истинным. Затем команда `while` завершается (достигает своего конечного состояния), а управление передается следующей команде приложения.

## 5.8. Формулировка алгоритмов: ЦИКЛ СО СЧЕТЧИКОМ

Чтобы продемонстрировать, как разрабатываются алгоритмы, мы изменим класс GradeBook из главы 4 и создадим две вариации на тему задачи усреднения оценок. Возьмем следующую постановку задачи:

*Группа из 10 студентов написала контрольную работу. Вам известны полученные ими оценки (целые числа в диапазоне от 0 до 100). Вычислите среднюю оценку группы.*

Средняя оценка группы равна сумме оценок, разделенной на количество студентов. Алгоритм решения этой задачи на компьютере должен ввести все оценки, вычислить сумму всех оценок, вычислить среднее значение и вывести результат.

Мы используем псевдокод для составления списка выполняемых действий и порядка их выполнения. Для последовательного ввода оценок будет использоваться цикл со счетчиком. В этой разновидности циклов количество выполнений команды определяется переменной, которая называется *счетчиком* (также используется термин «управляющая переменная»). Такие циклические конструкции часто называются *детерминированными*, поскольку количество итераций (повторений) известно до начала цикла. В данном примере цикл завершается, когда счетчик достигает 10. В этом разделе представлен готовый алгоритм на псевдокоде (ил. 5.5) и версия класса GradeBook (ил. 5.6), реализующая алгоритм в методе C#. Далее будет представлено приложение (ил. 5.7), демонстрирующее алгоритм в действии. В разделе 5.9 будет показано, как при помощи псевдокода разработать такой алгоритм «с нуля».

```
1  присвоить сумме 0
2  присвоить счетчику 1
3
4  пока значение счетчика меньше либо равно 10
5      запросить следующую оценку у пользователя
6      ввести следующую оценку
7      прибавить оценку к сумме
8      увеличить счетчик на 1
9
10 задать средней оценке результат деления суммы на 10
11 вывести среднюю оценку
```

**Ил. 5.5.** Алгоритм на псевдокоде, использующий цикл со счетчиком для вычисления средней оценки



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 5.1

Опыт показывает, что самой сложной частью решения задачи на компьютере является разработка алгоритма решения. После того как алгоритм будет сформулирован, процесс создания работоспособного приложения C# обычно весьма прямолинеен.

Обратите внимание на упоминания «суммы» и «счетчика» в алгоритме на ил. 5.5. Сумма — переменная, используемая для вычисления суммы нескольких значений. Счетчик — переменная, используемая для отсчета (в данном случае для отсчета 10 оценок, введенных пользователем). Переменные, используемые для вычисления суммы, обычно инициализируются нулями перед их использованием в приложениях.

### Реализация цикла со счетчиком в классе GradeBook

Класс GradeBook (см. ил. 5.6) содержит конструктор (строки 12–15), который присваивает значение переменной экземпляра, созданной автоматически реализуемым свойством CourseName в строке 9. В строках 18–23 объявляется метод DisplayMessage. В строках 26–52 объявляется метод DetermineClassAverage, который реализует алгоритм вычисления средней оценки из ил. 5.5.

```
1 // Ил. 5.6: GradeBook.cs
2 // Класс GradeBook, решающий задачу вычисления средней оценки
3 // с использованием цикла со счетчиком.
4 using System;
5
6 public class GradeBook
7 {
8     // Автоматически реализуемое свойство CourseName
9     public string CourseName { get; set; }
10
11     // Конструктор инициализирует свойство CourseName
12     public GradeBook( string name )
13     {
14         CourseName = name; // Присвоить name свойству CourseName
15     } // Конец конструктора
16
17     // Вывод приветствия для пользователя GradeBook
18     public void DisplayMessage()
19     {
20         // Свойство CourseName возвращает название курса
21         Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
22             CourseName );
23     } // Конец метода DisplayMessage
24
25     // Вычисление средней оценки для 10 оценок, введенных пользователем
26     public void DetermineClassAverage()
27     {
28         int total; // Сумма оценок, введенных пользователем
29         int gradeCounter; // Количество вводимых оценок
30         int grade; // Оценка, вводимая пользователем
31         int average; // Среднее арифметическое оценок
32
33         // Фаза инициализации
34         total = 0; // Инициализация суммы
35         gradeCounter = 1; // Инициализация счетчика цикла
36
37         // Фаза выполнения
38         while ( gradeCounter <= 10 ) // Выполнить 10 раз
```

**Ил. 5.6.** Класс GradeBook для вычисления средней оценки с использованием цикла со счетчиком (продолжение ➤)



```
39     {
40         Console.Write( "Enter grade: " ); // Запрос данных
41         grade = Convert.ToInt32( Console.ReadLine() ); // Чтение оценки
42         total = total + grade; // Прибавление оценки к сумме
43         gradeCounter = gradeCounter + 1; // Увеличение счетчика на 1
44     } // Конец while
45
46     // Завершающая фаза
47     average = total / 10; // Деление дает целочисленный результат
48
49     // Вывод суммы и средней оценки
50     Console.WriteLine( "\nTotal of all 10 grades is {0}", total );
51     Console.WriteLine( "Class average is {0}", average );
52 } // Конец метода DetermineClassAverage
53 } // Конец класса GradeBook
```

**Ил. 5.6.** Класс GradeBook для вычисления средней оценки  
с использованием цикла со счетчиком (окончание)

### Метод DetermineClassAverage

В строках 28–31 объявляются локальные переменные `total`, `gradeCounter`, `grade` и `average` типа `int`. В приведенном примере в переменной `total` накапливается сумма введенных оценок, а переменная `gradeCounter` подсчитывает количество введенных оценок. В переменной `grade` хранится последняя введенная оценка (строка 41). В переменной `average` хранится средняя оценка.

Объявления (строки 28–31) находятся в теле метода `DetermineClassAverage`. Переменные, объявленные в теле метода, являются локальными и могут использоваться только в промежутке от места объявления до закрывающей фигурной скобки блока, в котором они были объявлены. Объявление локальной переменной должно предшествовать ее использованию в методе. Локальная переменная недоступна за пределами блока, в котором она объявляется.

В версиях класса `GradeBook` из этой главы мы просто читаем и обрабатываем набор оценок. Средняя оценка вычисляется в методе `DetermineClassAverage` с использованием локальных переменных — информация об оценках не сохраняется в переменных экземпляров. В последующих версиях класса (глава 8) оценки будут сохраняться в переменной экземпляра, которая содержит ссылку на структуру данных, называемую массивом. Это позволит объекту `GradeBook` выполнять различные вычисления с одним набором оценок, не заставляя пользователя каждый раз вводить данные заново.



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 5.5

Отделяйте объявления от других команд метода пустой строкой для удобства чтения.

Переменная называется *безусловно инициализированной*, если перед использованием ей гарантированно было присвоено значение. Обратите внимание: все локальные переменные, объявленные в строках 28–31, являются безусловно

инициализированными. Команды присваивания (строки 34–35) инициализируют `total` значением 0, а `gradeCounter` — значением 1. Переменные `grade` и `average` (для пользовательского ввода и вычисленной средней оценки соответственно) здесь инициализировать не обязательно; их значения присваиваются при вводе или вычислении позже в методе.



### ТИПИЧНАЯ ОШИБКА 5.3

Попытка использования локальной переменной до того, как ей будет присвоено значение, приводит к ошибке компиляции. Все локальные переменные должны быть инициализированы перед использованием в выражениях.



### КАК ИЗБЕЖАТЬ ОШИБОК 5.1

Инициализируйте все переменные счетчиков и накапливаемых сумм — в объявлениях или в командах присваивания. Суммы обычно инициализируются 0. Счетчики обычно инициализируются 0 или 1 в зависимости от того, как они будут использоваться (позже мы рассмотрим примеры обоих видов).

В строке 38 указано, что команда `while` должна продолжать цикл, пока значение `gradeCounter` меньше либо равно 10. Пока это условие остается истинным, команда `while` многократно выполняет команды в круглых скобках, в которые заключено ее тело (строки 39–44).

Строка 40 выводит запрос "Enter grade: " в консольном окне. Строка 41 читает оценку, введенную пользователем, и присваивает ее переменной `grade`. Затем в строке 42 новая оценка, введенная пользователем, прибавляется к сумме, а результат присваивается переменной `total`, заменяя ее предыдущее значение.

В строке 43 значение `gradeCounter` увеличивается на 1; это означает, что приложение обработало оценку и готово к получению следующей оценки от пользователя. Из-за увеличения `gradeCounter` в какой-то момент значение `gradeCounter` превысит 10. В этой точке цикл `while` завершается, потому что его условие (строка 38) становится ложным.

При завершении цикла строка 47 вычисляет среднее значение и присваивает результат переменной `average`. В строке 50 метод `writeLine` класса `Console` используется для вывода строки "Total of all 10 grades is ", за которой следует значение переменной `total`. Затем строка 51 выводит строку "Class average is " и значение переменной `average`. При достижении строки 52 метод `DetermineClassAverage` возвращает управление вызывающему методу (то есть `Main` в `GradeBookTest` на ил. 5.7).

## Класс `GradeBookTest`

Класс `GradeBookTest` (см. ил. 5.7) создает объект класса `GradeBook` (см. ил. 5.6) и демонстрирует его возможности. В строках 9–10 на ил. 5.7 создается новый объект `GradeBook`, который присваивается переменной `myGradeBook`. Конструктору `GradeBook` передается строковое значение (строки 12–15 на ил. 5.6). Строка 12 (см. ил. 5.7) выводит приветствие для пользователя, вызывая метод `DisplayMessage` для объекта

myGradeBook. Затем строка 13 вызывает метод `DetermineClassAverage` для объекта `myGradeBook`, позволяя пользователю ввести 10 оценок, для которых метод затем вычисляет и выводит среднее значение; при этом используется алгоритм, показанный на ил. 5.5.

```

1  // Ил. 5.7: GradeBookTest.cs
2  // Создание объекта GradeBook и вызов его метода DetermineClassAverage
3  public class GradeBookTest
4  {
5      public static void Main( string[] args )
6      {
7          // Создание объекта myGradeBook класса GradeBook
8          // и передача названия курса конструктору
9          GradeBook myGradeBook = new GradeBook(
10             "CS101 Introduction to C# Programming" );
11
12         myGradeBook.DisplayMessage(); // Вывод приветствия
13         myGradeBook.DetermineClassAverage(); // Среднее по 10 оценкам
14     } // Конец Main
15 } // Конец класса GradeBookTest

```

```

Welcome to the grade book for
CS101 Introduction to C# Programming!

```

```

Enter grade: 88
Enter grade: 79
Enter grade: 95
Enter grade: 100
Enter grade: 48
Enter grade: 88
Enter grade: 92
Enter grade: 83
Enter grade: 90
Enter grade: 85

```

```

Total of all 10 grades is 848
Class average is 84

```

**Ил. 5.7.** Создание объекта `GradeBook` и вызов его метода `DetermineClassAverage`

### Целочисленное деление и округление

Вычисление средней оценки, выполняемое методом `DetermineClassAverage` по вызову метода в строке 13 на ил. 5.7, возвращает целочисленный результат. В выходных данных приложения указано, что сумма оценок для тестового запуска составляет 848, что при делении на 10 должно давать число 84,8. Однако результат вычисления `total/10` (строка 47 на ил. 5.6) равен целому числу 84, потому что и `total`, и 10 являются целыми числами.

Результат деления двух целых чисел является целым числом — дробная часть теряется (то есть выполняется усечение, а не округление). В следующем разделе вы увидите, как получить результат с плавающей точкой при вычислении среднего.

**ТИПИЧНАЯ ОШИБКА 5.4**

Предположение о том, что при целочисленном делении результат округляется (вместо простого усечения дробной части), может привести к неверным результатам. Например, результат операции  $7 \div 4$ , который в традиционной арифметике равен 1,75, в целочисленной арифметике усекается до 1 (вместо округления до 2).

## 5.9. Формулировка алгоритмов: цикл со сторожевым значением

Давайте обобщим задачу вычисления средней оценки из раздела 5.8. Допустим, наше приложение при каждом запуске должно вычислять среднюю оценку для произвольного количества студентов.

В постановке предыдущей задачи было задано количество студентов, так что количество оценок (10) было известно заранее. На этот раз мы не знаем, сколько оценок введет пользователь при запуске. Приложение должно обработать *произвольное* количество оценок. Как определить, когда следует остановить ввод оценок? Как приложение узнает, что нужно вычислить и вывести среднюю оценку?

Одно из решений подобных задач основано на использовании специального значения, называемого *сторожевым значением* (также используются термины «сигнальное значение» или «флаговое значение»), которое обозначает «конец ввода данных». Пользователь вводит оценки до тех пор, пока не будут введены все данные, после чего вводит сторожевое значение, означающее, что оценок больше нет. Циклы со сторожевым условием часто называются *неопределенными*, потому что количество повторений неизвестно до начала выполнения цикла.

Очевидно, что сторожевое значение должно выбираться таким образом, чтобы оно не путалось с допустимыми входными значениями. Оценки являются неотрицательными целыми числами, так что значение  $-1$  можно считать приемлемым сторожевым значением для этой задачи. Таким образом, приложение при запуске может получить поток значений вида 95, 96, 75, 74, 89 и  $-1$ . Далее приложение вычисляет и выводит среднюю оценку для значений 95, 96, 75, 74 и 89. Сторожевое значение  $-1$  не должно учитываться в вычислениях.

**ТИПИЧНАЯ ОШИБКА 5.5**

Выбор сторожевого значения, которое может оказаться допустимым значением данных, является логической ошибкой.

### Разработка алгоритма на псевдокоде методом нисходящего пошагового уточнения

Для создания приложения мы воспользуемся методом нисходящего пошагового уточнения, играющим важную роль в разработке структурированных приложений.

Начнем с псевдокода верхнего уровня — одного утверждения, описывающего общую задачу приложения:

*вычислить среднюю оценку по результатам контрольной.*

По сути эта формулировка полностью описывает приложения. К сожалению, такие «высокоуровневые» утверждения редко содержат достаточно подробностей для написания приложения С#, поэтому мы перейдем к процессу уточнения. Формулировка верхнего уровня делится на несколько меньших задач, которые перечисляются в порядке их выполнения. Результаты первой фазы уточнения:

*инициализировать переменные  
ввести, сложить и подсчитать оценки  
вычислить и вывести среднюю оценку*

В уточнении используется только последовательная структура — перечисленные шаги выполняются по порядку, друг за другом.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 5.2

Каждое уточнение, равно как и формулировка верхнего уровня, является полным описанием алгоритма — изменяется только уровень детализации.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 5.3

Многие приложения можно разделить на три логические фазы. В фазе инициализации происходит инициализация переменных; в фазе выполнения приложение получает данные и вносит необходимые изменения в переменные (счетчики и накапливаемые суммы); в фазе завершения происходит вычисление и вывод результатов.

### Второе уточнение

Нередко схема из Архитектурного решения 5.3 полностью определяет первое уточнение в нисходящем процессе. При переходе на следующий уровень (второе уточнение) задаются значения отдельных переменных. В данном примере нам понадобятся переменные для накапливаемой суммы, счетчик обработанных чисел, переменная для хранения текущей введенной оценки и переменная для хранения среднего. Псевдокод

*инициализировать переменные*

уточняется до следующего вида:

*инициализировать сумму 0  
инициализировать счетчик 0*

Только переменные суммы (`total`) и счетчика (`counter`) должны инициализироваться перед использованием. Переменные `average` и `grade` (вычисленное среднее и введенная оценка соответственно) инициализировать не нужно, потому что их значения все равно будут заменены при вычислении или вводе.

В реализации псевдокода

*ввести, сложить и подсчитать оценки*

необходимо применить команду повторения, которая последовательно вводит каждую оценку. Мы не знаем заранее, сколько оценок придется обработать, поэтому используем цикл со сторожевым значением. Пользователь вводит оценки одну за другой. После последней оценки вводится сторожевое значение. Приложение проверяет каждую полученную оценку и при обнаружении сторожевого значения завершает цикл. Второе уточнение приведенной строки псевдокода выглядит так:

*запросить у пользователя первую оценку  
ввести первую оценку (возможно, сторожевое значение)*

*пока пользователь еще не ввел сторожевое значение,  
прибавить оценку к накапливаемой сумме  
увеличить счетчик оценок на 1  
запросить у пользователя следующую оценку  
ввести следующую оценку (возможно, сторожевое значение)*

В псевдокоде команды, образующие тело структуры `while`, не заключаются в фигурные скобки. Чтобы показать, что команды принадлежат `while`, мы просто снабжаем их отступом. Стоит еще раз напомнить, что псевдокод является всего лишь неформальным средством разработки.

Псевдокод

*вычислить и вывести среднюю оценку*

уточняется до следующего фрагмента:

*если счетчик не равен 0  
    задать средней оценке результат деления суммы на счетчик  
    вывести среднее значение  
иначе  
    вывести сообщение об отсутствии оценок*

Обратите внимание на проверку возможного деления на ноль — без нее возникает логическая ошибка, которая может привести к сбою приложения. Полное второе уточнение псевдокода для задачи вычисления оценок показано на ил. 5.8.



## КАК ИЗБЕЖАТЬ ОШИБОК 5.2

При делении на выражение, значение которого может быть равно нулю, всегда явно проверяйте такую возможность и обрабатывайте ее (например, выводите сообщение об ошибке), не допуская возникновения ошибки.

```
1  инициализировать сумму 0
2  инициализировать счетчик 0
3
4  запросить у пользователя первую оценку
5  ввести первую оценку (возможно, сторожевое значение)
6
7  пока пользователь еще не ввел сторожевое значение,
8    прибавить оценку к накапливаемой сумме
9    увеличить счетчик оценок на 1
10  запросить у пользователя следующую оценку
11  ввести следующую оценку (возможно, сторожевое значение)
12
13  если счетчик не равен 0
14    задать средней оценке результат деления суммы на счетчик
15    вывести среднее значение
16  иначе
17    вывести сообщение об отсутствии оценок
```

**Ил. 5.8.** Псевдокод алгоритма вычисления средней оценки с использованием цикла со сторожевым значением

На ил. 5.5 и 5.8 были включены пустые строки и отступы, упрощающие чтение псевдокода. Пустые строки разделяют алгоритмы псевдокода на фазы и выделяют управляющие команды, а отступы обозначают тело управляющих команд.

Псевдокод на ил. 5.8 решает более общую задачу вычисления средней оценки, а для его разработки было достаточно всего двух уточнений. Иногда задача требует большего количества уточнений.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 5.4

Нисходящий процесс пошагового уточнения завершается тогда, когда алгоритм будет определен достаточно подробно для преобразования псевдокода в C#.




#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 5.5

Некоторые опытные программисты пишут приложения без применения средств, упрощающих разработку (таких, как псевдокод). Они считают, что их конечной целью является решение задачи на компьютере, а написание псевдокода только откладывает момент получения результата. Хотя такой подход работает для простых и знакомых задач, он может привести к серьезным ошибкам и задержкам в больших сложных проектах.

### Реализация цикла со сторожевым значением в классе GradeBook

На ил. 5.9 показан класс C# GradeBook с методом DetermineClassAverage, реализующим алгоритм на ил. 5.8. Хотя каждая оценка является целым числом, в результате вычисления с большой вероятностью будет получено дробное число (то есть вещественное число или число с плавающей точкой). Для представления таких чисел тип `int` не подходит, поэтому в классе используется тип `double`.

```
1 // Ил. 5.9: GradeBook.cs
2 // Класс GradeBook, решающий задачу вычисления средней оценки
3 // с использованием цикла со сторожевым значением.
4 using System;
5
6 public class GradeBook
7 {
8     // Автоматически реализуемое свойство CourseName
9     public string CourseName { get; set; }
10
11     // Конструктор инициализирует свойство CourseName
12     public GradeBook( string name )
13     {
14         CourseName = name; // Присвоить name свойству CourseName
15     } // Конец конструктора
16
17     // Вывод приветствия для пользователя GradeBook
18     public void DisplayMessage()
19     {
20         Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
21             CourseName );
22     } // Конец метода DisplayMessage
23
24     // Вычисление средней оценки для произвольного количества оценок
25     public void DetermineClassAverage()
26     {
27         int total; // Сумма оценок
28         int gradeCounter; // Количество введенных оценок
29         int grade; // Значение оценки
30         double average; // Дробное число для средней оценки
31
32         // Фаза инициализации
33         total = 0; // Инициализация total
34         gradeCounter = 0; // Инициализация счетчика цикла
35
36         // Фаза выполнения
37         // Запрос и получение оценки у пользователя
38         Console.Write( "Enter grade or -1 to quit: " );
39         grade = Convert.ToInt32( Console.ReadLine() );
40
41         // Цикл до получения сторожевого значения от пользователя
42         while ( grade != -1 )
43         {
44             total = total + grade; // Прибавление оценки к сумме
45             gradeCounter = gradeCounter + 1; // Увеличение счетчика
46
47             // Запрос и получение следующей оценки у пользователя
48             Console.Write( "Enter grade or -1 to quit: " );
49             grade = Convert.ToInt32( Console.ReadLine() );
50         } // Конец while
51
52         // Фаза завершения
53         // Если пользователь ввел хотя бы одну оценку...
```

**Ил. 5.9.** Класс GradeBook для вычисления средней оценки с использованием цикла со сторожевым значением (продолжение )



```
54     if ( gradeCounter != 0 )
55     {
56         // Вычисление средней оценки
57         average = ( double ) total / gradeCounter;
58
59         // Вывод суммы и средней оценки (с точностью до двух цифр)
60         Console.WriteLine( "\nTotal of the {0} grades entered is {1}",
61                             gradeCounter, total );
62         Console.WriteLine( "Class average is {0: F}",average );
63     } // Конец if
64     else // Введенных оценок нет, вывести сообщение об ошибке
65         Console.WriteLine( "No grades were entered" );
66 } // Конец метода DetermineClassAverage
67 } // Конец класса GradeBook
```

**Ил. 5.9.** Класс GradeBook для вычисления средней оценки с использованием цикла со сторожевым значением (окончание)

Как видно из листинга, управляющие команды могут объединяться в цепочки. За командой `while` (строки 42–50) следует команда `if...else` (строки 54–65). Большая часть кода приложения идентична коду на ил. 5.6, так что мы сейчас сосредоточимся на новых аспектах.

В строке 30 объявляется переменная `average` типа `double`. Эта переменная позволяет сохранить вычисленное значение в формате с плавающей точкой. В строке 34 переменная `gradeCounter` инициализируется 0, потому что оценки еще не вводились. Приложение увеличивает `gradeCounter` только тогда, когда пользователь вводит допустимое значение оценки.

### Циклы со счетчиком и сторожевым значением: сравнение логики

Сравним логику программы для цикла со сторожевым значением в этом приложении с логикой цикла со счетчиком из ил. 5.6. В цикле со счетчиком каждое повторение цикла `while` (например, строки 38–44 на ил. 5.6) получает значение от пользователя с заданным количеством повторений. В цикле со сторожевым значением приложение читает первое значение (строки 38–39 на ил. 5.9) перед входом в `while`. Это значение определяет, войдет ли поток операций приложения в тело `while`. Если условие `while` ложно, пользователь ввел сторожевое значение, поэтому тело `while` не выполняется (потому что ни одна оценка не введена). С другой стороны, если условие истинно, то тело начинает выполняться, цикл прибавляет `grade` к `total` (строка 44) и увеличивает `gradeCounter` на 1 (строка 45). Затем в строках 48–49 в теле цикла пользователь вводит следующее значение.

Затем программа достигает закрывающей фигурной скобки в строке 50, так что выполнение продолжается проверкой условия `while` (строка 42).

Чтобы определить, должно ли тело цикла выполняться снова, условие использует последнюю оценку, введенную пользователем. Значение переменной `grade` всегда вводится пользователем непосредственно перед проверкой условия приложением. Это позволяет определить, совпадает ли только что введенное значение со сторожевым, прежде чем оно будет обработано приложением (то есть прибавлено к `total`).

Если пользователь ввел сторожевое значение, цикл завершается; приложение не прибавляет  $-1$  к `total`.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 5.6

В цикле со сторожевым значением запросы данных должны явно напоминать пользователю, какое значение считается сторожевым.

Обратите внимание на блок команды `while` на ил. 5.9 (строки 43–50). Без фигурных скобок цикл будет считать, что его тело состоит только из первой команды, прибавляющей `grade` к `total`. Последние три команды в блоке выпадут из тела цикла, в результате чего компьютер будет неправильно интерпретировать код следующим образом:

```
while ( grade != -1 )
    total = total + grade; // Прибавление grade к total
    gradeCounter = gradeCounter + 1; // Увеличение счетчика

// Запрос и получение следующей оценки у пользователя
Console.Write( "Enter grade or -1 to quit: " );
grade = Convert.ToInt32( Console.ReadLine() );
```

Этот код создает бесконечный цикл, если пользователь не ввел сторожевое значение  $-1$  в строке 39 (перед командой `while`).



### КАК ИЗБЕЖАТЬ ОШИБОК 5.3

Отсутствие фигурных скобок, в которые заключается блок, может привести к логическим ошибкам — например, бесконечным циклам. Для решения этой проблемы некоторые программисты заключают в фигурные скобки тело каждой управляющей команды, даже если оно состоит всего из одной команды.

После завершения цикла выполняется команда `if...else` в строках 54–65. Условие в строке 54 проверяет, были ли введены оценки. Если ни одна оценка не введена, секция `else` (строки 64–65) команды `if...else` выполняется и выводит сообщение "No grades were entered", и управление возвращается вызывающему методу.

### Явное и неявное преобразование простых типов

Если была введена хотя бы одна оценка, строка 57 вычисляет среднее арифметическое оценок. Как было сказано ранее, целочисленное деление дает целочисленный результат. И хотя переменная `average` объявлена с типом `double` (строка 30), дробная часть результата

```
average = total / gradeCounter;
```

будет потеряна еще до того, как результат будет присвоен `average`. Это происходит из-за того, что и `total`, и `gradeCounter` являются целыми числами, а в результате целочисленного деления получается целый результат. Чтобы применить деление с плавающей точкой к целочисленным значениям, необходимо временно интерпретировать их как числа с плавающей точкой для использования в вычислениях.

В C# для этой цели существует унарный оператор преобразования типа. В строке 57 используется оператор преобразования (`double`) (обладающий более высоким приоритетом, чем арифметические операторы) для создания *временной* копии операнда `total` с плавающей точкой (находящегося справа от оператора). Такое применение оператора преобразования типа называется *явным преобразованием*. Значение, хранящееся в `total`, так и остается целым числом.

Теперь в вычислении участвует значение с плавающей точкой (временная версия `total` типа `double`), которое делится на целое число `gradeCounter`. C# умеет вычислять только арифметические выражения с одинаковыми типами операндов. Чтобы операнды относились к одному типу, C# выполняет с ними операцию *неявного преобразования*. Например, в выражении, содержащем значения типов `int` и `double`, значения `int` преобразуются к типу `double` для использования в вычислениях. В нашем примере значение `gradeCounter` преобразуется к типу `double`, после чего выполняется деление с плавающей точкой, а его результат присваивается `average`. При условии, что к любой из переменных в вычислениях применяется оператор (`double`), вычисление дает результат типа `double`.



#### ТИПИЧНАЯ ОШИБКА 5.6

Оператор преобразования может использоваться для преобразования между простыми числовыми типами (такими, как `int` и `double`), а также между взаимосвязанными ссылочными типами (см. главу 12). Преобразование к неверному типу может привести к ошибкам компиляции или времени выполнения.

Операторы преобразования типа доступны для всех простых типов. Операторы преобразования для ссылочных типов рассматриваются в главе 12. Оператор преобразования записывается с заключением имени типа в круглые скобки. Оператор является унарным (то есть получает только один операнд). В главе 3 рассматривались бинарные арифметические операторы. C# также поддерживает унарные версии операторов `+` и `-`, что позволяет использовать выражения вида `+5` или `-7`. Операторы преобразования применяются слева направо и имеют такой же приоритет, как другие унарные операторы. Этот приоритет на один уровень выше приоритета операторов умножения/деления (`*`, `/` и `%`).

В строке 62 выводится средняя оценка группы. В своем примере мы округляем ее до сотых и выводим оценку с двумя цифрами в дробной части. Спецификатор формата `F` в форматном элементе `WriteLine` (строка 62) указывает, что значение переменной `average` должно выводиться как вещественное число. По умолчанию числа со спецификатором `F` выводятся с двумя цифрами в дробной части (количество цифр в дробной части также называется *точностью*). Любое значение с плавающей точкой, выводимое с `F`, округляется до сотых — например, число 123,457 будет округлено до 123,46, а число 27,333 — до 27,33. В нашем приложении сумма трех оценок, введенных при тестовом запуске класса `GradeBookTest` (ил. 5.10), составляет 263, что дает среднюю оценку 87,66666.... Форматный элемент округляет ее до сотых, поэтому средняя оценка выводится в виде 87.67.

```
1 // Ил. 5.10: GradeBookTest.cs
2 // Создание объекта GradeBook и вызов его метода DetermineClassAverage.
3 public class GradeBookTest
4 {
5     public static void Main( string[] args )
6     {
7         // Создание объекта myGradeBook класса GradeBook
8         // с передачей названия курса конструктору
9         GradeBook myGradeBook = new GradeBook(
10             "CS101 Introduction to C# Programming" );
11
12         myGradeBook.DisplayMessage(); // Вывод приветствия
13         myGradeBook.DetermineClassAverage(); // Вычисление средней оценки
14     } // Конец Main
15 } // Конец класса GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to C# Programming!
```

```
Enter grade or -1 to quit: 96
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 79
Enter grade or -1 to quit: -1
```

```
Total of the 3 grades entered is 263
Class average is 87.67
```

**Ил. 5.10.** Создание объекта GradeBook и вызов метода DetermineClassAverage

## 5.10. Формулировка алгоритмов: вложенные управляющие команды

В следующем примере мы снова сформулируем алгоритм на псевдокоде с использованием метода нисходящего пошагового уточнения и напишем соответствующее приложение C#. Вы уже видели, что управляющие команды могут объединяться в последовательные цепочки. На этот раз будет рассмотрен другой способ связывания управляющих команд, а именно их вложение.

Возьмем следующую постановку задачи:

*Один из учебных курсов в колледже готовит студентов к лицензионному экзамену на должность агента по недвижимости. За последний год 10 студентов, прошедших этот курс, сдавали экзамен. Колледж хочет знать, какие результаты показали студенты на экзамене. Вам предложено написать приложение для обобщения результатов. Вы получаете список 10 студентов; рядом с каждым именем стоит 1, если студент сдал экзамен, или 2, если экзамен был провален.*

*Ваше приложение должно проанализировать результаты экзамена следующим образом:*

1. *Ввести каждый результат (то есть 1 или 2). Каждый раз, когда приложение запрашивает очередной результат, на экран должно выводиться сообщение «Enter result».*
2. *Подсчитать количество результатов каждого типа.*
3. *Вывести сводку результатов с количествами студентов, сдавших экзамен и проваливших его.*
4. *Если экзамен прошли более 8 студентов, вывести сообщение «Bonus to instructor!».*

Из чтения постановки задачи можно сделать следующие выводы:

1. Приложение должно обработать результаты 10 студентов. Так как количество результатов известно заранее, можно воспользоваться циклом со счетчиком.
2. Каждый результат представлен числовым значением — 1 или 2. Каждый раз, когда приложение читает очередной результат, оно должно определить, было ли прочитано число 1 или 2. В нашем алгоритме число проверяется на равенство 1. Если число отлично от 1, предполагается, что это 2.
3. Для отслеживания результатов используются два счетчика: в одном подсчитываются студенты, успешно сдавшие экзамен, а в другом — провалившиеся студенты.
4. После того как приложение обработает все результаты, оно должно определить, был ли экзамен сдан более чем 8 студентами.

Как и в предыдущем случае, будет использоваться метод нисходящего пошагового уточнения. Начнем с представления верхнего уровня на псевдокоде:

*проанализировать результаты экзамена и решить, заслужил ли преподаватель премию*

И снова высокоуровневая формулировка является полным описанием приложения, но для того, чтобы псевдокод мог естественно преобразоваться в приложение C#, потребуется несколько уточнений.

Первое уточнение выглядит так:

*инициализировать переменные  
ввести 10 результатов, подсчитать удачные и неудачные попытки  
вывести сводку результатов и решить, заслужил ли преподаватель премию*

Хотя и в этом случае псевдокод содержит полное описание всего приложения, потребуется дальнейшее уточнение. Теперь можно заняться отдельными переменными. Для подсчета удачных и неудачных попыток понадобятся счетчики; еще один счетчик будет управлять циклом, а также понадобится переменная для хранения пользовательского ввода. Эта переменная не инициализируется в начале алгоритма, потому что ее значение будет вводиться пользователем при каждой итерации цикла.

## Псевдокод

```
инициализировать переменные
```

после уточнения принимает следующий вид:

```
инициализировать счетчик удачных попыток 0
инициализировать счетчик неудачных попыток 0
инициализировать счетчик студентов 1
```

Обратите внимание: в начале алгоритма инициализируются только счетчики.

Для уточнения псевдокода

```
ввести 10 результатов, подсчитать удачные и неудачные попытки
```

потребуется цикл, тело которого успешно вводит результат одного экзамена. Нам заранее известно, что количество результатов равно 10, поэтому цикл со счетчиком будет уместен. Внутри цикла вложенная команда двойного выбора будет определять, является ли текущий результат удачным или неудачным, и увеличивать соответствующий счетчик. Псевдокод уточняется до следующего фрагмента:

```
пока счетчик студентов меньше либо равен 10
    запросить у пользователя следующий результат экзамена
    ввести следующий результат экзамена
    если экзамен сдан успешно
```

```
        увеличить счетчик удачных попыток на 1
    иначе
```

```
        увеличить счетчик неудачных попыток на 1
```

```
увеличить счетчик студентов на 1
```

Команда `if...else` выделена пустыми строками для удобства чтения. Псевдокод

```
вывести сводку результатов и решить, заслужил ли преподаватель премию
```

может быть уточнен до следующего фрагмента:

```
вывести количество удачных попыток
вывести количество неудачных попыток
если экзамен сдали более 8 студентов
    вывести сообщение о премии для преподавателя
```

**Полное второе уточнение псевдокода и преобразование в класс**

Полное второе уточнение псевдокода приведено на ил. 5.11. Пустые строки отделяют цикл от остального кода и упрощают чтение. Теперь псевдокод достаточно

детализирован для преобразования в код С#. Программа, реализующая алгоритм псевдокода, приведена на ил. 5.12 вместе с примерами результатов.

```
1  инициализировать счетчик удачных попыток 0
2  инициализировать счетчик неудачных попыток 0
3  инициализировать счетчик студентов 1
4
5  пока счетчик студентов меньше либо равен 10
6      запросить у пользователя следующий результат экзамена
7      ввести следующий результат экзамена
8
9      если экзамен сдан успешно
10         увеличить счетчик удачных попыток на 1
11     иначе
12         увеличить счетчик неудачных попыток на 1
13
14     увеличить счетчик студентов на 1
15
16     вывести количество удачных попыток
17     вывести количество неудачных попыток
18
19     если экзамен сдали более 8 студентов
20         вывести сообщение о премии для преподавателя
```

**Ил. 5.11.** Псевдокод задачи анализа результатов экзамена

Этот пример содержит всего один класс, а вся его работа выполняется в методе Main. Примеры, которые рассматривались в этой главе и в главе 4, состояли из двух классов — один класс содержал методы, выполнявшие полезные операции, а другой содержал метод Main, который создавал объект другого класса и вызывал его методы. В отдельных случаях, когда создаваемый класс заведомо непригоден для повторного использования, весь пример будет содержаться в методе Main одного класса.

В строках 10–13 на ил. 5.12 объявляются переменные, используемые методом Main для обработки результатов. В некоторых из этих объявлений используется возможность инициализации переменных при объявлении (passes присваивается 0, failures присваивается 0, studentCounter присваивается 1).

```
1  // Ил. 5.12: Analysis.cs
2  // Анализ результатов экзамена с вложением управляющих команд.
3  using System;
4
5  public class Analysis
6  {
7      public static void Main( string[] args )
8      {
9          // Инициализация переменных при объявлении
10         int passes = 0; // количество удачных попыток
11         int failures = 0; // количество неудачных попыток
12         int studentCounter = 1; // счетчик студентов
13         int result; // результат, введенный пользователем
14     }
```

**Ил. 5.12.** Анализ результатов экзаменов с использованием вложенных управляющих конструкций (продолжение ↗)

```
15 // Обработка 10 студентов с использованием цикла со счетчиком
16 while ( studentCounter <= 10 )
17 {
18     // Запрос и получение значения от пользователя
19     Console.Write( "Enter result (1 = pass, 2 = fail): " );
20     result = Convert.ToInt32( Console.ReadLine() );
21
22     // Команда if...else, вложенная в while
23     if ( result == 1 ) // Если переменная result равна 1,
24         passes = passes + 1; // увеличить счетчик удачных попыток
25     else // Если результат отличен от 1,
26         failures = failures + 1; // увеличить счетчик неудач.
27
28     // Увеличение studentCounter, чтобы цикл завершился
29     studentCounter = studentCounter + 1;
30 } // Конец while
31
32 // Фаза завершения; подготовка и вывод результатов
33 Console.WriteLine( "Passed: {0}\nFailed: {1}", passes, failures );
34
35 // Определить, сдали ли экзамен более 8 студентов
36 if ( passes > 8 )
37     Console.WriteLine( "Bonus to instructor!" );
38 } // Конец Main
39 } // Конец класса Analysis
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Passed: 5
Failed: 5
```

**Ил. 5.12.** Анализ результатов экзаменов с использованием вложенных управляющих конструкций (окончание)



Команда `while` (строки 16–30) выполняется 10 раз. При каждом повторении цикл вводит и обрабатывает один результат экзамена. Обратите внимание: команда `if...else` (строки 23–26) для обработки каждого результата вложена в команду `while`. Если переменная `result` равна 1, команда `if...else` увеличивает `passes`; в противном случае она считает, что `result` содержит 2, и увеличивает `failures`. Строка 29 увеличивает `studentCounter` перед тем, как условие цикла снова будет проверено в строке 16. После ввода 10 значений цикл завершается и в строке 33 выводится количество удачных и неудачных попыток. Строки 36–37 проверяют, что экзамен был сдан более чем 8 студентами, и если условие выполняется — выводят сообщение.

На ил. 5.12 также приведены входные и выходные данные двух тестовых запусков приложения. При первом запуске условие в строке 36 было истинным — экзамен сдали более 8 студентов, поэтому приложение выводит сообщение о премии для преподавателя.



#### КАК ИЗБЕЖАТЬ ОШИБОК 5.4

Инициализация локальных переменных при объявлении помогает избежать ошибок компиляции, возникающих из-за попыток использования неинициализированных данных. Хотя C# не требует, чтобы локальные переменные инициализировались при объявлении, они должны быть обязательно инициализированы до использования значений в выражениях.

## 5.11. Комбинированные операторы присваивания

В C# существует несколько комбинированных операторов присваивания для сокращения выражений присваивания. Любая команда в форме

```
переменная = переменная оператор выражение;
```

где *оператор* — один из бинарных операторов `+`, `-`, `*`, `/` или `%` (или другие операторы, которые будут рассматриваться далее в тексте), может быть записана в форме

```
переменная оператор = выражение;
```

Например, команда

```
c = c + 3;
```

с комбинированным оператором присваивания `+=` может быть записана в виде

```
c += 3;
```

Оператор `+=` прибавляет значение выражения в правой части оператора к значению переменной в левой части оператора и сохраняет результат в переменной в левой части. Таким образом, выражение `c+=3` увеличивает `c` на 3. На ил. 5.13 перечислены арифметические комбинированные операторы присваивания, примеры выражений с использованием этих операторов и краткие пояснения того, что эти операторы делают.

Оператор присваивания	Пример выражения	Объяснение	Присваивание
Исходное состояние: <code>int c = 3, d = 5, e = 4, f = 6, g = 12;</code>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	c присваивается 10
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	d присваивается 1
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	e присваивается 20
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	f присваивается 2
<code>%=</code>	<code>f %= 9</code>	<code>g = g % 9</code>	g присваивается 3

Ил. 5.13. Арифметические комбинированные операторы присваивания

## 5.12. Операторы инкремента и декремента

C# предоставляет два унарных оператора для увеличения или уменьшения значения числовой переменной на 1. Это унарный оператор *инкремента* `++` и унарный оператор *декремента* `--`, краткие описания которых представлены на ил. 5.14. Оператор инкремента или декремента может размещаться как до переменной, к которой он относится (*префиксная запись*), так и после нее (*постфиксная запись*).

Оператор	Название	Пример выражения	Объяснение
<code>++</code>	Префиксный инкремент	<code>++a</code>	Увеличивает a на 1 и использует новое значение a в выражении, в котором находится a
<code>++</code>	Постфиксный инкремент	<code>a++</code>	Увеличивает a на 1, но использует исходное значение a в выражении, в котором находится a
<code>--</code>	Постфиксный инкремент	<code>--b</code>	Уменьшает b на 1 и использует новое значение b в выражении, в котором находится b
<code>--</code>	Постфиксный декремент	<code>b--</code>	Уменьшает b на 1 и использует исходное значение b в выражении, в котором находится b

Ил. 5.14. Операторы инкремента и декремента

Применение оператора инкремента (или декремента) увеличивает (или уменьшает) переменную на 1. Для префиксной разновидности оператора инкремента (или декремента) в выражении используется измененное значение переменной, а в постфиксной разновидности используется исходное значение.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 5.7

В отличие от бинарных операторов, унарные операторы инкремента и декремента должны размещаться вплотную к операндам, без промежуточных пробелов.

На ил. 5.15 представлены различия между префиксной и постфиксной записью оператора инкремента ++. Оператор декремента (--) работает аналогичным образом. В этом примере мы просто хотим продемонстрировать механику оператора ++, поэтому в нем используется объявление одного класса с методом Main.

```

1  // Ил. 5.15: Increment.cs
2  // Префиксная и постфиксная запись оператора инкремента.
3  using System;
4
5  public class Increment
6  {
7      public static void Main( string[] args )
8      {
9          int c;
10
11         // Постфиксный инкремент
12         c = 5; // Присвоить переменной c значение 5
13         Console.WriteLine( c ); // Выводит 5
14         Console.WriteLine( c++ ); // Увеличивает c и выводит 5
15         Console.WriteLine( c ); // Выводит 6
16
17         Console.WriteLine(); // Пропуск строки
18
19         // Префиксный инкремент
20         c = 5; // Присвоить переменной c значение 5
21         Console.WriteLine( c ); // Выводит 5
22         Console.WriteLine( ++c ); // Увеличивает c и выводит 6
23         Console.WriteLine( c ); // Снова выводит 6
24     } // Конец Main
25 } // Конец класса Increment

```

```

5
5
6

5
6
6

```

**Ил. 5.15.** Префиксный и постфиксный операторы инкремента

Строка 12 инициализирует переменную с значением 5, а строка 13 выводит исходное значение с. В строке 14 выводится значение выражения c++. Выражение применяет постфиксный оператор инкремента к переменной c, так что несмотря на то, что значение с увеличилось, выводится исходное значение с (5). Таким образом, в строке 14 снова выводится исходное значение с (5). В строке 15 выводится новое значение с (6); это доказывает, что переменная действительно увеличилась в строке 14.

Строка 20 снова присваивает с значение 5, которое выводится в строке 21. Строка 22 выводит значение выражения ++c. Выражение применяет префиксный оператор инкремента к переменной c, поэтому выводится новое значение с (6).

Вывод переменной `c` (6) в строке 23 доказывает, что после выполнения строки 22 переменная `c` не изменилась.

Арифметические комбинированные операторы присваивания и операторы инкремента/декремента могут использоваться для упрощения команд. Например, три команды присваивания на ил. 5.12 (строки 24, 26 и 29)

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

можно более компактно записать с использованием комбинированных операторов присваивания в виде

```
passes += 1;
failures += 1;
studentCounter += 1;
```

и даже еще компактнее с префиксными операторами инкремента:

```
++passes;
++failures;
++studentCounter;
```

или постфиксными операторами:

```
passes++;
failures++;
studentCounter++;
```

В том, что касается увеличения или уменьшения самой переменной, префиксная и постфиксная запись приводят к одинаковому результату. Различия между префиксной и постфиксной формой проявляются только при использовании переменной в контексте большего выражения.



#### ТИПИЧНАЯ ОШИБКА 5.7

Попытка применения оператора инкремента или декремента к выражению, которое не может использоваться для присваивания, является синтаксической ошибкой. Например, запись `++(x + 1)` невозможна, потому что `(x + 1)` невозможно присвоить значение.

### Приоритет и порядок применения операторов

На ил. 5.16 представлены приоритеты и порядок применения операторов, рассмотренных нами до настоящего момента. Операторы перечисляются в порядке убывания приоритетов. Во втором столбце указан порядок применения операторов для каждого уровня приоритета. Условный оператор (`?:`); унарный префиксный инкремент (`++`), префиксный декремент (`--`), плюс (`+`) и минус (`-`); операторы преобразования типа; операторы присваивания `=`, `+=`, `-=`, `*=`, `/=` и `%=` применяются справа налево. Все остальные операторы в таблице на ил. 5.16 применяются слева направо. В третьем столбце перечислены группы операторов.

Операторы	Порядок применения	Тип
. new ++ (постфиксный) -- (постфиксный)	слева направо	максимальный приоритет
++ -- + - (тип)	справа налево	унарный префикс
* / %	слева направо	умножение/деление
+ -	слева направо	сложение/вычитание
< <= > >=	слева направо	сравнение
== !=	слева направо	проверка равенства
?:	справа налево	условный оператор
= += -= *= /= %=	справа налево	присваивание

**Ил. 5.16.** Приоритет и порядок применения операторов, рассмотренных до настоящего момента

## 5.13. Простые типы

В С#, как и в его предшественниках — языках С и С++, все переменные должны обладать некоторым типом. По этой причине С# называется *языком с сильной типизацией*.

В С и С++ часто приходится писать несколько версий приложений для поддержки разных компьютерных платформ, потому что идентичность простых типов на разных компьютерах не гарантирована. Например, значение `int` на одной машине может представляться 16 битами (2 байтами), тогда как на другой машине оно может представляться 32 битами (4 байтами). В С# значение типа `int` всегда занимает 32 бита (4 байта). Все числовые типы С# имеют фиксированные размеры.

Как говорилось в разделе 4.5, переменным простых типов, объявленным вне методов как поля класса, автоматически присваивается значение по умолчанию (если переменные не инициализируются явно). Переменным экземпляров типов `char`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal` по умолчанию присваивается значение 0. Переменные экземпляров типа `bool` по умолчанию инициализируются значением `false`. Аналогичным образом переменные экземпляров ссылочных типов по умолчанию инициализируются значением `null`.

## 5.14. Итоги

В этой главе представлены основные приемы, используемые при построении классов и разработки методов этих классов. Мы показали, как сконструировать алгоритм (то есть описание решения) и как уточнить алгоритм в нескольких фазах разработки псевдокода с получением кода С#, который может выполняться в составе метода.

Глава демонстрирует применение метода нисходящего пошагового уточнения для планирования конкретных действий, которые должны быть выполнены методом, и порядок их выполнения.

Для разработки любого алгоритма необходимы только три разновидности управляющих структур: последовательность, выбор и повторение. Мы рассмотрели команду одиночного выбора `if`, команду двойного выбора `if...else` и команду повторения `while`. Эти структурные элементы используются для построения решений многих задач. Сцепление управляющих команд применялось для суммирования и вычисления средней оценки с применением циклов со счетчиком и сторожевым условием, а вложение — для анализа набора результатов и принятия решений. Также были рассмотрены комбинированные операторы присваивания `C#`, унарный оператор преобразования типа, условный оператор `(?:)`, операторы инкремента и декремента.

В главе 6 мы продолжим рассмотрение управляющих команд и изучим команды `for`, `do...while` и `switch`.

# 6 Управляющие команды: часть 2

## 6.1. Введение

В этой главе будет представлено большинство оставшихся управляющих команд C# (команда `foreach` будет рассмотрена в главе 8). Управляющие команды, рассматриваемые здесь и в главе 5, используются при создании объектов и работе с ними.

В серии коротких примеров с использованием `while` и `for` рассматриваются основные принципы повторения со счетчиком. Мы создадим версию класса `GradeBook`, которая использует команду `switch` для подсчета разных числовых оценок из набора, введенного пользователем. Также будут представлены команды `break` и `continue`, управляющие ходом выполнения программы. Мы рассмотрим логические операторы C#, которые позволяют строить более сложные условные выражения в управляющих командах. Глава завершается сводкой управляющих команд C# и методов решения задач, представленных в этой главе и в главе 5.

## 6.2. Основные принципы повторения со счетчиком

В этом разделе команда `while` будет использоваться для формального представления элементов, необходимых для реализации выполнения со счетчиком. Для организации повторения со счетчиком в программе вам потребуются:

1. Управляющая переменная (счетчик цикла).
2. Исходное значение управляющей переменной.
3. Приращение управляющей переменной при каждом проходе цикла (итерации).
4. Условие продолжения цикла, определяющее, должен ли продолжаться цикл.

Все эти элементы повторения со счетчиком представлены в листинге на ил. 6.1, в котором цикл используется для вывода чисел от 1 до 10.

```
1 // Ил. 6.1: WhileCounter.cs
2 // Повторение со счетчиком на базе цикла while.
3 using System;
4
5 public class WhileCounter
6 {
7     public static void Main( string[] args )
8     {
9         int counter = 1; // Объявление и инициализация управляющей переменной
10
11         while ( counter <= 10 ) // Условие продолжения цикла
12         {
13             Console.Write( "{0} ", counter );
14             ++counter; // Увеличение управляющей переменной
15         } // Конец while
16
17         Console.WriteLine(); // Перевод строки
18     } // Конец Main
19 } // Конец класса WhileCounter
```

1 2 3 4 5 6 7 8 9 10

### Ил. 6.1. Повторение со счетчиком на базе цикла while

В методе `Main` (строки 7–18) элементы повторения со счетчиком определяются в строках 9, 11 и 14. Строка 9 объявляет переменную цикла (счетчик) с типом `int`, резервирует для нее память и задает начальное значение 1.

Строка 13 в команде `while` выводит значение счетчика при каждой итерации цикла. Строка 14 увеличивает управляющую переменную на 1 при каждой итерации. Условие продолжения цикла в `while` (строка 11) проверяет, что значение управляющей переменной меньше либо равно 10 (последнее значение, при котором условие истинно). Приложение выполняет тело `while` даже тогда, когда управляющая переменная равна 10. Цикл завершается тогда, когда управляющая переменная превысит 10 (то есть счетчик достигает 11).



#### КАК ИЗБЕЖАТЬ ОШИБОК 6.1

Поскольку значения с плавающей точкой могут быть приблизительными, использование переменной с плавающей точкой в качестве счетчика цикла может привести к погрешностям и неточной проверке завершения. Используйте целочисленные счетчики.

Чтобы сделать приложение на ил. 6.1 более компактным, инициализируйте `counter` значением 0 в строке 9 и увеличивайте счетчик в условии `while` префиксным оператором инкремента:

```
while ( ++counter <= 10 ) // Условие продолжения цикла
    Console.Write( "{0} ", counter );
```

Этот код экономит одну команду (и избавляет от необходимости заключать тело цикла в фигурные скобки), потому что условие `while` выполняет инкремент перед проверкой условия. (Как говорилось в разделе 5.12, приоритет оператора `++` выше,



чем у оператора `<=`.) Однако следует помнить, что такой компактный код создает больше проблем при чтении, отладке, модификации и сопровождении.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 6.1

«Не усложняйте» — хорошее правило для большей части всего кода, который вы напишете.

## 6.3. Цикл for

В разделе 6.2 были изложены основные принципы конструкций повторения со счетчиком. Команда `while` может использоваться для реализации любых циклов со счетчиком, но в C# также предусмотрен цикл `for`, в котором все управляющие элементы задаются в одной строке. Как правило, для реализации повторений со счетчиком рекомендуется применять циклы `for`. В листинге на ил. 6.2 приложение из ил. 6.1 реализовано на базе цикла `for`.

```

1 // Ил. 6.2: ForCounter.cs
2 // Counter-controlled repetition with the for repetition statement.
3 using System;
4
5 public class ForCounter
6 {
7     public static void Main( string[] args )
8     {
9         // Заголовок команды for включает инициализацию,
10        // условие продолжения цикла и приращение
11        for ( int counter = 1; counter <= 10; ++counter )
12            Console.Write( "{0} ", counter );
13
14        Console.WriteLine(); // Перевод строки
15    } // Конец Main
16 } // Конец класса ForCounter

```

1 2 3 4 5 6 7 8 9 10

**Ил. 6.2.** Повторение со счетчиком на базе цикла `for`

В начале выполнения команды `for` (строки 11–12) управляющая переменная `counter` объявляется и инициализируется значением 1. (Вспомните, о чем говорилось в разделе 6.2: первые два элемента повторения со счетчиком — управляющая переменная и ее исходное значение.) Затем приложение проверяет условие продолжения цикла `counter <= 10`, заключенное между двумя обязательными символами «точка с запятой». Исходное значение `counter` равно 1, так что условие изначально истинно. Соответственно, тело цикла (строка 12) выводит значение счетчика, равное 1. Затем приложение увеличивает счетчик выражением `++counter`, расположенным справа от второй точки с запятой. Следующая проверка продолжения цикла определяет, должно ли приложение перейти к следующей итерации. На этой стадии управляющая переменная равна 2, так что условие остается истинным — приложение снова

выполняет тело цикла (то есть следующую итерацию). Процесс продолжается до тех пор, пока не будут выведены все числа от 1 до 10 и значение счетчика не увеличится до 11; проверка условия дает отрицательный результат, а повторения прекращаются (после 10 повторений тела цикла в строке 12). Затем приложение выполняет первую команду после `for` — в данном случае это строка 14.

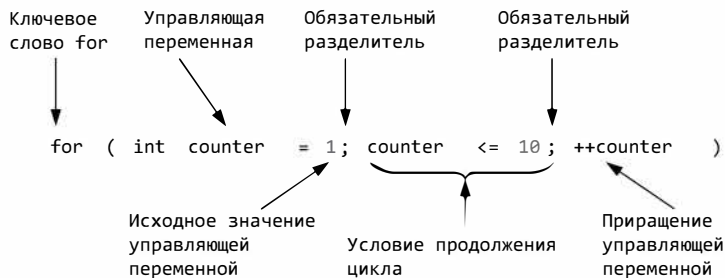
На ил. 6.2 (в строке 11) используется условие продолжения цикла `counter<=10`. Если неправильно указать в условии `counter< 10`, то цикл будет выполнен только 9 раз — распространенная логическая ошибка, называемая ошибкой «смещения на 1».



### КАК ИЗБЕЖАТЬ ОШИБОК 6.2

Использование итогового значения циклов `while` или `for` с оператором сравнения `<=` помогает избежать ошибок «смещения на 1». Для цикла, выводящего значения от 1 до 10, условие продолжения должно иметь вид `counter <= 10`, а не `counter < 10` (что приведет к ошибке «смещения на 1») или `counter < 11` (правильное условие). Многие программисты предпочитают «отсчет с нуля», при котором для отсчета 10 итераций счетчик инициализируется нулем, а условие продолжения цикла имеет вид `counter < 10`.

На ил. 6.3 изображена более подробная схема команды `for` с ил. 6.2. Первая строка `for` (включающая ключевое слово `for` и все, что следует за ним в круглых скобках) — строка 11 на ил. 6.2 — иногда называется заголовком команды `for` (или просто заголовком `for`). Заголовок `for` определяет каждый элемент, необходимый для повторения с управляющей переменной. Если тело `for` содержит более одной команды, оно должно быть заключено в фигурные скобки.



**Ил. 6.3.** Строение заголовка команды `for`

Обобщенный формат команды `for` выглядит так:

```
for ( инициализация; условие_продолжения; приращение )
    команда
```

Выражение *инициализация* определяет имя управляющей переменной цикла и задает ее исходное значение; *условие\_продолжения* определяет, должен ли цикл переходить к следующей итерации; а *приращение* изменяет значение управляющей переменной, чтобы *условие\_продолжения* в какой-то момент стало ложным. Два символа

«;» в заголовке `for` являются обязательными. Точка с запятой после команды не ставится; подразумевается, что она уже включена в саму команду. В большинстве случаев команда `for` может быть представлена эквивалентной командой `while`:

```
инициализация;
while ( условие_продолжения )
{
    команда
    приращение;
}
```

В разделе 6.7 рассмотрен пример, в котором команда `for` не может быть представлена командой `while` указанного вида.

Обычно циклы `for` используются для повторения со счетчиком, а циклы `while` — для повторения со сторожевым значением. Тем не менее формально `while` и `for` могут использоваться для организации повторения обоих типов.

Если выражение *инициализация* в заголовке `for` объявляет управляющую переменную (то есть если перед именем переменной указывается ее тип, как на ил. 6.2), управляющая переменная может использоваться только в этой команде — за ее пределами эта переменная не существует. Область, в которой имя переменной может использоваться, называется *областью действия* (scope). Например, локальная переменная может использоваться только в том методе, в котором она объявляется, и только от точки объявления до конца блока, в котором она объявляется. Область действия более подробно рассматривается в главе 7.



#### ТИПИЧНАЯ ОШИБКА 6.1

Если управляющая переменная цикла `for` объявляется в секции инициализации заголовка `for`, попытка ее использования после тела `for` приводит к ошибке компиляции.

Все три выражения в заголовке `for` не являются обязательными. Если опущено условие продолжения, C# считает, что оно всегда истинно; таким образом, возникает бесконечный цикл. Выражение инициализации можно опустить, если приложение инициализирует управляющую переменную до начала цикла — в этом случае область действия управляющей переменной не ограничивается циклом. Выражение приращения может отсутствовать, если приложение вычисляет приращение в теле цикла или приращение не нужно. Выражение приращения цикла `for` работает так, как если бы оно было самостоятельной командой в теле цикла `for`. Таким образом, выражения

```
counter = counter + 1
counter += 1
++counter
counter++
```

являются эквивалентными выражениями приращения в команде `for`. Многие программисты предпочитают выражение `counter++` из-за его компактности, а также потому, что цикл `for` вычисляет приращение после выполнения тела, так что

постфиксная форма выглядит более естественно. В данном случае изменяемая переменная не входит в большее выражение, так что префиксная и постфиксная формы работают одинаково.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 6.1

Префиксная форма оператора инкремента работает чуть более производительно, но если вы выберете постфиксную форму, потому что она выглядит более естественно (как в заголовке `for`), оптимизирующий компилятор сгенерирует код, в котором все равно будет использоваться более эффективная форма.



### КАК ИЗБЕЖАТЬ ОШИБОК 6.3

Если условие продолжения цикла всегда остается истинным, происходит заикливание. Для предотвращения заикливания в циклах со счетчиком следите за тем, чтобы управляющая переменная увеличивалась (или уменьшалась) при каждой итерации. В циклах со сторожевым значением позаботьтесь о том, чтобы сторожевое значение было введено.

Инициализация, условие продолжения цикла и приращение могут содержать арифметические выражения. Допустим,  $x = 2$  и  $y = 10$ ; если  $x$  и  $y$  не изменяются в теле цикла, то команда

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

эквивалентна команде

```
for ( int j = 2; j <= 80; j += 5 )
```

Приращение команды `for` может быть отрицательным; в этом случае счетчик не увеличивается, а уменьшается.

Если условие продолжения изначально ложно, приложение не выполняет тело цикла `for`. Вместо этого выполнение продолжается с команды, следующей за `for`.

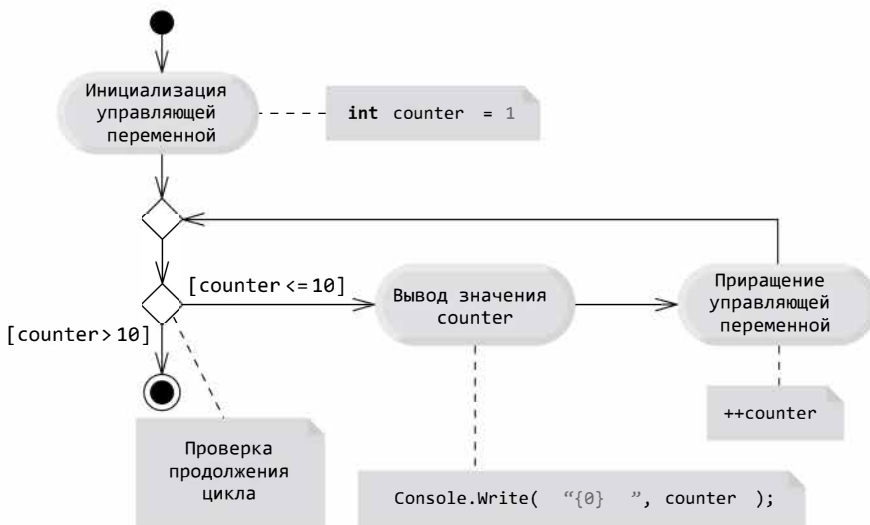
Приложения часто выводят управляющую переменную цикла или используют ее для вычислений в теле цикла, но это не обязательно. Использование управляющей переменной для управления повторением без использования в теле цикла `for` тоже встречается достаточно часто.



### КАК ИЗБЕЖАТЬ ОШИБОК 6.4

Хотя значение управляющей переменной может изменяться в теле цикла `for`, так поступать не рекомендуется из-за возможного появления коварных ошибок.

На ил. 6.4 изображена диаграмма активности команды `for` на ил. 6.2. Из диаграммы ясно видно, что инициализация выполняется *только один раз* до первой проверки условия продолжения, а приращение происходит после *каждого* выполнения тела цикла.



Ил. 6.4. Диаграмма активности UML для команды for на илл. 6.2

## 6.4. Примеры использования цикла for

В этом разделе продемонстрированы разные способы изменения управляющей переменной в команде for. В каждом случае мы записываем соответствующий заголовок for. Обратите внимание на изменения в операторе сравнения для циклов, уменьшающих управляющую переменную.

- ❑ Изменение управляющей переменной от 1 до 100 с увеличением на 1.

```
for ( int i = 1; i <= 100; ++i )
```

- ❑ Изменение управляющей переменной от 100 до 1 с уменьшением на 1.

```
for ( int i = 100; i >= 1; --i )
```

- ❑ Изменение управляющей переменной от 7 до 77 с увеличением на 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

- ❑ Изменение управляющей переменной от 20 до 2 с уменьшением на 2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

- ❑ Изменение управляющей переменной в последовательности 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

- ❑ Изменение управляющей переменной в последовательности 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```



### ТИПИЧНАЯ ОШИБКА 6.2

Использование неправильного оператора сравнения в условии проверки цикла с обратным отсчетом (например,  $i \leq 1$  вместо  $i \geq 1$  в цикле с отсчетом вниз до 1) является логической ошибкой.

### Суммирование четных целых чисел от 2 до 20

Рассмотрим пару приложений, демонстрирующих простое применение `for`. В приложении на ил. 6.5 цикл `for` используется для суммирования четных целых чисел от 2 до 20, а результат сохраняется в переменной с именем `total`.

```
1 // Ил. 6.5: Sum.cs
2 // Суммирование целых чисел командой for.
3 using System;
4
5 public class Sum
6 {
7     public static void Main( string[] args )
8     {
9         int total = 0; // Инициализация total
10
11         // Суммирование четных целых чисел от 2 до 20
12         for ( int number = 2; number <= 20; number += 2 )
13             total += number;
14
15         Console.WriteLine( "Sum is {0}", total ); // Вывод результата
16     } // Конец Main
17 } // Конец класса Sum
```

Sum is 110

**Ил. 6.5.** Суммирование целых чисел командой `for`

Выражения инициализации и приращения могут состоять из нескольких компонентов, разделенных запятыми; это позволяет использовать сразу несколько выражений инициализации или приращения. Например, тело команды `for` в строках 12–13 на ил. 6.5 можно включить в секцию приращения в заголовке:

```
for ( int number = 2; number <= 20; total += number, number += 2 )
    ; // Пустая команда
```

Мы *не рекомендуем* усложнять заголовки подобным образом.

### Вычисление сложных процентов

Следующее приложение использует команду `for` для вычисления сложных процентов. Постановка задачи выглядит так:

Человек кладет \$1000 на накопительный счет в банке под 5 % годовых, с ежегодным начислением процентов. Предполагается, что вся сумма процентов остается на счете. Вычислите и выведите сумму денег на счете в конце каждого года за 10-летний период. В расчетах используйте следующую формулу:

$$a = p (1 + r)^n$$

где  $p$  — исходная внесенная сумма;  $r$  — годовая процентная ставка (например, 0,05 для 5 %);  $n$  — число лет;  $a$  — сумма депозита на конец  $n$ -го года.

В задаче задействован цикл, выполняющий указанное вычисление для каждого года из 10 лет, в течение которых деньги остаются на счете. Решение задачи показано в листинге на ил. 6.6. В строках 9–11 метода Main объявляются переменные `amount` и `principal` типа `decimal` и переменная `rate` типа `double`. В строках 10–11 переменная `principal` инициализируется значением 1000 (представляющим \$1000,00), а переменная `rate` — значением 0,05. В C# вещественные константы вида 0.05 интерпретируются как значения типа `double`. Аналогичным образом C# интерпретирует целочисленные константы (такие, как 7 и 1000) как относящиеся к типу `int`. Когда переменная `principal` инициализируется значением 1000, значение 1000 типа `int` неявно преобразуется к типу `decimal` — явного преобразования не требуется.

```

1  // Ил. 6.6: Interest.cs
2  // Вычисление сложных процентов с использованием цикла for.
3  using System;
4
5  public class Interest
6  {
7      public static void Main( string[] args )
8      {
9          decimal amount; // Сумма на счете в конце каждого года
10         decimal principal = 1000; // Исходная сумма в начале периода
11         double rate = 0.05; // Процентная ставка
12
13         // Вывод заголовков
14         Console.WriteLine( "Year{0,20}", "Amount on deposit" );
15
16         // Вычисление суммы на счете за каждый год из 10 лет
17         for ( int year = 1; year <= 10; ++year )
18         {
19             // Вычисление новой суммы для заданного года
20             amount = principal *
21                 ( ( decimal ) Math.Pow( 1.0 + rate, year ) );
22
23             // Вывод года и суммы
24             Console.WriteLine( "{0,4}{1,20:C}", year, amount );
25         } // Конец for
26     } // Конец Main
27 } // Конец класса Interest

```

```

Year Amount on deposit

```

```

1  $1,050.00
2  $1,102.50
3  $1,157.63
4  $1,215.51
5  $1,276.28
6  $1,340.10
7  $1,407.10
8  $1,477.46
9  $1,551.33
10 $1,628.89

```

**Ил. 6.6.** Вычисление сложных процентов с использованием цикла for

В строке 14 выводятся заголовки двух столбцов вывода приложения. В первом столбце выводится год, а во втором — сумма на счете к концу года. Для вывода строки "Amount on deposit" используется форматный элемент {0,20}. Число 20 после запятой показывает, что значение должно выводиться с шириной поля 20 — то есть WriteLine выводит его минимум в 20 позициях символов. Если длина выводимого значения меньше 20 символов (17 символов в данном примере), значение по умолчанию выравнивается по правому краю поля (в этом случае значению предшествуют три пробела). Если выводимое значение year имеет ширину более четырех символов, то ширина поля увеличивается вправо, чтобы в нем помещалось все значение (что приведет к нарушению аккуратных столбцов нашего табличного вывода). Чтобы показать, что вывод должен выравниваться по левому краю, укажите отрицательную ширину поля.

Команда for (строки 17–25) выполняет свое тело 10 раз, с изменением управляющей переменной year от 1 до 10 с приращением 1. Цикл завершается, когда управляющая переменная year становится равной 11 (year соответствует n в постановке задачи).

Классы предоставляют методы для выполнения стандартных операций с объектами. Методы чаще всего вызываются для конкретного объекта. Например, чтобы вывести приветствие на ил. 4.2, мы вызываем метод DisplayMessage для объекта myGradeBook. Многие классы также предоставляют методы для выполнения обобщенных операций, которые вызываются не для конкретного объекта — при их вызове должно указываться имя класса. Такие методы называются *статическими*. Например, в C# не поддерживается оператор возведения в степень, поэтому проектировщики класса C# Math определили статический метод Pow. При вызове статического метода указывается имя класса, оператор обращения к члену класса (.) и имя метода:

```
ИмяКласса.ИмяМетода( аргументы )
```

Методы Write и WriteLine класса Console являются статическими. В главе 7 вы узнаете, как реализовать статические методы в ваших собственных классах.

Мы используем статический метод Pow класса Math для вычисления сложных процентов на ил. 6.6. Выражение Math.Pow(x, y) вычисляет результат возведения x в степень y. Метод получает два аргумента double и возвращает результат double. В строках 20–21 выполняется вычисление  $a = p(1 + r)^n$ , где a — сумма на счете, p — исходная сумма, r — процентная ставка, a n — год. Обратите внимание: при вычислении значение типа decimal (переменная principal) умножается на значение double (возвращаемое значение Math.Pow). Язык C# не выполняет неявного преобразования double в decimal или, наоборот, из-за возможной потери информации, поэтому в строке 21 применяется оператор (decimal), который явно преобразует возвращаемое значение Math.Pow в decimal.

После каждого вычисления в строке 24 выводится год и сумма на счете в конце этого года. Год выводится с шириной поля 4 символа (как указывает форматный элемент {0,4}). Сумма выводится как денежная величина с форматным элементом {1,20:c}. Число 20 означает, что значение должно быть выровнено по правому краю



поля шириной 20 символов. Форматный спецификатор `C` указывает на то, что число должно быть отформатировано как денежная величина.

Обратите внимание: переменные `amount` и `principal` объявлены с типом `decimal` вместо `double`. Об использовании типа `decimal` вместо `double` для финансовых вычислений было рассказано в разделе 4.11. Тип `decimal` также использовался для этой цели на ил. 6.6. Почему мы так поступаем? При работе с нецелыми денежными суммами нужен тип, поддерживающий дробные значения. К сожалению, числа с плавающей точкой типа `double` (или `float`) могут создать проблемы в финансовых вычислениях. Допустим, на компьютере хранятся две денежные суммы в переменных типа `double`: 14,234 (обычно округляемая до 14,23 при выводе) и 18,673 (обычно округляемая до 18,67 при выводе).

При сложении этих величин получается сумма 32,907, которая при выводе обычно округляется до 32,91. Таким образом, мы получаем:

```

14.23
+ 18.67
-----
32.91

```

Однако человек, складывающий числа в том виде, в котором они выводятся, рассчитывает получить 32,90. Будьте внимательны!



#### КАК ИЗБЕЖАТЬ ОШИБОК 6.5

Не используйте переменные типа `double` (или `float`) для выполнения точных финансовых вычислений; используйте тип `decimal`. Неточность чисел с плавающей точкой может создать погрешность, которая приведет к ошибкам в денежных суммах.

Тело команды `for` содержит выражение `1.0 + rate`, которое передается в аргументе метода `Math.Pow`. Собственно, это вычисление дает один и тот же результат при каждой итерации, так что каждый раз вычислять его заново было бы неэффективно.



#### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 6.2

По возможности избегайте в циклах выражений, результат которых никогда не изменяется, — такие выражения обычно следует выполнять до начала цикла. Оптимизирующие компиляторы обычно выносят такие вычисления в откомпилированном коде за пределы цикла.

## 6.5. Циклы do...while

Команда повторения `do...while` похожа на команду `while`. В команде `while` условие продолжения цикла проверяется в начале цикла, до выполнения его тела. Если условие ложно, то тело цикла не выполняется. Команда `do...while` проверяет условие продолжения цикла после выполнения его тела; таким образом, тело цикла всегда выполняется по крайней мере один раз. При завершении команды `do...while`

выполнение продолжается со следующей команды. На ил. 6.7 цикл `do...while` (строки 11–15) используется для вывода чисел 1–10.

```
1 // Ил. 6.7: DoWhileTest.cs
2 // Команда повторения do...while.
3 using System;
4
5 public class DoWhileTest
6 {
7     public static void Main( string[] args )
8     {
9         int counter = 1; // Инициализация счетчика
10
11         do
12         {
13             Console.Write( "{0} ", counter );
14             ++counter;
15         } while ( counter <= 10 ); // Конец do...while
16
17         Console.WriteLine(); // Перевод строки
18     } // Конец Main
19 } // Конец класса DoWhileTest
```

1 2 3 4 5 6 7 8 9 10

**Ил. 6.7.** Команда повторения `do...while`

В строке 9 объявляется и инициализируется управляющая переменная `counter`. При входе в цикл `do...while` строка 13 выводит текущее значение `counter`, а строка 14 увеличивает переменную. Затем приложение проверяет условие продолжения в конце цикла (строка 15). Если условие истинно, то цикл продолжается с первой команды его тела (строка 13). Если условие ложно, то цикл завершается, а приложение продолжает выполняться со следующей команды после цикла.

На ил. 6.8 приведена диаграмма активности UML для команды `do...while`. Из нее ясно видно, что условие продолжения цикла проверяется только после того, как цикл выполнит свое действие как минимум один раз. Сравните ее с диаграммой активности цикла `while` (ил. 5.4). Если тело цикла состоит только из одной команды, использовать фигурные скобки в `do...while` не обязательно. Однако многие программисты включают фигурные скобки, чтобы избежать путаницы между командами `while` и `do...while`. Например,

```
while ( условие )
```

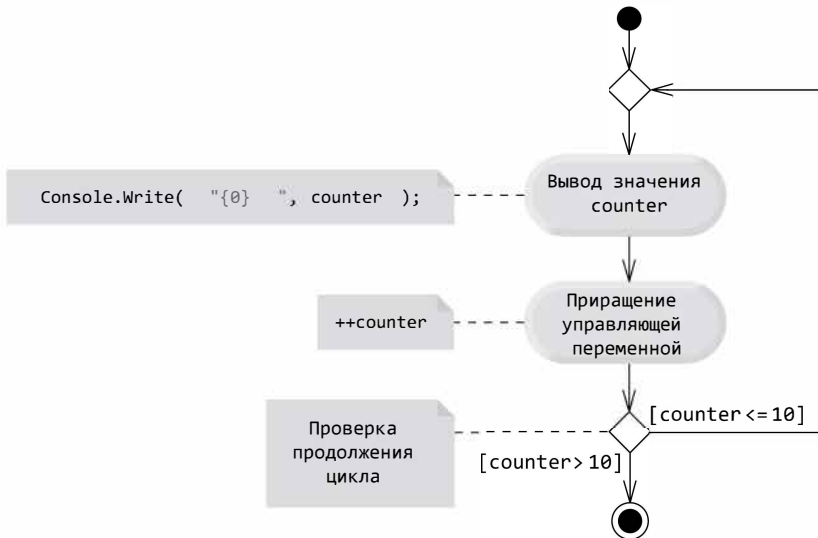
обычно является первой строкой команды `while`. Команда `do...while` без фигурных скобок, заключающих тело с одной командой, выглядит так:

```
do
    команда
while ( условие );
```

Все это может создать путаницу, если читатель кода ошибочно интерпретирует последнюю строку — `while( условие );` — как команду `while` с пустым телом. Чтобы

избежать недоразумений, можно записать команду `do...while` с телом из одной команды следующим образом:

```
do
{
    команда
} while ( условие );
```



Ил. 6.8. Диаграмма активности UML для цикла `do...while`

## 6.6. Команда множественного выбора switch

В главе 5 мы уже рассмотрели команду одиночного выбора `if` и команду двойного выбора `if...else`. В C# также предусмотрена команда множественного выбора для выполнения разных действий в зависимости от возможных значений выражения. Каждое действие связывается с отдельным значением константного целочисленного или строкового выражения, которое может принять заданная переменная или выражение. Константное целочисленное выражение представляет собой любое выражение с символьными и целочисленными константами, результатом вычисления которого является целочисленное значение — то есть значение типа `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` и `char`, или константа перечислимого типа (перечислимые типы рассматриваются в разделе 7.10). Константное строковое выражение представляет собой выражение, составленное из строковых литералов, результатом которого всегда является одна и та же строка.

### Класс `GradeBook` с переключением по оценкам

На ил. 6.9 представлена усовершенствованная версия класса `GradeBook`, который был представлен в главе 4 и далее разрабатывался в главе 5. Эта версия класса не только

вычисляет среднюю оценку по набору числовых оценок, введенных пользователем, но и использует команду `switch` для определения эквивалента каждой оценки (А, В, С, D или F), после чего увеличивает соответствующий счетчик. Класс также выводит информацию о количестве студентов, получивших каждую оценку. На ил. 6.10 показан пример ввода и вывода приложения `GradeBookTest`, использующего класс `GradeBook` для обработки набора оценок.

```

1  // Ил. 6.9: GradeBook.cs
2  // Класс GradeBook, использующий команду switch для подсчета оценок.
3  using System;
4
5  public class GradeBook
6  {
7      private int total; // Сумма оценок
8      private int gradeCounter; // Количество введенных оценок
9      private int aCount; // Счетчик оценок A
10     private int bCount; // Счетчик оценок B
11     private int cCount; // Счетчик оценок C
12     private int dCount; // Счетчик оценок D
13     private int fCount; // Счетчик оценок F
14
15     // Автоматически реализуемое свойство CourseName
16     public string CourseName { get; set; }
17
18     // Конструктор инициализирует автоматическое свойство CourseName;
19     // переменные экземпляров int инициализируются 0 по умолчанию
20     public GradeBook( string name )
21     {
22         CourseName = name; // Присвоить name свойству CourseName
23     } // end constructor
24
25     // Вывод приветствия для пользователя GradeBook
26     public void DisplayMessage()
27     {
28         // CourseName присваивается название учебного курса
29         Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
30             CourseName );
31     } // Конец метода DisplayMessage
32
33     // Ввод произвольного количества оценок пользователем
34     public void InputGrades()
35     {
36         int grade; // Оценка, введенная пользователем
37         string input; // Текст, введенный пользователем
38
39         Console.WriteLine( "{0}\n{1}",
40             "Enter the integer grades in the range 0-100.",
41             "Type <Ctrl> z and press Enter to terminate input:" );
42
43         input = Console.ReadLine(); // Ввод оценки пользователем
44
45         // Цикл до ввода пользователем признака конца файла (<Ctrl> z)
46         while ( input != null )

```

**Ил. 6.9.** Класс `GradeBook`, использующий команду `switch` для подсчета оценок  
(продолжение ↗)

```

47     {
48         grade = Convert.ToInt32( input ); // Получение оценки
49         total += grade; // Прибавление grade к total
50         ++gradeCounter; // Увеличение количества оценок
51
52         // Вызов метода для увеличения соответствующего счетчика
53         IncrementLetterGradeCounter( grade );
54
55         input = Console.ReadLine(); // Ввод оценки пользователем
56     } // Конец while
57 } // Конец метода InputGrades
58
59 // Увеличение счетчика заданной оценки на 1
60 private void IncrementLetterGradeCounter( int grade )
61 {
62     // Определение введенной оценки
63     switch ( grade / 10 )
64     {
65         case 9: // Значение grade в диапазоне от 90 до 99
66         case 10: // Значение grade равно 100
67             ++aCount; // Увеличение aCount
68             break; // Необходимо для выхода из switch
69         case 8: // Значение grade в диапазоне от 80 до 89
70             ++bCount; // Увеличение bCount
71             break; // Выход из switch
72         case 7: // Значение grade в диапазоне от 70 до 79
73             ++cCount; // Увеличение cCount
74             break; // Выход из switch
75         case 6: // Значение grade в диапазоне от 60 до 69
76             ++dCount; // Увеличение dCount
77             break; // Выход из switch
78         default: // Значение grade меньше 60
79             ++fCount; // Увеличение fCount
80             break; // Выход из switch
81     } // Конец switch
82 } // Конец метода IncrementLetterGradeCounter
83
84 // Вывод отчета по оценкам, введенным пользователем
85 public void DisplayGradeReport()
86 {
87     Console.WriteLine( "\nGrade Report:" );
88
89     // Если пользователь ввел хотя бы одну оценку...
90     if ( gradeCounter != 0 )
91     {
92         // Вычисление среднего значения по всем введенным оценкам
93         double average = ( double ) total / gradeCounter;
94
95         // Вывод сводки результатов
96         Console.WriteLine( "Total of the {0} grades entered is {1}",
97             gradeCounter, total );
98         Console.WriteLine( "Class average is {0:F}", average );
99         Console.WriteLine( "{0}A: {1}\nB: {2}\nC: {3}\nD: {4}\nF: {5}",
100             "Number of students who received each grade:\n",

```

**Ил. 6.9.** Класс GradeBook, использующий команду switch для подсчета оценок  
(продолжение ↗)

```
101         aCount, // Вывод количества оценок категории A
102         bCount, // Вывод количества оценок категории B
103         cCount, // Вывод количества оценок категории C
104         dCount, // Вывод количества оценок категории D
105         fCount ); // Вывод количества оценок категории F
106     } // Конец if
107     else // Ни одной оценки не введено, вывести сообщение
108         Console.WriteLine( "No grades were entered" );
109 } // Конец метода DisplayGradeReport
110 } // Конец класса GradeBook
```

**Ил. 6.9.** Класс `GradeBook`, использующий команду `switch` для подсчета оценок (окончание)

### Переменные экземпляров

Класс `GradeBook` (см. ил. 6.9) объявляет переменные экземпляров `total` (строка 7) и `gradeCounter` (строка 8), в которых соответственно хранятся сумма и количество оценок, введенных пользователем. Строки 9–13 объявляют переменные-счетчики для всех категорий оценок. Класс `GradeBook` хранит `total`, `gradeCounter` и пять счетчиков категорий в переменных экземпляров, чтобы их можно было использовать или изменять в любом из методов класса.

### Свойство `CourseName`, метод `DisplayMessage` и конструктор

Класс `GradeBook`, как и его более ранние версии, объявляет автоматическое свойство `CourseName` (строка 16) и метод `DisplayMessage` (строки 26–31) для вывода приветствия. Класс также содержит конструктор (строки 20–23), инициализирующий название учебного курса.

В конструкторе задается только название курса — остальные семь переменных экземпляров относятся к типу `int` и по умолчанию инициализируются нулями.

### Методы `InputGrades` и `DisplayGradeReport`

Класс `GradeBook` содержит три дополнительных метода — `InputGrades`, `IncrementLetterGradeCounter` и `DisplayGradeReport`. Метод `InputGrades` (строки 34–57) читает произвольное количество целочисленных оценок, вводимых пользователем; при этом используется повторение со сторожевым значением, с обновлением переменных экземпляров `total` и `gradeCounter`. Метод `InputGrades` вызывает метод `IncrementLetterGradeCounter` (строки 60–82), обновляющий соответствующий счетчик буквенных оценок для каждой введенной оценки. Класс `GradeBook` также содержит метод `DisplayGradeReport` (строки 85–109), который выводит отчет с суммой всех введенных оценок, средней оценкой и количеством студентов, получивших оценки из каждой категории. Рассмотрим эти методы более подробно.

В строках 36–37 метода `InputGrades` объявляются переменные `grade` и `input`, которые сначала сохраняют ввод пользователя в виде строки (в переменной `input`), а затем преобразуют его в `int` для сохранения в переменной `grade`. В строках 39–41 пользователю предлагается ввести целочисленные оценки, а затем нажать клавиши `Ctrl+z` и `Enter` для завершения ввода (`Ctrl+z` — комбинация клавиш `Windows`,

обозначающая конец файла). Это один из способов сообщить приложению о том, что данных для ввода больше не осталось. Если комбинация `Ctrl+z` будет введена в то время, когда приложение ожидает ввода методом `ReadLine`, возвращается `null`. (Комбинация клавиш, выполняющая функции признака конца файла, зависит от системы. Во многих системах, не входящих в семейство Windows, конец файла обозначается комбинацией `Ctrl+d`.) В главе 17 мы рассмотрим пример использования признака конца файла при чтении данных из файла. Обычно при вводе признака конца файла в окне командной строки Windows отображаются символы `^Z`, как видно из ил. 6.10.

В строке 43 метод `ReadLine` используется для получения первой строки, введенной пользователем, и ее сохранения в переменной `input`. Команда `while` (строки 46–56) обрабатывает введенные данные. Условие в строке 46 проверяет, содержит ли переменная `input` значение `null`. Метод `ReadLine` класса `Console` возвращает `null` только в том случае, если пользователь ввел признак конца файла. Если признак конца файла не вводился, значение `input` отлично от `null`, и проверка проходит успешно.

В строке 48 содержимое `input` преобразуется в значение типа `int`. В строке 49 `grade` прибавляется к `total`. Строка 50 увеличивает `gradeCounter`. Метод `DisplayGradeReport` использует эти переменные для вычисления средней оценки. В строке 53 вызывается метод `IncrementLetterGradeCounter` (объявленный в строках 60–82) для увеличения счетчика соответствующей категории, определенной по введенному числовому значению.

### Метод `IncrementLetterGradeCounter`

Метод `IncrementLetterGradeCounter` содержит команду `switch` (строки 63–81), которая определяет, какой из счетчиков следует увеличить. В нашем примере предполагается, что пользователь вводит действительную оценку в диапазоне 0–100. Оценки из диапазона 90–100 соответствуют категории А, диапазон 80–89 соответствует категории В, диапазон 70–79 соответствует категории С, диапазон 60–69 соответствует категории D, и диапазон 0–59 соответствует категории F. Команда `switch` состоит из блока, содержащего набор меток `case` и необязательную метку `default`. Они используются в нашем примере для определения увеличиваемого счетчика на основании введенной оценки.

### Команда `switch`

Когда управление передается команде `switch`, приложение вычисляет выражение в круглых скобках (`grade/10`) за ключевым словом `switch` (оно называется выражением `switch`). Приложение пытается найти значение выражения среди меток `case`. Выражение `switch` в строке 63 выполняет целочисленное деление, при котором дробная часть результата теряется. Таким образом, при делении любого значения из диапазона 0–100 на 10 результат всегда является числом от 0 до 10. Некоторые из этих чисел используются в качестве меток `case`. Например, если пользователь ввел целое число 85, результат выражения `switch` равен 8. Если между выражением `switch` и секцией `case` находится совпадение (`case 8:` в строке 69), приложение выполняет команды для этого случая. Для целого значения 8 строка 70 обновляет счетчик

bCount, потому что оценки в диапазоне 80–89 соответствуют категории В. Команда `break` (строка 71) осуществляет переход к первой команде после `switch` — в нашем приложении при этом достигается конец метода `IncrementLetterGradeCounter`, так что управление возвращается в строку 55 метода `InputGrades` (первая строка после вызова `IncrementLetterGradeCounter`). Эта строка использует метод `ReadLine` для чтения следующей оценки, введенной пользователем, и ее присваивания переменной `input`. Строка 56 отмечает конец тела команды `while`, в которой вводятся оценки (строки 46–56), так что управление передается условию `while` (строка 46) для проверки продолжения цикла в зависимости от значения, только что присвоенного переменной `input`.

### Последовательные метки `case`

В команде `switch` из нашего примера явно проверяются значения 10, 9, 8, 7 и 6. Обратите внимание на метки `case` в строках 65–66, проверяющих значения 9 и 10 (оба соответствуют категории А). Последовательное перечисление меток позволяет выполнить для них один набор команд — если выражение `switch` равно 9 или 10, то выполняются команды в строках 67–68. Команда `switch` не поддерживает проверку диапазонов значений, так что каждое проверяемое значение должно быть включено в отдельную секцию `case`.

Каждая секция `case` может содержать несколько команд. Конструкция `switch` отличается от других управляющих команд тем, что она не требует заключать команды каждой секции в фигурные скобки.

### Отсутствие сквозного выполнения в `C#`

В `C`, `C++` и многих других языках программирования, поддерживающих команду `switch`, присутствие команды `break` в конце `case` не обязательно. Без команд `break` при обнаружении совпадения выполняются все команды этой и последующих секций `case` вплоть до команды `break` или конца `switch`. Так называемое «сквозное выполнение» (`fall through`) приводит к логическим ошибкам, если вы забудете вставить команду `break`. `C#` отличается от других языков программирования — после основных команд `case` необходимо вставить завершающую команду: `break`, `return` или `throw`; в противном случае происходит ошибка компиляции (команда `throw` рассматривается в главе 13).

### Секция `default`

Если среди меток `case` не будет найдено совпадение для выражения `switch`, выполняются команды, следующие после метки `default` (строки 79–80). В нашем примере секция `default` обрабатывает все значения выражения `switch`, меньшие 6. Если совпадения не найдено и команда `switch` не содержит секции `case`, управление просто передается первой команде после `switch`.

### Класс `GradeBookTest` для тестирования `GradeBook`

Класс `GradeBookTest` (ил. 6.10) создает объект `GradeBook` (строки 10–11). Строка 13 вызывает метод `DisplayMessage` объекта, чтобы вывести приветствие для пользователя. Вызов метода `InputGrades` объекта в строке 14 читает набор оценок, введенных пользователем, и сохраняет сумму и количество всех введенных оценок.



Вспомните, что метод `InputGrades` также вызывает метод `IncrementLetterGradeCounter` для отслеживания количества студентов, получивших оценку каждой категории. В строке 15 вызывается метод `DisplayGradeReport` класса `GradeBook`, который выдает отчет по введенным оценкам. Строка 90 класса `GradeBook` (см. ил. 6.9) проверяет, ввел ли пользователь хотя бы одну оценку — это необходимо для предотвращения деления на нуль. Если оценки были введены, в строке 93 вычисляется средняя оценка. В строках 96–105 выводится сумма всех оценок, средняя оценка класса и количество студентов, получивших оценки каждой категории. Если оценок нет, в строке 108 выводится соответствующее сообщение. На ил. 6.10 приведен пример отчета для 9 введенных оценок.

```

1 // Ил. 6.10: GradeBookTest.cs
2 // Создание объекта GradeBook, ввод оценок и вывод отчета.
3
4 public class GradeBookTest
5 {
6     public static void Main( string[] args )
7     {
8         // Создание объекта myGradeBook класса GradeBook
9         // с передачей названия курса конструктору
10        GradeBook myGradeBook = new GradeBook(
11            "CS101 Introduction to C# Programming"
12        )
13        myGradeBook.DisplayMessage(); // Вывод приветствия
14        myGradeBook.InputGrades(); // Получение оценок у пользователя
15        myGradeBook.DisplayGradeReport(); // Вывод отчета по оценкам
16    } // end Main
17 } // end class GradeBookTest

```

```

Welcome to the grade book for
CS101 Introduction to C# Programming!
Enter the integer grades in the range 0-100.
Type <Ctrl> z and press Enter to terminate input:

```

```

99
92
45
100
57
63
76
14
92
^Z

```

```

Grade Report:
Total of the 9 grades entered is 638
Class average is 70.89
Number of students who received each grade:
A: 4
B: 0
C: 1
D: 1
F: 3

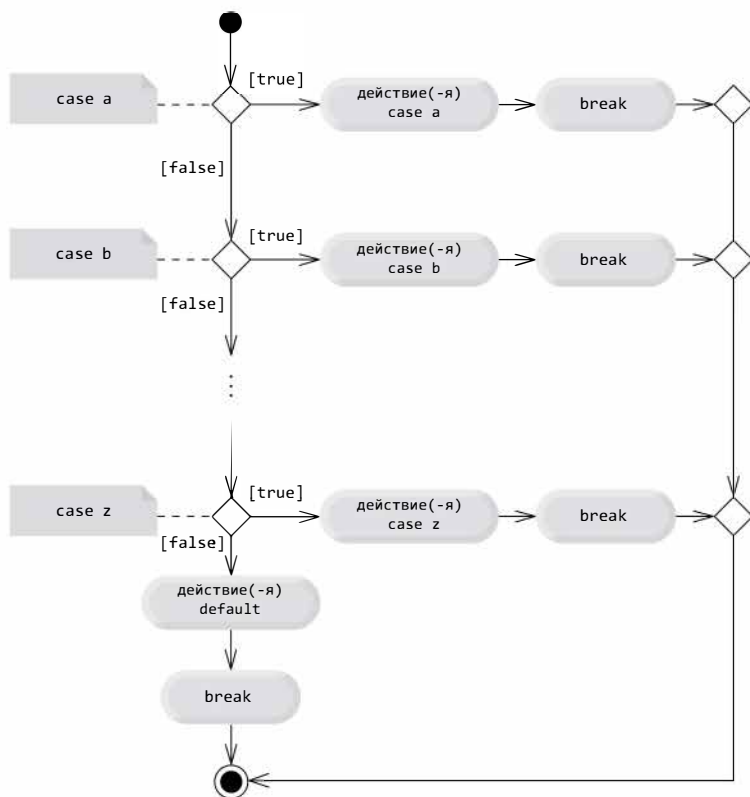
```

**Ил. 6.10.** Создание объекта `GradeBook`, ввод оценок и вывод отчета

Класс `GradeBookTest` (см. ил. 6.10) не вызывает метод `IncrementLetterGradeCounter` класса `GradeBook` напрямую (строки 60–82 на ил. 6.9). Этот метод используется исключительно методом `InputGrades` класса `GradeBook` для обновления счетчика оценок соответствующей категории при вводе новой оценки. Метод `IncrementLetterGradeCounter` существует только для поддержки работы других методов класса `GradeBook`, поэтому он объявляется закрытым. Члены класса, объявленные с модификатором `private`, доступны *только* для членов того же класса, в котором они были объявлены. Приватные методы часто называются *вспомогательными* методами, потому что не предназначены для самостоятельного использования за пределами класса.

### Диаграмма активности UML для команды switch

На ил. 6.11 представлена диаграмма активности UML для обобщенной команды `switch`. Каждый набор команд после метки `case` обычно завершается командой `break` или `return`, которая завершает выполнение `switch` после обработки `case`. Чаще для этой цели используются команды `break`. На ил. 6.11 этот факт подчеркивается включением команд `break` в диаграмму активности. Из диаграммы ясно видно, что команда `break` в конце `case` немедленно выводит управление за пределы команды `switch`.



Ил. 6.11. Команда множественного выбора `switch` с командами `break`



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 6.1

Хотя секции case и default в команде switch могут следовать в произвольном порядке, разместите секцию default — это сделает команду более понятной.

Помните, что в метках case могут использоваться только константные целочисленные или строковые выражения (то есть произвольные комбинации констант, результатом вычисления которых является константное значение целочисленного или строкового типа). Целочисленная константа представляет собой простое целочисленное значение (например, `-7`, `0` или `221`). Также допускается использование символьных констант — символов, заключенных в апострофы (`'A'`, `'7'` или `'$'`), представляющих целочисленные коды символов. Строковая константа (или строковый литерал) представляет собой последовательность символов, заключенную в кавычки (например, `"Welcome to C# Programming!"`).

В секциях case также могут использоваться *константы* — значения, не изменяющиеся во время работы приложения. Константы объявляются с ключевым словом `const` (см. главу 7). В C# также существуют *перечислимые типы*, которые тоже будут рассмотрены в главе 7. Константы перечислимых типов также могут использоваться в метках case. В главе 12 мы рассмотрим более элегантный способ реализации логики switch — мы воспользуемся *полиморфизмом* для создания приложений, код которых часто оказывается более понятным, простым в сопровождении и расширении, чем в приложениях с использованием логики switch.

## 6.7. Команды break и continue

Кроме команд выбора и повторения, для управления потоком операций в C# также используются команды `break` и `continue`. В предыдущем разделе было продемонстрировано использование `break` для управления выполнением switch. В этом разделе вы узнаете, как использовать `break` для завершения любой команды цикла.

### Команда break

Команда `break`, выполненная в цикле `while`, `for`, `do...while`, `switch` или `foreach`, приводит к немедленному выходу из цикла. Обычно выполнение продолжается с первой команды после цикла — Впрочем, после знакомств с другими типами команд C# вы увидите, что существуют и другие возможности. Команда `break` чаще всего используется для преждевременного выхода из команды повторения или пропуска оставшейся части switch (как на ил. 6.9). На ил. 6.12 продемонстрировано применение команды `break` в цикле `for`.

```
1 // Ил. 6.12: BreakTest.cs
2 // Использование команды break для выхода из цикла for.
3 using System;
4
5 public class BreakTest
```

**Ил. 6.12.** Использование команды `break` для выхода из цикла `for` (продолжение ↗)

```

6 {
7     public static void Main( string[] args )
8     {
9         int count; // Переменная также используется после завершения цикла
10
11         for ( count = 1; count <= 10; ++count ) // Выполнить 10 раз
12         {
13             if ( count == 5 ) // Если переменная count равна 5
14                 break; // Завершение цикла
15
16             Console.Write( "{0} ", count );
17         } // Конец for
18
19         Console.WriteLine( "\nBroke out of loop at count = {0}", count );
20     } // Конец Main
21 } // Конец класса BreakTest

```

```

1 2 3 4
Broke out of loop at count = 5

```

#### Ил. 6.12. Использование команды break для выхода из цикла for (окончание)

Если условие `if` в строке 13 команды `for` (строки 11–17) определяет, что значение `count` равно 5, выполняется команда `break` в строке 14. При этом команда `for` завершается, а приложение переходит к строке 19 (следующей за командой `for`), в которой выводится сообщение с указанием значения управляющей переменной при завершении цикла. Из-за команды `break` тело цикла полностью выполняется всего 4 раза вместо 10.

#### Команда `continue`

Команда `continue`, выполняемая в цикле `while`, `for`, `do...while` или `foreach`, пропускает оставшиеся команды в теле цикла и переходит к следующей итерации. В командах `while` и `do...while` приложение сразу же после команды `continue` проверяет условие продолжения цикла. В команде `for` обычно следующим выполняется выражение приращения, а затем проверяется условие продолжения цикла.

На ил. 6.13 команда `continue` используется в команде `for` для пропуска команды в строке 14, когда вложенная команда `if` (строка 11) определяет, что значение `count` равно 5. При выполнении команды `continue` выполнение программы продолжается с приращения управляющей переменной в команде `for` (строка 9).

```

1 // Ил. 6.13: ContinueTest.cs
2 // Команда continue завершает итерацию команды for.
3 using System;
4
5 public class ContinueTest
6 {
7     public static void Main( string[] args )
8     {
9         for ( int count = 1; count <= 10; ++count ) // Выполнить 10 раз
10        {
11            if ( count == 5 ) // Если переменная count равна 5

```

#### Ил. 6.13. Команда `continue` завершает итерацию цикла `for` (продолжение ↗)

```

12         continue; // Пропуск оставшегося кода итерации
13
14         Console.Write( "{0} ", count );
15     } // Конец for
16
17     Console.WriteLine( "\nUsed continue to skip displaying 5" );
18 } // Конец Main
19 } // Конец класса ContinueTest

```

```

1 2 3 4 6 7 8 9 10
Used continue to skip displaying 5

```

### Ил. 6.13. Команда continue завершает итерацию цикла for (окончание)

В разделе 6.3 было сказано, что команда `while` часто может использоваться вместо `for`. Одно из возможных исключений — когда выражение приращения в `while` следует за командой `continue`. В этом случае приращение не выполняется до проверки условия продолжения цикла, так что цикл `while` работает не так, как `for`.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 6.2

Некоторые программисты считают, что команды `break` и `continue` нарушают принципы структурного программирования. Так как того же результата можно добиться при помощи средств структурного программирования, эти программисты предпочитают обходиться без команд `break` и `continue`.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 6.3

Между качественным проектированием программных продуктов и обеспечением оптимального быстродействия существует определенное противоречие. Часто одна из этих целей достигается в ущерб другой. Всегда, кроме ситуаций с предельно жесткими требованиями к быстродействию, руководствуйтесь следующим правилом: сначала сделайте свой код простым и работоспособным; затем займитесь ускорением его работы, но только если это действительно необходимо.

## 6.8. Логические операторы

В каждой из команд `if`, `if...else`, `while`, `do...while` и `for` должно присутствовать условие, управляющее продолжением потока операций приложения. До настоящего момента мы рассматривали только простые условия типа `count <= 10`, `number != sentinelValue` или `total > 1000`. Простые условия выражаются с использованием операторов сравнения `>`, `<`, `>=` и `<=`, а также операторов проверки равенства `==` и `!=`. Каждое выражение проверяет только одно условие. Для проверки нескольких условий в процессе принятия решения нам приходилось выполнять дополнительные проверки в отдельных командах или вложенных командах `if` или `if...else`. Иногда в управляющих командах поток операций должен определяться более сложными условиями.

В С# поддерживаются логические операторы, позволяющие строить сложные условия из нескольких простых. В эту категорию входят операторы `&&` (условный

оператор И), || (условный оператор ИЛИ), & (булевский оператор И), | (булевский оператор ИЛИ), ^ (булевский исключающий оператор ИЛИ) и ! (логический оператор отрицания).

### Условный оператор И (&&)

Предположим, перед входом в некоторую ветвь приложения мы хотим убедиться в том, что выполняются два заданных условия. Для этого можно воспользоваться оператором && (условным оператором И):

```
if ( gender == 'F' && age >= 65 )
    ++seniorFemales;
```

Команда if состоит из двух простых условий: `gender == 'F'` и `age >= 65`. Таким образом, для команды if комбинированное условие

```
gender == 'F' && age >= 65
```

истинно тогда и только тогда, когда истинны оба простых условия. Если объединенное условие истинно, то тело команды if увеличивает `seniorFemales` на 1. Если хотя бы одно из двух условий оказывается ложным, приложение пропускает команду с инкрементом. Некоторые программисты считают, что приведенное условие лучше читается с добавлением круглых скобок — строго говоря, избыточных:

```
( gender == 'F' ) && ( age >= 65 )
```

В таблице на ил. 6.14 приведена сводка значений оператора &&. В таблице представлены все четыре возможные комбинации значений `true` и `false` для выражений *выражение1* и *выражение2*. Такие таблицы называются *таблицами истинности*. В C# при вычислении выражений, включающих операторы сравнения, операторы проверки равенства и логические операторы, образуется значение логического типа `bool` — то есть `true` или `false`.

выражение1	выражение2	выражение1 && выражение2
false	false	false
false	true	false
true	false	false
true	true	true

**Ил. 6.14.** Таблица истинности оператора && (условный оператор И)

### Условный оператор ИЛИ (||)

Допустим, перед выбором некоторой ветви выполнения мы хотим удостовериться в том, что истинно *хотя бы одно* из двух условий. В этом случае используется оператор || (условный оператор ИЛИ), как в следующем фрагменте:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
    Console.WriteLine ( "Student grade is A" );
```

Эта команда также содержит два простых условия: `semesterAverage >= 90` и `finalExam >= 90`. Для команды if комбинированное условие

```
( semesterAverage >= 90 ) || ( finalExam >= 90 )
```

истинно в том случае, если истинно одно или оба условия. Сообщение `writeln` не будет выведено только в одном случае: если оба простых условия ложны. На ил. 6.15 приведена таблица истинности для условного оператора ИЛИ (`||`). Оператор `&&` обладает более высоким приоритетом, чем оператор `||`. Оба оператора применяются слева направо.

выражение1	выражение2	выражение1 && выражение2
false	false	false
false	true	true
true	false	true
true	true	true

**Ил. 6.15.** Таблица истинности оператора `||` (условный оператор ИЛИ)

### Ускоренное вычисление сложных условий

Части выражения, содержащего операторы `&&` или `||`, вычисляются только до того момента, пока не станет ясно, истинно условие или ложно. Таким образом, обработка выражения

```
( gender == 'F' ) && ( age >= 65 )
```

немедленно прерывается, если значение `gender` отлично от `'F'` (то есть когда становится ясно, что все выражение заведомо ложно). Если переменная `gender` равна `'F'`, то вычисление продолжается (поскольку все выражение еще может быть истинным, если истинно условие `age >= 65`). Эта особенность выражений с условными операторами `И` и `ИЛИ` называется *ускоренным вычислением*.



#### ТИПИЧНАЯ ОШИБКА 6.3

В выражениях, использующих оператор `&&`, одно условие может иметь смысл только в том случае, если другое условие будет проверено и окажется истинным. В таких ситуациях зависимое выражение следует ставить на второе место, чтобы избежать ошибок. Например, в выражении `(i != 0) && (10 / i == 2)` второе условие должно располагаться после первого, иначе может возникнуть ошибка деления на ноль.

### Булевские операторы `И (&)` и `ИЛИ (|)`

Булевские операторы `И (&)` и `ИЛИ (|)` похожи на операторы `&&` (условный оператор `И`) и `||` (условный оператор `ИЛИ`), с одним исключением — у булевских операторов всегда обрабатываются оба операнда (то есть ускоренное вычисление с ними не применяется). Таким образом, в выражении

```
( gender == 'F' ) & ( age >= 65 )
```

условие `age >= 65` проверяется независимо от значения `gender`. Это может быть полезно, если правый операнд булевского оператора `И` или `ИЛИ` имеет необходимый побочный эффект — например, в нем изменяется значение переменной. Например, выражение

```
( birthday == true ) | ( ++age >= 65 )
```

гарантирует, что условие `++age >= 65` будет обработано. Соответственно, значение переменной `age` будет увеличено независимо от того, ложно все выражение или истинно.



### КАК ИЗБЕЖАТЬ ОШИБОК 6.6

Старайтесь избегать выражений с побочными эффектами в условиях. Такие конструкции выглядят впечатляюще, но они усложняют понимание кода и порождают коварные логические ошибки.

## Булевский исключающий оператор ИЛИ (^)

Сложное условие с булевым исключающим оператором ИЛИ (^) истинно только в том случае, если один из его операндов является истинным, а другой ложным. Если оба операнда истинны или оба ложны, то все условие ложно.

На ил. 6.16 приведена таблица истинности для булевского исключающего оператора ИЛИ (^). Этот оператор также заведомо вычисляет оба своих операнда.

выражение1	выражение2	выражение1 && выражение2
false	false	false
false	true	true
true	false	true
true	true	false

**Ил. 6.16.** Таблица истинности оператора ^ (исключающий оператор ИЛИ)

## Логический оператор отрицания (!)

Оператор ! (логическое отрицание) позволяет изменить значение условия на противоположное. В отличие от логических операторов `&&`, `||`, `&`, `|` и `^` — бинарных операторов, объединяющих два условия, логический оператор отрицания является унарным и имеет только один операнд. Логический оператор отрицания размещается перед условием для выбора ветви выполнения в том случае, если исходное условие (без логического отрицания) окажется ложным, как в следующем фрагменте:

```
if ( ! ( grade == sentinelValue ) )  
    Console.WriteLine( "The next grade is {0}", grade );
```

Вызов `writeLine` выполняется только в том случае, если значение `grade` не равно `sentinelValue`. Круглые скобки, в которые заключено условие `grade==sentinelValue`, необходимы, потому что логический оператор отрицания обладает большим приоритетом, чем оператор проверки равенства.

В большинстве случаев без логического отрицания можно обойтись за счет изменения формулировки исходного условия. Например, предыдущую команду можно записать в следующем виде:

```
if ( grade != sentinelValue )  
    Console.WriteLine( "The next grade is {0}", grade );
```



Такая гибкость позволяет представить выражение более удобным способом. На ил. 6.17 приведена таблица истинности для логического оператора отрицания.

выражение	!выражение
false	true
true	false

**Ил. 6.17.** Таблица истинности оператора ! (логическое отрицание)

### Пример использования логических операторов

На ил. 6.18 использование логических и булевских операторов продемонстрировано на примере построения их таблиц истинности. В результатах выводится вычисляемое выражение и его результат типа bool. В строках 10–14 строится таблица истинности для оператора && (условный оператор И). В строках 17–21 строится таблица истинности для оператора || (условный оператор ИЛИ), в строках 24–28 — для оператора & (булевский оператор И), в строках 31–36 — для оператора | (булевский оператор ИЛИ), в строках 39–44 — для оператора ^ (булевский исключающий оператор ИЛИ) и в строках 47–49 — для оператора ! (логическое отрицание).

```

1 // Ил. 6.18: LogicalOperators.cs
2 // Логические операторы
3 using System;
4
5 public class LogicalOperators
6 {
7     public static void Main( string[] args )
8     {
9         // Построение таблицы истинности для оператора &&
10        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
11            "Conditional AND (&&)", "false && false", ( false && false ),
12            "false && true", ( false && true ),
13            "true && false", ( true && false ),
14            "true && true", ( true && true ) );
15
16        // Построение таблицы истинности для оператора ||
17        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
18            "Conditional OR (||)", "false || false", ( false || false ),
19            "false || true", ( false || true ),
20            "true || false", ( true || false ),
21            "true || true", ( true || true ) );
22
23        // Построение таблицы истинности для оператора &
24        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
25            "Boolean logical AND (&)", "false & false", ( false & false ),
26            "false & true", ( false & true ),
27            "true & false", ( true & false ),
28            "true & true", ( true & true ) );
29
30        // Построение таблицы истинности для оператора |
31        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",

```

**Ил. 6.18.** Логические операторы (продолжение ☞)

```

32     "Boolean logical inclusive OR (|)",
33     "false | false", ( false | false ),
34     "false | true", ( false | true ),
35     "true | false", ( true | false ),
36     "true | true", ( true | true ) );
37
38     // Построение таблицы истинности для оператора ^
39     Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
40     "Boolean logical exclusive OR (^)",
41     "false ^ false", ( false ^ false ),
42     "false ^ true", ( false ^ true ),
43     "true ^ false", ( true ^ false ),
44     "true ^ true", ( true ^ true ) );
45
46     // Построение таблицы истинности для оператора !
47     Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}",
48     "Logical negation (!)", "!false", ( !false ),
49     "!true", ( !true ) );
50 } // Конец Main
51 } // Конец класса LogicalOperators

```

```

Conditional AND (&)
false && false: False
false && true: False
true && false: False
true && true: True

```

```

Conditional OR (||)
false || false: False
false || true: True
true || false: True
true || true: True

```

```

Boolean logical AND (&)
false & false: False
false & true: False
true & false: False
true & true: True

```

```

Boolean logical inclusive OR (|)
false | false: False
false | true: True
true | false: True
true | true: True

```

```

Boolean logical exclusive OR (^)
false ^ false: False
false ^ true: True
true ^ false: True
true ^ true: False

```

```

Logical negation (!)
!false: True
!true: False

```

**Ил. 6.18.** Логические операторы (окончание)

На ил. 6.19 представлены приоритеты и порядок применения операторов, рассмотренных нами до настоящего момента. Операторы перечисляются в порядке убывания приоритетов.

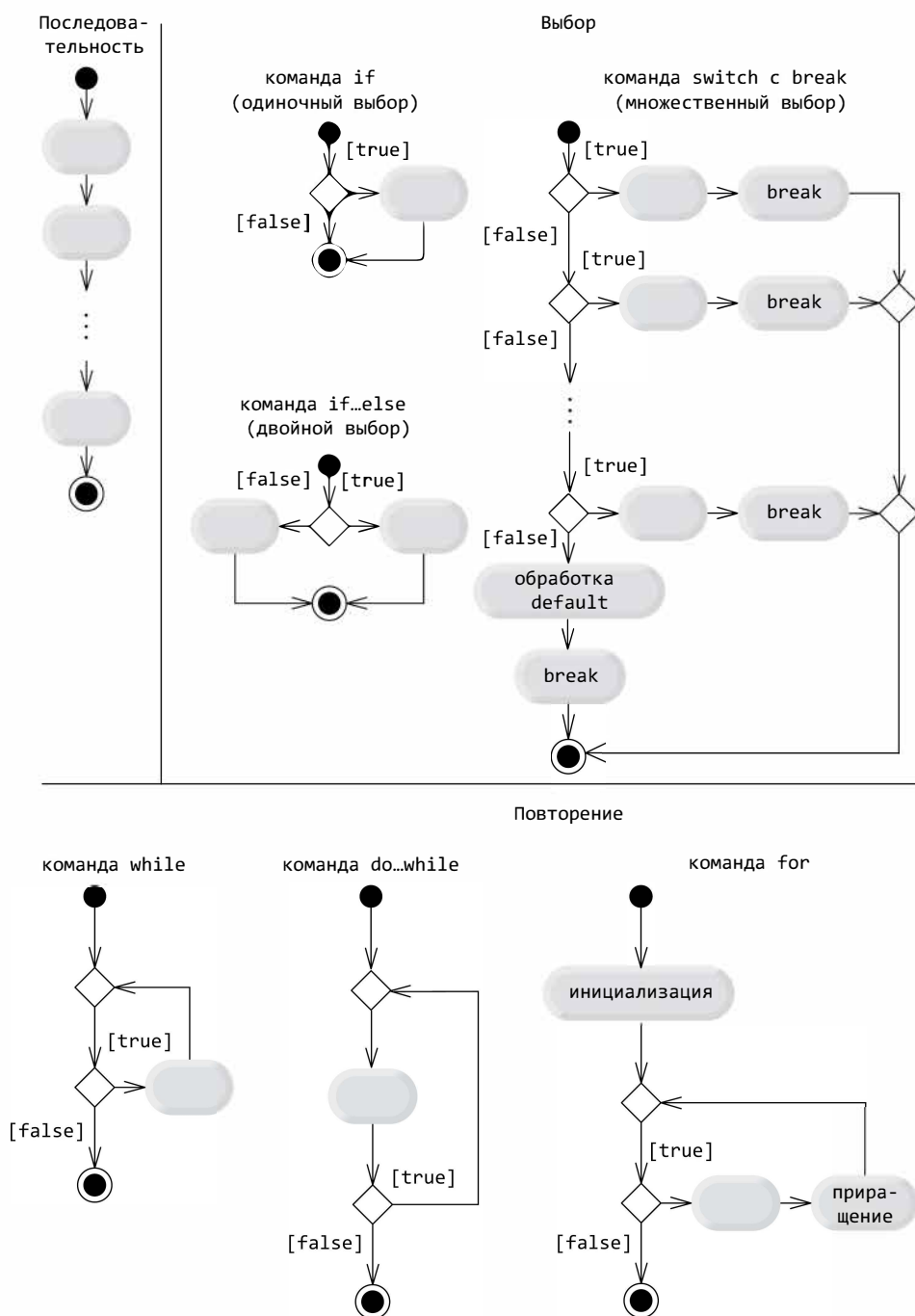
Операторы	Порядок применения	Тип
. new ++(постфиксный) --(постфиксный)	слева направо	максимальный приоритет
++ -- + - ! (тип)	справа налево	унарный префикс
* / %	слева направо	умножение/деление
+ -	слева направо	сложение/вычитание
< <= > >=	слева направо	сравнение
== !=	слева направо	проверка равенства
&	слева направо	булевский оператор И
^	слева направо	булевский исключающий оператор ИЛИ
	слева направо	булевский оператор ИЛИ
&&	слева направо	условный оператор И
	слева направо	условный оператор ИЛИ
? :	справа налево	условный оператор
= += -= *= /= %=	справа налево	присваивание

**Ил. 6.19.** Приоритет и порядок применения операторов, рассмотренных до настоящего момента

## 6.9. Основы структурного программирования

Архитектор проектирует здание, руководствуясь коллективным опытом своей профессии; точно так же должен действовать программист при проектировании приложений. Конечно, наша профессия моложе архитектуры, а наш коллективный опыт существенно меньше. За прошедшие годы выяснилось, что приложения, созданные с применением методологии структурного программирования, более понятны и просты в тестировании, отладке и модификации; более того, их правильность даже можно проверить в математическом смысле.

На ил. 6.20 диаграммы активности UML используются для представления управляющих команд C#. Исходное и конечное состояние соответствуют одной входной и одной выходной точке каждой управляющей команды. Произвольное соединение отдельных блоков на диаграмме активности может привести к нарушению структурированности приложений. По этой причине при создании структурированных приложений может использоваться ограниченное подмножество команд, объединяемых всего двумя простыми способами.



Ил. 6.20. Команды последовательности, выбора и повторения в C#

Для простоты используются только управляющие команды с одним входом и одним выходом. Объединение управляющих команд в последовательность для построения структурированных приложений выполняется тривиально. Конечное состояние одной управляющей команды соединяется с исходным состоянием другой, то есть управляющие команды последовательно размещаются друг за другом. Правила построения структурированных приложений также допускают вложение управляющих команд.

На ил. 6.21 представлены правила построения структурированных приложений. Предполагается, что построение начинается с простейшей диаграммы активности (ил. 6.22), содержащей только исходное состояние, состояние действия и конечное состояние, связанные стрелками переходов.

#### Правила построения структурированных приложений

1. Построение начинается с простейшей диаграммы активности (см. илл. 6.22).
2. Любое действие может быть заменено двумя последовательными состояниями действий.
3. Любое состояние действия может быть заменено любой управляющей командой (последовательностью состояний действий, if, if...else, switch, while, do...while, for или foreach — см. главу 8).
4. Правила 2 и 3 могут применяться столько раз, сколько потребуется, в любом порядке.

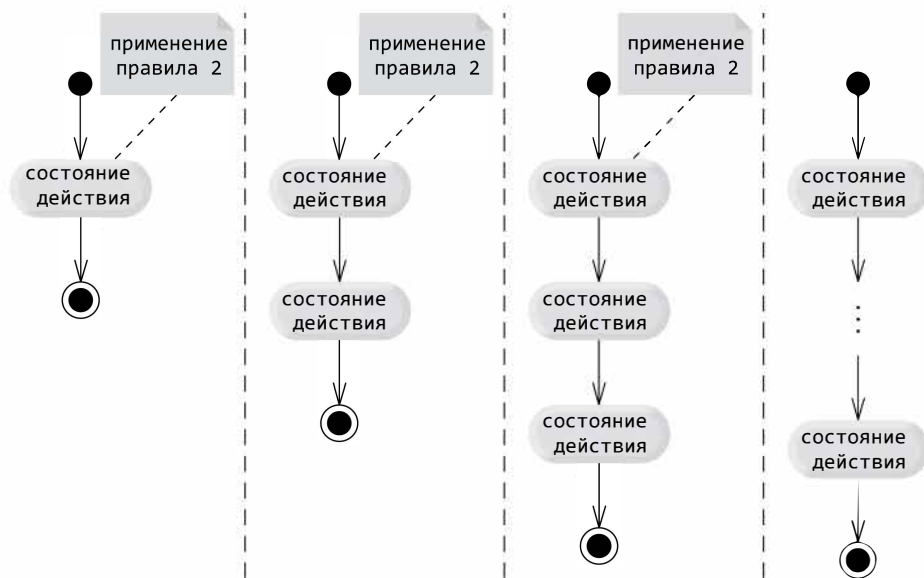
**Ил. 6.21.** Правила построения структурированных приложений



**Ил. 6.22.** Простейшая диаграмма активности

Соблюдение правил на ил. 6.21 всегда приводит к созданию диаграммы активности с четкой, правильной структурой. Например, многократное применение правила 2 к простейшей диаграмме активности приводит к созданию диаграммы активности, содержащей линейную последовательность состояний действий (ил. 6.23). [*Примечание:* вертикальные пунктирные линии на ил. 6.23 не являются частью UML — мы используем их для разделения четырех диаграмм, демонстрирующих применение правила 2 к рис. 6.21.] Правило 3 называется правилом вложения. Многократное применение правила 3 к простейшей диаграмме активности приводит к созданию диаграммы, содержащей несколько вложенных управляющих команд. Например, на ил. 6.24 состояние действия простейшей диаграммы активности заменяется командой двойного выбора (if...else). Затем к состояниям действий команды двойного

выбора снова применяется правило 3, в результате чего каждое состояние заменяется командой двойного выбора. Пунктирные обозначения состояний действий вокруг каждой команды двойного выбора представляют замененное состояние действия.



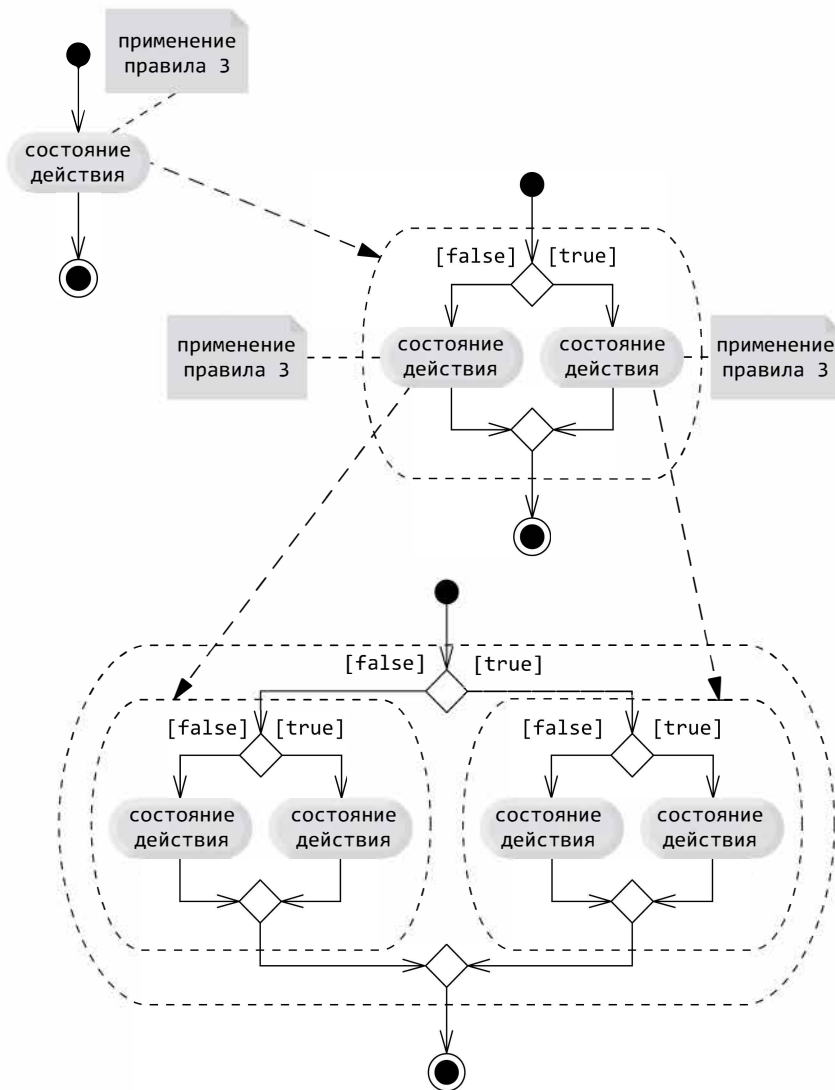
**Ил. 6.23.** Многократное применение правила 2 к простейшей диаграмме активности (см. ил. 6.22)

Правило 4 создает более крупные, более сложные команды с большим уровнем вложенности. Элегантность структурной методологии заключается в том, что мы используем всего восемь простых управляющих команд с одним входом и одним выходом (считая команду `foreach`, представленную в разделе 8.6) и объединяем их всего двумя простыми способами.

При соблюдении правил на ил. 6.21 невозможно создать «неструктурированную» диаграмму активности (вроде изображенной на ил. 6.25). Если вы не уверены в том, является ли конкретная диаграмма структурированной, применяйте правила с ил. 6.21 в обратном направлении, чтобы свести диаграмму к простейшей диаграмме активности. Если у вас это получится, то исходная диаграмма является структурированной.

Структурное программирование выдвигает на первый план простоту. Исследования показали, что для реализации любого алгоритма достаточно всего трех управляющих конструкций:

- ☐ последовательность;
- ☐ выбор;
- ☐ повторение.

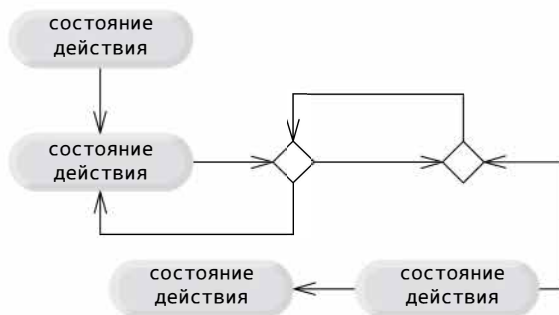


**Ил. 6.24.** Многократное применение правила 3  
к простейшей диаграмме активности

Последовательность реализуется тривиально — просто перечислите команды в порядке их выполнения. Выбор реализуется одним из трех способов:

- ☐ команда `if` (одиначный выбор);
- ☐ команда `if...else` (двойной выбор);
- ☐ команда `switch` (множественный выбор).

На самом деле легко доказать, что простой команды `if` достаточно для реализации любой формы выбора — все, что делается командами `if...else` и `switch`, может быть сделано объединением команд `if` (хотя, возможно, не так четко и эффективно).



**Ил. 6.25.** «Неструктурированная» диаграмма активности

Повторение реализуется четырьмя способами:

- ☐ команда `while`;
- ☐ команда `do...while`;
- ☐ команда `for`;
- ☐ команда `foreach`.

Все эти формы повторения могут быть реализованы командой `while`. Другими словами, все, что делается командами `do...while`, `for` и `foreach`, может быть сделано командой `while` (хотя, возможно, не так удобно).

Объединяя эти результаты, мы видим, что любая форма управления в приложениях C# может быть выражена в виде совокупности:

- ☐ последовательной структуры;
- ☐ команды `if` (выбор);
- ☐ команды `while` (повторение);

и что эти конструкции могут объединяться только двумя способами — сцеплением и вложением. Действительно, структурное программирование является воплощением простоты.

## 6.10. Итоги

В главе 5 рассматривались управляющие команды `if`, `if...else` и `while`. В этой главе были рассмотрены управляющие команды `for`, `do...while` и `switch` (команда `foreach` будет рассмотрена в главе 8). Мы выяснили, что любой алгоритм может



быть представлен в виде сочетания последовательности (то есть перечисления команд в порядке их выполнения), трех команд выбора (`if`, `if...else` и `switch`) и четырех команд повторения (`while`, `do...while`, `for` и `foreach`). При этом команды `for` и `do...while` представляют собой более удобную запись для выражения некоторых типов повторения. Аналогичным образом было показано, что команда `switch` представляет собой более удобную запись для множественного выбора (вместо набора вложенных команд `if...else`). Вы узнали, как использовать команды `break` и `continue` для изменения потока операций в командах повторения. Также в этой главе были представлены логические операторы, позволяющие использовать более сложные условия в управляющих командах. В главе 7 мы перейдем к подробному рассмотрению методов.

# 7 Методы

## 7.1. Введение

Большинство приложений, решающих реальные задачи, гораздо объемнее приложений, представленных в начальных главах этой книги. Практический опыт показывает, что большие приложения лучше всего строить из простых, небольших частей (по принципу «разделяй и властвуй»). Методы были кратко представлены в главе 4, а в этой главе мы рассмотрим их более подробно. Основное внимание будет уделено объявлению и использованию методов для проектирования, реализации и сопровождения больших приложений.

В частности, вы узнаете, что некоторые методы (*статические* методы) могут вызываться без указания объекта, и научитесь объявлять методы с несколькими параметрами. Также вы узнаете, как С# следит за тем, какой метод выполняется в настоящий момент, как аргументы ссылочных и значимых типов передаются методам, как локальные переменные методов хранятся в памяти и как метод узнает, по какому адресу следует вернуть управление после его завершения.

Мы рассмотрим генерирование случайных чисел и напишем приложение для игры в кости, в котором будет использоваться большинство приемов, рассмотренных до настоящего момента. Кроме того, вы научитесь объявлять константы и писать рекурсивные методы (то есть методы, вызывающие сами себя).

Многие классы, которые будут использоваться или создаваться в ваших программах, будут содержать несколько разноименных версий одного метода. Этот прием, называемый «*перегрузкой* методов», используется для реализации методов, выполняющих сходные задачи с разными типами и/или количеством аргументов.

## 7.2. Упаковка кода в С#

Стандартные способы упаковки кода — свойства, методы, классы и пространства имен. При создании приложения С# разработчик объединяет новые свойства, методы и классы, написанные с использованием уже существующих свойств, методов и классов из библиотеки .NET Framework Class Library и других библиотек классов. Взаимосвязанные классы часто объединяются в пространства имен

и компилируются в библиотеки классов для повторного использования в других приложениях. О том, как создавать собственные пространства имен и библиотеки классов, рассказано в главе 15. Framework Class Library предоставляет много готовых классов с методами для выполнения стандартных математических вычислений, манипуляций со строками, операций ввода-вывода, операций с базами данных, сетевых операций, обработки файлов, проверки ошибок и т. д.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 7.1

Не пытайтесь «изобретать велосипед». Там, где это возможно, используйте классы и методы Framework Class Library ([msdn.microsoft.com/en-us/library/ms229335.aspx](http://msdn.microsoft.com/en-us/library/ms229335.aspx)). Они ускоряют разработку, снижают вероятность ошибок программирования и повышают быстродействие приложения.

### Модульная организация программ

Методы (называемые *функциями* или *процедурами* в других языках программирования) позволяют выделить операции, выполняемые приложением, в автономные единицы. Мы объявляли методы в каждом написанном нами приложении. Команды в теле метода пишутся только один раз, могут использоваться из нескольких мест и скрываются от других методов.

Существует несколько причин для модульной организации программ с применением методов. Первая причина заключается в том, что принцип «разделяй и властвуй» упрощает разработку приложения, которое строится из небольших, простых фрагментов. Другая причина связана с возможностью повторного использования кода — существующие методы могут использоваться как структурные элементы при создании новых приложений. Нередко разработка приложения состоит в основном из повторного использования существующих методов (вместо написания собственного специализированного кода). Например, в предыдущих приложениях нам не пришлось определять механизм чтения данных с клавиатуры — Framework Class Library предоставляет эту функциональность в классе `Console`. Третья причина связана с нежелательностью дублирования кода. Разбиение приложения на осмысленные методы упрощает отладку и сопровождение приложения.



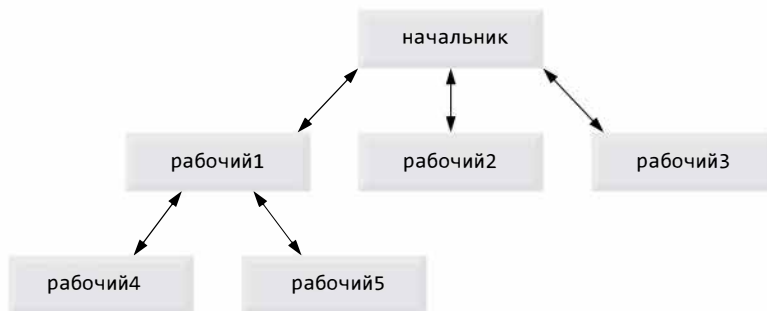
#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 7.2

Чтобы ваш код мог использоваться повторно, каждый метод должен ограничиваться выполнением одной четко определенной операции, которая должна выражаться именем метода. Такие методы упрощают написание, отладку, сопровождение и модификацию приложений.

### Вызов методов

Как вы знаете, управление передается методу в точке его вызова, а после завершения своей работы метод возвращает результат или просто возвращает управление вызывающей стороне. Код, в котором вызывается метод, иногда называется клиентским кодом — иначе говоря, любая команда, вызывающая метод объекта за его пределами, является *клиентом* метода. Структуру вызова и возвращения управления можно

сравнить с иерархией управления (ил. 7.1). Начальник (вызывающая сторона) приказывает рабочему (вызываемому методу) выполнить некую операцию и сообщить о результатах после ее завершения. Метод-начальник не знает, как рабочий будет выполнять свою операцию. Возможно, он будет обращаться за помощью к другим рабочим (то есть вызывать другие методы) — начальник об этом знать не будет. Соккрытие подробностей реализации способствует качественному проектированию. На ил. 7.1 изображена иерархическая схема взаимодействия метода-начальника с несколькими методами-рабочими. Метод-начальник распределяет обязанности между несколькими рабочими. Обратите внимание: *рабочий1* одновременно является «начальником» для методов *рабочий4* и *рабочий5*.



Ил. 7.1. Иерархия отношений «начальник–рабочий»

### 7.3. Статические методы, статические переменные и класс Math

Хотя большинство методов выполняется для конкретных объектов, это не всегда так. Иногда метод выполняет операцию, которая не зависит от внутреннего состояния какого-либо объекта. Такой метод работает в масштабе класса, в котором он объявлен, и называется статическим методом. Классы нередко содержат группы статических методов для выполнения стандартных операций. Например, вспомните, что мы использовали статический метод `Pow` класса `Math` для возведения значения в степень (см. ил. 6.6). Чтобы объявить метод статическим, поставьте ключевое слово `static` перед возвращаемым типом в объявлении метода. Вызов статического метода состоит из имени класса, в котором он объявлен, оператора «точка» (`.`) и имени метода, как в следующем примере:

```
ИмяКласса.ИмяМетода( аргументы )
```

Для представления концепции статических методов мы воспользуемся различными методами класса `Math` из пространства имен `System`. Этот класс содержит набор методов для выполнения распространенных математических вычислений. Например, для вычисления квадратного корня 900 используется вызов статического метода

```
Math.Sqrt( 900.0 )
```

В этом примере метод возвращает значение 30.0. Метод Sqrt получает аргумент типа double и возвращает результат типа double. Для вывода результата приведенного вызова метода в консольном окне можно воспользоваться командой

```
Console.WriteLine( Math.Sqrt( 900.0 ) );
```

В этой команде значение, возвращаемое Sqrt, становится аргументом метода WriteLine. Объект Math не создавался перед вызовом метода Sqrt. Кроме того, все методы Math являются статическими — соответственно, при вызове каждого из них перед именем метода указывается имя класса и оператор «точка» (.). По аналогии метод WriteLine класса Console является статическим методом класса Console, поэтому при его вызове перед именем ставится имя класса Console и оператор «точка».

Аргументами методов могут быть константы, переменные или выражения. Если  $c = 13.0$ ,  $d = 3.0$  и  $f = 4.0$ , то команда

```
Console.WriteLine( Math.Sqrt( c + d * f ) );
```

вычисляет и выводит квадратный корень  $13.0 + 3.0 * 4.0 = 25.0$  — то есть 5.0.

На ил. 7.2 приведена сводка методов класса Math. Предполагается, что аргументы  $x$  и  $y$  относятся к типу double.

Метод	Описание	Пример
Abs( $x$ )	Абсолютное значение (модуль) $x$	Abs( 23.7 ) = 23.7 Abs( 0.0 ) = 0.0 Abs( -23.7 ) = 23.7
Ceiling( $x$ )	Округление $x$ до наименьшего целого, не меньшего $x$	Ceiling( 9.2 ) = 10.0 Ceiling( -9.8 ) = -9.0
Cos( $x$ )	Косинус угла $x$ (в радианах)	Cos( 0.0 ) = 1.0
Exp( $x$ )	Экспонента ( $e^x$ )	Exp( 1.0 ) = 2.71828 Exp( 2.0 ) = 7.38906
Floor( $x$ )	Округление $x$ до наибольшего целого, не большего $x$	Floor( 9.2 ) = 9.0 Floor( -9.8 ) = -10.0
Log( $x$ )	Натуральный логарифм $x$ (по основанию $e$ )	Log( Math.E ) = 1.0 Log( Math.E * Math.E ) = 2.0
Max( $x$ , $y$ )	Большее из значений $x$ и $y$	Max( 2.3, 12.7 ) = 12.7 Max( -2.3, -12.7 ) = -2.3
Min( $x$ , $y$ )	Меньшее из значений $x$ и $y$	Min( 2.3, 12.7 ) = 2.3 Min( -2.3, -12.7 ) = -12.7
Pow( $x$ , $y$ )	$x$ в степени $y$ (например, $x^y$ )	Pow( 2.0, 7.0 ) = 128.0 Pow( 9.0, 0.5 ) = 3.0
Sin( $x$ )	Синус угла $x$ (в радианах)	Sin( 0.0 ) = 0.0
Sqrt( $x$ )	Квадратный корень $x$	Sqrt( 900.0 ) = 30.0
Tan( $x$ )	Тангенс угла $x$ (в радианах)	Tan( 0.0 ) = 0.0

**Ил. 7.2.** Методы класса Math

## Константы PI и E

Класс `Math` также объявляет две константы `static double`, представляющие распространенные математические величины: `Math.PI` и `Math.E`. Константа `Math.PI` (3.14159265358979323846) определяется как отношение длины окружности к длине ее диаметра. Константа `Math.E` (2.7182818284590452354) является основанием натуральных логарифмов (вычисляемых статическим методом `Log` класса `Math`). Эти константы объявляются в классе `Math` с модификаторами `public` и `const`. Объявление их открытыми (`public`) позволяет другим программистам использовать эти переменные в написанных ими классах. Константа объявляется с ключевым словом `const`, означающим, что ее значение не может изменяться после объявления. И `PI`, и `E` объявляются с ключевым словом `const`. Так как эти константы являются статическими, к ним можно обращаться по имени класса `Math` и оператору «точка», как и к методам класса `Math`.

Вспомните, о чем говорилось в разделе 4.5: каждый объект (экземпляр) класса содержит собственную копию переменных, представляющих атрибуты класса; эти переменные называются *переменными экземпляров*. Также существуют переменные, которые не создаются отдельно для каждого экземпляра класса. Статические переменные относятся именно к этой категории. Все объекты класса, содержащего статические переменные, совместно используют одну копию статических переменных. Статические переменные и переменные экземпляров образуют *поля* класса.



### ТИПИЧНАЯ ОШИБКА 7.1

Каждая константа, объявляемая в классе, но не внутри метода класса, считается статической, поэтому явное объявление таких констант с ключевым словом `static` считается синтаксической ошибкой.

## Почему метод `Main` объявляется статическим?

Почему метод `Main` должен объявляться статическим? При запуске приложения, когда объекты классов еще не созданы, метод `Main` должен быть вызван для запуска программы. Метод `Main` иногда называется *точкой входа* приложения. Объявление метода `Main` статическим позволяет исполнительной среде вызвать `Main` без создания экземпляра класса. Метод `Main` часто объявляется с заголовком:

```
public static void Main( string args[] )
```

При выполнении приложения из командной строки указывается имя приложения:

```
ИмяПриложения аргумент1 аргумент2 ...
```

В этой команде *аргумент1* и *аргумент2* — аргументы командной строки приложения, то есть строки (разделенные пробелами), которые передаются исполнительной средой методу `Main` вашего приложения. В аргументах могут передаваться значения, необходимые для запуска приложения (например, имена файлов). Как вы узнаете в главе 8, приложение может обращаться к аргументам командной строки (через параметр `args`) и использовать их для настройки приложения.

### Подробнее о методе Main

Если приложение не получает аргументы командной строки, параметр `string[] args` можно опустить, как и ключевое слово `public`. Также метод `Main` может объявляться с возвращаемым типом `int` (вместо `void`), чтобы он мог вернуть код ошибки. Метод `Main`, объявленный с одним из этих заголовков, может использоваться как точка входа приложения — но в каждом классе может быть объявлен только один метод `Main`.

В большинстве предшествующих примеров использовался только один класс, содержащий только `Main`; в других примерах определялся второй класс, который использовался `Main` для создания объектов и работы с ними. На самом деле любой класс может содержать метод `Main`, и каждый из примеров с двумя классами можно было реализовать в одном классе. Например, в приложении на ил. 6.9 и 6.10 метод `Main` (строки 6–16 на ил. 6.10) можно было бы переместить в класс `GradeBook` (см. ил. 6.9). Результат работы приложения не отличался бы от результата версии с двумя классами. Метод `Main` можно включить в каждый объявленный вами класс, и некоторые программисты используют это обстоятельство для построения маленьких тестовых приложений для каждого объявляемого класса. Однако если в классах вашего проекта объявляется более одного метода `Main`, необходимо сообщить IDE, какой из них должен использоваться в качестве точки входа приложения. Для этого выполните команду **PROJECT ▸ [ИмяПроекта] Properties...** (где `[ProjectName]` — имя вашего проекта) и выберите в списке **Startup Object** класс, содержащий метод `Main`, который должен стать точкой входа.

## 7.4. Объявление методов с несколькими параметрами

А теперь посмотрим, как написать метод с несколькими параметрами. Приложение на ил. 7.3 использует пользовательский метод с именем `Maximum` для определения и возвращения большего из *трех* значений типа `double`. При запуске приложения выполняется метод `Main` (строки 8–22). В строке 18 вызывается метод `Maximum` (объявленный в строках 25–38), который определяет и возвращает большее из трех значений. В конце этого раздела мы поговорим об использовании оператора `+` в строке 21.

```
1 // Ил. 7.3: MaximumFinder.cs
2 // Пользовательский метод Maximum
3 using System;
4
5 public class MaximumFinder
6 {
7     // Получение трех значений типа double и определение максимума
8     public static void Main( string[] args )
9     {
10         // Запрос и ввод трех значений типа double
```

**Ил. 7.3.** Пользовательский метод `Maximum` (продолжение ↗)

```

11 Console.WriteLine( "Enter three floating-point values,\n" +
12     " pressing 'Enter' after each one: " );
13 double number1 = Convert.ToDouble( Console.ReadLine() );
14 double number2 = Convert.ToDouble( Console.ReadLine() );
15 double number3 = Convert.ToDouble( Console.ReadLine() );
16
17 // Определение максимального значения
18 double result = Maximum( number1, number2, number3 );
19
20 // Вывод максимального значения
21 Console.WriteLine( "Maximum is: " + result );
22 } // end Main
23
24 // Метод возвращает максимум для трех параметров типа double
25 public static double Maximum( double x, double y, double z )
26 {
27     double maximumValue = x; // Сначала предполагаем, что x - максимум
28
29     // Сравниваем y с maximumValue
30     if ( y > maximumValue )
31         maximumValue = y;
32
33     // Сравниваем z с maximumValue
34     if ( z > maximumValue )
35         maximumValue = z;
36
37     return maximumValue;
38 } // Конец метода Maximum
39 } // Конец класса MaximumFinder

```

Enter three floating-point values,  
pressing 'Enter' after each one:  
2.22  
3.33  
1.11  
Maximum is: 3.33

**Ил. 7.3.** Пользовательский метод Maximum (окончание)

### Ключевые слова `public` и `static`

Объявление метода `Maximum` начинается с ключевого слова `public`, которое указывает, что метод является открытым, то есть может вызываться методами других классов. Ключевое слово `static` позволяет методу `Main` (еще один статический метод) вызывать `Maximum` так, как показано в строке 18, — без уточнения имени метода именем класса `MaximumFinder`; статические методы одного класса могут вызывать друг друга напрямую. Любой другой класс, использующий `Maximum`, должен полностью уточнить имя метода, добавив к нему имя класса.

### Метод `Maximum`

Объявление метода `Maximum` находится в строках 25–38. Строка 25 указывает, что метод возвращает значение `double`, называется `Maximum` и получает три параметра типа `double` (`x`, `y` и `z`) для выполнения своей операции. Если метод получает более одного параметра, то параметры передаются в виде списка, разделенного запятыми.



При вызове `Maximum` в строке 18 параметр `x` инициализируется значением аргумента `number1`, параметр `y` инициализируется значением аргумента `number2`, а параметр `z` — значением аргумента `number3`. При вызове метода должен быть указан один аргумент для каждого обязательного параметра (также иногда называемого *формальным параметром*) в объявлении метода. Кроме того, каждый аргумент должен быть совместим с типом соответствующего параметра. Например, параметр типа `double` может получать значения `7.35` (`double`), `22` (`int`) или `-0.03456` (`double`), но не строку `"hello"`. В разделе 7.7 рассматриваются типы аргументов, которые могут передаваться для каждого параметра простого типа при вызове метода.



### ТИПИЧНАЯ ОШИБКА 7.2

Объявление однотипных параметров метода в форме `double x, y` (вместо `double x, double y`) является синтаксической ошибкой: тип должен быть указан для каждого параметра в списке.

### Реализация метода `Maximum` на базе `Math.Max`

Как видно из листинга на ил. 7.2, класс `Math` содержит метод `Max`, определяющий большее из двух значений. Все тело метода для определения максимума может быть реализовано вложенными вызовами `Math.Max` следующим образом:

```
return Math.Max( x, Math.Max( y, z ) );
```

При левом вызове `Math.Max` передаются аргументы `x` и `Math.Max( y, z )`. Прежде чем вызывать метод, исполнительная подсистема .NET вычисляет все аргументы для определения их значений. Если аргумент содержит вызов метода, то этот метод будет вызван для определения возвращаемого значения. Таким образом, в приведенной команде сначала выполняется вызов `Math.Max( y, z )`, определяющий максимум для `y` и `z`. Результат передается во втором аргументе левого вызова `Math.Max`. Такое использование `Math.Max` является хорошим примером повторного использования кода — максимум для трех значений вычисляется при помощи метода `Math.Max`, определяющего максимум для двух значений. Обратите внимание, насколько компактно выглядит код по сравнению со строками 27–37 на ил. 7.3.

### Построение строк посредством конкатенации

C# позволяет создавать объекты `string` посредством «сцепления» меньших строк с использованием оператора `+` (или комбинированного оператора присваивания `+=`). Эта операция называется *конкатенацией*. Если оба операнда оператора `+` представляют собой объекты `string`, оператор `+` создает новый объект `string`, в котором копия символов правого операнда помещается в конец копии символов левого операнда. Например, выражение `"hello "+"there"` создает строку `"hellothere"` без изменения исходных строк.

В строке 21 на ил. 7.3 выражение `"Maximum is: " + result` использует оператор `+` с операндами типов `string` и `double`. У любого значения простого типа в C# имеется строковое представление. Если один из операндов оператора `+` представляет собой строку, другой оператор неявно преобразуется в строку, после чего выполняется конкатенация этих строк. В строке 21 значение `double` неявно преобразуется в `string`.

и помещается в конец строки "Maximum is:". Завершающие нули в значении `double` отбрасываются при преобразовании числа в `string`. Таким образом, числовое значение 9.3500 будет представлено в итоговой строке в формате 9.35.

### Строковое представление

Значения простых типов, используемые при конкатенации строк, преобразуются в строки. Если `bool` объединяется посредством конкатенации с `string`, то `bool` преобразуется в строку "True" или "False" (с прописной буквы). Каждый объект содержит метод `ToString`, который возвращает строковое представление объекта. При конкатенации объекта с `string` неявно вызывается метод `ToString` для получения строкового представления объекта. Если объект равен `null`, выводится пустая строка.

### Форматирование строк при выводе

Строку 21 на ил. 7.3 также можно записать с использованием форматных элементов строк в виде

```
Console.WriteLine( "Maximum is: {0}", result );
```

Как и в случае с конкатенацией, при использовании форматного элемента для преобразования объекта в строку неявно вызывается метод `ToString` этого объекта для получения строкового представления объекта. Метод `ToString` более подробно рассматривается в главе 8.

### Разбиение длинных строковых литералов

Когда в исходном коде приложения вводится длинный строковый литерал, для удобства чтения его можно разбить на несколько меньших строк, которые размещаются в нескольких строках для удобства чтения. Строки-компоненты собираются воедино посредством конкатенации или строкового форматирования. Работа со строками более подробно рассматривается в главе 16.



#### ТИПИЧНАЯ ОШИБКА 7.3

Простая разбивка строкового литерала на несколько строк исходного кода приложения является синтаксической ошибкой. Если строковый литерал не помещается в одной строке, разбейте его на несколько меньших строк и воспользуйтесь конкатенацией для формирования полной строки. В C# также поддерживаются «буквальные» (verbatim) строковые литералы, начинающиеся с символа `@`. Такие литералы могут разбиваться на несколько строк, и все символы в литерале, включая пропуски, обрабатываются точно в таком виде, в каком они задаются в литерале.



#### ТИПИЧНАЯ ОШИБКА 7.4

Не смешивайте оператор `+`, используемый для конкатенации строк, с оператором `+`, используемым для сложения — это может приводить к странным результатам. Оператор `+` применяется слева направо. Например, если целочисленная переменная `y` равна 5, то при вычислении "`y + 2 =`" `y + 2` получится строка "`y + 2 = 52`", а не "`y + 2 = 7`", потому что сначала конкатенация объединяет значение `y` со строкой "`y + 2 =`", а затем значение 2 с новой большой строкой "`y + 2 = 5`". Выражение "`y + 2 =`" + ( `y + 2` ) дает желаемый результат "`y + 2 = 7`".

## 7.5. Объявление и использование методов

Мы рассмотрели три способа вызова методов:

1. Простой вызов по имени для вызова метода того же класса — например, `Maximum(number1, number2, number3)` в строке 18 на ил. 7.3.
2. Имя переменной, содержащей ссылку на объект, за которым следует оператор «точка» и имя метода для вызова нестатического метода для указанного объекта, — например, вызов метода `myGradeBook.DisplayMessage()` в строке 13 на ил. 6.10, вызывающий метод класса `GradeBook` из метода `Main` класса `GradeBookTest`.
3. Имя класса, за которым следует оператор «точка» и имя метода для вызова статического метода класса, — например, `Convert.ToDouble(Console.ReadLine())` в строках 13–15 на ил. 7.3 или `Math.Sqrt(900.0)` в разделе 7.3.

Также существуют три способа возврата управления команде, вызвавшей метод. Если метод имеет возвращаемый тип `void` (то есть не возвращает результат), то управление возвращается при достижении правой фигурной скобки, завершающей метод, или при выполнении команды

```
return;
```

Если метод возвращает результат, то команда

```
return выражение;
```

вычисляет выражение и возвращает результат (и управление) вызывающей стороне.



### ТИПИЧНАЯ ОШИБКА 7.5

Объявление метода вне тела объявления класса или в теле другого метода является синтаксической ошибкой.



### ТИПИЧНАЯ ОШИБКА 7.6

Переобъявление параметра метода как локальной переменной в теле метода является ошибкой компиляции.



### ТИПИЧНАЯ ОШИБКА 7.7

Если вы забудете вернуть значение из метода, который должен его возвращать, произойдет ошибка компиляции. Если в заголовке метода указан возвращаемый тип, отличный от `void`, метод должен использовать команду `return` для возвращения значения, совместимого с возвращаемым типом метода. Попытка вернуть значение из метода, объявленного с возвращаемым типом `void`, является ошибкой компиляции.

Статический метод может *напрямую* вызывать только другие статические методы того же класса (то есть с указанием только имени метода) и работать только со

статическими переменными того же класса (с указанием только имени переменной). Чтобы обратиться к нестатическим членам класса, статический метод должен использовать ссылку на объект класса. Вспомните, что статические методы относятся к классу в целом, тогда как нестатические методы ассоциируются с конкретным экземпляром (объектом) класса и могут работать с переменными экземпляра этого объекта. Одновременно могут существовать много объектов класса, каждый из которых содержит собственные копии переменных экземпляров. Допустим, статический метод попытается напрямую вызвать нестатический метод. Откуда метод узнает, с переменными какого экземпляра он должен работать? И что произойдет, если на момент вызова нестатического метода еще не существует ни одного объекта класса? Из-за этого C# не позволяет статическим методам напрямую обращаться к нестатическим членам того же класса.

## 7.6. Стек вызовов и кадр стека

Чтобы понять, как в C# происходят вызовы методов, необходимо сначала рассмотреть структуру данных, называемую *стеком* (структуры данных более подробно рассматриваются в главах 18–21). Стек можно представить себе в виде стопки тарелок. Новые тарелки обычно кладутся на стопку сверху (занесение в стек). Если вам потребуется взять новую тарелку, вы тоже берете ее сверху (извлечение из стека). Стеки относятся к структурам данных LIFO (Last In, First Out, то есть «последним пришел, первым вышел») — последний элемент, занесенный в стек, первым извлекается из него.

Когда приложение содержит вызов метода, вызванный метод должен знать, как вернуть управление вызывающей стороне. Для этого адрес возврата (адрес вызывающего метода) заносится в программный стек (также называемый стеком вызовов). Если в программе происходит серия вызовов методов, адреса возврата последовательно заносятся в стек в порядке LIFO, так что каждый метод возвращает управление своей вызывающей стороне.

В стеке вызовов также хранятся локальные переменные, используемые при каждом вызове метода во время выполнения приложения. Набор данных, хранимых в стеке, называется *кадрам стека*, или записью активизации. При вызове метода в стеке выделяется память для его кадра. При возврате управления кадр извлекается из стека, а локальные переменные становятся недоступными для приложения. Если ссылка на некоторый объект хранится только в локальной переменной, то при извлечении из стека кадра, содержащего эту переменную, объект становится недоступным для приложения и через некоторое время будет удален из памяти в ходе «уборки мусора». Механизм уборки мусора рассматривается в разделе 10.8.

Конечно, объем памяти на компьютере неограничен, так что в стеке может храниться ограниченное количество кадров. Если количество вызовов методов превышает количество кадров, которые могут храниться в стеке, происходит фатальная ошибка, называемая *переполнением стека*.

## 7.7. Преобразования типа аргументов

Еще один важный аспект вызовов методов связан с неявным преобразованием значений аргументов к типам, которые метод рассчитывает получить в соответствующем параметре (если это возможно). Например, приложение может вызвать метод `Sqrt` класса `Math` с целочисленным аргументом несмотря на то, что метод рассчитывает получить аргумент `double`. Команда

```
Console.WriteLine( Math.Sqrt( 4 ) );
```

правильно обработает вызов `Math.Sqrt(4)` и выведет значение `2.0`. Список параметров метода `Sqrt` заставляет C# преобразовать значение `4` типа `int` в значение `4.0` типа `double`, прежде чем передавать его `Sqrt`. Такие преобразования могут привести к ошибкам компиляции при нарушении правил *приведения типа* аргументов. Правила приведения типов определяют допустимые преобразования (то есть преобразования, не приводящие к потере данных). В приведенном ранее примере с `Sqrt` значение `int` преобразуется в `double` без изменения. Однако попытка преобразования `double` в `int` приведет к усечению дробной части значения `double` — таким образом, часть значения будет потеряна. Кроме того, переменные `double` могут содержать гораздо большие (и гораздо меньшие) значения, чем переменные `int`, так что присваивание значения `double` переменной `int` может привести к потере информации, если присваиваемое значение не помещается в переменной.

Преобразование больших целочисленных типов в меньшие целочисленные типы (например, `long` в `int`) тоже может привести к изменению значения.

### Правила приведения типов

Правила приведения типов применяются к выражениям, содержащим значения двух и более простых типов, а также к значениям простых типов, передаваемых в аргументах методов. Каждое значение приводится к соответствующему типу в выражении (а точнее, в выражении используется временная копия каждого значения — типы исходных значений остаются неизменными). На ил. 7.4 перечислены простые типы (в алфавитном порядке) и указаны типы, к которым они могут приводиться. Значения всех простых типов также могут неявно преобразовываться к типу `object`. Примеры неявных преобразований будут представлены в главе 19.

Тип	Преобразования
<code>bool</code>	Неявные преобразования в другие простые типы невозможны
<code>byte</code>	<code>ushort</code> , <code>short</code> , <code>uint</code> , <code>int</code> , <code>ulong</code> , <code>long</code> , <code>decimal</code> , <code>float</code> или <code>double</code>
<code>char</code>	<code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>decimal</code> , <code>float</code> или <code>double</code>
<code>decimal</code>	Неявные преобразования в другие простые типы невозможны
<code>double</code>	Неявные преобразования в другие простые типы невозможны
<code>float</code>	<code>double</code>

**Ил. 7.4.** Неявные преобразования между простыми типами (продолжение ↗)

Тип	Преобразования
int	long, decimal, float или double
long	decimal, float или double
sbyte	short, int, long, decimal, float или double
short	int, long, decimal, float или double
uint	ulong, long, decimal, float или double
ulong	decimal, float или double
ushort	uint, int, ulong, long, decimal, float или double

**Ил. 7.4.** Неявные преобразования между простыми типами (окончание)

### Необходимые явные преобразования

По умолчанию C# не разрешает выполнять неявные преобразования значений простых типов, если целевой тип не может представлять значение исходного типа (например, значение 2 000 000 типа `int` не может быть представлено типом `short`, а любое вещественное число с дробной частью не может быть представлено целочисленным типом — `long`, `int` или `short`). Для предотвращения ошибок компиляции при возможной потере информации, обусловленной неявным преобразованием простых типов, компилятор требует использовать оператор преобразования типа. По сути, вы говорите компилятору: «Я знаю, что преобразование может привести к потере информации, но для моих целей оно подойдет». Допустим, вы написали метод `Square` для возведения в квадрат целого числа; соответственно, методу передается аргумент `int`. Чтобы вызвать `Square` для целой части аргумента `doubleValue` типа `double`, используется вызов вида `Square((int) doubleValue)`. Значение `doubleValue` явно преобразуется в целое число для использования с методом `Square`. Таким образом, если переменная `doubleValue` равна 4.5, то метод получает значение 4 и возвращает 16, а не 20.25 (что, к сожалению, приводит к потере информации).



#### ТИПИЧНАЯ ОШИБКА 7.8

Преобразование значения простого типа к другому простому типу может привести к изменению значения, если приведение запрещено. Например, преобразование значения с плавающей точкой в целочисленное значение может привести к ошибкам округления (потере дробной части).

## 7.8. .NET Framework Class Library

Многие готовые классы группируются в категории, называемые *пространствами имен*. Эти пространства имен образуют библиотеку *.NET Framework Class Library*. Директивы `using` позволяют нам использовать библиотечные классы `Framework Class Library` без указания их полных имен. Например, приложение включает директиву

```
using System;
```

для использования имен классов пространства имен `System` без полного уточнения их имен. Это позволяет разработчику использовать в коде неуточненное имя класса `Console` вместо полного имени `System.Console`. Многочисленные классы в пространствах имен .NET Framework Class Library — одна из сильных сторон C#. В таблице на ил. 7.5 представлены некоторые пространства имен Framework Class Library, составляющие лишь незначительную часть библиотеки.

Пространство имен	Описание
<code>System.Windows.Forms</code>	Классы для создания и управления графическим интерфейсом. (Различные классы этого пространства имен рассматриваются в главах 14 и 15)
<code>System.Windows.Controls</code> <code>System.Windows.Input</code> <code>System.Windows.Media</code> <code>System.Windows.Shapes</code>	Классы Windows Presentation Foundation для графических интерфейсов, 2D- и 3D-графики, мультимедиа и анимации
<code>System.Linq</code>	Классы поддержки LINQ (Language Integrated Query). Технология LINQ и коллекция <code>List</code> представлены в главе 9
<code>System.Data.Entity</code>	Классы для работы с информацией в базах данных, включая поддержку LINQ to Entities (см. главу 22)
<code>System.IO</code>	Классы для ввода и вывода данных в программах (см. главу 17)
<code>System.Web</code>	Классы для создания веб-приложений, используемых по Интернету (см. главу 23)
<code>System.Xml</code>	Классы для создания и работы с данными в формате XML. Поддерживается как чтение, так и запись в файлы XML (см. главу 24)
<code>System.Xml.Linq</code>	Классы поддержки LINQ для документов XML (см. главу 24)
<code>System.Collections</code> <code>System.Collections.Generic</code>	Классы, определяющие структуры данных для работы с коллекциями (см. главу 21)
<code>System.Text</code>	Классы для работы с символами и строками (см. главу 16)

**Ил. 7.5.** Некоторые пространства имен .NET Framework Class Library

.NET Framework Class Library также содержит пространства имен для работы со сложной графикой, построения расширенных графических интерфейсов, печати, расширенной поддержки сети, безопасности, обработки баз данных, мультимедиа и многих других областей — всего более 200 пространств имен.

### Подробные описания методов классов .NET

Подробные описания методов классов .NET приведены в справочной документации .NET Framework Class Library ([msdn.microsoft.com/en-us/library/ms229335.aspx](http://msdn.microsoft.com/en-us/library/ms229335.aspx)). При посещении этого сайта вы получаете алфавитный список всех пространств имен из Framework Class Library. Найдите пространство имен и щелкните на его ссылке, чтобы просмотреть алфавитный перечень всех его классов с краткими описаниями. Щелкните на ссылке класса, чтобы получить его подробное описание. Ссылка **Methods** в левом столбце открывает список методов класса.





### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 7.1

Электронная документация .NET Framework упрощает поиск и предоставляет подробную информацию о каждом классе. В процессе изучения классов обращайтесь к электронной документации за дополнительными сведениями.

## 7.9. Пример: генератор случайных чисел

В этом и следующем разделах мы разработаем структурированное приложение с несколькими методами. В приложении используются многие управляющие команды, представленные ранее в книге, а также некоторые концепции программирования.

В атмосфере казино есть нечто, разжигающее людской азарт — от крупных игроков за столами для игры в кости до прожигателей мелочи у «одноруких бандитов». Это элемент удачи; шанс на то, что небольшая стопка купюр вдруг превратится в денежный фонтан. Элемент удачи можно ввести в приложение при помощи объекта класса `Random` (из пространства имен `System`). Объекты класса `Random` могут генерировать случайные значения типов `byte`, `int` и `double`. В нескольких ближайших примерах мы используем объекты класса `Random` для получения случайных чисел.

### Создание объекта для генерирования случайных чисел

Новый объект-генератор случайных чисел создается следующей командой:

```
Random randomNumbers = new Random();
```

Генератор может использоваться для создания случайных значений типов `byte`, `int` и `double`; мы ограничимся только значениями `int`.

### Генерирование случайных целых чисел

Рассмотрим следующую команду:

```
int randomValue = randomNumbers.Next();
```

Метод `Next` класса `Random` генерирует случайные значения `int` в диапазоне от 0 до +2 147 483 646 включительно. Чтобы генерируемые методом `Next` значения были действительно случайными, вероятности выбора всех значений в диапазоне при очередном вызове `Next` должны быть равны. Значения, возвращаемые `Next`, на самом деле являются *псевдослучайными*, то есть образуют серию значений, вычисляемых по сложным математическим формулам. Текущее время суток (которое, естественно, постоянно изменяется) используется для инициализации генератора, чтобы при каждом запуске приложения выдавались разные серии случайных чисел.

### Масштабирование диапазона генерируемых чисел

Диапазон чисел, генерируемых методом `Next`, часто отличается от диапазона значений, необходимых конкретному приложению C#. Например, приложение, моделирующее бросание монетки, может использовать только два значения, 0 и 1 («орел» и «решка»). Приложению, моделирующему бросок шестигранного кубика,



требуются случайные целые числа в диапазоне 1–6. Для подобных ситуаций класс `Random` предоставляет несколько разных версий метода `Next`. Одна версия получает аргумент `int` и возвращает значение от 0 до значения аргумента (не включая последнее). Например, команда

```
int randomValue = randomNumbers.Next( 6 );
```

возвращает случайные значения из набора 0, 1, 2, 3, 4 и 5. Аргумент 6 определяет количество уникальных значений, генерируемых методом `Next` (в приведенном примере их шесть — 0, 1, 2, 3, 4 и 5).

Эта манипуляция называется *масштабированием* диапазона значений, генерируемых методом `Next` класса `Random`.

### Сдвиг диапазона случайных значений

Предположим, мы хотим моделировать бросок шестигранного кубика, на гранях которого стоят числа от 1 до 6 (а не от 0 до 5). В этом случае масштабирования оказывается недостаточно — также необходимо выполнить *сдвиг* диапазона генерируемых чисел. Для этого следует прибавить к результату вызова `Next` величину сдвига (в данном случае 1):

```
face = 1 + randomNumbers.Next( 6 );
```

Сдвиг (1) определяет первое число в диапазоне случайных целых чисел. Приведенная команда генерирует числа в диапазоне от 1 до 6.

### Объединение масштабирования со сдвигом

Третья версия метода `Next` предоставляет более наглядный способ сочетания масштабирования со сдвигом. Этот метод получает два аргумента типа `int` и возвращает значение в диапазоне от первого аргумента до второго (не включая последнего).

При использовании этого метода команда, эквивалентная предыдущей, записывается следующим образом:

```
face = randomNumbers.Next( 1, 7 );
```

### Моделирование шестигранного кубика

Для демонстрации генерирования случайных чисел мы напишем приложение, которое моделирует серию из 20 бросков шестигранного кубика и выводит значение каждого броска. На ил. 7.6 представлены два примерных результата, которые подтверждают, что наша формула действительно генерирует целые числа в диапазоне от 1 до 6 и в разных сериях генерируются разные числа. Директива `using` (строка 3) позволяет использовать класс `Random` без указания его полного имени. В строке 9 создается объект `randomNumbers` класса `Random` для генерирования случайных значений. Строка 16 выполняется в цикле 20 раз для моделирования серии бросков. Команда `if` (строки 21–22) переходит на новую строку вывода через каждые 5 чисел, чтобы результаты выводились в несколько строк.

```

1 // Ил. 7.6: RandomIntegers.cs
2 // Сдвиг и масштабирование диапазона случайных чисел.
3 using System;
4
5 public class RandomIntegers
6 {
7     public static void Main( string[] args )
8     {
9         Random randomNumbers = new Random(); // Генератор случайных чисел
10        int face; // Переменная для хранения сгенерированного числа
11
12        // Выполнить в цикле 20 раз
13        for ( int counter = 1; counter <= 20; counter++ )
14        {
15            // Выбор случайного целого числа от 1 до 6
16            face = randomNumbers.Next( 1, 7 );
17
18            Console.Write( "{0} ", face ); // Вывод сгенерированного значения
19
20            // Если counter делится на 5, перейти на новую строку вывода
21            if ( counter % 5 == 0 )
22                Console.WriteLine();
23        } // Конец for
24    } // Конец Main
25 } // Конец класса RandomIntegers

```

```

3 3 3 1 1
2 1 2 4 2
2 3 6 2 5
3 4 6 6 1

```

```

6 2 5 1 3
5 2 1 6 5
4 1 6 1 3
3 1 4 3 4

```

**Ил. 7.6.** Сдвиг и масштабирование диапазона случайных целых чисел

### Моделирование длинной серии бросков

Чтобы убедиться в том, что числа, генерируемые методом `Next`, выпадают с примерно равной вероятностью, мы смоделируем 6 000 000 бросков кубика (ил. 7.7). Каждое число от 1 до 6 должно встречаться в результатах приблизительно 1 000 000 раз.

```

1 // Ил. 7.7: RollDie.cs
2 // Моделирование серии из 6 000 000 бросков кубика.
3 using System;
4
5 public class RollDie
6 {
7     public static void Main( string[] args )
8     {
9         Random randomNumbers = new Random(); // Генератор случайных чисел
10
11        int frequency1 = 0; // Количество выпавших 1
12        int frequency2 = 0; // Количество выпавших 2
13        int frequency3 = 0; // Количество выпавших 3
14        int frequency4 = 0; // Количество выпавших 4

```

**Ил. 7.7.** Моделирование серии из 6 000 000 бросков кубика (продолжение ➞)

```

15     int frequency5 = 0; // Количество выпавших 5
16     int frequency6 = 0; // Количество выпавших 6
17
18     int face; // Переменная для хранения сгенерированного числа
19
20     // Вывод сводки результатов 6 000 000 бросков кубика
21     for ( int roll = 1; roll <= 6000000; ++roll )
22     {
23         face = randomNumbers.Next( 1, 7 ); // Число от 1 до 6
24
25         // Определение результата броска и увеличение счетчика
26         switch ( face )
27         {
28             case 1:
29                 ++frequency1; // Увеличение счетчика единиц
30                 break;
31             case 2:
32                 ++frequency2; // Увеличение счетчика двоек
33                 break;
34             case 3:
35                 ++frequency3; // Увеличение счетчика троек
36                 break;
37             case 4:
38                 ++frequency4; // Увеличение счетчика четверок
39                 break;
40             case 5:
41                 ++frequency5; // Увеличение счетчика пятерок
42                 break;
43             case 6:
44                 ++frequency6; // Увеличение счетчика шестерок
45                 break;
46         } // Конец switch
47     } // Конец for
48
49     Console.WriteLine( "Face\tFrequency" ); // Вывод заголовка
50     Console.WriteLine(
51         "1\t{0}\n2\t{1}\n3\t{2}\n4\t{3}\n5\t{4}\n6\t{5}", frequency1,
52         frequency2, frequency3, frequency4, frequency5, frequency6 );
53 } // Конец Main
54 } // Конец класса RollDie

```

```

Face Frequency
1 999147
2 1001249
3 999929
4 1000301
5 1000294
6 999080

```

```

Face Frequency
1 1000538
2 1002700
3 1000294
4 997662
5 999507
6 999299

```

Как видно из двух приведенных результатов, значения, сгенерированные методом `Next`, позволяют реалистично моделировать броски шестигранного кубика. Приложение использует вложенные управляющие команды (`switch` внутри `for`) для подсчета выпадений каждой стороны кубика. Команда `for` (строки 21–47) выполняет 6 000 000 итераций цикла. В каждой итерации строка 23 генерирует случайное значение от 1 до 6. Значение `face` используется как выражение `switch` (строка 26) команды `switch` (строки 26–46). В зависимости от значения `face` при каждой итерации увеличивается одна из шести переменных-счетчиков. (В разделе 8.4 будет представлен элегантный способ замены всей конструкции `switch` в приложении одной командой.) Команда `switch` не содержит секции `default`, потому что в ней присутствует секция для каждого значения, которое может генерироваться выражением в строке 23. Запустите приложение несколько раз и наблюдайте за результатами. Вы увидите, что при каждом запуске приложение выдает разные результаты.

### 7.9.1. Масштабирование и сдвиг случайных чисел

Ранее была приведена команда

```
face = randomNumbers.Next( 1, 7 );
```

моделирующая бросок шестигранного кубика. Эта команда всегда присваивает переменной `face` целое число в диапазоне  $1 \leq \text{face} < 7$ . Ширина диапазона (то есть количество последовательных целых чисел в диапазоне) всегда равна 6, при этом диапазон начинается с 1. В предыдущей команде мы видим, что ширина диапазона определяется разностью между двумя числами, переданными методу `Next` класса `Random`, а диапазон начинается со значения первого аргумента. Результат можно обобщить в форме

```
number = randomNumbers.Next( сдвиг, сдвиг + масштаб );
```

где *сдвиг* определяет первое число в требуемом диапазоне целых чисел, а *масштаб* — количество чисел в диапазоне.

Целые числа также могут выбираться из набора значений вместо диапазона последовательных целых чисел. Для этого проще использовать версию метода `Next`, получающую только один аргумент. Например, для получения случайного значения из серии 2, 5, 8, 11 и 14 можно воспользоваться следующей командой:

```
number = 2 + 3 * randomNumbers.Next( 5 );
```

Вызов `randomNumberGenerator.Next(5)` генерирует значения в диапазоне 0–4. Каждое сгенерированное значение умножается на 3, в результате чего сгенерированное число входит в последовательность 0, 3, 6, 9 и 12.

Затем каждое значение увеличивается на 2, чтобы сдвинуть диапазон и получить значение из последовательности 2, 5, 8, 11 и 14. Результат можно обобщить в следующем виде:

```
number = сдвиг + разность * randomNumbers.Next( масштаб );
```

где *сдвиг* определяет первое число в требуемом диапазоне целых чисел, *разность* — разность между соседними членами последовательности, а *масштаб* — количество чисел в диапазоне.

## 7.9.2. Повторение случайных чисел при тестировании и отладке

Как упоминалось ранее в этом разделе, методы класса `Random` в действительности генерируют псевдослучайные числа по сложным математическим формулам. Многократные вызовы методов `Random` генерируют последовательность чисел, которые кажутся случайными. Формула, по которой генерируются псевдослучайные числа, использует время суток для изменения отправной точки последовательности. Каждый новый объект `Random` инициализируется значением, основанным на показаниях часов компьютера на момент своего создания, что позволяет приложению получать разные серии случайных чисел при каждом запуске приложения.

При отладке приложения иногда бывает нужно, чтобы одна серия псевдослучайных чисел повторялась при каждом запуске. Такое повторение позволит вам убедиться в том, что приложение работает для конкретной последовательности случайных чисел, прежде чем переходить к другим последовательностям. В таких ситуациях объект `Random` создается следующим образом:

```
Random randomNumbers = new Random( seedValue );
```

Аргумент `seedValue` (типа `int`) инициализирует генератор. При использовании одинаковых значений `seedValue` объект `Random` выдает одинаковые последовательности случайных чисел.

## 7.10. Пример: игра «крэпс» и перечисления

Одной из самых популярных азартных игр, в которую играют в казино по всему миру, называется «крэпс». Ее правила очень просты:

Игрок бросает два кубика. У каждого кубика шесть граней, на которых нанесена разметка с 1, 2, 3, 4, 5 и 6 точками. После того как брошенные кубики остановятся, вычисляется сумма очков на двух верхних гранях. Если сумма равна 7 или 11 при первом броске, то игрок выигрывает. Если сумма равна 2, 3 или 12 при первом броске («крэпс»), игрок проигрывает (то есть побеждает казино). Если сумма равна 4, 5, 6, 8, 9 или 10 при первом броске, то эта сумма становится «пойнтом» игрока. Чтобы победить, игрок должен продолжать бросать кубики, пока не выпадет поинт (то есть та же сумма). Если до выпадения поинта выпадет 7, игрок проигрывает.

Приложение на ил. 7.8 моделирует игру «крэпс» с использованием методов для определения игровой логики. Метод `Main` (строки 24–70) вызывает статический метод `RollDice` (строки 73–85) для моделирования броска двух кубиков и вычисления

их суммы. В четырех приведенных результатах игрок выигрывает с первого броска, проигрывает с первого броска, проигрывает при последующем броске и выигрывает при последующем броске соответственно. Переменная `randomNumbers` (строка 8) объявляется статической, чтобы она могла создаваться при выполнении программы и использоваться в методе `RollDice`.

```

1  // Ил. 7.8: Craps.cs
2  // Моделирование игры "крэпс".
3  using System;
4
5  public class Craps
6  {
7      // Создание генератора случайных чисел для метода RollDice
8      private static Random randomNumbers = new Random();
9
10     // Перечисление с константами, представляющими состояние игры
11     private enum Status { CONTINUE, WON, LOST }
12
13     // Перечисление с константами распространенных комбинаций
14     private enum DiceNames
15     {
16         SNAKE_EYES = 2,
17         TREY = 3,
18         SEVEN = 7,
19         YO_LEVEN = 11,
20         BOX_CARS = 12
21     }
22
23     // Партия в "крэпс"
24     public static void Main( string[] args )
25     {
26         // gameStatus может содержать значения CONTINUE, WON и LOST
27         Status gameStatus = Status.CONTINUE;
28         int myPoint = 0; // Пойнт при отсутствии выигрыша или проигрыша
29             // при первом броске
30         int sumOfDice = RollDice(); // Первый бросок кубика
31
32         // Определение состояния игры и пойнта по первому броску
33         switch ( ( DiceNames ) sumOfDice )
34         {
35             case DiceNames.SEVEN: // Победа с 7 при первом броске
36             case DiceNames.YO_LEVEN: // Победа с 11 при первом броске
37                 gameStatus = Status.WON;
38                 break;
39             case DiceNames.SNAKE_EYES: // Проигрыш с 2 при первом броске
40             case DiceNames.TREY: // Проигрыш с 3 при первом броске
41             case DiceNames.BOX_CARS: // Проигрыш с 12 при первом броске
42                 gameStatus = Status.LOST;
43                 break;
44             default: // Ни выигрыша, ни проигрыша; запомнить пойнт
45                 gameStatus = Status.CONTINUE; // Игра продолжается
46                 myPoint = sumOfDice; // Сохранение пойнта
47                 Console.WriteLine( "Point is {0}", myPoint );

```

**Ил. 7.8.** Класс `Craps` моделирует игру «крэпс» (продолжение ↗)

```

48         break;
49     } // Конец switch
50
51     // Пока игра не закончена
52     while ( gameStatus == Status.CONTINUE ) // Не WON и не LOST
53     {
54         sumOfDice = RollDice(); // Повторный бросок
55
56         // Определение состояния игры
57         if ( sumOfDice == myPoint ) // Победа при достижении пойнта
58             gameStatus = Status.WON;
59         else
60             // Проигрыш по выпадению 7 перед пойнтом
61             if ( sumOfDice == ( int ) DiceNames.SEVEN )
62                 gameStatus = Status.LOST;
63     } // Конец while
64
65     // Вывод сообщения о выигрыше или проигрыше
66     if ( gameStatus == Status.WON )
67         Console.WriteLine( "Player wins" );
68     else
69         Console.WriteLine( "Player loses" );
70 } // Конец Main
71
72 // Бросок кубиков, вычисление суммы и вывод результатов
73 public static int RollDice()
74 {
75     // Генерирование случайных значений
76     int die1 = randomNumbers.Next( 1, 7 ); // Первый кубик
77     int die2 = randomNumbers.Next( 1, 7 ); // Второй кубик
78
79     int sum = die1 + die2; // Сумма очков
80
81     // Вывод результатов броска
82     Console.WriteLine( "Player rolled {0} + {1} = {2}",
83         die1, die2, sum );
84     return sum; // Вернуть сумму
85 } // Конец метода RollDice
86 } // Конец класса Craps

```

```

Player rolled 2 + 5 = 7
Player wins

```

```

Player rolled 2 + 1 = 3
Player loses

```

```

Player rolled 2 + 4 = 6
Point is 6
Player rolled 3 + 1 = 4
Player rolled 5 + 5 = 10
Player rolled 6 + 1 = 7
Player loses

```

```

Player rolled 4 + 6 = 10

```

**Ил. 7.8.** Класс Craps моделирует игру «крэпс» (продолжение ↗)

```
Point is 10
Player rolled 1 + 3 = 4
Player rolled 1 + 3 = 4
Player rolled 2 + 3 = 5
Player rolled 4 + 4 = 8
Player rolled 6 + 6 = 12
Player rolled 4 + 4 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 6 = 8
Player rolled 6 + 6 = 12
Player rolled 6 + 4 = 10
Player wins
```

**Ил. 7.8.** Класс Craps моделирует игру «крэпс» (окончание)

### Метод RollDice

По правилам игры игрок должен бросать два кубика как в первый раз, так и при всех последующих бросках. Мы объявляем метод `RollDice` (строки 73–85), который моделирует броски кубиков, вычисляет и выводит их сумму. Метод `RollDice` объявляется один раз, но вызывается в двух местах (строки 30 и 54) метода `Main`, содержащего логику одной полной игры. Метод `RollDice` вызывается без аргументов, поэтому его список параметров пуст. При каждом вызове `RollDice` возвращает сумму очков на кубиках, поэтому в заголовке метода обозначен возвращаемый тип `int` (строка 73). Хотя строки 76 и 77 почти совпадают (за исключением имен переменных), это не значит, что они выдают одинаковый результат. Каждая команда генерирует случайное число в диапазоне от 1 до 6. Переменная `randomNumbers` (используемая в строках 76 и 77) не объявляется в этом методе — она объявляется как закрытая статическая переменная класса и инициализируется в строке 8. Это позволяет нам создать один объект `Random`, который повторно используется при всех последующих вызовах `RollDice`.

### Локальные переменные метода Main

Игрок может проиграть или выиграть как при первом броске, так и при любом из последующих бросков. Метод `Main` (строки 24–70) использует локальную переменную `gameStatus` (строка 27) для хранения общего состояния игры, локальную переменную `myPoint` (строка 28) для хранения пойнта, если игрок не выиграл и не проиграл при первом броске, и локальную переменную `sumOfDice` (строка 30) для хранения суммы очков последнего броска. Переменная `myPoint` инициализируется 0, чтобы приложение нормально компилировалось. Если не инициализировать `myPoint`, компилятор выдаст ошибку, потому что значение `myPoint` не присваивается во всех секциях команды `switch`, а следовательно, приложение может попытаться использовать переменную `myPoint` до того, как ей заведомо будет присвоено значение.

С другой стороны, переменная `gameStatus` не требует инициализации, потому что ей присваивается значение во всех ветвях команды `switch`, а следовательно, переменная заведомо будет инициализирована перед использованием. Впрочем, по правилам хорошего стиля программирования мы все равно ее инициализируем.



## Перечислимый тип Status

Локальная переменная `gameStatus` объявляется с типом `Status`, который мы объявляем в строке 11. Тип `Status` объявляется как закрытый член класса `Craps`, потому что `Status` может использоваться только в этом классе. `Status` представляет собой пользовательский тип, называемый *перечислимым типом* (или *перечислением*); он объявляет набор констант, представленных идентификаторами. Перечисление объявляется ключевым словом `enum` и именем типа (в нашем примере `Status`). Как и при объявлении классов классами, тело объявления `enum` заключается в фигурные скобки `{ }`. В скобках указывается список констант, разделенных запятыми. Имена констант должны быть уникальными — в отличие от значений, связываемых с каждой константой.

Переменной типа `Status` может быть присвоена только одна из трех констант, объявленных в перечислении. Если игрок выиграл, приложение присваивает локальной переменной `gameStatus` значение `Status.WON` (строки 37 и 58). Если игрок проиграл, приложение присваивает локальной переменной `gameStatus` значение `Status.LOST` (строки 42 и 62). В противном случае приложение присваивает `gameStatus` значение `Status.CONTINUE` (строка 45), которое означает, что потребуется еще один бросок.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 7.2

Использование констант перечислений (например, `Status.WON`, `Status.LOST` и `Status.CONTINUE`) вместо целочисленных литералов (0, 1 и 2) упрощает чтение и сопровождение кода.

## Первый бросок

Строка 30 метода `Main` вызывает метод `RollDice`, который выбирает два случайных числа от 1 до 6, выводит значения обоих кубиков и их сумму и возвращает сумму вызывающей стороне. Затем метод `Main` входит в команду `switch` (строки 33–49), которая использует значение `sumOfDice` из строки 30 для определения результата броска: игрок выиграл, игрок проиграл или игра должна продолжиться следующим броском.

## Перечисление DiceNames

Суммы, приводящие к выигрышу или проигрышу при первом броске, объявляются в перечислении `DiceNames` в строках 14–21. Они используются в секциях `case` команды `switch`. Имена констант позаимствованы из жаргона казино. Обратите внимание: в перечислении `DiceNames` для каждого идентификатора явно задается значение. В объявлении перечисления каждая константа представляет собой константное значение типа `int`. Если вы не присвоили значение идентификатору в объявлении перечисления, компилятор сделает это за вас. Если значение первой константы не задано, компилятор присваивает ей 0. Если не задано значение любой другой константы, компилятор присваивает ей значение на 1 больше значения предыдущей константы. Например, в перечислении `Status` компилятор неявно присваивает константе `Status.WON` значение 0, константе `Status.CONTINUE` — значение 1 и константе `Status.LOST` — значение 2.

### Базовый тип перечисления

В качестве базового типа перечисления также можно использовать тип `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`. Для этого следует использовать команду

```
private enum MyEnum : тип { Константа1, Константа2, ... }
```

где *тип* — один из целочисленных простых типов.

### Сравнение целочисленного типа с константой из перечисления

Если вам потребуется сравнить значение простого целочисленного типа со значением, представленным константой перечисляемого типа, используйте оператор преобразования. В команде `switch` в строках 33–49 мы преобразуем значение `sumOfDice (int)` к типу `DiceNames` и сравниваем его с каждой из констант `DiceNames`. Строки 35–36 проверяют, выиграл ли игрок при первом броске комбинацией `SEVEN (7)` или `YO_LEVEN (11)`. Строки 39–41 проверяют, проиграл ли игрок при первом броске комбинацией `SNAKE_EYES (2)`, `TREY (3)` или `BOX_CARS (12)`. После первого броска, если игра не закончена, секция `default` (строки 44–48) сохраняет `sumOfDice` в `myPoint` (строка 46) и выводит пойнт (строка 47).

### Дополнительные броски

Если мы все еще пытаемся «выбросить пойнт» (то есть игра продолжается с предыдущего броска), выполняется цикл в строках 52–63. В строке 54 снова бросаются кубики. Если `sumOfDice` совпадает с `myPoint` (строка 57), то строка 58 присваивает `gameStatus` значение `Status.WON`, а цикл завершается, потому что партия закончена. В строке 61 оператор преобразования типа `( int )` получает базовое значение константы `DiceNames.SEVEN` для сравнения с `sumOfDice`. Если значение `sumOfDice` равно `SEVEN (7)`, то строка 62 присваивает `gameStatus` значение `Status.LOST`, а цикл также завершается. При окончании игры строки 66–69 выводят сообщение о результате, а приложение завершается.

Обратите внимание на использование различных механизмов управления выполнением. Класс `Craps` содержит два метода `Main` и `RollDice` (дважды вызываемых из `Main`), а также управляющие команды `switch`, `while`, `if...else` и вложенные команды `if`. Также обратите внимание на использование секций `case` в команде `switch` для выполнения одних команд для сумм `SEVEN` и `YO_LEVEN` (строки 35–36) и для сумм `SNAKE_EYES`, `TREY` и `BOX_CARS` (строки 39–41). Чтобы легко создать команду `switch` со всеми возможными значениями перечисляемого типа, можно воспользоваться фрагментом кода `switch`. Введите ключевое слово `switch` в коде C# и дважды нажмите клавишу `Tab`. Если указать переменную перечисляемого типа в выражении команды `switch` и нажать `Enter`, для каждой константы из перечисления автоматически генерируется секция `case`.

## 7.11. Область действия объявлений

Вы уже видели объявления разных сущностей C#: классов, методов, свойств, переменных и параметров. Объявления создают имена, которые могут использоваться

для обращений к этим сущностям. *Областью действия* (scope) объявления называется часть приложения, которая может ссылаться на объявленную сущность по неуточненному имени.

В этом разделе представлены некоторые важные аспекты областей действия. Начнем с основных правил:

1. Область действия объявления параметра ограничивается телом метода, в котором это объявление находится.
2. Область действия объявления локальной переменной начинается в точке объявления и завершается в конце блока, содержащего объявление.
3. Область действия объявления локальной переменной, находящегося в секции инициализации заголовка команды `for`, ограничивается телом команды `for` и другими выражениями в заголовке.
4. Область действия метода, свойства или поля класса ограничивается телом класса. Это позволяет нестатическим методам и свойствам класса использовать любые поля, методы и свойства класса независимо от порядка их объявления. По этой же причине статические свойства и методы могут использовать любые статические члены класса.

Любой блок может содержать объявления переменных. Если имя локальной переменной или параметра метода совпадает с именем поля, то поле скрывается до завершения блока. Если вложенный блок в методе содержит переменную с таким же именем, как у локальной переменной во внешнем блоке, происходит ошибка компиляции. О том, как обратиться к скрытым полям, рассказано в главе 10. Приложение на ил. 7.9 демонстрирует некоторые аспекты области действия полей и локальных переменных.



### КАК ИЗБЕЖАТЬ ОШИБОК 7.1

Чтобы избежать коварных логических ошибок, возникающих при замещении локальной переменной вызываемого метода одноименного поля класса, присваивайте полям и локальным переменным разные имена.

```

1 // Ил. 7.9: Scope.cs
2 // Демонстрация области действия статических и локальных переменных.
3 using System;
4
5 public class Scope
6 {
7     // Статическая переменная, доступная для всех методов класса
8     private static int x = 1;
9
10    // Main создает и инициализирует локальную переменную x
11    // и вызывает методы UseLocalVariable и UseStaticVariable
12    public static void Main( string[] args )
13    {

```

**Ил. 7.9.** Класс `Scope` демонстрирует области действия статических и локальных переменных (продолжение ↗)

```

14     int x = 5; // Локальная переменная x скрывает статическую
15
16     Console.WriteLine( "local x in method Main is {0}", x );
17
18     // Содержит собственную локальную переменную x
19     UseLocalVariable();
20
21     // Использует статическую переменную x класса Scope
22     UseStaticVariable();
23
24     // Заново инициализирует свою локальную переменную x
25     UseLocalVariable();
26
27     // Статическая переменная x класса Scope сохраняет значение
28     UseStaticVariable();
29
30     Console.WriteLine( "\nlocal x in method Main is {0}", x );
31 } // Конец Main
32
33 // Создание и инициализация локальной переменной x при каждом вызове
34 public static void UseLocalVariable()
35 {
36     int x = 25; // Инициализируется при каждом вызове UseLocalVariable
37
38     Console.WriteLine(
39         "\nlocal x on entering method UseLocalVariable is {0}", x );
40     ++x; // Изменяет локальную переменную x этого метода
41     Console.WriteLine(
42         "local x before exiting method UseLocalVariable is {0}", x );
43 } // Конец метода UseLocalVariable
44
45 // Изменение статической переменной x класса Scope при каждом вызове
46 public static void UseStaticVariable()
47 {
48     Console.WriteLine( "\nstatic variable x on entering {0} is {1}",
49         "method UseStaticVariable", x );
50     x *= 10; // Изменение статической переменной x класса Scope
51     Console.WriteLine( "static variable x before exiting {0} is {1}",
52         "method UseStaticVariable", x );
53 } // Конец метода UseStaticVariable
54 } // Конец класса Scope

```

```
local x in method Main is 5
```

```
local x on entering method UseLocalVariable is 25
```


```
local x before exiting method UseLocalVariable is 26
```

```
static variable x on entering method UseStaticVariable is 1
```

```
static variable x before exiting method UseStaticVariable is 10
```

```
local x on entering method UseLocalVariable is 25
```

```
local x before exiting method UseLocalVariable is 26
```

**Ил. 7.9.** Класс Scope демонстрирует области действия статических и локальных переменных (продолжение )

```
static variable x on entering method UseStaticVariable is 10
static variable x before exiting method UseStaticVariable is 100

local x in method Main is 5
```

**Ил. 7.9.** Класс Score демонстрирует области действия статических и локальных переменных (окончание)

В строке 8 статическая переменная `x` объявляется и инициализируется значением 1. Статическая переменная скрывается в любом блоке (или методе), объявляющем локальную переменную с именем `x`. Метод `Main` (строки 12–31) объявляет локальную переменную `x` (строка 14) и инициализирует ее значением 5. Вывод значения этой локальной переменной демонстрирует, что статическая переменная `x` (со значением 1) скрыта в методе `Main`. Приложение объявляет два других метода — `UseLocalVariable` (строки 34–43) и `UseStaticVariable` (строки 46–53), которые не получают аргументов и не возвращают результатов. Метод `Main` вызывает каждый метод дважды (строки 19–28). Метод `UseLocalVariable` объявляет локальную переменную `x` (строка 36). При первом вызове `UseLocalVariable` (строка 19) метод создает локальную переменную `x` и инициализирует ее значением 25 (строка 36), выводит значение `x` (строки 38–39), увеличивает `x` (строка 40) и снова выводит значение `x` (строки 41–42). При повторном вызове `UseLocalVariable` (строка 25) локальная переменная `x` создается заново и снова инициализируется значением 25, так что выходные данные вызовов `UseLocalVariable` идентичны.

Метод `UseStaticVariable` не объявляет никаких локальных переменных. Соответственно когда он ссылается на `x`, используется статическая переменная `x` (строка 8) класса. При первом вызове метода `UseStaticVariable` (строка 22) выводится значение (1) статической переменной `x` (строки 48–49), которое умножается на 10 (строка 50), после чего новое значение (10) статической переменной `x` выводится снова перед возвратом управления (строки 51–52). При следующем вызове метода `UseStaticVariable` (строка 28) статическая переменная содержит измененное значение 10, так что метод сначала выводит 10, а потом 100. Наконец, в методе `Main` приложение снова выводит значение локальной переменной `x` (строка 30), демонстрируя, что вызовы метода не изменили локальную переменную `x`, потому что все методы обращались к переменной с именем `x` в других областях действия.

## 7.12. Перегрузка методов

В одном классе можно объявить несколько одноименных методов — при условии, что они имеют разные наборы параметров (определяемые количеством, типами и порядком параметров). Такое объявление называется *перегрузкой методов*. При вызове перегруженного метода компилятор C# выбирает подходящий метод в зависимости от количества, типа и порядка аргументов в вызове. Перегрузка

методов часто применяется для создания нескольких методов с одинаковыми именами, которые выполняют одинаковые или похожие задачи, но имеют разные типы или разное количество аргументов. Например, методы `Min` и `Max` класса `Math` (см. раздел 7.3) перегружаются в 11 версиях, которые определяют соответственно минимум и максимум для двух значений каждого из 11 числовых простых типов. Следующий пример демонстрирует объявление и вызов перегруженных методов. Примеры перегруженных конструкторов приведены в главе 10.

### Объявление перегруженных методов

Класс `MethodOverload` (ил. 7.10) включает две перегруженные версии метода `Square` — одна возводит в квадрат значение `int` (и возвращает `int`), а другая возводит в квадрат значение `double` (и возвращает `double`). И хотя эти методы имеют одинаковые имена и похожие списки параметров и тела, их проще рассматривать как разные методы.

```
1 // Ил. 7.10: MethodOverload.cs
2 // Перегрузка объявлений методов.
3 using System;
4
5 public class MethodOverload
6 {
7     // Тестирование перегруженных версий Square
8     public static void Main( string[] args )
9     {
10         Console.WriteLine( "Square of integer 7 is {0}", Square( 7 ) );
11         Console.WriteLine( "Square of double 7.5 is {0}", Square( 7.5 ) );
12     } // Конец Main
13
14     // Метод Square с аргументом int
15     public static int Square( int intValue )
16     {
17         Console.WriteLine( "Called square with int argument: {0}",
18             intValue );
19         return intValue * intValue;
20     } // Конец метода Square с аргументом int
21
22     // Метод Square с аргументом double
23     public static double Square( double doubleValue )
24     {
25         Console.WriteLine( "Called square with double argument: {0}",
26             doubleValue );
27         return doubleValue * doubleValue;
28     } // Конец метода Square с аргументом double
29 } // Конец класса MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49
Called square with double argument: 7.5
Square of double 7.5 is 56.25
```

**Ил. 7.10.** Перегрузка объявлений методов

Строка 10 метода `Main` вызывает метод `Square` с аргументом 7. Литеральные целочисленные значения интерпретируются как тип `int`, так что вызов в строке 10 активизирует версию `Square` с параметром `int` (строки 15–20). Аналогичным образом строка 11 вызывает метод `Square` с аргументом 7.5. Литеральные вещественные значения интерпретируются как тип `double`, так что вызов в строке 11 активизирует версию `Square` в строках 23–28 с параметром `double`. Каждый метод сначала выводит строку текста, доказывающую, что в каждом случае вызывается правильная версия метода.

Обратите внимание: перегруженные методы на ил. 7.10 выполняют одинаковые вычисления, но с двумя разными типами. Механизм *обобщений* C# позволяет написать один «обобщенный» метод, который выполняет те же задачи, что и целый набор обобщенных методов. Обобщенные методы рассматриваются в главе 20.

### Выбор перегруженной версии метода

Компилятор различает перегруженные методы по сигнатуре — совокупности имени метода и количества, типов и порядка параметров. Сигнатура также включает способ передачи параметров, который может изменяться ключевыми словами `ref` и `out` (см. раздел 7.16). Если бы компилятор в процессе компиляции обращал внимание только на имена методов, код на ил. 7.10 был бы неоднозначным — компилятор не смог бы различить два метода `Square` (строки 15–20 и 23–28). Во внутренней реализации компилятор проверяет уникальность методов класса по сигнатуре. Например, на ил. 7.10 компилятор на основании сигнатуры различает методы «`Square` для `int`» (метод `Square` с параметром `int`) и «`Square` для `double`» (метод `Square` с параметром `double`). Если объявление метода `Method1` начинается с заголовка

```
void Method1( int a, float b )
```

то этот метод имеет иную сигнатуру, чем метод, объявленный с заголовком

```
void Method1( float a, int b )
```

Порядок типов параметров важен — компилятор считает два приведенных заголовка `Method1` разными.

### Возвращаемые типы перегруженных методов

Говоря о сигнатурах методов, используемых компилятором, мы не упомянули о возвращаемых типах. Дело в том, что вызовы методов не могут различаться по возвращаемому типу. Приложение на ил. 7.11 демонстрирует ошибки компиляции, сгенерированные для двух методов с одинаковыми сигнатурами при разных возвращаемых типах. Если списки параметров методов различаются, возвращаемые типы перегруженных версий могут различаться, а могут и совпадать. Кроме того, перегруженные методы не обязаны иметь одинаковое количество параметров.



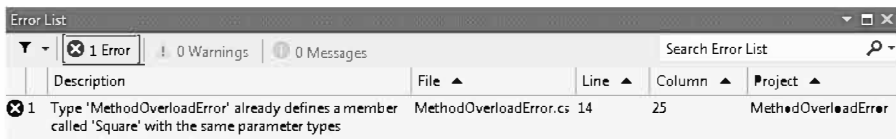
#### ТИПИЧНАЯ ОШИБКА 7.9

Объявление перегруженных методов с одинаковыми списками параметров приводит к ошибке компиляции независимо от того, совпадают или различаются возвращаемые типы.

```

1 // Ил. 7.11: MethodOverload.cs
2 // Перегруженные методы с идентичными сигнатурами приводят к ошибке
3 // компиляции даже в том случае, если возвращаемые типы различаются.
4 public class MethodOverloadError
5 {
6     // Объявление метода Square с аргументом int
7     public int Square( int x )
8     {
9         return x * x;
10    } // Конец метода Square
11
12    // Второе объявление метода Square с аргументом int.
13    // Компилятор выдает ошибку, хотя возвращаемые типы различаются.
14    public double Square( int y )
15    {
16        return y * y;
17    } // Конец метода Square
18 } // Конец класса MethodOverloadError

```



**Ил. 7.11.** Перегруженные методы с идентичными сигнатурами приводят к ошибке компиляции даже при различающихся возвращаемых типах

## 7.13. Необязательные параметры

Методы могут иметь необязательные параметры, позволяющие передавать разное количество аргументов при вызове. Необязательный параметр определяет значение по умолчанию, которое присваивается ему при отсутствии соответствующего аргумента.

Методы можно создавать с одним или несколькими параметрами. Все необязательные параметры метода должны располагаться справа от обязательных, то есть *в конце списка параметров*.



### ТИПИЧНАЯ ОШИБКА 7.10

Объявление необязательного параметра справа от обязательного является ошибкой компиляции.

Если для параметра задано значение по умолчанию, вызывающая сторона может опустить соответствующий аргумент. Например, в заголовке метода

```
public int Power( int baseValue, int exponentValue = 2 )
```

определяется необязательный второй параметр. При вызове `Power` должен передаваться как минимум аргумент для параметра `baseValue`, в противном случае произойдет ошибка компиляции. Также `Power` может передаваться второй аргумент (для параметра `exponentValue`). Рассмотрим следующие вызовы `Power`:



```
Power()
Power(10)
Power(10, 3)
```

Для первого вызова компилятор выдает ошибку, потому что методу должен передаваться минимум один аргумент. Второй вызов допустим, потому что в нем передается один аргумент (10) — необязательный параметр `exponentValue` при вызове не указывается. Последний вызов тоже допустим; в обязательном аргументе передается значение 10, а в необязательном — значение 3.

При вызове, передающем только один аргумент (10), параметр `exponentValue` по умолчанию равен 2 (значение, указанное в заголовке метода). Для каждого необязательного параметра должно задаваться значение по умолчанию, состоящее из знака равенства (=) и значения. Например, заголовок `Power` назначает `exponentValue` значение по умолчанию 2.

На ил. 7.12 продемонстрировано использование необязательного параметра. Программа вычисляет результат возведения в степень. Метод `Power` (строки 15–23) определяет второй параметр как необязательный. В строках 10–11 метода `DisplayPowers` вызывается метод `Power`. В строке 10 он вызывается без необязательного второго аргумента. В этом случае компилятор предоставляет второй аргумент 2; для необязательного аргумента используется значение по умолчанию, невидимое при вызове.

```
1 // Ил. 7.12: Power.vb
2 // Использование необязательных аргументов.
3 using System;
4
5 class CalculatePowers
6 {
7     // Вызов Power с необязательным аргументом и без
8     public static void Main( string[] args )
9     {
10         Console.WriteLine( "Power(10) = {0}", Power( 10 ) );
11         Console.WriteLine( "Power(2, 10) = {0}", Power( 2, 10 ) );
12     } // Конец Main
13
14     // Использование цикла для возведения в степень
15     public int Power( int baseValue, int exponentValue = 2 )
16     {
17         int result = 1; // initialize total
18
19         for ( int i = 1; i <= exponentValue; i++ )
20             result *= baseValue;
21
22         return result;
23     } // Конец метода Power
24 } // Конец класса CalculatePowers
```

```
Power(10) = 100
Power(2, 10) = 1024
```

**Ил. 7.12.** Использование необязательных аргументов

## 7.14. Именованные параметры

Обычно при вызове метода с необязательными параметрами значения аргументов — в указанном порядке — присваиваются параметрам слева направо по списку параметров. Допустим, класс `Time` хранит время суток в 24-часовом формате с тремя компонентами: часы (0–23), минуты (0–59) и секунды (0–59). Такой класс может предоставлять метод установки времени `SetTime` с необязательными параметрами:

```
public void SetTime( int hour = 0, int minute = 0, int second = 0 )
```

В приведенном заголовке все три параметра `SetTime` не являются обязательными. Допустим, для объекта `Time` с именем `t` вызов `SetTime` может выглядеть следующим образом:

```
t.SetTime(); // Устанавливается время 00:00:00
t.SetTime( 12 ); // Устанавливается время 12:00:00
t.SetTime( 12, 30 ); // Устанавливается время 12:30:00
t.SetTime( 12, 30, 22 ); // Устанавливается время 12:30:22
```

В первом вызове не задан ни один аргумент, поэтому компилятор присваивает 0 каждому параметру. Во втором вызове компилятор присваивает аргумент 12 первому параметру `hour`, а второй и третий параметры имеют значение по умолчанию 0. В третьем вызове компилятор присваивает два аргумента, 12 и 30, параметрам `hour` и `minute` соответственно, а параметру `second` присваивается значение по умолчанию 0. В последнем вызове компилятор присваивает три аргумента, 12, 30 и 22, параметрам `hour`, `minute` и `second` соответственно.

Но что если вы захотите передать аргументы только для `hour` и `second`? Казалось бы, метод можно вызвать следующим образом:

```
t.SetTime( 12, , 22 ); // ОШИБКА КОМПИЛЯЦИИ
```

Однако в отличие от некоторых языков программирования, в С# подобный пропуск аргументов не поддерживается. В С# реализован механизм *именованных параметров*, позволяющий при вызове методов с необязательными параметрами передавать только те необязательные аргументы, которые вы пожелаете указать. Для этого в списке аргументов метода указывается имя и значение параметра, разделенные двоеточием. Например, приведенная выше команда может быть реализована так:

```
t.SetTime( hour: 12, second: 22 ); // Устанавливается время 12:00:22
```

В этом случае компилятор присваивает параметру `hour` аргумент 12, а параметру `second` — аргумент 22. Параметр `minute` не задан, и компилятор присваивает ему значение по умолчанию 0. При использовании именованных параметров возможна передача аргументов в порядке, отличном от порядка их объявления. Аргументы для обязательных параметров должны передаваться всегда.

## 7.15. Рекурсия

Приложения, рассматривавшиеся до настоящего момента, состояли из методов, которые организованно вызывали друг друга по иерархическому принципу. Однако в некоторых задачах методу бывает полезно вызывать самого себя. Метод, который вызывает сам себя (напрямую или косвенно через другой метод), называется *рекурсивным*.

Сначала мы рассмотрим рекурсию на концептуальном уровне, а затем займемся приложением, содержащим рекурсивный метод. Все рекурсивные решения имеют ряд общих элементов. Рекурсивный метод, вызываемый для решения задачи, в действительности способен напрямую решить только самый простейший (базовый) случай. Если рекурсивный метод вызывается для базового случая, он возвращает результат. Если метод вызывается для более сложной задачи, то эта задача делится на две концептуальные части: решение одной методу известно, а как решать другую, он не знает. Вторая часть должна напоминать исходную задачу, но быть упрощенным или слегка уменьшенным ее вариантом. Так как новая задача похожа на исходную, метод вызывает новую копию (или несколько новых копий) самого себя для работы над меньшей задачей; это называется *рекурсивным вызовом* (или *шагом рекурсии*). Шаг рекурсии обычно включает команду `return`, результат которой объединяется с частью задачи, известной методу, для получения результата, возвращаемого исходной вызывающей стороне.

Шаг рекурсии может привести к созданию множества дополнительных рекурсивных вызовов, так как каждая новая подзадача также делится на две части. Чтобы рекурсия в какой-то момент завершилась, метод каждый раз вызывает себя с последовательно упрощаемой версией исходной задачи. Эта серия упрощаемых задач должна сводиться к базовому случаю. В этот момент метод распознает базовый случай и возвращает результат предыдущей копии. Далее следует серия возвратов до того момента, когда исходный вызов метода вернет результат на сторону вызова. Впрочем, это описание выглядит гораздо сложнее классических решений задач, которые мы рассматривали до настоящего момента.

### Рекурсивное вычисление факториала

В качестве примера использования рекурсии мы напишем приложение для выполнения известных математических вычислений. Факториал неотрицательного целого числа  $n$  обозначается  $n!$  и представляет собой произведение

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

Значение  $1!$  равно 1, а значение  $0!$  по определению равно 1. Например,  $5!$  вычисляется как произведение  $5 \times 4 \times 3 \times 2 \times 1$ , то есть 120.

Факториал целого числа `number`, большего либо равного 0, может быть вычислен итеративно (то есть без использования рекурсии) командой `for` следующего вида:

```
factorial = 1;

for ( int counter = number; counter >= 1; --counter )
    factorial *= counter;
```

Чтобы сформулировать рекурсивное объявление, достаточно заметить следующую связь:

$$n! = n \cdot (n - 1)!$$

Например, очевидно, что значение  $5!$  равно  $5 \times 4!$ :

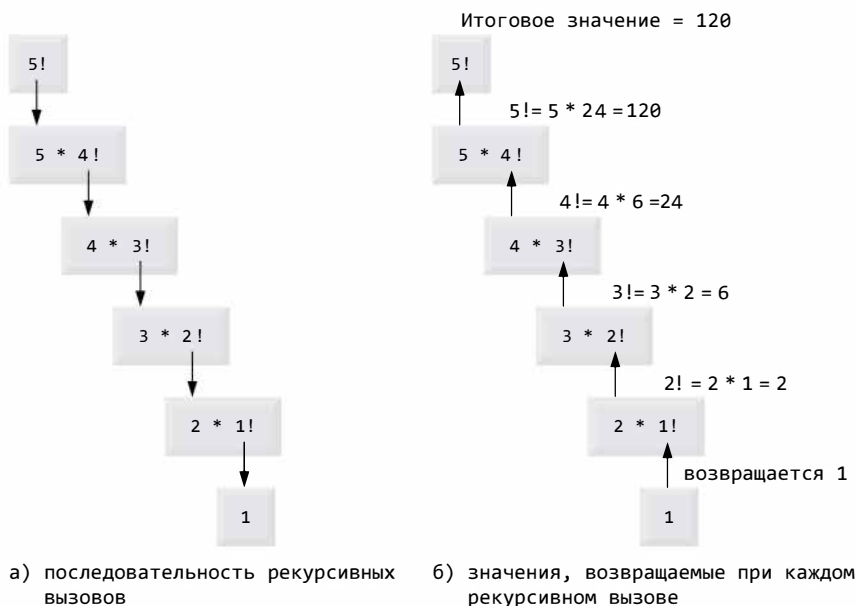
$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

$$5! = 5 \times (4 \times 3 \times 2 \times 1)$$

$$5! = 5 \times (4!)$$

Процесс вычисления  $5!$  представлен на ил. 7.13. На ил. 7.13, а показано, как серия рекурсивных вызовов продолжается до тех пор, пока вычисление  $1!$  не даст результат 1; на этом рекурсия завершается. На ил. 7.13, б показаны значения, возвращаемые при каждом рекурсивном вызове до момента вычисления и возвращения исходного значения.

В листинге на ил. 7.14 рекурсия используется для вычисления и вывода факториалов целых чисел от 0 до 10. Рекурсивный метод `Factorial` (строки 16–24) сначала проверяет истинность завершающего условия (строка 19). Если значение `number` меньше либо равно 1 (базовый случай), то `Factorial` возвращает 1, дальнейшая рекурсия не нужна, а метод возвращает управление. Если `number` больше 1, то в строке 23 результат выражается как произведение `number` и рекурсивного вызова `Factorial`, вычисляющего факториал `number - 1`, — задача, несколько упрощенная по сравнению с исходным вычислением `Factorial( number )`.



**Ил. 7.13.** Рекурсивное вычисление  $5!$

```

1 // Ил. 7.14: FactorialTest.cs
2 // Рекурсивный метод Factorial.
3 using System;
4
5 public class FactorialTest
6 {
7     public static void Main( string[] args )
8     {
9         // Вычисление факториалов от 0 до 10
10        for ( long counter = 0; counter <= 10; ++counter )
11            Console.WriteLine( "{0}! = {1}",
12                               counter, Factorial( counter ) );
13    } // Конец Main
14
15    // Рекурсивное объявление метода Factorial
16    public static long Factorial( long number )
17    {
18        // Базовый случай
19        if ( number <= 1 )
20            return 1;
21        // Шаг рекурсии
22        else
23            return number * Factorial( number - 1 );
24    } // Конец метода Factorial
25 } // Конец класса FactorialTest

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

#### Ил. 7.14. Рекурсивный метод Factorial

Метод `Factorial` (строки 16–24) получает параметр типа `long` и возвращает результат типа `long`. Как видно из ил. 7.14, значения факториала стремительно возрастают. Мы выбрали тип `long` (способный представлять относительно большие целые числа), чтобы приложение могло вычислять факториалы более 20!. К сожалению, факториалы увеличиваются так быстро, что вскоре они превышают даже максимальное значение, хранящееся в переменной `long`. Из-за ограничений целочисленных типов для вычисления факториалов больших чисел могут потребоваться переменные типа `float`, `double` или `decimal`. В этом проявляется общая слабость многих традиционных языков программирования — невозможность простого расширения под специфические потребности разных приложений. В C# вы можете создать собственный тип для вычисления факториалов произвольных чисел или же использовать новый тип `BigInteger` из библиотеки классов .NET Framework.



### ТИПИЧНАЯ ОШИБКА 7.11

Если вы забудете запрограммировать базовый случай или неправильно реализуете шаг рекурсии и она не будет сходиться к базовому случаю, возникает бесконечная рекурсия, которая в конечном итоге приведет к исчерпанию всей свободной памяти. Эта ошибка аналогична ошибке заикливания в итеративных (нерекурсивных) решениях.

## 7.16. Передача аргументов по значению и по ссылке

Во многих языках программирования поддерживаются два способа передачи аргументов функциям: передача по значению и передача по ссылке. При передаче аргумента по значению (используется по умолчанию в C#) создается копия значения, которая передается вызываемой функции. Изменения в копии не отражаются на значении исходной переменной на стороне вызова. Тем самым предотвращаются случайные побочные эффекты, которые так сильно мешают разработке правильных и надежных программных систем. Во всех примерах этой главы аргументы передаются по значению. Когда аргумент передается по ссылке, метод получает возможность обращаться к исходной переменной на стороне вызова и изменять ее значение.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 7.3

Передача по ссылке может снизить уровень безопасности, потому что вызываемая функция может повредить данные на стороне вызова.

Чтобы передать объект по ссылке, просто передайте в аргументе при вызове метода переменную, содержащую ссылку на объект. Затем в теле цикла обратитесь к объекту по имени параметра. Параметр ссылается на исходный объект в памяти, так что вызываемый метод может работать с исходным объектом напрямую.

Ранее мы упоминали различия между значимыми и ссылочными типами. Основное различие между ними заключается в том, что *в переменных значимого типа хранятся значения*, так что при передаче переменной значимого типа методу передается копия значения этой переменной. *В переменных ссылочного типа хранятся ссылки на объекты*, так что при передаче переменной ссылочного типа в аргументе передается копия ссылки, указывающей на объект. Несмотря на то что сама ссылка передается по значению, метод может использовать полученную ссылку для взаимодействия с исходным объектом (и его возможной модификации). Аналогичным образом при получении информации из метода командой `return` метод возвращает копию значения, хранящегося в переменной значимого типа, или копию ссылки, хранящейся в переменной ссылочного типа. При возвращении ссылки вызывающий метод может использовать ссылку для взаимодействия с объектом, на который она указывает.

### ref и out

А если вы хотите передать переменную по ссылке, чтобы вызванный метод мог изменить ее значение? Для этого в C# существуют ключевые слова `ref` и `out`. Если в объявлении параметра присутствует ключевое слово `ref`, переменная передается по ссылке — вызывающий метод может изменить исходную переменную на стороне вызова. Ключевое слово `ref` используется для переменных, уже инициализированных в вызывающем методе. Если при вызове метода в аргументе параметра `ref` передается неинициализированная переменная, компилятор выдает сообщение об ошибке. Ключевое слово `out` обозначает выходной параметр; оно указывает, что аргумент будет передаваться вызванному методу по ссылке, а вызванный метод присвоит значение исходной переменной на стороне вызова. Если метод не присваивает значение выходному параметру во всех возможных путях выполнения, компилятор выдает сообщение об ошибке. При этом компилятор не считает ошибкой передачу неинициализированной переменной в аргументе метода. Формально метод может вернуть только одно значение вызывающей стороне в команде `return`, но параметры `ref` и/или `out` позволяют вернуть дополнительные значения.

Также переменную ссылочного типа можно передать по ссылке; это позволит изменить ее таким образом, чтобы она ссылалась на новый объект. Передача ссылочных переменных по ссылке — непростой, но мощный прием, который будет рассмотрен в разделе 8.8.

### Пример использования ref и out

Приложение на ил. 7.15 использует ключевые слова `ref` и `out` для работы с целочисленными значениями. Класс содержит три метода для возведения в квадрат целых чисел. Метод `SquareRef` (строки 37–40) умножает параметр `x` на себя и присваивает новое значение `x`. Параметр `SquareRef` объявляется с ключевыми словами `ref int`, которые обозначают целочисленный аргумент, передаваемый по ссылке. Так как аргумент передается по ссылке, присваивание в строке 39 изменяет значение исходного аргумента на стороне вызова.

Метод `SquareOut` (строки 44–48) присваивает своему параметру значение 6 (строка 46), после чего возводит его в квадрат. Параметр `SquareOut` объявляется с ключевыми словами `out int`, обозначающими целочисленный аргумент, передаваемый по ссылке, который не обязательно инициализировать заранее.

```
1 // Ил. 7.15: ReferenceAndOutputParameters.cs
2 // Ключевые слова ref/out и передача значений.
3 using System;
4
5 class ReferenceAndOutputParameters
6 {
7     // Вызов методов с разными объявлениями параметров
8     public static void Main( string[] args )
9     {
10         int y = 5; // Инициализирует y значением 5
11         int z; // Объявляет z, но не инициализирует
```

**Ил. 7.15.** Ключевые слова `ref/out` и передача значений (продолжение ➤)

```

12
13 // Выход исходных значений y и z
14 Console.WriteLine( "Original value of y: {0}", y );
15 Console.WriteLine( "Original value of z: uninitialized\n" );
16
17 // Передача y и z по ссылке
18 SquareRef( ref y ); // Необходимо использовать ключевое слово ref
19 SquareOut( out z ); // Необходимо использовать ключевое слово out
20
21 // Вывод значений y и z после их изменения
22 // методами SquareRef и SquareOut соответственно
23 Console.WriteLine( "Value of y after SquareRef: {0}", y );
24 Console.WriteLine( "Value of z after SquareOut: {0}\n", z );
25
26 // Передача y и z по значению
27 Square( y );
28 Square( z );
29
30 // Вывод значений y и z после передачи методу Square демонстрирует,
31 // что переданные по значению аргументы не изменяются
32 Console.WriteLine( "Value of y after Square: {0}", y );
33 Console.WriteLine( "Value of z after Square: {0}", z );
34 } // Конец Main
35
36 // использует ref-параметр x для изменения переменной
37 static void SquareRef( ref int x ) // на стороне вызова
38 {
39     x = x * x; // Переменная на стороне вызова возводится в квадрат
40 } // Конец метода SquareRef
41
42 // Использует out-параметр x для присваивания значения
43 // неинициализированной переменной
44 static void SquareOut( out int x )
45 {
46     x = 6; // Присваивает значение переменной на стороне вызова
47     x = x * x; // Возводит в квадрат переменную на стороне вызова
48 } // Конец метода SquareOut
49
50 // Параметр x получает копию значения, переданного в аргументе,
51 // так что переменная на стороне вызова не изменяется.
52 static void Square( int x )
53 {
54     x = x * x;
55 } // Конец метода Square
56 } // Конец класса ReferenceAndOutputParameters

```

```

Original value of y: 5
Original value of z: uninitialized

```

```

Value of y after SquareRef: 25
Value of z after SquareOut: 36

```

```

Value of y after Square: 25
Value of z after Square: 36

```

**Ил. 7.15.** Ключевые слова ref/out и передача значений (окончание)



Метод `Square` (строки 52–55) умножает параметр `x` на себя и присваивает новое значение `x`. При вызове метода в параметре `x` передается копия аргумента. Хотя параметр `x` изменяется в методе, исходное значение на стороне вызова не изменяется.

В методе `Main` (строки 8–34) вызываются методы `SquareRef`, `SquareOut` и `Square`. Сначала переменная `y` инициализируется значением 5, а переменная `z` объявляется, но не инициализируется. В строках 18–19 вызываются методы `SquareRef` и `SquareOut`. Обратите внимание: при передаче переменной методу со ссылочным параметром перед аргументом необходимо поставить то же ключевое слово (`ref` или `out`), которое использовалось при объявлении ссылочного параметра. В строках 23–24 выводятся значения `y` и `z` после вызовов `SquareRef` и `SquareOut`. Обратите внимание: переменные `y` и `z` стали равны 25 и 36 соответственно.

В строках 27–28 метод `Square` вызывается с аргументами `y` и `z`. В этом случае обе переменные передаются по значению, то есть метод `Square` получает копии их значений. В результате значения `y` и `z` остаются равными 25 и 36 соответственно. Вывод значений `y` и `z` в строках 32–33 показывает, что они *не* изменились.



#### ТИПИЧНАЯ ОШИБКА 7.12

Аргументы `ref` и `out` при вызове метода должны соответствовать параметрам, указанным в объявлении метода; в противном случае компилятор выдает сообщение об ошибке.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 7.4

По умолчанию `C#` не позволяет выбрать, как должен передаваться каждый аргумент — по значению или по ссылке. Значимые типы передаются по значению. Объекты методам не передаются; вместо них передаются ссылки на объекты. Ссылки сами по себе передаются по значению. Когда метод получает ссылку на объект, он может работать с объектом напрямую, но при этом значение ссылки не может быть изменено так, чтобы ссылка стала указывать на другой объект. В разделе 8.8 будет показано, что ссылки тоже могут передаваться по ссылке.

## 7.17. Итоги

В этой главе были рассмотрены различия между нестатическими и статическими методами. Вы узнали, что для вызова статического метода перед его именем указывается имя класса и оператор «точка». Статические методы были представлены на примере методов класса `Math` из библиотеки `.NET Framework Class Library`, предназначенных для выполнения математических вычислений. Мы рассмотрели некоторые часто используемые пространства имен `Framework Class Library` и выяснили, как использовать оператор `+` для выполнения конкатенации строк.

Далее были представлены два способа объявления константных значений — с ключевым словом `const` и с перечислимыми типами. Мы рассмотрели концепцию области

действия полей и локальных переменных класса и возможность перегрузки методов посредством определения методов с одинаковыми именами, но разными сигнатурами. Вы научились использовать необязательные и именованные параметры, а также узнали, как рекурсивные методы вызывают сами себя для последовательного разбиения больших задач на меньшие вплоть до решения исходной задачи. Глава завершается описанием различий между значимыми и ссылочными типами в отношении их передачи методам, а также примером использования ключевых слов `ref` и `out` для передачи аргументов по ссылке.

# 8

# Массивы и обработка исключений

## 8.1. Введение

В этой главе рассматривается важная тема *структур данных* — наборов взаимосвязанных элементов данных. Массив представляет собой структуру данных, состоящую из взаимосвязанных элементов данных одного типа. Массивы имеют фиксированную длину; после создания их длина остается неизменной, хотя переменной массива можно присвоить новое значение, чтобы она ссылалась на новый массив другой длины. После описания синтаксиса объявления, создания и инициализации массивов будут представлены примеры, демонстрирующие некоторые стандартные операции с массивами. В частности, мы воспользуемся массивами для моделирования тасования колоды и сдачи карт. В этой главе также будет представлена последняя управляющая команда C# — цикл `foreach`, предоставляющий компактную запись для перебора данных в массивах (и других структурах данных, как будет показано в главе 9 и последующих главах книги). Мы доработаем пример `GradeBook` с использованием массивов, чтобы класс мог хранить набор оценок и анализировать оценки студентов по нескольким экзаменам.

## 8.2. Массивы

Массив представляет собой набор переменных (называемых *элементами*), содержащих значения одного типа. Вспомните, что типы делятся на две категории — значимые и ссылочные. Массивы относятся к ссылочным типам. Как вы увидите, то, что мы обычно рассматриваем как массив, в действительности является ссылкой на объект массива. Элементы массива могут относиться как к значимым, так и к ссылочным типам (включая другие массивы). Чтобы обратиться к конкретному

элементу массива, следует указать имя ссылки на массив и порядковый номер элемента в массиве (*индекс элемента*).

На ил. 8.1 изображено логическое представление целочисленного массива с именем `c`, заполненного тестовыми данными. Массив состоит из 12 элементов. Приложение обращается к любому элементу с использованием выражения, включающего имя массива и индекс элемента в квадратных скобках (`[ ]`). Индекс первого элемента в любом массиве равен 0. Таким образом, элементы массива `c` обозначаются `c[0]`, `c[1]`, `c[2]` и т. д. Максимальный индекс в массиве `c` равен 11, что на 1 меньше количества элементов в массиве, потому что индексы начинаются с 0. Имена массивов подчиняются тем же правилам, что и имена других переменных.

Имя переменной массива

Индекс элемента массива `c`

<code>c[ 0 ]</code>	-45
<code>c[ 1 ]</code>	6
<code>c[ 2 ]</code>	0
<code>c[ 3 ]</code>	72
<code>c[ 4 ]</code>	1543
<code>c[ 5 ]</code>	-89
<code>c[ 6 ]</code>	0
<code>c[ 7 ]</code>	62
<code>c[ 8 ]</code>	-3
<code>c[ 9 ]</code>	1
<code>c[ 10 ]</code>	6453
<code>c[ 11 ]</code>	78

**Ил. 8.1.** Массив с 12 элементами

Индекс должен быть неотрицательным целым числом; допускается использование выражений. Например, если переменная `a` равна 5, а переменная `b` равна 6, команда `c[ a + b ] += 2;`

увеличивает элемент массива `c[11]` на 2. Имя массива с индексом называется *выражением обращения к элементу массива*. Такие выражения могут использоваться в левой части оператора присваивания для присваивания нового значения элементу массива. Индекс массива должен быть значением типа `int`, `uint`, `long` или `ulong` или значением типа, который может быть неявно приведен к одному из этих типов.

Рассмотрим массив на ил. 8.1 более подробно. Здесь `c` — имя переменной, ссылающейся на массив. Каждый экземпляр массива знает свою длину и предоставляет доступ к этой информации через свойство `Length`. Например, выражение `c.Length` использует свойство `Length` массива `c` для определения длины массива (то есть 12). Свойство `Length` массива нельзя изменить, потому что оно не предоставляет сет-метод доступа. 12 элементов массива обозначаются `c[0]`, `c[1]`, `c[2]`, ..., `c[11]`. Попытка обращения к элементам за пределами этого диапазона (например, `c[-1]` или `c[12]`)

является ошибкой времени выполнения (как будет показано на ил. 8.8). Элемент `c[0]` содержит значение `-45`, элемент `c[1]` — `6`, элемент `c[2]` — `0` и т. д. Чтобы вычислить сумму значений, содержащихся в первых трех элементах массива `c`, и сохранить результат в переменной `sum`, можно использовать следующую команду:

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

Следующая команда делит значение `c[6]` на `2` и присваивает результат переменной `x`:

```
x = c[ 6 ] / 2;
```

## 8.3. Объявление и создание массивов

Массивы занимают место в памяти. Так как они являются объектами, обычно они создаются ключевым словом `new`. При создании объекта массива тип и количество его элементов указываются в *выражении создания массива*, использующем ключевое слово `new`. Такое выражение возвращает ссылку, которая сохраняется в переменной массива. Следующее объявление и выражение создания массива создает объект массива с 12 элементами и сохраняет ссылку на него в переменной `c`:

```
int[] c = new int[ 12 ];
```

Выражение может использоваться для создания массива на ил. 8.1 (но не исходных значений в массиве — вскоре вы увидите, как происходит инициализация элементов массива). Эта задача решается следующим образом:

```
int[] c; // Объявление переменной массива
c = new int[ 12 ]; // Создание массива; присваивание переменной массива
```

В этом объявлении квадратные скобки за типом `int` указывают, что переменная `c` будет ссылаться на массив с элементами `int` (то есть в `c` будет храниться ссылка на объект массива). Во второй команде переменной массива `c` присваивается ссылка на новый объект массива с 12 элементами `int`. Количество элементов также может задаваться выражением, вычисляемым на стадии выполнения. При создании массива каждому элементу присваивается значение по умолчанию — `0` для простых числовых типов, `false` для элементов `bool` и `null` для ссылок. Как вы вскоре увидите, при создании массива также можно явно задать начальные значения элементов.



### ТИПИЧНАЯ ОШИБКА 8.1

В объявлении переменной, ссылающейся на массив, указание количества элементов в квадратных скобках (например, `int[12] c;`) является синтаксической ошибкой.

Приложение может создать несколько массивов в одном объявлении. Следующая команда резервирует память для строкового массива `b` из 100 элементов и строкового массива `x` из 27 элементов:

```
string[] b = new string[ 100 ], x = newstring[ 27 ];
```

В этой команде `string[]` применяется к каждой переменной. Впрочем, для удобства чтения кода и простоты комментирования такие объявления обычно разбиваются на строки:

```
string[] b = new string[ 100 ]; // Создание строкового массива b
string[] x = new string[ 27 ];  // Создание строкового массива x
```

Приложение может объявлять переменные, ссылающиеся на массивы с элементами значимых или ссылочных типов. Например, каждый элемент массива `int` является значением типа `int`, а каждый элемент массива `string` содержит ссылку на объект `string`.

### Изменение размеров массива

Хотя массивы имеют фиксированную длину, вы можете использовать статический метод `Resize` класса `Array` с двумя аргументами (массив и новая длина) для создания нового массива заданной длины. Метод копирует содержимое старого массива в новый массив и задает переменной, полученной в первом аргументе, ссылку на новый массив. Рассмотрим пример:

```
int[] newArray = new int[ 5 ];
Array.Resize( ref newArray, 10 );
```

Переменная `newArray` изначально содержит ссылку на массив из 5 элементов. В результате вызова метода `Resize` переменная `newArray` начинает указывать на новый массив из 10 элементов. Если новый массив меньше старого, то все данные, не поместившиеся в новый массив, усекаются без каких-либо предупреждений.

## 8.4. Примеры использования массивов

В этом разделе приведены примеры, демонстрирующие объявление, создание, инициализацию и обработку элементов массивов.

### 8.4.1. Создание и инициализация массива

Приложение на ил. 8.2 использует ключевое слово `new` для создания массива из пяти элементов `int`, изначально равных 0 (значение по умолчанию для переменных `int`).

```
1 // Fig. 8.2: InitArray.cs
2 // Создание массива.
3 using System;
4
5 public class InitArray
6 {
7     public static void Main( string[] args )
8     {
9         int[] array; // Объявление массива с именем array
```

**Ил. 8.2.** Создание массива (продолжение ➞)

```
10
11      // Выделение памяти для массива и инициализация элементов нулями
12      array = new int[ 5 ]; // 5 элементов типа int
13
14      Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // Заголовки
15
16      // Вывод значений всех элементов
17      for ( int counter = 0; counter < array.Length; ++counter )
18          Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
19  } // Конец Main
20 } // Конец класса InitArray
```

Index	Value
0	0
1	0
2	0
3	0
4	0

**Ил. 8.2.** Создание массива (окончание)

В строке 9 объявляется `array` — переменная, которая может ссылаться на массив элементов типа `int`. Строка 12 создает объект массива с пятью элементами и присваивает ссылку на него переменной `array`. В строке 14 выводятся заголовки столбцов. Первый столбец содержит индексы (0–4) всех элементов массива, а во втором выводятся значения по умолчанию (0) всех элементов (с выравниванием по ширине поля 8).

Команда `for` в строках 17–18 выводит индекс (представленный переменной `counter`) и значение (представленное выражением `array[counter]`) каждого элемента массива. Переменная цикла `counter` изначально равна 0, поскольку значения индексов начинаются с 0. В условии продолжения цикла команды `for` свойство `array.Length` (строка 17) используется для получения длины массива. В этом примере длина массива равна 5, так что цикл будет выполняться до тех пор, пока значение управляющей переменной `counter` остается меньше 5. Максимальное значение индекса в массиве с пятью элементами равно 4, так что оператор `<` в условии гарантирует, что цикл не будет обращаться к элементам за концом массива (то есть при последней итерации цикла значение `counter` равно 4). Вскоре вы увидите, что происходит при выходе индекса за границу массива во время выполнения.

### 8.4.2. Использование инициализатора массива

Приложение может создать массив и инициализировать его элементы при помощи *инициализатора массива* — разделенного запятыми списка выражений (называемого *списком инициализаторов*), заключенного в фигурные скобки. В этом случае длина массива определяется количеством элементов в списке инициализаторов. Например, объявление

```
int[] n = { 10, 20, 30, 40, 50 };
```

создает массив из пяти элементов с индексами 0, 1, 2, 3 и 4. Элемент `n[0]` инициализируется значением 10, элемент `n[1]` — значением 20 и т. д. Эта команда *не требует* ключевого слова `new` при создании объекта массива. Обнаружив список инициализаторов массива, компилятор по количеству инициализаторов в списке определяет размер массива и генерирует нужную операцию `new` «за кулисами». Приложение на ил. 8.3 инициализирует целочисленный массив 10 значениями (строка 10) и выводит массив в табличном формате. Код вывода элементов массива (строки 15–16) идентичен коду на ил. 8.2 (строки 17–18).

```
1 // Ил. 8.3: InitArray.cs
2 // Инициализация элементов массива
3 using System;
4
5 public class InitArray
6 {
7     public static void Main( string[] args )
8     {
9         // В списке инициализаторов задаются значения всех элементов
10        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
11
12        Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // Заголовки
13
14        // Вывод значений элементов массива
15        for ( int counter = 0; counter < array.Length; ++counter )
16            Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
17    } // Конец Main
18 } // Конец класса InitArray
```

Index	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

**Ил. 8.3.** Инициализация элементов массива

### 8.4.3. Вычисление значения, сохраняемого в каждом элементе массива

В некоторых приложениях значение, сохраняемое в каждом элементе массива, вычисляется в ходе выполнения. В приложении на ил. 8.4 создается массив из 10 элементов, каждому элементу которого присваивается одно из четных целых чисел от 2 до 20 (2, 4, 6, ..., 20). Содержимое массива выводится в табличном формате.



Команда `for` в строках 13–14 вычисляет значение элемента массива, умножая текущее значение управляющей переменной цикла `for` на 2 и увеличивая результат на 2.

В строке 9 модификатор `const` используется для объявления константы `ARRAY_LENGTH`, значение которой равно 10. Константы должны инициализироваться при объявлении и в дальнейшем изменяться уже не могут. Мы записываем имена констант прописными буквами, чтобы они выделялись в коде.

```

1 // Ил. 8.4: InitArray.cs
2 // Вычисление значений, сохраняемых в элементе массива.
3 using System;
4
5 public class InitArray
6 {
7     public static void Main( string[] args )
8     {
9         const int ARRAY_LENGTH = 10; // Создание именованной константы
10        int[] array = new int[ ARRAY_LENGTH ]; // Создание массива
11
12        // Вычисление значения каждого элемента массива
13        for ( int counter = 0; counter < array.Length; ++counter )
14            array[ counter ] = 2 + 2 * counter;
15
16        Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // Заголовки
17
18        // Вывод значений элемента массива
19        for ( int counter = 0; counter < array.Length; ++counter )
20            Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
21    } // Конец Main
22 } // Конец класса InitArray

```

Index	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

**Ил. 8.4.** Вычисление значений, помещаемых в элементы массива



#### ТИПИЧНАЯ ОШИБКА 8.2

Попытка присвоить значение именованной константе после ее инициализации приводит к ошибке компиляции.



#### ТИПИЧНАЯ ОШИБКА 8.3

Попытка объявить именованную константу без ее инициализации приводит к ошибке компиляции.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 8.1

Приложения, в которых используются константы, обычно проще читаются, чем приложения с литералами (например, 10). Так, именованная константа `ARRAY_LENGTH` четко обозначает свое предназначение, тогда как литерал может иметь разный смысл в зависимости от контекста его использования. У именованных констант есть и другое преимущество: если значение константы когда-либо потребуется изменить, изменения достаточно внести только в объявлении, что снижает затраты на сопровождение кода.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 8.2

Определение размера массива константой (вместо литерала) делает код более понятным, поскольку из программы исключаются так называемые «волшебные числа». Например, многократное упоминание размера 10 в коде обработки 10-элементного массива придает числу 10 некий искусственный смысл, который может только запутать дело, если программа включает другие литералы 10, не имеющие никакого отношения к размеру массива.

## 8.4.4. Суммирование элементов массива

Часто элементы массива представляют серию значений, используемых в вычислениях. Например, если элементы массива представляют оценки, полученные на экзамене, преподаватель может просуммировать элементы и использовать результат для вычисления средней оценки класса. Данная возможность будет использоваться в примерах GradeBook этой главы (ил. 8.15 и 8.20).

```
1 // Ил. 8.5: SumArray.cs
2 // Суммирование элементов массива.
3 using System;
4
5 public class SumArray
6 {
7     public static void Main( string[] args )
8     {
9         int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10        int total = 0;
11
12        // Значение каждого элемента прибавляется к total
13        for ( int counter = 0; counter < array.Length; ++counter )
14            total += array[ counter ];
15
16        Console.WriteLine( "Total of array elements: {0}", total );
17    } // Конец Main
18 } // Конец класса SumArray
```

Total of array elements: 849

**Ил. 8.5.** Вычисление суммы элементов массива

Приложение на ил. 8.5 суммирует значения, содержащиеся в целочисленном массиве из 10 элементов. Приложение создает и инициализирует массив в строке 9.

Вычисления выполняются в цикле `for`. [*Примечание:* значения, указанные в инициализаторах массива, часто читаются в приложении вместо использования фиксированного списка. Например, приложение может получить данные у пользователя или прочитать их из файла на диске — см. главу 17. Чтение данных делает код более универсальным, поскольку он может использоваться с разными наборами исходных данных.]

### 8.4.5. Использование гистограмм для графического представления данных массива

Многие приложения представляют данные пользователям в графическом формате. Например, числовые значения часто выводятся в виде полос на гистограммах. Более длинные полосы представляют большие числовые значения. В одном из простых способов графического вывода числовых данных каждое числовое значение изображается последовательностью звездочек (\*).

Преподаватель может построить гистограмму количества оценок в нескольких категориях, чтобы наглядно представить распределение оценок. Допустим, на экзамене были получены оценки 87, 68, 94, 100, 83, 78, 85, 91, 76 и 87. Одна оценка равна 100, две оценки лежат в диапазоне 90–99, четыре — в диапазоне 80–89, две — в диапазоне 70–79, одна в диапазоне 60–69 и ни одной оценки ниже 60. В нашем следующем приложении (ил. 8.6) информация о распределении оценок хранится в массиве из 11 элементов, каждый из которых соответствует определенной категории оценок. Например, элемент `array[0]` обозначает количество оценок в диапазоне 0–9, `array[7]` — количество оценок в диапазоне 70–79, а `array[10]` — количество оценок, равных 100. Две версии класса `GradeBook`, приведенные в этой главе (см. ил. 8.15 и 8.20), содержат код, вычисляющий распределение по набору оценок. В нашей первой версии массив создается вручную — мы анализируем набор оценок и инициализируем элементы массива количеством значений в каждом диапазоне (строка 9).

```

1  // Ил. 8.6: BarChart.cs
2  // Гистограмма с данными приложения.
3  using System;
4
5  public class BarChart
6  {
7      public static void Main( string[] args )
8      {
9          int[] array = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 }; // Распределение
10
11         Console.WriteLine( "Grade distribution:" );
12
13         // Для каждого элемента массива выводится полоса на гистограмме
14         for ( int counter = 0; counter < array.Length; ++counter )
15         {
16             // Метки полос ( "00-09: ", ..., "90-99: ", "100: " )

```

**Ил. 8.6.** Построение гистограммы распределения оценок (продолжение ➤)

```

17         if ( counter == 10 )
18             Console.Write( " 100: " );
19         else
20             Console.Write( "{0:D2}-{1:D2}: ",
21                             counter * 10, counter * 10 + 9 );
22
23         // Вывод
24         for ( int stars = 0; stars < array[ counter ]; ++stars )
25             Console.Write( "*" );
26
27         Console.WriteLine(); // Переход на новую строку вывода
28     } // Конец внешнего цикла for
29 } // Конец Main
30 } // Конец класса BarChart

```

```

Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *

```

### Ил. 8.6. Построение гистограммы распределения оценок (окончание)

Приложение читает числа из массива и отображает информацию на гистограмме. Каждый диапазон оценок представлен полосой из звездочек, обозначающих количество оценок в данном диапазоне. В строках 17–21 рядом с каждой полосой выводится метка (например, "70-79: "), основанная на текущем значении счетчика. Когда счетчик достигает 10, в строке 18 выводится метка " 100: " для выравнивания двоеточия с другими строками. Если счетчик отличен от 10, строка 20 использует форматные элементы {0:D2} и {1:D2} для вывода метки. Спецификатор формата D означает, что значение должно форматироваться как целое число, а число после D определяет количество цифр в отформатированном значении. Если оно равно 2, любые значения, содержащие менее двух цифр, выводятся с начальным 0.

Вложенная команда for (строки 24–25) выводит полосы из звездочек. Обратите внимание на условие продолжения цикла в строке 24 (`stars < array[ counter ]`). Каждый раз, когда приложение достигает внутреннего цикла for, цикл проходит от 0 до значения, на 1 меньше `array[ counter ]`; таким образом, значение в массиве определяет количество выводимых звездочек. В нашем примере элементы `array[0]–array[5]` заполнены нулями, потому что ни один студент не получил оценку ниже 60. Из-за этого в строках первых шести диапазонов звездочки не выводятся.

### 8.4.6. Использование элементов массива как счетчиков

Иногда приложения используют переменные-счетчики для обобщения данных (например, результатов опроса). На ил. 7.7 мы используем разные счетчики в приложении, моделирующем броски кубика, для подсчета выпадений каждой грани при 6 000 000 бросках. На ил. 8.7 приведена переработанная версия приложения с ил. 7.7, использующая массивы.

```

1  // Ил. 8.7: RollDie.cs
2  // Моделирование серии из 6 000 000 бросков кубика.
3  using System;
4
5  public class RollDie
6  {
7      public static void Main( string[] args )
8      {
9          Random randomNumbers = new Random(); // Генератор случайных чисел
10         int[] frequency = new int[ 7 ]; // Массив счетчиков
11
12         // Результат каждого из 6 000 000 бросков используется как индекс
13         for ( int roll = 1; roll <= 6000000; ++roll )
14             ++frequency[ randomNumbers.Next( 1, 7 ) ];
15
16         Console.WriteLine( "{0}{1,10}", "Face", "Frequency" );
17
18         // Вывод значения каждого элемента массива
19         for ( int face = 1; face < frequency.Length; ++face )
20             Console.WriteLine( "{0,4}{1,10}", face, frequency[ face ] );
21     } // Конец Main
22 } // Конец класса RollDie

```

Face	Frequency
1	999924
2	1000939
3	1001249
4	998454
5	1000233
6	999201

**Ил. 8.7.** Моделирование серии из 6 000 000 бросков кубика

Приложение использует массив `frequency` (строка 10) для подсчета выпадений каждой из граней кубика. Команда в строке 14 заменяет строки 26–46 на ил. 7.7. Строка 14 использует случайное значение для выбора увеличиваемого элемента `frequency` при каждой итерации цикла. Выражение в строке 14 генерирует случайные числа от 1 до 6, так что размер массива `frequency` должен быть достаточен для хранения шести счетчиков. Мы используем массив из семи элементов, в котором элемент `frequency[0]` игнорируется — для грани 1 увеличение счетчика `frequency[1]` выглядит логичнее, чем увеличение `frequency[0]`. Таким образом, значение каждой грани используется как индекс в массиве `frequency`. Кроме того, строки 50–52 на ил. 7.7 заменяются перебором элементов массива `frequency` для вывода результатов (см. ил. 8.7, строки 19–20).

### 8.4.7. Основы обработки исключений

В нашем следующем примере массивы будут использоваться для обобщения данных, собранных в ходе опроса. Задача сформулирована следующим образом:

20 студентам предложено оценить качество еды в студенческом кафетерии по шкале от 1 до 5 (1 — «ужасно», 5 — «превосходно»). Сохраните 20 ответов в целочисленном массиве и вычислите распределение частот оценок.

Это типичная задача по обработке массивов (ил. 8.8). Мы хотим подсчитать ответы каждого типа (то есть 1–5). Целочисленный массив `responses` (строки 10–11) с 20 элементами содержит результаты опроса. В последнем элементе намеренно сохранен недопустимый ответ (14). В ходе выполнения программы C# индексы элементов проверяются на действительность — все индексы должны быть больше либо равны 0 и меньше длины массива. Любая попытка обращения к элементу за пределами диапазона индексов вызывает ошибку времени выполнения `IndexOutOfRangeException`. В конце этого раздела мы рассмотрим некорректный ответ, продемонстрируем проверку границ и рассмотрим механизм обработки исключений C#, который может использоваться для обнаружения и обработки `IndexOutOfRangeException`.

```
1  // Ил. 8.8: StudentPoll.cs
2  // Анализ результатов опроса.
3  using System;
4
5  public class StudentPoll
6  {
7      public static void Main( string[] args )
8      {
9          // Массив ответов (чаще данные вводятся во время выполнения)
10         int[] responses = { 1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
11                             2, 3, 3, 2, };
12         int[] frequency = new int[ 6 ]; // Массив счетчиков частот
13
14         // Для каждого ответа выбрать элемент responses и использовать его
15         // как индекс для определения увеличиваемого элемента
16         for ( int answer = 0; answer < responses.Length; ++answer )
17         {
18             try
19             {
20                 ++frequency[ responses[ answer ] ];
21             } // end try
22             catch ( IndexOutOfRangeException ex )
23             {
24                 Console.WriteLine( ex.Message );
25                 Console.WriteLine( " responses[{0}] = {1}\n",
26                                     answer, responses[ answer ] );
27             } // Конец catch
28         } // Конец for
29
30         Console.WriteLine( "{0}{1,10}", "Rating", "Frequency" );
31
```

**Ил. 8.8.** Приложение для анализа результатов опроса (продолжение ➤)

```

32      // Вывод значения каждого элемента массива
33      for ( int rating = 1; rating < frequency.Length; ++rating )
34          Console.WriteLine( "{0,6}{1,10}", rating, frequency[ rating ] );
35  } // Конец Main
36 } // Конец класса StudentPoll

```

```

Index was outside the bounds of the array.
responses(19) = 14

```

Rating	Frequency
1	3
2	4
3	8
4	2
5	2

### Ил. 8.8. Приложение для анализа результатов опроса (окончание)

#### Массив frequency

Массив `frequency` (строка 12) используется для подсчета каждого из вариантов ответа. Каждый элемент массива используется как счетчик для одного из вариантов — в `frequency[1]` подсчитывается количество студентов, оценивших качество еды на 1, в `frequency[2]` подсчитывается количество студентов с оценкой 2 и т. д.

#### Обобщение результатов

Команда `for` (строки 16–28) последовательно читает ответы из массива `responses` и увеличивает элементы от `frequency[1]` до `frequency[5]`; значение `frequency[0]` игнорируется, потому что ответы в опросе ограничены диапазоном 1–5. Ключевая команда цикла находится в строке 20. Команда увеличивает счетчик, определяемый значением `responses[answer]`.

Рассмотрим несколько начальных итераций команды `for`.

- ❑ Когда счетчик `answer` равен 0, `responses[answer]` соответствует значению `responses[0]` (то есть 1 — см. строку 10). В этом случае `frequency[responses[answer]]` интерпретируется как `frequency[1]` и счетчик `frequency[1]` увеличивается на 1. Вычисление выражения начинается со значения во внутренней паре скобок (`answer`, в данный момент 0). Значение `answer` подставляется в выражение, после чего вычисляется содержимое следующей пары скобок (`responses[answer]`). Полученное значение используется как индекс массива `frequency` для определения увеличиваемого счетчика (в данном случае `frequency[1]`).
- ❑ При следующей итерации счетчик `answer` равен 1, а выражение `responses[answer]` соответствует значению `responses[1]` (то есть 2 — см. строку 10). Выражение `frequency[responses[answer]]` интерпретируется как `frequency[2]`, и счетчик `frequency[2]` увеличивается на 1.
- ❑ Когда счетчик `answer` равен 2, `responses[answer]` соответствует значению `responses[2]` (то есть 5 — см. строку 10). В этом случае `frequency[responses[answer]]` интерпретируется как `frequency[5]`, счетчик `frequency[5]` увеличивается на 1 и т. д.

Независимо от количества ответов, обработанных в ходе опроса, для обобщения результатов достаточно массива из шести элементов (из которых нулевой элемент игнорируется), так как все правильные ответы лежат в диапазоне 1–5, а индексы в массиве с шестью элементами лежат в диапазоне 0–5. На ил. 8.8 в столбце `frequency` приведена сводка только по 19 из 20 значений массива `responses` — последний элемент содержит недействительный ответ, который не был учтен при подсчете.

### Обработка исключений: недействительный ответ

*Исключение* (exception) свидетельствует о проблеме, возникшей в ходе выполнения программы. Само название предполагает, что проблема возникает нечасто, — если нормальное выполнение команды считается «правилом», то проблема составляет «исключение из правила».

Обработка исключений позволяет повысить надежность программ. Во многих случаях программа после обработки исключения продолжает выполняться так, словно никакой проблемы не было. Например, приложение на ил. 8.8 все равно выводит результаты, хотя один из ответов был недействителен. Впрочем, более серьезная проблема может сделать продолжение невозможным; в этом случае программа сообщает пользователю о проблеме и завершается. При обнаружении проблемы (например, недействительного индекса массива или недействительного аргумента метода) исполнительная среда или метод выдает исключение.

### Команда `try`

Чтобы обработать исключение, следует поместить любой код, в котором оно может возникнуть, в команду `try` (строки 18–27). Блок `try` (строки 18–21) содержит код, в котором может возникнуть исключение, а блок `catch` (строки 22–27) содержит код обработки исключения, если оно возникнет. Вы можете определить несколько блоков `catch` для перехвата разных типов исключений, которые могут быть иницированы в соответствующем блоке `try`. Если в строке 20 происходит нормальное увеличение элемента массива `frequency`, строки 22–27 игнорируются. Фигурные скобки, в которые заключаются тела блоков `try` и `catch`, обязательны.

### Выполнение блока `catch`

Когда программа доходит до значения 14 в массиве `responses`, она пытается прибавить 1 к несуществующему элементу `frequency[14]` — массив `frequency` состоит лишь из шести элементов. Так как проверка границ массивов выполняется во время выполнения, исполнительная среда Common Language Runtime генерирует исключение, оповещающее программу о возникшей проблеме (в строке 20 иницируется исключение `IndexOutOfRangeException`). В этот момент блок `try` завершается, а блок `catch` начинает выполняться; если в блоке `try` были объявлены какие-либо переменные, они недоступны в блоке `catch` из-за выхода из области действия.

В блоке `catch` объявляется тип (`IndexOutOfRangeException`) и параметр исключения (`ex`). Блок `catch` обрабатывает исключения заданного типа и в нем можно обращаться к перехваченному объекту исключения через идентификатор.





### КАК ИЗБЕЖАТЬ ОШИБОК 8.1

Если ваш код обращается к элементу массива, проверьте, что индекс больше либо равен 0 и меньше либо равен длине массива. Такая проверка поможет избежать исключений `IndexOutOfRangeException` в ваших программах.

#### Свойство `Message` параметра исключения

При перехвате исключения в строках 22–27 программа выводит сообщение о возникшей проблеме. В строке 24 свойство `Message` объекта исключения используется для вывода сообщения об ошибке, хранящегося в объекте исключения. После вывода сообщения в данном примере исключение считается обработанным, а выполнение программы продолжается с команды, следующей после закрывающей фигурной скобки блока `catch`. Так как в нашем примере достигается конец команды `for` (строка 28), выполнение продолжается с увеличения управляющей переменной в строке 16. Тема обработки исключений более глубоко рассматривается в главах 10 и 13.

## 8.5. Пример: моделирование тасования колоды и сдачи карт

До настоящего момента в примерах этой главы массивы использовались для элементов значимых типов. В этом разделе генератор случайных чисел и массив с элементами ссылочного типа (объектами, представляющими игральные карты) используются для разработки класса, моделирующего тасование и сдачу карт.

Мы начнем с разработки класса `Card` (ил. 8.9), представляющего игровую карту с номиналом ("Ace", "Deuce", "Three", ..., "Jack", "Queen", "King") и мастью ("Hearts", "Diamonds", "Clubs", "Spades"). Затем будет создан класс `DeckOfCards` (ил. 8.10), который создает массив из 52 объектов `Card`. В завершение возможности класса `DeckOfCards` по тасованию и сдаче карт будут использованы для построения приложения (ил. 8.11).

#### Класс `Card`

Класс `Card` (см. ил. 8.9) содержит две строковые переменные экземпляров, `face` и `suit`, в которых хранятся ссылки на номинал и масть конкретной карты. Конструктор класса (строки 9–13) получает две строки, которые используются для инициализации `face` и `suit`. Метод `ToString` (строки 16–19) создает строку, состоящую из номинала карты, связки "of" и масти. Вспомните, о чем говорилось в главе 7: оператор `+` может использоваться для конкатенации (то есть объединения) нескольких строк в одну большую строку. Метод `ToString` класса `Card` вызывается для получения строкового представления объекта `Card` (например, "Ace of Spades"). Метод `ToString` объекта вызывается неявно во многих случаях при использовании объекта там, где ожидается строка (например, при выводе объекта вызовом `WriteLine` или конкатенации объекта в строку оператором `+`). Для этого заголовок объявления

ToString должен выглядеть точно так, как показано в строке 16 на ил. 8.9. Смысл ключевого слова `override` будет более подробно рассмотрен при описании наследования в главе 11.

```
1 // Ил. 8.9: Card.cs
2 // Класс Card представляет игральную карту.
3 public class Card
4 {
5     private string face; // Номинал карты ("Ace", "Deuce", ...)
6     private string suit; // Масть ("Hearts", "Diamonds", ...)
7
8     // Конструктор Card инициализирует номинал и масть карты
9     public Card( string cardFace, string cardSuit )
10    {
11        face = cardFace; // initialize face of card
12        suit = cardSuit; // initialize suit of card
13    } // Конец конструктора Card с двумя параметрами
14
15    // Метод возвращает строковое представление Card
16    public override string ToString()
17    {
18        return face + " of " + suit;
19    } // Конец метода ToString
20 } // Конец класса Card
```

**Ил. 8.9.** Класс Card представляет игральную карту

### Класс DeckOfCards

Класс `DeckOfCards` (см. ил. 8.10) объявляет переменную экземпляра с именем `deck`, ссылающуюся на массив объектов `Card` (строка 7). Объявление переменной для массива объектов (например, `Card[] deck`), как и объявление переменной для массива простых типов, состоит из типа элементов, квадратных скобок и имени переменной массива. В классе `DeckOfCards` также объявляется переменная экземпляра `currentCard` типа `int` (строка 8), которая представляет следующую карту из массива `deck`, и именованная константа `NUMBER_OF_CARDS` (строка 9), определяющая количество объектов `Card` в колоде (52).

```
1 // Ил. 8.10: DeckOfCards.cs
2 // Класс DeckOfCards представляет колоду игровых карт.
3 using System;
4
5 public class DeckOfCards
6 {
7     private Card[] deck; // Массив объектов Card
8     private int currentCard; // Индекс следующей карты (0-51)
9     private const int NUMBER_OF_CARDS = 52; // Количество объектов Card
10    private Random randomNumbers; // Генератор случайных чисел
11
12    // Конструктор заполняет массив элементов Card
13    public DeckOfCards()
14    {
```

**Ил. 8.10.** Класс `DeckOfCards` представляет колоду карт (продолжение ↗)

```

15     string[] faces = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
16         "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
17     string[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
18
19     deck = new Card[ NUMBER_OF_CARDS ]; // Создание массива
20     currentCard = 0; // Первой сдается карта deck[ 0 ]
21     randomNumbers = new Random(); // Создание генератора
22                                     // случайных чисел
23     // Заполнение колоды объектами Card
24     for ( int count = 0; count < deck.Length; ++count )
25         deck[ count ] =
26             new Card( faces[ count % 13 ], suits[ count / 13 ] );
27 } // Конец конструктора DeckOfCards
28
29 // Тасование колоды карт по однопроходному алгоритму
30 public void Shuffle()
31 {
32     // После тасования сдача снова начинается с deck[ 0 ]
33     currentCard = 0; // Повторная инициализация currentCard
34
35     // Для каждой карты выбрать случайную карту и поменять их местами
36     for ( int first = 0; first < deck.Length; ++first )
37     {
38         // Выбор случайного числа от 0 до 51
39         int second = randomNumbers.Next( NUMBER_OF_CARDS );
40
41         // Текущая карта меняется местами со случайной картой
42         Card temp = deck[ first ];
43         deck[ first ] = deck[ second ];
44         deck[ second ] = temp;
45     } // Конец for
46 } // Конец метода Shuffle
47
48 // Сдача одной карты
49 public Card DealCard()
50 {
51     // Проверить, остались ли карты в колоде
52     if ( currentCard < deck.Length )
53         return deck[ currentCard++ ]; // Вернуть следующую карту
54     else
55         return null; // Сданы все карты
56 } // Конец метода DealCard
57 } // Конец класса DeckOfCards

```

**Ил. 8.10.** Класс DeckOfCards представляет колоду карт (окончание)

### Класс DeckOfCards: конструктор

Конструктор класса создает экземпляр массива `deck` (строка 19) с размером `NUMBER_OF_CARDS`. При создании элементы массива `deck` по умолчанию инициализируются значением `null`, поэтому конструктор использует цикл `for` (строки 24–26) для заполнения массива `deck` объектами `Card`. Цикл `for` инициализирует управляющую переменную `count` значением 0 и выполняется до тех пор, пока значение

`count` остается меньше `deck.Length`; в результате `count` последовательно принимает целочисленные значения от 0 до 1 (индексы массива `deck`). Каждый объект `Card` создается и инициализируется двумя строками — одна берется из массива `faces` (содержащего строки от "Ace" до "King"), а другая из массива `suits` (содержащего названия мастей). Выражение `count % 13` всегда дает значение от 0 до 12 (13 индексов массива `faces` в строках 15–16), а выражение `count / 13` всегда возвращает значение от 0 до 3 (четыре индекса массива `suits` в строке 17). Инициализированный массив `deck` содержит объекты `Card` с номиналами от "Ace" до "King" в каждой масти.

### Класс `DeckOfCards`: метод `Shuffle`

Метод `Shuffle` (строки 30–46) переставляет объекты `Card` в массиве, моделируя тасование колоды. Метод перебирает все 52 объекта `Card` с индексами от 0 до 51. Для каждого объекта `Card` случайным образом выбирается число от 0 до 51, и текущий объект `Card` меняется местами со случайно выбранным объектом. Этот обмен выполняется тремя командами присваивания в строках 42–44. Дополнительная переменная `temp` используется для временного хранения одного из двух объектов `Card`. Перестановку не удастся выполнить всего двумя командами:

```
deck[ first ] = deck[ second ];  
deck[ second ] = deck[ first ];
```

Если `deck[ first ]` содержит карту с номиналом "Ace" и мастью "Spades", а `deck[ second ]` содержит карту с номиналом "Queen" и мастью "Hearts", то после первого присваивания оба элемента будут содержать комбинацию "Queen"/"Hearts", а комбинация "Ace"/"Spades" будет потеряна — следовательно, нужна временная переменная для ее хранения. После завершения цикла `for` объекты `Card` размещаются в случайном порядке. За один проход по массиву выполняются всего 52 перестановки, а объекты `Card` размещаются в случайном порядке.

### Рекомендация: использование несмещенного алгоритма тасования

Для моделирования реальных карточных игр рекомендуется использовать так называемый несмещенный алгоритм тасования. Такие алгоритмы гарантируют, что все возможные комбинации перетасованных последовательностей карт встречаются с равной вероятностью — как, например, популярный алгоритм Фишера–Йетса<sup>1</sup>.

### Класс `DeckOfCards`: метод `DealCard`

Метод `DealCard` (строки 49–56) сдает один объект `Card` из массива. Вспомните, что переменная `currentCard` обозначает индекс следующей сдаваемой карты (то есть определяет карту на вершине колоды). Строка 52 сравнивает `currentCard` с длиной массива `deck`. Если колода не пуста (то есть `currentCard` меньше 52), то строка 53 возвращает верхний объект `Card` и увеличивает `currentCard` для подготовки к следующему вызову `DealCard`; в противном случае возвращается `null`.

<sup>1</sup> [ru.wikipedia.org/wiki/Тасование\\_Фишера–Йетса](http://ru.wikipedia.org/wiki/Тасование_Фишера–Йетса). — *Примеч. перев.*

### Тасование и сдача карт

Приложение на ил. 8.11 демонстрирует функциональность класса `DeckOfCards` (см. ил. 8.10) по тасованию и сдаче карт. Строка 10 создает объект `DeckOfCards` с именем `myDeckOfCards`. Вспомните, что конструктор `DeckOfCards` создает колоду из 52 объектов `Card`, упорядоченную по мастям и номиналам. В строке 11 метод `Shuffle` объекта `myDeckOfCards` переставляет объекты `Card` в массиве.

Цикл `for` в строках 14–20 сдает все 52 карты в колоде и выводит их в 4 столбца по 13 объектов `Card` в каждом. Строка 16 сдает карту и выводит объект `Card`; при этом вызывается метод `DealCard` объекта `myDeckOfCards`. Когда `Console.Write` выводит объект `Card` в формате `string`, неявно вызывается метод `ToString` класса `Card` (объявляемый в строках 16–19 на ил. 8.9). Так как ширина поля задана отрицательной величиной, результат выводится с выравниванием по левому краю поля ширины 19.

```

1  // Ил. 8.11: DeckOfCardsTest.cs
2  // Приложение для тасования и сдачи карт.
3  using System;
4
5  public class DeckOfCardsTest
6  {
7      // Выполнение приложения
8      public static void Main( string[] args )
9      {
10         DeckOfCards myDeckOfCards = new DeckOfCards();
11         myDeckOfCards.Shuffle(); // Случайная перестановка объектов Card
12
13         // Вывод всех 52 карт в порядке сдачи
14         for ( int i = 0; i < 52; ++i )
15         {
16             Console.Write( "{0,-19}", myDeckOfCards.DealCard() );
17
18             if ( ( i + 1 ) % 4 == 0 )
19                 Console.WriteLine();
20         } // Конец for
21     } // Конец Main
22 } // Конец класса DeckOfCardsTest

```

Eight of Clubs	Ten of Clubs	Ten of Spades	Four of Spades
Ace of Spades	Jack of Spades	Three of Spades	Seven of Spades
Three of Diamonds	Five of Clubs	Eight of Spades	Five of Hearts
Ace of Hearts	Ten of Hearts	Deuce of Hearts	Deuce of Clubs
Jack of Hearts	Nine of Spades	Four of Hearts	Seven of Clubs
Queen of Spades	Seven of Diamonds	Five of Diamonds	Ace of Clubs
Four of Clubs	Ten of Diamonds	Jack of Clubs	Six of Diamonds
Eight of Diamonds	King of Hearts	Three of Clubs	King of Spades
King of Diamonds	Six of Spades	Deuce of Spades	Five of Spades
Queen of Clubs	King of Clubs	Queen of Hearts	Seven of Hearts
Ace of Diamonds	Deuce of Diamonds	Four of Diamonds	Nine of Clubs
Queen of Diamonds	Jack of Diamonds	Six of Hearts	Nine of Diamonds
Nine of Hearts	Three of Hearts	Six of Clubs	Eight of Hearts

**Ил. 8.11.** Приложение для тасования и сдачи карт

## 8.6. Команда foreach

В предыдущих примерах было продемонстрировано использование цикла `for` со счетчиком для перебора элементов массива. В этом разделе будет представлен цикл `foreach`, который перебирает элементы всего массива (об использовании цикла `foreach` для перебора коллекции будет рассказано в главе 21). Синтаксис команды `foreach` выглядит так:

```
foreach ( тип идентификатор in имя_массива )
    команда
```

где *тип* и *идентификатор* — соответственно тип и имя управляющей переменной, а *имя\_массива* — массив, содержимое которого перебирается командой. Тип управляющей переменной должен соответствовать типу элементов массива. Как показывает следующий пример, управляющая переменная последовательно представляет значения элементов массива при последовательных итерациях команды `foreach`.

Команда `foreach` (строки 13–14 на ил. 8.12) вычисляет сумму целых чисел в массиве оценок. Управляющая переменная объявлена с типом `int`, потому что массив содержит значения `int`, — таким образом, при каждой итерации цикла из массива выбирается одно значение типа `int`. Команда `foreach` последовательно перебирает значения, хранящиеся в массиве. Заголовок `foreach` можно прочесть в следующем виде: «для каждой итерации присвоить следующий элемент массива переменной `number` типа `int`, а затем выполнить следующую команду». Таким образом, для каждой итерации идентификатор `number` представляет очередное значение `int` из массива. Строки 13–14 эквивалентны следующему циклу со счетчиком из ил. 8.5, суммирующему целые числа в массиве:

```
for ( int counter = 0; counter < array.Length; ++counter )
    total += array[ counter ];
```



### ТИПИЧНАЯ ОШИБКА 8.4

Управляющая переменная цикла `foreach` может использоваться только для обращения к элементам массива. Любая попытка изменить значение управляющей переменной в теле команды `foreach` приведет к ошибке компиляции.

```
1 // Ил. 8.12: ForEachTest.cs
2 // Использование команды foreach для суммирования чисел в массиве.
3 using System;
4
5 public class ForEachTest
6 {
7     public static void Main( string[] args )
8     {
9         int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10        int total = 0;
11
12        // Значение каждого элемента прибавляется к total
```

**Ил. 8.12.** Использование команды `foreach` для суммирования чисел в массиве (продолжение ↗)

```
13     foreach ( int number in array )
14         total += number;
15
16     Console.WriteLine( "Total of array elements: {0}", total );
17 } // Конец Main
18 } // Конец класса ForEachTest
```

```
Total of array elements: 849
```

**Ил. 8.12.** Использование команды `foreach` для суммирования чисел в массиве (окончание)

Команда `foreach` может использоваться вместо команды `for` в любой ситуации, в которой коду перебора не нужен доступ к счетчику, обозначающему индекс текущего элемента массива. Например, при суммировании чисел в массиве требуется доступ только к значениям элементов — индекс элемента несущественен. Но если приложение должно использовать счетчик для каких-то целей, кроме перебора элементов (например, для вывода индекса рядом с каждым значением элемента, как в предшествующих примерах этой главы), используйте команду `for`.

## 8.7. Передача массивов и элементов массивов методам

Чтобы передать методу аргумент-массив, укажите имя массива без квадратных скобок. Допустим, массив `hourlyTemperatures` объявляется в следующем виде:

```
double[] hourlyTemperatures = new double[ 24 ];
```

Вызов метода

```
ModifyArray( hourlyTemperatures );
```

передает ссылку на массив `hourlyTemperatures` методу `ModifyArray`. Каждый объект массива «знает» свою длину (и предоставляет доступ к ней в свойстве `Length`). Таким образом, при передаче методу ссылки на объект массива не нужно передавать длину массива в отдельном аргументе.

### Определение параметра-массива

Чтобы метод получал ссылку на массив при вызове метода, в списке параметров метода должен быть указан параметр-массив. Например, заголовок метода `ModifyArray` должен быть записан в виде

```
void ModifyArray( double[] b )
```

Он показывает, что `ModifyArray` получает в параметре `b` ссылку на массив с элементами `double`. Вызов метода передает ссылку на массив `hourlyTemperature`, так что при использовании переменной `b` в вызванном методе она ссылается на тот же объект массива, что и переменная `hourlyTemperatures` в вызывающем методе.

### Передача по значению и передача по ссылке

Если аргумент метода представляет собой целый массив или отдельный элемент массива ссылочного типа, вызываемый метод получает копию ссылки. Но если аргумент представляет собой отдельный элемент массива значимого типа, вызываемому методу передается копия значения элемента. Чтобы передать методу отдельный элемент массива, укажите в качестве аргумента имя массива с индексом. Если вы хотите передать элемент массива значимого типа по ссылке, необходимо использовать ключевое слово `ref`, как показано в разделе 7.16.

На ил. 8.13 продемонстрированы различия между передачей всего массива и передачей методу элемента значимого типа. Команда `foreach` в строках 17–18 выводит пять элементов `array` (массив значений `int`). В строке 20 вызывается метод `ModifyArray`, которому в аргументе передается `array`. Метод `ModifyArray` (строки 37–41) получает *копию ссылки* на `array` и использует ее для умножения каждого элемента массива на 2. Чтобы пользователь мог убедиться в том, что элементы `array` (из `Main`) изменились, цикл `foreach` в строках 24–25 снова выводит пять элементов `array`. Как видно из выходных данных, метод `ModifyArray` действительно удвоил значение каждого элемента.

```
1 // Ил. 8.13: PassArray.cs
2 // Передача методам массивов и отдельных элементов.
3 using System;
4
5 public class PassArray
6 {
7     // Main создает массив и вызывает методы ModifyArray и ModifyElement
8     public static void Main( string[] args )
9     {
10         int[] array = { 1, 2, 3, 4, 5 };
11
12         Console.WriteLine(
13             "Effects of passing reference to entire array:\n" +
14             "The values of the original array are:" );
15
16         // Вывод исходных элементов массива
17         foreach ( int value in array )
18             Console.Write( " {0}", value );
19
20         ModifyArray( array ); // Передача ссылки на массив
21         Console.WriteLine( "\n\nThe values of the modified array are:" );
22
23         // Вывод измененных элементов массива
24         foreach ( int value in array )
25             Console.Write( " {0}", value );
26
27         Console.WriteLine(
28             "\n\nEffects of passing array element value:\n" +
29             "array[3] before ModifyElement: {0}", array[ 3 ] );
30
31         ModifyElement( array[ 3 ] ); // Попытка изменения array[ 3 ]
```

**Ил. 8.13.** Передача методам массивов и элементов (продолжение ↗)



```
32     Console.WriteLine(  
33         "array[3] after ModifyElement: {0}", array[ 3 ] );  
34 } // Конец Main  
35  
36 // Умножение каждого элемента массива на 2  
37 public static void ModifyArray( int[] array2 )  
38 {  
39     for ( int counter = 0; counter < array2.Length; ++counter )  
40         array2[ counter ] *= 2;  
41 } // Конец метода ModifyArray  
42  
43 // Умножение аргумента на 2  
44 public static void ModifyElement( int element )  
45 {  
46     element *= 2;  
47     Console.WriteLine(  
48         "Value of element in ModifyElement: {0}", element );  
49 } // Конец метода ModifyElement  
50 } // Конец класса PassArray
```

Effects of passing reference to entire array:

The values of the original array are:

1    2    3    4    5

The values of the modified array are:

2    4    6    8    10

Effects of passing array element value:

array[3] before ModifyElement: 8

Value of element in ModifyElement: 16

array[3] after ModifyElement: 8

### Ил. 8.13. Передача методам массивов и элементов (окончание)

Из листинга на ил. 8.13 видно, что при передаче методу копии отдельного элемента значимого типа изменение этой копии в вызванном методе *не отражается* на исходном значении элемента в массиве вызывающего метода. В строках 27–29 выводится значение `array[3]` перед вызовом метода `ModifyElement`, равное 8. В строке 31 вызывается метод `ModifyElement` с передачей аргумента `array[3]`. Помните, что `array[3]` в действительности представляет собой одно значение `int` (8) из массива. Таким образом, в приложении передается копия значения `array[3]`. Метод `ModifyElement` (строки 44–49) умножает значение, полученное в аргументе, на 2, сохраняет результат в своем параметре `element`, после чего выводит значение `element` (16). Так как параметры методов, как и локальные переменные, перестают существовать после завершения метода, в котором они были объявлены, параметр `element` уничтожается при завершении метода `ModifyElement`. Когда приложение возвращает управление в `Main`, строки 32–33 выводят неизменное значение `array[3]` (то есть 8).

## 8.8. Передача массивов по значению и по ссылке

В C# переменная, в которой хранится объект (например, массив), не содержит данные объекта. Вместо этого в переменной хранится ссылка на объект. Различия

между переменными ссылочных и значимых типов создают некоторые нюансы, в которых необходимо разбираться для создания надежных, стабильных программ.

Как известно, при передаче аргумента методу вызываемый метод получает копию значения аргумента. Изменения в локальной копии вызванного метода не отражаются на исходной переменной на стороне вызова. Если аргумент относится к ссылочному типу, то метод копирует *ссылку*, а не сам объект, на который эта ссылка указывает. Локальная копия ссылки также указывает на исходный объект, поэтому изменения объекта в вызванном методе отразятся на исходном объекте.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 8.1

Ссылки на массивы и другие объекты передаются по соображениям производительности. Если бы массивы передавались по значению, то приходилось бы передавать копии всех элементов. Для больших, часто передаваемых массивов это привело бы к лишним затратам времени и памяти для хранения копий массивов.

В разделе 7.16 вы узнали, что C# позволяет передавать переменные по ссылке с использованием ключевого слова `ref`. Ключевое слово `ref` также может использоваться для передачи переменной ссылочного типа по ссылке, чтобы вызванный метод мог изменить исходную переменную и связать ее с другим объектом. Это достаточно нетривиальная возможность, которая при неправильном использовании может создать проблемы. Например, при передаче объекта ссылочного типа (такого, как массив) с ключевым словом `ref` вызванный метод берет под свой контроль *саму ссылку*, что позволяет ему заменить исходную ссылку ссылкой на другой объект (или даже `null`). Такое поведение может привести к непредсказуемым последствиям, которые могут обернуться катастрофой в критичных приложениях. Приложение на ил. 8.14 демонстрирует тонкие различия между передачей ссылки по значению и передачей ссылки по ссылке с ключевым словом `ref`.

```
1  // Ил. 8.14: ArrayReferenceTest.cs
2  // Тестирование передачи ссылок на массивы
3  // по значению и по ссылке.
4  using System;
5
6  public class ArrayReferenceTest
7  {
8      public static void Main( string[] args )
9      {
10         // Создание и инициализация firstArray
11         int[] firstArray = { 1, 2, 3 };
12
13         // Копирование ссылки в переменную firstArray
14         int[] firstArrayCopy = firstArray;
15
16         Console.WriteLine(
17             "Test passing firstArray reference by value" );
18
19         Console.Write( "\nContents of firstArray " +
```

**Ил. 8.14.** Передача ссылки на массив по ссылке и по значению (продолжение 3)

```
20         "before calling FirstDouble:\n\t" );
21
22     // Вывод содержимого firstArray
23     for ( int i = 0; i < firstArray.Length; ++i )
24         Console.Write( "{0} ", firstArray[ i ] );
25
26     // Передача переменной firstArray по значению
27     FirstDouble( firstArray );
28
29     Console.Write( "\n\nContents of firstArray after " +
30         "calling FirstDouble\n\t" );
31
32     // Вывод содержимого firstArray
33     for ( int i = 0; i < firstArray.Length; ++i )
34         Console.Write( "{0} ", firstArray[ i ] );
35
36     // Проверяем, была ли ссылка изменена в FirstDouble
37     if ( firstArray == firstArrayCopy )
38         Console.WriteLine(
39             "\n\nThe references refer to the same array" );
40     else
41         Console.WriteLine(
42             "\n\nThe references refer to different arrays" );
43
44     // Создание и инициализация secondArray
45     int[] secondArray = { 1, 2, 3 };
46
47     // Копирование ссылки в переменную secondArray
48     int[] secondArrayCopy = secondArray;
49
50     Console.WriteLine( "\nTest passing secondArray " +
51         "reference by reference" );
52
53     Console.Write( "\n\nContents of secondArray " +
54         "before calling SecondDouble:\n\t" );
55
56     // Вывод содержимого secondArray перед вызовом метода
57     for ( int i = 0; i < secondArray.Length; ++i )
58         Console.Write( "{0} ", secondArray[ i ] );
59
60     // Передача переменной secondArray по значению
61     SecondDouble( ref secondArray );
62
63     Console.Write( "\n\nContents of secondArray " +
64         "after calling SecondDouble:\n\t" );
65
66     // Вывод содержимого secondArray после вызова метода
67     for ( int i = 0; i < secondArray.Length; ++i )
68         Console.Write( "{0} ", secondArray[ i ] );
69
70     // Проверяем, была ли ссылка изменена в SecondDouble
71     if ( secondArray == secondArrayCopy )
72         Console.WriteLine(
73             "\n\nThe references refer to the same array" );
74     else
```

**Ил. 8.14.** Передача ссылки на массив по ссылке и по значению (продолжение ↗)

```

75         Console.WriteLine(
76             "\n\nThe references refer to different arrays" );
77     } // Конец Main
78
79     // Изменение элементов массива и попытка изменения ссылки
80     public static void FirstDouble( int[] array )
81     {
82         // Значение каждого элемента удваивается
83         for ( int i = 0; i < array.Length; ++i )
84             array[ i ] *= 2;
85
86         // Создание нового объекта и присваивание array ссылки на него
87         array = new int[] { 11, 12, 13 };
88     } // end method FirstDouble
89
90     // Изменение элементов массива и изменение ссылки array
91     // с переводом ее на новый массив
92     public static void SecondDouble( ref int[] array )
93     {
94         // Значение каждого элемента удваивается
95         for ( int i = 0; i < array.Length; ++i )
96             array[ i ] *= 2;
97
98         // Создание нового объекта и присваивание array ссылки на него
99         array = new int[] { 11, 12, 13 };
100     } // Конец метода SecondDouble
101 } // Конец класса ArrayReferenceTest

```

Test passing firstArray reference by value

Contents of firstArray before calling FirstDouble:

1 2 3

Contents of firstArray after calling FirstDouble

2 4 6

The references refer to the same array

Test passing secondArray reference by reference

Contents of secondArray before calling SecondDouble:

1 2 3

Contents of secondArray after calling SecondDouble:

11 12 13

The references refer to different arrays

#### Ил. 8.14. Передача ссылки на массив по ссылке и по значению (окончание)

В строках 11 и 14 объявляются переменные для двух целочисленных массивов, `firstArray` и `firstArrayCopy`. Строка 11 инициализирует `firstArray` значениями 1, 2 и 3. Команда присваивания в строке 14 копирует ссылку, хранящуюся в `firstArray`, в переменную `firstArrayCopy`, в результате чего эти переменные указывают на один объект массива. Мы копируем ссылку, чтобы позже определить, изменилась

ли ссылка `firstArray`. Цикл `for` в строках 23–24 выводит содержимое `firstArray`, прежде чем передавать его методу `FirstDouble` (строка 27); это позволяет нам убедиться в том, что вызванный метод действительно изменяет содержимое массива.

### Метод `FirstDouble`

Команда `for` в методе `FirstDouble` (строки 83–84) умножает значения всех элементов в массиве на 2. Строка 87 создает новый массив, содержащий значения 11, 12 и 13, и присваивает ссылку на него параметру `array` для попытки перезаписи ссылки `firstArray` на стороне вызова — чего, естественно, не происходит, потому что ссылка передавалась по значению. После выполнения метода `FirstDouble` команда `for` в строках 33–34 выводит содержимое `firstArray`, демонстрируя, что значения элементов были изменены методом. Команда `if...else` в строках 37–42 использует оператор `==` для проверки равенства ссылок `firstArray` (которую мы только что попытались заменить) и `firstArrayCopy`. Выражение в строке 37 дает результат `true`, если операнды оператора `==` ссылаются на один объект. В данном случае объект, представленный ссылкой `firstArray`, является массивом, созданным в строке 11, а не массивом, созданным в методе `FirstDouble` (строка 87), так что ссылка, хранимая в `firstArray`, осталась неизменной.

### Метод `SecondDouble`

В строках 45–76 выполняется аналогичная проверка с использованием переменных-массивов `secondArray` и `secondArrayCopy` и метода `SecondDouble` (строки 92–100). Метод `SecondDouble` выполняет те же операции, что и `FirstDouble`, но получает аргумент-массив с использованием ключевого слова `ref`.

В этом случае ссылка, хранимая в `secondArray` после вызова метода, указывает на массив, созданный в строке 99 метода `SecondDouble`; мы видим, что переменная, переданная с ключевым словом `ref`, может быть изменена в вызванном методе, так что переменная на стороне вызова в действительности указывает на другой объект — в данном случае массив, созданный в `SecondDouble`. Команда `if...else` в строках 71–76 подтверждает, что `secondArray` и `secondArrayCopy` ссылаются на разные массивы.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 8.1

При передаче параметра ссылочного типа по значению метод получает копию ссылки на объект. При этом метод не может изменить ссылку, переданную методу. В подавляющем большинстве случаев такое поведение — защита передаваемой ссылки на объект, существующий на стороне вызова, — является именно тем, что требуется. Если же вызываемая процедура действительно должна изменять ссылку на объект, передайте параметр ссылочного типа с использованием ключевого слова `ref` — но еще раз подчеркнем, что такие ситуации встречаются редко.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 8.2

В C# вызываемым методам передаются ссылки на объекты (включая массивы). Вызываемый метод получает ссылку на объект, может работать с этим объектом, созданным на стороне вызова, — и, возможно, изменять его.

## 8.9. Пример: класс `GradeBook` с массивом для хранения оценок

В этом разделе мы продолжим работу над классом `GradeBook`, созданным в главе 4 и доработанным в главах 5 и 6. Напомним, что класс используется для хранения и анализа набора оценок. Предыдущие версии класса обрабатывали набор оценок, введенных пользователем, но не сохраняли значения в переменных экземпляра класса. Таким образом, для повторения вычислений пользователю приходилось заново вводить те же оценки. Одно из возможных решений этой проблемы основано на сохранении введенных оценок в отдельных экземплярах класса. Например, в классе `GradeBook` можно создать переменные экземпляров `grade1`, `grade2`, ..., `grade10` для хранения 10 оценок. Однако код суммирования оценок и вычисления средней оценки группы будет весьма громоздким, а класс не сможет обработать более 10 оценок за раз. В этом разделе проблема решается сохранением оценок в массиве.

### Сохранение оценок в массиве класса `GradeBook`

Представленная в этом разделе версия класса `GradeBook` (ил. 8.15) использует целочисленный массив для хранения оценок группы студентов на одном экзамене. Это избавляет пользователя от необходимости многократно вводить один набор оценок. Переменная экземпляра `grades` (которая будет ссылаться на массив с элементами `int`) объявляется в строке 7 — таким образом, каждый объект `GradeBook` поддерживает собственный набор оценок. Конструктор класса (строки 14–18) получает два параметра — название учебного курса и массив оценок. При создании объекта `GradeBook` приложение (например, класс `GradeBookTest` на ил. 8.16) передает существующий массив `int` конструктору, который присваивает ссылку на массив переменной экземпляра `grades` (строка 17). Размер массива `grades` определяется классом, передающим массив конструктору. Таким образом, объект `GradeBook` способен обработать переменное количество оценок — столько, сколько хранится в массиве на стороне вызова. Оценки в переданном массиве могут вводиться пользователем с клавиатуры или загружаться из файла на диске (см. главу 17). В нашем тестовом приложении массив просто инициализируется набором оценок (см. ил. 8.16, строка 9).

После того как оценки будут сохранены в переменной экземпляра `grades` класса `GradeBook`, все методы класса могут обращаться к элементам `grades` для выполнения необходимых вычислений.

```
1 // Ил. 8.15: GradeBook.cs
2 // Использование массива для хранения оценок.
3 using System;
4
5 public class GradeBook
6 {
7     private int[] grades; // Массив оценок
8
9     // Автоматически реализуемое свойство CourseName
10    public string CourseName { get; set; }
11
```

**Ил. 8.15.** Использование массива для хранения набора оценок (продолжение ↗)


```
12 // Конструктор с двумя параметрами инициализирует
13 // свойство CourseName и массив grades
14 public GradeBook( string name, int[] gradesArray )
15 {
16     CourseName = name; // CourseName присваивается значение name
17     grades = gradesArray; // Инициализация массива grades
18 } // Конец конструктора GradeBook с двумя параметрами
19
20 // Вывод сообщения для пользователя GradeBook
21 public void DisplayMessage()
22 {
23     // В свойстве CourseName хранится название курса
24     Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
25         CourseName );
26 } // Конец метода DisplayMessage
27
28 // Выполнение различных операций с данными
29 public void ProcessGrades()
30 {
31     // Вывод массива grades
32     OutputGrades();
33
34     // Вызов метода GetAverage для вычисления средней оценки
35     Console.WriteLine( "\nClass average is {0:F}", GetAverage() );
36
37     // Вызов методов GetMinimum и GetMaximum
38     Console.WriteLine( "Lowest grade is {0}\nHighest grade is {1}\n",
39         GetMinimum(), GetMaximum() );
40
41     // Вызов OutputBarChart для вывода гистограммы оценок
42     OutputBarChart();
43 } // Конец метода ProcessGrades
44
45 // Определение минимальной оценки
46 public int GetMinimum()
47 {
48     int lowGrade = grades[ 0 ]; // Предполагаем, что grades[ 0 ]
49                               // содержит минимальную оценку
50     // Перебор элементов массива grades
51     foreach ( int grade in grades )
52     {
53         // Если оценка меньше lowGrade, присвоить ее lowGrade
54         if ( grade < lowGrade )
55             lowGrade = grade; // Новая минимальная оценка
56     } // Конец for
57
58     return lowGrade; // Вернуть минимальную оценку
59 } // Конец метода GetMinimum
60
61 // Определение максимальной оценки
62 public int GetMaximum()
63 {
64     int highGrade = grades[ 0 ]; // Предполагаем, что grades[ 0 ]
65                               // содержит максимальную оценку
66     // Перебор элементов массива grades
```

**Ил. 8.15.** Использование массива для хранения набора оценок (продолжение ↗)

```

67     foreach ( int grade in grades )
68     {
69         // Если оценка больше highGrade, присвоить ее highGrade
70         if ( grade > highGrade )
71             highGrade = grade; // Новая максимальная оценка
72     } // Конец for
73
74     return highGrade; // Вернуть максимальную оценку
75 } // Конец метода GetMaximum
76
77 // Вычисление средней оценки
78 public double GetAverage()
79 {
80     int total = 0; // Инициализация total
81
82     // Суммирование оценок одного студента
83     foreach ( int grade in grades )
84         total += grade;
85
86     // Команда возвращает среднюю оценку
87     return ( double ) total / grades.Length;
88 } // Конец метода GetAverage
89
90 // Вывод гистограммы с распределением оценок
91 public void OutputBarChart()
92 {
93     Console.WriteLine( "Grade distribution:" );
94
95     // Сохранение распределения оценок по диапазонам
96     int[] frequency = new int[ 11 ];
97
98     // Для каждой оценки увеличивается соответствующий счетчик
99     foreach ( int grade in grades )
100         ++frequency[ grade / 10 ];
101
102     // Для каждого элемента массива выводится полоса на гистограмме
103     for ( int count = 0; count < frequency.Length; ++count )
104     {
105         // Метки полос ( "00-09: ", ..., "90-99: ", "100: " )
106         if ( count == 10 )
107             Console.Write( " 100: " );
108         else
109             Console.Write( "{0:D2}-{1:D2}:",
110                 count * 10, count * 10 + 9 );
111
112         // Вывод полосы звездочек
113         for ( int stars = 0; stars < frequency[ count ]; ++stars )
114             Console.Write( "*" );
115
116         Console.WriteLine(); // Переход на новую строку вывода
117     } // Конец внешнего цикла for
118 } // Конец метода OutputBarChart
119
120 // Вывод содержимого массива grades

```

**Ил. 8.15.** Использование массива для хранения набора оценок (продолжение )



```
121 public void OutputGrades()
122 {
123     Console.WriteLine( "The grades are:\n" );
124
125     // Вывод оценки каждого студента
126     for ( int student = 0; student < grades.Length; ++student )
127         Console.WriteLine( "Student {0,2}: {1,3}",
128                             student + 1, grades[ student ] );
129 } // Конец метода OutputGrades
130 } // Конец класса GradeBook
```

**Ил. 8.15.** Использование массива для хранения набора оценок (окончание)

### Метод ProcessGrades

Метод ProcessGrades (строки 29–43) содержит серию вызовов для вывода отчета с информацией об оценках. В строках 32 вызывается метод OutputGrades, выводящий содержимое массива grades. Строки 126–128 метода OutputGrades выводят оценки в цикле for. В данном случае вместо цикла foreach необходимо использовать цикл for, потому что в строках 127–128 значение счетчика student используется для вывода каждой оценки рядом с номером студента (см. ил. 8.16). Хотя индексы массива начинаются с 0, преподаватель обычно нумерует студентов с 1, поэтому в строках 127–128 в качестве номера выводится выражение student + 1; так строятся метки вида "Student 1: ", "Student 2: " и т. д.

### Метод GetAverage

Затем метод ProcessGrades вызывает метод GetAverage (строка 35) для вычисления средней оценки по массиву. Метод GetAverage (строки 78–88) использует команду foreach для суммирования значений в массиве grades перед вычислением средней оценки. Итерационная переменная в заголовке foreach (int grade в нашем случае) указывает, что для каждой итерации целочисленная переменная grade принимает значение из массива grades. Выражение в строке 87 использует свойство grades.Length для определения количества усредняемых оценок.

### Методы GetMinimum и GetMaximum

Строки 38–39 метода ProcessGrades вызывают методы GetMinimum и GetMaximum для определения минимальной и максимальной оценки по экзамену. В каждом из методов для перебора элементов массива grades используется команда foreach. Строки 51–56 метода GetMinimum перебирают элементы массива, а строки 54–55 сравнивают каждую оценку с lowGrade. Если оценка меньше lowGrade, то значение lowGrade обновляется значением оценки. При выполнении строки 58 переменная lowGrade содержит наименьшую оценку в массиве. Метод GetMaximum (строки 62–75) работает по тому же принципу, что и метод GetMinimum.

### Метод OutputBarChart

Наконец, строка 42 метода ProcessGrades вызывает метод OutputBarChart для вывода гистограммы оценок по аналогии с ил. 8.6. В данном примере мы вручную вычисляем количество оценок в каждой категории (то есть 0–9, 10–19, ..., 90–99 и 100) простым

перебором набора оценок. В строках 99–100 распределение оценок по категориям вычисляется по тем же принципам, что и в листингах на ил. 8.7 и 8.8. Строка 96 объявляет переменную `frequency` и инициализирует ее массивом из 11 элементов `int` для хранения количества оценок в каждой категории. Для каждой оценки в массиве `grades` строки 99–100 увеличивают соответствующий элемент массива `frequency`. Чтобы определить, какой элемент должен увеличиваться, строка 100 делит текущую оценку на 10 с использованием целочисленного деления. Например, если оценка равна 85, строка 100 увеличивает `frequency[8]` для обновления количества оценок в диапазоне 80–89. Строки 103–117 строят гистограмму (см. ил. 8.6) на основании значений из массива `frequency`. Строки 113–114 на ил. 8.15, как и строки 24–25 на ил. 8.6, используют значение из массива `frequency` для определения количества звездочек, образующих каждую полосу.

### Класс `GradeBookTest`

Приложение на ил. 8.16 создает объект класса `GradeBook` (см. ил. 8.15) с использованием массива `gradesArray` типа `int` (объявляется и инициализируется в строке 9). Строки 11–12 передают название курса и массив `gradesArray` конструктору `GradeBook`. Строка 13 выводит приветствие, а строка 14 вызывает метод `ProcessGrades` объекта `GradeBook`. В выходных данных представлена сводка по 10 оценкам `myGradeBook`.

```
1  // Ил. 8.16: GradeBookTest.cs
2  // Создание объекта GradeBook по массиву оценок.
3  public class GradeBookTest
4  {
5      // Метод Main начинает выполнение приложения
6      public static void Main( string[] args )
7      {
8          // Одномерный массив оценок
9          int[] gradesArray = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10
11          GradeBook myGradeBook = new GradeBook(
12              "CS101 Introduction to C# Programming", gradesArray );
13          myGradeBook.DisplayMessage();
14          myGradeBook.ProcessGrades();
15      } // Конец Main
16 } // Конец класса GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to C# Programming!
```

```
The grades are:
```

```
Student 1: 87
Student 2: 68
Student 3: 94
Student 4: 100
Student 5: 83
Student 6: 78
Student 7: 85
Student 8: 91
```

**Ил. 8.16.** Создание объекта `GradeBook` по массиву оценок (продолжение ↗)

```

Student 9: 76
Student 10: 87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *
```

**Ил. 8.16.** Создание объекта GradeBook по массиву оценок (окончание)



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 8.3

Тестовое приложение (также называемое «тестовой оснасткой») создает объект тестируемого класса и заполняет его данными. Эти данные могут поступать из разных источников: они могут помещаться прямо в массив с использованием инициализатора, вводиться пользователем с клавиатуры или загружаться из файла (см. главу 17). После передачи этих данных (обычно через конструктор класса) и создания объекта усилия должны направляться на тестирование его методов и работу с данными. Такой сбор данных в тестовой оснастке позволяет классу работать с данными из нескольких источников.

## 8.10. Многомерные массивы

Двумерные массивы часто используются для представления таблиц с информацией, упорядоченной по строкам и столбцам. Конкретный элемент таблицы идентифицируется двумя индексами. Обычно первый индекс определяет строку, а второй — столбец, на пересечении которых находится элемент. Массивы, у которых элемент определяется двумя индексами, называются *двумерными* массивами. (Многомерные массивы могут иметь более двух измерений, но эта тема выходит за рамки данной книги). В C# поддерживаются два типа двумерных массивов — *прямоугольные* и *ступенчатые*.

### Прямоугольные массивы

Прямоугольные массивы используются для представления таблиц, организованных в форме строк и столбцов, у которых каждая строка содержит одинаковое число столбцов. На ил. 8.17 изображен прямоугольный массив из трех строк и четырех столбцов (массив  $3 \times 4$ ). Обычно двумерный массив с  $m$  строк и  $n$  столбцов называется массивом  $m \times n$ .

	Столбец 0	Столбец 1	Столбец 2	Столбец 3
Строка 0	a[ 0, 0 ]	a[ 0, 1 ]	a[ 0, 2 ]	a[ 0, 3 ]
Строка 1	a[ 1, 0 ]	a[ 1, 1 ]	a[ 1, 2 ]	a[ 1, 3 ]
Строка 2	a[ 2, 0 ]	a[ 2, 1 ]	a[ 2, 2 ]	a[ 2, 3 ]

Имя массива  
 Индекс строки  
 Индекс столбца

**Ил. 8.17.** Прямоугольный массив с тремя строками и четырьмя столбцами

Каждый элемент массива *a* на ил. 8.17 идентифицируется выражением *a[строка, столбец]*, где *a* — имя массива, а *строка* и *столбец* — индексы, однозначно идентифицирующие каждый элемент массива *a* номерами строки и столбца. У всех элементов строки 0 первый индекс равен 0, а у всех элементов столбца 3 второй индекс равен 3.

### Инициализатор двумерного прямоугольного массива

Многомерные массивы (как и одномерные) могут инициализироваться при объявлении. Прямоугольный массив *b* из двух строк и двух столбцов объявляется и инициализируется вложенными инициализаторами массивов:

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

Инициализаторы группируются по строкам в фигурных скобках. Таким образом, значения 1 и 2 инициализируют элементы *b[0, 0]* и *b[0, 1]*, а 3 и 4 — элементы *b[1, 0]* и *b[1, 1]* соответственно. Компилятор подсчитывает количество вложенных инициализаторов (представленных двумя парами внутренних фигурных скобок во внешних скобках) в списке для определения количества строк в массиве *b*. Количество столбцов (два) в строке определяется подсчетом инициализаторов во вложенном инициализаторе массива. Если количество инициализаторов в разных строках не совпадает, компилятор выдает сообщение об ошибке, потому что все строки прямоугольного массива должны иметь одинаковую длину.

### Ступенчатые массивы

Ступенчатый массив хранится как одномерный массив, в котором каждый элемент ссылается на одномерный массив. Этот способ представления ступенчатых массивов делает их чрезвычайно гибкими, потому что длины строк массива могут быть разными. Например, ступенчатый массив может использоваться для хранения оценок одного студента по нескольким учебным курсам, с разным количеством экзаменов в разных курсах.

### Инициализатор двумерного ступенчатого массива

Для обращения к элементам ступенчатого массива используется выражение в форме *ИмяМассива[строка][столбец]* — по аналогии с выражением для прямоугольных

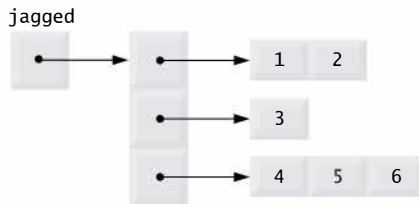
массивов, но с отдельным набором квадратных скобок для каждого измерения. Ступенчатый массив с тремя строками разной длины объявляется и инициализируется следующим образом:

```
int[][] jagged = { new int[] { 1, 2 },
                  new int[] { 3 },
                  new int[] { 4, 5, 6 } };
```

В этой команде значения 1 и 2 инициализируют элементы `jagged[0][0]` и `jagged[0][1]` соответственно; значение 3 инициализирует `jagged[1][0]`; а значения 4, 5 и 6 инициализируют `jagged[2][0]`, `jagged[2][1]` и `jagged[2][2]` соответственно. Таким образом, массив `jagged` из предыдущего объявления фактически состоит из четырех разных одномерных массивов — один представляет строки, другой содержит значения первой строки (`{1, 2}`), третий содержит значения второй строки (`{3}`) и последний содержит значения третьей строки (`{4, 5, 6}`). Таким образом, массив `jagged` состоит из трех элементов, каждый из которых содержит ссылку на одномерный массив значений `int`.

### Структура двумерного ступенчатого массива в памяти

Обратите внимание на различия между выражениями создания прямоугольных и ступенчатых массивов. За типом ступенчатого массива следуют две пары квадратных скобок, указывающие, что это массив массивов `int`. Кроме того, C# требует, чтобы в инициализаторе массива C# для объекта массива каждой строки присутствовало ключевое слово `new`. На ил. 8.18 изображена ссылка на массив `jagged` после его объявления и инициализации.



**Ил. 8.18.** Массив `jagged` с тремя строками разной длины

### Создание двумерных массивов

Прямоугольный массив может создаваться традиционным выражением создания массива. Например, следующий фрагмент объявляет переменную `b` и присваивает ей ссылку на прямоугольный массив  $3 \times 4$ :

```
int[ , ] b;
b = new int[ 3, 4 ];
```

В этом случае количества строк и столбцов задаются литералами 3 и 4, но это не обязательно — размеры массивов могут также задаваться переменными и выражениями. Как и в случае с одномерными массивами, элементы прямоугольного массива инициализируются при создании объекта массива.

Ступенчатый массив не может быть полностью создан в одном выражении создания массива. Следующая команда содержит синтаксическую ошибку:

```
int[][] c = new int[ 2 ][ 5 ]; // Ошибка
```

Вместо этого каждый одномерный массив ступенчатого массива должен инициализироваться отдельно. Ступенчатый массив создается следующим образом:

```
int[][] c;
c = new int[ 2 ][ ]; // создать 2 строки
c[ 0 ] = new int[ 5 ]; // создать 5 столбцов для строки 0
c[ 1 ] = new int[ 3 ]; // создать 3 столбца для строки 1
```

Этот фрагмент создает ступенчатый массив с двумя строками. Строка 0 состоит из 5 столбцов, а строка 1 — из 3 столбцов.

### Пример использования двумерного массива: вывод значений элементов

В листинге на ил. 8.19 продемонстрирована инициализация прямоугольных и ступенчатых массивов с использованием инициализаторов массивов и перебором элементов во вложенных циклах `for`. Метод `Main` класса `InitArray` создает два массива. Строка 12 использует вложенные инициализаторы массивов для инициализации переменной `rectangular` массивом, у которого строка 0 содержит значения 1, 2 и 3, а строка 1 — значения 4, 5 и 6. Строки 17–19 используют вложенные инициализаторы разной длины для инициализации переменной `jagged`. В этом случае инициализация использует ключевое слово `new` для создания одномерного массива для каждой строки. Строка 0 инициализируется двумя элементами со значениями 1 и 2 соответственно. Строка 1 инициализируется одним элементом со значением 3. Строка 2 инициализируется тремя элементами со значениями 4, 5 и 6 соответственно.

```
1 // Ил. 8.19: InitArray.cs
2 // Инициализация прямоугольных и ступенчатых массивов.
3 using System;
4
5 public class InitArray
6 {
7     // Создание и вывод прямоугольных и ступенчатых массивов
8     public static void Main( string[] args )
9     {
10         // У прямоугольных массивов все строки должны
11         // иметь одинаковую длину.
12         int[ , ] rectangular = { { 1, 2, 3 }, { 4, 5, 6 } };
13
14         // У ступенчатых массивов
15         // для каждой строки необходимо использовать "new int[]",
16         // но длина строк может различаться.
17         int[][] jagged = { newint[] { 1, 2 },
18                             new int[] { 3 },
19                             new int[] { 4, 5, 6 } };
20
21         OutputArray( rectangular ); // Вывод массива rectangular по строкам
```

**Ил. 8.19.** Инициализация ступенчатых и прямоугольных массивов (продолжение ☞)

```

22     Console.WriteLine(); // Вывод пустой строки
23     OutputArray( jagged ); // Вывод массива jagged по строкам
24 } // end Main
25
26 // Вывод строк и столбцов прямоугольного массива
27 public static void OutputArray( int[ , ] array )
28 {
29     Console.WriteLine( "Values in the rectangular array by row are" );
30
31     // Перебор строк массива
32     for ( int row = 0; row < array.GetLength( 0 ); ++row )
33     {
34         // Перебор столбцов текущей строки
35         for ( int column = 0; column < array.GetLength( 1 ); ++column )
36             Console.Write( "{0} ", array[ row, column ] );
37
38         Console.WriteLine(); // Начало новой строки вывода
39     } // Конец внешнего цикла for
40 } // Конец метода OutputArray
41
42 // Вывод строк и столбцов ступенчатого массива
43 public static void OutputArray( int[][] array )
44 {
45     Console.WriteLine( "Values in the jagged array by row are" );
46
47     // Перебор строк
48     foreach ( int[] row in array )
49     {
50         // Перебор всех элементов текущей строки
51         foreach ( int element in row )
52             Console.Write( "{0} ", element );
53
54         Console.WriteLine(); // Переход на новую строку вывода
55     } // Конец внешнего цикла foreach
56 } // Конец метода OutputArray
57 } // Конец класса InitArray

```

```

Values in the rectangular array by row are
1 2 3
4 5 6
Values in the jagged array by row are
1 2
3
4 5 6

```

**Ил. 8.19.** Инициализация ступенчатых и прямоугольных массивов (окончание)

### Перегруженный метод **OutputArray**

Метод `OutputArray` перегружается в двух версиях. У первой версии (строки 27–40) определяется параметр `int[ , ] array`, который означает, что версия получает прямоугольный массив. Вторая версия (строки 43–56) получает ступенчатый массив, потому что ее параметр определен в виде `int[][] array`.

### Метод `OutputArray` для прямоугольных массивов

В строке 21 метод `OutputArray` вызывается с аргументом `rectangular`, поэтому вызывается версия `OutputArray` в строках 27–40. Вложенная команда `for` (строки 32–39) выводит строки прямоугольного массива. Условие продолжения цикла каждой команды `for` (строки 32 и 35) использует метод `GetLength` прямоугольного массива для получения длины каждого измерения. Нумерация измерений начинается с 0, поэтому вызов `GetLength(0)` для объекта массива возвращает длину первого измерения (количество строк), а вызов `GetLength(1)` возвращает длину второго измерения (количество столбцов).

### Метод `OutputArray` для ступенчатых массивов

В строке 23 метод `OutputArray` вызывается для ступенчатого массива `jagged`, поэтому вызывается версия `OutputArray` в строках 43–56. Вложенная команда `foreach` (строки 48–55) выводит строки ступенчатого массива. Внутренняя команда `foreach` (строки 51–52) перебирает все элементы текущей строки массива, чтобы цикл мог определить точное количество столбцов в каждой строке. Поскольку ступенчатый массив создается как массив массивов, мы можем использовать вложенные команды `foreach` для вывода элементов в консольном окне. Внешний цикл перебирает элементы массива, которые представляют собой ссылки на одномерные массивы значений `int`, представляющие каждую строку. Внутренний цикл перебирает элементы текущей строки. Команда `foreach` также может перебрать все элементы прямоугольного массива. В этом случае `foreach` перебирает все строки и столбцы начиная с 0, как если бы элементы хранились в одномерном массиве.

### Операции с многомерными массивами, часто выполняемые в циклах `for`

Во многих типичных операциях с массивами используются команды `for`. Например, следующая команда `for` заполняет нулями все элементы строки 2 прямоугольного массива `a` на ил. 8.17:

```
for ( int column = 0; column < a.GetLength( 1 ); ++column )
    a[ 2, column ] = 0;
```

Первый индекс всегда равен 2 (0 для первой строки, 1 для второй строки). В цикле изменяется только второй индекс (то есть индекс столбца). Приведенный цикл `for` эквивалентен следующим командам присваивания:

```
a[ 2, 0 ] = 0;
a[ 2, 1 ] = 0;
a[ 2, 2 ] = 0;
a[ 2, 3 ] = 0;
```

Следующая конструкция со вложенным циклом `for` суммирует значения всех элементов массива `a`:

```
int total = 0;

for ( int row = 0; row < a.GetLength( 0 ); ++row )
{
    for ( int column = 0; column < a.GetLength( 1 ); ++column )
        total += a[ row, column ];
} // Конец внешнего цикла for
```



Вложенные циклы `for` суммируют элементы массива по строкам. Внешний цикл сначала задает индекс строки равным 0, чтобы внутренний цикл `for` мог просуммировать элементы строки 0. Затем внешний цикл `for` увеличивает `row` до 1, чтобы внутренний цикл просуммировал элементы строки 1. Далее внешний цикл `for` увеличивает `row` до 2 для суммирования элементов строки 2. После завершения внешнего цикла выводится накопленное значение `total`. В следующем примере вы увидите, как обработать прямоугольный массив в более компактной записи с использованием `foreach`.

## 8.11. Пример: класс GradeBook с использованием прямоугольного массива

В разделе 8.9 был представлен класс `GradeBook` (см. ил. 8.15), который использовал одномерный массив для хранения оценок одного экзамена. На большинстве учебных курсов студенты сдают несколько экзаменов. Возможно, преподаватель захочет проанализировать оценки за весь курс — как для одного студента, так и для группы в целом.

### Хранение оценок в прямоугольном массиве

На ил. 8.20 представлена версия класса `GradeBook`, использующая прямоугольный массив `grades` для хранения оценок группы студентов на нескольких экзаменах. Каждая строка массива представляет оценки одного студента за весь курс, а каждый столбец — оценки всей группы по одному из экзаменов, сдаваемых в этом курсе. Такое приложение, как `GradeBookTest` (ил. 8.21), передает массив в аргументе конструктора `GradeBook`. В нашем примере используется массив  $10 \times 3$ , содержащий оценки 10 студентов на трех экзаменах. Пять методов выполняют операции с массивом для обработки оценок. Каждый метод является аналогом метода одномерной версии класса `GradeBook` (см. ил. 8.15). Метод `GetMinimum` (строки 44–58 на ил. 8.20) определяет наименьшую оценку среди студентов за семестр. Метод `GetMaximum` (строки 61–75) определяет наивысшую оценку среди студентов за семестр. Метод `GetAverage` (строки 78–90) вычисляет среднюю оценку конкретного студента за семестр. Метод `OutputBarChart` (строки 93–122) строит гистограмму распределения всех оценок за семестр. Метод `OutputGrades` (строки 125–149) выводит двумерный массив в табличном формате, вместе со средними оценками всех студентов за семестр.

```
1 // Ил. 8.20: GradeBook.cs
2 // Хранение оценок в прямоугольном массиве.
3 using System;
4
5 public class GradeBook
6 {
7     private int[ , ] grades; // Прямоугольный массив оценок
8 }
```

**Ил. 8.20.** Хранение оценок в прямоугольном массиве (продолжение ➤)

```

9      // Автоматически реализуемое свойство CourseName
10     public string CourseName { get; set; }
11
12     // Конструктор с двумя параметрами инициализирует
13     // свойство CourseName и массив grades
14     public GradeBook( string name, int[ , ] gradesArray )
15     {
16         CourseName = name; // CourseName присваивается значение name
17         grades = gradesArray; // Инициализация массива grades
18     } // Конец конструктора GradeBook с двумя параметрами
19
20     // Вывод сообщения для пользователя GradeBook
21     public void DisplayMessage()
22     {
23         // В свойстве CourseName хранится название курса
24         Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
25                             CourseName );
26     } // Конец метода DisplayMessage
27
28     // Выполнение различных операций с данными
29     public void ProcessGrades()
30     {
31         // Вывод массива оценок
32         OutputGrades();
33
34         // Вызов методов GetMinimum и GetMaximum
35         Console.WriteLine( "\n{0} {1}\n{2} {3}\n",
36                             "Lowest grade in the grade book is", GetMinimum(),
37                             "Highest grade in the grade book is", GetMaximum() );
38
39         // Вывод гистограммы распределения оценок по всем экзаменам
40         OutputBarChart();
41     } // Конец метода ProcessGrades
42
43     // Определение минимальной оценки
44     public int GetMinimum()
45     {
46         // Предполагаем, что первый элемент grades содержит
47         int lowGrade = grades[ 0, 0 ]; // минимальную оценку
48
49         // Перебор элементов прямоугольного массива grades
50         foreach ( int grade in grades )
51         {
52             // Если оценка меньше lowGrade, присвоить ее lowGrade
53             if ( grade < lowGrade )
54                 lowGrade = grade;
55         } // Конец foreach
56
57         return lowGrade; // Вернуть минимальную оценку
58     } // Конец метода GetMinimum
59
60     // Определение максимальной оценки
61     public int GetMaximum()
62     {

```

**Ил. 8.20.** Хранение оценок в прямоугольном массиве (продолжение ➤)

```

63      // Предполагаем, что первый элемент grades содержит
64      int highGrade = grades[ 0, 0 ]; // максимальную оценку
65
66      // Перебор элементов прямоугольного массива grades
67      foreach ( int grade in grades )
68      {
69          // Если оценка выше highGrade, присвоить ее highGrade
70          if ( grade > highGrade )
71              highGrade = grade;
72      } // Конец foreach
73
74      return highGrade; // Вернуть максимальную оценку
75 } // Конец метода GetMaximum
76
77 // Вычисление средней оценки для конкретного студента
78 public double GetAverage( int student )
79 {
80     // Определение количества оценок на студента
81     int amount = grades.GetLength( 1 );
82     int total = 0; // Инициализация total
83
84     // Суммирование оценок для одного студента
85     for ( int exam = 0; exam < amount; ++exam )
86         total += grades[ student, exam ];
87
88     // Вернуть среднюю оценку
89     return ( double ) total / amount;
90 } // Конец метода GetAverage
91
92 // Вызов OutputBarChart для вывода гистограммы оценок
93 public void OutputBarChart()
94 {
95     Console.WriteLine( "Overall grade distribution:" );
96
97     // Сохранение распределения оценок по диапазонам
98     int[] frequency = new int[ 11 ];
99
100    // Для каждой оценки увеличивается соответствующий счетчик
101    foreach ( int grade in grades )
102    {
103        ++frequency[ grade / 10 ];
104    } // Конец foreach
105
106    // Для каждого элемента массива выводится полоса на гистограмме
107    for ( int count = 0; count < frequency.Length; ++count )
108    {
109        // Метки полос ( "00-09: ", ..., "90-99: ", "100: " )
110        if ( count == 10 )
111            Console.Write( " 100: " );
112        else
113            Console.Write( "{0:D2}-{1:D2}: ",
114                count * 10, count * 10 + 9 );
115
116        // Вывод полосы звездочек

```

**Ил. 8.20.** Хранение оценок в прямоугольном массиве (продолжение ➤)

```

117         for ( int stars = 0; stars < frequency[ count ]; ++stars )
118             Console.Write( "*" );
119
120         Console.WriteLine(); // Переход на новую строку вывода
121     } // Конец внешнего цикла for
122 } // Конец метода OutputBarChart
123
124 // Вывод содержимого массива оценок
125 public void OutputGrades()
126 {
127     Console.WriteLine( "The grades are:\n" );
128     Console.Write( " " ); // Выравнивание заголовков столбцов
129
130     // Создание заголовка столбца для каждого экзамена
131     for ( int test = 0; test < grades.GetLength( 1 ); ++test )
132         Console.Write( "Test {0} ", test + 1 );
133
134     Console.WriteLine( "Average" ); // Заголовок столбца
135                                     // средней оценки
136     // Создание строк/столбцов массива оценок
137     for ( int student = 0; student < grades.GetLength( 0 ); ++student )
138     {
139         Console.Write( "Student {0,2}", student + 1 );
140
141         // Вывод оценок студента
142         for ( int grade = 0; grade < grades.GetLength( 1 ); ++grade )
143             Console.Write( "{0,8}", grades[ student, grade ] );
144
145         // Вызов метода GetAverage для вычисления средней оценки.
146         // Номер строки передается в аргументе GetAverage.
147         Console.WriteLine( "{0,9:F}", GetAverage( student ) );
148     } // Конец внешнего цикла for
149 } // Конец метода OutputGrades
150 } // Конец класса GradeBook

```

**Ил. 8.20.** Хранение оценок в прямоугольном массиве (окончание)

### Обработка двумерного массива командой foreach

Методы `GetMinimum`, `GetMaximum` и `OutputBarChart` перебирают элементы массива `grades` в цикле `foreach`. Например, команда `foreach` метода `GetMinimum` (строки 50–55) для определения минимальной оценки перебирает элементы прямоугольного массива `grades` и сравнивает каждый элемент с переменной `lowGrade`. Если оценка ниже `lowGrade`, то `lowGrade` заменяется новым значением.

Когда команда `foreach` перебирает элементы массива `grades`, она сначала просматривает все элементы первой строки в порядке индексов, затем все элементы второй строки и т. д. Порядок перебора элементов командой `foreach` в строках 50–55 эквивалентен следующему коду, использующему вложенные команды `for`:

```

for ( int row = 0; row < grades.GetLength( 0 ); ++row )
    for ( int column = 0; column < grades.GetLength( 1 ); ++column )
    {
        if ( grades[ row, column ] < lowGrade )
            lowGrade = grades[ row, column ];
    }

```

После завершения цикла `foreach` переменная `lowGrade` содержит наименьшую оценку в прямоугольном массиве. Метод `GetMaximum` работает аналогично `GetMinimum`.

### Метод `OutputBarChart`

Метод `OutputBarChart` (строки 93–122) выводит информацию о распределении оценок в формате гистограммы. Синтаксис команды `foreach` (строки 101–104) для одно- и двумерных массивов идентичен.

### Метод `OutputGrades`

Метод `OutputGrades` (строки 125–149) использует вложенные циклы `for` для вывода значений массива `grades`, а также средней оценки каждого студента за семестр. Результат показан на ил. 8.21. Строки 131–132 (на ил. 8.20) выводят заголовки столбцов для каждого экзамена. В данном случае вместо `foreach` используется команда `for`, чтобы каждый экзамен можно было идентифицировать числом. Аналогичным образом команда `for` в строках 137–148 сначала выводит метку строки, используя переменную-счетчик для идентификации каждого студента (строка 139). Индексы массивов начинаются с 0, поэтому строки 132 и 139 выводят значения `test + 1` и `student + 1` соответственно, чтобы нумерация экзаменов и студентов начиналась с 1 (см. ил. 8.21). Внутренний цикл `for` в строках 142–143 использует переменную `student` внешнего цикла `for` для перебора конкретной строки массива `grades` и вывода оценок студента. Наконец, строка 147 получает среднюю оценку каждого студента, передавая индекс строки `grades` (то есть `student`) методу `GetAverage`.

### Метод `GetAverage`

Метод `GetAverage` (строки 78–90) получает один аргумент — индекс строки конкретного студента. При вызове метода `GetAverage` в строке 147 передается аргумент `student` типа `int`, задающий конкретную строку прямоугольного массива `grades`. Метод `GetAverage` вычисляет сумму элементов массива этой строки, делит ее на количество оценок и возвращает вещественный результат как значение `double` (строка 89).

### Класс `GradeBookTest` для тестирования класса `GradeBook`

Приложение на ил. 8.21 создает объект класса `GradeBook` (см. ил. 8.20) с использованием двумерного массива `int`, на который ссылается `gradesArray` (см. ил. 8.21, строки 9–18). В строках 20–21 название курса и массив `gradesArray` передаются конструктору `GradeBook`. Затем в строках 22–23 вызываются методы `DisplayMessage` и `ProcessGrades` объекта `myGradeBook`, которые выводят сообщение и отчет со сводкой оценок студентов за семестр.

```
1 // Ил. 8.21: GradeBookTest.cs
2 // Создание объекта GradeBook с использованием прямоугольного массива.
3 public class GradeBookTest
4 {
5     // Метод Main начинает выполнение программы
6     public static void Main( string[] args )
7     {
8         // Прямоугольный массив оценок
```

**Ил. 8.21.** Создание объекта `GradeBook` с использованием прямоугольного массива оценок (продолжение ↗)

```

9      int[ , ] gradesArray = { { 87, 96, 70 },
10                               { 68, 87, 90 },
11                               { 94, 100, 90 },
12                               { 100, 81, 82 },
13                               { 83, 65, 85 },
14                               { 78, 87, 65 },
15                               { 85, 75, 83 },
16                               { 91, 94, 100 },
17                               { 76, 72, 84 },
18                               { 87, 93, 73 } };
19
20      GradeBook myGradeBook = new GradeBook(
21          "CS101 Introduction to C# Programming", gradesArray );
22      myGradeBook.DisplayMessage();
23      myGradeBook.ProcessGrades();
24  } // Конец Main
25 } // Конец класса GradeBookTest

```

Welcome to the grade book for  
CS101 Introduction to C# Programming!  
The grades are:

	Test 1	Test 2	Test 3	Average
Student	1	87	96	70 84.33
Student	2	68	87	90 81.67
Student	3	94	100	90 94.67
Student	4	100	81	82 87.67
Student	5	83	65	85 77.67
Student	6	78	87	65 76.67
Student	7	85	75	83 81.00
Student	8	91	94	100 95.00
Student	9	76	72	84 77.33
Student	10	87	93	73 84.33

Lowest grade in the grade book is 65  
Highest grade in the grade book is 100

Overall grade distribution:

```

00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: *****
80-89: *****
90-99: *****
100: ***

```

**Ил. 8.21.** Создание объекта GradeBook с использованием прямоугольного массива оценок (окончание)

## 8.12. Списки аргументов переменной длины

Списки аргументов переменной длины позволяют создавать методы с переменным количеством аргументов. Аргумент с типом одномерного массива, перед которым в списке параметров ставится ключевое слово `params`, означает, что метод получает переменное количество аргументов с типом элементов массива. Такое использование модификатора `params` допускается только в последнем элементе списка параметров. Хотя перегрузка методов и передача массивов позволяют сделать многое из того, для чего обычно применяются списки аргументов переменной длины, модификатор `params` делает код более четким и понятным.

На ил. 8.22 приведен метод `Average` (строки 8–17), получающий последовательность `double` переменной длины (строка 8). С# интерпретирует список аргументов переменной длины как одномерный массив, все элементы которого относятся к одному типу. Таким образом, тело метода может работать с параметром `numbers` как с массивом `double`. Строки 13–14 использует цикл `foreach` для перебора элементов и вычисления суммы хранящихся в нем чисел. Строка 16 использует свойство `numbers.Length` для получения размера массива `numbers`, задействованного в вычислении средней оценки. В строках 31, 33 и 35 метода `Main` вызывается метод `Average` с двумя, тремя и четырьмя аргументами соответственно. Метод `Average` имеет список аргументов переменной длины, чтобы он мог обработать столько аргументов `double`, сколько передаст вызывающая сторона. Из выходных данных видно, что каждый вызов метода `Average` возвращает правильное значение.



### ТИПИЧНАЯ ОШИБКА 8.5

Модификатор `params` может использоваться только с последним параметром списка параметров.

```

1 // Ил. 8.22: ParamArrayTest.cs
2 // Использование списков аргументов переменной длины.
3 using System;
4
5 public class ParamArrayTest
6 {
7     // Вычисление средней оценки.
8     public static double Average( params double[] numbers )
9     {
10         double total = 0.0; // Инициализация total
11
12         // Вычисление суммы в цикле foreach
13         foreach ( double d in numbers )
14             total += d;
15
16         return total / numbers.Length;
17     } // Конец метода Average
18

```

**Ил. 8.22.** Использование списков аргументов переменной длины (продолжение ↗)

```
19 public static void Main( string[] args )
20 {
21     double d1 = 10.0;
22     double d2 = 20.0;
23     double d3 = 30.0;
24     double d4 = 40.0;
25
26     Console.WriteLine(
27         "d1 = {0:F1}\nd2 = {1:F1}\nd3 = {2:F1}\nd4 = {3:F1}\n",
28         d1, d2, d3, d4 );
29
30     Console.WriteLine( "Average of d1 and d2 is {0:F1}",
31         Average( d1, d2 ) );
32     Console.WriteLine( "Average of d1, d2 and d3 is {0:F1}",
33         Average( d1, d2, d3 ) );
34     Console.WriteLine( "Average of d1, d2, d3 and d4 is {0:F1}",
35         Average( d1, d2, d3, d4 ) );
36 } // Конец Main
37 } // Конец класса ParamArrayTest
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0
```

```
Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

**Ил. 8.22.** Использование списков аргументов переменной длины (окончание)

## 8.13. Аргументы командной строки

Во многих системах пользователь может передать аргументы из командной строки приложению; для этого в список параметров `Main` включается параметр типа `string[]` (то есть массив с элементами `string`), как это делалось во всех приведенных ранее приложениях. По общепринятой схеме этому параметру присваивается имя `args` (ил. 8.23, строка 7). При запуске приложения из командной строки исполнительная среда передает методу `Main` аргументы, следующие за именем приложения, как строки, объединенные в одномерный массив `args`. Количество аргументов, переданных из командной строки, определяется значением свойства `Length` массива. Например, команда `"MyApp a b"` передает приложению `MyApp` два аргумента командной строки. Аргументы командной строки разделяются пропусками (whitespaces), а не запятыми. При выполнении приведенной команды точка входа `Main` получает массив `args` из двух элементов (то есть свойство `args.Length` равно 2), в котором элемент `args[0]` содержит строку `"a"`, а `args[1]` содержит строку `"b"`. Обычно аргументы командной строки используются для передачи приложениям параметров и имен файлов.



```

1 // Ил. 8.23: InitArray.cs
2 // инициализация массива по аргументам командной строки.
3 using System;
4
5 public class InitArray
6 {
7     public static void Main( string[] args )
8     {
9         // Проверка количества аргументов командной строки
10        if ( args.Length != 3 )
11            Console.WriteLine(
12                "Error: Please re-enter the entire command, including\n" +
13                "an array size, initial value and increment." );
14        else
15        {
16            // Получение размера массива из первого аргумента
17            int arrayLength = Convert.ToInt32( args[ 0 ] );
18            int[] array = new int[ arrayLength ]; // Создание массива
19
20            // Получение исходного значения и приращения из аргументов
21            int initialValue = Convert.ToInt32( args[ 1 ] );
22            int increment = Convert.ToInt32( args[ 2 ] );
23
24            // Вычисление значения каждого элемента массива
25            for ( int counter = 0; counter < array.Length; ++counter )
26                array[ counter ] = initialValue + increment * counter;
27
28            Console.WriteLine( "{0}{1,8}", "Index", "Value" );
29
30            // Вывод индексов и значений
31            for ( int counter = 0; counter < array.Length; ++counter )
32                Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
33        } // Конец else
34    } // Конец Main
35 } // Конец класса InitArray

```

C:\Examples\ch08\fig08\_23>InitArray.exe  
Error: Please re-enter the entire command, including  
an array size, initial value and increment.

```

C:\Examples\ch08\fig08_23>InitArray.exe 5 0 4
Index Value
0      0
1      4
2      8
3     12
4     16

```

```

C:\Examples\ch08\fig08_23>InitArray.exe 10 1 2
Index Value
0      1
1      3
2      5
3      7
4      9

```

**Ил. 8.23.** Использование аргументов командной строки  
для инициализации массива (продолжение ↗)

```
5    11
6    13
7    15
8    17
9    19
```

**Ил. 8.23.** Использование аргументов командной строки для инициализации массива (окончание)

В листинге на ил. 8.23 три аргумента командной строки используются для инициализации массива. Если значение `args.Length` отлично от 3, приложение выводит сообщение об ошибке и завершается (строки 10–13). В противном случае строки 16–32 инициализируют и выводят массив на основании значений аргументов.

Метод `Main` работает с аргументами командной строки как с объектами `string` в массиве `args`. Строка 17 получает `args[0]` — строку с размером массива, и преобразует ее в значение `int`, которое используется приложением для создания массива в строке 18. Статический метод `ToInt32` класса `Convert` преобразует свой аргумент `string` в `int`.

Строки 21–22 преобразуют аргументы командной строки `args[1]` и `args[2]` в значения `int` и сохраняют их в переменных `initialValue` и `increment` соответственно. Строки 25–26 вычисляют значение каждого элемента массива.

В первом примере приложение было запущено с недостаточным количеством аргументов командной строки. Во втором примере аргументы командной строки 5, 0 и 4 используются для задания размера массива (5), значения первого элемента (0) и приращения значений в массиве (4) соответственно. Из результатов видно, что для этих значений создается массив с целыми числами 0, 4, 8, 12 и 16. Вывод третьего примера показывает, что с аргументами командной строки 10, 1 и 2 создается массив с 10 элементами, содержащими неотрицательные нечетные числа от 1 до 19.

### Определение аргументов командной строки в Visual Studio

Мы запустили этот пример в окне командной строки, но аргументы командной строки также можно задать в IDE. Щелкните правой кнопкой мыши на узле `Properties` проекта в окне `Solution Explorer` и выберите команду `Open`. Перейдите на вкладку `Debug` и введите аргументы в текстовом поле `Command line arguments`.

## 8.14. Итоги

В этой главе началось наше знакомство со структурами данных. Мы рассмотрели примеры использования массивов для хранения и загрузки различных данных, от списков до таблиц. Также вы узнали, как объявлять переменные массивов, как инициализировать массивы и обращаться к их отдельным элементам.

В этой главе была представлена команда `foreach`, обеспечивающая дополнительный способ перебора массивов (кроме команды `for`). Вы узнали, как передавать массивы методам, как объявлять многомерные массивы и работать в них. В завершение главы было показано, как пишутся методы со списками аргументов переменной длины и как приложение получает аргументы, заданные в командной строке.

# 9 LINQ и коллекция List

## 9.1. Введение

В главе 8 были рассмотрены массивы — простые структуры данных для хранения элементов определенного типа. При всей широте распространения массивы обладают ограниченными возможностями. Например, размер массива должен быть задан при его создании, а если во время выполнения его потребуется изменить — это приходится делать вручную, создавая новый массив или вызывая метод `Resize` класса `Array`, а это связано с затратами ресурсов на создание нового массива и копирование существующих элементов.

В этой главе будут представлены другие структуры данных — классы коллекций .NET Framework, обладающие более широкими возможностями, чем традиционные массивы. Эти классы надежны, универсальны и эффективны; они были тщательно спроектированы и протестированы на правильность работы и производительность. В этой главе основное внимание уделяется коллекции `List`. Эта коллекция похожа на массив, но обладает дополнительными возможностями, включая динамическое изменение размеров (размер такой коллекции может увеличиваться при добавлении элементов и уменьшаться при их удалении). Мы используем коллекцию `List` для реализации нескольких примеров, сходных с примерами предыдущей главы.

Большие объемы данных, которые должны сохраняться после завершения приложения, обычно хранятся в *базах данных* (более подробно эта тема рассматривается в главе 22). Система управления базами данных (*СУБД*) предоставляет механизмы хранения, организации, чтения и изменения данных в базе. Международным стандартом выполнения запросов (то есть выборки информации, удовлетворяющей некоторому критерию) и обработки данных является язык *SQL*. Многие коды программы, работающие с реляционными базами данных, передавали СУБД запросы на языке *SQL*, а затем обрабатывали результаты.

В этой главе представлена новая функциональность языка *C#* — технология *LINQ* (Language Integrated Query). *LINQ* позволяет писать запросы, сходные с запросами *SQL*, для чтения информации из различных источников (не только баз данных).

В этой главе мы используем технологию *LINQ to Objects* для запроса информации из массивов и списков и отбора элементов, удовлетворяющих набору условий (это называется *фильтрацией*). В таблице на ил. 9.1 показано, где и как в книге используется технология LINQ.

Глава	Использование
Глава 9	Получение информации из массивов и List
Глава 17	Поиск в каталогах и операции с текстовыми файлами
Глава 22	Выборка информации из базы данных
Глава 23	Выборка информации из базы данных для использования в веб-приложении

**Ил. 9.1.** Использование LINQ в книге

### Поставщики LINQ

Синтаксис LINQ встроен в C#, но запросы LINQ могут использоваться во множестве разных контекстов благодаря библиотекам, называемым *поставщиками* (providers) LINQ. Поставщик LINQ представляет собой набор классов, реализующих операции LINQ и позволяющих программам взаимодействовать с источниками данных для выполнения таких операций, как сортировка, группировка и фильтрация элементов.

В этой книге мы рассматриваем поставщиков *LINQ to Entities* и *LINQ to XML*, предназначенных для запросов к базам данных и документам XML. Эти поставщики, наряду с упоминавшимся выше поставщиком *LINQ to Objects*, включены в поставку C# и .NET Framework. Также существует много других, более специализированных поставщиков для взаимодействия с конкретными веб-сайтами или форматами данных. За информацией о существующих поставщиках LINQ обращайтесь на сайт *codeplex.com* (проведите поиск по строке «LINQ providers»).

## 9.2. Выборка из массива с использованием LINQ

На ил. 9.2 продемонстрирована выборка из целочисленного массива с использованием LINQ. Команды повторения, фильтрующие содержимое массива, перебирают элементы и проверяют, удовлетворяют ли они заданному критерию. LINQ определяет условия, которым должны удовлетворять отобранные элементы. Такой стиль программирования называется *декларативным* (в отличие от императивного программирования, с которым мы имели дело до настоящего момента, когда задаются конкретные действия по выполнению нужной операции). Запрос в строках 20–22 определяет, что результаты должны состоять из всех значений `int` массива `values`, больших 4. Он не указывает, как именно будут получены эти результаты, — весь необходимый код генерируется компилятором C#, что является одним из важнейших преимуществ LINQ. Чтобы использовать LINQ to Objects, необходимо импортировать пространство имен `System.Linq` (строка 4).

```

1 // Ил. 9.2: LINQWithSimpleTypeArray.cs
2 // Использование LINQ to Objects для работы с массивом int.
3 using System;
4 using System.Linq;
5
6 class LINQWithSimpleTypeArray
7 {
8     public static void Main( string[] args )
9     {
10         // Создание целочисленного массива
11         int[] values = { 2, 9, 5, 0, 3, 7, 1, 4, 8, 5 };
12
13         // Вывод исходных значений
14         Console.Write( "Original array:" );
15         foreach ( var element in values )
16             Console.Write( " {0}", element );
17
18         // Запрос LINQ для выборки из массива значений, больших 4
19         var filtered =
20             from value in values
21             where value > 4
22             select value;
23
24         // Вывод отфильтрованных результатов
25         Console.Write( "\nArray values greater than 4:" );
26         foreach ( var element in filtered )
27             Console.Write( " {0}", element );
28
29         // Условие orderby упорядочивает выборку по возрастанию
30         var sorted =
31             from value in values
32             orderby value
33             select value;
34
35         // Вывод отсортированных результатов
36         Console.Write( "\nOriginal array, sorted:" );
37         foreach ( var element in sorted )
38             Console.Write( " {0}", element );
39
40         // Сортировка отфильтрованных результатов по убыванию
41         var sortFilteredResults =
42             from value in filtered
43             orderby value descending
44             select value;
45
46         // Вывод отсортированных результатов
47         Console.Write(
48             "\nValues greater than 4, descending order (separately):" );
49         foreach ( var element in sortFilteredResults )
50             Console.Write( " {0}", element );
51
52         // Фильтрация исходного массива и сортировка по убыванию
53         var sortAndFilter =

```

**Ил. 9.2.** Использование LINQ to Objects с массивом int (продолжение ↗)

```
54         from value in values
55         where value > 4
56         orderby value descending
57         select value;
58
59     // Вывод отфильтрованных и отсортированных результатов
60     Console.Write(
61         "\nValues greater than 4, descending order (one query):" );
62     foreach ( var element in sortAndFilter )
63         Console.Write( " {0}", element );
64
65     Console.WriteLine();
66 } // Конец Main
67 } // Конец класса LINQWithSimpleTypeArray
```

```
Original array: 2 9 5 0 3 7 1 4 8 5
Array values greater than 4: 9 5 7 8 5
Original array, sorted: 0 1 2 3 4 5 5 7 8 9
Values greater than 4, descending order (separately): 9 8 7 5 5
Values greater than 4, descending order (one query): 9 8 7 5 5
```

### Ил. 9.2. Использование LINQ to Objects с массивом int (окончание)

#### Секция from и неявная типизация локальных переменных

Запрос LINQ начинается с секции from (строка 20), определяющей диапазонную переменную (value) и источник данных, к которому обращен запрос (values). Диапазонная переменная представляет каждый отдельный элемент источника данных (по одному за раз), по аналогии с управляющей переменной в команде foreach. Тип диапазонной переменной не указывается: так как ей последовательно присваиваются отдельные элементы массива values, который является массивом int, компилятор определяет, что диапазонная переменная должна относиться к типу int. Эта особенность C#, называемая *неявной типизацией локальных переменных*, позволяет компилятору вычислить тип локальной переменной в зависимости от контекста ее использования. Включение диапазонной переменной в секцию from в начале запроса позволяет IDE применить технологию IntelliSense при написании запроса. Когда в редакторе вводится имя диапазонной переменной, за которым следует точка (.), IDE выводит список методов и свойств переменной.

#### Ключевое слово var

Вы также можете объявить локальную переменную и поручить компилятору определить ее тип по инициализатору переменной. Для этого вместо типа переменной в объявление включается ключевое слово var. Возьмем следующее объявление:

```
var x = 7;
```

Компилятор делает вывод, что переменная x должна относиться к типу int, поскольку целочисленные значения (такие, как 7) относятся к типу int. Аналогичным образом в объявлении

```
var y = -123.45;
```

компилятор делает вывод, что `y` относится к типу `double`, потому что компилятор считает, что вещественные числа (такие, как `-123.45`) относятся к типу `double`. Как правило, локальные переменные с неявной типизацией используются для более сложных типов данных — например, коллекций, возвращаемых запросами LINQ. Эта возможность используется в строках 19, 30, 41 и 53 для определения типа каждой переменной, в которой хранятся результаты запросов LINQ. Также неявная типизация используется для объявления управляющей переменной в командах `foreach` в строках 15–16, 26–27, 37–38, 49–50 и 62–63. В каждом случае компилятор считает, что управляющая переменная относится к типу `int`, потому что и значения из массива, и результаты запросов LINQ содержат данные `int`.

### Секция `where`

Если условие в секции `where` (строка 21) истинно, то элемент включается в выборку, то есть в результат запроса. В данном примере элементы `int` из массива отбираются запросом только в том случае, если они больше 4. Выражение, которое получает элемент коллекции и возвращает результат (`true` или `false`) проверки заданного условия для этого элемента, называется *предикатом*.

### Секция `select`

Для каждого элемента в источнике данных условие `select` (строка 22) определяет значение, которое должно быть включено в результат. В нашем случае это значение `int`, представляемое диапазоной переменной. Запрос LINQ обычно завершается секцией `select`.

### Перебор результатов запроса LINQ

В строках 26–27 результаты запроса выводятся командой `foreach`. Как вы знаете, команда `foreach` перебирает элементы массива, предоставляя возможность обработать каждый элемент. Однако наряду с содержимым массивов команды `foreach` также позволяют перебирать коллекции и результаты запросов LINQ. Команда `foreach` в строках 26–27 перебирает отфильтрованный результат запроса и выводит каждый элемент.

### LINQ и команды повторения

Для вывода целых чисел больших 4 также можно воспользоваться командой повторения, которая проверяет каждое значение перед выводом. Но в этом случае код выбора элементов смешивается с кодом их вывода. При использовании LINQ они отделяются друг от друга, что упрощает понимание и сопровождение приложения.

### Секция `orderby`

Секция `orderby` (строка 32) сортирует результаты запроса по возрастанию. В строках 43 и 56 в секцию `orderby` включается модификатор `descending` для сортировки результатов по убыванию. Модификатор `ascending` тоже существует, но обычно явно не включается, потому что используется по умолчанию. В секции `orderby` используется любое значение, которое может сравниваться с другими значениями

того же типа. Значения простых типов (например, `int`) всегда могут сравниваться с другими значениями того же типа; сравнение значений ссылочных типов более подробно рассматривается в главе 12.

Запросы в строках 42–44 и 54–57 выдают одинаковые результаты, но делают это разными способами. Первый запрос использует LINQ для сортировки результатов в строках 20–22. Второй запрос использует секции `where` и `orderby`. Так как запросы могут применяться к результатам других запросов, их можно строить последовательно, передавая результаты следующему методу для дальнейшей обработки.

### Подробнее о неявной типизации локальных переменных

Неявно типизованные локальные переменные также могут использоваться для инициализации массивов без явного указания типа. Например, следующая команда создает массив со значениями `int`:

```
var array = new[] { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
```

Обратите внимание на отсутствие квадратных скобок в левой части оператора присваивания и на конструкцию `new[]`, которая указывает, что переменная является массивом.

### Интерфейс `IEnumerable<T>`

Как упоминалось ранее, команда `foreach` может использоваться для перебора содержимого массивов, коллекций и результатов запросов LINQ. Кроме того, команда `foreach` поддерживает перебор в любом объекте `IEnumerable<T>`, а именно такие объекты возвращаются большинством запросов LINQ.

`IEnumerable<T>` представляет собой *интерфейс*. Интерфейсы определяют и стандартизируют взаимодействия между подсистемами. Например, элементы настройки радиоприемника образуют интерфейс между пользователем и внутренними компонентами. С их помощью пользователь может выполнить операции из ограниченного набора (переключить станцию, отрегулировать громкость и т. д.), а разные приемники могут реализовать эти средства управления по-разному (кнопки, ручки регулировки, голосовые команды). Интерфейс определяет, какую операцию может выполнить пользователь, но не указывает, как именно эта операция должна быть реализована.

Программные объекты тоже взаимодействуют друг с другом через интерфейсы. Интерфейс C# описывает набор методов, которые могут вызываться для объекта (например, чтобы приказать объекту выполнить некоторую операцию или получить от него информацию). Интерфейс `IEnumerable<T>` описывает функциональность объекта, поддерживающего возможность перебора, и предоставляет соответствующие методы. Класс, реализующий интерфейс, должен определить все члены интерфейса с сигнатурой, идентичной сигнатуре в определении интерфейса. Реализация интерфейса по сути является соглашением, которое объект заключает с компилятором: «Я обязуюсь объявить все члены, входящие в интерфейс». Интерфейсы более подробно рассматриваются в главе 12.



Массивы реализуют интерфейс `IEnumerable<T>`, поэтому команда `foreach` может перебирать элементы массива. Кроме того, запросы LINQ возвращают объекты, реализующие `IEnumerable<T>`, поэтому для перебора результатов запросов LINQ можно использовать команду `foreach`. Запись `<T>` обозначает обобщенный интерфейс, который может использоваться с любым типом данных (например, `int`, `string` или `Employee`). Механизм обобщений и запись `<T>` более подробно рассматриваются в разделе 9.4, а интерфейсы — в разделе 12.7.

## 9.3. Запрос к массиву объектов Employee с использованием LINQ

Технология LINQ не ограничивается запросами к массивам простых типов (скажем, `int`). Она также может использоваться с большинством типов данных, включая строки и классы, определяемые пользователем. С другой стороны, она неприменима, если смысл запроса не определен, — например, условие `orderby` не может использоваться с несравнимыми объектами. *Сравнимыми* типами в .NET называются типы, реализующие интерфейс `IComparable` (см. раздел 20.4). Все встроенные типы (такие, как `string`, `int` и `double`) реализуют `IComparable`. В листинге на ил. 9.3 представлен класс `Employee`, а на ил. 9.4 технология LINQ используется для работы с массивом объектов `Employee`.

```

1  // Ил. 9.3: Employee.cs
2  // Класс Employee со свойствами FirstName, LastName и MonthlySalary.
3  public class Employee
4  {
5      private decimal monthlySalaryValue; // Зарплата работника
6
7      // Автоматически реализованное свойство FirstName
8      public string FirstName { get; set; }
9
10     // Автоматически реализованное свойство LastName
11     public string LastName { get; set; }
12
13     // Конструктор инициализирует имя, фамилию и зарплату
14     public Employee( string first, string last, decimal salary )
15     {
16         FirstName = first;
17         LastName = last;
18         MonthlySalary = salary;
19     } // Конец конструктора
20
21     // Свойство для чтения и записи зарплаты работника
22     public decimal MonthlySalary
23     {
24         get
25         {
26             return monthlySalaryValue;

```

**Ил. 9.3.** Класс `Employee` (продолжение ↗)

```

27     } // Конец get
28     set
29     {
30         if ( value >= 0M ) // Если зарплата не отрицательна
31         {
32             monthlySalaryValue = value;
33         } // Конец if
34     } // Конец set
35 } // Конец свойства MonthlySalary
36
37 // Метод возвращает строку с информацией о работнике
38 public override string ToString()
39 {
40     return string.Format( "{0,-10} {1,-10} {2,10:C}",
41         FirstName, LastName, MonthlySalary );
42 } // Конец метода ToString
43 } // Конец класса Employee

```

### Ил. 9.3. Класс Employee (окончание)

```

1 // Ил. 9.4: LINQWithArrayOfObjects.cs
2 // Использование LINQ to Objects с массивом объектов Employee.
3 using System;
4 using System.Linq;
5
6 public class LINQWithArrayOfObjects
7 {
8     public static void Main( string[] args )
9     {
10         // Инициализация массива объектов Employee
11         Employee[] employees = {
12             new Employee( "Jason", "Red", 5000M ),
13             new Employee( "Ashley", "Green", 7600M ),
14             new Employee( "Matthew", "Indigo", 3587.5M ),
15             new Employee( "James", "Indigo", 4700.77M ),
16             new Employee( "Luke", "Indigo", 6200M ),
17             new Employee( "Jason", "Blue", 3200M ),
18             new Employee( "Wendy", "Brown", 4236.4M ) }; // Конец списка
19                                                         // инициализации
20
21         // Вывод информации обо всех работниках
22         Console.WriteLine( "Original array:" );
23         foreach ( var element in employees )
24             Console.WriteLine( element );
25
26         // Фильтрация по диапазону зарплат в запросе LINQ
27         var between4K6K =
28             from e in employees
29             where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
30             select e;
31
32         // Вывод работников с зарплатой в диапазоне от 4000 до 6000
33         Console.WriteLine( string.Format(
34             "\nEmployees earning in the range {0:C}-{1:C} per month:",
35             4000, 6000 ) );

```

### Ил. 9.4. Использование LINQ to Objects с массивом объектов Employee (продолжение ⤵)

```

35     foreach ( var element in between4K6K )
36         Console.WriteLine( element );
37
38     // Сортировка сначала по фамилии, затем по имени
39     var nameSorted =
40         from e in employees
41         orderby e.LastName, e.FirstName
42         select e;
43
44     // Заголовок
45     Console.WriteLine( "\nFirst employee when sorted by name:" );
46
47     // Попытка вывода первого результата предыдущего запроса LINQ
48     if ( nameSorted.Any() )
49         Console.WriteLine( nameSorted.First() );
50     else
51         Console.WriteLine( "not found" );
52
53     // Использование LINQ для выборки фамилий работников
54     var lastNames =
55         from e in employees
56         select e.LastName;
57
58     // Использование метода Distinct для выборки уникальных фамилий
59     Console.WriteLine( "\nUnique employee last names:" );
60     foreach ( var element in lastNames.Distinct() )
61         Console.WriteLine( element );
62
63     // Использование LINQ для выборки имени и фамилии
64     var names =
65         from e in employees
66         select new { e.FirstName, Last = e.LastName };
67
68     // Вывод полных имен
69     Console.WriteLine( "\nNames only:" );
70     foreach ( var element in names )
71         Console.WriteLine( element );
72
73     Console.WriteLine();
74 } // Конец Main
75 } // Конец класса LINQWithArrayOfObjects


```

Original array:

Jason	Red	\$5,000.00
Ashley	Green	\$7,600.00
Matthew	Indigo	\$3,587.50
James	Indigo	\$4,700.77
Luke	Indigo	\$6,200.00
Jason	Blue	\$3,200.00
Wendy	Brown	\$4,236.40

Employees earning in the range \$4,000.00-\$6,000.00 per month:

Jason	Red	\$5,000.00
-------	-----	------------

**Ил. 9.4.** Использование LINQ to Objects с массивом объектов Employee (продолжение )

```

James Indigo $4,700.77
Wendy Brown $4,236.40

First employee when sorted by name:
Jason Blue $3,200.00
Unique employee last names:
Red
Green
Indigo
Blue
Brown

Names only:
{ FirstName = Jason, Last = Red }
{ FirstName = Ashley, Last = Green }
{ FirstName = Matthew, Last = Indigo }
{ FirstName = James, Last = Indigo }
{ FirstName = Luke, Last = Indigo }
{ FirstName = Jason, Last = Blue }
{ FirstName = Wendy, Last = Brown }

```

**Ил. 9.4.** Использование LINQ to Objects с массивом объектов Employee (окончание)

### Обращение к свойствам диапазонной переменной запроса LINQ

В строке 28 на ил. 9.4 приведена секция `where`, в которой используются свойства диапазонной переменной. В данном примере компилятор определяет, что диапазонная переменная относится к типу `Employee`, на основании того, что массив `employees` был определен как массив объектов `Employee` (строки 11–18). В секции `where` может использоваться любое выражение `bool`. В строке 28 условия объединяются условным оператором И (`&&`). В нашем примере в результат запроса, выводимый в строках 35–36, включаются только работники с зарплатой в диапазоне от 4000 до 6000.

### Сортировка результатов запроса LINQ по нескольким свойствам

В строке 41 секция `orderby` используется для сортировки результатов по нескольким свойствам (заданным в виде списка, разделенного запятыми). В этом запросе работники сортируются в алфавитном порядке по фамилиям. Далее в пределах каждой группы работники с одинаковой фамилией сортируются по именам.

### Методы расширения Any, First и Count

В строке 48 представлен метод `Any` объекта результата, который возвращает `true` при наличии хотя бы одного элемента и `false` при отсутствии элементов. Метод `First` результата запроса (строка 49) возвращает первый элемент в результате. Перед вызовом `First` следует убедиться в том, что результат запроса не пуст (строка 48).

Мы не указали класс, определяющий методы `First` и `Any`. Вероятно, ваша интуиция подсказывает, что эти документы определяются в интерфейсе `IEnumerable<T>`, но это не так. В действительности они являются *методами расширения* — такие методы могут использоваться для расширения функциональности класса без изменения его определения. Методы расширения LINQ могут использоваться так, словно они являются методами `IEnumerable<T>`.

LINQ определяет много других методов расширения — например, метод `Count`, возвращающий количество элементов в результате запроса. Вместо вызова `Any` также можно проверить, что результат `Count` отличен от нуля, но проверка наличия хотя бы одного элемента эффективнее подсчета всех элементов. Синтаксис запроса LINQ преобразуется компилятором в цепочку вызовов методов расширения, в которой результат одного вызова метода используется другим методом. Именно такая архитектура позволяет применять запросы к результатам предыдущих запросов, так как все сводится к простой передаче результата вызова одного метода другому методу.

### Выбор свойства объекта

В строке 56 секция `select` используется для выборки свойства `LastName` диапазонной переменной (вместо самой диапазонной переменной). Это приводит к тому, что результат запроса содержит только фамилии работников (в формате `string`) вместо полных объектов `Employee`. В строках 60–61 выводятся уникальные фамилии. Метод расширения `Distinct` (строка 60) удаляет дубликаты, а в результате остаются только уникальные элементы.

### Создание новых типов в секции `select` запроса LINQ

Последний запрос LINQ в этом примере (строки 65–66) выбирает свойства `FirstName` и `LastName`. Синтаксис

```
new { e.FirstName, Last = e.LastName }
```

создает новый объект анонимного типа (то есть типа, которому не присвоено имя), сгенерированного компилятором на основании свойств в фигурных скобках (`{}`). В данном случае анонимный тип состоит из свойств имени (`FirstName`) и фамилии (`LastName`) выбранного работника. Свойство `LastName` присваивается свойству `Last` в секции `select`; так можно задать новое имя для выбранного свойства. Если новое имя не указано, используется исходное имя свойства (как для `FirstName` в данном примере). Предыдущий запрос является примером *проекции* — он преобразует выбранные данные. В данном случае преобразование создает новые объекты, содержащие только свойства `FirstName` и `Last`. Преобразования также могут выполнять дополнительную обработку данных — например, можно дать всем работникам 10%-ную прибавку, умножая их свойство `MonthlySalary` на 1.1.

При создании нового анонимного типа можно выбрать любое количество свойств, перечисляя их через запятую в фигурных скобках (`{}`), ограничивающих определение анонимного типа. В нашем примере компилятор автоматически создает новый класс со свойствами `FirstName` и `Last`, значения которых копируются из объектов `Employee`. Приложение может обратиться к выбранным свойствам во время перебора результатов. Неявная типизация локальных переменных позволяет использовать *анонимные типы*, потому что при их объявлении не нужно явно задавать тип.

Создавая анонимный тип, компилятор автоматически генерирует метод `ToString`, возвращающий строковое представление объекта. Результат его выполнения встречается в выходных данных — он состоит из имен и значений свойств, заключенных в фигурные скобки. Анонимные типы более подробно рассматриваются в главе 22.

## 9.4. Коллекции

Библиотека .NET Framework Class Library содержит набор классов *коллекций*, предназначенных для хранения групп взаимосвязанных объектов. Эти классы предоставляют эффективные методы для упорядочения, хранения и выборки данных, причем пользователю не нужно знать, как эти данные хранятся. Использование готовых коллекций сокращает время разработки приложений.

Мы уже использовали массивы для хранения серий объектов. Массивы не могут автоматически изменять свои размеры во время выполнения для размещения дополнительных элементов — это приходится делать вручную, создавая новый массив или используя метод `Resize` класса `Array`.

Класс `List<T>` (из пространства имен `System.Collections.Generic`) предоставляет удобное решение этой проблемы. Тип `T` является «заместителем» — при объявлении нового объекта `List` он заменяется конкретным типом элементов, которые должны храниться в `List`, по аналогии с указанием типа при объявлении массива. Например, строка

```
List< int > list1;
```

объявляет `list1` как коллекцию `List`, в которой могут храниться только значения `int`, а строка

```
List< string > list2;
```

объявляет `list2` как коллекцию `List` с элементами `string`. Классы с такими «заместителями», которые могут использоваться с любыми типами, называются *обобщенными классами*. Обобщенные классы, а также другие обобщенные коллекции рассматриваются в главах 20 и 21 соответственно. На ил. 9.5 представлены важнейшие методы и свойства класса `List<T>`.

Метод или свойство	Описание
<code>Add</code>	Добавляет элемент в конец списка
<code>Capacity</code>	Свойство для чтения или записи количества элементов, которые могут храниться в <code>List</code> без изменения размеров
<code>Clear</code>	Удаляет все элементы из <code>List</code>
<code>Contains</code>	Возвращает <code>true</code> , если <code>List</code> содержит заданный элемент, или <code>false</code> в противном случае
<code>Count</code>	Свойство возвращает текущее количество элементов в <code>List</code>
<code>IndexOf</code>	Возвращает индекс первого вхождения заданного значения в <code>List</code>
<code>Insert</code>	Вставляет элемент с заданным индексом
<code>Remove</code>	Удаляет первое вхождение заданного значения
<code>RemoveAt</code>	Удаляет элемент с заданным индексом
<code>RemoveRange</code>	Удаляет заданное количество элементов начиная с заданного индекса

**Ил. 9.5.** Некоторые методы и свойства класса `List<T>` (продолжение ↗)

Метод или свойство	Описание
Sort	Сортирует элементы
TrimExcess	Задаёт значение Capacity в соответствии с количеством элементов, хранящихся в List в настоящее время (Count)

**Ил. 9.5.** Некоторые методы и свойства класса List<T> (окончание)

Листинг на ил. 9.6 демонстрирует динамическое изменение размера списка (то есть объекта List). Методы Add и Insert добавляют новые элементы в List (строки 13–14). Метод Add присоединяет свой аргумент в конец List. Метод Insert вставляет новый элемент в заданную позицию. В первом аргументе передается индекс — как и в случае с массивами, индексы коллекций начинаются с 0. Второй аргумент содержит значение, вставляемое по заданному индексу. Индексы элементов, начиная с заданного, увеличиваются на 1. Обычно эта операция выполняется медленнее, чем добавление элемента в конец списка.

```

1 // Ил. 9.6: ListCollection.cs
2 // Использование обобщенной коллекции List<T>.
3 using System;
4 using System.Collections.Generic;
5
6 public class ListCollection
7 {
8     public static void Main( string[] args )
9     {
10         // Создание нового списка строк
11         List< string > items = new List< string >();
12
13         items.Add( "red" ); // Присоединение в конец списка
14         items.Insert( 0, "yellow" ); // Вставка значения по индексу 0
15
16         // Вывод цветов из списка
17         Console.Write(
18             "Display list contents with counter-controlled loop:" );
19         for ( int i = 0; i < items.Count; i++ )
20             Console.Write( " {0}", items[ i ] );
21
22         // Вывод цветов с использованием foreach
23         Console.Write(
24             "\nDisplay list contents with foreach statement:" );
25         foreach ( var item in items )
26             Console.Write( " {0}", item );
27
28         items.Add( "green" ); // Добавление "green" в конец списка
29         items.Add( "yellow" ); // Добавление "yellow" в конец списка
30
31         // Вывод содержимого List
32         Console.Write( "\nList with two new elements:" );
33         foreach ( var item in items )
34             Console.Write( " {0}", item );
35
36         items.Remove( "yellow" ); // Удаление первого вхождения "yellow"

```

**Ил. 9.6.** Использование обобщенной коллекции List (продолжение ➤)

```

37
38 // Вывод содержимого List
39 Console.Write( "\nRemove first instance of yellow:" );
40 foreach ( var item in items )
41     Console.Write( " {0}", item );
42
43 items.RemoveAt( 1 ); // Удаление элемента с индексом 1
44
45 // Вывод содержимого List
46 Console.Write( "\nRemove second list element (green):" );
47 foreach ( var item in items )
48     Console.Write( " {0}", item );
49
50 // Проверка присутствия значения в List
51 Console.WriteLine( "\n\"red\" is {0}in the list",
52     items.Contains( "red" ) ? string.Empty : "not " );
53
54 // Вывод количества элементов в коллекции List
55 Console.WriteLine( "Count: {0}", items.Count );
56
57 // Вывод емкости коллекции List
58 Console.WriteLine( "Capacity: {0}", items.Capacity );
59 } // Конец Main
60 } // Конец класса ListCollection

```

```

Display list contents with counter-controlled loop: yellow red
Display list contents with foreach statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Count: 2
Capacity: 4

```

### Ил. 9.6. Использование обобщенной коллекции List (окончание)

В строках 19–20 выводятся элементы коллекции List. Свойство Count возвращает текущее количество элементов. К элементам коллекции List можно обращаться по индексу, как к элементам списков, — для этого индекс указывается в квадратных скобках после имени переменной List. Индексированное выражение List может использоваться для изменения элемента по заданному индексу. В строках 25–26 содержимое List выводится командой foreach. В список добавляются новые элементы, после чего его содержимое выводится снова (строки 28–34). Метод Remove используется для удаления первого элемента с заданным значением (строка 36). Если в списке нет таких элементов, Remove ничего не делает. Похожий метод RemoveAt удаляет элемент в позиции с заданным индексом (строка 43). При удалении элемента одним из этих методов индексы всех элементов в последующих позициях уменьшаются на 1 — в отличие от метода Insert.

В строке 52 метод Contains используется для проверки присутствия значения в List. Метод Contains возвращает true, если элемент найден в списке, и false в противном случае. Метод последовательно сравнивает свой аргумент с каждым элементом



List, пока не будут проверены все элементы, так что для больших списков Contains работает неэффективно.

В строках 55 и 58 выводятся значения Count и Capacity. Вспомните, что свойство Count (строка 55) определяет количество элементов в списке. Свойство Capacity (строка 58) определяет, сколько элементов может вместить List без увеличения размера. При увеличении размера класс List должен создать увеличенный внутренний массив и скопировать в него все элементы. Это медленная операция, и увеличивать List при каждом добавлении элемента было бы неэффективно. Вместо этого List увеличивается только в том случае, если при добавлении нового элемента значения свойств Count и Capacity равны — то есть для нового элемента нет места.

## 9.5. Запросы к обобщенной коллекции с использованием LINQ

Технология LINQ to Objects может использоваться для создания запросов не только к массивам, но и к коллекциям List. На ил. 9.7 коллекция List с элементами string преобразуется к верхнему регистру, и в ней ищутся элементы, начинающиеся с буквы «R».

```

1 // Ил. 9.7: LINQWithListCollection.cs
2 // Использование LINQ to Objects с List< string >.
3 using System;
4 using System.Linq;
5 using System.Collections.Generic;
6
7 public class LINQWithListCollection
8 {
9     public static void Main( string[] args )
10    {
11        // Заполнение коллекции List объектами string
12        List< string > items = new List< string >();
13        items.Add( "aQua" ); // Добавить "aQua" в конец List
14        items.Add( "RuSt" ); // Добавить "RuSt" в конец List
15        items.Add( "yElLow" ); // Добавить "yElLow" в конец List
16        items.Add( "rEd" ); // Добавить "rEd" в конец List
17
18        // Преобразовать все строки к верхнему регистру; выбрать те из них,
19        var startsWithR = // которые начинаются с "R"
20            from item in items
21            let uppercaseString = item.ToUpper()
22            where uppercaseString.StartsWith( "R" )
23            orderby uppercaseString
24            select uppercaseString;
25
26        // Вывод результатов запроса
27        foreach ( var item in startsWithR )
28            Console.Write( "{0} ", item );
29    }

```

**Ил. 9.7.** Использование LINQ to Objects с List<string> (продолжение ↗)

```
30     Console.WriteLine();    // Переход на следующую строку
31
32     items.Add( "rUbY" );    // Добавить "rUbY" в конец List
33     items.Add( "SaFFRon" ); // Добавить "SaFFRon" в конец List
34
35     // Вывод обновленных результатов запроса
36     foreach ( var item in startswithR )
37         Console.Write( "{0} ", item );
38
39     Console.WriteLine(); // Переход на следующую строку
40 } // Конец Main
41 } // Конец класса LINQWithListCollection
```

```
RED RUST
RED RUBY RUST
```

### Ил. 9.7. Использование LINQ to Objects с List<string> (окончание)

В строке 21 секция `let` используется для создания новой диапазонной переменной. Такая возможность может быть полезна, если вам потребуется сохранить временный результат для последующего использования в запросе LINQ. Обычно `let` объявляет новую диапазонную переменную, которой присваивается результат выражения, примененного к исходной диапазонной переменной запроса. В нашем случае метод `ToUpper` класса `string` преобразует каждый элемент к верхнему регистру символов, после чего результат сохраняется в новой диапазонной переменной `uppercaseString`. Эта новая переменная используется в секциях `where`, `orderby` и `select`. Секция `where` (строка 22) использует метод `StartsWith` класса `string` для определения того, начинается ли `uppercaseString` с символа "R". Метод `StartsWith` проверяет с учетом регистра символов, начинается ли строка с подстроки, заданной аргументом. Если `uppercaseString` начинается с "R", метод `StartsWith` возвращает `true`, а элемент включается в результаты запроса. Для проверки более сложных критериев используются средства работы с регулярными выражениями, представленные в главе 16.

Запрос создается только один раз (строки 20–24), а перебор его результатов (строки 27–28 и 36–37) дает два разных списка цветов. Запросы LINQ выполняются только при обращении к результатам (например, при их переборе или вызове метода `Count`), но не при определении запроса. Это позволяет создать запрос однократно для последующего многократного выполнения. Любые изменения в источнике данных отражаются в результатах при каждом выполнении запроса.

Возможно, в каких-то случаях это поведение окажется нежелательным, и вы предпочтете получить результаты немедленно. В LINQ для этой цели существуют методы расширения `ToArray` и `ToList`. Эти методы выполняют запрос, для которого они вызываются, и предоставляют результат в виде массива или `List<T>` соответственно. Эти методы также повышают эффективность многократного перебора одного и того же результата, поскольку запрос выполняется только один раз.

### Инициализаторы коллекций

В C# предусмотрен удобный синтаксис инициализаторов коллекций (по аналогии с инициализаторами массивов). Например, строки 12–16 листинга на ил. 9.7 можно заменить следующей командой:

```
List< string > items =
new List< string > { "aQua", "RuST", "yElLow", "rEd" };
```

## 9.6. Итоги

В этой главе рассматривается LINQ (Language Integrated Query) — мощный механизм определения запросов к данным. Вы узнали, как отфильтровать содержимое массива или коллекции при помощи секции `where` и как отсортировать результаты с использованием секции `orderby`. Секция `select` использовалась для выборки конкретных свойств объекта, а секция `let` создавала новую диапазонную переменную для более удобной работы с запросами. Метод `StartsWith` класса `string` использовался для фильтрации строк начиная с заданного символа или последовательности символов. Мы использовали методы расширения LINQ для выполнения операций, не поддерживаемых синтаксисом запроса, — метод `Distinct` исключал из результатов дубликаты, метод `Any` проверял наличие хотя бы одного элемента в результате, а метод `First` возвращал первый элемент в результате.

Далее была рассмотрена обобщенная коллекция `List<T>`, которая предоставляет всю функциональность массивов наряду с такими полезными возможностями, как динамическое изменение размера. Мы использовали метод `Add` для добавления новых элементов в конец списка, метод `Insert` для вставки новых элементов в конкретные позиции, метод `Remove` для удаления первого вхождения заданного элемента, метод `RemoveAt` для удаления элемента с заданным индексом и метод `Contains` для определения того, содержится ли элемент в `List`. Свойство `Count` использовалось для получения текущего количества элементов, а свойство `Capacity` — для определения количества элементов, которые могут храниться в коллекции `List` без увеличения ее размера. Более сложные возможности LINQ будут представлены в следующих главах, а в главе 10 мы поближе познакомимся с классами и объектами.

## 9.7. Deitel LINQ Resource Center

Мы создали сайт LINQ Resource Center ([www.deitel.com/LINQ/](http://www.deitel.com/LINQ/)) со ссылками на дополнительную информацию, посвященную LINQ: блоги участников группы Microsoft LINQ, книги, примеры глав, списки FAQ, учебники, видеоролики, веб-касты и т. д.

# 10

## Подробнее о классах и объектах

### 10.1. Введение

В этой главе мы разберемся с созданием классов, управлением доступом к членам классов и созданием конструкторов. В частности, будет рассмотрен механизм *композиции*, позволяющий хранить в членах класса ссылки на объекты других классов. Мы снова вернемся к использованию свойств, более подробно рассмотрим статические члены классов и переменные экземпляров, доступные только для чтения; поговорим о таких аспектах разработки, как пригодность кода для повторного использования, абстракция данных и инкапсуляция. Также будут рассмотрены некоторые сопутствующие вопросы, связанные с определением классов.

### 10.2. Класс Time

#### Объявление класса Time1

Наш первый пример состоит из классов `Time1` (ил. 10.1) и `Time1Test` (ил. 10.2). Класс `Time1` представляет время суток. Метод `Main` класса `Time1Test` создает объект класса `Time1` и вызывает его методы. Вывод приложения показан на ил. 10.2. [Примечание: в C# для работы с временем и датой существует тип `DateTime`. Классы в наших примерах созданы в демонстрационных целях — вам не нужно создавать собственные типы для даты и времени.]

Класс `Time1` содержит три закрытые переменные экземпляров типа `int` (ил. 10.1, строки 7–9) — `hour`, `minute` и `second`, представляющие время в 24-часовом формате (значение `hours` лежит в диапазоне 0–23). Класс `Time1` содержит открытые методы `SetTime` (строки 13–25), `ToUniversalString` (строки 28–32) и `ToString` (строки 35–40). Они образуют *открытый интерфейс*, предоставляемый классом своим клиентам.

```

1 // ил. 10.1: Time1.cs
2 // Класс Time1 для хранения времени в 24-часовом формате.
3 using System; // Пространство имен с ArgumentOutOfRangeException
4
5 public class Time1
6 {
7     private int hour; // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11     // Новое время задается в 24-часовом формате; если значение часов,
12     // минут или секунд недействительно, выдается исключение
13     public void SetTime( int h, int m, int s )
14     {
15         // Проверка часов, минут и секунд
16         if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
17             ( s >= 0 && s < 60 ) )
18         {
19             hour = h;
20             minute = m;
21             second = s;
22         } // Конец if
23         else
24             throw new ArgumentOutOfRangeException();
25     } // Конец метода SetTime
26
27     // Преобразование в строку в 24-часовом формате (ЧЧ:ММ:СС)
28     public string ToUniversalString()
29     {
30         return string.Format( "{0:D2}:{1:D2}:{2:D2}",
31             hour, minute, second );
32     } // Конец метода ToUniversalString
33
34     // Преобразование в строку в 12-часовом формате (Ч:ММ:СС AM/PM)
35     public override string ToString()
36     {
37         return string.Format( "{0}:{1:D2}:{2:D2} {3}",
38             ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
39             minute, second, ( hour < 12 ? "AM" : "PM" ) );
40     } // Конец метода ToString
41 } // Конец класса Time1

```

**Ил. 10.1.** Объявление класса Time1 для хранения времени в 24-часовом формате

В этом примере класс Time1 не объявляет конструктор, поэтому компилятор генерирует для него конструктор по умолчанию. Каждая переменная экземпляра типа int неявно инициализируется значением 0. Когда переменные экземпляров объявляются в теле класса, они могут инициализироваться с тем же синтаксисом, что и локальные переменные.

### Метод SetTime и выдача исключений

Открытый метод SetTime (строки 13–25) получает три параметра int, которые используются для задания времени. В строках 16–17 каждый аргумент проверяется

на принадлежность диапазону допустимых значений, и если все значения находятся в своих диапазонах, в строках 19–21 значения присваиваются переменным экземпляра `hour`, `minute` и `second`. Значение `hour` (строка 13) должно быть больше или равно 0 и меньше 24, так как в 24-часовом формате часы представляются целыми числами от 0 до 23. Аналогичным образом значения `minute` и `second` должны быть больше или равны 0 и меньше 60. Если значения выходят за пределы этих диапазонов, `SetTime` выдает исключение типа `ArgumentOutOfRangeException` (строки 23–24), которое оповещает клиентский код о том, что аргументу был передан недопустимый аргумент. Как вы узнали из главы 8, исключение можно перехватить конструкцией `try...catch` и попытаться продолжить работу программы; мы сделаем это в листинге на ил. 10.2. Команда `throw` (строка 24) создает новый объект типа `ArgumentOutOfRangeException`. Круглые скобки за именем класса обозначают вызов конструктора `ArgumentOutOfRangeException`. После создания объекта исключения команда `throw` немедленно завершает метод `SetTime`, а программному коду, попытавшемуся задать некорректное время, возвращается исключение.

### Метод `ToUniversalString`

Метод `ToUniversalString` (строки 28–32) вызывается без аргументов и возвращает строку в 24-часовом формате из 6 цифр — по две цифры для часов, минут и секунд. Команда `return` (строки 30–31) использует статический метод `Format` класса `class` для возвращения строки, содержащей отформатированные значения переменных `hour`, `minute` и `second`; каждое значение состоит из двух цифр и при необходимости включает начальный 0 (спецификатор формата `D2` дополняет целое число начальными нулями, если оно содержит менее двух цифр). Метод `Format` работает аналогично форматированию `string` методом `Console.Write`, не считая того, что `Format` возвращает отформатированную строку вместо того, чтобы выводить ее в консольном окне. Отформатированная строка возвращается методом `ToUniversalString`.

### Метод `ToString`

Метод `ToString` (строки 35–40) не получает аргументов и возвращает строку в 12-часовом формате из значений `hour`, `minute` и `second`, разделенных двоеточиями, за которыми следует индикатор АМ или РМ (например, 1:27:06 PM). Как и метод `ToUniversalString`, метод `ToString` использует статический метод `Format` класса `string` для форматирования минут и секунд в виде значений из двух цифр с начальными нулями при необходимости. Строка 38 использует условный оператор (`?:`) для определения значения `hour` в строке — для значений 0 и 12 в строку включается 12, а остальные значения выводятся по 12-часовой шкале. Условный оператор в строке 39 определяет, какой из индикаторов (АМ или РМ) должен возвращаться в итоговой строке.

Вспомните, о чем говорилось в разделе 7.4: все объекты в C# содержат метод `ToString`, возвращающий строковое представление объекта. Мы решили вернуть строку с временем в 12-часовом формате. Метод `ToString` вызывается неявно при выводе значения объекта с использованием форматного элемента при вызове `Console.Write`.

Не забудьте: чтобы объект мог быть преобразован в свое строковое представление, необходимо объявить метод `ToString` с ключевым словом `override` — причина станет понятна при рассмотрении наследования в главе 11.

### Использование класса Time1

Как было показано в главе 4, каждый объявленный вами класс представляет новый тип в C#. Следовательно, после объявления класса `Time1` он может быть использован как тип в объявлениях вида

```
Time1 sunset; // В переменной sunset может храниться ссылка на объект Time1
```

Тестовый класс `Time1Test` (см. ил. 10.2) использует класс `Time1`. Строка 10 создает объект `Time1` и присваивает его локальной переменной `time`. Оператор `new` вызывает конструктор по умолчанию класса `Time1`, поскольку `Time1` не объявляет своих конструкторов. В строках 13–17 выводится время — сначала в 24-часовом формате (вызовом метода `ToUniversalString` объекта `time` в строке 14), затем в 12-часовом (явным вызовом метода `ToString` объекта `time` в строке 16); результат подтверждает, что объект `Time1` был инициализирован правильно. В строке 20 вызывается метод `SetTime` объекта `time` для изменения времени. Затем в строках 21–25 время снова выводится в обоих форматах, чтобы пользователь мог убедиться в том, что оно было задано правильно.

```
1 // Ил. 10.2: Time1Test.cs
2 // Использование объекта Time1 в приложении.
3 using System;
4
5 public class Time1Test
6 {
7     public static void Main( string[] args )
8     {
9         // Создание и инициализация объекта Time1
10        Time1 time = new Time1(); // Вызов конструктора Time1
11
12        // Вывод строковых представлений времени
13        Console.Write( "The initial universal time is: " );
14        Console.WriteLine( time.ToUniversalString() );
15        Console.Write( "The initial standard time is: " );
16        Console.WriteLine( time.ToString() );
17        Console.WriteLine(); // Вывод пустой строки
18
19        // Изменение времени и вывод обновленного времени
20        time.SetTime( 13, 27, 6 );
21        Console.Write( "Universal time after SetTime is: " );
22        Console.WriteLine( time.ToUniversalString() );
23        Console.Write( "Standard time after SetTime is: " );
24        Console.WriteLine( time.ToString() );
25        Console.WriteLine(); // Вывод пустой строки
26
27        // Попытка задания времени с недействительными значениями
28        try
```

**Ил. 10.2.** Использование объекта `Time1` в приложении (продолжение ↗)

```
29     {  
30         time.SetTime( 99, 99, 99 );  
31     } // Конец try  
32     catch ( ArgumentOutOfRangeException ex )  
33     {  
34         Console.WriteLine( ex.Message + "\n" );  
35     } // Конец catch  
36  
37     // Вывод времени после попытки задания недействительных значений  
38     Console.WriteLine( "After attempting invalid settings:" );  
39     Console.Write( "Universal time: " );  
40     Console.WriteLine( time.ToUniversalString() );  
41     Console.Write( "Standard time: " );  
42     Console.WriteLine( time.ToString() );  
43 } // Конец Main  
44 } // Конец класса Time1Test
```

```
The initial universal time is: 00:00:00  
The initial standard time is: 12:00:00 AM
```

```
Universal time after SetTime is: 13:27:06  
Standard time after SetTime is: 1:27:06 PM
```

```
Specified argument was out of the range of valid values.  
After attempting invalid settings:  
Universal time: 13:27:06  
Standard time: 1:27:06 PM
```

### Ил. 10.2. Использование объекта Time1 в приложении (окончание)

#### Вызов метода SetTime с недействительными значениями

Чтобы продемонстрировать проверку аргументов методом `SetTime`, в строке 30 метод `SetTime` вызывается с недействительными аргументами 99 (для значений `hour`, `minute` и `second`). Эта команда заключается в блок `try` (строки 28–31) на случай выдачи исключения `ArgumentOutOfRangeException` методом `SetTime`, что и произойдет на этот раз, потому что все значения аргументов недействительны. Исключение перехватывается в строках 32–35, и выводится сообщение `Message` объекта исключения. Строки 38–42 снова выводят время в обоих форматах, чтобы пользователь мог убедиться в том, что при передаче недействительных аргументов время не изменилось.

#### Объявление класса Time1

В объявлении класса `Time1` стоит обратить внимание на некоторые моменты. Переменные экземпляров `hour`, `minute` и `second` объявлены закрытыми (`private`). Представление данных, используемое внутри класса, не должно интересовать клиентов класса. Например, ничто не мешает `Time1` хранить время в формате количества секунд или количества минут и секунд от полуночи. Клиент использует те же открытые методы и свойства и получает те же результаты, ничего не зная о внутреннем формате данных.



**АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.1**

Классы упрощают программирование, потому что клиент может использовать только открытые члены, предоставляемые классом. Обычно эти члены ориентированы на интересы клиента, а не на реализацию. Клиент не знает реализацию класса и никак не связан с ней. Обычно с точки зрения клиента важно то, что делает класс, а не то, как он это делает. Клиента интересует лишь то, чтобы класс работал правильно и эффективно.

**АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.2**

Интерфейсы изменяются реже, чем реализации. При изменении реализации приходится вносить соответствующие изменения в зависящий от нее код. Скрытие реализации снижает вероятность того, что другие части приложения окажутся зависящими от подробностей реализации.

## 10.3. Управление доступом к членам

Модификаторы доступа `public` и `private` управляют доступностью переменных, методов и свойств класса (в главе 11 будет представлен модификатор доступа `protected`). Как было сказано в разделе 10.2, открытые методы нужны прежде всего для формирования клиентского представления о сервисе, предоставляемом классом (то есть об открытом интерфейсе класса). Клиент класса не обязан знать внутреннее устройство класса. По этой причине закрытые переменные, свойства и методы класса (то есть подробности реализации) недоступны клиентам класса напрямую.

На ил. 10.3 продемонстрированы закрытые переменные класса, недоступные за пределами класса. В строках 9–11 мы пытаемся напрямую обратиться к закрытым переменным экземпляров `hour`, `minute` и `second` объекта `time` класса `Time1`. При компиляции приложения компилятор сообщает, что эти закрытые члены класса недоступны. [*Примечание:* в приложении используется класс `Time1` из листинга на ил. 10.1.]

```
1 // Ил. 10.3: MemberAccessTest.cs
2 // Закрытые члены класса Time1 недоступны за пределами класса.
3 public class MemberAccessTest
4 {
5     public static void Main( string[] args )
6     {
7         Time1 time = new Time1(); // Создание и инициализация объекта Time1
8
9         time.hour = 7; // ошибка: переменная hour объявлена закрытой
10        time.minute = 15; // ошибка: переменная minute объявлена закрытой
11        time.second = 30; // ошибка: переменная second объявлена закрытой
12    } // Конец Main
13 } // Конец класса MemberAccessTest
```

**Ил. 10.3.** Закрытые члены класса `Time1` недоступны за пределами класса (продолжение ↗)

Error List					
3 Errors		0 Warnings	0 Messages	Search Error List	
	Description	File	Line	Column	Project
1	'Time1.hour' is inaccessible due to its protection level	MemberAccessTest.cs	9	12	MemberAccessTest
2	'Time1.minute' is inaccessible due to its protection level	MemberAccessTest.cs	10	12	MemberAccessTest
3	'Time1.second' is inaccessible due to its protection level	MemberAccessTest.cs	11	12	MemberAccessTest

### Ил. 10.3. Закрытые члены класса Time1 недоступны за пределами класса (окончание)

Учтите, что члены класса — свойства, методы, переменные экземпляров — не обязательно явно объявлять закрытыми. Если в объявлении члена класса не указан модификатор доступа, то по умолчанию используется закрытый доступ. Чтобы избежать путаницы, мы всегда будем явно объявлять закрытые члены.

## 10.4. Ссылка this

Каждый объект может обратиться по ссылке к самому себе при помощи ключевого слова `this` (также называемого *ссылкой this*). При вызове нестатического метода для конкретного объекта тело метода неявно использует ключевое слово `this` при обращениях к переменным экземпляров объекта и другим методам. Как будет показано в листинге 10.4, ключевое слово `this` также может использоваться явно в теле нестатических методов. В разделе 10.5 представлено более интересное применение ключевого слова `this`, а в разделе 10.9 объявляется, почему оно не может использоваться в статических методах.

А сейчас мы продемонстрируем явное и неявное использование ссылки `this` для вывода данных объекта класса `SimpleTime` (см. ил. 10.4) из метода `Main` класса `ThisTest`. Для краткости два класса объявляются в одном файле — класс `ThisTest` объявляется в строках 5–12, а класс `SimpleTime` объявляется в строках 15–48.

```

1 // Ил. 10.4: ThisTest.cs
2 // Явное и неявное использование this для обращения к членам объекта.
3 using System;
4
5 public class ThisTest
6 {
7     public static void Main( string[] args )
8     {
9         SimpleTime time = new SimpleTime( 15, 30, 19 );
10        Console.WriteLine( time.BuildString() );
11    } // Конец Main
12 } // Конец класса ThisTest
13
14 // Класс SimpleTime демонстрирует использование ссылки "this"
15 public class SimpleTime
16 {
17     private int hour;    // 0-23
18     private int minute; // 0-59

```

### Ил. 10.4. Явное и неявное использование this для обращения к членам объекта (продолжение ↗)

```

19 private int second; // 0-59
20
21 // Если имена параметров конструктора совпадают с именами
22 // переменных экземпляров, то ссылка "this" необходима
23 // для того, чтобы отличить их друг от друга.
24 public SimpleTime( int hour, int minute, int second )
25 {
26     this.hour = hour; // Инициализация переменной hour объекта "this"
27     this.minute = minute; // Инициализация переменной minute объекта "this"
28     this.second = second; // Инициализация переменной second объекта "this"
29 } // Конец конструктора SimpleTime
30
31 // Явное и неявное использование "this" для вызова ToUniversalString
32 public string BuildString()
33 {
34     return string.Format( "{0,24}: {1}\n{2,24}: {3}",
35         "this.ToUniversalString()", this.ToUniversalString(),
36         "ToUniversalString()", ToUniversalString() );
37 } // Конец метода BuildString
38
39 // Преобразование в строку в 24-часовом формате (ЧЧ:ММ:СС)
40 public string ToUniversalString()
41 {
42     // Здесь уточнение "this" не обязательно, потому что метод
43     // не содержит локальных переменных, имена которых совпадают
44     // с именами переменных экземпляра.
45     return string.Format( "{0:D2}:{1:D2}:{2:D2}",
46         this.hour, this.minute, this.second );
47 } // Конец метода ToUniversalString
48 } // Конец класса SimpleTime

```

this.ToUniversalString(): 15:30:19  
 ToUniversalString(): 15:30:19

**Ил. 10.4.** Явное и неявное использование this для обращения к членам объекта (окончание)

Класс `SimpleTime` объявляет три закрытые переменные экземпляров — `hour`, `minute` и `second` (строки 17–19). Конструктор (строки 24–29) получает три аргумента `int` для инициализации объекта `SimpleTime`. В конструкторе используются параметры с именами, совпадающими с именами переменных экземпляров класса (строки 17–19). Вообще так поступать не рекомендуется, но здесь мы сделали это намеренно, чтобы скрыть имена переменных экземпляров для демонстрации явного использования ссылки `this`. Вспомните, о чем говорилось в разделе 7.11: если метод содержит локальную переменную, имя которой совпадает с именем поля, в этом методе имя относится к локальной переменной, а не к полю. В данном случае параметр скрывает поле в области действия метода. Однако метод может явно обратиться к скрытой переменной экземпляра при помощи ссылки `this`, как показано в строках 26–28 для скрытых переменных экземпляров `SimpleTime`.

Метод `BuildString` (строки 32–37) возвращает объект `string`, созданный командой, в которой явно и неявно используется ссылка `this`. В строке 35 ссылка `this` используется явно для вызова метода `ToUniversalString`. Строка 36 использует `this`

неявно для вызова того же метода. Программисты обычно не используют `this` явно для вызова других методов текущего объекта. Кроме того, строка 46 в методе `ToUniversalString` явно использует ссылку `this` для обращения к каждой переменной экземпляра. В данном случае это делать не обязательно, потому что метод не содержит локальных переменных, скрывающих переменные экземпляров класса.



### ТИПИЧНАЯ ОШИБКА 10.1

Совпадение имен параметров или локальных переменных с именами переменных экземпляров класса часто является логической ошибкой. В таких случаях для обращения к переменной экземпляра класса следует использовать ссылку `this` — без нее имя будет относиться к параметру метода или локальной переменной.



### КАК ИЗБЕЖАТЬ ОШИБОК 10.1

Постарайтесь избежать создания имен параметров и локальных переменных, конфликтующих с именами полей. Такие конфликты часто становятся причиной коварных, трудноуловимых ошибок.

Класс `ThisTest` (строки 5–12) демонстрирует использование класса `SimpleTime`. Строка 9 создает экземпляр класса `SimpleTime` и вызывает его конструктор. Строка 10 вызывает метод `BuildString` объекта и выводит его результат.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 10.1

Для экономии памяти C# хранит в памяти только одну копию каждого метода для класса — этот метод вызывается для каждого объекта класса. С другой стороны, каждый объект содержит собственную копию переменных экземпляров (то есть нестатических переменных). Каждый нестатический метод класса неявно использует ссылку `this` для определения конкретного объекта класса, с которым он должен работать.

## 10.5. Перегруженные конструкторы

В этом разделе будет представлен класс с несколькими перегруженными конструкторами, позволяющими удобно инициализировать объекты этого класса разными способами. Чтобы создать перегруженные конструкторы, просто определите несколько конструкторов с разными сигнатурами.

### Класс `Time2` с перегруженными конструкторами


По умолчанию переменные экземпляров `hour`, `minute` и `second` класса `Time1` (см. ил. 10.1) инициализируются нулями (полночь в 24-часовом формате). Класс `Time1` не позволяет клиенту класса инициализировать объект конкретными значениями, отличными от 0. В классе `Time2` (ил. 10.5) объявляются перегруженные конструкторы. В этом приложении один конструктор вызывает другой конструктор, который, в свою очередь, вызывает `SetTime` для инициализации `hour`, `minute` и `second`.

Компилятор вызывает подходящий конструктор Time2, сопоставляя количество и тип аргументов, переданных при вызове конструктора, с количеством и типом параметров в объявлении каждого конструктора.

```

1  // Ил. 10.5: Time2.cs
2  // Объявление класса Time2 с перегруженными конструкторами.
3  using System; // Для класса ArgumentOutOfRangeException
4
5  public class Time2
6  {
7      private int hour; // 0 - 23
8      private int minute; // 0 - 59
9      private int second; // 0 - 59
10
11     // Конструктор может вызываться с 0, 1, 2 или 3 аргументами
12     public Time2( int h = 0, int m = 0, int s = 0 )
13     {
14         SetTime( h, m, s ); // Вызвать SetTime для проверки времени
15     } // Конец конструктора Time2 с тремя аргументами
16
17     // Конструктор Time2: в аргументе передается другой объект Time2
18     public Time2( Time2 time )
19     : this( time.Hour, time.Minute, time.Second ) { }
20
21     // Новое время задается в 24-часовом формате; недействительные
22     // значения компонентов обнуляются.
23     public void SetTime( int h, int m, int s )
24     {
25         Hour = h; // Задать свойство Hour
26         Minute = m; // Задать свойство Minute
27         Second = s; // Задать свойство Second
28     } // Конец метода SetTime
29
30     // Свойство для чтения и записи hour
31     public int Hour
32     {
33         get
34         {
35             return hour;
36         } // Конец get
37         set
38         {
39             if ( value >= 0 && value < 24 )
40                 hour = value;
41             else
42                 throw new ArgumentOutOfRangeException(
43                     "Hour", value, "Hour must be 0-23" );
44         } // Конец set
45     } // Конец свойства Hour
46
47     // Свойство для чтения и записи minute
48     public int Minute
49     {
50         get

```

**Ил. 10.5.** Объявление класса Time2 с перегруженными конструкторами (продолжение )

```

51     {
52         return minute;
53     } // Конец get
54     set
55     {
56         if ( value >= 0 && value < 60 )
57             minute = value;
58         else
59             throw new ArgumentOutOfRangeException(
60                 "Minute", value, "Minute must be 0-59" );
61     } // Конец set
62 } // Конец свойства Minute
63
64 // Свойство для чтения и записи second
65 public int Second
66 {
67     get
68     {
69         return second;
70     } // Конец get
71     set
72     {
73         if ( value >= 0 && value < 60 )
74             second = value;
75         else
76             throw new ArgumentOutOfRangeException(
77                 "Second", value, "Second must be 0-59" );
78     } // Конец set
79 } // Конец свойства Second
80
81 // Преобразование в строку в 24-часовом формате (ЧЧ:ММ:СС)
82 public string ToUniversalString()
83 {
84     return string.Format(
85         "{0:D2}:{1:D2}:{2:D2}", Hour, Minute, Second );
86 } // Конец метода ToUniversalString
87
88 // Преобразование в строку в 12-часовом формате (Ч:ММ:СС AM/PM)
89 public override string ToString()
90 {
91     return string.Format( "{0}:{1:D2}:{2:D2} {3}",
92         ( ( Hour == 0 || Hour == 12 ) ? 12 : Hour % 12 ),
93         Minute, Second, ( Hour < 12 ? "AM" : "PM" ) );
94 } // Конец метода ToString
95 } // Конец класса Time2

```

**Ил. 10.5.** Объявление класса Time2 с перегруженными конструкторами (окончание)

### Конструктор класса Time2 без параметров

В строках 12–15 объявляется конструктор с тремя параметрами по умолчанию. Этот конструктор также рассматривается как конструктор класса без параметров, потому что его можно вызвать без аргументов, и компилятор автоматически использует значения по умолчанию. Конструктор также может вызываться с одним

аргументом (`hour`), двумя аргументами (`hour` и `minute`) и тремя аргументами (`hour`, `minute` и `second`). Для инициализации времени конструктор вызывает `SetTime`.



### ТИПИЧНАЯ ОШИБКА 10.2

Конструктор может вызывать методы своего класса. Учтите, что переменные экземпляра к этому моменту могут оказаться неинициализированными, потому что конструктор находится в процессе инициализации объекта. Попытка использования переменных экземпляров до того, как они будут нормально инициализированы, является логической ошибкой.

### Конструктор класса `Time2`, получающий ссылку на другой объект `Time2`

В строках 18–19 объявляется конструктор `time2`, получающий ссылку на объект `time2`. В этом конструкторе значения из аргумента типа `Time2` передаются конструктору с тремя параметрами (строки 12–15) для инициализации `hour`, `minute` и `second`. В этом конструкторе ссылка `this` используется способом, допустимым только в заголовке конструктора. В строке 19 за обычным заголовком конструктора следует двоеточие (`:`) и ключевое слово `this`. Ссылка `this` используется в синтаксисе вызова методов (наряду с тремя аргументами `int`) для вызова конструктора `Time2`, получающего три аргумента `int` (строки 12–15). Конструктор передает значения свойств `Hour`, `Minute` и `Second` аргумента `time` для инициализации переменных `hour`, `minute` и `second` конструируемого объекта `time2`. В тело конструктора может быть включен дополнительный код инициализации, который выполняется после вызова другого конструктора.

### Инициализаторы конструкторов

Применение ссылки `this`, представленное в строке 19, называется *инициализатором конструктора*. Это популярный способ повторного использования кода инициализации, предоставляемого одним из конструкторов класса, вместо определения похожего кода в теле другого конструктора. Этот синтаксис упрощает сопровождение класса, потому что один конструктор использует код другого конструктора. Если нам понадобится изменить способ инициализации объектов класса `Time2`, достаточно изменить только конструктор в строках 12–15. А возможно, даже этот конструктор изменять не придется — он просто вызывает метод `SetTime` для выполнения непосредственной инициализации, так что изменения в классе могут быть локализованы в этом методе.

Строка 19 могла бы напрямую обращаться к переменным экземпляров `hour`, `minute` и `second` аргумента `time` конструктора, используя выражения `time.hour`, `time.minute` и `time.second`, — несмотря на то, что они объявляются как закрытые переменные класса `Time2`.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.3

Когда один объект класса содержит ссылку на другой объект того же класса, первый объект может обращаться ко всем данным и методам другого объекта (включая закрытые).

### Метод `SetTime` класса `Time2`

Метод `SetTime` (строки 23–28) вызывает `set`-методы новых свойств `Hour` (строки 31–45), `Minute` (строки 48–62) и `Second` (строки 65–79), которые проверяют, что переданное значение `hour` лежит в диапазоне от 0 до 23, а значения `minute` и `second` лежат в диапазоне от 0 до 59. Если значение выходит за пределы диапазона, каждый `set`-метод выдает исключение `ArgumentOutOfRangeException` (строки 42–43, 59–60 и 76–77). В нашем примере используется конструктор `ArgumentOutOfRangeException`, получающий три аргумента — имя компонента, выходящего за пределы диапазона, переданное значение и сообщение об ошибке.

### О методах, свойствах и конструкторах класса `Time2`

Обращения к свойствам `Time2` используются в теле класса. Метод `SetTime` задает значения свойств `Hour`, `Minute` и `Second` в строках 25–27, а методы `ToUniversalString` и `ToString` используют свойства `Hour`, `Minute` и `Second` в строке 85 и строках 92–93 соответственно. Эти методы могли бы обращаться к закрытым данным класса напрямую. Однако представьте, что мы решили изменить способ представления времени, заменив три значения `int` (занимающих 12 байт памяти) одним значением `int`, представляющим полное количество секунд, прошедших с полуночи (для которого достаточно всего 4 байт). При внесении такого изменения достаточно изменить только тела методов, напрямую обращающихся к закрытым данным, а конкретно свойства `Hour`, `Minute` и `Second`. Изменять тела методов `SetTime`, `ToUniversalString` и `ToString` не нужно, потому что они не обращаются к закрытым данным напрямую. Такой подход к проектированию класса снижает вероятность ошибок программирования при изменении реализации класса.

Каждый конструктор можно переписать так, чтобы в нем содержалась копия соответствующих команд из метода `SetTime`. Возможно, такое решение будет чуть более эффективным из-за устранения лишних вызовов конструктора и `SetTime`. Однако дублирование команд в нескольких методах или конструкторах усложняет изменение внутреннего представления данных класса и повышает вероятность ошибок. Если один конструктор вызывает другой конструктор (или даже напрямую вызывает `SetTime`), все изменения в реализации `SetTime` будет достаточно внести в одном месте.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.4

При реализации метода класса для обращения к закрытым данным класса рекомендуется использовать свойства класса — это упрощает сопровождение кода и снижает вероятность ошибок.

### Использование перегруженных конструкторов класса `Time2`

Класс `Time2Test` (ил. 10.6) создает шесть объектов `Time2` (строки 9–13 и 42) для вызова перегруженных конструкторов `Time2`. В строках 9–13 демонстрируется передача аргументов конструкторам `Time2`. C# вызывает перегруженный конструктор, сопоставляя количество и тип аргументов, переданных при вызове конструктора, с количеством и типом параметров в объявлении каждого конструктора. Каждая из



строки 9–12 вызывает конструктор, определяемый в строках 12–15 на ил. 10.5. Строка 9 на ил. 10.6 вызывает конструктор без аргументов, поэтому компилятор присваивает каждому из трех параметров значение по умолчанию 0. Строка 10 вызывает конструктор с одним аргументом, представляющим часы, — компилятор задает для минут и секунд значение по умолчанию 0. Строка 11 вызывает конструктор с двумя аргументами, представляющими часы и минуты, — компилятор инициализирует секунды значением по умолчанию 0. В строке 12 конструктор вызывается с заданием значений всех трех параметров. Строка 13 вызывает конструктор в строках 18–19 ил. 10.5. Строки 16–37 выводят строковое представление каждого инициализированного объекта `Time2`, чтобы подтвердить правильность их инициализации.

В строке 42 приложение пытается инициализировать `t6`, создавая новый объект `Time2` с передачей конструктору трех недействительных значений. Когда конструктор пытается использовать недействительное значение `hour` для инициализации свойства `Hour` объекта, происходит исключение `ArgumentOutOfRangeException`. Приложение перехватывает его в строке 44 и выводит свойство `Message` исключения (три последние строки в выходных данных ил. 10.6). Так как при создании объекта исключения использовался конструктор `ArgumentOutOfRangeException` с тремя аргументами, свойство `Message` исключения также включает информацию о значении, нарушающем границу диапазона.

```

1 // Ил. 10.6: Time2Test.cs
2 // Перегруженные конструкторы и инициализация объектов Time2.
3 using System;
4
5 public class Time2Test
6 {
7     public static void Main( string[] args )
8     {
9         Time2 t1 = new Time2(); // 00:00:00
10        Time2 t2 = new Time2( 2 ); // 02:00:00
11        Time2 t3 = new Time2( 21, 34 ); // 21:34:00
12        Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
13        Time2 t5 = new Time2( t4 ); // 12:25:42
14        Time2 t6; // Инициализируется позднее в программе
15
16        Console.WriteLine( "Constructed with:\n" );
17        Console.WriteLine( "t1: all arguments defaulted" );
18        Console.WriteLine( " {0}", t1.ToUniversalString() ); // 00:00:00
19        Console.WriteLine( " {0}\n", t1.ToString() ); // 12:00:00 AM
20
21        Console.WriteLine(
22            "t2: hour specified; minute and second defaulted" );
23        Console.WriteLine( " {0}", t2.ToUniversalString() ); // 02:00:00
24        Console.WriteLine( " {0}\n", t2.ToString() ); // 2:00:00 AM
25
26        Console.WriteLine(
27            "t3: hour and minute specified; second defaulted" );

```

**Ил. 10.6.** Использование перегруженных конструкторов для инициализации объектов `Time2` (продолжение ↗)

```

28     Console.WriteLine( " {0}", t3.ToUniversalString() ); // 21:34:00
29     Console.WriteLine( " {0}\n", t3.ToString() ); // 9:34:00 PM
30
31     Console.WriteLine( "t4: hour, minute and second specified" );
32     Console.WriteLine( " {0}", t4.ToUniversalString() ); // 12:25:42
33     Console.WriteLine( " {0}\n", t4.ToString() ); // 12:25:42 PM
34
35     Console.WriteLine( "t5: Time2 object t4 specified" );
36     Console.WriteLine( " {0}", t5.ToUniversalString() ); // 12:25:42
37     Console.WriteLine( " {0}", t5.ToString() ); // 12:25:42 PM
38
39     // Попытка инициализировать t6 недействительными значениями
40     try
41     {
42         t6 = new Time2( 27, 74, 99 ); // Недействительные значения
43     } // Конец try
44     catch ( ArgumentOutOfRangeException ex )
45     {
46         Console.WriteLine( "\nException while initializing t6:" );
47         Console.WriteLine( ex.Message );
48     } // Конец catch
49 } // Конец Main
50 } // Конец класса Time2Test

```

Constructed with:

```

t1: all arguments defaulted
    00:00:00
    12:00:00 AM

t2: hour specified; minute and second defaulted
    02:00:00
    2:00:00 AM

t3: hour and minute specified; second defaulted
    21:34:00
    9:34:00 PM

t4: hour, minute and second specified
    12:25:42
    12:25:42 PM

t5: Time2 object t4 specified
    12:25:42
    12:25:42 PM

```

```

Exception while initializing t6:
hour must be 0-23
Parameter name: hour
Actual value was 27.

```

**Ил. 10.6.** Использование перегруженных конструкторов для инициализации объектов Time2 (окончание)

## 10.6. Конструкторы по умолчанию и конструкторы без параметров

Каждый класс должен содержать минимум один конструктор. Вспомните, о чем говорилось в разделе 4.10: если в объявлении класса не объявлен ни один конструктор, то компилятор создает конструктор по умолчанию, вызываемый без аргументов. В разделе 11.4.1 вы узнаете, что конструктор по умолчанию неявно выполняет специальную задачу.

Компилятор не создает конструктор по умолчанию для класса, явно объявляющего хотя бы один конструктор. Если в такой ситуации вам потребуется вызвать конструктор без передачи аргументов, объявите конструктор без параметров — как сделано в строке 12 на ил. 10.5. Конструктор без параметров, как и конструктор по умолчанию, вызывается с парой пустых скобок. Конструктор без параметров `Time2` явно инициализирует объект `Time2`, передавая `SetTime` значения 0 для каждого параметра. Если убрать из объявления конструктор без параметров, то клиент класса не сможет создать объект `Time2` выражением `new Time2()`.



### ТИПИЧНАЯ ОШИБКА 10.3

Если класс содержит конструкторы, но ни один из открытых конструкторов не является конструктором без параметров, попытка вызова конструктора без параметров для инициализации объекта класса приводит к ошибке компиляции. Конструктор может вызываться без аргументов только в том случае, если класс не имеет конструкторов (в этом случае вызывается конструктор по умолчанию) или в нем определен конструктор без параметров.

## 10.7. Композиция

В полях класса могут храниться ссылки на объекты других классов. Например, объект класса `AlarmClock` должен знать текущее время и время, в которое нужно подать сигнал. Соответственно, в объект `AlarmClock` разумно включить две ссылки на объекты `Time`.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.5

Композиция, то есть хранение в классе ссылок на другие объекты, является одной из форм повторного использования кода. Класс может содержать свойство своего типа — например, класс `Person` может содержать свойство `Mom` типа `Person`.

### Класс `Date`

В нашем примере композиции будут использоваться три класса — `Date` (ил. 10.7), `Employee` (ил. 10.8) и `EmployeeTest` (ил. 10.9). Класс `Date` (см. ил. 10.7) объявляет переменные экземпляров `month` и `day` (строки 7–8) и автоматически реализуемое

свойство `Year` (строка 11) для представления даты. Конструктор получает три параметра типа `int`. Строка 17 вызывает `set`-метод свойства `Month` (строки 24–38) для проверки месяца — если значение выходит за границы диапазона допустимых значений, метод выдает исключение. В строке 18 свойство `Year` используется для инициализации года. Так как свойство `Year` реализуется автоматически, мы предполагаем, что в этом примере значение `Year` действительно. В строке 19 используется свойство `Day` (строки 41–63), которое проверяет значение дня в зависимости от текущего месяца и года (для получения которых используются свойства `Month` и `Year`). Порядок инициализации важен, поскольку `set`-метод свойства `Day` проверяет значение `day` в предположении, что значения `month` и `Year` верны. Строка 53 определяет правильность дня в зависимости от количества дней в конкретном месяце. Если день неправилен, строки 56–57 проверяют, соответствует ли день 29 февраля високосного года. В противном случае при недействительном значении `day` `set`-метод доступа выдает исключение. Строка 20 в конструкторе выводит ссылку `this` в формате `string`. Так как ссылка указывает на текущий объект `Date`, для получения строкового представления *неявно* вызывается метод `ToString` объекта (строки 66–69).

```

1  // Ил. 10.7: Date.cs
2  // Объявление класса Date.
3  using System;
4
5  public class Date
6  {
7      private int month; // 1-12
8      private int day; // 1-31 в зависимости от месяца
9
10     // Автоматически реализуемое свойство Year
11     public int Year { get; private set; }
12
13     // Конструктор : свойство Month используется для проверки месяца,
14     // а свойство Day - для проверки дня.
15     public Date( int theMonth, int theDay, int theYear )
16     {
17         Month = theMonth; // Проверка месяца
18         Year = theYear;   // Может проверять год
19         Day = theDay;     // Проверка дня
20         Console.WriteLine( "Date object constructor for date {0}", this );
21     } // Конец конструктора Date
22
23     // Свойство для чтения и записи месяца
24     public int Month
25     {
26         get
27         {
28             return month;
29         } // Конец get
30         private set // Запись в свойство невозможна за пределами класса
31         {
32             if ( value > 0 && value <= 12 ) // Проверка месяца

```

**Ил. 10.7.** Объявление класса `Date` (продолжение ↗)

```

33         month = value;
34     else // Месяц недействителен
35         throw new ArgumentOutOfRangeException(
36             "Month", value, "Month must be 1-12" );
37     } // Конец set
38 } // Конец свойства Month
39
40 // Свойство для чтения и записи дня
41 public int Day
42 {
43     get
44     {
45         return day;
46     } // Конец get
47     private set // Запись в свойство невозможна за пределами класса
48     {
49         int[] daysPerMonth = { 0, 31, 28, 31, 30, 31, 30,
50                               31, 31, 30, 31, 30, 31 };
51
52         // Проверка действительности значения дня для месяца
53         if ( value > 0 && value <= daysPerMonth[ Month ] )
54             day = value;
55         // Проверка високосного года
56         else if ( Month == 2 && value == 29 &&
57             ( Year % 400 == 0 || ( Year % 4 == 0 && Year % 100 != 0 ) ) )
58             day = value;
59         else // Недействительный день
60             throw new ArgumentOutOfRangeException(
61                 "Day", value, "Day out of range for current month/year" );
62     } // Конец set
63 } // Конец свойства Day
64
65 // Метод возвращает строку в форме месяц/день/год
66 public override string ToString()
67 {
68     return string.Format( "{0}/{1}/{2}", Month, Day, Year );
69 } // Конец метода ToString
70 } // Конец класса Date

```

**Ил. 10.7.** Объявление класса Date (окончание)

### Закрытые set-методы класса Date

В классе Date используются модификаторы доступа, которые гарантируют, что для работы с закрытыми данными клиенты класса будут использовать методы и свойства. В частности, свойства Year, Month и Day объявляют закрытые set-методы (строки 11, 30 и 47) для того, чтобы set-методы могли использоваться только членами класса. Они объявляются закрытыми по тем же причинам, по которым мы объявляем закрытыми переменные экземпляров, — для упрощения сопровождения кода и управления доступом к данным класса. Конструктор, методы и свойства класса Date продолжают пользоваться всеми преимуществами использования set-методов для проверки данных, клиенты класса должны использовать конструктор класса для инициализации данных в объекте Date. Get-методы свойств Year, Month

и `Day` неявно объявляются открытыми, потому что их свойства объявлены открытыми, — при отсутствии модификатора доступа перед `get-` или `set-`методом этот метод наследует модификатор доступа, предшествующий имени свойства.

## Класс `Employee`

Класс `Employee` (см. ил. 10.8) содержит открытые, автоматически реализуемые свойства `FirstName`, `LastName`, `BirthDate` и `HireDate`. В `BirthDate` и `HireDate` (строки 7–8) хранятся ссылки на объекты `Date`; эти свойства показывают, что в членах класса могут храниться ссылки на другие объекты. Конечно, это относится и к свойствам `FirstName` и `LastName`, в которых хранятся ссылки на `string`. Конструктор `Employee` (строки 11–18) получает четыре параметра — `first`, `last`, `dateOfBirth` и `dateOfHire`. Объекты, на которые ссылаются параметры `dateOfBirth` и `dateOfHire`, присваиваются свойствам `BirthDate` и `HireDate` объекта `Employee` соответственно. При вызове метода `ToString` класса `Employee` возвращается строка, содержащая строковые представления двух объектов `Date`. Каждая из этих строк строится *неявным* вызовом метода `ToString` класса `Date`.

```
1  // Ил. 10.8: Employee.cs
2  // Класс Employee содержит ссылки на другие объекты.
3  public class Employee
4  {
5      public string FirstName { get; private set; }
6      public string LastName { get; private set; }
7      public Date BirthDate { get; private set; }
8      public Date HireDate { get; private set; }
9
10     // Конструктор для инициализации четырех полей класса
11     public Employee( string first, string last,
12         Date dateOfBirth, Date dateOfHire )
13     {
14         firstName = first;
15         lastName = last;
16         birthDate = dateOfBirth;
17         hireDate = dateOfHire;
18     } // Конец конструктора Employee
19
20     // Преобразование Employee в строковый формат
21     public override string ToString()
22     {
23         return string.Format( "{0}, {1} Hired: {2} Birthday: {3}",
24             lastName, firstName, hireDate, birthDate );
25     } // Конец метода ToString
26 } // Конец класса Employee
```

**Ил. 10.8.** Класс `Employee` со ссылками на другие объекты

## Класс `EmployeeTest`

Класс `EmployeeTest` (см. ил. 10.9) создает два объекта `Date` (строки 9–10) для представления даты рождения и приема на работу (`birth` и `hire` соответственно). Строка 11 создает объект `Employee` и инициализирует его переменные экземпляров,

передавая конструктору две строки (имя и фамилия работника) и два объекта `Date` (даты рождения и приема на работу). В строке 13 неявно вызывается метод `ToString` класса `Employee` для вывода значений переменных экземпляров и демонстрации того, что объект был инициализирован правильно.

```

1 // Ил. 10.9: EmployeeTest.cs
2 // Пример использования композиции
3 using System;
4
5 public class EmployeeTest
6 {
7     public static void Main( string[] args )
8     {
9         Date birth = new Date( 7, 24, 1949 );
10        Date hire = new Date( 3, 12, 1988 );
11        Employee employee = new Employee( "Bob", "Blue", birth, hire );
12
13        Console.WriteLine( employee );
14    } // Конец Main
15 } // Конец класса EmployeeTest

```

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949

```

**Ил. 10.9.** Пример использования композиции

## 10.8. Уборка мусора и деструкторы

Каждый созданный вами объект использует различные системные ресурсы (например, память). Во многих языках программирования эти ресурсы резервируются до момента их явного освобождения программистом. Если все ссылки на объект, управляющий ресурсом, будут потеряны до момента явного освобождения ресурса, приложение не сможет обратиться к ресурсу и освободить его. Это явление называется *утечкой ресурсов*.

Исполнительной среде необходим механизм упорядоченного возвращения неиспользуемых ресурсов системе, предотвращающий утечку. В CLR (Common Language Runtime) используется автоматическое управление памятью: *уборщик мусора* освобождает память неиспользуемых объектов, чтобы она могла использоваться другими объектами. Когда в программе не остается ни одной ссылки на объект, объект становится кандидатом на уничтожение. Каждый объект содержит специальный метод — так называемый *деструктор*, который вызывается уборщиком мусора для выполнения завершающих действий перед тем, как память объекта будет освобождена уборщиком мусора. Деструктор выглядит как конструктор без параметров, но перед именем класса ставится тильда (~), а в заголовке отсутствует модификатор доступа. После того как уборщик мусора вызовет дескриптор объекта, объект становится пригодным для уничтожения. Память такого объекта может быть возвращена в систему уборщиком мусора.

Проблема утечки памяти, характерная для других языков без автоматического освобождения памяти (например, С и С++), в С# встречается гораздо реже, хотя и не исключается полностью в некоторых специфических ситуациях. Также возможны другие виды утечки ресурсов. Например, приложение может открыть файл на диске для изменения его содержимого. Если файл не будет закрыт, то другое приложение не сможет изменить (а возможно, даже использовать) файл до завершения того приложения, которое этот файл открыло.

Недостаток уборщика мусора заключается в том, что он не может гарантировать, что его операции будут выполнены в какой-то конкретный момент времени. Следовательно, уборщик мусора может вызвать деструктор в любое время после того, как объект станет пригодным для уничтожения, а память может быть освобождена в любое время после выполнения деструктора. Более того, может оказаться, что ни то ни другое не произойдет до завершения приложения. Программист не знает, когда именно будет вызван деструктор и будет ли он вызван вообще. По этой причине деструкторы редко применяются на практике.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.6

Класс, использующий ресурсы (например, файлы на диске), должен предоставить метод для освобождения ресурсов. Многие классы Framework Class Library предоставляют для этой цели метод `Close` или `Dispose`. В разделе 13.6 представлен метод `Dispose`, который используется во многих дальнейших примерах. Методы `Close` обычно используются с объектами, связанными с файлами (глава 17) и другими типами потоков данных.

## 10.9. Статические члены классов

Каждый объект содержит собственную копию переменных экземпляров класса, однако в некоторых ситуациях одна копия конкретной переменной должна совместно использоваться всеми объектами класса. В таких случаях используются *статические* переменные классов. Статическая переменная представляет информацию уровня класса, общую для всех объектов. Объявление статической переменной должно начинаться с ключевого слова `static`.

Давайте поясним пользу статических данных на примере. Допустим, имеется видеоигра с марсианами и другими инопланетянами. Марсиане смелеют и нападают на других инопланетян, если поблизости находятся не менее четырех других марсиан. Если же марсиан меньше пяти, каждый из них предпочитает уклониться от боя. Таким образом, каждому марсианину (`Martian`) необходимо знать значение счетчика `martianCount`. Счетчик `martianCount` можно включить в класс `Martian` в виде переменной экземпляра. В этом случае каждый объект `Martian` будет содержать собственную копию счетчика, и при каждом создании нового объекта необходимо обновлять счетчик во всех существующих объектах `Martian`. Такое решение расходует лишнюю память на хранение ненужных копий, лишнее время — на обновление отдельных копий, и отличается меньшей надежностью. Вместо этого



счетчик `martianCount` объявляется как статическая переменная, то есть существует на уровне класса. Каждый объект `Martian` может обратиться к `martianCount`, но при этом в памяти хранится только одна копия счетчика; так достигается экономия памяти. Конструктор `Martian` увеличивает статическую переменную `martianCount` — ее единственную копию, а не множество отдельных копий `martianCount` в разных объектах `Martian`; так ускоряется работа программы.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.7

Используйте статическую переменную, если все объекты класса должны совместно использовать одну копию переменной.

Область действия статической переменной ограничивается телом класса. Для обращения к открытым статическим членам класса их имена уточняются именем класса и оператором «точка» (`.`), как в случае константы `Math.PI`. Закрытые статические члены класса доступны только в методах и свойствах класса. Статические члены классов существуют даже тогда, когда не существует ни один объект класса, — они становятся доступными сразу же после загрузки класса в память во время выполнения. Для работы с закрытым статическим членом класса за его пределами необходимо использовать открытый статический метод или свойство.



#### ТИПИЧНАЯ ОШИБКА 10.4

Попытка обращения к статическому члену класса с указанием экземпляра класса (по аналогии с нестатическими членами) приводит к ошибке компиляции.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.8

Статические переменные, методы и свойства существуют и могут использоваться даже в том случае, если ни один объект этого класса еще не был создан.

### Класс `Employee`

В нашем следующем приложении объявляются два класса — `Employee` (ил. 10.10) и `EmployeeTest` (ил. 10.11). Класс `Employee` объявляет закрытое статическое автоматически реализуемое свойство `Count`. `Set`-метод `Count` объявляется закрытым, потому что мы не хотим, чтобы клиент класса мог изменить значение свойства. Компилятор автоматически создает закрытую статическую переменную, которой будет управлять свойство `Count`. Если статическая переменная не инициализирована, компилятор присваивает ей значение по умолчанию — в этом случае статическая переменная автоматически реализуемого свойства `Count` инициализируется `0` (значение по умолчанию для типа `int`). В свойстве `Count` хранится счетчик количества созданных объектов класса `Employee`.

При существующих объектах `Employee` свойство `Count` может использоваться в любом методе объекта `Employee` — в нашем примере `Count` увеличивается в конструкторе

(см. ил. 10.10, строка 22). Клиентский код обращается к `Count` при помощи выражения `Employee.Count`, которое возвращает количество созданных объектов `Employee`.

```
1 // Ил. 10.10: Employee.cs
2 // Статическая переменная используется для хранения количества
3 // созданных объектов Employee.
4 using System;
5
6 public class Employee
7 {
8     public static int Count { get; private set; } // Объекты в памяти
9
10    // Автоматическое свойство FirstName доступно только для чтения
11    public string FirstName { get; private set; }
12
13    // Автоматическое свойство LastName доступно только для чтения
14    public string LastName { get; private set; }
15
16    // Инициализация Employee, увеличение статического счетчика на 1
17    // и вывод сообщения о вызове конструктора
18    public Employee( string first, string last )
19    {
20        FirstName = first;
21        LastName = last;
22        ++Count; // Увеличение статического счетчика объектов Employee
23        Console.WriteLine( "Employee constructor: {0} {1}; Count = {2}",
24            FirstName, LastName, Count );
25    } // Конец конструктора Employee
26 } // Конец класса Employee
```

**Ил. 10.10.** Статическое свойство используется для хранения счетчика созданных объектов `Employee`

### Класс `EmployeeTest`

Метод `Main` класса `EmployeeTest` (см. ил. 10.11) создает два объекта `Employee` (строки 14–15). При вызове конструктора каждого объекта в строках 20–21 (см. ил. 10.10) переданные значения `first` и `last` задаются свойствам `FirstName` и `LastName`. Эти две команды *не создают* копий исходных аргументов `string`. Объекты `string` в C# неизменяемы, то есть они не могут модифицироваться после создания. Таким образом, существование многих ссылок на одну строку безопасно (с объектами других классов дело обычно обстоит иначе). Логично спросить: если объекты `string` неизменяемы, как мы используем операторы `+` и `+=` для конкатенации? Дело в том, что конкатенация приводит к созданию новых объектов `string`, содержащих объединенные данные. Исходные объекты при этом *не* изменяются.

```
1 // Ил. 10.11: EmployeeTest.cs
2 // Использование статических переменных.
3 using System;
4
5 public class EmployeeTest
```

**Ил. 10.11.** Использование статических членов класса (продолжение ↗)

```

6 {
7     public static void Main( string[] args )
8     {
9         // Перед созданием объекта Employee значение Count равно 0
10        Console.WriteLine( "Employees before instantiation: {0}",
11            Employee.Count );
12
13        // Создание двух объектов Employee
14        Employee e1 = new Employee( "Susan", "Baker" );
15        Employee e2 = new Employee( "Bob", "Blue" );
16
17        // После создания двух объектов Employee значение Count равно 2
18        Console.WriteLine( "\nEmployees after instantiation: {0}",
19            Employee.Count );
20
21        // Получение имен Employee
22        Console.WriteLine( "\nEmployee 1: {0} {1}\nEmployee 2: {2} {3}\n",
23            e1.FirstName, e1.LastName,
24            e2.FirstName, e2.LastName );
25
26        // В нашем примере на каждый объект Employee существует только
27        // одна ссылка, поэтому следующие команды заставляют CLR пометить
28        // каждый объект Employee как пригодный для уборки мусора.
29        e1 = null; // Объект по ссылке e1 помечается как неиспользуемый
30        e2 = null; // Объект по ссылке e2 помечается как неиспользуемый
31    } // Конец Main
32 } // Конец класса EmployeeTest

```

```

Employees before instantiation: 0
Employee constructor: Susan Baker; Count = 1
Employee constructor: Bob Blue; Count = 2

```

```

Employees after instantiation: 2

```

```

Employee 1: Susan Baker
Employee 2: Bob Blue

```

### Ил. 10.11. Использование статических членов класса (окончание)

В строках 18–19 на ил. 10.11 выводится обновленное значение `Count`. Когда `Main` завершает работу с двумя объектами `Employee`, ссылкам `e1` и `e2` присваивается `null` в строках 29–30, поэтому они больше не ссылаются на объекты, созданные в строках 14–15. Объекты становятся кандидатами на уничтожение, потому что на них не осталось ни одной ссылки в приложении. После вызова деструкторов объекты становятся кандидатами на уборку мусора.

Со временем уборщик мусора может освободить память этих объектов (или операционная система освободит память при завершении приложения). C# не гарантирует, когда отработает уборщик мусора (и отработает ли вообще). Может оказаться, что при выполнении уборщик мусора освободит память лишь подмножества объектов-кандидатов (или не освободит ее вообще).

Метод, объявленный статическим, не может обращаться к нестатическим членам напрямую, потому что статический метод может быть вызван, даже если у класса нет

существующих объектов. По той же причине ссылка `this` не может использоваться в статическом методе — она должна ссылаться на конкретный объект класса, а при вызове статического метода таких объектов может *не быть* в памяти.

## 10.10. Ключевое слово `readonly`

Принцип наименьших привилегий играет основополагающую роль в качественном проектировании программного обеспечения. Этот принцип утверждает, что код должен получать привилегии и уровень доступа, необходимый для выполнения поставленной задачи, — но не более того. Давайте посмотрим, как этот принцип действует в отношении переменных экземпляров.

Одни переменные экземпляров должны быть доступны для изменения, другие нет. В разделе 8.4 мы использовали ключевое слово `const` для объявления констант. Эти константы должны инициализироваться константным значением при объявлении. Но допустим, мы хотим инициализировать в конструкторе объекта константу, принадлежащую конкретному объекту класса. В C# существует ключевое слово `readonly`, которое означает, что переменная экземпляра объекта не может изменяться, а любая попытка ее изменения после конструирования объекта является ошибкой. Например, команда

```
private readonly int INCREMENT;
```

объявляет `readonly`-переменную экземпляра `INCREMENT` типа `int`. Имена переменных с ключевым словом `readonly`, как и имена констант, принято записывать прописными буквами. И хотя `readonly`-переменные могут инициализироваться при объявлении, это не обязательно. Инициализация таких переменных должна выполняться во всех конструкторах классов. Каждый конструктор может присваивать значения `readonly`-переменной несколько раз — переменная становится неизменяемой только после завершения конструктора. Если конструктор не инициализирует `readonly`-переменную, переменной присваивается такое же значение по умолчанию, как для других переменных экземпляров (0 для простых числовых типов, `false` для типа `bool` и `null` для ссылочных типов), а компилятор выдает предупреждение.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.9

Объявление переменной экземпляра с ключевым словом `readonly` способствует соблюдению принципа наименьших привилегий. Если переменная экземпляра не должна изменяться после конструирования объекта, объявите ее с ключевым словом `readonly`, чтобы запретить модификацию.

Значения членов, объявленных с ключевым словом `const`, должны присваиваться на стадии компиляции. Это означает, что для инициализации могут использоваться только другие константные значения: целые числа, строковые литералы, символы и т. д. Константные члены классов со значениями, которые не могут быть определены на стадии компиляции, объявляются с ключевым словом `readonly` для

инициализации на стадии выполнения. `readonly`-переменные могут инициализироваться более сложными выражениями — например, инициализаторами массивов или вызовами методов, возвращающими значение или ссылку на объект.



#### **ТИПИЧНАЯ ОШИБКА 10.5**

Попытка изменения `readonly`-переменной экземпляра в любом месте, кроме объявления или конструктора объекта, является ошибкой компиляции.



#### **КАК ИЗБЕЖАТЬ ОШИБОК 10.2**

Попытки изменения `readonly`-переменных экземпляров обнаруживаются во время компиляции, не приводя к ошибкам стадии выполнения. Научные исследования показали, что исправление ошибок на стадии выполнения обходится существенно дороже.



#### **АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.10**

Если `readonly`-переменная экземпляра инициализируется константой только в объявлении, включать ее отдельную копию в каждый объект класса не обязательно. Вместо этого лучше воспользоваться ключевым словом `const`. Константы, объявленные с ключевым словом `const`, неявно считаются статическими, поэтому во всем классе будет существовать только одна копия такой константы.

## 10.11. Абстракция данных и инкапсуляция

Классы обычно скрывают подробности своей реализации от клиентов. Это называется *сокрытием информации* (information hiding). Возьмем структуру данных стека, представленную в разделе 7.6. Вспомните, что стек является структурой данных LIFO (Last In, First Out) — последний элемент, занесенный в стек, первым извлекается из него.

### **Абстракция данных**

Стеки реализуются на базе массивов и других структур данных — например, связанных списков. (Стеки и связанные списки рассматриваются в главах 19 и 21.) Клиента класса стека не интересует, как реализован стек. Клиент знает лишь то, что данные, помещенные в стек, извлекаются из него в порядке LIFO. Клиента интересует функциональность, предоставляемая стеком, а не то, как эта функциональность реализована. Эта концепция называется абстракцией данных. Даже если вам известны подробности реализации класса, не пишите код, который зависит от этих подробностей, потому что позднее они могут измениться. В этом случае отдельный класс (например, класс реализации стека и его операций занесения и извлечения) может быть заменен другой версией (например, работающей быстрее или расходующей меньше памяти), и эта замена не отразится на остальных компонентах системы. Пока открытый интерфейс класса остается неизменным (то

есть в новом объявлении класса все исходные методы сохраняют прежние имена, типы возвращаемых значений и списки параметров), изменения реализации не нарушат работоспособности других компонентов.

### Абстрактные типы данных

В ранних языках программирования (таких, как C) центральное место занимали действия. Данные в таких языках существовали для поддержки действий, выполняемых приложением; они были «примитивными» и «менее интересными», чем действия. В них существовал ограниченный набор простых типов, а программистам было относительно трудно создавать собственные типы. В C# и объектно-ориентированном стиле программирования данные играют более важную роль. Основными направлениями работы в объектно-ориентированном программировании и C# является создание типов (то есть классов) и выражение взаимодействий между объектами этих типов. Для создания языков, ориентированных на работу с данными, нужно было формализовать некоторые понятия, относящиеся к данным. Здесь мы рассмотрим понятие абстрактных типов данных (ADT, Abstract Data Types), совершенствующих процесс разработки приложений.

Рассмотрим тип `int`, который обычно ассоциируется с целыми числами в математике. В отличие от математических целых чисел, компьютерные данные `int` имеют ограниченный размер. Тип `int` в C# ограничивается диапазоном значений от  $-2\,147\,483\,648$  до  $+2\,147\,483\,647$ . Если результат вычислений выходит за пределы этого диапазона, происходит ошибка, на которую компьютер как-то реагирует — он может выдать неверный результат или сгенерировать исключение `OverflowException`. (О том, что делать при переполнении, будет рассказано в разделе 13.8.) У математических целых чисел этой проблемы не существует. Таким образом, компьютерный тип `int` является лишь приближенным представлением для математических целых чисел. Простые типы (такие, как `int`, `double` и `char`) являются примерами абстрактных типов данных — представлений математических концепций до некоторого уровня точности в компьютерной системе.

Абстрактный тип данных в действительности совмещает два понятия: представление данных и операции, которые могут выполняться с этими данными. Например, в C# тип `int` содержит целочисленное значение (данные) и поддерживает операции сложения, вычитания, умножения, деления и вычисления остатка (результат деления на ноль не определен).



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 10.11

Для создания типов программисты используют механизм классов. Язык легко расширяется введением новых типов, но базовый язык изменить нельзя.

### Очередь как абстрактный тип данных

Еще один абстрактный тип данных — *очередь* — широко используется во внутренней реализации компьютерных систем. Очередь предоставляет своим клиентам четко определенное поведение: клиент последовательно помещает элементы в очередь,

а затем так же последовательно извлекает их. Очередь возвращает элементы в порядке FIFO (First In, First Out), то есть элемент первым поставленный в очередь, первым извлекается из нее. Концептуально размер очереди не ограничен, но реальные очереди имеют конечный размер.

Очередь скрывает внутреннее представление данных, в котором отслеживается текущее количество элементов в очереди. Клиента не интересует реализация очереди — он просто рассчитывает на то, что очередь работает «как обещано». Когда клиент ставит в очередь новый элемент, очередь должна принять этот элемент и поместить его в некую внутреннюю структуру данных FIFO. Аналогичным образом, когда клиент хочет получить следующий элемент, очередь должна извлечь элемент из своего внутреннего представления и выдать его клиенту в порядке FIFO — следующая операция извлечения должна вернуть элемент, который был помещен в очередь ранее остальных.

Абстрактный тип данных очереди гарантирует целостность внутренней структуры данных. Клиенты не могут работать с этой структурой данных напрямую — она доступна только для очереди. Клиентам разрешается выполнять только допустимые операции с представлением данных; очередь отвергает операции, не поддерживаемые ее открытым интерфейсом. Стеки и очереди более подробно рассматриваются в главе 19.

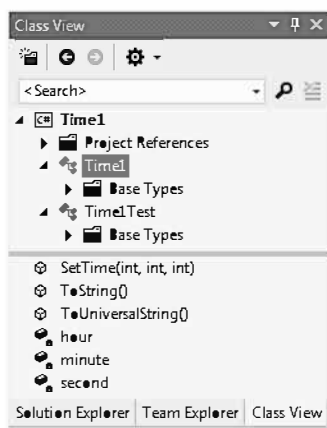
## 10.12. Окно Class View и Object Browser

После знакомства с ключевыми концепциями объектно-ориентированного программирования мы рассмотрим два инструмента Visual Studio, упрощающих разработку объектно-ориентированных приложений, — окна **Class View** и **Object Browser**.

### Окно Class View

В окне **Class View** выводятся поля, методы и свойства классов проекта. Выполните команду **VIEW ▸ Class View**, чтобы открыть **Class View** на вкладке в одной позиции с **Solution Explorer**. На ил. 10.12 изображено окно **Class View** для классов проекта **Time1: time1** (см. ил. 10.1) и **time1Test** (см. ил. 10.2). В представлении используется иерархическая структура: имя проекта (**Time1**) находится в корне, далее следует серия узлов, представляющих классы, переменные, методы и свойства проекта. Если слева от узла располагается знак ▷, этот узел можно развернуть для отображения других узлов. Если слева от узла находится знак ▲, значит, узел может быть свернут. Из окна **Class View** на ил. 10.12 видно, что классы **Time1** и **time1Test** являются дочерними узлами класса проекта **Time1**. При выделении класса **time1** в нижней половине окна выводится список его членов. Класс **time1** содержит методы **SetTime**, **ToString** и **ToUniversalString** (обозначенные фиолетовыми кубиками ☼) и переменные экземпляров **hour**, **minute** и **second** (обозначенные вот так ☼). Изображение замка справа от синих кубиков переменных экземпляров показывает, что переменные являются закрытыми. Оба класса, **Time1** и **time1Test**, содержат узел **Base Types**. Если

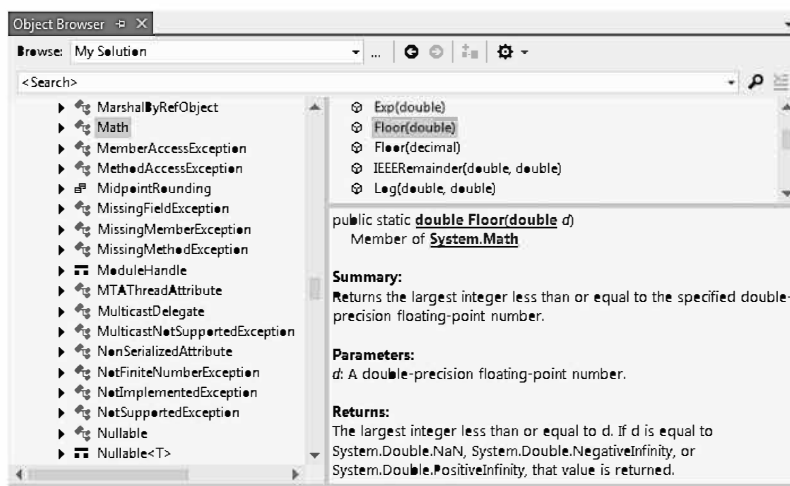
раскрыть этот узел, в обоих случаях выводится класс `Object`, потому что каждый класс наследует от класса `System.Object` (см. главу 11).



**Ил. 10.12.** Окно Class View для классов `Time1` (см. ил. 10.1) и `Time1Test` (см. ил. 10.2)

## Использование Object Browser

В окне Object Browser можно получить информацию о функциональности любого класса библиотеки .NET. Чтобы открыть окно Object Browser, выполните команду **VIEW ▸ Object Browser**. На ил. 10.13 изображено окно Object Browser, в котором был открыт класс `Math` из пространства имен `System`. Для этого мы развернули узел `mscorlib` (Microsoft Core Library) на левой верхней панели Object Browser, а затем развернули подузел `System`. [Примечание: большинство классов пространства имен `System`, включая `System.Math`, находятся в библиотеке `mscorlib`.]



**Ил. 10.13.** Окно Object Browser для класса `Math`



На правой верхней панели окна **Object Browser** выводится список всех методов, предоставляемых классом **Math**, — вы получаете «мгновенный доступ» к информации, относящейся к функциональности различных объектов. Если щелкнуть на имени члена класса на правой верхней панели, справа внизу появляется соответствующее описание. В окне **Object Browser** можно быстро получить нужную информацию о классе или одном из его методов. Для получения полного описания класса или метода в электронной документации выберите тип или член класса в **Object Browser** и нажмите F1.

## 10.13. Инициализаторы объектов

В Visual C# поддерживаются *инициализаторы объектов*, позволяющие создать объект и инициализировать его открытые свойства (и открытые переменные экземпляров, если они имеются) в одной команде. Такая возможность может быть полезна, если класс не предоставляет конструктора, подходящего для ваших целей, но у него имеются свойства для работы с данными класса. Следующий фрагмент демонстрирует использование инициализаторов объектов на примере класса **Time2** из ил. 10.5.

```
// Создание объекта Time2 с инициализацией всех свойств
Time2 aTime = new Time2 { Hour = 14, Minute = 30, Second = 12 };
// Создание объекта Time2 с инициализацией только свойства Minute
Time2 anotherTime = new Time2 { Minute = 45 };
```

Первая команда создает объект **Time2** (**aTime**), инициализирует его конструктором класса **Time2**, который может вызываться без аргументов, после чего использует инициализаторы объекта для задания свойств **Hour**, **Minute** и **Second**. Обратите внимание: за выражением **newTime2** немедленно следует список инициализаторов объекта — разделенный запятыми список свойств и их значений, заключенный в фигурные скобки (**{}**). Каждое имя свойства может упоминаться в списке не более одного раза. Инициализация объекта выполняет инициализаторы свойств в порядке их следования.

Вторая команда создает другой объект **Time2** (**anotherTime**), инициализирует его конструктором класса **Time2**, который может вызываться без аргументов, после чего задает только значение свойства **Minute** с использованием инициализатора. При вызове конструктора **Time2** без аргументов поля объекта инициализируются нулями, после чего инициализатор объекта присваивает каждому свойству из списка указанное значение. В нашем примере свойству **Minute** присваивается значение 45. Свойства **Hour** и **Second** сохраняют значения по умолчанию, потому что в списке инициализаторов для них не указаны значения.

## 10.14. Итоги

В этой главе были рассмотрены дополнительные концепции классов. Пример с классом **Time** представил полное объявление класса с закрытыми данными,

перегруженными открытыми конструкторами для более гибкого выполнения инициализации, свойствами для манипуляций с данными, а также методами для получения строковых представлений объектов `Time` в двух разных форматах. Вы узнали, что каждый класс может объявить метод `ToString` для получения представления объекта класса в формате `string`; этот метод неявно вызывается при выводе объекта класса в виде строки или его конкатенации со строкой.

Также мы рассмотрели ссылку `this`, которая неявно используется в нестатических методах класса для обращения к переменным экземпляров класса и вызова других нестатических методов. Мы рассмотрели примеры явного использования ссылки `this` для обращения к членам класса (включая скрытые поля) и научились использовать ключевое слово `this` в конструкторе для вызова другого конструктора этого класса. Вы узнали, что композиция основана на включении в класс ссылок на объекты других классов. Далее был рассмотрен механизм уборки мусора C# и особенности освобождения памяти неиспользуемых объектов. Мы выяснили, для чего нужны статические переменные класса и как объявлять и использовать статические переменные и методы в ваших классах. Также была рассмотрена тема объявления и инициализации переменных с ключевым словом `readonly`.

Вы познакомились с окнами `Visual Studio Class View` и `Object Browser`, предназначенными для получения информации о классах `Framework Class Library` и классах ваших приложений. В завершение главы рассматривается тема инициализации свойств при создании объекта с использованием списка инициализаторов.

# 11 Наследование

## 11.1. Введение

В этой главе будет рассмотрен один из основных механизмов объектно-ориентированного программирования (ООП) — *наследование*. Это форма повторного использования программного кода, при которой создаваемый класс поглощает все члены существующего класса, дополняя их новыми или видоизмененными возможностями. Наследование ускоряет процесс разработки за счет использования проверенного, качественного кода. Также применение наследования повышает вероятность того, что система будет реализована эффективно.

Существующий класс, члены которого наследуются новым классом, называется *базовым классом*, а новый класс называется *производным*. Каждый производный класс может стать базовым классом для будущих производных классов.

Производный класс обычно дополняется собственными полями и методами. Таким образом, он получается более специализированным, чем базовый класс, и представляет более специализированную группу объектов. Как правило, производный класс обладает поведением базового класса и дополнительным поведением, присущим только ему.

*Прямым базовым классом* называется базовый класс, который является непосредственным «родителем» производного класса. *Косвенным базовым классом* называется любой другой класс, находящийся выше прямого базового класса в иерархии классов, определяющей отношения наследования между классами. Иерархия классов начинается с класса `object` (синоним C# для класса `System.Object` из Framework Class Library), производными от которого (прямо или косвенно) являются все остальные классы. В C# поддерживается только модель одиночного наследования, при котором класс может иметь только один прямой базовый класс. В главе 12 будет показано, как использовать интерфейсы для реализации многих преимуществ множественного наследования, избегая присущих ему проблем.

Опыт построения программных систем показывает, что значительные объемы кода пишутся для решения частных проблем, между которыми имеется много общего. Если заниматься только частными случаями, подробности могут скрыть общую перспективу. Методология объектно-ориентированного программирования

помогает сосредоточиться на общих характеристиках объектов системы вместо частных случаев.

В области проектирования различаются связи «является частным случаем» (is-a) и «содержит» (has-a). Отношения первого вида представляют наследование; объект производного класса может интерпретироваться как объект базового класса. Например, и грузовик и легковая машина являются транспортным средством. С другой стороны, отношения типа «содержит» представляют композицию. В таких отношениях в членах объекта содержатся ссылки на другие объекты — например, объект машины содержит ссылку на объект педали газа.

Новые классы могут создаваться производными от классов из библиотек. Организации разрабатывают собственные библиотеки классов и пользуются уже готовыми библиотеками. Когда-нибудь большая часть новых программных продуктов будет создаваться из стандартизированных компонентов, предназначенных для повторного использования, — подобно тому, как проектируется большинство современных машин и компьютерных устройств. Такой подход упростит разработку более функциональных и экономичных программ.

## 11.2. Базовые и производные классы

Часто объект одного класса также может рассматриваться как объект другого класса. Например, в геометрии прямоугольник одновременно является четырехугольником (как квадрат, параллелограмм или трапеция). В этом контексте класс четырехугольника `Quadrilateral` является базовым классом, а класс прямоугольника `Rectangle` — производным. Прямоугольник является конкретной разновидностью четырехугольника, но было бы неправильно утверждать, что любой четырехугольник является прямоугольником, — он также может оказаться параллелограммом или другой фигурой. На ил. 11.1 представлены некоторые простые примеры базовых и производных классов; базовые классы обычно имеют более общий характер, а производные классы более специализированы.

Базовый класс	Производный класс
<code>Student</code>	<code>GraduateStudent</code> , <code>UndergraduateStudent</code>
<code>Shape</code>	<code>Circle</code> , <code>Triangle</code> , <code>Rectangle</code>
<code>Loan</code>	<code>CarLoan</code> , <code>HomeImprovementLoan</code> , <code>MortgageLoan</code>
<code>Employee</code>	<code>Faculty Staff</code> <code>HourlyWorker</code> <code>CommissionWorker</code>
<code>BankAccount</code>	<code>CheckingAccount</code> , <code>SavingsAccount</code>

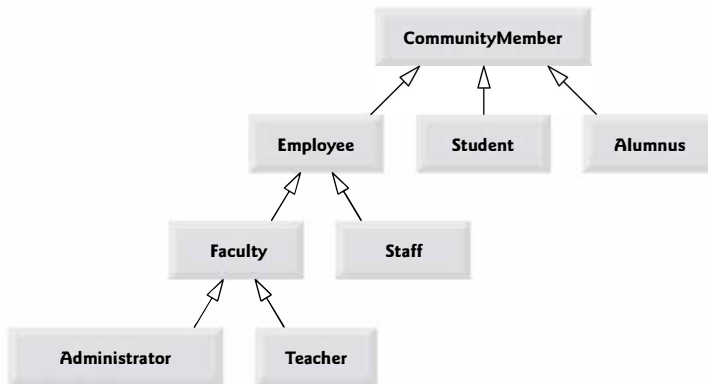
**Ил. 11.1.** Примеры наследования

Так как каждый объект производного класса одновременно может рассматриваться как объект базового класса, а один базовый класс может иметь несколько производных классов, набор объектов, представляемых базовым классом, обычно шире набора объектов, представляемых любым из его производных классов. Например,

базовый класс `Vehicle` представляет любые виды транспорта — машины, грузовики, велосипеды, лодки и т. д. С другой стороны, производный класс `Car` представляет меньшее, более конкретное подмножество транспортных средств.

Отношения наследования образуют древовидные иерархические структуры (ил. 11.2 и 11.3). Базовый класс существует в системе иерархических отношений со своими производными классами. Классы в иерархиях наследования определенным образом связываются с другими классами: они становятся либо базовыми классами, поставляющими свои члены другим классам, либо производными классами, наследующими члены от других классов. Многие классы являются одновременно базовыми и производными.

Давайте разработаем простую иерархию классов, также называемую иерархией наследования (см. ил. 11.2). На диаграмме классов UML на ил. 11.2 изображена структура университетского сообщества, включающего работников (`Employee`), студентов (`Student`) и выпускников (`Alumnus`). Работники делятся на преподавателей (`Faculty`) и прочий персонал (`Staff`). Преподаватели делятся на администраторов (деканы, заведующие кафедрами и т. д.) и лекторов. Иерархия может содержать много других классов; например, в иерархии студентов могут дополнительно выделяться бакалавры, аспиранты и т. д.

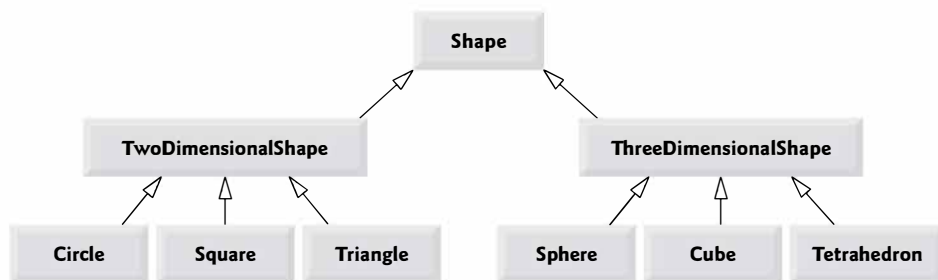


**Ил. 11.2.** Диаграмма классов UML с иерархией наследования университетского сообщества

Стрелки в виде пустых треугольников на иерархической диаграмме представляют отношения типа «является частным случаем». Например, по стрелкам можно сделать вывод о том, что работник (`Employee`) является членом сообщества (`CommunityMember`) и т. д. Класс `CommunityMember` является прямым базовым классом для `Employee`, `Student` и `Alumnus` и косвенным базовым классом для всех остальных классов на диаграмме.

Теперь рассмотрим иерархию геометрических фигур на ил. 11.3, начинающуюся с базового класса `Shape`. Этот класс расширяется производными классами `TwoDimensionalShape` и `ThreeDimensionalShape` — каждая фигура (`Shape`) является либо двумерной (`TwoDimensionalShape`), либо трехмерной (`ThreeDimensionalShape`). На третьем уровне иерархии находятся конкретные разновидности `TwoDimensionalShape`

и `ThreeDimensionalShape`. Поднимаясь по стрелкам от нижних классов к корневому базовому классу, можно отследить отношения «является частным случаем». Например, треугольник (`Triangle`) является частным случаем двумерной фигуры (`TwoDimensionalShape`) и фигуры (`Shape`), а сфера (`Sphere`) является частным случаем трехмерной фигуры (`ThreeDimensionalShape`) и фигуры (`Shape`). Эта иерархия может содержать много других классов: эллипсы, трапеции и т. д.



**Ил. 11.3.** Диаграмма классов UML с иерархией наследования геометрических фигур

Не все отношения между классами являются отношениями наследования. В главе 10 были рассмотрены отношения типа «содержит», при которых в полях класса хранятся ссылки на объекты других классов. При таких отношениях новые классы создаются посредством композиции существующих классов. Например, было бы неправильно сказать, что класс работника (`Employee`) является частным случаем класса даты рождения (`BirthDate`) — эти два класса связаны отношениями типа «содержит».

Объекты базовых и производных классов могут обрабатываться аналогичным образом — сходство между ними выражается в членах базового класса. Объекты всех классов, расширяющих общий базовый класс, могут рассматриваться как объекты этого базового класса — такие объекты связаны с базовым классом отношением «является частным случаем». С другой стороны, объекты базового класса не могут рассматриваться как объекты своих производных классов. Например, все машины являются транспортными средствами, но не любое транспортное средство является машиной (оно также может быть грузовиком, велосипедом, самолетом и т. д.). В этой главе и в главе 12 приводятся многочисленные примеры отношений типа «является частным случаем».

Производный класс может изменять методы, унаследованные им от базового класса. В таких случаях производный класс может переопределить метод базового класса более подходящей реализацией, как будет показано в примерах кода.

## 11.3. Защищенные члены классов

В главе 10 были представлены модификаторы доступа `public` и `private`. Открытые (`public`) члены классов доступны в любой точке, в которой в приложении существует

ссылка на объект этого класса или одного из его производных классов. Закрытые (`private`) члены классов доступны только в границах самого класса. Закрытые члены базового класса наследуются его производными классами, но методы и свойства производных классов не могут обращаться к ним напрямую. В этом разделе будет рассмотрен модификатор доступа `protected`. Защищенный уровень доступа `protected` может рассматриваться как промежуточный между `public` и `private`. Защищенные (`protected`) члены базового класса доступны для членов как базового класса, так и его производных классов.

Все члены базового класса, не являющиеся закрытыми, сохраняют свой исходный модификатор доступа в производных классах — открытые члены базового класса становятся открытыми членами производного класса, а защищенные члены базового класса становятся защищенными членами производного класса.

Методы производного класса могут обращаться к открытым и защищенным членам, унаследованным от базового класса, по именам. Если метод производного класса переопределяет метод базового класса, то производный класс может обратиться к версии базового класса; для этого перед именем метода базового класса ставится ключевое слово `base` и оператор «точка» (`.`). Обращение к переопределенным методам базового класса рассматривается в разделе 11.4.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 11.1

Свойства и методы производного класса не могут напрямую обращаться к закрытым членам базового класса. Производный класс может изменить состояние полей базового класса только при помощи незакрытых методов и свойств, предоставляемых базовым классом.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 11.2

Объявление закрытых полей в базовом классе упрощает тестирование, отладку и модификацию программных систем. Если бы производный класс мог обращаться к закрытым полям своего базового класса, то эти поля были бы доступны и для классов, производных от этого производного класса. В результате возможность доступа к закрытым полям распространяется по иерархии, а все преимущества сокрытия информации теряются.

## 11.4. Отношения между базовыми и производными классами

В этом разделе отношения между базовым и производными классами будут продемонстрированы на примере иерархии типов работников в приложении для расчета платежной ведомости. Первая категория работников получает процент от продаж, а работники второй категории получают базовую зарплату и процент от продаж.

Наше рассмотрение отношений между двумя категориями работников будет разделено на пять примеров.

1. В первом примере создается класс `CommissionEmployee`, производный непосредственно от класса `object`. В нем объявляются закрытые переменные экземпляров для имени, фамилии, номера социального страхования, комиссионной ставки и общего объема продаж.
2. Во втором примере объявляется класс `BasePlusCommissionEmployee`, который также является производным непосредственно от `object` и объявляет закрытые переменные экземпляров для имени, фамилии, номера социального страхования, комиссионной ставки, общего объема продаж и базовой зарплаты. При создании этого класса мы напишем весь необходимый код, — но как вы убедитесь, эффективнее создать этот класс наследованием от `CommissionEmployee`.
3. В третьем примере объявляется отдельный класс `BasePlusCommissionEmployee`, который расширяет `CommissionEmployee` (то есть `BasePlusCommissionEmployee` представляет собой класс `BasePlusCommissionEmployee` с добавлением базовой зарплаты). Вы узнаете, что методы базового класса, которые должны переопределяться в производных классах, должны быть объявлены с ключевым словом `virtual`. Класс `BasePlusCommissionEmployee` пытается обратиться к закрытым членам класса `CommissionEmployee`, но это приводит к ошибке компиляции, потому что производный класс не может обращаться к закрытым переменным своего базового класса.
4. Четвертый пример показывает, что при объявлении переменных экземпляров базового класса `CommissionEmployee` с ключевым словом `protected` класс `BasePlusCommissionEmployee`, производный от `CommissionEmployee`, сможет обращаться к этим данным напрямую.
5. Пятый пример демонстрирует решение, рекомендуемое правилами качественного проектирования: переменные экземпляра в классе `CommissionEmployee` снова объявляются закрытыми. Класс `BasePlusCommissionEmployee`, производный от класса `CommissionEmployee`, может использовать открытые методы класса `CommissionEmployee` для работы с закрытыми переменными экземпляров `CommissionEmployee`.

### 11.4.1. Создание и использование класса `CommissionEmployee`

Начнем с объявления класса `CommissionEmployee` (ил. 11.4). Оно начинается в строке 5. Двоеточие (`:`) за именем класса `object` в конце заголовка объявления указывает, что класс `CommissionEmployee` наследует (то есть является производным) от класса `object` (`System.Object` в `Framework Class Library`). Программисты C# используют наследование для создания классов на базе существующих классов. Более того, любой класс в C# (кроме `object`) расширяет существующий класс. Так как класс `CommissionEmployee` расширяет класс `object`, он наследует методы этого класса (класс `object` не имеет полей). Каждый класс C# прямо или косвенно наследует методы `object`. Если класс не указывает, что он является производным от другого



класса, то новый класс неявно считается производным от `object`. По этой причине часть «:`object`» в код обычно не включается — мы сделали это исключительно для наглядности.

```

1  // Ил. 11.4: CommissionEmployee.cs
2  // Класс CommissionEmployee представляет работника, получающего комиссионные
3  using System;
4
5  public class CommissionEmployee : object
6  {
7      private string firstName;
8      private string lastName;
9      private string socialSecurityNumber;
10     private decimal grossSales; // Общий объем продаж
11     private decimal commissionRate; // Комиссионная ставка
12
13     // Конструктор с пятью параметрами
14     public CommissionEmployee( string first, string last, string ssn,
15         decimal sales, decimal rate )
16     {
17         // Здесь происходит неявный вызов конструктора
18         firstName = first;
19         lastName = last;
20         socialSecurityNumber = ssn;
21         GrossSales = sales; // Проверка объема продаж через свойство
22         CommissionRate = rate; // Проверка комиссионной ставки через свойство
23     } // Конец конструктора CommissionEmployee с пятью параметрами
24
25     // Свойство для получения имени работника (только для чтения)
26     public string FirstName
27     {
28         get
29         {
30             return firstName;
31         } // Конец get
32     } // Конец свойства FirstName
33
34     // Свойство для получения фамилии работника (только для чтения)
35     public string LastName
36     {
37         get
38         {
39             return lastName;
40         } // Конец get
41     } // Конец свойства LastName
42
43     // Свойство для получения номера социального страхования
44     // работника (только для чтения)
45     public string SocialSecurityNumber
46     {
47         get
48         {
49             return socialSecurityNumber;

```

**Ил. 11.4.** Класс `CommissionEmployee` представляет работника, получающего комиссионные (продолжение ↗)

```

50     } // Конец get
51 } // Конец свойства SocialSecurityNumber
52
53 // Свойство для чтения и записи объема продаж работника
54 public decimal GrossSales
55 {
56     get
57     {
58         return grossSales;
59     } // Конец get
60     set
61     {
62         if ( value >= 0 )
63             grossSales = value;
64         else
65             throw new ArgumentOutOfRangeException(
66                 "GrossSales", value, "GrossSales must be >= 0" );
67     } // Конец set
68 } // Конец свойства GrossSales
69
70 // Свойство для чтения и записи комиссионной ставки работника
71 public decimal CommissionRate
72 {
73     get
74     {
75         return commissionRate;
76     } // Конец get
77     set
78     {
79         if ( value > 0 && value < 1 )
80             commissionRate = value;
81         else
82             throw new ArgumentOutOfRangeException( "CommissionRate",
83                 value, "CommissionRate must be > 0 and < 1" );
84     } // Конец set
85 } // Конец свойства CommissionRate
86
87 // Вычисление комиссионного вознаграждения работника
88 public decimal Earnings()
89 {
90     return commissionRate * grossSales;
91 } // Конец метода Earnings
92
93 // Получение строкового представления объекта CommissionEmployee
94 public override string ToString()
95 {
96     return string.Format(
97         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}",
98         "commission employee", FirstName, LastName,
99         "social security number", SocialSecurityNumber,
100        "gross sales", GrossSales, "commission rate", CommissionRate );
101 } // Конец метода method ToString
102 // Конец класса CommissionEmployee

```

**Ил. 11.4.** Класс `CommissionEmployee` представляет работника, получающего комиссионные (окончание)

### Описание класса `CommissionEmployee`

Открытый интерфейс `CommissionEmployee` включает конструктор (строки 14–23), методы `Earnings` (строки 88–91) и `ToString` (строки 94–101), а также открытые свойства (строки 26–85) для работы с переменными экземпляров `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` и `commissionRate` (объявленными в строках 7–11). Все переменные экземпляров объявлены закрытыми, поэтому объекты других классов не могут напрямую обращаться к этим переменным. Объявление закрытых переменных экземпляров и предоставление открытых свойств для работы с ними и проверки данных — признак качественно спроектированного продукта. Например, `set`-методы свойств `GrossSales` и `CommissionRate` проверяют свои аргументы перед присваиванием значений переменным экземпляров `grossSales` и `commissionRate` соответственно.

### Конструктор `CommissionEmployee`

Конструкторы не наследуются, поэтому класс `CommissionEmployee` не наследует конструктор класса `object`. Тем не менее конструктор класса `CommissionEmployee` неявно вызывает конструктор класса `object`. Более того, перед выполнением кода в своем теле конструктор производного класса вызывает конструктор своего прямого базового класса, явно или неявно (если вызов конструктора не указан); тем самым обеспечивается инициализация переменных экземпляров, унаследованных от базового класса. Синтаксис явного вызова конструктора базового класса описан в разделе 11.4.3. Если код не включает явный вызов конструктора базового класса, то компилятор генерирует неявный вызов конструктора по умолчанию или конструктора без параметров базового класса. Комментарий в строке 17 указывает, где происходит неявный вызов конструктора по умолчанию базового класса `object's` (вам не нужно самостоятельно программировать этот вызов). Конструктор по умолчанию класса `object` не делает ничего. Даже если класс не имеет конструкторов, конструктор по умолчанию, неявно объявленный компилятором для класса, будет вызывать конструктор по умолчанию или конструктор без параметров базового класса. Только класс `object` не имеет базового класса.

После неявного вызова конструктора в строках 18–22 конструктора присваиваются значения переменных экземпляров. Мы не проверяем значения аргументов `first`, `last` и `ssn`, прежде чем присвоить их переменным экземпляров. Конечно, фамилию и имя (`first` и `last`) можно проверить — например, убедиться в том, что они имеют разумную длину. Также можно проверить и номер социального страхования (`ssn`): например, что он состоит из девяти цифр с дефисами или без них (например, 123-45-6789 или 123456789).

### Метод `Earnings`

Метод `Earnings` (строки 88–91) вычисляет заработок `CommissionEmployee`. Строка 90 умножает комиссионную ставку `commissionRate` на объем продаж `grossSales` и возвращает результат.

## Метод ToString

Метод `ToString` (строки 94–101) отличается от других — это один из методов, прямо или косвенно наследуемых каждым классом от класса `object`, корневого класса иерархии C#.

В разделе 11.7 приведена сводка методов класса `object`. Метод `ToString` возвращает строковое представление объекта. Он  *неявно* вызывается приложением тогда, когда объект требуется преобразовать в строковое представление, — например, для метода `Write` класса `Console` или строкового метода `Format` при использовании форматного элемента. Метод `ToString` класса `object` возвращает строку с именем класса объекта. Этот метод может (и обычно должен) переопределяться производным классом для определения нужного строкового представления данных объекта производного класса. Метод `ToString` класса `CommissionEmployee` переопределяет метод `ToString`, унаследованный от класса `object`. При вызове метод `ToString` класса `CommissionEmployee` использует метод `Format` класса `string` для возвращения строки с описанием объекта `CommissionEmployee`. В строке 97 форматный спецификатор `C` (в "{6:C}") форматирует `grossSales` как денежную сумму, а форматный спецификатор `F2` (в "{8:F2}") форматирует `commissionRate` с двумя цифрами в дробной части. Чтобы переопределить метод базового класса, производный класс должен объявить метод с ключевым словом `override`, имеющий ту же сигнатуру (имя метода, количество параметров и типы параметров) и возвращающий тот же тип, что и метод базового класса. Метод `ToString` класса `object` вызывается без параметров и возвращает тип `string`, поэтому класс `CommissionEmployee` объявляет метод `ToString` с тем же списком параметров и возвращаемым типом.



### ТИПИЧНАЯ ОШИБКА 11.1

Переопределение метода с изменением модификатора доступа приводит к ошибке компиляции. Установление более жесткого модификатора доступа нарушит отношение «является частным случаем»: если открытый метод можно было бы переопределить как защищенный или закрытый, то объекты производного класса не смогут реагировать на те же вызовы методов, на которые реагируют методы базового класса. Метод, объявленный в базовом классе, должен сохранить свой модификатор доступа в производных классах (прямом и косвенных).

## Класс `CommissionEmployeeTest`

В листинге на ил. 11.5 представлен класс для тестирования класса `CommissionEmployee`. В строках 10–11 создается объект `CommissionEmployee` и вызывается его конструктор (строки 14–23 на ил. 11.4) для выполнения инициализации. К значениям объема продаж и комиссионной ставки присоединяется суффикс, означающий, что компилятор должен интерпретировать их как литералы типа `decimal`, а не `double`. В строках 16–22 свойства `CommissionEmployee` используются для получения значений переменных экземпляров, выводимых на консоль. Строка 23 выводит сумму, вычисленную методом `Earnings`. Строки 25–26 вызывают `set`-методы свойств `GrossSales` и `CommissionRate` для изменения значений переменных экземпляров `grossSales`

и `commissionRate`. Строки 28–29 выводят строковое представление обновленного объекта `CommissionEmployee`. При выводе объекта с использованием форматного элемента неявно вызывается метод `ToString` для получения строкового представления объекта. В строке 30 снова выводится результат вызова `Earnings()`.

```

1  // Ил. 11.5: CommissionEmployeeTest.cs
2  // Тестовый класс для CommissionEmployee.
3  using System;
4
5  public class CommissionEmployeeTest
6  {
7      public static void Main( string[] args )
8      {
9          // Создание объекта CommissionEmployee
10         CommissionEmployee employee = new CommissionEmployee( "Sue",
11             "Jones", "222-22-2222", 10000.00M, .06M );
12
13         // Вывод данных CommissionEmployee
14         Console.WriteLine(
15             "Employee information obtained by properties and methods: \n" );
16         Console.WriteLine( "First name is {0}", employee.FirstName );
17         Console.WriteLine( "Last name is {0}", employee.LastName );
18         Console.WriteLine( "Social security number is {0}",
19             employee.SocialSecurityNumber );
20         Console.WriteLine( "Gross sales are {0:C}", employee.GrossSales );
21         Console.WriteLine( "Commission rate is {0:F2}",
22             employee.CommissionRate );
23         Console.WriteLine( "Earnings are {0:C}", employee.Earnings() );
24
25         employee.GrossSales = 5000.00M; // Присваивание объема продаж
26         employee.CommissionRate = .1M; // Присваивание комиссионной ставки
27
28         Console.WriteLine( "\n{0}:\n{n{1}}",
29             "Updated employee information obtained by ToString", employee );
30         Console.WriteLine( "earnings: {0:C}", employee.Earnings() );
31     } // Конец Main
32 } // Конец класса CommissionEmployeeTest

```

Employee information obtained by properties and methods:

```

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales are $10,000.00
Commission rate is 0.06
Earnings are $600.00

```

Updated employee information obtained by ToString:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: $5,000.00
commission rate: 0.10
earnings: $500.00

```

**Ил. 11.5.** Тестовый класс для `CommissionEmployee`

## 11.4.2. Создание класса BasePlusCommissionEmployee без применения наследования

Пора сделать следующий шаг в изучении наследования — объявление и тестирование (полностью нового и самостоятельного) класса `BasePlusCommissionEmployee` (ил. 11.6). В полях этого класса хранятся имя, фамилия, номер социального страхования, объем продаж, комиссионная ставка и базовая зарплата («base» в имени класса относится к базовой зарплате, а не к базовому классу). Открытый интерфейс класса `BasePlusCommissionEmployee` включает конструктор `BasePlusCommissionEmployee` (строки 16–26), методы `Earnings` (строки 113–116) и `ToString` (строки 119–127) и открытые свойства (строки 30–110) для закрытых переменных `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` и `baseSalary` (объявляемых в строках 8–13). В этих переменных, свойствах и методах инкапсулируется вся необходимая функциональность класса. Обратите внимание на сходство между этим классом и `CommissionEmployee` (см. ил. 11.4) — в данном примере это сходство не используется.

Класс `BasePlusCommissionEmployee` расширяет класс `object`, хотя в строке 6 об этом явно не сказано. Кроме того, конструктор класса `BasePlusCommissionEmployee`, как и конструктор `CommissionEmployee`, неявно вызывает конструктор по умолчанию класса `object` (строка 19 на ил. 11.6).

```
1  // Ил. 11.6: BasePlusCommissionEmployee.cs
2  // Класс BasePlusCommissionEmployee представляет работника,
3  // получающего базовую зарплату в дополнение к комиссионным.
4  using System;
5
6  public class BasePlusCommissionEmployee
7  {
8      private string firstName;
9      private string lastName;
10     private string socialSecurityNumber;
11     private decimal grossSales;      // Объем продаж
12     private decimal commissionRate; // Комиссионная ставка
13     private decimal baseSalary;      // Базовая зарплата
14
15     // Конструктор с шестью параметрами
16     public BasePlusCommissionEmployee( string first, string last,
17     string ssn, decimal sales, decimal rate, decimal salary )
18     {
19         // Здесь происходит неявный вызов конструктора object
20         firstName = first;
21         lastName = last;
22         socialSecurityNumber = ssn;
23         GrossSales = sales;      // Проверка объема продаж через свойство
24         CommissionRate = rate;  // Проверка комиссионной ставки через свойство
25         BaseSalary = salary;     // Проверка базовой зарплаты через свойство
26     } // Конец конструктора BasePlusCommissionEmployee
27
```

**Ил. 11.6.** Класс `BasePlusCommissionEmployee` представляет работника, получающего базовую зарплату помимо комиссионных (продолжение ↗)

```
28 // Свойство для получения имени
29 // объекта BasePlusCommissionEmployee (только для чтения)
30 public string FirstName
31 {
32     get
33     {
34         return firstName;
35     } // Конец get
36 } // Конец свойства FirstName
37
38 // Свойство для получения фамилии
39 // объекта BasePlusCommissionEmployee (только для чтения)
40 public string LastName
41 {
42     get
43     {
44         return lastName;
45     } // Конец get
46 } // Конец свойства LastName
47
48 // Свойство для получения номера социального страхования
49 // объекта BasePlusCommissionEmployee (только для чтения)
50 public string SocialSecurityNumber
51 {
52     get
53     {
54         return socialSecurityNumber;
55     } // Конец get
56 } // Конец свойства SocialSecurityNumber
57
58 // Свойство для чтения и записи объема продаж
59 // объекта BasePlusCommissionEmployee
60 public decimal GrossSales
61 {
62     get
63     {
64         return grossSales;
65     } // Конец get
66     set
67     {
68         if ( value >= 0 )
69             grossSales = value;
70         else
71             throw new ArgumentOutOfRangeException(
72                 "GrossSales", value, "GrossSales must be >= 0" );
73     } // Конец set
74 } // Конец свойства GrossSales
75
76 // Свойство для чтения и записи комиссионной ставки
77 // объекта BasePlusCommissionEmployee
78 public decimal CommissionRate
79 {
80     get
```

**Ил. 11.6.** Класс `BasePlusCommissionEmployee` представляет работника, получающего базовую зарплату помимо комиссионных (продолжение ↗)

```

81     {
82         return commissionRate;
83     } // Конец get
84     set
85     {
86         if ( value > 0 && value < 1 )
87             commissionRate = value;
88         else
89             throw new ArgumentOutOfRangeException( "CommissionRate",
90                 value, "CommissionRate must be > 0 and < 1" );
91     } // Конец set
92 } // Конец свойства CommissionRate
93
94 // Свойство для чтения и записи базовой зарплаты
95 // объекта BasePlusCommissionEmployee
96 public decimal BaseSalary
97 {
98     get
99     {
100         return baseSalary;
101     } // Конец get
102     set
103     {
104         if ( value >= 0 )
105             baseSalary = value;
106         else
107             throw new ArgumentOutOfRangeException( "BaseSalary",
108                 value, "BaseSalary must be >= 0" );
109     } // Конец set
110 } // Конец свойства BaseSalary
111
112 // Вычисление заработка
113 public decimal Earnings()
114 {
115     return baseSalary + ( commissionRate * grossSales );
116 } // Конец метода Earnings
117
118 // Получение строкового представления BasePlusCommissionEmployee
119 public override string ToString()
120 {
121     return string.Format(
122         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}",
123         "base-salaried commission employee", firstName, lastName,
124         "social security number", socialSecurityNumber,
125         "gross sales", grossSales, "commission rate", commissionRate,
126         "base salary", baseSalary );
127     } // Конец метода ToString
128 } // Конец класса BasePlusCommissionEmployee

```

**Ил. 11.6.** Класс `BasePlusCommissionEmployee` представляет работника, получающего базовую зарплату помимо комиссионных (окончание)

Метод `Earnings` класса `BasePlusCommissionEmployee` (строки 113–116) вычисляет заработок работника, получающего базовую зарплату и комиссионные. Строка 115 суммирует базовую зарплату с произведением комиссионной ставки и объема



продаж и возвращает результат. Класс `BasePlusCommissionEmployee` переопределяет метод `ToString` класса `object` так, чтобы он возвращал строку с информацией об объекте `BasePlusCommissionEmployee` (строки 119–127). Как и в предыдущем случае, форматный спецификатор `C` используется для форматирования объема продаж и базовой зарплаты как денежных величин, а спецификатор `F2` — для форматирования комиссионной ставки с двумя цифрами в дробной части (строка 122).

### Класс `BasePlusCommissionEmployeeTest`

На ил. 11.7 приведен класс для тестирования `BasePlusCommissionEmployee`. В строках 10–12 создается объект `BasePlusCommissionEmployee`, при этом конструктору передаются параметры "Bob", "Lewis", "333-33-3333", 5000.00M, .04M и 300.00M. Строки 17–25 используют свойства и методы `BasePlusCommissionEmployee` для получения значений переменных и вычисления заработка. В строке 27 свойство `BaseSalary` используется для изменения базовой зарплаты. `Set`-метод свойства `BaseSalary` (см. ил. 11.6, строки 102–109) проверяет, что переменной экземпляра `baseSalary` не присваивается отрицательное значение, потому что базовая зарплата не может быть отрицательной. В строках 29–30 на ил. 11.7 неявно вызывается метод `ToString` для получения строкового представления объекта.

```

1  // Ил. 11.7: BasePlusCommissionEmployeeTest.cs
2  // Тестовый класс для BasePlusCommissionEmployee.
3  using System;
4
5  public class BasePlusCommissionEmployeeTest
6  {
7      public static void Main( string[] args )
8      {
9          // Создание объекта BasePlusCommissionEmployee
10         BasePlusCommissionEmployee employee =
11             new BasePlusCommissionEmployee( "Bob", "Lewis",
12                 "333-33-3333", 5000.00M, .04M, 300.00M );
13
14         // Вывод данных BasePlusCommissionEmployee
15         Console.WriteLine(
16             "Employee information obtained by properties and methods: \n" );
17         Console.WriteLine( "First name is {0}", employee.FirstName );
18         Console.WriteLine( "Last name is {0}", employee.LastName );
19         Console.WriteLine( "Social security number is {0}",
20             employee.SocialSecurityNumber );
21         Console.WriteLine( "Gross sales are {0:C}", employee.GrossSales );
22         Console.WriteLine( "Commission rate is {0:F2}",
23             employee.CommissionRate );
24         Console.WriteLine( "Earnings are {0:C}", employee.Earnings() );
25         Console.WriteLine( "Base salary is {0:C}", employee.BaseSalary );
26
27         employee.BaseSalary = 1000.00M; // Присваивание базовой зарплаты
28
29         Console.WriteLine( "\n{0}:\n\n{1}",
30             "Updated employee information obtained by ToString", employee );

```

**Ил. 11.7.** Тестовый класс для `BasePlusCommissionEmployee` (продолжение ↗)

```
31     Console.WriteLine( "earnings: {0:C}", employee.Earnings() );
32 } // Конец Main
33 } // Конец класса BasePlusCommissionEmployeeTest
```

Employee information obtained by properties and methods:

```
First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales are $5,000.00
Commission rate is 0.04
Earnings are $500.00
Base salary is $300.00
```

Updated employee information obtained by ToString:

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $1,000.00
earnings: $1,200.00
```

#### Ил. 11.7. Тестовый класс для BasePlusCommissionEmployee (окончание)

Большая часть кода класса BasePlusCommissionEmployee (см. ил. 11.6) похожа на код CommissionEmployee, а то и полностью совпадает с ним (см. ил. 11.4). Например, в классе BasePlusCommissionEmployee закрытые переменные firstName и lastName, а также свойства FirstName и LastName идентичны одноименным членам CommissionEmployee. Классы CommissionEmployee и BasePlusCommissionEmployee также содержат закрытые переменные socialSecurityNumber, commissionRate и grossSales, как и свойства для работы с этими переменными. Кроме того, конструктор BasePlusCommissionEmployee почти идентичен конструктору класса CommissionEmployee, за исключением того, что конструктор BasePlusCommissionEmployee также инициализирует baseSalary. Среди других дополнений класса BasePlusCommissionEmployee можно выделить закрытую переменную экземпляра baseSalary и свойство BaseSalary. Метод Earnings класса BasePlusCommissionEmployee почти идентичен одноименному методу CommissionEmployee, не считая того, что BasePlusCommissionEmployee прибавляет значение baseSalary. Метод ToString класса BasePlusCommissionEmployee тоже почти не отличается от одноименного метода класса CommissionEmployee, за исключением того, что метод ToString класса BasePlusCommissionEmployee также форматирует значение переменной baseSalary как денежную сумму.

Мы буквально скопировали код из класса CommissionEmployee и вставили его в BasePlusCommissionEmployee, а затем внесли изменения в BasePlusCommissionEmployee: добавили переменную базовой зарплаты, а также методы и свойства для работы с этой переменной. Решения, основанные на копировании кода, часто ненадежны, а их реализация занимает много времени. Что еще хуже, в системе появляется несколько физических копий одного кода, что основательно усложняет сопровождение. Можно ли «поглотить» члены одного класса так, чтобы они стали частью

другого класса, без копирования кода? В нескольких ближайших примерах мы ответим на этот вопрос, используя более элегантный механизм построения классов, а именно *наследование*.



### КАК ИЗБЕЖАТЬ ОШИБОК 11.1

Копирование кода между классами может привести к распространению ошибок по файлам с исходным кодом. Чтобы избежать дублирования кода (и возможного появления ошибок) в ситуациях, в которых один класс должен «поглотить» члены другого класса, применяйте наследование вместо копирования кода.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 11.3

При наследовании общие члены всех классов иерархии объявляются в базовом классе. Если потребуются изменить общую функциональность иерархии, то изменения достаточно внести только в базовом классе — изменения будут унаследованы производными классами. Без наследования такие изменения придется вносить во всех файлах исходным кодом, содержащим копию соответствующего фрагмента.

## 11.4.3. Создание иерархии наследования

На следующем этапе мы объявляем класс `BasePlusCommissionEmployee` (ил. 11.8), производный от `CommissionEmployee` (см. ил. 11.4). С объектом `BasePlusCommissionEmployee` можно работать как с `CommissionEmployee` (потому что наследование передает функциональность класса `CommissionEmployee`), но класс `BasePlusCommissionEmployee` также содержит переменную экземпляра `baseSalary` (см. ил. 11.8, строка 7). Двоеточие (:) в строке 5 объявления класса означает наследование. Производный класс `BasePlusCommissionEmployee` наследует члены класса `CommissionEmployee` и может обращаться к его незакрытым членам. Конструктор класса `CommissionEmployee` не наследуется. Таким образом, открытый интерфейс `BasePlusCommissionEmployee` включает конструктор (строки 11–16), открытые методы и свойства, унаследованные от `CommissionEmployee`, свойство `BaseSalary` (строки 20–34), метод `Earnings` (строки 37–41) и метод `ToString` (строки 44–53).

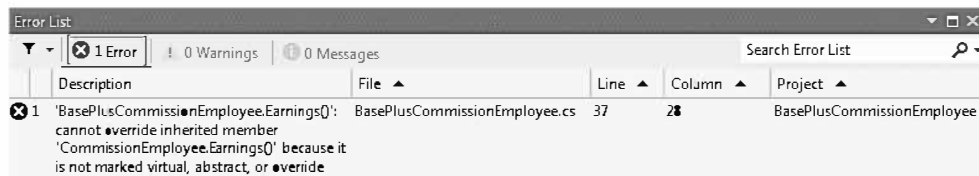
```
1 // Ил. 11.8: BasePlusCommissionEmployee.cs
2 // Класс BasePlusCommissionEmployee, производный от CommissionEmployee.
3 using System;
4
5 public class BasePlusCommissionEmployee : CommissionEmployee
6 {
7     private decimal baseSalary; // Базовая зарплата
8
9     // Конструктор производного класса с шестью параметрами
10    // вызывает конструктор базового класса CommissionEmployee
```

**Ил. 11.8.** Класс `BasePlusCommissionEmployee` наследует от `CommissionEmployee` (продолжение ↗)

```

11 public BasePlusCommissionEmployee( string first, string last,
12     string ssn, decimal sales, decimal rate, decimal salary )
13     : base( first, last, ssn, sales, rate )
14 {
15     BaseSalary = salary; // Проверка базовой зарплаты через свойство
16 } // Конец конструктора BasePlusCommissionEmployee
17
18 // Свойство для чтения и записи
19 // базовой зарплаты объекта BasePlusCommissionEmployee
20 public decimal BaseSalary
21 {
22     get
23     {
24         return baseSalary;
25     } // Конец get
26     set
27     {
28         if ( value >= 0 )
29             baseSalary = value;
30         else
31             throw new ArgumentOutOfRangeException( "BaseSalary",
32                 value, "BaseSalary must be >= 0" );
33     } // Конец set
34 } // Конец свойства BaseSalary
35
36 // Вычисление заработка
37 public override decimal Earnings()
38 {
39     // Запрещено: commissionRate и grossSales закрыты в базовом классе
40     return baseSalary + ( commissionRate * grossSales );
41 } // Конец метода Earnings
42
43 // Получение строкового представления BasePlusCommissionEmployee
44 public override string ToString()
45 {
46     // Запрещено: попытка обращения к закрытым членам базового класса
47     return string.Format(
48         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}",
49         "base-salaried commission employee", firstName, lastName,
50         "social security number", socialSecurityNumber,
51         "gross sales", grossSales, "commission rate", commissionRate,
52         "base salary", baseSalary );
53 } // Конец метода ToString
54 } // Конец класса BasePlusCommissionEmployee

```



**Ил. 11.8.** Класс BasePlusCommissionEmployee наследует от CommissionEmployee (окончание)

### Конструктор производного класса должен вызывать конструктор своего базового класса

Каждый конструктор производного класса должен явно или неявно вызывать конструктор своего базового класса, чтобы обеспечить правильную инициализацию переменных экземпляров, унаследованных от базового класса. Конструктор `BasePlusCommissionEmployee` с шестью параметрами явно вызывает конструктор с пятью параметрами класса `CommissionEmployee` для инициализации части объекта `BasePlusCommissionEmployee`, унаследованной от `CommissionEmployee`, — то есть переменных экземпляров `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` и `commissionRate`. Строка 13 в заголовке конструктора `BasePlusCommissionEmployee` вызывает конструктор `CommissionEmployee` с пятью параметрами (объявленный в строках 14–23 на ил. 11.4) посредством использования инициализатора конструктора. В разделе 10.5 мы использовали инициализаторы конструкторов с ключевым словом `this` для вызова перегруженных конструкторов одного класса. В строке 13 на ил. 11.8 инициализатор используется с ключевым словом `base` для вызова конструктора базового класса. Аргументы `first`, `last`, `ssn`, `sales` и `rate` используются для инициализации переменных базового класса `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` и `commissionRate` соответственно. Если бы конструктор `BasePlusCommissionEmployee` не вызывал конструктор `CommissionEmployee` явно, то компилятор C# попытался бы неявно вызвать конструктор `CommissionEmployee` без параметров или конструктор по умолчанию — но у класса такого конструктора нет, так что компилятор выдал бы сообщение об ошибке. Если базовый класс содержит конструктор без параметров, вы можете включить в инициализатор явный вызов `base()`, но эта возможность используется редко.



#### ТИПИЧНАЯ ОШИБКА 11.2

Если конструктор производного класса пытается вызвать один из конструкторов своего базового класса с аргументами, не соответствующими объявлениям конструкторов базового класса по количеству и типу параметров, происходит ошибка компиляции.

### Метод `Earnings` класса `BasePlusCommissionEmployee`

В строках 37–41 на ил. 11.8 метод `Earnings` объявляется с ключевым словом `override` для переопределения метода `Earnings` класса `CommissionEmployee`, как это было сделано с методом `ToString` в предыдущих примерах. В строке 37 компилятор выдает ошибку с сообщением о том, что переопределение метода `Earnings` базового класса невозможно, поскольку этот метод не был явно помечен ключевым словом `virtual`, `abstract` или `override`. Ключевые слова `virtual` и `abstract` указывают, что метод базового класса может переопределяться в производных классах. (Как вы узнаете в разделе 12.4, абстрактные методы также неявно объявляются виртуальными.) Модификатор `override` объявляет, что метод производного класса переопределяет виртуальный или абстрактный метод базового класса. Этот модификатор также неявно объявляет метод производного класса виртуальным и разрешает его переопределение в производных классах далее по иерархии наследования.

Если добавить ключевое слово `virtual` в объявление метода `Earnings` на ил. 11.4 и перекомпилировать программу, появляются другие ошибки компиляции (ил. 11.9). Компилятор выдает дополнительные ошибки в строке 40 на ил. 11.8, потому что переменные экземпляров `commissionRate` и `grossSales` базового класса `CommissionEmployee` объявлены закрытыми — методам производного класса `BasePlusCommissionEmployee` запрещен доступ к закрытым переменным базового класса `CommissionEmployee`. Компилятор выдает дополнительные ошибки в строках 49–51 метода `ToString` класса `BasePlusCommissionEmployee` по той же причине. Ошибки в `BasePlusCommissionEmployee` можно было бы предотвратить использованием открытых свойств, унаследованных от класса `CommissionEmployee`. Например, строка 40 могла бы вызывать `get`-методы свойств `CommissionRate` и `GrossSales` для обращения к закрытым переменным `commissionRate` и `grossSales` класса `CommissionEmployee`. Строки 49–51 также могли бы использовать соответствующие свойства для получения значений переменных базового класса.

	Description	File	Line	Column	Project
1	'CommissionEmployee.commissionRate' is inaccessible due to its protection level	BasePlusCommissionEmployee.cs	40	29	BasePlusCommissionEmployee
2	'CommissionEmployee.grossSales' is inaccessible due to its protection level	BasePlusCommissionEmployee.cs	40	46	BasePlusCommissionEmployee
3	'CommissionEmployee.firstName' is inaccessible due to its protection level	BasePlusCommissionEmployee.cs	49	47	BasePlusCommissionEmployee
4	'CommissionEmployee.lastName' is inaccessible due to its protection level	BasePlusCommissionEmployee.cs	49	58	BasePlusCommissionEmployee
5	'CommissionEmployee.socialSecurityNumber' is inaccessible due to its protection level	BasePlusCommissionEmployee.cs	50	36	BasePlusCommissionEmployee
6	'CommissionEmployee.grossSales' is inaccessible due to its protection level	BasePlusCommissionEmployee.cs	51	25	BasePlusCommissionEmployee
7	'CommissionEmployee.commissionRate' is inaccessible due to its protection level	BasePlusCommissionEmployee.cs	51	56	BasePlusCommissionEmployee

**Ил. 11.9.** Ошибки компиляции класса `BasePlusCommissionEmployee` (см. ил. 11.8) после объявления метода `Earnings` с ключевым словом `virtual`

#### 11.4.4. Иерархия наследования `CommissionEmployee`–`BasePlusCommissionEmployee` с использованием защищенных переменных экземпляров

Чтобы класс `BasePlusCommissionEmployee` мог напрямую обращаться к переменным экземпляров базового класса `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` и `commissionRate`, можно объявить эти переменные защищенными (`protected`) в базовом классе. Как было указано в разделе 11.3, защищенные члены базового класса наследуются всеми производными классами. Класс `CommissionEmployee` в данном примере представляет собой измененную версию реализации на ил. 11.4, в которой переменные экземпляров `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` и `commissionRate` объявлены защищенными, а не закрытыми. Также метод `Earnings`

объявляется виртуальным (`virtual`), чтобы он мог быть переопределен классом `BasePlusCommissionEmployee`:

```
public virtual decimal Earnings()
```

Остальной код объявления класса в этом примере идентичен коду на ил. 11.4. Полный исходный код класса `CommissionEmployee` находится в проекте данного примера.

### Открытые и защищенные данные

Мы также могли объявить переменные экземпляров базового класса `CommissionEmployee` (`firstName`, `lastName`, `socialSecurityNumber`, `grossSales` и `commissionRate`) открытыми, чтобы производный класс `BasePlusCommissionEmployee` смог обращаться к ним. Однако объявление открытых переменных экземпляров является признаком некачественного проектирования, поскольку оно допускает неограниченный доступ к переменным экземпляров, существенно увеличивающий вероятность ошибок. Защищенные переменные доступны для производного класса, тогда как другие классы, не производные от базового, не смогут напрямую обращаться к ним.

### Класс `BasePlusCommissionEmployee`

Класс `BasePlusCommissionEmployee` (ил. 11.10) в этом примере расширяет версию класса `CommissionEmployee` защищенными данными (в отличие от версии с закрытыми данными на ил. 11.4). Каждый объект `BasePlusCommissionEmployee` наследует защищенные переменные класса `CommissionEmployee` (`firstName`, `lastName`, `socialSecurityNumber`, `grossSales` и `commissionRate`); все эти переменные становятся защищенными членами `BasePlusCommissionEmployee`. В результате компилятор не выдает сообщение об ошибке в строке 40 метода `Earnings` и в строках 48–50 метода `ToString`. Если у `BasePlusCommissionEmployee` появится свой производный класс, то он тоже унаследует защищенные члены.

Класс `BasePlusCommissionEmployee` не наследует конструктор класса `CommissionEmployee`. Однако конструктор класса `BasePlusCommissionEmployee` с шестью параметрами (строки 12–17) вызывает конструктор класса `CommissionEmployee` с пятью параметрами в инициализаторе. Конструктор `BasePlusCommissionEmployee` с шестью параметрами должен явно вызывать конструктор `CommissionEmployee` с пятью параметрами, потому что класс `CommissionEmployee` не предоставляет конструктор без параметров, который мог бы вызываться неявно.

```
1 // Ил. 11.10: BasePlusCommissionEmployee.cs
2 // Класс BasePlusCommissionEmployee, производный от CommissionEmployee,
3 // может обращаться к защищенным переменным CommissionEmployee.
4 using System;
5
6 public class BasePlusCommissionEmployee : CommissionEmployee
7 {
8     private decimal baseSalary; // Базовая зарплата
9
10    // Конструктор производного класса с шестью параметрами
```

**Ил. 11.10.** Класс `BasePlusCommissionEmployee`, производный от `CommissionEmployee`, может обращаться к защищенным переменным `CommissionEmployee` (продолжение ↗)

```

11 // вызывает конструктор CommissionEmployee базового класса
12 public BasePlusCommissionEmployee( string first, string last,
13     string ssn, decimal sales, decimal rate, decimal salary )
14     : base( first, last, ssn, sales, rate )
15 {
16     BaseSalary = salary; // Проверка базовой зарплаты через свойство
17 } // Конец конструктора BasePlusCommissionEmployee
18
19 // Свойство для чтения и записи
20 // базовой зарплаты BasePlusCommissionEmployee
21 public decimal BaseSalary
22 {
23     get
24     {
25         return baseSalary;
26     } // Конец get
27     set
28     {
29         if ( value >= 0 )
30             baseSalary = value;
31         else
32             throw new ArgumentOutOfRangeException( "BaseSalary",
33                 value, "BaseSalary must be >= 0" );
34     } // Конец set
35 } // Конец свойства BaseSalary
36
37 // Вычисление заработка
38 public override decimal Earnings()
39 {
40     return baseSalary + ( commissionRate * grossSales );
41 } // Конец метода Earnings
42
43 // Получение строкового представления BasePlusCommissionEmployee
44 public override string ToString()
45 {
46     return string.Format(
47         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}",
48         "base-salaried commission employee", firstName, lastName,
49         "social security number", socialSecurityNumber,
50         "gross sales", grossSales, "commission rate", commissionRate,
51         "base salary", baseSalary );
52 } // Конец метода ToString
53 } // Конец класса BasePlusCommissionEmployee

```

**Ил. 11.10.** Класс `BasePlusCommissionEmployee`, производный от `CommissionEmployee`, может обращаться к защищенным переменным `CommissionEmployee` (окончание)

### Класс `BasePlusCommissionEmployeeTest`

В листинге на ил. 11.11 объект `BasePlusCommissionEmployee` выполняет те же операции, которые класс на ил. 11.7 выполнял версий класса на ил. 11.6. Результаты работы двух приложений идентичны. Хотя версия класса на ил. 11.6 объявлялась без наследования, а в версии на ил. 11.10 используется наследование, оба класса предоставляют одинаковую функциональность. Исходный код на ил. 11.10 (состоящий



из 53 строк) существенно короче версии на ил. 11.6 (которая состоит из 128 строк), потому что новый класс наследует большую часть функциональности от `CommissionEmployee`, а версия на ил. 11.6 наследует только функциональность класса `object`. Кроме того, в новой версии существует только одна копия функциональности, объявленной в классе `CommissionEmployee`. Это существенно упрощает сопровождение, изменение и отладку кода, потому что код, относящийся к работе за комиссионные, существует *только* в классе `CommissionEmployee`.

```

1  // Ил. 11.11: BasePlusCommissionEmployee.cs
2  // Тестовый класс для BasePlusCommissionEmployee.
3  using System;
4
5  public class BasePlusCommissionEmployeeTest
6  {
7      public static void Main( string[] args )
8      {
9          // Создание объекта BasePlusCommissionEmployee
10         BasePlusCommissionEmployee basePlusCommissionEmployee =
11             new BasePlusCommissionEmployee( "Bob", "Lewis",
12                 "333-33-3333", 5000.00M, .04M, 300.00M );
13
14         // вывод данных BasePlusCommissionEmployee
15         Console.WriteLine(
16             "Employee information obtained by properties and methods: \n" );
17         Console.WriteLine( "First name is {0}",
18             basePlusCommissionEmployee.FirstName );
19         Console.WriteLine( "Last name is {0}",
20             basePlusCommissionEmployee.LastName );
21         Console.WriteLine( "Social security number is {0}",
22             basePlusCommissionEmployee.SocialSecurityNumber );
23         Console.WriteLine( "Gross sales are {0:C}",
24             basePlusCommissionEmployee.GrossSales );
25         Console.WriteLine( "Commission rate is {0:F2}",
26             basePlusCommissionEmployee.CommissionRate );
27         Console.WriteLine( "Earnings are {0:C}",
28             basePlusCommissionEmployee.Earnings() );
29         Console.WriteLine( "Base salary is {0:C}",
30             basePlusCommissionEmployee.BaseSalary );
31
32         basePlusCommissionEmployee.BaseSalary = 1000.00M; // Присваивание
33                                                             // базовой зарплаты
34         Console.WriteLine( "\n{0}:\n\n{1}",
35             "Updated employee information obtained by ToString",
36             basePlusCommissionEmployee );
37         Console.WriteLine( "earnings: {0:C}",
38             basePlusCommissionEmployee.Earnings() );
39     } // Конец Main
40 } // Конец класса BasePlusCommissionEmployeeTest

```

```
Employee information obtained by properties and methods:
```

```
First name is Bob
Last name is Lewis
```

**Ил. 11.11.** Тестовый класс для `BasePlusCommissionEmployee` (продолжение ➤)

```
Social security number is 333-33-3333
Gross sales are $5,000.00
Commission rate is 0.04
Earnings are $500.00
Base salary is $300.00

Updated employee information obtained by ToString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $1,000.00
earnings: $1,200.00
```

### Ил. 11.11. Тестовый класс для BasePlusCommissionEmployee (окончание)

## Проблемы с защищенными переменными экземпляров

В этом примере переменные экземпляров базового класса объявлены защищенными, чтобы производные классы могли обращаться к ним. Наследование защищенных переменных экземпляров позволяет напрямую обращаться к ним в производных классах без вызова set- или get-методов соответствующих свойств, а это означает нарушение инкапсуляции. В большинстве случаев лучше использовать закрытые переменные экземпляров для повышения качества проектирования. Это упростит сопровождение, изменение и отладку кода.

Использование защищенных переменных экземпляров создает ряд потенциальных проблем. Во-первых, поскольку объект производного класса может напрямую задать значение унаследованной переменной без set-метода, в объекте производного класса эта переменная может принимать недействительные значения. Например, если бы мы объявили переменную `grossSales` класса `CommissionEmployee` защищенной, то объект производного класса (например, `BasePlusCommissionEmployee`) смог бы присвоить `grossSales` отрицательное значение. Во-вторых, использование защищенных переменных экземпляров повышает вероятность того, что методы производных классов будут зависеть от реализации данных базового класса. На практике производные классы должны зависеть только от интерфейса базового класса (то есть незакрытых методов и свойств), а не от реализации данных в базовом классе.

Если реализация базового класса изменится, возможно, нам придется обновлять все производные классы. Например, если по каким-то причинам имена переменных экземпляров `firstName` и `lastName` будут заменены именами `first` и `last`, то замену придется повторить везде, где производные классы обращаются к переменным `firstName` и `lastName` по именам. Такая архитектура программы называется *хрупкой* (fragile), потому что незначительное изменение в базовом классе «ломает» реализацию производных классов. Программист должен иметь возможность изменить реализацию базового класса без изменения сервиса, предоставляемого производным классам. Конечно, если изменится сервис базового класса, производные классы все равно придется реализовывать заново.

**АРХИТЕКТУРНОЕ РЕШЕНИЕ 11.4**

Объявление переменных экземпляров базового класса закрытыми (в отличие от защищенных) позволяет изменять реализацию этих переменных в базовом классе без изменения реализаций производных классов.

### 11.4.5. Иерархия наследования `CommissionEmployee`–`BasePlusCommissionEmployee` с использованием закрытых переменных экземпляров

Пришло время пересмотреть нашу иерархию классов еще один раз — с применением современной практики проектирования. Класс `CommissionEmployee` (ил. 11.12) объявляет переменные экземпляров `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` и `commissionRate` закрытыми (строки 7–11) и предоставляет открытые свойства `FirstName`, `LastName`, `SocialSecurityNumber`, `GrossSales` и `CommissionRate` для работы с этими значениями. Методы `Earnings` (строки 88–91) и `ToString` (строки 94–101) используют свойства класса для получения значений его переменных. Если в будущем потребуется изменить имена переменных экземпляров, то объявления `Earnings` и `ToString` изменять не придется — все изменения вносятся только в теле свойств, непосредственно обращающихся к переменным экземпляров. Эти изменения ограничиваются базовым классом; в производных классах ничего изменять не нужно. Подобная локализация изменений является признаком качественного проектирования. Производный класс `BasePlusCommissionEmployee` (ил. 11.13) наследует от `CommissionEmployee` и может обращаться к закрытым членам базового класса через унаследованные открытые свойства.

```
1  // Ил. 11.12: CommissionEmployee.cs
2  // Класс представляет работника, получающего комиссионные.
3  using System;
4
5  public class CommissionEmployee
6  {
7      private string firstName;
8      private string lastName;
9      private string socialSecurityNumber;
10     private decimal grossSales; // Объем продаж
11     private decimal commissionRate; // Комиссионная ставка
12
13     // Конструктор с пятью параметрами
14     public CommissionEmployee( string first, string last, string ssn,
15         decimal sales, decimal rate )
16     {
17         // Здесь происходит неявный вызов конструктора
18         firstName = first;
19         lastName = last;
20         socialSecurityNumber = ssn;
```

**Ил. 11.12.** Класс `CommissionEmployee` представляет работника, получающего комиссионные (продолжение ↗)

```
21     GrossSales = sales;    // Проверка объема продаж через свойство
22     CommissionRate = rate; // Проверка комиссионной ставки через свойство
23 } // Конец конструктора CommissionEmployee
24
25 // Свойство для получения имени работника (только для чтения)
26 public string FirstName
27 {
28     get
29     {
30         return firstName;
31     } // Конец get
32 } // Конец свойства FirstName
33
34 // Свойство для получения фамилии работника (только для чтения)
35 public string LastName
36 {
37     get
38     {
39         return lastName;
40     } // Конец get
41 } // Конец свойства LastName
42
43 // Свойство для получения номера социального страхования
44 // работника (только для чтения)
45 public string SocialSecurityNumber
46 {
47     get
48     {
49         return socialSecurityNumber;
50     } // Конец get
51 } // Конец свойства SocialSecurityNumber
52
53 // Свойство для чтения и записи объема продаж
54 public decimal GrossSales
55 {
56     get
57     {
58         return grossSales;
59     } // Конец get
60     set
61     {
62         if ( value >= 0 )
63             grossSales = value;
64         else
65             throw new ArgumentOutOfRangeException(
66                 "GrossSales", value, "GrossSales must be >= 0" );
67     } // Конец set
68 } // Конец свойства GrossSales
69
70 // Свойство для чтения и записи комиссионной ставки
71 public decimal CommissionRate
72 {
73     get
```

**Ил. 11.12.** Класс `CommissionEmployee` представляет работника, получающего комиссионные (продолжение ↗)

```

74     {
75         return commissionRate;
76     } // Конец get
77     set
78     {
79         if ( value > 0 && value < 1 )
80             commissionRate = value;
81         else
82             throw new ArgumentOutOfRangeException( "CommissionRate",
83                 value, "CommissionRate must be > 0 and < 1" );
84     } // Конец set
85 } // Конец свойства CommissionRate
86
87 // Вычисление заработка
88 public virtual decimal Earnings()
89 {
90     return CommissionRate * GrossSales ;
91 } // end method Earnings
92
93 // Получение строкового представления объекта CommissionEmployee
94 public override string ToString()
95 {
96     return string.Format(
97         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}",
98         "commission employee", FirstName, LastName,
99         "social security number", SocialSecurityNumber,
100        "gross sales", GrossSales, "commission rate", CommissionRate);
101 } // Конец метода ToString
102 } // Конец класса CommissionEmployee

```

**Ил. 11.12.** Класс `CommissionEmployee` представляет работника, получающего комиссионные (окончание)

В реализациях методов класса `BasePlusCommissionEmployee` (см. ил. 11.13) внесены некоторые изменения, отличающие их от версий на ил. 11.10. Методы `Earnings` (см. ил. 11.13, строки 39–42) и `ToString` (строки 45–49) для получения базовой зарплаты вызывают `get`-метод свойства `BaseSalary` вместо того, чтобы обращаться к `baseSalary` напрямую. Если мы решим переименовать переменную экземпляра `baseSalary`, то достаточно будет внести изменения в теле свойства `BaseSalary`.

```

1 // Ил. 11.13: BasePlusCommissionEmployee.cs
2 // Класс BasePlusCommissionEmployee наследует от CommissionEmployee
3 // и работает с закрытыми данными CommissionEmployee
4 // через его открытые свойства.
5 using System;
6
7 public class BasePlusCommissionEmployee : CommissionEmployee
8 {
9     private decimal baseSalary; // Базовая зарплата
10
11     // Конструктор производного класса с шестью параметрами
12     // вызывает конструктор базового класса CommissionEmployee

```

**Ил. 11.13.** Класс `BasePlusCommissionEmployee`, производный от `CommissionEmployee`, может обращаться к закрытым данным `CommissionEmployee` через открытые свойства (продолжение ↗)

```

13 public BasePlusCommissionEmployee( string first, string last,
14     string ssn, decimal sales, decimal rate, decimal salary )
15     : base( first, last, ssn, sales, rate )
16 {
17     BaseSalary = salary; // Проверка базовой зарплаты через свойство
18 } // Конец конструктора BasePlusCommissionEmployee
19
20 // Свойство для чтения и записи
21 // базовой зарплаты BasePlusCommissionEmployee
22 public decimal BaseSalary
23 {
24     get
25     {
26         return baseSalary;
27     } // Конец get
28     set
29     {
30         if ( value >= 0 )
31             baseSalary = value;
32         else
33             throw new ArgumentOutOfRangeException( "BaseSalary",
34                 value, "BaseSalary must be >= 0" );
35     } // Конец set
36 } // Конец свойства BaseSalary
37
38 // Вычисление заработка
39 public override decimal Earnings()
40 {
41     return BaseSalary + base.Earnings();
42 } // Конец метода Earnings
43
44 // Получение строкового представления BasePlusCommissionEmployee
45 public override string ToString()
46 {
47     return string.Format( "base-salaried {0}\nbase salary: {1:C}",
48         base.ToString(), BaseSalary );
49 } // Конец метода ToString
50 } // Конец класса BasePlusCommissionEmployee

```

**Ил. 11.13.** Класс `BasePlusCommissionEmployee`, производный от `CommissionEmployee`, может обращаться к закрытым данным `CommissionEmployee` через открытые свойства (окончание)

### Метод `Earnings` класса `BasePlusCommissionEmployee`

Метод `Earnings` класса `BasePlusCommissionEmployee` (см. ил. 11.13, строки 39–42) переопределяет метод `Earnings` класса `CommissionEmployee` (см. ил. 11.12, строки 88–91) для вычисления заработка `BasePlusCommissionEmployee`. Новая версия получает комиссионную часть заработка, вызывая метод `Earnings` класса `CommissionEmployee` в выражении `base.Earnings()` (см. ил. 11.13, строка 41), после чего прибавляет к полученному значению базовую зарплату; сумма равна общему заработку. Обратите внимание на синтаксис вызова метода базового класса из производного класса — перед именем метода базового класса ставится ключевое слово `base` и оператор «точка» (`.`). Этот механизм вызова тоже является признаком

качественного проектирования — вызов метода `Earnings` класса `CommissionEmployee` в методе `Earnings` класса `BasePlusCommissionEmployee` для вычисления части заработка объекта `BasePlusCommissionEmployee` предотвращает дублирование кода и упрощает сопровождение.



### ТИПИЧНАЯ ОШИБКА 11.3

При переопределении метода версия производного класса часто вызывает версию базового класса для выполнения части работы. Если вы забудете использовать префикс метода базового класса (ключевое слово `base` и оператор «точка») при обращении к методу базового класса, то метод производного класса вызовет сам себя, создавая бесконечную рекурсию.

### Метод `ToString` класса `BasePlusCommissionEmployee`

Метод `ToString` класса `BasePlusCommissionEmployee` (см. ил. 11.13, строки 45–49) переопределяет одноименный метод `CommissionEmployee` (см. ил. 11.12, строки 94–101) для получения строкового представления, подходящего для работника, получающего базовую зарплату вместе с комиссионными. Новая версия создает часть строкового представления `BasePlusCommissionEmployee` (строку `"commission employee"` и значения закрытых переменных экземпляров `CommissionEmployee`), вызывая метод `ToString` класса `CommissionEmployee` выражением `base.ToString()` (см. ил. 11.13, строка 48). Затем метод `ToString` производного класса выводит оставшуюся часть строкового представления объекта (то есть базовую зарплату).

### Класс `BasePlusCommissionEmployeeTest`

В листинге на ил. 11.14 с объектом `BasePlusCommissionEmployee` выполняются те же операции, что и на ил. 11.7 и 11.11. Хотя в обоих случаях классы выдают одинаковые результаты, версия `BasePlusCommissionEmployee` в этом примере обладает более качественной архитектурой за счет применения наследования и свойств, скрывающих данные и проверяющих целостность состояния объекта.

```

1 // Ил. 11.14: BasePlusCommissionEmployeeTest.cs
2 // Тестовый класс для BasePlusCommissionEmployee.
3 using System;
4
5 public class BasePlusCommissionEmployeeTest
6 {
7     public static void Main( string[] args )
8     {
9         // Создание объекта BasePlusCommissionEmployee
10        BasePlusCommissionEmployee employee =
11            new BasePlusCommissionEmployee( "Bob", "Lewis",
12                "333-33-3333", 5000.00M, .04M, 300.00M );
13
14        // Вывод данных BasePlusCommissionEmployee
15        Console.WriteLine(
16            "Employee information obtained by properties and methods: \n" );
17        Console.WriteLine( "First name is {0}", employee.FirstName );

```

**Ил. 11.14.** Тестовый класс для `BasePlusCommissionEmployee` (продолжение ↗)

```

18     Console.WriteLine( "Last name is {0}", employee.LastName );
19     Console.WriteLine( "Social security number is {0}",
20         employee.SocialSecurityNumber );
21     Console.WriteLine( "Gross sales are {0:C}", employee.GrossSales );
22     Console.WriteLine( "Commission rate is {0:F2}",
23         employee.CommissionRate );
24     Console.WriteLine( "Earnings are {0:C}", employee.Earnings() );
25     Console.WriteLine( "Base salary is {0:C}", employee.BaseSalary );
26
27     employee.BaseSalary = 1000.00M; // Присваивание базовой зарплаты
28
29     Console.WriteLine( "\n{0}:\n\n{1}",
30         "Updated employee information obtained by ToString", employee );
31     Console.WriteLine( "earnings: {0:C}", employee.Earnings() );
32 } // Конец Main
33 } // Конец класса BasePlusCommissionEmployeeTest

```

Employee information obtained by properties and methods:

```

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales are $5,000.00
Commission rate is 0.04
Earnings are $500.00
Base salary is $300.00

```

Updated employee information obtained by ToString:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $1,000.00
earnings: $1,200.00

```

#### Ил. 11.14. Тестовый класс для BasePlusCommissionEmployee (окончание)

В этом разделе мы последовательно разработали серию примеров, демонстрирующих возможности улучшения архитектуры приложения за счет применения наследования. Вы узнали, как создать производный класс при помощи наследования, как использовать защищенные члены базовых классов для того, чтобы производный класс мог обращаться к унаследованным переменным экземпляров базового класса, и как переопределить методы базового класса для определения версий, более подходящих для объектов производных классов. Кроме того, применение средств проектирования, описанные в главах 4, 10 и в этой главе, позволило нам создать классы, простые в сопровождении, модификации и отладке.

## 11.5. Конструкторы в производных классах

Как объяснялось в предыдущем разделе, создание экземпляра производного класса запускает цепочку вызовов конструкторов. Конструктор производного класса перед выполнением собственных операций вызывает конструктор своего прямого



базового класса — либо прямо (через инициализацию конструктора со ссылкой `base`), либо неявно (вызовом конструктора по умолчанию или конструктора без параметров базового класса). Если базовый класс тоже является производным от другого класса (как любой класс, кроме `object`), конструктор базового класса вызывает конструктор следующего класса в иерархии и т. д. Последним конструктором в цепочке вызовов всегда является конструктор класса `object`. Тело исходного конструктора производного класса выполняется в последнюю очередь. Каждый из конструкторов базового класса работает с переменными экземпляров базового класса, унаследованными объектом производного класса. Для примера возьмем иерархию `CommissionEmployee–BasePlusCommissionEmployee` на ил. 11.12 и 11.13. Когда в приложении создается объект `BasePlusCommissionEmployee`, вызывается конструктор `BasePlusCommissionEmployee`. Этот конструктор немедленно вызывает конструктор `CommissionEmployee`, который, в свою очередь, неявно вызывает конструктор `object`. Конструктор класса `object` имеет пустое тело, поэтому он немедленно возвращает управление конструктору `CommissionEmployee`, который инициализирует закрытые переменные `CommissionEmployee`, входящие в объект `BasePlusCommissionEmployee`. Когда выполнение конструктора `CommissionEmployee` завершается, он возвращает управление конструктору `BasePlusCommissionEmployee`, который инициализирует поле `baseSalary` объекта `BasePlusCommissionEmployee`.

## 11.6. Применение наследования при проектировании программных продуктов

В этом разделе рассматривается применение наследования при проектировании. При расширении существующего класса новый класс наследует члены существующего класса. Новый класс можно дополнить новыми членами и переопределить члены базового класса. При этом программисту производного класса не приходится изменять исходный код базового класса. Чтобы откомпилировать и выполнить любое приложение, использующее и расширяющее базовый класс, C# достаточно иметь доступ к откомпилированному коду базового класса. Эта возможность чрезвычайно полезна для независимых разработчиков, которые могут разрабатывать классы для продажи или лицензирования и распространять их в форме библиотек. Пользователи создают новые классы, объявляя их производными от классов из библиотеки, не обращаясь к закрытому исходному коду библиотеки.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 11.5

Хотя для создания производного класса не обязательно иметь доступ к исходному коду базового класса, разработчики часто хотят видеть исходный код, чтобы разобраться в реализации класса — например, убедиться в том, что расширяемый класс работает правильно и надежно.

Начинающим программистам трудно оценить масштаб проблем, с которыми сталкиваются проектировщики, работающие над крупными проектами. Опытные специалисты считают, что повторное использование кода повышает эффективность

процесса разработки. Объектно-ориентированное программирование упрощает повторное использование и способствует сокращению времени разработки. Основные, хорошо проработанные библиотеки классов помогают извлечь максимум пользы из повторного использования кода посредством наследования.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 11.6

На стадии проектирования объектно-ориентированной системы часто выясняется, что между некоторыми классами существует тесная связь. Проектировщик должен «выделить» общие члены и поместить их в базовый класс. Далее наследование используется для разработки производных классов и их специализации помимо возможностей, унаследованных от базового класса.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 11.7

Объявление производного класса не влияет на исходный код базового класса. Таким образом, наследование сохраняет целостность базового класса.

Читать объявления производного класса бывает непросто, потому что унаследованные члены не объявляются явно в производных классах, но тайно присутствуют в них. Похожая проблема существует при документировании членов производных классов.

## 11.7. Класс object

Как упоминалось ранее в этой главе, все классы прямо или косвенно наследуют от класса `object` (`System.Object` в Framework Class Library), поэтому семь методов `object` наследуются всеми классами. Дополнительную информацию о методах `object` можно получить по адресу [msdn.microsoft.com/en-us/library/system.object.aspx](http://msdn.microsoft.com/en-us/library/system.object.aspx).

Метод	Описание
<code>Equals</code>	Метод проверяет два объекта на равенство и возвращает <code>true</code> , если они равны, или <code>false</code> в противном случае. В аргументе метода передается произвольный объект. Если объекты некоторого класса должны проверяться на равенство, класс должен переопределить метод <code>Equals</code> для сравнения содержимого двух объектов. Правила переопределения <code>Equals</code> объясняются на сайте <a href="http://bit.ly/OverridingEqualsCSharp">http://bit.ly/OverridingEqualsCSharp</a>
<code>Finalize</code>	Явное объявление или вызов этого метода запрещены. Если класс содержит деструктор, компилятор неявно переименовывает его для переопределения защищенного метода <code>Finalize</code> , который вызывается только уборщиком мусора перед освобождением памяти объекта. Освобождение объекта уборщиком мусора не гарантировано, поэтому выполнение метода <code>Finalize</code> объекта не гарантировано. При выполнении метода <code>Finalize</code> производного класса он выполняет свою задачу, а затем вызывает метод <code>Finalize</code> базового класса. В общем случае использовать метод <code>Finalize</code> не рекомендуется

**Ил. 11.15.** Методы `object`, прямо или косвенно наследуемые всеми классами (продолжение ↗)

Метод	Описание
GetHashCode	Структура данных, называемая хеш-таблицей, связывает объекты-ключи с объектами-значениями. Класс Hashtable рассматривается в главе 21. При исходной вставке значения в хеш-таблицу вызывается метод GetHashCode. Возвращаемое значение используется хеш-таблицей для определения позиции вставки соответствующего значения. Хеш-код также используется хеш-таблицей при выборке соответствующего значения
GetType	Каждый объект может определить свой фактический тип во время выполнения. Метод GetType (см. раздел 12.5) возвращает объект типа Type (пространство имен System) с информацией о типе объекта, включающей имя класса (свойство FullName объекта Type)
MemberwiseClone	Защищенный метод вызывается без аргументов и возвращает ссылку на объект; он создает копию объекта, для которого был вызван. Реализация метода выполняет поверхностное копирование — значения переменных экземпляров одного объекта копируются в другой объект того же типа. Для ссылочных типов копируются только ссылки
ReferenceEquals	Статический метод получает две ссылки на объект и возвращает true, если они относятся к одному экземпляру или являются null-ссылками. В противном случае возвращается false
ToString	Этот метод (см. раздел 7.4) возвращает представление объекта в формате string. Реализация метода по умолчанию возвращает пространство имен, за которым следует точка и имя класса

**Ил. 11.15.** Методы object, прямо или косвенно наследуемые всеми классами (окончание)

## 11.8. Итоги

В этой главе представлена концепция *наследования* — механизма создания классов, которые вмещают члены существующих классов и дополняют их новыми возможностями. Мы рассмотрели понятия базовых и производных классов и создали производный класс, наследующий члены от базового класса. Также был представлен модификатор доступа `protected`; защищенные члены базового класса доступны для членов базовых классов. Вы узнали, как обращаться к членам базового класса при помощи ключевого слова `base` и как используются конструкторы в иерархиях наследования. Глава завершается описанием методов класса `object`, прямого или косвенного базового класса для всех классов.

Глава 12 посвящена полиморфизму — объектно-ориентированной концепции, позволяющей писать приложения для обобщенной обработки объектов разных классов, связанных отношениями наследования.

# 12 ООП: полиморфизм, интерфейсы и перегрузка операторов

## 12.1. Введение

В этой главе мы продолжим изучать объектно-ориентированное программирование и рассмотрим применение полиморфизма в иерархиях наследования. Полиморфизм позволяет программировать на уровне общих понятий, а не конкретики. В частности, полиморфизм позволяет писать приложения для обработки объектов разных классов, имеющих общий базовый класс, как если бы все они были объектами базового класса.

Рассмотрим пример. Предположим, мы создаем приложение для моделирования нескольких видов животных в биологическом исследовании. Классы `Fish`, `Frog` и `Bird` представляют типы исследуемых животных. Каждый класс расширяет базовый класс `Animal`, содержащий метод `Move` и текущее местонахождение животного в координатах `xyz`. Каждый производный класс реализует метод `Move`. В приложении поддерживается массив ссылок на объекты разных классов, производных от `Animal`. Для моделирования перемещения животного приложение каждую секунду отправляет каждому объекту сообщение `Move`. Каждый конкретный тип `Animal` реагирует на `Move` определенным образом — рыба (`Fish`) может проплыть полметра, лягушка (`Frog`) — прыгнуть на метр, а птица (`Bird`) — пролететь три метра. Приложение передает объектам сообщение `Move` одинаково, а каждый объект изменяет свои координаты `xyz` в соответствии с конкретным типом движения. По сути, мы полагаемся на то, что каждый объект выполнит «правильное» действие в ответ на вызов одного метода; в этом проявляется ключевая концепция полиморфизма. Отправка одного сообщения (`Move` в данном случае) разным объектам приводит к разным результатам.

### Простота расширения систем

Системы, спроектированные и реализованные с использованием полиморфизма, легко расширяются — добавление новых классов практически не требует изменения

общих частей приложения (при условии, что новые классы являются частью иерархии наследования, которая обрабатывается приложением на общем уровне). При введении новых классов изменяются только те части приложения, которым необходима непосредственная информация о новых классах, добавляемых в иерархию. Например, при создании класса `Tortoise`, производного от `Animal`, необходимо написать только класс `Tortoise` и часть модели, создающую экземпляр `Tortoise`. Части модели, работающие с `Animal` на общем уровне, остаются неизменными.

Эта глава состоит из нескольких частей. Сначала мы обсудим несколько примеров полиморфизма, а затем разработаем реальный пример использования полиморфного поведения. Как вы вскоре увидите, ссылки на базовые классы могут использоваться для работы с объектами как базовых, так и производных классов.

### Полиморфная иерархия наследования

В следующем примере будет использована иерархия `Employee` из раздела 11.4.5. Мы создадим простое приложение, которое осуществляет полиморфное вычисление еженедельного заработка разных типов работников, вызывая метод `Earnings` для каждого работника. Хотя заработок каждого типа работников вычисляется особым образом, полиморфизм позволяет действовать на уровне «работников вообще». В нашем примере иерархия дополняется двумя новыми классами: `SalariedEmployee` (для работников, получающих фиксированную зарплату) и `HourlyEmployee` (для работников с почасовой оплатой и начислениями за сверхурочные). Общая функциональность всех классов обновленной иерархии будет объявлена в абстрактном классе `Employee` (раздел 12.5.1), от которого классы `SalariedEmployee`, `HourlyEmployee` и `CommissionEmployee` наследуют напрямую, а класс `BasePlusCommissionEmployee` наследует косвенно. Как вы вскоре увидите, при вызове метода `Earnings` каждого работника через ссылку на объект базового класса `Employee` механизм полиморфного выполнения C# обеспечивает выполнение правильного метода.

### Определение типа объекта во время выполнения

В отдельных ситуациях при выполнении полиморфной обработки приходится программировать на «конкретном» уровне. Наш пример показывает, что приложение может определить фактический тип объекта во время выполнения и выполнить соответствующие действия. Эта возможность будет использована для определения того, относится ли объект конкретного работника к типу `BasePlusCommissionEmployee`, и если относится, — для повышения базовой зарплаты этого работника на 10 %.

### Интерфейсы

В этой главе также продолжается наше знакомство с темой интерфейсов C#. Интерфейс описывает набор методов и свойств, которые могут вызываться для объекта, но не предоставляет конкретной реализации. Разработчик может объявлять классы, которые реализуют (то есть предоставляют конкретные реализации методов и свойств) один или несколько интерфейсов. Каждый член интерфейса должен быть объявлен во всех классах, реализующих этот интерфейс. Если класс реализует интерфейс, все объекты класса находятся в отношении «является частным случаем»

с типом интерфейса, а все объекты класса заведомо предоставляют функциональность, описанную интерфейсом. Сказанное также распространяется на все классы, производные от этого класса.

Интерфейсы особенно полезны для назначения общей функциональности в классах, которые могут быть не связаны друг с другом. Это позволяет организовать полиморфную обработку объектов несвязанных классов — объекты классов, реализующих один интерфейс, могут реагировать на одни и те же вызовы методов. Чтобы продемонстрировать возможности создания и использования интерфейсов, мы изменим бухгалтерское приложение так, чтобы оно могло вычислять причитающиеся платежи по заработкам работников компании и по счетам за приобретенные товары. Как вы увидите, интерфейсы позволяют выполнять полиморфные вызовы по аналогии с наследованием.

### Перегрузка операторов

Глава завершается описанием перегрузки операторов. В предшествующих главах мы объявили собственные классы и использовали методы для выполнения операций с объектами этих классов. Механизм перегрузки операторов позволяет определить поведение встроенных операторов (таких, как `+`, `-` и `<`) с объектами наших собственных классов. Запись с операторами при выполнении некоторых операций с объектами получается намного более удобной, чем запись с вызовами методов.

## 12.2. Примеры полиморфизма

Рассмотрим несколько дополнительных примеров полиморфизма.

### Иерархия четырехугольников

Если класс `Rectangle` (прямоугольник) объявляется производным от класса `Quadrilateral` (четыреугольник), то по сути он представляет собой специализированную версию `Quadrilateral`. Любая операция (например, вычисление длины периметра), которая может быть выполнена с объектом `Quadrilateral`, также может выполняться с объектом `Rectangle`. Эти операции также могут выполняться и с другими специализированными версиями `Quadrilateral`: квадратами (`Square`), параллелограммами (`Parallelogram`) и трапециями (`Trapezoid`). Полиморфизм работает при вызове метода через переменную базового класса — во время выполнения вызывается версия метода, принадлежащая правильному производному классу. Простой пример, демонстрирующий этот процесс, приводится в разделе 12.3.

### Иерархия наследования объектов в видеоигре

Другой пример: предположим, мы проектируем видеоигру, в которой задействованы объекты классов `Martian`, `Venusian`, `Plutonian`, `SpaceShip`, `LaserBeam` и т. д. Предположим, все классы являются производными от общего базового класса `SpaceObject`, содержащего метод `Draw`. Метод реализуется всеми производными классами. В приложении поддерживается коллекция (например, массив `SpaceObject`) ссылок

на объекты различных классов. Чтобы обновить изображение, менеджер экрана периодически отправляет всем объектам одно и то же сообщение `Draw`. Каждый объект реагирует на это сообщение по-своему. Например, объект `Martian` рисует красную фигуру, объект `SpaceShip` рисует серебристую «летающую тарелку», а объект `LaserBeam` рисует на экране ярко-красный луч. И снова отправка одного сообщения (в данном случае `Draw`) разным объектам приводит к различным результатам.

Полиморфный менеджер экрана может использовать полиморфизм для добавления в систему новых классов с минимальными изменениями в коде системы. Допустим, мы хотим добавить в свою видеоигру новый вид объектов `Mercurian`. Для этого необходимо построить класс `Mercurian`, который расширяет `SpaceObject` и предоставляет свою реализацию метода `Draw`. Когда объекты класса `Mercurian` появляются в коллекции `SpaceObject`, код менеджера экрана вызывает метод `Draw` точно так же, как он делает для всех остальных объектов в коллекции независимо от типа, то есть новые объекты `Mercurian` просто подключаются к системе, не требуя никаких изменений в коде менеджера экрана. Таким образом, разработчик может добавлять в систему новые типы, которые не были запланированы при ее создании, без изменения самой системы (не считая построения новых классов и изменения кода создания новых объектов).



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 12.1

Полиморфизм упрощает расширение системы: программа с полиморфным поведением не зависит от типов объектов, которым отправляются сообщения. Новые типы, способные реагировать на вызовы существующих методов, встраиваются в систему без изменения ее программной базы. Все изменения вносятся только в клиентском коде, создающем экземпляры новых объектов.

## 12.3. Демонстрация полиморфного поведения

В разделе 11.4 была создана иерархия классов, в которой класс `BasePlusCommissionEmployee` наследовал от класса `CommissionEmployee`. Примеры этого раздела работали с объектами `CommissionEmployee` и `BasePlusCommissionEmployee`, вызывая их методы по ссылкам на объекты. Для обращения к объектам базового класса использовались ссылки на базовый класс, а для обращения к объектам производного класса — ссылки на производный класс. Такие операции выглядят естественно и понятно, однако существуют и другие возможности.

В следующем примере для обращения к объекту производного класса используется ссылка на объект базового класса. Затем вы увидите, как вызов метода объекта производного класса через ссылку на объект базового класса может активизировать функциональность производного класса — выбор метода зависит от фактического типа объекта, на который указывает ссылка, а не от типа ссылки. В этом проявляется важнейший принцип: *объект производного класса может интерпретироваться как объект базового класса*, что открывает много интересных возможностей. Приложение может создать массив ссылок на базовый класс, которые на самом деле

указывают на объекты многих производных классов. Это возможно потому, что каждый объект производного класса является объектом базового класса. Например, переменной базового класса `CommissionEmployee` можно присвоить ссылку на объект `BasePlusCommissionEmployee`, потому что `BasePlusCommissionEmployee` является частным случаем `CommissionEmployee`, а, следовательно, `BasePlusCommissionEmployee` может рассматриваться как `CommissionEmployee`.

Объект базового класса не может рассматриваться как объект любого из своих производных классов. Например, ссылку на объект `CommissionEmployee` нельзя напрямую присвоить переменной производного класса `BasePlusCommissionEmployee`, потому что `CommissionEmployee` не является частным случаем `BasePlusCommissionEmployee` — в частности, `CommissionEmployee` не содержит переменную экземпляра `baseSalary` и свойство `BaseSalary`. Отношение «является частным случаем» направлено от производного класса к прямому и косвенным базовым классам, но не наоборот.

Компилятор позволит присвоить ссылку на базовый класс переменной производного класса, если выполнить явное преобразование ссылки на базовый класс к типу производного класса (эта возможность более подробно рассматривается в разделе 12.5.6). Зачем выполнять такое присваивание? Ссылка на базовый класс может использоваться для вызова только методов, объявленных в базовом классе, — попытка вызова метода только производного класса по ссылке на базовый класс приводит к ошибке компиляции. Если приложению потребуется выполнить операцию, определенную только в производном классе, с объектом производного класса через ссылку на объект базового класса, приложение должно сначала преобразовать ссылку на базовый класс в ссылку на производный класс с использованием приема *понижающего преобразования* (downcasting). Это позволяет приложению вызывать методы производного класса, отсутствующие в базовом классе. Пример понижающего преобразования представлен в разделе 12.5.6.

В листинге на ил. 12.1 представлены три способа использования переменных базовых и производного классов для хранения ссылок на объекты базового и производного классов. Первые два примера тривиальны — как и в разделе 11.4, мы присваиваем ссылку на базовый класс переменной базового класса, а ссылку на производный класс — переменной производного класса. Затем отношения между производными и базовым классом (то есть отношения «является частным случаем») демонстрируются присваиванием ссылки на производный класс переменной базового класса. [*Примечание:* в приложении используются классы `CommissionEmployee` и `BasePlusCommissionEmployee` из ил. 11.12 и 11.13 соответственно.]

```
1 // Ил. 12.1: PolymorphismTest.cs
2 // Присваивание ссылок на базовый и производный классы
3 // переменным базового и производного классов.
4 using System;
5
6 public class PolymorphismTest
7 {
```

**Ил. 12.1.** Присваивание ссылок на базовый и производный классы переменным базового и производного классов (продолжение ↗)



```

8 public static void Main( string[] args )
9 {
10     // Присваивание переменной базового класса ссылки на базовый класс
11     CommissionEmployee commissionEmployee = new CommissionEmployee(
12         "Sue", "Jones", "222-22-2222", 10000.00M, .06M );
13
14     // Присваивание переменной производного класса ссылки на производный
15     // класс
16     BasePlusCommissionEmployee basePlusCommissionEmployee =
17         new BasePlusCommissionEmployee( "Bob", "Lewis",
18         "333-33-3333", 5000.00M, .04M, 300.00M );
19
20     // Вызов ToString и Earnings для объекта базового класса
21     // через переменную базового класса
22     Console.WriteLine( "{0} {1}:\n{n}{2}\n{3}: {4:C}\n",
23         "Call CommissionEmployee's ToString and Earnings methods ",
24         "with base-class reference to base class object",
25         commissionEmployee.ToString(),
26         commissionEmployee.Earnings() );
27
28     // Вызов ToString и Earnings для объекта производного класса
29     // через переменную производного класса
30     Console.WriteLine( "{0} {1}:\n{n}{2}\n{3}: {4:C}\n",
31         "Call BasePlusCommissionEmployee's ToString and Earnings ",
32         "methods with derived class reference to derived-class object",
33         basePlusCommissionEmployee.ToString(),
34         basePlusCommissionEmployee.Earnings() );
35
36     // Вызов ToString и Earnings для объекта производного класса
37     // через переменную базового класса
38     CommissionEmployee commissionEmployee2 =
39         basePlusCommissionEmployee;
40     Console.WriteLine( "{0} {1}:\n{n}{2}\n{3}: {4:C}",
41         "Call BasePlusCommissionEmployee's ToString and Earnings ",
42         "with base class reference to derived-class object",
43         commissionEmployee2.ToString(), "earnings",
44         commissionEmployee2.Earnings() );
45 } // Конец Main
46 } // Конец класса PolymorphismTest

```

Call CommissionEmployee's ToString and Earnings methods with base class reference to base class object:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: $10,000.00
commission rate: 0.06
earnings: $600.00

```

Call BasePlusCommissionEmployee's ToString and Earnings methods with derived class reference to derived class object:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333

```

**Ил. 12.1.** Присваивание ссылок на базовый и производный классы переменным базового и производного классов (продолжение ↗)

```
gross sales: $5,000.00  
commission rate: 0.04  
base salary: $300.00  
earnings: $500.00
```

Call `BasePlusCommissionEmployee`'s `ToString` and `Earnings` methods with base class reference to derived class object:

```
base-salaried commission employee: Bob Lewis  
social security number: 333-33-3333  
gross sales: $5,000.00  
commission rate: 0.04  
base salary: $300.00  
earnings: $500.00
```

**Ил. 12.1.** Присваивание ссылок на базовый и производный классы переменным базового и производного классов (окончание)

В строках 11–12 создается объект `CommissionEmployee`, а ссылка на него присваивается переменной `CommissionEmployee`. В строках 15–17 создается объект `BasePlusCommissionEmployee`, а ссылка на него присваивается переменной `BasePlusCommissionEmployee`. Такое присваивание естественно — например, переменная `CommissionEmployee` предназначена прежде всего для хранения ссылки на объект `CommissionEmployee`. Строки 21–25 используют ссылку `commissionEmployee` для вызова методов `ToString` и `Earnings`. Так как переменная `commissionEmployee` ссылается на объект `CommissionEmployee`, вызываются версии этих методов из базового класса `CommissionEmployee`. Аналогичным образом в строках 29–33 ссылка `basePlusCommissionEmployee` используется для вызова методов `ToString` и `Earnings` объекта `BasePlusCommissionEmployee`. При этом вызываются версии этих методов производного класса `BasePlusCommissionEmployee`.

Затем в строках 37–38 ссылка на объект производного класса `basePlusCommissionEmployee` присваивается переменной базового класса `CommissionEmployee`, которая используется в строках 39–43 для вызова методов `ToString` и `Earnings`. Когда переменная базового класса, которая содержит ссылку на объект производного класса, используется для вызова виртуального метода, будет вызвана переопределенная версия метода производного класса. Следовательно, выражение `commissionEmployee2.ToString()` в строке 42 вызывает версию метода `ToString` производного класса `BasePlusCommissionEmployee`. Компилятор допускает подобные «переключения», потому что объект производного класса является частным случаем объекта базового класса (но не наоборот). Обнаруживая вызов виртуального метода через переменную, компилятор проверяет тип переменной для определения возможности вызова метода. Если класс содержит объявление метода (или наследует его), компилятор разрешает компиляцию метода. Во время выполнения используемый метод выбирается в зависимости от фактического типа объекта, на который ссылается переменная.

## 12.4. Абстрактные классы и методы

Обычно при создании класса предполагается, что приложения будут создавать объекты этого класса. Однако в некоторых ситуациях бывает полезно объявлять

классы, не предназначенные для создания объектов. Такие классы называются *абстрактными*. Так как они используются только в качестве базовых классов в иерархиях наследования, мы будем называть их *абстрактными базовыми классами*. Эти классы не могут использоваться для создания объектов, потому что, как вы вскоре увидите, абстрактные классы не завершены — производные классы должны предоставить «недостающие части». Использование абстрактных классов будет продемонстрировано в разделе 12.5.1.

### Назначение абстрактных базовых классов

Абстрактные классы используются прежде всего для создания базовых классов, которые расширяются другими классами, наследующими набор общих характеристик. Так, в иерархии `Shape` на ил. 11.3 производные классы наследуют общие характеристики геометрической фигуры: общие атрибуты (координаты, цвет, толщина границы) и общее поведение (прорисовка, перемещение, изменение размеров, изменение цвета). Классы, которые могут использоваться для создания объектов, называются *конкретными классами*. Такие классы предоставляют реализации всех объявляемых в них методов (некоторые реализации могут наследоваться). Например, на основе абстрактного базового класса `TwoDimensionalShape` можно создать производные конкретные классы `Circle`, `Square` и `Triangle`. Абстрактные базовые классы слишком «расплывчаты» для создания реальных объектов — они определяют только набор общих характеристик для производных классов. Для создания объектов требуется более точное описание. Например, если отправить сообщение `Draw` абстрактному классу `TwoDimensionalShape`, класс будет знать, что двумерная фигура должна поддерживать прорисовку, но не будет знать, какую именно фигуру следует вывести, и поэтому не сможет реализовать «настоящий» метод `Draw`. Конкретные классы предоставляют те подробности, с которыми становится возможным создание объектов.

### Использование абстрактных базовых классов в клиентском коде

Не во всех иерархиях наследования присутствуют базовые классы. Однако достаточно часто программист пишет клиентский код, использующий только абстрактные базовые классы, для сокращения зависимости клиентского кода от набора конкретных производных классов. Например, можно написать метод с параметром, относящимся к типу абстрактного базового класса. При вызове такого метода можно передать объект любого конкретного класса, прямо или косвенно расширяющего базовый класс, заданный как тип параметра.

### Множественные уровни абстрактных базовых классов в иерархии

Абстрактные классы иногда образуют несколько уровней иерархии. Например, иерархия на рис. 11.3 начинается с абстрактного класса `Shape`. На следующем уровне иерархии находятся еще два абстрактных класса `TwoDimensionalShape` и `ThreeDimensionalShape`. Следующий уровень составляют *конкретные* классы — специализации `TwoDimensionalShape` (`Circle`, `Square` и `Triangle`) и `ThreeDimensionalShape` (`Sphere`, `Cube` и `Tetrahedron`).

## Создание абстрактного класса

Абстрактные классы объявляются с ключевым словом `abstract`. Абстрактный класс обычно содержит один или несколько *абстрактных методов*. В объявлении абстрактного метода присутствует ключевое слово `abstract`:

```
public abstract void Draw(); // Абстрактный метод
```

Абстрактные методы неявно являются виртуальными и не предоставляют реализаций. Класс, содержащий абстрактные методы, должен объявляться абстрактным даже в том случае, если в нем присутствуют конкретные (не абстрактные) методы. Каждый конкретный производный класс абстрактного базового класса также должен предоставлять конкретные реализации абстрактных методов базового класса. Пример абстрактного класса с абстрактным методом представлен на ил. 12.4.

## Абстрактные свойства

Свойства тоже могут объявляться абстрактными или виртуальными и переопределяться в производных классах с ключевым словом `override`, как и методы. Это позволяет абстрактному базовому классу задавать общие свойства своих производных классов. Абстрактные свойства объявляются в форме:

```
public abstract TunСвойства ИмяСвойства
{
    get;
    set;
} // Конец абстрактного свойства
```

Символ «`;`» после ключевых слов `get` и `set` означает, что реализация этих методов не предоставляется. Конкретные производные классы должны предоставить реализацию для каждого метода доступа, объявленного в абстрактном свойстве. Если в свойстве указаны оба метода доступа, `get`- и `set`-, каждый конкретный производный класс должен реализовать оба метода. Если один из методов доступа отсутствует, он не может быть реализован в производном классе (такая попытка приводит к ошибке компиляции).

## Конструкторы и статические методы не могут быть абстрактными

Конструкторы и статические методы не могут объявляться абстрактными. Конструкторы не наследуются, поэтому абстрактный конструктор никогда не будет реализован. Производные классы не могут переопределять статические методы, так что абстрактный статический метод тоже не реализуется.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 12.2

Абстрактный класс объявляет общие атрибуты и поведение классов, следующих от него (прямо или косвенно) в иерархии классов. Как правило, абстрактный класс содержит один или несколько абстрактных методов или свойств, которые должны переопределяться в конкретных производных классах. Переменные экземпляров, конкретные методы и конкретные свойства абстрактного класса подчиняются стандартным правилам наследования.



### ТИПИЧНАЯ ОШИБКА 12.1

Попытка создания объекта абстрактного класса приводит к ошибке компиляции.



### ТИПИЧНАЯ ОШИБКА 12.2

Если производный класс не реализует абстрактные методы и свойства базового класса, происходит ошибка компиляции (если только производный класс тоже не объявлен абстрактным).

## Объявление переменных с типом абстрактного базового класса

Создать объект абстрактного базового класса невозможно, но вы вскоре увидите, что абстрактные базовые классы могут использоваться для объявления переменных, в которых хранятся ссылки на объекты конкретных классов, производных от этих абстрактных классов. Обычно такие приложения используются для полиморфной обработки объектов производных классов. Кроме того, имена абстрактных базовых классов могут использоваться для вызова статических методов, объявленных в этих абстрактных базовых классах.

## Полиморфизм и драйверы устройств

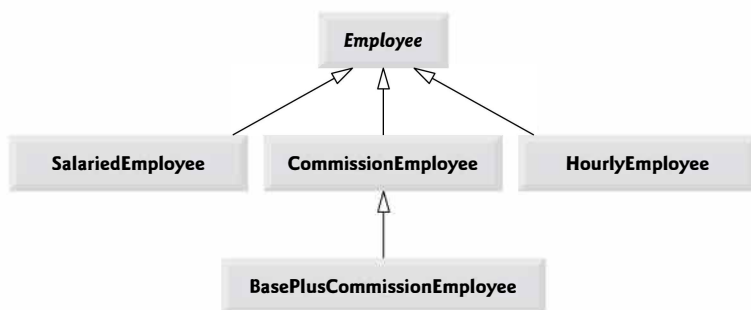
Полиморфизм особенно эффективен при реализации *многоуровневых программных систем*. Например, в операционной системе разные физические устройства могут работать по разным принципам, но даже в этом случае возможно определение общих команд чтения и записи данных на устройства. Для каждого устройства операционная система использует программный модуль, называемый *драйвером устройства*, для управления всем обменом данных между системой и устройством. Команда записи, отправленная объекту драйвера устройства, должна интерпретироваться особым образом в контексте этого драйвера и особенностей его работы с конкретным устройством. Тем не менее сама по себе команда записи не так уж сильно отличается для разных устройств в системе: устройству передается некоторое количество байтов из памяти. Объектно-ориентированная операционная система может использовать абстрактный базовый класс для определения «интерфейса», подходящего для всех драйверов устройств. Посредством наследования от абстрактного базового класса создаются производные классы со сходным поведением. Методы драйверов устройств объявляются как абстрактные методы в абстрактном базовом классе. Производные классы предоставляют реализации абстрактных методов, соответствующие конкретным видам драйверов устройств. На рынке постоянно появляются новые устройства, часто значительно позже выпуска операционной системы. Когда вы покупаете новое устройство, к нему прилагается драйвер от фирмы-разработчика устройства. Стоит подключить устройство к компьютеру и установить драйвер, как оно немедленно начинает работать. Это еще один элегантный пример расширения систем на базе полиморфизма.

## 12.5. Пример: система начисления заработка с использованием полиморфизма

В этом разделе мы вернемся к иерархии `CommissionEmployee-BasePlusCommissionEmployee` из раздела 11.4. На этот раз для выполнения вычислений в зависимости от типа работника будут использоваться абстрактный метод и полиморфизм. Мы создадим расширенную иерархию классов для решения следующей задачи.

Компания еженедельно начисляет заработок своим работникам. Работники делятся на четыре типа: одни получают фиксированную зарплату независимо от количества проработанных часов, другие получают почасовую оплату с премиальными за переработку сверх 40 часов, третьи получают процент от продаж, а четвертые получают базовую зарплату и процент от продаж. За текущий платежный период компания решила премировать работников четвертой категории, увеличив их базовую зарплату на 10 %. Требуется реализовать приложение, которое начисляет заработок с использованием полиморфных вызовов.

Мы используем абстрактный класс `Employee` для представления общей концепции работника. Он расширяется классами `SalariedEmployee`, `HourlyEmployee` и `CommissionEmployee` (первые три категории). Класс `BasePlusCommissionEmployee`, расширяющий `CommissionEmployee`, представляет последний тип работников. На ил. 12.2 представлена диаграмма наследования для нашего полиморфного приложения. Имя абстрактного класса `Employee` записано курсивом, как предписано правилами UML.



Ил. 12.2. Диаграмма классов UML для иерархии `Employee`

Абстрактный базовый класс `Employee` объявляет «интерфейс» иерархии, то есть набор членов, которые могут вызываться приложением для всех объектов `Employee`. В данном случае термин «интерфейс» используется в общем смысле для обозначения различных способов взаимодействия приложения с объектами любого класса, производного от `Employee`. Будьте внимательны и не путайте общую концепцию «интерфейса» с формальным синтаксисом интерфейсов C# (см. раздел 12.7). У каждого работника независимо от способа вычисления заработка есть имя, фамилия и номер

социального страхования, поэтому указанные данные присутствуют в абстрактном базовом классе `Employee`.

В следующих разделах мы реализуем иерархию классов `Employee`. Раздел 12.5.1 реализует абстрактный базовый класс `Employee`. В каждом из разделов 12.5.2–12.5.5 реализуется один из конкретных классов, а в разделе 12.5.6 мы создадим тестовое приложение, которое строит объекты всех классов и организует их полиморфную обработку.

### 12.5.1. Создание абстрактного базового класса `Employee`

Класс `Employee` (ил. 12.4) содержит методы `Earnings` и `ToString`, а также автоматические реализованные свойства для работы с данными `Employee`. Разумеется, метод вычисления заработка `Earnings` актуален для всех работников, но конкретный способ вычисления зависит от класса работника. По этой причине метод `Earnings` объявляется абстрактным в базовом классе `Employee`, потому что реализация по умолчанию для этого метода не имеет смысла — имеющейся информации недостаточно для определения того, какую сумму должен возвращать метод `Earnings`. Каждый производный класс переопределяет `Earnings` конкретной реализацией. Чтобы вычислить заработок работника, приложение присваивает ссылку на объект работника переменной базового класса `Employee`, а затем вызывает метод `Earnings` для этой переменной. Мы будем использовать массив переменных `Employee`, каждая из которых хранит ссылку на объект `Employee` (конечно, объекты `Employee` существовать не могут, потому что класс `Employee` является абстрактным, — однако из-за наследования все объекты классов, производных от `Employee`, могут рассматриваться как объекты `Employee`). Приложение перебирает элементы массива и вызывает метод `Earnings` для каждого объекта `Employee`. Эти вызовы обрабатываются полиморфно. Включение абстрактного метода `Earnings` в `Employee` заставляет каждый конкретный класс, производный от `Employee`, переопределить `Earnings` для выполнения соответствующего вычисления заработка.

Метод `ToString` класса `Employee` возвращает строку с именем, фамилией и номером социального страхования работника. Каждый класс, производный от `Employee`, переопределяет метод `ToString` для создания строкового представления объекта этого класса с указанием типа работника (например, "salaried employee:") и остальной информацией.

Строки таблицы на ил. 12.3 соответствуют пяти классам иерархии, а столбцы — методам `Earnings` и `ToString`. Для каждого класса на диаграмме показан предполагаемый результат каждого метода. [*Примечание:* свойства базового класса `Employee` не включены в таблицу, потому что они не переопределяются в производных классах — каждое свойство наследуется и используется «как есть» в каждом из производных классов.]

	Earnings	ToString
Employee	абстрактный	<i>firstName lastName</i> social security number: <i>SSN</i>
SalariedEmployee	weeklySalary	salaried employee: <i>firstName lastName</i> social security number: <i>SSN</i> weekly salary: <i>weeklSalary</i>
HourlyEmployee	если <i>hours</i> ≤ 40 wage * <i>hours</i> если <i>hours</i> > 40 40 * wage + ( <i>hours</i> - 40) * wage * 1.5	hourly employee: <i>firstName lastName</i> social security number: <i>SSN</i> hourly wage: <i>wage</i> hours worked: <i>hours</i>
CommissionEmployee	commissionRate * grossSales	commission employee: <i>firstName lastName</i> social security number: <i>SSN</i> gross sales: <i>grossSales</i> commission rate: <i>commissionRate</i>
BasePlusCommissionEmployee	( commissionRate * grossSales ) + baseSalary	base salaried commission employee: <i>firstName lastName</i> social security number: <i>SSN</i> gross sales: <i>grossSales</i> commission rate: <i>commissionRate</i> base salary: <i>baseSalary</i>

Ил. 12.3. Полиморфный интерфейс иерархии классов Employee

## Класс Employee

Рассмотрим объявление класса Employee (см. ил. 12.4). Класс включает конструктор, в аргументах которого передается имя, фамилия и номер социального страхования (строки 15–20); свойства с открытыми get-методами для получения имени, фамилии и номера социального страхования (строки 6, 9 и 12 соответственно); метод ToString (строки 23–27), использующий свойства для получения строкового представления Employee; и абстрактный метод Earnings (строка 30), который *должен* быть реализован в *конкретных* производных классах. Конструктор Employee в этом примере не проверяет номер социального страхования, хотя обычно такая проверка должна выполняться.

```

1 // Ил. 12.4: Employee.cs
2 // Абстрактный базовый класс Employee.
3 public abstract class Employee
4 {
5     // Свойство для получения имени (только для чтения)
6     public string FirstName { get; private set; }
7
8     // Свойство для получения фамилии (только для чтения)
9     public string LastName { get; private set; }
10

```

Ил. 12.4. Абстрактный базовый класс Employee (продолжение ↗)



```
11 // Свойство для получения номера соцстрахования (только для чтения)
12 public string SocialSecurityNumber { get; private set; }
13
14 // Конструктор с тремя параметрами
15 public Employee( string first, string last, string ssn )
16 {
17     FirstName = first;
18     LastName = last;
19     SocialSecurityNumber = ssn;
20 } // Конец конструктора Employee с тремя параметрами
21
22 // Получение строкового представления объекта Employee
23 public override string ToString()
24 {
25     return string.Format( "{0} {1}\nsocial security number: {2}",
26         FirstName, LastName, SocialSecurityNumber );
27 } // Конец метода ToString
28
29 // Абстрактный метод переопределяется в производных классах
30 public abstract decimal Earnings(); // Без реализации
31 } // Конец абстрактного класса Employee
```

#### Ил. 12.4. Абстрактный базовый класс Employee (окончание)

Почему мы объявили метод `Earnings` как абстрактный? Как объяснялось ранее, предоставление реализации этого метода в классе `Employee` просто не имеет смысла. Мы не можем вычислить заработок «работника вообще» — сначала необходимо определить конкретный тип работника. Объявляя этот метод абстрактным, мы показываем, что каждый конкретный производный класс должен предоставить подходящую реализацию `Earnings`, а приложение сможет использовать переменные базового класса `Employee` для полиморфного вызова `Earnings` для любого типа `Employee`.

### 12.5.2. Создание конкретного производного класса `SalariedEmployee`

Класс `SalariedEmployee` (ил. 12.5) расширяет класс `Employee` (строка 5) и переопределяет метод `Earnings` (строки 34–37), вследствие чего `SalariedEmployee` становится конкретным классом. Класс содержит конструктор (строки 10–14), в аргументах которого передается имя, фамилия, номер социального страхования и зарплата; свойство `WeeklySalary` (строки 17–31) для работы с переменной экземпляра `weeklySalary` (с `set`-методом, проверяющим неотрицательность присваиваемого значения); метод `Earnings` (строки 34–37) для вычисления заработка `SalariedEmployee`; и метод `ToString` (строки 40–44), возвращающий строку с типом работника (а именно "salaried employee:"), за которым следует информация, возвращенная методом `ToString` базового класса `Employee`, и свойство `WeeklySalary` класса `SalariedEmployee`. Конструктор класса `SalariedEmployee` передает имя, фамилию и номер социального страхования конструктору `Employee` (строка 11) для инициализации данных базового

класса. Метод `Earnings` переопределяет абстрактный метод `Employee` и предоставляет конкретную реализацию, которая возвращает заработок `SalariedEmployee`. Если бы метод `Earnings` не был реализован, то класс `SalariedEmployee` пришлось бы объявить абстрактным — в противном случае произойдет ошибка компиляции (а мы, естественно, хотим, чтобы класс `SalariedEmployee` был конкретным).

```
1  // Ил. 12.5: SalariedEmployee.cs
2  // Класс SalariedEmployee расширяет Employee.
3  using System;
4
5  public class SalariedEmployee : Employee
6  {
7      private decimal weeklySalary;
8
9      // Конструктор с четырьмя параметрами
10     public SalariedEmployee( string first, string last, string ssn,
11         decimal salary ) : base( first, last, ssn )
12     {
13         WeeklySalary = salary; // Проверка зарплаты через свойство
14     } // Конец конструктора SalariedEmployee с четырьмя параметрами
15
16     // Свойство для чтения и записи зарплаты работника
17     public decimal WeeklySalary
18     {
19         get
20         {
21             return weeklySalary;
22         } // Конец get
23         set
24         {
25             if ( value >= 0 ) // Проверка
26                 weeklySalary = value;
27             else
28                 throw new ArgumentOutOfRangeException( "WeeklySalary",
29                     value, "WeeklySalary must be >= 0" );
30         } // Конец set
31     } // Конец свойства WeeklySalary
32
33     // Вычисление заработка; переопределение абстрактного метода Earnings
34     public override decimal Earnings() // в классе Employee
35     {
36         return WeeklySalary;
37     } // Конец метода Earnings
38
39     // Получение строкового представления объекта SalariedEmployee
40     public override string ToString()
41     {
42         return string.Format( "salaried employee: {0}\n{1}: {2:C}",
43             base.ToString(), "weekly salary", WeeklySalary );
44     } // Конец метода ToString
45 } // Конец класса SalariedEmployee
```

**Ил. 12.5.** Класс `SalariedEmployee` расширяет `Employee`

Метод `ToString` класса `SalariedEmployee` (строки 40–44) переопределяет версию класса `Employee`. Если бы класс `SalariedEmployee` не переопределял `ToString`, то он унаследовал бы версию `Employee`. В этом случае вызов метода `ToString` для объекта `SalariedEmployee` вернул бы полное имя и номер социального страхования; этих данных недостаточно для представления `SalariedEmployee`. Метод `ToString` выдает полное строковое представление `SalariedEmployee` из строки `"salariedemployee:"`, за которой следует информация `Employee` базового класса (имя, фамилия, номер социального страхования), полученная вызовом метода `ToString` базового класса (строка 43), — это хороший пример повторного использования кода. Строковое представление `SalariedEmployee` также содержит еженедельную зарплату работника, полученную при помощи свойства `WeeklySalary`.

### 12.5.3. Создание конкретного производного класса `HourlyEmployee`

Класс `HourlyEmployee` (ил. 12.6) также расширяет класс `Employee` (строка 5). Он включает конструктор (строки 11–17), в аргументах которого передаются имя, фамилия, номер социального страхования, почасовая ставка и количество отработанных часов. В строках 20–34 и 37–51 объявляются свойства `Wage` и `Hours` для переменных экземпляров `wage` и `hours` соответственно. `Set`-метод свойства `Wage` проверяет, что значение `wage` неотрицательно, а `set`-метод свойства `Hours` убеждается в том, что значение `hours` лежит в диапазоне 0–168 (общее количество часов в неделе) включительно. Класс переопределяет метод `Earnings` (строки 54–60) для вычисления заработка объекта `HourlyEmployee` и метод `ToString` (строки 63–68) для получения строкового представления работника. Конструктор `HourlyEmployee`, как и конструктор `SalariedEmployee`, передает имя, фамилию и номер социального страхования конструктору базового класса `Employee` (строка 13) для инициализации данных базового класса. Кроме того, метод `ToString` вызывает метод `ToString` базового класса (строка 67) для получения информации, относящейся к `Employee` (имя, фамилия, номер социального страхования).

```
1 // Ил. 12.6: HourlyEmployee.cs
2 // Класс HourlyEmployee расширяет Employee.
3 using System;
4
5 public class HourlyEmployee : Employee
6 {
7     private decimal wage; // Почасовая ставка
8     private decimal hours; // Количество отработанных часов
9
10    // Конструктор с пятью параметрами
11    public HourlyEmployee( string first, string last, string ssn,
12        decimal hourlyWage, decimal hoursWorked )
13        : base( first, last, ssn )
```

**Ил. 12.6.** Класс `HourlyEmployee` расширяет `Employee` (продолжение ↗)

```

14  {
15      Wage = hourlyWage; // Проверка почасовой ставки (через свойство)
16      Hours = hoursWorked; // Проверка отработанных часов (через свойство)
17  } // Конец конструктора HourlyEmployee с пятью параметрами
18
19  // Свойство для чтения и записи почасовой ставки
20  public decimal Wage
21  {
22      get
23      {
24          return wage;
25      } // Конец get
26      set
27      {
28          if ( value >= 0 ) // Проверка
29              wage = value;
30          else
31              throw new ArgumentOutOfRangeException( "Wage",
32                  value, "Wage must be >= 0" );
33      } // Конец set
34  } // Конец свойства Wage
35
36  // Свойство для чтения и записи отработанных часов
37  public decimal Hours
38  {
39      get
40      {
41          return hours;
42      } // Конец get
43      set
44      {
45          if ( value >= 0 && value <= 168 ) // Проверка
46              hours = value;
47          else
48              throw new ArgumentOutOfRangeException( "Hours",
49                  value, "Hours must be >= 0 and <= 168" );
50      } // Конец set
51  } // Конец свойства Hours
52
53  // Вычисление заработка; переопределение абстрактного метода
54  public override decimal Earnings() // Earnings класса Employee
55  {
56      if ( Hours <= 40 ) // Без переработки
57          return Wage * Hours;
58      else
59          return ( 40 * Wage ) + ( ( Hours - 40 ) * Wage * 1.5M );
60  } // Конец метода Earnings
61
62  //Получение строкового представления объекта HourlyEmployee
63  public override string ToString()
64  {
65      return string.Format(
66          "hourly employee: {0}\n{1}: {2:C}; {3}: {4:F2}",
67          base.ToString(), "hourly wage", Wage, "hours worked", Hours );
68  } // Конец метода ToString
69 } // Конец класса HourlyEmployee

```

**Ил. 12.6.** Класс HourlyEmployee расширяет Employee (окончание)

### 12.5.4. Создание конкретного производного класса `CommissionEmployee`

Класс `CommissionEmployee` (ил. 12.7) расширяет класс `Employee` (строка 5). Класс включает конструктор (строки 11–16), который получает имя, фамилию, номер социального страхования, объем продаж и комиссионную ставку; свойства (строки 19–33 и 36–50) для переменных `grossSales` и `commissionRate` соответственно; метод `Earnings` (строки 53–56) для вычисления заработка `CommissionEmployee` и метод `ToString` (строки 59–64), возвращающий строковое представление работника. Конструктор `CommissionEmployee` также передает имя, фамилию и номер социального страхования конструктору базового класса `Employee` (строка 12) для инициализации данных `Employee`. Метод `ToString` вызывает метод `ToString` базового класса (строка 62) для получения информации, относящейся к `Employee` (имя, фамилия, номер социального страхования).

```
1 // Ил. 12.7: CommissionEmployee.cs
2 // Класс CommissionEmployee расширяет Employee.
3 using System;
4
5 public class CommissionEmployee : Employee
6 {
7     private decimal grossSales;    // Еже недельные продажи
8     private decimal commissionRate; // Комиссионная ставка
9
10    // Конструктор с пятью параметрами
11    public CommissionEmployee( string first, string last, string ssn,
12        decimal sales, decimal rate ) : base( first, last, ssn )
13    {
14        GrossSales = sales;    // Проверка объема продаж (через свойство)
15        CommissionRate = rate; // Проверка комиссионной ставки (через свойство)
16    } // Конец конструктора CommissionEmployee
17
18    // Свойство для чтения и записи объема продаж
19    public decimal GrossSales
20    {
21        get
22        {
23            return grossSales;
24        } // Конец get
25        set
26        {
27            if ( value >= 0 )
28                grossSales = value;
29            else
30                throw new ArgumentOutOfRangeException(
31                    "GrossSales", value, "GrossSales must be >= 0" );
32        } // Конец set
33    } // Конец свойства GrossSales
34
35    // Свойство для чтения и записи комиссионной ставки
36    public decimal CommissionRate
```

**Ил. 12.7.** Класс `CommissionEmployee` расширяет `Employee` (продолжение ↗)

```
37 {
38     get
39     {
40         return commissionRate;
41     } // Конец get
42     set
43     {
44         if ( value > 0 && value < 1 )
45             commissionRate = value;
46         else
47             throw new ArgumentOutOfRangeException( "CommissionRate",
48                 value, "CommissionRate must be > 0 and < 1" );
49     } // Конец set
50 } // Конец свойства CommissionRate
51
52 // Вычисление заработка; переопределение метода Earnings класса Employee
53 public override decimal Earnings()
54 {
55     return CommissionRate * GrossSales;
56 } // Конец метода Earnings
57
58 // Получение строкового представления объекта CommissionEmployee
59 public override string ToString()
60 {
61     return string.Format( "{0}: {1}\n{2}: {3:C}\n{4}: {5:F2}",
62         "commission employee", base.ToString(),
63         "gross sales", GrossSales, "commission rate", CommissionRate );
64 } // Конец метода ToString
65 } // Конец класса CommissionEmployee
```

**Ил. 12.7.** Класс `CommissionEmployee` расширяет `Employee` (окончание)

### 12.5.5. Создание конкретного класса `BasePlusCommissionEmployee`

Класс `BasePlusCommissionEmployee` (ил. 12.8) расширяет класс `CommissionEmployee` (строка 5), а следовательно, является косвенно производным от класса `Employee`. Класс `BasePlusCommissionEmployee` содержит конструктор (строки 10–15), который получает имя, фамилию, номер социального страхования, объем продаж, комиссионную ставку и базовую зарплату. Затем имя, фамилия, номер социального страхования, объем продаж и комиссионная ставка передаются конструктору `CommissionEmployee` (строка 12) для инициализации данных базового класса. `BasePlusCommissionEmployee` также содержит свойство `BaseSalary` (строки 19–33) для работы с переменной экземпляра `baseSalary`. Метод `Earnings` (строки 36–39) вычисляет заработок для объекта `BasePlusCommissionEmployee`. Строка 38 метода `Earnings` вызывает метод `Earnings` базового класса `CommissionEmployee` для вычисления комиссионной части заработка работника.

И снова налицо преимущества повторного использования кода. Метод `ToString` класса `BasePlusCommissionEmployee` (строки 42–46) создает строковое представление

`BasePlusCommissionEmployee`, которое состоит из строки "base-salaried", за которой следует строка, полученная вызовом метода `ToString` базового класса `CommissionEmployee` (другой пример повторного использования кода и базовая зарплата). В результате получается строка, начинающаяся с префикса "base-salaried commissionemployee", за которым следует остальная информация `BasePlusCommissionEmployee`. Вспомните, что метод `ToString` класса `CommissionEmployee` получает имя, фамилию и номер социального страхования работника вызовом метода `ToString` своего базового класса (то есть `Employee`) — еще одно проявление повторного использования кода. Метод `ToString` класса `BasePlusCommissionEmployee` инициирует цепочку вызовов, проходящую через все три уровня иерархии `Employee`.

```
1  // Ил. 12.8: BasePlusCommissionEmployee.cs
2  // Класс BasePlusCommissionEmployee расширяет CommissionEmployee.
3  using System;
4
5  public class BasePlusCommissionEmployee : CommissionEmployee
6  {
7      private decimal baseSalary; // Базовая зарплата
8
9      // Конструктор с шестью параметрами
10     public BasePlusCommissionEmployee( string first, string last,
11         string ssn, decimal sales, decimal rate, decimal salary )
12         : base( first, last, ssn, sales, rate )
13     {
14         BaseSalary = salary; // Проверка базовой зарплаты через свойство
15     } // Конец конструктора BasePlusCommissionEmployee
16
17     // Свойство для чтения и записи базовой зарплаты
18     // работника, получающего зарплату и комиссионные
19     public decimal BaseSalary
20     {
21         get
22         {
23             return baseSalary;
24         } // Конец get
25         set
26         {
27             if ( value >= 0 )
28                 baseSalary = value;
29             else
30                 throw new ArgumentOutOfRangeException( "BaseSalary",
31                     value, "BaseSalary must be >= 0" );
32         } // Конец set
33     } // Конец свойства BaseSalary
34
35     // Вычисление заработка; переопределение метода Earnings
36     public override decimal Earnings() // класса CommissionEmployee
37     {
38         return BaseSalary + base.Earnings();
39     } // Конец метода Earnings
40
```

**Ил. 12.8.** Класс `BasePlusCommissionEmployee` расширяет `CommissionEmployee` (продолжение ↗)

```

41 // Получение строкового представления BasePlusCommissionEmployee
42 public override string ToString()
43 {
44     return string.Format( "base-salaried {0}; base salary: {1:C}",
45         base.ToString(), BaseSalary );
46 } // Конец метода ToString
47 } // Конец класса BasePlusCommissionEmployee

```

**Ил. 12.8.** Класс `BasePlusCommissionEmployee` расширяет `CommissionEmployee` (окончание)

### 12.5.6. Полиморфная обработка, оператор `is` и понижающее преобразование

Для тестирования иерархии `Employee` приложение на ил. 12.9 создает объекты всех четырех конкретных классов: `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` и `BasePlusCommissionEmployee`. Приложение работает с этими объектами — сначала через переменные каждого из фактических типов объектов, а затем полиморфно, с использованием массива переменных `Employee`. В ходе полиморфной обработки объектов приложение увеличивает базовую зарплату каждого объекта `BasePlusCommissionEmployee` на 10 % (конечно, для этого необходимо проверить фактический тип объекта во время выполнения). Наконец, приложение полиморфно определяет и выводит тип каждого объекта из массива `Employee`. В строках 10–20 создаются объекты каждого из четырех конкретных классов, производных от `Employee`. В строках 24–32 выводятся строковые представления и заработки для всех этих объектов. Метод `ToString` каждого объекта неявно вызывается методом `WriteLine` при выводе объекта со строковым форматным элементом.

#### Присваивание объектов производных классов ссылкам на базовый класс

В строке 35 объявляется переменная `employees`, которой присваивается массив из четырех переменных `Employee`. В строках 38–41 элементам `employees[0]`, `employees[1]`, `employees[2]` и `employees[3]` присваиваются объект `SalariedEmployee`, объект `HourlyEmployee`, объект `CommissionEmployee` и объект `BasePlusCommissionEmployee` соответственно. Все присваивания допустимы, потому что и `SalariedEmployee`, и `HourlyEmployee`, и `CommissionEmployee`, и `BasePlusCommissionEmployee` являются частным случаем `Employee`. Таким образом, мы можем присвоить ссылки на объекты `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` переменным базового класса `Employee`, несмотря на то что класс `Employee` является *абстрактным*.

```

1 // Ил. 12.9: PayrollSystemTest.cs
2 // Тестовое приложение для иерархии Employee.
3 using System;
4
5 public class PayrollSystemTest
6 {
7     public static void Main( string[] args )

```


**Ил. 12.9.** Тестовое приложение для иерархии `Employee` (продолжение ↗)



```

8      {
9          // create derived-class objects
10         SalariedEmployee salariedEmployee =
11             new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00M );
12         HourlyEmployee hourlyEmployee =
13             new HourlyEmployee( "Karen", "Price",
14                 "222-22-2222", 16.75M, 40.0M );
15         CommissionEmployee commissionEmployee =
16             new CommissionEmployee( "Sue", "Jones",
17                 "333-33-3333", 10000.00M, .06M );
18         BasePlusCommissionEmployee basePlusCommissionEmployee =
19             new BasePlusCommissionEmployee( "Bob", "Lewis",
20                 "444-44-4444", 5000.00M, .04M, 300.00M );
21
22         Console.WriteLine( "Employees processed individually:\n" );
23
24         Console.WriteLine( "{0}\nearned: {1:C}\n",
25             salariedEmployee, salariedEmployee.Earnings() );
26         Console.WriteLine( "{0}\nearned: {1:C}\n",
27             hourlyEmployee, hourlyEmployee.Earnings() );
28         Console.WriteLine( "{0}\nearned: {1:C}\n",
29             commissionEmployee, commissionEmployee.Earnings() );
30         Console.WriteLine( "{0}\nearned: {1:C}\n",
31             basePlusCommissionEmployee,
32             basePlusCommissionEmployee.Earnings() );
33
34         // Создание массива с четырьмя элементами Employee
35         Employee[] employees = new Employee[ 4 ];
36
37         // Инициализация массива элементами производных типов
38         employees[ 0 ] = salariedEmployee;
39         employees[ 1 ] = hourlyEmployee;
40         employees[ 2 ] = commissionEmployee;
41         employees[ 3 ] = basePlusCommissionEmployee;
42
43         Console.WriteLine( "Employees processed polymorphically:\n" );
44
45         // Полиморфная обработка всех элементов массива employees
46         foreach ( Employee currentEmployee in employees )
47         {
48             Console.WriteLine( currentEmployee ); // Вызывает ToString
49
50             // Относится ли элемент к типу BasePlusCommissionEmployee?
51             if ( currentEmployee is BasePlusCommissionEmployee )
52             {
53                 // Выполнить понижающее преобразование ссылки на Employee
54                 // в ссылку на BasePlusCommissionEmployee
55                 BasePlusCommissionEmployee employee =
56                     ( BasePlusCommissionEmployee ) currentEmployee;
57
58                 employee.BaseSalary *= 1.10M;
59                 Console.WriteLine(
60                     "new base salary with 10% increase is: {0:C}",
61                     employee.BaseSalary );

```

**Ил. 12.9.** Тестовое приложение для иерархии Employee (продолжение )

```
62         } // Конец if
63
64         Console.WriteLine(
65             "earned {0:C}\n", currentEmployee.Earnings() );
66     } // Конец foreach
67
68     // Получение имени типа каждого объекта в массиве employees
69     for ( int j = 0; j < employees.Length; j++ )
70         Console.WriteLine( "Employee {0} is a {1}", j,
71             employees[ j ].GetType() );
72     } // Конец Main
73 } // Конец класса PayrollSystemTest
```

Employees processed individually:

salaried employee: John Smith  
social security number: 111-11-1111  
weekly salary: \$800.00  
earned: \$800.00

hourly employee: Karen Price  
social security number: 222-22-2222  
hourly wage: \$16.75; hours worked: 40.00  
earned: \$670.00

commission employee: Sue Jones  
social security number: 333-33-3333  
gross sales: \$10,000.00  
commission rate: 0.06  
earned: \$600.00

base-salaried commission employee: Bob Lewis  
social security number: 444-44-4444  
gross sales: \$5,000.00  
commission rate: 0.04; base salary: \$300.00  
earned: \$500.00

Employees processed polymorphically:

salaried employee: John Smith  
social security number: 111-11-1111  
weekly salary: \$800.00  
earned \$800.00

hourly employee: Karen Price  
social security number: 222-22-2222  
hourly wage: \$16.75; hours worked: 40.00  
earned \$670.00

commission employee: Sue Jones  
social security number: 333-33-3333  
gross sales: \$10,000.00  
commission rate: 0.06

**Ил. 12.9.** Тестовое приложение для иерархии Employee (продолжение ↗)

```
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00
commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

**Ил. 12.9.** Тестовое приложение для иерархии Employee (окончание)

### Полиморфная обработка массива

В строках 46–66 приложение перебирает элементы массива `employees` и вызывает методы `ToString` и `Earnings` с переменной `currentEmployee` типа `Employee`, которой при каждой итерации присваивается ссылка на очередной объект `Employee`. Из выходных данных видно, что при этом действительно вызываются правильные методы всех классов. Все вызовы виртуальных методов `ToString` и `Earnings` разрешаются во время выполнения в зависимости от типа объекта, на который указывает текущее значение `currentEmployee`. Этот процесс называется *динамической*, или *поздней, привязкой*. Например, строка 48 неявно вызывает метод `ToString` объекта, на который указывает `currentEmployee`. Через переменную `Employee` могут вызываться только методы класса `Employee`, а `Employee` включает методы класса `object`, такие как `ToString` (методы, наследуемые всеми классами от класса `object`, рассматриваются в разделе 11.7). Ссылка на базовый класс может использоваться для вызова методов только базового класса.

### 10%-ная надбавка для `BasePlusCommissionEmployee`

Для объектов `BasePlusCommissionEmployee` выполняется специальная обработка — если текущий элемент массива относится к этому типу, его базовую зарплату следует увеличить на 10 %. При полиморфной обработке объектов специфика отдельных типов часто игнорируется, но для регулировки базовой зарплаты нам необходимо определить конкретный тип каждого объекта `Employee` во время выполнения. Условие в строке 51 истинно, если объект, на который указывает `currentEmployee`, имеет фактический тип `BasePlusCommissionEmployee`. Оно также будет истинно, если объект относится к классу, производному от `BasePlusCommissionEmployee` (если они есть), потому что производный класс является частным случаем своего базового класса. В строках 55–56 выполняется понижающее преобразование `currentEmployee` к типу `BasePlusCommissionEmployee` — это преобразование разрешено только в том случае, если объект связан с `BasePlusCommissionEmployee` отношением «является частным случаем». Условие в строке 51 проверяет выполнение этого требования. Преобразование необходимо, если мы хотим использовать свойство `BaseSalary` производного класса `BasePlusCommissionEmployee` с текущим объектом `Employee`, — попытка прямого

вызова в базовом классе метода, определенного только в производном классе, приводит к ошибке компиляции.



### ТИПИЧНАЯ ОШИБКА 12.3

Попытка присвоить переменную базового класса переменной производного класса (без явного понижающего преобразования) является ошибкой компиляции.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 12.3

Если во время выполнения ссылка на объект производного класса присваивается переменной его прямого или одного из косвенных базовых классов, то ссылку, хранящуюся в этой переменной базового класса, можно преобразовать в ссылку на производный класс. Перед выполнением такого преобразования используйте оператор `is` и убедитесь в том, что объект действительно является объектом соответствующего производного класса.

Если при понижающем преобразовании объекта объект не связан с типом, указанным в операторе преобразования, отношением «является частным случаем», происходит исключение `InvalidCastException` (пространство имен `System`). Объект может быть преобразован только к собственному типу или к типу одного из своих базовых классов. Чтобы избежать возможной выдачи исключения `InvalidCastException`, можно использовать для преобразования оператор `as` вместо оператора преобразования. Например, в команде

```
BasePlusCommissionEmployee employee =  
    currentEmployee as BasePlusCommissionEmployee;
```

переменной `employee` присваивается ссылка на объект или значение `null`, если `currentEmployee` не является частным случаем `BasePlusCommissionEmployee`. Чтобы узнать, успешно ли прошло преобразование, можно проверить `employee` на равенство `null`.

Если выражение в строке 51 истинно, то команда `if` (строки 51–62) выполняет специальные действия только для объекта `BasePlusCommissionEmployee`. Используя переменную `employee` типа `BasePlusCommissionEmployee`, строка 58 обращается к свойству `BaseSalary`, определенному только в производном классе, для чтения и обновления базовой зарплаты работника с 10%-ной надбавкой.

В строках 64–65 для `currentEmployee` вызывается метод `Earnings`, что приводит к полиморфному вызову метода `Earnings` соответствующего производного класса. Полиморфное вычисление заработка `SalariedEmployee`, `HourlyEmployee` и `CommissionEmployee` в строках 64–65 дает почти такой же результат, что и получение заработка в строках 24–29 (заработок `BasePlusCommissionEmployee` в строках 64–65 выше из-за 10%-ного увеличения базовой зарплаты).

### Каждый объект знает свой тип

В строках 69–71 тип каждого объекта работника выводится в формате `string`. Каждый объект знает свой тип и может обращаться к этой информации при помощи метода `GetType`, наследуемого всеми классами от класса `object`. Метод `GetType`

возвращает объект класса `Type` (пространство имен `System`), содержащий информацию о типе объекта, включая имя класса, имена методов и имя базового класса. В строке 71 вызывается метод `GetType` объекта для получения его фактического класса (то есть объекта `Type`, описывающего тип объекта). Затем для объекта, возвращенного `GetType`, неявно вызывается метод `ToString`. Для класса `Type` метод `ToString` возвращает имя класса.

### Ошибки компиляции при понижающем преобразовании

В рассмотренном примере для предотвращения ошибок компиляции переменная `Employee` преобразуется в переменную `BasePlusCommissionEmployee` в строках 55–56. Если убрать оператор преобразования (`BasePlusCommissionEmployee`) из строки 56 и попытаться напрямую присвоить переменную `currentEmployee` типа `Employee` переменной `employee` типа `BasePlusCommissionEmployee`, компилятор выдает ошибку с сообщением о невозможности неявного преобразования типа. Компилятор запрещает такое присваивание, потому что `CommissionEmployee` не является частным случаем `BasePlusCommissionEmployee`, — еще раз напомним, что отношение «является частным случаем» направлено только от производного класса к его базовым классам, но не наоборот.

Аналогичным образом, если бы в строках 58 и 61 при использовании свойства `BaseSalary` (определенного только для производного класса) использовалась переменная базового класса `currentEmployee` вместо переменной производного класса `employee`, компилятор бы выдал ошибку «`'Employee'` не содержит определения `'BaseSalary'`». Попытки вызова по ссылке на базовый класс методов, существующих только в производном классе, запрещены. Хотя строки 58 и 61 выполняются только в том случае, если `is` в строке 51 возвращает `true` (то есть `currentEmployee` была присвоена ссылка на объект `BasePlusCommissionEmployee`), мы не можем использовать свойство `BaseSalary` производного класса `BasePlusCommissionEmployee` с `currentEmployee` — ссылкой на базовый класс `Employee`. Компилятор выдаст ошибки в строках 58 и 61, потому что `BaseSalary` не является членом базового класса и не может использоваться с переменной базового класса. И хотя фактически вызываемый метод определяется в зависимости от типа объекта во время выполнения, переменная может использоваться для вызова только тех методов, которые входят в тип этой переменной; это условие проверяется компилятором. Через переменную базового класса `Employee` можно вызывать только методы и свойства, принадлежащие классу `Employee` (методы `Earnings` и `ToString`, свойства `FirstName`, `LastName` и `SocialSecurityNumber`), а также методы, унаследованные от класса `object`.

## 12.5.7. Сводка разрешенных присваиваний между переменными базового и производного класса

Теперь, когда мы рассмотрели приложение с полиморфной обработкой объектов разных производных классов, подведем итог того, что можно и чего нельзя делать с объектами и переменными базового класса и производных классов. Хотя объект

производного класса также является частным случаем объекта базового класса, это не одно и то же. Как упоминалось ранее, объекты производного класса могут интерпретироваться как объекты базового класса, но при этом производный класс может содержать дополнительные члены, отсутствующие в базовом классе. По этой причине присваивание ссылки на базовый класс переменной производного класса недопустимо без явного преобразования типа — при таком присваивании члены производного класса останутся неопределенными для объекта базового класса.

Мы рассмотрели четыре варианта присваивания ссылок на базовый и производный классы переменным базового и производных классов:

1. Присваивание ссылки на базовый класс переменной базового класса — тривиально.
2. Присваивание ссылки на производный класс переменной производного класса — тривиально.
3. Присваивание ссылки на производный класс переменной базового класса безопасно, потому что объект производного класса является частным случаем объекта базового класса. Однако эта ссылка может использоваться только для обращения к членам базового класса. Если программа попытается обратиться к членам, присутствующим только в производном классе, через переменную базового класса, компилятор выдаст сообщение об ошибке.
4. Попытка присваивания ссылки на базовый класс переменной производного класса приводит к ошибке компиляции. Чтобы избежать этой ошибки, необходимо преобразовать ссылку на базовый класс к типу производного класса или выполнить преобразование с использованием оператора `as`. Во время выполнения, если объект, на который указывает ссылка, не является объектом производного класса, произойдет исключение (если не использовать оператор `as`). Также можно воспользоваться оператором `is` и проследить за тем, чтобы преобразование выполнялось только для объекта производного класса.

## 12.6. Запечатанные методы и классы

В производных классах могут переопределяться только методы, объявленные с ключевыми словами `virtual`, `override` и `abstract`. Метод, объявленный в базовом классе *запечатанным* (`sealed`), не может переопределяться в производном классе. Методы, объявленные закрытыми, неявно являются запечатанными, потому что их невозможно переопределить в производном классе (хотя производный класс может объявить новый метод с такой же сигнатурой, как у закрытого метода базового класса). Методы, объявленные статическими, также неявно являются запечатанными, потому что статические методы тоже не могут переопределяться. Метод производного класса, объявленный с ключевыми словами `override` и `sealed`, может переопределить метод базового класса, но не может переопределяться в производных классах ниже в иерархии наследования.

Объявление запечатанного метода изменяться не может, поэтому все производные классы используют одну реализацию метода, а вызовы запечатанных методов (и неvirtуальных методов) разрешаются во время компиляции (так называемая статическая привязка). Поскольку компилятор знает, что запечатанные методы не могут переопределяться, часто применяется оптимизация с удалением вызовов запечатанных методов и заменой их кодом объявления в точке вызова (так называемая *подстановка кода*).



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 12.1

Компилятор может решить выполнить подстановку вызова для простого, компактного запечатанного метода. Подстановка не нарушает инкапсуляции и сокрытия информации, но повышает быстродействие за счет отказа от лишних затрат ресурсов на вызов метода.

Класс, объявленный запечатанным, не может быть базовым (то есть класс не может расширять класс с ключевым словом `sealed`). Все методы запечатанного класса неявно являются запечатанными. Например, класс `string` является запечатанным. Он не может расширяться, так что приложения, использующие объекты `string`, ограничиваются функциональностью объектов `string`, определенной в Framework Class Library.



### ТИПИЧНАЯ ОШИБКА 12.4

Попытка объявить класс производным от запечатанного класса является ошибкой компиляции.

## 12.7. Пример: создание и использование интерфейсов

В следующем примере (ил. 12.11–12.15) мы снова вернемся к системе начисления заработка из раздела 12.5. Предположим, в одном приложении должны выполняться разные бухгалтерские операции — кроме вычисления сумм, выплачиваемых каждому работнику, компания также должна вычислять платежи по счетам за приобретенные товары. Хотя эти операции применяются к разным сущностям (работники и счета), в обоих случаях речь идет о вычислении некоторой выплачиваемой суммы. Для работника вычисляется сумма заработка, а для счета — общая стоимость товаров, включенных в счет. Можно ли организовать полиморфную обработку таких разных платежей в одном приложении? Существует ли механизм, требующий, чтобы не связанные между собой классы реализовали набор общих методов (например, метод для вычисления суммы платежа)? Именно для таких целей и существуют интерфейсы.

### Интерфейсы в программировании

Интерфейс C# описывает набор методов и свойств, которые могут вызываться для объекта, — например, чтобы приказать ему выполнить некую операцию или

получить некие данные. В следующем примере будет представлен интерфейс с именем `IPayable`, описывающий функциональность любого объекта, который поддерживает начисление платежа, а следовательно, должен предоставлять метод для определения выплачиваемой суммы. Объявление интерфейса начинается с ключевого слова `interface` и содержит только абстрактные методы, абстрактные свойства, абстрактные индексы (в этой книге не рассматриваемые) и абстрактные события (см. главу 14). Все методы интерфейсов неявно объявляются открытыми и абстрактными. Кроме того, каждый интерфейс может расширять один или несколько интерфейсов для построения более сложных интерфейсов, которые могут быть реализованы другими классами.



### ТИПИЧНАЯ ОШИБКА 12.5

Явное объявление членов интерфейса с ключевыми словами `public` или `abstract` считается ошибкой компиляции, потому что в объявлениях членов интерфейсов эти ключевые слова излишни. Также ошибкой компиляции считается включение в интерфейс каких-либо подробностей реализации (например, объявлений конкретных методов).

## Реализация интерфейса

Чтобы использовать интерфейс, класс должен указать, что он реализует этот интерфейс; для этого имя интерфейса указывается после двоеточия (`:`) в объявлении класса. Аналогичный синтаксис используется для обозначения наследования от базового класса. Конкретный класс, реализующий интерфейс, должен объявить каждый член интерфейса с сигнатурой, указанной в объявлении интерфейса. Если класс реализует интерфейс, но не реализует все его члены, он является *абстрактным* классом — он должен быть объявлен с ключевым словом `abstract` и содержать объявление `abstract` для каждого нереализованного члена интерфейса. Реализация интерфейса напоминает заключение контракта с компилятором, в котором говорится: «Обязуюсь предоставить реализацию всех членов, указанных в интерфейсе, или же объявить их абстрактными».



### ТИПИЧНАЯ ОШИБКА 12.6

Если класс, реализующий интерфейс, не определяет или не объявляет какие-либо члены интерфейса, происходит ошибка компиляции.

## Общие методы несвязанных классов

Интерфейс обычно используется в тех случаях, когда классы, не связанные между собой, должны содержать общие методы. Это позволяет организовать полиморфную обработку объектов несвязанных классов — объекты классов, реализующих один интерфейс, могут реагировать на одни и те же вызовы методов. Программист создает интерфейс, описывающий нужную функциональность, а затем реализует его во всех классах, которым нужна эта функциональность. Например, в приложении этого раздела мы реализуем интерфейс `IPayable` во всех классах, которые должны быть способны вычислять сумму платежа (например, `Employee` или `Invoice`).



## Интерфейсы и абстрактные классы

Интерфейс часто используется вместо абстрактного класса при отсутствии реализации по умолчанию, которая могла бы наследоваться (то есть при отсутствии полей и реализаций методов по умолчанию). Интерфейсы, как и абстрактные классы, обычно являются открытыми типами, поэтому обычно они объявляются в файлах с именем, совпадающим с именем интерфейса, и расширением `.cs`.

### 12.7.1. Разработка иерархии `IPayable`

Построение приложения, которое может вычислять платежи и для объектов-работников, и для объектов-счетов, начнется с создания интерфейса с именем `IPayable`. Интерфейс `IPayable` содержит метод `GetPaymentAmount`, который возвращает выплачиваемую сумму для объекта любого класса, реализующего интерфейс. Метод `GetPaymentAmount` представляет собой обобщенную версию метода `Earnings` иерархии `Employee` — метод `Earnings` вычисляет платеж конкретно для `Employee`, а `GetPaymentAmount` может применяться к широкому диапазону несвязанных объектов. После объявления интерфейса `IPayable` определяется класс `Invoice`, реализующий интерфейс `IPayable`. Затем мы изменяем класс `Employee` так, чтобы он реализовал интерфейс `IPayable`. Наконец, класс `SalariedEmployee`, производный от `Employee`, адаптируется для использования в иерархии `IPayable` (метод `Earnings` класса `SalariedEmployee` переименовывается в `GetPaymentAmount`).



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 12.1

По общепринятой схеме имена интерфейсов начинаются с буквы `I`. Такая схема помогает отличить интерфейсы от классов и упрощает чтение кода.



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 12.2

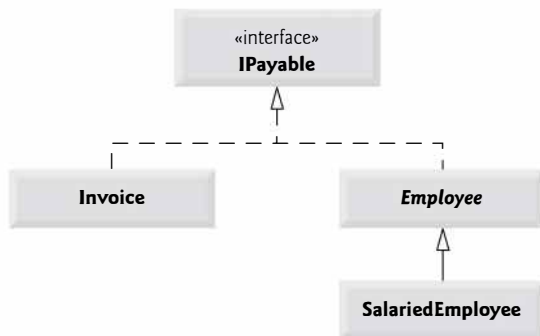
При объявлении метода в интерфейсе выберите имя, описывающее назначение метода на общем уровне, потому что метод может быть реализован во множестве классов, никак не связанных друг с другом.

Классы `Invoice` и `Employee` представляют сущности, для которых компания должна вычислить выплачиваемую сумму. Оба класса реализуют `IPayable`, так что приложение может вызывать метод `GetPaymentAmount` как для объектов `Invoice`, так и для объектов `Employee`. Это позволяет выполнять полиморфную обработку объектов `Invoice` и `Employee`, необходимую для бухгалтерского приложения из нашего примера.

## Интерфейс на диаграмме UML

На диаграмме классов UML с ил. 12.10 изображен интерфейс и иерархия классов нашего приложения. Иерархия начинается с интерфейса `IPayable`. На диаграммах UML над именами интерфейсов ставится слово `interface`, заключенное в кавычки-«елочки» (« и »). На диаграммах UML реализация интерфейса классом обозначается

пунктирной линией с незаполненной стрелкой, ведущей от реализующего класса к интерфейсу. Из диаграммы на ил. 12.10 видно, что каждый из классов, *Invoice* и *Employee*, реализует интерфейс *IPayable*. Как и на диаграмме на ил. 12.2, имя класса *Employee* записано курсивом — признак абстрактного класса. Конкретный класс *SalariedEmployee* расширяет *Employee* и наследует от своего базового класса отношение реализации интерфейса *IPayable*.



**Ил. 12.10.** Диаграмма классов UML с интерфейсом *IPayable* и иерархией классов

## 12.7.2. Объявление интерфейса *IPayable*

Объявление интерфейса *IPayable* начинается на ил. 12.11 в строке 3. Интерфейс *IPayable* содержит открытый абстрактный метод *GetPaymentAmount* (строка 5). Этот метод не может быть явно объявлен открытым или абстрактным. Интерфейсы могут содержать любое количество членов, а методы интерфейсов могут иметь параметры.

```

1 // Ил. 12.11: IPayable.cs
2 // Объявление интерфейса IPayable.
3 public interface IPayable
4 {
5     decimal GetPaymentAmount(); // Вычисление суммы; без реализации
6 } // Конец интерфейса IPayable
  
```

**Ил. 12.11.** Объявление интерфейса *IPayable*

## 12.7.3. Создание класса *Invoice*

Следующим шагом будет создание класса *Invoice* (ил. 12.12), представляющего простой счет с информацией по одному виду деталей. Класс содержит свойства *PartNumber* (строка 11), *PartDescription* (строка 14), *Quantity* (строки 27–41) и *PricePerItem* (строки 44–58), определяющие соответственно номер детали, описание, количество заказанных деталей и цену за единицу. Класс *Invoice* также содержит конструктор (строки 17–24) и метод *ToString* (строки 61–67), возвращающий

строковое представление объекта Invoice. Set-методы свойств Quantity и PricePerItem следят за тем, чтобы переменным quantity и pricePerItem присваивались только неотрицательные значения.

```
1 // Ил. 12.12: Invoice.cs
2 // Класс Invoice реализует IPayable.
3 using System;
4
5 public class Invoice : IPayable
6 {
7     private int quantity;
8     private decimal pricePerItem;
9
10    // Свойство для чтения и записи номера детали
11    public string PartNumber { get; set; }
12
13    // Свойство для чтения и записи описания детали
14    public string PartDescription { get; set; }
15
16    // Конструктор с четырьмя параметрами
17    public Invoice( string part, string description, int count,
18        decimal price )
19    {
20        PartNumber = part;
21        PartDescription = description;
22        Quantity = count;    // Проверка количества через свойство
23        PricePerItem = price; // Проверка цены за единицу через свойство
24    } // Конец конструктора Invoice с четырьмя параметрами
25
26    // Свойство для чтения и записи количества деталей
27    public int Quantity
28    {
29        get
30        {
31            return quantity;
32        } // Конец get
33        set
34        {
35            if ( value >= 0 ) // Проверка количества
36                quantity = value;
37            else
38                throw new ArgumentOutOfRangeException( "Quantity",
39                    value, "Quantity must be >= 0" );
40        } // Конец set
41    } // Конец свойства Quantity
42
43    // Свойство для чтения и записи цены за единицу
44    public decimal PricePerItem
45    {
46        get
47        {
48            return pricePerItem;
49        } // Конец get
50        set
```

**Ил. 12.12.** Класс Invoice реализует интерфейс IPayable (продолжение ↗)

```

51     {
52         if ( value >= 0 ) // Проверка цены
53             quantity = value;
54         else
55             throw new ArgumentOutOfRangeException( "PricePerItem",
56                 value, "PricePerItem must be >= 0" );
57     } // Конец set
58 } // Конец свойства PricePerItem
59
60 // Получение строкового представления объекта Invoice
61 public override string ToString()
62 {
63     return string.Format(
64         "{0}: \n{1}: {2} ({3}) \n{4}: {5} \n{6}: {7:C}",
65         "invoice", "part number", PartNumber, PartDescription,
66         "quantity", Quantity, "price per item", PricePerItem );
67 } // Конец метода ToString
68
69 // Метод, необходимый для выполнения контракта интерфейса IPayable
70 public decimal GetPaymentAmount()
71 {
72     return Quantity * PricePerItem; // Вычисление полной стоимости
73 } // Конец метода GetPaymentAmount
74 } // Конец класса Invoice

```

**Ил. 12.12.** Класс Invoice реализует интерфейс IPayable (окончание)

Строка 5 указывает, что класс Invoice реализует интерфейс IPayable. Как и все классы, Invoice также неявно наследует от класса object. C# не позволяет производным классам иметь более одного прямого базового класса, но класс может иметь базовый класс и реализовать любое количество интерфейсов. Все объекты класса, реализующего несколько интерфейсов, связаны отношением «является частным случаем» с каждым типом реализованного интерфейса. Чтобы реализовать несколько интерфейсов, разместите список имен интерфейсов, разделенных запятыми, после двоеточия (:) в объявлении класса:

```
public class ClassName : BaseClassName, FirstInterface, SecondInterface, ...
```

Если класс наследует от базового класса и реализует один или несколько интерфейсов, в его объявлении имя базового класса должно предшествовать именам интерфейсов.

Класс Invoice реализует один метод интерфейса IPayable — метод GetPaymentAmount объявляется в строках 70–73. Метод вычисляет сумму, необходимую для оплаты счета. Он перемножает значения quantity и pricePerItem (полученные при помощи соответствующих свойств) и возвращает результат (строка 72). Этот метод выполняет требование к реализации метода интерфейса IPayable — таким образом мы выполняем контракт с компилятором.

## 12.7.4. Изменение класса Employee для реализации интерфейса IPayable

Теперь мы внесем изменения в класс `Employee` для реализации интерфейса `IPayable`. Измененный класс `Employee` приведен на ил. 12.13. Объявление класса идентично объявлению на ил. 12.4 с двумя исключениями. Во-первых, строка 3 на ил. 12.13 указывает, что класс `Employee` теперь реализует интерфейс `IPayable`. По этой причине мы переименовали `Earnings` в `GetPaymentAmount` в иерархии `Employee`. Как и в случае с методом `Earnings` на ил. 12.4, реализация метода `GetPaymentAmount` в классе `Employee` не имеет смысла, потому что мы не можем вычислить сумму заработка, причитающуюся «работнику вообще», — сначала необходимо узнать конкретный тип работника. На ил. 12.4 метод `Earnings` по этой причине объявляется абстрактным, и в результате класс `Employee` тоже приходится объявлять абстрактным. Тем самым мы заставляем каждый класс, производный от `Employee`, переопределять `Earnings` конкретной реализацией. [*Примечание:* хотя мы переименовали `Earnings` в `GetPaymentAmount`, также можно было определить в классе `Employee` метод `GetPaymentAmount`, вызывающий `Earnings`. Тогда другие классы иерархии `Employee` изменять не придется.]

```
1 // Ил. 12.13: Employee.cs
2 // Абстрактный базовый класс Employee.
3 public abstract class Employee : IPayable
4 {
5     // Свойство для получения имени (только для чтения)
6     public string FirstName { get; private set; }
7
8     // Свойство для получения фамилии (только для чтения)
9     public string LastName { get; private set; }
10
11     // Свойство для получения номера соцстрахования (только для чтения)
12     public string SocialSecurityNumber { get; private set; }
13
14     // Конструктор с тремя параметрами
15     public Employee( string first, string last, string ssn )
16     {
17         FirstName = first;
18         LastName = last;
19         SocialSecurityNumber = ssn;
20     } // Конец конструктора Employee с тремя параметрами
21
22     // Получение строкового представления объекта Employee
23     public override string ToString()
24     {
25         return string.Format( "{0} {1}\nsocial security number: {2}",
26             FirstName, LastName, SocialSecurityNumber );
27     } // Конец метода ToString
28
29     // Примечание: метод GetPaymentAmount интерфейса IPayable
30     // не реализуется, чтобы этот класс можно было объявить абстрактным.
31     public abstract decimal GetPaymentAmount();
32 } // Конец абстрактного класса Employee
```

**Ил. 12.13.** Абстрактный базовый класс `Employee`

На ил. 12.13 происходит то же самое. Вспомните, что при реализации интерфейса класс заключает контракт с компилятором, согласно которому класс должен реализовать все методы интерфейса или объявить их абстрактными. Если выбирается второй вариант, класс тоже должен быть объявлен абстрактным. Как упоминалось в разделе 12.4, все конкретные производные классы абстрактного класса должны реализовать абстрактные методы базового класса. Если производный класс этого не делает, он тоже должен быть объявлен абстрактным. Как указано в комментарии в строках 29–30 ил. 12.13, класс `Employee` не реализует метод `GetPaymentAmount`, поэтому класс объявляется абстрактным.

### 12.7.5. Изменение класса `SalariedEmployee` для использования с `IPayable`

На ил. 12.14 представлена измененная версия класса `SalariedEmployee`, которая расширяет `Employee` и реализует метод `GetPaymentAmount`. Она идентична версии на ил. 12.5, не считая того, что новая версия реализует метод `GetPaymentAmount` (строки 35–38) вместо метода `Earnings`. Два метода содержат одинаковую функциональность, но имеют разные имена. Вспомните, что `IPayable`-версия метода имеет более общее имя, применимое в сильно различающихся классах. В остальных классах, производных от `Employee` (`HourlyEmployee`, `CommissionEmployee`, `BasePlusCommissionEmployee` и т. д.), тоже необходимо внести изменения; они должны содержать метод `GetPaymentAmount` вместо `Earnings` в соответствии с тем фактом, что `Employee` теперь реализует `IPayable`. Мы оставляем эти изменения читателю для самостоятельной работы, а в тестовом приложении этого раздела используется только класс `SalariedEmployee`.

```
1 // Ил. 12.14: SalariedEmployee.cs
2 // Класс SalariedEmployee расширяет Employee.
3 using System;
4
5 public class SalariedEmployee : Employee
6 {
7     private decimal weeklySalary;
8
9     // Конструктор с четырьмя параметрами
10    public SalariedEmployee( string first, string last, string ssn,
11        decimal salary ) : base( first, last, ssn )
12    {
13        WeeklySalary = salary; // Проверка зарплаты через свойство
14    } // Конец конструктора SalariedEmployee
15
16    // Свойство для чтения и записи зарплаты
17    public decimal WeeklySalary
18    {
19        get
```

**Ил. 12.14.** Класс `SalariedEmployee` расширяет `Employee` (продолжение ➤)

```

20     {
21         return weeklySalary;
22     } // Конец get
23     set
24     {
25         if ( value >= 0 ) // Проверка
26             weeklySalary = value;
27         else
28             throw new ArgumentOutOfRangeException( "WeeklySalary",
29                 value, "WeeklySalary must be >= 0" );
30     } // Конец set
31 } // Конец свойства WeeklySalary
32
33 // Вычисление заработка; реализация метода интерфейса IPayable,
34 // который был абстрактным в базовом классе Employee
35 public override decimal GetPaymentAmount()
36 {
37     return WeeklySalary;
38 } // Конец метода GetPaymentAmount
39
40 // Получение строкового представления объекта SalariedEmployee
41 public override string ToString()
42 {
43     return string.Format( "salaried employee: {0}\n{1}: {2:C}",
44         base.ToString(), "weekly salary", WeeklySalary );
45 } // Конец метода ToString
46 } // Конец класса SalariedEmployee

```

**Ил. 12.14.** Класс `SalariedEmployee` расширяет `Employee` (окончание)

При расширении интерфейса классом, как и при наследовании, действует отношение «является частным случаем». Класс `Employee` реализует `IPayable`, поэтому мы можем сказать, что `Employee` является частным случаем `IPayable` (как и любые классы, расширяющие `Employee`). Соответственно объекты `SalariedEmployee` могут интерпретироваться как объекты `IPayable`. Объект класса, реализующего интерфейс, может обрабатываться как объект интерфейсного типа (то есть объект, относящийся к типу интерфейса). Объекты любого класса, производного от класса, реализующего интерфейс, также могут рассматриваться как объекты интерфейсного типа. По аналогии с тем, как ссылка на `SalariedEmployee` может присваиваться переменной базового класса `Employee`, мы также можем присвоить ссылку на объект `SalariedEmployee` интерфейсной переменной `IPayable`. Класс `Invoice` реализует `IPayable`, поэтому объект `Invoice` также является объектом `IPayable`, и ссылка на объект `Invoice` может быть присвоена переменной `IPayable`.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 12.4

И наследование и интерфейсы реализуют отношение «является частным случаем». Объект класса, реализующего интерфейс, может интерпретироваться как объект интерфейсного типа. Объект любого класса, производного от класса, реализующего интерфейс, также может интерпретироваться как объект интерфейсного типа.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 12.5

Отношение «является частным случаем», существующее между базовыми и производными классами, а также интерфейсами и реализующими их классами, сохраняется при передаче объекта методу. Если в параметре метода передается аргумент базового класса или интерфейсного типа, то метод обеспечивает полиморфную обработку объекта, полученного в аргументе.

## 12.7.6. Использование интерфейса IPayable для полиморфной обработки объектов Invoice и Employee

Класс PayableInterfaceTest (ил. 12.15) показывает, что интерфейс IPayable может использоваться для полиморфной обработки набора объектов Invoice и Employee в одном приложении. Строка 10 объявляет переменную payableObjects и присваивает ей массив из четырех переменных IPayable. В строках 13–14 первым двум элементам payableObjects присваиваются ссылки на объекты Invoice, а в строках 15–18 двум оставшимся элементам payableObjects присваиваются ссылки на объекты SalariedEmployee. Эти присваивания возможны, потому что Invoice является частным случаем IPayable, SalariedEmployee — частным случаем Employee, а Employee — частным случаем IPayable. В строках 24–29 команда foreach используется для полиморфной обработки каждого объекта IPayable в payableObjects, с выводом каждого объекта в строковом виде и суммы платежа. В строках 27–28 метод ToString неявно вызывается по ссылке на интерфейс IPayable при том, что метод ToString не объявлен в интерфейсе IPayable, — все ссылки (включая ссылки на интерфейсные типы) относятся к объектам, расширяющим object, и следовательно, содержащим метод ToString. Строка 28 вызывает метод GetPaymentAmount класса IPayable для получения суммы платежа по каждому объекту из payableObjects независимо от его фактического типа. Из выходных данных видно, что вызовы методов в строках 27–28 вызывают реализации методов ToString и GetPaymentAmount соответствующих классов. Например, при первой итерации цикла foreach переменная currentPayable относится к объекту Invoice, поэтому выполняются методы ToString и GetPaymentAmount класса Invoice.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 12.6

По ссылке на интерфейсный тип могут вызываться все методы класса object — ссылка относится к объекту, а все объекты наследуют методы класса object.

```

1 // Ил. 12.15: PayableInterfaceTest.cs
2 // Тестирование интерфейса IPayable с разными классами.
3 using System;
4
5 public class PayableInterfaceTest
6 {
7     public static void Main( string[] args )
8     {
9         // Создание массива с четырьмя элементами IPayable
10         IPayable[] payableObjects = new IPayable[ 4 ];

```

**Ил. 12.15.** Тестирование интерфейса IPayable с разными классами (продолжение ➞)



```

11
12     // Заполнение массива объектами, реализующими IPayable
13     payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00M );
14     payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95M );
15     payableObjects[ 2 ] = new SalariedEmployee( "John", "Smith",
16         "111-11-1111", 800.00M );
17     payableObjects[ 3 ] = new SalariedEmployee( "Lisa", "Barnes",
18         "888-88-8888", 1200.00M );
19
20     Console.WriteLine(
21         "Invoices and Employees processed polymorphically:\n" );
22
23     // Полиморфная обработка элементов массива payableObjects
24     foreach ( var currentPayable in payableObjects )
25     {
26         // Вывод currentPayable и соответствующей суммы платежа
27         Console.WriteLine( "{0}\npayment due: {1:C}\n",
28             currentPayable, currentPayable.GetPaymentAmount() );
29     } // Конец foreach
30 } // Конец Main
31 } // Конец класса PayableInterfaceTest

```

Invoices and Employees processed polymorphically:

```

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

```

```

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

```

```

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

```

```

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00

```

**Ил. 12.15.** Тестирование интерфейса IPayable с разными классами (окончание)

## 12.7.7. Часто используемые интерфейсы .NET Framework Class Library

В этом разделе мы рассмотрим несколько часто используемых интерфейсов, определенных в .NET Framework Class Library. Эти интерфейсы реализуются и используются точно так же, как интерфейсы, созданные вами (например, интерфейс

Comparable из раздела 12.7.2). Интерфейсы Framework Class Library позволяют включать многие важные аспекты C# в ваши собственные классы. На ил. 12.16 приведена сводка часто используемых интерфейсов Framework Class Library.

Интерфейс	Описание
IComparable	Как упоминалось в главе 3, в C# существуют операторы сравнения (<, <=, >, >=, ==, !=) для сравнения значений простых типов. В разделе 12.8 вы увидите, что эти операторы можно определить и для сравнения двух объектов. Интерфейс IComparable также позволяет сравнивать два объекта класса, реализующего этот интерфейс. Интерфейс содержит один метод CompareTo, который сравнивает объект, вызывающий метод, с объектом, переданным в аргументе. Класс, реализующий CompareTo, возвращает признак того, что объект, для которого был вызван метод, меньше (отрицательное целочисленное значение), равен (0) или больше (положительное целочисленное значение) объекта, переданного в аргументе. Сравнение осуществляется по любым критериям на усмотрение разработчика. Например, если класс Employee реализует IComparable, его метод CompareTo может сравнивать объекты Employee по величине заработка. Интерфейс IComparable часто используется для упорядочения объектов в коллекциях (например, в массивах). Примеры использования IComparable приведены в главах 20 и 21
IComponent	Реализуется любым классом, представляющим компонент, включая элементы управления графического интерфейса (кнопки, надписи и т. д.). Интерфейс IComponent определяет поведение, которое должно реализовываться компонентами. IComponent и элементы графического интерфейса, реализующие интерфейс, рассматриваются в главах 14 и 15
IDisposable	Реализуется классами, предоставляющими явный механизм освобождения ресурсов. Некоторые ресурсы в любой момент времени могут использоваться только одной программой. Кроме того, некоторые ресурсы (такие, как файлы на диске) являются неуправляемыми и, в отличие от памяти, не могут освобождаться уборщиком мусора. Классы, реализующие интерфейс IDisposable, предоставляют метод Dispose, который должен явно вызываться для освобождения ресурсов. Интерфейс IDisposable кратко описан в главе 13. Об этом интерфейсе можно подробнее узнать по адресу <a href="http://msdn.microsoft.com/en-us/library/system.idisposable.aspx">msdn.microsoft.com/en-us/library/system.idisposable.aspx</a> . В статье MSDN «Implementing a Dispose Method» по адресу <a href="http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx">msdn.microsoft.com/en-us/library/fs2xkftw.aspx</a> рассказано, как правильно реализовать интерфейс в ваших классах
IEnumerator	Используется для последовательного перебора элементов коллекций (например, массивов). В интерфейс IEnumerator входит метод MoveNext для перехода к следующему элементу коллекции, метод Reset для возвращения к позиции перед первым элементом и свойство Current для получения объекта в текущей позиции. Интерфейс IEnumerator рассмотрен в главе 21

**Ил. 12.16.** Основные интерфейсы .NET Framework Class Library

## 12.8. Перегрузка операторов

Операции с объектами часто выполняются посредством отправки сообщений (в форме вызова методов). Однако запись с вызовами методов получается слишком

громоздкой для некоторых разновидностей классов — особенно математических. В таких классах для выполнения операций с объектами удобно использовать богатый набор встроенных операторов C#. В этом разделе мы покажем, как обеспечить работу этих операторов с объектами классов, — для этого используется механизм, называемый *перегрузкой операторов*. Перегрузка операторов заставляет их учитывать контекст, в котором они используются. Некоторые операторы — прежде всего арифметические операторы (такие, как + и -) — перегружаются чаще других.

На ил. 12.17 и 12.18 приведен пример использования перегрузки операторов для класса `ComplexNumber`. Полный список перегружаемых операторов находится по адресу [msdn.microsoft.com/en-us/library/8edha89s.aspx](http://msdn.microsoft.com/en-us/library/8edha89s.aspx).

### Класс `ComplexNumber`

Класс `ComplexNumber` (см. ил. 12.17) перегружает операторы сложения (+), вычитания (-) и умножения (\*), чтобы программа могла складывать, вычитать и перемножать экземпляры класса `ComplexNumber` с использованием стандартной математической записи. В строках 9 и 12 определяются свойства `Real` и `Imaginary` для действительной и мнимой части комплексного числа.

```

1 // Ил. 12.17: ComplexNumber.cs
2 // Класс, перегружающий операторы для сложения, вычитания
3 // и умножения комплексных чисел.
4 using System;
5
6 public class ComplexNumber
7 {
8     // Свойство для получения действительной части (только для чтения)
9     public double Real { get; private set; }
10
11     // Свойство для получения мнимой части (только для чтения)
12     public double Imaginary { get; private set; }
13
14     // Конструктор
15     public ComplexNumber( double a, double b )
16     {
17         Real = a;
18         Imaginary = b;
19     } // Конец конструктора
20
21     // Получение строкового представления ComplexNumber
22     public override string ToString()
23     {
24         return string.Format( "{0} {1} {2}i)",
25             Real, ( Imaginary < 0 ? "-" : "+" ), Math.Abs( Imaginary ) );
26     } // Конец метода ToString
27
28     // Перегрузка оператора сложения
29     public static ComplexNumber operator+ (
30         ComplexNumber x, ComplexNumber y )

```

**Ил. 12.17.** Класс с перегрузкой операторов сложения, вычитания и умножения комплексных чисел (продолжение ➊)

```

31 {
32     return new ComplexNumber( x.Real + y.Real,
33         x.Imaginary + y.Imaginary );
34 } // Конец оператора +
35
36 // Перегрузка оператора вычитания
37 public static ComplexNumber operator- (
38     ComplexNumber x, ComplexNumber y )
39 {
40     return new ComplexNumber( x.Real - y.Real,
41         x.Imaginary - y.Imaginary );
42 } // Конец оператора -
43
44 // Перегрузка оператора умножения
45 public static ComplexNumber operator* (
46     ComplexNumber x, ComplexNumber y )
47 {
48     return new ComplexNumber(
49         x.Real * y.Real - x.Imaginary * y.Imaginary,
50         x.Real * y.Imaginary + y.Real * x.Imaginary );
51 } // Конец оператора *
52 } // Конец класса ComplexNumber

```

**Ил. 12.17.** Класс с перегрузкой операторов сложения, вычитания и умножения комплексных чисел (окончание)

В строках 29–34 оператор сложения (+) перегружается для сложения объектов `ComplexNumber`. Ключевое слово `operator`, за которым следует обозначение оператора, указывает, что метод перегружает указанный оператор. Методы, перегружающие бинарные операторы, должны получать два аргумента: первый определяет левый операнд, а второй — правый операнд. Перегруженный оператор + класса `ComplexNumber` получает две ссылки на объекты `ComplexNumber` и возвращает объект `ComplexNumber`, представляющий сумму аргументов. Этот метод помечен ключевыми словами `public` и `static`, что необходимо для перегруженных операторов. Тело метода (строки 32–33) выполняет сложение и возвращает результат в виде нового объекта `ComplexNumber`. Обратите внимание: мы не изменяем содержимое ни одного из исходных операндов, переданных в аргументах `x` и `y`. Это соответствует интуитивным представлениям о поведении оператора — при сложении двух чисел исходные слагаемые не изменяются. Строки 37–51 определяют аналогичные перегруженные операторы для вычитания и умножения объектов `ComplexNumber`.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 12.7

Перегружайте операторы для выполнения объектами классов операций, совпадающих или сходных с операциями объектов простых типов. Избегайте неинтуитивного использования операторов.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 12.8

По крайней мере один параметр перегруженного операторного метода должен быть ссылкой на объект класса, в котором перегружается оператор. Тем самым предотвращается изменение поведения операторов для простых типов.

## Класс ComplexNumber

Класс ComplexTest (ил. 12.18) демонстрирует использование перегруженных в классе ComplexNumber операторов +, - и \*. В строках 14–27 пользователю предлагается ввести два комплексных числа, после чего программа создает два объекта ComplexNumber и присваивает их переменным x и y.

```

1  // Ил. 12.18: ComplexTest.cs
2  // Перегрузка операторов для комплексных чисел.
3  using System;
4
5  public class ComplexTest
6  {
7      public static void Main( string[] args )
8      {
9          // Объявление двух переменных для хранения комплексных чисел,
10         // вводимых пользователем
11         ComplexNumber x, y;
12
13         // Программа запрашивает первое комплексное число
14         Console.Write( "Enter the real part of complex number x: " );
15         double realPart = Convert.ToDouble( Console.ReadLine() );
16         Console.Write(
17             "Enter the imaginary part of complex number x: " );
18         double imaginaryPart = Convert.ToDouble( Console.ReadLine() );
19         x = new ComplexNumber( realPart, imaginaryPart );
20
21         // Программа запрашивает второе комплексное число
22         Console.Write( "\nEnter the real part of complex number y: " );
23         realPart = Convert.ToDouble( Console.ReadLine() );
24         Console.Write(
25             "Enter the imaginary part of complex number y: " );
26         imaginaryPart = Convert.ToDouble( Console.ReadLine() );
27         y = new ComplexNumber( realPart, imaginaryPart );
28
29         // Вывод результатов вычислений с x и y
30         Console.WriteLine();
31         Console.WriteLine( "{0} + {1} = {2}", x, y, x + y );
32         Console.WriteLine( "{0} - {1} = {2}", x, y, x - y );
33         Console.WriteLine( "{0} * {1} = {2}", x, y, x * y );
34     } // Конец метода Main
35 } // Конец класса ComplexTest

```

```

Enter the real part of complex number x: 2
Enter the imaginary part of complex number x: 4

```

```

Enter the real part of complex number y: 4
Enter the imaginary part of complex number y: -2

```

```

(2 + 4i) + (4 - 2i) = (6 + 2i)
(2 + 4i) - (4 - 2i) = (-2 + 6i)
(2 + 4i) * (4 - 2i) = (16 + 12i)

```

**Ил. 12.18.** Перегрузка операторов для комплексных чисел

В строках 31–33 выполняются операции сложения, вычитания и умножения  $x$  и  $y$  с перегруженными операторами и выводятся результаты этих операций. В строке 31 операция сложения выполняется применением оператора  $+$  к операндам  $x$  и  $y$  класса `ComplexNumber`. Без перегрузки оператора выражение  $x + y$  не имеет смысла — компилятор не будет знать, как складывать два объекта класса `ComplexNumber`. В нашем примере это выражение осмысленно, потому что мы определили оператор  $+$  для двух объектов `ComplexNumber` в строках 29–34 на ил. 12.17. При «сложении» двух объектов `ComplexNumber` в строке 31 на ил. 12.18 вызывается метод оператора  $+$ ; в первом аргументе передается левый операнд, а во втором — правый операнд. При использовании операторов вычитания и умножения в строках 32–33 соответствующие операторные методы вызываются аналогичным образом.

Результат каждого вычисления представляет собой ссылку на новый объект `ComplexNumber`. При передаче нового объекта методу `WriteLine` класса `Console` неявно вызывается его метод `ToString` (см. ил. 12.17, строки 22–26). Строку 31 на ил. 12.18 можно переписать с явным вызовом метода `ToString` для объекта, созданного перегруженным оператором  $+$ :

```
Console.WriteLine( "{0} + {1} = {2}", x, y, ( x + y ).ToString() );
```

## 12.9. Итоги

Эта глава посвящена полиморфизму — возможности обработки объектов, имеющих общий базовый класс в иерархии классов, словно все они являются объектами базового класса. В этой главе вы узнали, как полиморфизм упрощает расширение и сопровождение системы и как использовать перегруженные методы для проявления полиморфного поведения. Мы рассмотрели концепцию абстрактного класса — определения базового класса, от которого наследуют другие классы. Абстрактный класс может объявлять абстрактные методы, которые должны быть реализованы каждым производным классом, чтобы он стал конкретным классом; приложение может использовать переменные абстрактного класса для полиморфного вызова реализаций абстрактных методов в производных классах. Вы также узнали, как определить фактический тип объекта во время выполнения и как создаются запечатанные методы и классы. Мы обсудили объявление и реализацию интерфейсов как другой способ достижения полиморфного поведения, часто среди объектов разных классов. В завершающей части главы было показано, как определить поведение встроенных операторов для объектов ваших классов с использованием перегрузки операторов.

# 13

## Обработка исключений: следующий шаг

### 13.1. Введение

В этой главе более подробно рассмотрена обработка исключений. Как вы знаете из раздела 8.4, исключение свидетельствует о проблеме, возникшей во время выполнения программы. Сам термин «исключение» говорит о том, что хотя проблема возможна, она происходит относительно редко. Как было показано в разделе 8.4 и главе 10, механизм обработки исключений позволяет отреагировать на исключение в приложении, а во многих случаях даже продолжить выполнение программы так, словно никаких проблем не было. С более серьезной проблемой нормальное выполнение может стать невозможным; в этом случае программа оповещает пользователя о проблеме и организовано завершается. Инструменты, представленные в этой главе, позволят вам писать надежный, понятный и устойчивый к ошибкам код (то есть программы, способные справиться с возникающими проблемами и продолжить выполнение). Рекомендации по обработке исключений в Visual C# приведены в документации Visual Studio<sup>1</sup>.

После рассмотрения основных концепций и приемов обработки исключений мы рассмотрим иерархию классов исключений .NET. Программы в процессе выполнения часто запрашивают и освобождают ресурсы (например, файлы на диске). Часто запас таких ресурсов ограничен, или же ресурсы могут использоваться только одной программой в любой момент времени. В этой главе будет рассмотрена та часть механизма обработки исключений, которая позволяет программе использовать ресурс, а затем гарантировать его освобождение для других программ даже при возникновении исключения. Мы изучим некоторые свойства класса `System.Exception` (базовый класс всех классов исключений) и поговорим о том, как создавать и использовать ваши собственные классы исключений.

---

<sup>1</sup> «Best Practices for Handling Exceptions [C#]», .NET Framework Developer's Guide, электронная документация Visual Studio .NET ([msdn.microsoft.com/en-us/library/seyhstzts.aspx](http://msdn.microsoft.com/en-us/library/seyhstzts.aspx)).

### О версии Visual Studio, использованной в этой главе

Все более ранние версии Visual Studio включали полезный инструмент Exception Assistant (см. раздел 13.3.3). К сожалению, в различных изданиях Visual Studio Express 2012 этот инструмент был недоступен (на момент написания книги), поэтому в этой главе используется Visual Studio Professional 2012. Если вы работаете в Visual Studio Express 2012 for Windows Desktop, программы этой главы все равно будут выполняться. Начиная со следующего раздела мы будем упоминать о различиях между версиями IDE при запуске программы командой `DEBUG ▶ StartDebugging` и возникновении исключений.

## 13.2. Пример: деление на нуль без обработки исключений

Давайте посмотрим, что произойдет при возникновении ошибок в консольном приложении, не использующем обработку исключений. Программа на ил. 13.1 запрашивает два целых числа и выполняет целочисленное деление первого числа на второе для получения результата `int`. В этом примере при обнаружении проблемы, которую метод не может обработать, происходит (инициируется) исключение.

```
1 // Ил. 13.1: DivideByZeroNoExceptionHandling.cs
2 // Целочисленное деление без обработки ошибок.
3 using System;
4
5 class DivideByZeroNoExceptionHandling
6 {
7     static void Main()
8     {
9         // Получение делимого
10        Console.Write( "Please enter an integer numerator: " );
11        int numerator = Convert.ToInt32( Console.ReadLine() );
12
13        // Получение делителя
14        Console.Write( "Please enter an integer denominator: " );
15        int denominator = Convert.ToInt32( Console.ReadLine() );
16
17        // Деление двух целых чисел и вывод результата
18        int result = numerator / denominator;
19        Console.WriteLine( "\nResult: {0:D} / {1:D} = {2:D}",
20            numerator, denominator, result );
21    } // Конец Main
22 } // Конец класса DivideByZeroNoExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
```

```
Result: 100 / 7 = 14
Please enter an integer numerator: 100
Please enter an integer denominator: 0
```

**Ил. 13.1.** Целочисленное деление без обработки исключений (продолжение ↗)



```
Unhandled Exception: System.DivideByZeroException:
Attempted to divide by zero.
at DivideByZeroNoExceptionHandling.Main()
in C:\examples\ch13\Fig13_01\DivideByZeroNoExceptionHandling\
DivideByZeroNoExceptionHandling\
DivideByZeroNoExceptionHandling.cs: line 18
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
```

```
Unhandled Exception: System.FormatException:
Input string was not in a correct format.
at System.Number.StringToNumber(String str, NumberStyles options,
    NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
at System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
at DivideByZeroNoExceptionHandling.Main()
in C:\examples\ch13\Fig13_01\DivideByZeroNoExceptionHandling\
DivideByZeroNoExceptionHandling\
DivideByZeroNoExceptionHandling.cs: line 15
```

**Ил. 13.1.** Целочисленное деление без обработки исключений (окончание)

### Запуск приложения

В большинстве наших примеров приложение одинаково работает как в режиме отладки, так и без него. Как мы вскоре обсудим, приложение на ил. 13.1 может вызвать исключения в зависимости от того, какие данные введет пользователь. Если вы запустите приложение в Visual Studio Express 2012 for Windows Desktop командой меню **DEBUG** ► **StartDebugging** и произойдет исключение, в IDE открывается диалоговое окно, которое выглядит примерно так.



Кнопка **Break** приостанавливает программу в строке, в которой произошло исключение, чтобы вы могли проанализировать состояние программы и отладить ее.

В нашем примере отладка не нужна; мы просто хотим посмотреть, что происходит при возникновении ошибки. По этой причине приложение запускается командой `DEBUG ▶ StartWithoutDebugging`. Если во время выполнения происходит исключение, появляется диалоговое окно с сообщением о том, что приложение «прервало работу». Кнопка `Close the Program` просто завершает приложение. В окне вывода появляется сообщение об ошибке. В листинге на ил. 13.1 сообщения об ошибках были отформатированы для удобства чтения.

### Успешное выполнение

В первом примере деление было выполнено успешно.

### Попытка деления на ноль

Во втором примере пользователь вводит делитель 0. Для недопустимых входных данных выводятся несколько строк с информацией. Эта информация, называемая *трассировкой стека*, включает имя класса исключения (`System.DivideByZeroException`), описание возникшей проблемы и путь выполнения, приведший к исключению. Трассировка стека помогает в отладке проблемы. В первой строке сообщения указано, что произошло исключение `DivideByZeroException`. Когда программа делит целое число на 0, CLR выдает исключение `DivideByZeroException` (пространство имен `System`). Текст после имени исключения (`Attempted to divide by zero`) сообщает, что стало причиной исключения. Деление на ноль в целочисленной арифметике недопустимо<sup>1</sup>. Каждая строка «at» в трассировке стека обозначает строку кода метода, которая выполнялась на момент возникновения исключения. В строке «at» указано пространство имен, класс и метод, в котором произошло исключение (`DivideByZeroNoExceptionHandling.Main`), местонахождение и имя файла с кодом (`C:\examples\ch13\Fig13_01\DivideByZeroNoExceptionHandling\DivideByZeroNoExceptionHandling\DivideByZeroNoExceptionHandling.cs`) и номер строки (`:line 18`), в которой произошло исключение. В данном случае из трассировки видно, что исключение `DivideByZeroException` произошло во время выполнения строки 18 метода `Main`. Первая строка «at» в трассировке задает позицию выдачи исключения — исходную точку, в которой оно произошло (то есть строка 18 метода `Main`). По этой информации можно легко определить, какой вызов метода привел к исключению и какие методы вызывались для перехода к этой точке программы.

### Попытка ввести некорректное значение делителя

В третьем примере вместо целочисленного делителя пользователь вводит строку `"hello"`. Происходит исключение `FormatException`, а приложение выдает другую трассировку стека. Если вместо числа пользователь ввел какое-то другое значение,

<sup>1</sup> В вычислениях с плавающей точкой деление на ноль разрешено, а его результатом является бесконечность, представленная константой `Double.PositiveInfinity` или `Double.NegativeInfinity` (в зависимости от положительности или отрицательности делимого). Значения отображаются в формате `Infinity` или `-Infinity`. Если и делимое и делитель равны 0, то результатом вычисления является константа `Double.NaN` («Not A Number») — признак неопределенного результата.

происходит исключение `FormatException` (пространство имен `System`) — например, если метод `Convert` класса `ToInt32` получает строку, не представляющую действительное целое число. Начиная с последней строки «at», в трассировке стека мы видим, что в строке 15 метода `Main` произошло исключение. В данных трассировки также перечислены другие методы, приведшие к точке выдачи исключения. Для выполнения своей задачи `Convert.ToInt32` вызывает метод `Number.ParseInt32`, который, в свою очередь вызывает `Number.StringToNumber`. Исключение выдается непосредственно в методе `Number.StringToNumber`, на что указывает первая строка «at» в трассировке стека. Метод `Convert.ToInt32` в трассировку не включен, потому что компилятор исключил его вызов из кода в процессе оптимизации; вся полезная работа этого метода сводилась к передаче аргументов `Number.ParseInt32`.

### Завершение программы вследствие необработанного исключения

При выдаче необработанного исключения программа на ил. 13.1 завершается с выдачей трассировки стека. Такое происходит не всегда — иногда программа может продолжить выполнение даже при возникновении исключения и выводе трассировки. В таких случаях приложение выдает некорректные результаты. В следующем разделе вы узнаете, как организовать обработку исключений, чтобы программа могла продолжить работу до нормального завершения.

## 13.3. Пример: обработка исключений `DivideByZeroException` и `FormatException`

Рассмотрим простой пример обработки исключений. В приложении на ил. 13.2 средства C# используются для обработки исключений `DivideByZeroException` и `FormatException`, возникающих в программе. Приложение запрашивает у пользователя два целых числа (строки 18–21). Если пользователь ввел целые числа, а делитель отличен от 0, строка 25 выполняет деление, а в строках 28–29 выводится результат. Но если пользователь ввел нецелые данные или ввел делитель, равный 0, происходит исключение. Программа демонстрирует перехват и обработку таких исключений; в данном примере выдается сообщение об ошибке, а пользователю предлагается ввести другие значения.

```
1 // Ил. 13.2: DivideByZeroExceptionHandling.cs
2 // Обработчик исключений FormatException и DivideByZeroException.
3 using System;
4
5 class DivideByZeroExceptionHandling
6 {
7     static void Main( string[] args )
8     {
9         bool continueLoop = true; // Признак продолжения цикла
10
11         do
```

**Ил. 13.2.** Обработчики исключений `FormatException` и `DivideByZeroException` (продолжение ↗)

```

12     {
13         // Получение ввода и вычисление частного
14         try
15         {
16             // Если аргумент не преобразуется в целое число,
17             // метод Convert.ToInt32 выдает FormatException
18             Console.Write( "Enter an integer numerator: " );
19             int numerator = Convert.ToInt32( Console.ReadLine() );
20             Console.Write( "Enter an integer denominator: " );
21             int denominator = Convert.ToInt32( Console.ReadLine() );
22
23             // Если делитель равен 0, при делении
24             // выдается исключение DivideByZeroException
25             int result = numerator / denominator;
26
27             // Вывод результата
28             Console.WriteLine( "\nResult: {0} / {1} = {2}",
29                 numerator, denominator, result );
30             continueLoop = false;
31         } // Конец try
32         catch ( FormatException formatException )
33         {
34             Console.WriteLine( "\n" + formatException.Message );
35             Console.WriteLine(
36                 "You must enter two integers. Please try again.\n" );
37         } // Конец catch
38         catch ( DivideByZeroException divideByZeroException )
39         {
40             Console.WriteLine( "\n" + divideByZeroException.Message );
41             Console.WriteLine(
42                 "Zero is an invalid denominator. Please try again.\n" );
43         } // Конец catch
44     } while ( continueLoop ); // Конец do...while
45 } // Конец Main
46 } // Конец класса DivideByZeroExceptionHandling

```

```

Please enter an integer numerator: 100
Please enter an integer denominator: 7

```

```
Result: 100 / 7 = 14
```

```

Enter an integer numerator: 100
Enter an integer denominator: 0

```

```

Attempted to divide by zero.
Zero is an invalid denominator. Please try again.

```

```

Enter an integer numerator: 100
Enter an integer denominator: 7

```

```
Result: 100 / 7 = 14
```

### Ил. 13.2. Обработчики исключений FormatException и DivideByZeroException (продолжение ↗)

```
Enter an integer numerator: 100
Enter an integer denominator: hello

Input string was not in a correct format.
You must enter two integers. Please try again.

Enter an integer numerator: 100
Enter an integer denominator: 7

Result: 100 / 7 = 14
```

### Ил. 13.2. Обработчики исключений `FormatException` и `DivideByZeroException` (окончание)

Прежде чем рассматривать подробности кода, стоит сказать пару слов о результатах выполнения программы. В первом примере представлено успешное вычисление: пользователь вводит делимое 100 и делитель 7. Результат (14) является целым числом, потому что целочисленное деление всегда дает результат типа `int`. Во втором примере продемонстрирован результат деления на ноль. В целочисленной арифметике CLR проверяет возможное деление на 0 и выдает исключение `DivideByZeroException`, если делитель равен 0. Программа обнаруживает исключение и выводит сообщение об ошибке. В последнем примере представлен результат ввода некорректного значения — вместо делителя пользователь вводит строку "hello". Программа пытается преобразовать введенные строки в целые числа методом `Convert.ToInt32` (строки 19 и 21). Если аргумент не преобразуется к типу `int`, метод выдает исключение `FormatException`. Программа перехватывает исключение и выводит сообщение о том, что пользователь должен ввести два значения `int`.

#### Другой способ преобразования строк в целые числа

Также существует другой способ проверки введенных данных: метод `Int32.TryParse` проверяет, возможно ли преобразование строки в значение `int`. Метод `TryParse` существует у всех числовых типов. Метод получает два аргумента — строку и переменную, в которой должно храниться преобразованное значение. Метод возвращает значение `true` только в том случае, если строка была преобразована успешно. Если преобразовать строку не удалось, второму аргументу присваивается 0 (аргумент передается по ссылке, поэтому он может быть изменен в вызываемом методе). Метод `TryParse` может использоваться для программной проверки ввода вместо выдачи исключения; обычно этот способ является предпочтительным.

### 13.3.1. Размещение кода в блоке `try`

А теперь проанализируем действия пользователя и логику, которая приводит к результатам в приведенных примерах. В строках 14–31 определяется блок `try`, в который заключается код, способный выдавать исключения, а также код, который пропускается при возникновении исключения. Например, программа не должна

выводить результат (строки 28–29), если вычисления в строке 25 не будут выполнены успешно.

Пользователь вводит числа, представляющие делимое и делитель. Две команды, читающие данные `int` (строки 19 и 21), вызывают метод `Convert.ToInt32` для преобразования строк в значения `int`. Если этот метод не может преобразовать свой аргумент `string` в `int`, он выдает исключение `FormatException`. Если преобразование в строках 19 и 21 выполняется корректно (то есть без возникновения исключений), то строка 25 выполняет деление и присваивает результат переменной `result`. Если делитель равен 0, то в строке 25 CLR выдает исключение `DivideByZeroException`. Если в строке 25 исключение не выдается, то строки 28–29 выводят результат деления.

### 13.3.2. Перехват исключений

Код обработки исключений находится в блоке `catch`. В общем случае при возникновении исключения в блоке `try` соответствующий блок `catch` перехватывает исключение и обрабатывает его. За блоком `try` в этом примере следуют два блока `catch` — один обрабатывает исключение `FormatException` (строки 32–37), а другой обрабатывает `DivideByZeroException` (строки 38–43). Параметр исключения в блоке `catch` представляет исключение, которое может обрабатываться блоком `catch`. Блок `catch` может использовать идентификатор параметра (выбранный вами) для взаимодействия с объектом перехваченного исключения. Если использовать объект исключения в блоке `catch` не нужно, идентификатор параметра исключения можно опустить. Типом параметра `catch` является тип исключения, обрабатываемого блоком `catch`. Также возможен блок `catch`, в котором тип исключения не указан, — такой блок `catch` перехватывает все типы исключений (и называется обобщенным условием `catch`). За блоком `try` должен следовать минимум один блок `catch` и/или блок `finally` (см. раздел 13.5). На ил. 13.2 первый блок `catch` перехватывает исключения `FormatException` (выдаваемые методом `Convert.ToInt32`), а второй блок `catch` перехватывает исключения `DivideByZeroException` (которые выдает CLR). При возникновении исключения программа выполняет только первый подходящий блок `catch`. Оба обработчика в приведенном примере выводят диалоговое окно с сообщением об ошибке. После завершения одного из блоков `catch` выполнение программы продолжается с первой команды, следующей за последним блоком `catch` (конец метода в данном примере). Вскоре передача управления при обработке исключений будет рассмотрена более подробно.

### 13.3.3. Неперехваченные исключения

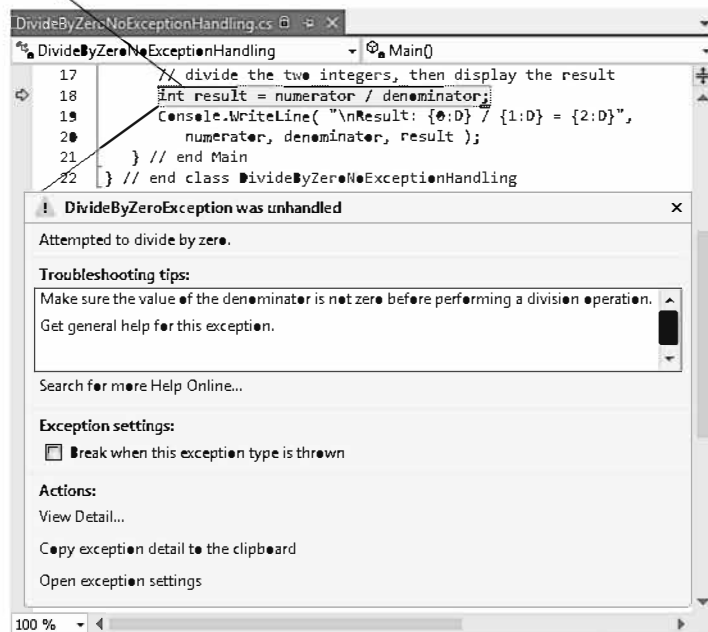
Неперехваченным (или необработанным) исключением называется исключение, для которого не найден подходящий блок `catch`. Результаты неперехваченных исключений представлены во втором и третьем результатах на ил. 13.1. Вспомните, что при возникновении исключений в этом примере приложение аварийно завершается

(после выдачи трассировки стека). Результат перехваченного исключения зависит от способа запуска программы — на ил. 13.1 показан результат возникновения исключения при запуске приложения командой **DEBUG ▶ Start Without Debugging**. Если запустить приложение из другой версии Visual Studio (отличной от Express) командой **DEBUG ▶ Start Debugging**, то при обнаружении перехваченного исключения приложение приостанавливается, а на экране появляется диалоговое окно **Exception Assistant** со следующей информацией:

- ☐ линия от окна **Exception Assistant** к строке кода, вызвавшей исключение;
- ☐ тип исключения;
- ☐ диагностические рекомендации со ссылками на полезную информацию относительно возможной причины исключения и способа его обработки;
- ☐ ссылки для просмотра или копирования полной информации об исключении.

На ил. 13.3 изображено окно **Exception Assistant**, которое выводится при попытке деления на 0 в приложении на ил. 13.1.

Точка срабатывания исключения



Ил. 13.3. Окно **Exception Assistant**

### 13.3.4. Модель обработки исключений

Напомним, что точка программы, в которой возникло исключение, называется *точкой выдачи исключения* — эта позиция играет важную роль в процессе отладки

(см. раздел 13.7). Если исключение происходит в блоке `try` (как, например, при выдаче исключения `FormatException` в результате выполнения кода в строках 19 и 21 на ил. 13.2), то блок `try` завершается немедленно, а управление в программе передается первому из блоков `catch`, у которых тип параметра исключения соответствует типу выданного исключения. На ил. 13.2 первый блок `catch` перехватывает исключения `FormatException` (возникающие при вводе значения недопустимого типа); второй блок `catch` перехватывает исключения `DivideByZeroException` (возникающие при попытке деления на ноль). После того как исключение будет обработано, программа не возвращается в точку выдачи исключения, потому что она уже вышла из блока `try` (а все локальные переменные вышли из области действия). Вместо этого выполнение продолжается с позиции за последним блоком `catch`. Такая модель обработки исключений называется *моделью с завершением*. [Примечание: в некоторых языках используется *модель с продолжением*, при которой после обработки исключения программа продолжает выполнение после точки выдачи исключения.]

Если в блоке `try` исключения не возникали, то программа на ил. 13.2 успешно завершает блок `try`, игнорируя блоки `catch` в строках 32–37 и 38–43, и передает управление в строку 43. Далее выполняется первая команда, следующая за блоками `try` и `catch`. В нашем примере достигается конец цикла `do...while` (строка 44), метод завершается, а программа ожидает следующего взаимодействия с пользователем.

Блок `try` с соответствующими блоками `catch` и `finally` образуют команду `try`. Не путайте термины «блок `try`» и «команда `try`» — первым обозначается блок кода, следующий за ключевым словом `try` (до блока `catch` или `finally`), тогда как второй обозначает *весь* код от открывающего ключевого слова `try` до конца последнего блока `catch` или `finally`. В команду `try` входит как блок `try`, так и все сопутствующие блоки `catch` и блок `finally`.

При завершении блока `try` локальные переменные, определенные в этом блоке, выходят из области действия. Если блок `try` завершается из-за исключения, CLR ищет первый подходящий блок `catch`, способный обработать тип возникшего исключения. CLR ищет совпадения, сравнивая тип исключения с типом параметра каждого условия `catch`. Совпадение считается обнаруженным, если типы совпадают или если тип выданного исключения является производным классом от типа параметра `catch`. Когда для исключения будет найден подходящий блок `catch`, выполняется код этого блока, а другие блоки `catch` в команде `try` игнорируются.

### 13.3.5. Передача управления при возникновении исключений

В третьем примере на ил. 13.2 пользователь вводит вместо делителя строку "hello". При выполнении строки 21 метод `Convert.ToInt32` не может преобразовать эту строку в `int`, поэтому метод выдает исключение `FormatException`. При возникновении исключения происходит выход из блока `try`; CLR пытается найти подходящий блок



`catch`. Совпадение обнаруживается в блоке `catch` из строки 32, так что обработчик исключения выводит значение свойства `Message` объекта исключения (для получения сообщения об ошибке, связанного с исключением), а все остальные обработчики после блока `try` игнорируются. Выполнение программы продолжается со строки 44.



#### ТИПИЧНАЯ ОШИБКА 13.1

Не пытайтесь включить в блок `catch` список параметров, разделенных запятыми. Блок `catch` может содержать не более одного параметра.

Во втором примере на ил. 13.2 пользователь вводит делитель 0. При выполнении деления в строке 25 происходит исключение `DivideByZeroException`. И снова блок `try` завершается, а программа пытается найти подходящий блок `catch`. На этот раз первый блок `catch` не подходит — тип исключения в объявлении обработчика `catch` не совпадает с типом инициированного исключения, а `FormatException` не является базовым классом `DivideByZeroException`. Программа продолжает поиск блока `catch` и находит его в строке 38. В строке 40 выводится свойство `Message` объекта исключения, а выполнение программы снова продолжается в строке 44.

## 13.4. Иерархия исключений .NET

Механизм обработки исключений C# поддерживает выдачу и перехват объектов исключений только класса `Exception` (пространство имен `System`) и его производных классов. Однако следует учитывать, что программы C# могут взаимодействовать с программными компонентами, написанными на других языках .NET (таких, как C++), не ограничивающих типы исключений. Для перехвата таких исключений можно использовать универсальный блок `catch`.

В этом разделе рассматриваются некоторые классы исключений .NET Framework, производные от класса `Exception`. Вы также узнаете, как определить, выдает ли некоторый метод исключения.

### 13.4.1. Класс `SystemException`

Класс `Exception` (пространство имен `System`) является базовым классом иерархии классов исключений .NET. Среди производных классов важную роль играет класс `SystemException`. Исключения `SystemException` генерируются средой CLR. Многих из них можно избежать при аккуратном программировании приложения. Например, если программа пытается обратиться к элементу за границей массива, CLR выдает исключение типа `IndexOutOfRangeException` (производный класс от `SystemException`). Аналогичным образом исключение происходит, когда программа пытается вызвать метод через переменную ссылочного типа, которая содержит `null`. При этом выдается исключение `NullReferenceException` (другой класс, производный от

`SystemException`). Ранее в этой главе было показано, что исключение `DivideByZeroException` происходит при попытке целочисленного деления на нуль.

Среди других исключений, выдаваемых CLR, стоит упомянуть исключения `OutOfMemoryException`, `StackOverflowException` и `ExecutionEngineException`, которые выдаются в ситуациях, приводящих к нестабильности CLR. Иногда такие исключения даже не перехватываются. Лучше просто сохранить информацию о них в журнале (при помощи таких инструментов, как Apache log4net, — см. [logging.apache.org/log4net/](http://logging.apache.org/log4net/)), а затем завершить приложение.

Преимущество иерархии классов исключений заключается в том, что блок `catch` может перехватывать исключения конкретного типа, или же (из-за отношений «является частным случаем» при наследовании) базовый класс может использоваться для перехвата исключений в иерархии взаимосвязанных типов исключений. Например, в разделе 13.3.2 рассматривается блок `catch` без параметров, который перехватывает любые типы исключений (в том числе и не являющиеся производными от `Exception`). Блок `catch` с параметром типа `Exception` может перехватывать любые исключения, производные от `Exception`, потому что `Exception` является базовым классом всех классов исключений. Преимущество такого подхода заключается в том, что обработчик может обратиться к информации перехваченного исключения через параметр. Использование информации исключений более подробно рассматривается в разделе 13.7.

Использование наследования с исключениями позволяет организовать перехват логически связанных исключений. Набор обработчиков исключений может перехватывать исключения всех производных классов по отдельности, но перехват исключения базового класса обеспечивает более компактную запись. Впрочем, этот прием имеет смысл только в том случае, если обработка для базового класса и всех производных классов выполняется одинаково.



### ТИПИЧНАЯ ОШИБКА 13.2

Если блок `catch` для исключения базового класса находится до блока `catch` любого из производных классов этого класса, компилятор выдает сообщение об ошибке. В этом случае блок `catch` базового класса перехватывает все исключения базового класса и производных классов, так что обработчик исключения производного класса никогда не выполняется.

## 13.4.2. Определение исключений, иницируемых методом

Как определить, может ли в программе произойти исключение? Для методов классов .NET Framework следует прочитать подробное описание в электронной документации. Если метод выдает исключение, в его описании присутствует раздел `Exceptions` с информацией об исключениях и краткими описаниями их причин. Например,

поищите в документации Visual Studio описание метода `Convert.ToInt32`. В разделе Exceptions веб-страницы этого метода указано, что метод `Convert.ToInt32` выдает исключения двух типов — `FormatException` и `OverflowException`, а также приводится информация о возможных причинах. [*Примечание:* эту информацию также можно найти в окне Object Browser — см. раздел 10.12.]



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 13.1

Если метод выдает исключения, то команды, прямо или косвенно вызывающие этот метод, следует заключать в блоки `try`, а исключения должны перехватываться и обрабатываться программой.

Труднее определить, когда исключения могут выдаваться средой CLR. Такая информация содержится в спецификации C# Language Specification (см. [bit.ly/CSharp-4Spec](http://bit.ly/CSharp-4Spec)). В этом документе определяется синтаксис исключений C# и описываются ситуации, в которых выдаются исключения.

## 13.5. Блок finally

Программы часто запрашивают и освобождают ресурсы динамически (то есть во время выполнения). Например, программа, читающая данные из файла на диске, сначала выдает запрос на открытие файла (см. главу 17). Если запрос завершается успешно, то программа читает содержимое файла. Операционные системы обычно запрещают нескольким программам одновременно осуществлять запись в файл. Таким образом, при завершении работы с файлом программа должна закрыть его (то есть освободить ресурс), чтобы он мог использоваться другими программами. Если файл не был закрыт, возникает утечка ресурсов, а ресурс файла становится недоступным для других программ.

В таких языках программирования, как C и C++, в которых программист отвечает за динамическое управление памятью, самым распространенным видом утечки ресурсов является утечка памяти. Она возникает в том случае, когда программа выделяет память (в программах C# это делается ключевым словом `new`), но не освобождает ее после использования. Обычно в C# утечка памяти не создает проблем, потому что CLR освобождает неиспользуемую память посредством уборки мусора (см. раздел 10.8). Тем не менее возможна утечка ресурсов других видов — например, незакрытых файлов.



### КАК ИЗБЕЖАТЬ ОШИБОК 13.1

CLR не устраняет утечку памяти полностью. Объекты уничтожаются уборщиком мусора только после того, как на них не остается ни одной ссылки, и даже после этого возможны задержки до того момента, когда системе потребуется память. Таким образом, непреднамеренное хранение ссылок на ненужные объекты может привести к утечке памяти.

## Перемещение кода освобождения ресурсов в блок `finally`

Исключения часто возникают при работе с ресурсами, требующими явного освобождения. Например, программа может получать исключения `IOException` во время работы с файлами. По этой причине код обработки файлов обычно размещается в блоке `try`. Независимо от того, произойдет исключение или нет, программа должна закрыть файл, когда тот станет ненужным. Предположим, весь код запроса и освобождения ресурсов находится в блоке `try`. Если программа выполняется без исключений, то блок `try` будет выполнен нормально и освободит ресурсы после использования. Но при возникновении исключения программа может выйти из блока `try` до того, как выполнится код освобождения ресурсов. Конечно, этот код можно было бы продублировать во всех блоках `catch`, но это усложнит изменение и сопровождение кода. Также код освобождения ресурсов можно было бы разместить после команды `try`; но если блок `try` завершится командой `return` или из-за возникшего исключения, то код, следующий за `try`, никогда не выполнится.

Для решения подобных проблем механизм обработки исключений C# предоставляет блок `finally`, который гарантированно выполняется независимо от того, будет ли блок `try` выполнен успешно или произойдет исключение. Блок `finally` идеально подходит для размещения кода освобождения ресурсов, которые захватываются программой и обрабатываются в соответствующем блоке `try`. Если блок `try` будет выполнен успешно, то блок `finally` выполняется немедленно после его завершения. Если в блоке `try` произойдет исключение, то блок `finally` выполняется немедленно после завершения блока `catch`. Если исключение не будет перехвачено блоком `catch`, связанным с блоком `try`, или этот блок сам выдаст исключение, то блок `finally` выполняется перед обработкой исключения следующим внешним блоком `try`, который может находиться в вызывающем методе. Размещая код освобождения ресурсов в блоке `finally`, мы гарантируем, что даже при завершении программы из-за непревзвешенного исключения ресурс будет освобожден. Локальные переменные в блоке `try` недоступны в соответствующем блоке `finally`. По этой причине переменные, которые должны быть доступны как в блоке `try`, так и в соответствующем блоке `finally`, должны объявляться до блока `try`.



### КАК ИЗБЕЖАТЬ ОШИБОК 13.2

В блоке `finally` обычно размещается код освобождения ресурсов, захваченных в соответствующем блоке `try`; это эффективный механизм предотвращения утечки ресурсов.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 13.1

Как правило, ресурсы должны освобождаться сразу же после завершения работы с ними, чтобы они быстро становились доступными для повторного использования.

Если за блоком `try` следует один или несколько блоков `catch`, наличие блока `finally` не обязательно. Но если за блоком `try` не следует ни один блок `catch`, то за

ним обязательно должен располагаться блок `finally`. Если за блоком `try` следует хотя бы один блок `catch`, блок `finally` (если он есть) должен располагаться за последним блоком `catch`. Блоки в команде `try` могут разделяться только пропусками (whitespace) и комментариями.

### Пример использования блока finally

Приложение на ил. 13.4 демонстрирует, что блок `finally` выполняется *всегда* — независимо от того, произошло ли исключение в соответствующем блоке `try`. Приложение состоит из метода `Main` (строки 8–47) и еще четырех методов, которые `Main` вызывает для демонстрации `finally`: `DoesNotThrowException` (строки 50–67), `ThrowExceptionWithCatch` (строки 70–89), `ThrowExceptionWithoutCatch` (строки 92–108) и `ThrowExceptionCatchRethrow` (строки 111–136).


```

1  // Ил. 13.4: UsingExceptions.cs
2  // Использование блоков finally. Блоки finally
3  // выполняются всегда, даже если исключение не выдавалось.
4  using System;
5
6  class UsingExceptions
7  {
8      static void Main()
9      {
10         // Случай 1: в вызванном методе не было исключений
11         Console.WriteLine( "Calling DoesNotThrowException" );
12         DoesNotThrowException();
13
14         // Случай 2: исключение произошло и перехвачено в вызванном методе
15         Console.WriteLine( "\nCalling ThrowExceptionWithCatch" );
16         ThrowExceptionWithCatch();
17
18         // Случай 3: исключение произошло, но не перехвачено
19         // в вызванном методе из-за отсутствия блока catch.
20         Console.WriteLine( "\nCalling ThrowExceptionWithoutCatch" );
21
22         // Вызов ThrowExceptionWithoutCatch
23         try
24         {
25             ThrowExceptionWithoutCatch();
26         } // Конец try
27         catch
28         {
29             Console.WriteLine( "Caught exception from " +
30                 "ThrowExceptionWithoutCatch in Main" );
31         } // Конец catch
32
33         // Случай 4: исключение произошло и перехвачено в вызванном
34         // методе, затем инициировано заново для вызывающей стороны.
35         Console.WriteLine( "\nCalling ThrowExceptionCatchRethrow" );
36

```

**Ил. 13.4.** Блоки `finally` выполняются всегда, даже если исключение не выдавалось (продолжение ↗)

```
37     // Вызов ThrowExceptionCatchRethrow
38     try
39     {
40         ThrowExceptionCatchRethrow();
41     } // Конец try
42     catch
43     {
44         Console.WriteLine( "Caught exception from " +
45             "ThrowExceptionCatchRethrow in Main" );
46     } // Конец catch
47 } // Конец метода Main
48
49 // Исключения не выдаются
50 static void DoesNotThrowException()
51 {
52     // Блок try не выдает исключения
53     try
54     {
55         Console.WriteLine( "In DoesNotThrowException" );
56     } // Конец try
57     catch
58     {
59         Console.WriteLine( "This catch never executes" );
60     } // Конец catch
61     finally
62     {
63         Console.WriteLine( "finally executed in DoesNotThrowException" );
64     } // Конец finally
65
66     Console.WriteLine( "End of DoesNotThrowException" );
67 } // Конец метода DoesNotThrowException
68
69 // Метод выдает исключение и перехватывает его локально
70 static void ThrowExceptionWithCatch()
71 {
72     // В блоке try выдается исключение
73     try
74     {
75         Console.WriteLine( "In ThrowExceptionWithCatch" );
76         throw new Exception( "Exception in ThrowExceptionWithCatch" );
77     } // Конец try
78     catch ( Exception exceptionParameter )
79     {
80         Console.WriteLine( "Message: " + exceptionParameter.Message );
81     } // Конец catch
82     finally
83     {
84         Console.WriteLine(
85             "finally executed in ThrowExceptionWithCatch" );
86     } // Конец finally
87
88     Console.WriteLine( "End of ThrowExceptionWithCatch" );
89 } // Конец метода ThrowExceptionWithCatch
90
```

**Ил. 13.4.** Блоки finally выполняются всегда, даже если исключение не выдавалось (продолжение )

```

91 // Метод выдает исключение и не перехватывает его локально
92 static void ThrowExceptionWithoutCatch()
93 {
94     // Исключение выдается, но не перехватывается
95     try
96     {
97         Console.WriteLine( "In ThrowExceptionWithoutCatch" );
98         throw new Exception( "Exception in ThrowExceptionWithoutCatch" );
99     } // Конец try
100 finally
101 {
102     Console.WriteLine( "finally executed in " +
103         "ThrowExceptionWithoutCatch" );
104 } // Конец finally
105
106 // Недостижимый код; логическая ошибка
107 Console.WriteLine( "End of ThrowExceptionWithoutCatch" );
108 } // Конец метода ThrowExceptionWithoutCatch
109
110 // Метод выдает исключение, перехватывает его и выдает повторно
111 static void ThrowExceptionCatchRethrow()
112 {
113     // Блок block выдает исключение
114     try
115     {
116         Console.WriteLine( "In ThrowExceptionCatchRethrow" );
117         throw new Exception( "Exception in ThrowExceptionCatchRethrow" );
118     } // Конец try
119     catch ( Exception exceptionParameter )
120     {
121         Console.WriteLine( "Message: " + exceptionParameter.Message );
122
123         // Повторное инициирование исключения для дальнейшей обработки
124         throw;
125
126         // Недостижимый код; логическая ошибка
127     } // Конец catch
128 finally
129 {
130     Console.WriteLine( "finally executed in " +
131         "ThrowExceptionCatchRethrow" );
132 } // Конец finally
133
134 // Находящийся здесь код недостижим
135 Console.WriteLine( "End of ThrowExceptionCatchRethrow" );
136 } // Конец метода ThrowExceptionCatchRethrow
137 } // Конец класса UsingExceptions

```

```

Calling DoesNotThrowException
In DoesNotThrowException
finally executed in DoesNotThrowException
End of DoesNotThrowException

```

```

Calling ThrowExceptionWithCatch

```

**Ил. 13.4.** Блоки finally выполняются всегда, даже если исключение не выдавалось (продолжение ↗)

```
In ThrowExceptionWithCatch
Message: Exception in ThrowExceptionWithCatch
finally executed in ThrowExceptionWithCatch
End of ThrowExceptionWithCatch

Calling ThrowExceptionWithoutCatch
In ThrowExceptionWithoutCatch
finally executed in ThrowExceptionWithoutCatch
Caught exception from ThrowExceptionWithoutCatch in Main

Calling ThrowExceptionCatchRethrow
In ThrowExceptionCatchRethrow
Message: Exception in ThrowExceptionCatchRethrow
finally executed in ThrowExceptionCatchRethrow
Caught exception from ThrowExceptionCatchRethrow in Main
```

**Ил. 13.4.** Блоки `finally` выполняются всегда, даже если исключение не выдавалось (окончание)

В строке 12 метода `Main` вызывается метод `DoesNotThrowException`. В блоке `try` этого метода выводится сообщение (строка 55). Так как блок `try` выполняется без исключений, программа игнорирует блок `catch` (строки 57–60) и выполняет блок `finally` (строки 61–64), в котором выводится сообщение. В этой точке выполнение программы продолжается с первой команды после блока `finally` (строка 66), которая выводит сообщение о достижении конца метода. Затем управление возвращается методу `Main`.

### Инициирование исключений командой `throw`

В строке 16 метода `Main` вызывается метод `ThrowExceptionWithCatch` (строки 70–89), выполнение которого начинается с вывода сообщения в блоке `try` (строки 73–77). Затем блок `try` создает объект `Exception` и иницирует исключение командой `throw` для этого объекта (строка 76). Выполнение команды `throw` указывает на возникновение проблем в коде. Как было показано в предшествующих главах, команда `throw` может использоваться для выдачи исключений в программах. Как и исключения, выдаваемые методами `Framework Class Library` и `CLR`, они сообщают клиентским приложениям о возникновении ошибки. Команде `throw` передается объект иницируемого исключения. Операнд команды `throw` может относиться к типу `Exception` или к любому типу, производному от `Exception`.

Строка, передаваемая конструктору, становится сообщением об ошибке объекта исключения. При выполнении команды `throw` в блоке `try` происходит немедленный выход из блока `try`, а выполнение программы продолжается с первого подходящего блока `catch` (строки 78–81), следующего за блоком `try`. В нашем примере тип выдаваемого исключения (`Exception`) соответствует типу, указанному в `catch`, поэтому в строке 80 выводится сообщение с информацией об исключении. Затем выполняется блок `finally` (строки 82–86) с выводом сообщения. Далее выполнение программы продолжается первой командой после завершения блока `finally` (строка 88), которая выводит сообщение о достижении конца метода. Управление возвращается



методу `Main`. В строке 80 свойство `Message` объекта исключения используется для получения сообщения об ошибке, связанного с исключением (то есть сообщения, переданного конструктору `Exception`). Некоторые свойства класса `Exception` рассматриваются в разделе 13.7.

В строках 23–31 метода `Main` определяется команда `try`, в которой метод `Main` вызывает метод `ThrowExceptionWithoutCatch` (строки 92–108). Блок `try` позволяет `Main` перехватить любое исключение, инициированное в `ThrowExceptionWithoutCatch`. Блок `try` в строках 95–99 `ThrowExceptionWithoutCatch` начинается с вывода сообщения. Далее блок `try` выдает `Exception` (строка 98) и немедленно завершается.

Обычно выполнение программы продолжается с первого блока `catch`, следующего за блоком `try`. Однако в данном случае у блока `try` нет подходящих блоков `catch`, поэтому исключение в методе `ThrowExceptionWithoutCatch` не перехватывается. Управление передается блоку `finally` (строки 100–104), в котором выводится сообщение. В этой точке управление возвращается `Main` — команды, следующие за блоком `finally` (например, строка 107), не выполняются. В нашем примере такие команды могут привести к логическим ошибкам, потому что исключение, выданное в строке 98, не перехватывается. В `Main` блок `catch` в строках 27–31 перехватывает исключение и выводит соответствующее сообщение.

### **Повторная выдача исключений**

В строках 38–46 метода `Main` определяется команда `try`, в которой из метода `Main` вызывается метод `ThrowExceptionCatchRethrow` (строки 111–136). Команда `try` позволяет `Main` перехватывать любые исключения, выданные в `ThrowExceptionCatchRethrow`. Команда `try` в строках 114–132 метода `ThrowExceptionCatchRethrow` начинает работу с выдачи сообщения. Затем блок `try` выдает исключение `Exception` (строка 117). Блок `try` немедленно завершается, а выполнение программы продолжается с первого блока `catch` (строки 119–127) после блока `try`. В нашем примере тип выданного исключения (`Exception`) совпадает с типом, указанным в `catch`, так что строка 121 выдает сообщение о возникшем исключении. В строке 124 команда `throw` используется для повторной выдачи исключения. Она показывает, что блок `catch` выполнил частичную обработку исключения, а теперь передает исключение (в данном случае методу `Main`) для дальнейшей обработки. В общем случае рекомендуется выдать новый объект исключения и указать исходный объект в конструкторе нового исключения, чтобы сохранить информацию трассировки стека от исходного исключения.

Обработка исключения в методе `ThrowExceptionCatchRethrow` не завершена, потому что команда `throw` в строке 124 немедленно завершает блок `catch` — если бы между строкой 124 и концом блока находился какой-то код, то он бы не был выполнен. При выполнении строки 124 метод `ThrowExceptionCatchRethrow` завершается и возвращает управление `Main`. И снова блок `finally` (строки 128–132) выполняется и выводит сообщение перед возвратом управления `Main`. Когда управление возвращается в `Main`, блок `catch` в строках 42–46 перехватывает его и выводит сообщение о перехвате. После этого программа завершается.

### Возвращение после блока `finally`

Следующая команда, выполняемая после завершения блока `finally`, зависит от состояния обработки исключения. Если блок `try` завершился успешно или если блок `catch` перехватил и обработал исключение, то выполнение продолжается со следующей команды после блока `finally`. Но если исключение не было перехвачено или если блок `catch` заново запустил исключение, то выполнение продолжается в следующем внешнем блоке `try`. Внешний блок `try` может находиться в вызывающем методе или в одном из методов, из которых он был вызван. Также команда `try` может быть вложена в блок `try`; в таком случае блоки `catch` внешней команды `try` обрабатывают любые исключения, не перехваченные во внутренней команде `try`. Если блок `try` был выполнен и у него имеется соответствующий блок `finally`, последний выполняется даже в том случае, если выполнение блока `try` завершилось командой `return`. Возврат управления происходит после выполнения блока `finally`.



#### ТИПИЧНАЯ ОШИБКА 13.3

Если во время выполнения блока `finally` неперехваченное исключение ожидает обработки, а блок `finally` выдает новое исключение, которое в нем не перехватывается, первое исключение теряется, а новое исключение передается следующему внешнему блоку `try`.



#### КАК ИЗБЕЖАТЬ ОШИБОК 13.3

При размещении в блоке `finally` кода, способного выдавать исключения, этот код всегда должен заключаться в команду `try` для перехвата соответствующих типов исключений. Тем самым предотвращается потеря неперехваченных и повторно инициированных исключений, возникающих до выполнения блока `finally`.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 13.2

Не пытайтесь заключать каждую команду, которая может выдать исключение, в отдельный блок `try` — вашу программу будет трудно читать. Вместо этого заключите значительную часть кода в один блок `try`, за которым следуют блоки `catch` для всех возможных исключений. За блоками `catch` следует один блок `finally`. Используйте разные блоки `try` для команд, которые могут выдавать исключения одного типа.

## 13.6. Команда `using`

Код освобождения ресурсов обычно размещается в блоке `finally` для того, чтобы ресурсы гарантированно освобождались независимо от того, возникали ли исключения при использовании ресурса в соответствующем блоке `try`. Альтернативная запись — команда `using` (не путайте с директивой `using` для пространств имен) — упрощает написание кода, в котором вы получаете ресурс, используете его в блоке `try` и освобождаете в соответствующем блоке `finally`. Например, приложение для работы с файлами (см. главу 17) может обрабатывать файл в команде `using`,

чтобы после завершения работы файл закрывался. Ресурсом должен быть объект, реализующий интерфейс `IDisposable`, а следовательно, содержащий метод `Dispose`. Обобщенная форма команды `using` выглядит так:

```
using ( ExampleClass exampleObject = new ExampleClass() )
{
    exampleObject.SomeMethod();
}
```

где `ExampleClass` — класс, реализующий интерфейс `IDisposable`. Код создает объект типа `ExampleClass` и использует его в команде, после чего вызывает его метод `Dispose` для освобождения любых ресурсов, задействованных объектом. Команда `using` неявно заключает код, находящийся в ее теле, в блок `try` с соответствующим блоком `finally`, который вызывает метод `Dispose` объекта. Например, приведенный фрагмент эквивалентен следующему:

```
{
    ExampleClass exampleObject = new ExampleClass();
    try
    {
        exampleObject.SomeMethod();
    }
    finally
    {
        if ( exampleObject != null )
            ( ( IDisposable ) exampleObject ).Dispose();
    }
}
```

Команда `if` проверяет, что `exampleObject` все еще содержит ссылку на объект; в противном случае может возникнуть исключение `NullReferenceException`.

## 13.7. Свойства исключений

Как упоминалось в разделе 13.4, типы исключений объявляются производными от класса `Exception`, обладающего рядом свойств. Эти свойства часто используются для определения сообщений об ошибках с информацией о перехваченном исключении; самыми важными из них являются `Message` и `StackTrace`. В свойстве `Message` содержится сообщение об ошибке, связанное с объектом `Exception`: стандартное сообщение, связанное с типом исключения, или пользовательское сообщение, переданное конструктору объекта `Exception` при инициировании исключения. Свойство `StackTrace` содержит строку, представляющую состояние стека вызовов. Вспомните, что исполнительная среда постоянно поддерживает список незавершенных методов, которые были вызваны, но еще не вернули управление. `StackTrace` представляет серию методов, которые не завершили обработку на момент возникновения исключения. Если отладочная информация, сгенерированная компилятором для метода, доступна для IDE, в трассировку стека также включаются номера строк; первый номер строки определяет точку выдачи исключения, а последующие — позиции, из которых вызывались методы в трассировке. Для хранения отладочной информации ваших проектов IDE создает файлы `PDB`.

### Свойство `InnerException`

Также в программах часто используется свойство `InnerException`. Обычно разработчики библиотек классов определяют «обертки» для исключений, перехваченных в своем коде, чтобы выдавать новые типы исключений, специфические для их библиотек. Например, в реализации бухгалтерской системы номера счетов могут вводиться в строковом виде, но представляться в коде в формате `int`. Как говорилось ранее, для преобразования строк в значения `int` может использоваться метод `Convert.ToInt32`, который выдает исключение `FormatException` для недействительных значений. В такой ситуации разработчик бухгалтерской системы предпочтет использовать сообщение об ошибке, отличное от стандартного сообщения `FormatException`, или же определит новый тип исключения — например, `InvalidAccountNumberFormatException`. В таких случаях разработчик пишет код перехвата `FormatException`, создает соответствующий тип объекта `Exception` в блоке `catch` и передает исходное исключение в одном из аргументов конструктора. При возникновении в коде, использующем бухгалтерскую библиотеку, исключения `InvalidAccountNumberFormatException` перехвативший исключение блок `catch` может получить ссылку на исходное исключение через свойство `InnerException`. Таким образом, исключение сообщает, что пользователь ввел недопустимый номер счета, а формат числа недействителен. Если свойство `InnerException` равно `null`, значит, исключение не было порождено другим исключением.

### Другие свойства `Exception`

Класс `Exception` также содержит и другие свойства, в число которых входят `HelpLink`, `Source` и `TargetSite`.

Свойство `HelpLink` задает местонахождение файла справки, описывающего возникшую проблему. Если файл не существует, свойство содержит `null`. Свойство `Source` задает имя приложения или объекта, ставшего причиной исключения. Свойство `TargetSite` задает метод, в котором произошло исключение.

### Использование свойств исключений и раскрытие стека

Следующий пример (илл.13.5) демонстрирует использование свойств `Message`, `StackTrace` и `InnerException` класса `Exception`. Кроме того, в примере представлен механизм *раскрытия стека* — если исключение инициировано, но не перехвачено в конкретной области действия, стек вызова «раскручивается», и среда пытается перехватить исключение в следующем внешнем блоке `try`. Для получения правильной трассировки стека следует выполнить эту программу в последовательности, описанной в разделе 13.2.

```
1 // Ил. 13.5: Properties.cs
2 // Раскрытие стека и свойства класса Exception.
3 // Демонстрация использования свойств Message, StackTrace и InnerException.
4 using System;
5
6 class Properties
```

**Ил. 13.5.** Раскрытие стека и свойства класса `Exception` (продолжение ➤)

```

7 {
8     static void Main()
9     {
10         // Вызов Method1; все сгенерированные исключения Exception
11         // перехватываются в следующем блоке catch
12         try
13         {
14             Method1();
15         } // Конец try
16         catch ( Exception exceptionParameter )
17         {
18             // Вывод строкового представления Exception с последующим
19             // выводом свойств Message, StackTrace и InnerException
20             Console.WriteLine( "exceptionParameter.ToString: \n{0}\n",
21                 exceptionParameter );
22             Console.WriteLine( "exceptionParameter.Message: \n{0}\n",
23                 exceptionParameter.Message );
24             Console.WriteLine( "exceptionParameter.StackTrace: \n{0}\n",
25                 exceptionParameter.StackTrace );
26             Console.WriteLine( "exceptionParameter.InnerException: \n{0}\n",
27                 exceptionParameter.InnerException );
28         } // Конец catch
29     } // Конец метода Main
30
31     // Вызывает Method2
32     static void Method1()
33     {
34         Method2();
35     } // Конец метода Method1
36
37     // Вызывает Method3
38     static void Method2()
39     {
40         Method3();
41     } // Конец метода Method2
42
43     // Выдает исключение Exception с InnerException
44     static void Method3()
45     {
46         // Попытка преобразования string в int
47         try
48         {
49             Convert.ToInt32( "Not an integer" );
50         } // Конец try
51         catch ( FormatException formatExceptionParameter )
52         {
53             // Упаковка FormatException в новом объекте Exception
54             throw new Exception( "Exception occurred in Method3",
55                 formatExceptionParameter );
56         } // Конец catch
57     } // Конец метода Method3
58 } // Конец класса Properties

```

**exceptionParameter.ToString:**

**Ил. 13.5.** Раскрутка стека и свойства класса Exception (продолжение ↗)

```

System.Exception: Exception occurred in Method3 --->
System.FormatException: Input string was not in a correct format.
at System.Number.StringToNumber(String str, NumberStyles options,
    NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
at System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
at Properties.Method3() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 49
--- End of inner exception stack trace ---
at Properties.Method3() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 54
at Properties.Method2() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 40
at Properties.Method1() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 34
at Properties.Main() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 14

exceptionParameter.Message:
Exception occurred in Method3

exceptionParameter.StackTrace:
at Properties.Method3() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 54
at Properties.Method2() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 40
at Properties.Method1() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 34
at Properties.Main() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 14

exceptionParameter.InnerException:
System.FormatException: Input string was not in a correct format.
at System.Number.StringToNumber(String str, NumberStyles options,
    NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
at System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
at Properties.Method3() in C:\examples\ch13\Fig13_05\Properties\
    Properties\Properties.cs: line 49

```

### Ил. 13.5. Раскрытие стека и свойства класса Exception (окончание)

Выполнение программы начинается с метода Main, который становится первым методом в стеке вызовов. В строке 14 блока try метода Main вызывается метод Method1 (строки 32–35), который становится вторым методом в стеке. Если Method1 выдает исключение, блок catch в строках 16–28 обрабатывает исключение и выводит информацию о возникшем исключении. Строка 34 Method1 вызывает метод Method2 (строки 38–41), который становится третьим методом в стеке. Затем строка 40 Method2 вызывает метод Method3 (строки 44–57), который становится четвертым методом в стеке.

К этому моменту стек вызова (сверху вниз) выглядит так:

```
Method3  
Method2  
Method1  
Main
```

Метод, вызванный последним (`Method3`), находится на вершине стека; первый вызванный метод (`Main`) находится в нижней позиции стека. Команда `try` (строки 47–56) в `Method3` вызывает метод `Convert.ToInt32` (строка 49), который пытается преобразовать `string` в `int`. В этой точке `Convert.ToInt32` становится пятым и последним методом в стеке вызовов.

### Выдача исключения `Exception` с `InnerException`

Так как аргумент `Convert.ToInt32` не соответствует формату `int`, строка 49 выдает исключение `FormatException`, которое перехватывается в строке 51 метода `Method3`. Исключение завершает вызов `Convert.ToInt32`, поэтому метод выводится из стека вызовов. Блок `catch` метода `Method3` создает и выдает объект `Exception`. Первый аргумент конструктора `Exception` содержит пользовательскую строку сообщения об ошибке для нашего примера: «Exception occurred in Method3». Второй аргумент содержит `InnerException` — объект перехваченного исключения `FormatException`. Свойство `StackTrace` нового объекта исключения отражает состояние точки, в которой было выдано исключение (строки 54–55). Теперь метод `Method3` завершается, потому что исключение, выданное в блоке `catch`, не было перехвачено в теле метода. Управление возвращается команде, вызвавшей `Method3` в предыдущем методе стека вызовов (`Method2`). При этом `Method3` выводится из стека вызовов.

Когда управление возвращается в строку 40 метода `Method2`, CLR определяет, что строка 40 не находится в блоке `try`. Следовательно, исключение не может быть перехвачено в `Method2`, и `Method2` завершается. В результате `Method2` выводится из стека вызовов, а управление возвращается в строку 34 метода `Method1`.

И снова строка 34 не находится в блоке `try`, поэтому `Method1` не может перехватить исключение. Метод завершается и выводится из стека вызовов, а управление возвращается в строку 14 метода `Main`, которая находится в блоке `try`. Происходит выход из блока `try` в `Main`, а блок `catch` (строки 16–28) перехватывает исключение. Блок `catch` использует свойства `Message`, `StackTrace` и `InnerException` для создания вывода. Раскрутка стека продолжается до того момента, когда блок `catch` перехватит исключение или программа завершится.

### Вывод информации об исключении

Первая часть вывода (которую мы переформатировали для удобства чтения) на ил. 13.5 содержит строковое представление исключения, возвращаемое неявным вызовом метода `ToString`. Строка начинается с имени класса исключения, за которым следует значение свойства `Message`. Следующие четыре элемента представляют содержимое трассировки стека объекта `InnerException`.

В оставшейся части вывода приводится содержимое `StackTrace` для исключения, выданного в методе `Method3`. `StackTrace` представляет состояние стека вызовов

в точке выдачи исключения (а не в точке, в которой исключение будет в конечном итоге перехвачено). Каждая строка `StackTrace`, начинающаяся с «at», представляет метод в стеке вызовов. В ней указывается метод, в котором произошло исключение, файл, в котором находится метод, и номер строки для точки выдачи исключения в файле. Информация внутреннего исключения включает трассировку стека внутреннего исключения.



#### КАК ИЗБЕЖАТЬ ОШИБОК 13.4

При перехвате и повторной выдаче исключения включайте дополнительную отладочную информацию в перезапущенное исключение. Для этого создайте объект `Exception` с более подробной отладочной информацией и передайте исходное перехваченное исключение конструктору нового объекта исключения для инициализации свойства `InnerException`.

Следующая часть вывода (две строки) просто выводит значение свойства `Message` (`Exception occurred in Method3`) исключения, выданного в `Method3`.

В третьей части выводится свойство `StackTrace` исключения, выданного в `Method3`. Свойство `StackTrace` содержит трассировку стека начиная со строки 54 метода `Method3` — точки, в которой был создан и выдан объект `Exception`. Трассировка стека *всегда* начинается с точки выдачи исключения.

Наконец, в последней части выводится строковое представление свойства `InnerException`, включающее пространство имен и имя класса объекта исключения, а также его свойства `Message` и `StackTrace`.

## 13.8. Пользовательские классы исключений

Нередко для обозначения исключений, происходящих в программах, можно использовать готовые классы исключений из `.NET Framework Class Library`. Однако в некоторых случаях лучше создать новый класс исключения для конкретных проблем, возникающих в ваших программах. Пользовательские классы исключений являются производными (прямо или косвенно) от класса `Exception` из пространства имен `System`. Если ваш код выдает исключения, они должны быть хорошо документированы, чтобы другие разработчики, использующие ваш код, знали, как их обрабатывать.



#### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 13.1

Связывая класс исключений для каждой разновидности проблем, возникающих в вашей программе, вы делаете программу более понятной.



#### АРХИТЕКТУРНОЕ РЕШЕНИЕ 13.3

Прежде чем определять пользовательский класс исключения, просмотрите существующие исключения в `.NET Framework Class Library` — возможно, вам удастся найти среди них подходящий тип.



## Класс `NegativeNumberException`

На ил. 13.6–13.7 приведен пользовательский класс исключений. Класс `NegativeNumberException` (см. ил. 13.6) представляет исключения, возникающие при выполнении недопустимой операции с отрицательным числом (например, извлечения квадратного корня).

```

1 // Ил. 13.6: NegativeNumberException.cs
2 // NegativeNumberException представляет исключения, возникающие
3 // при выполнении недопустимых операций с отрицательными числами.
4 using System;
5
6 class NegativeNumberException : Exception
7 {
8     // Конструктор по умолчанию
9     public NegativeNumberException()
10         : base( "Illegal operation for a negative number" )
11     {
12         // Пустое тело
13     } // Конец конструктора по умолчанию
14
15     // Конструктор для настройки сообщения об ошибке
16     public NegativeNumberException( string messageValue )
17         : base( messageValue )
18     {
19         // Пустое тело
20     } // Конец конструктора с одним аргументом
21
22     // Конструктор для настройки сообщения об ошибке
23     // и определения объекта InnerException
24     public NegativeNumberException( string messageValue,
25                                     Exception inner )
26         : base( messageValue, inner )
27     {
28         // Пустое тело
29     } // Конец конструктора с двумя аргументами
30 } // Конец класса NegativeNumberException

```

**Ил. 13.6.** Класс `NegativeNumberException` представляет исключения, возникающие при выполнении недопустимых операций с отрицательными числами

Согласно документации Microsoft «Best Practices for Handling Exceptions» ([bit.ly/ExceptionsBestPractices](http://bit.ly/ExceptionsBestPractices)), типичное пользовательское исключение должно расширять класс `Exception`, его имя должно начинаться с «Exception», и исключение должно определять три конструктора: конструктор без параметров; конструктор со строковым аргументом (сообщение об ошибке); и конструктор, получающий строку и аргумент `Exception` (сообщение об ошибке и объект внутреннего исключения). Определение этих трех конструкторов делает ваш класс исключений более гибким, чтобы другие программисты могли легко использовать и расширять его. Исключения `NegativeNumberExceptions` обычно возникают при выполнении арифметических операций, поэтому класс `NegativeNumberException` логично определить производным от класса `ArithmeticException`. Однако класс `ArithmeticException` является производным

от класса `SystemException` — категории исключений, выдаваемых CLR. Согласно рекомендациям Microsoft, классы пользовательских исключений должны быть производными от `Exception`, а не от `SystemException`. В этом конкретном примере можно было бы использовать встроенный класс `ArgumentException`, рекомендуемый для недействительных значений аргументов. Мы создаем собственный тип исключения только для демонстрации.

### Тестирование класса `NegativeNumberException`

Класс `SquareRootTest` (см. ил. 13.7) демонстрирует использование пользовательского класса исключений. Приложение предлагает пользователю ввести число, а затем вызывает метод `SquareRoot` (строки 40–48) для вычисления квадратного корня. `SquareRoot` вызывает метод `Sqrt` класса `Math`, получающий аргумент `double`. Обычно при отрицательном значении аргумента метод `Sqrt` возвращает `NaN`. В нашей программе мы хотели бы предотвратить попытку вычисления квадратного корня из отрицательного числа. Если числовое значение, введенное пользователем, отрицательно, метод `SquareRoot` выдает `NegativeNumberException` (строки 44–45). В противном случае `SquareRoot` вызывает метод `Sqrt` класса `Math` для вычисления квадратного корня (строка 47).

При вводе значения команда `try` (строки 14–34) пытается вызвать `SquareRoot` для значения, введенного пользователем. Если введенные пользователем данные не являются числом, происходит исключение `FormatException`, которое обрабатывается блоком `catch` в строках 25–29. Если пользователь ввел отрицательное число, то метод `SquareRoot` выдает исключение `NegativeNumberException` (строки 44–45); блок `catch` в строках 30–34 перехватывает и обрабатывает этот тип исключений.

```
1 // Ил. 13.7: SquareRootTest.cs
2 // Пользовательский класс исключений.
3 using System;
4
5 class SquareRootTest
6 {
7     static void Main( string[] args )
8     {
9         bool continueLoop = true;
10
11         do
12         {
13             // Перехват исключений NegativeNumberException
14             try
15             {
16                 Console.Write(
17                     "Enter a value to calculate the square root of: " );
18                 double inputValue = Convert.ToDouble( Console.ReadLine() );
19                 double result = SquareRoot( inputValue );
20
21                 Console.WriteLine( "The square root of {0} is {1:F6}\n",
22                     inputValue, result );
23                 continueLoop = false;
```

**Ил. 13.7.** Тестирование пользовательского класса исключений (продолжение )

```

24         } // Конец try
25     catch ( FormatException formatException )
26     {
27         Console.WriteLine( "\n" + formatException.Message );
28         Console.WriteLine( "Please enter a double value.\n" );
29     } // Конец catch
30     catch ( NegativeNumberException negativeNumberException )
31     {
32         Console.WriteLine( "\n" + negativeNumberException.Message );
33         Console.WriteLine( "Please enter a non-negative value.\n" );
34     } // Конец catch
35     } while ( continueLoop );
36 } // Конец Main
37
38 // Вычисление квадратного корня из параметра; если параметр
39 // отрицателен, выдается исключение NegativeNumberException
40 public static double SquareRoot( double value )
41 {
42     // Если операнд отрицателен, выдается NegativeNumberException
43     if ( value < 0 )
44         throw new NegativeNumberException(
45             "Square root of negative number not permitted" );
46     else
47         return Math.Sqrt( value ); // Вычисление квадратного корня
48 } // Конец метода SquareRoot
49 } // Конец класса SquareRootTest

```

```

Enter a value to calculate the square root of: 30
The square root of 30 is 5.477226

```

```

Enter a value to calculate the square root of: hello

```

```

Input string was not in a correct format.
Please enter a double value.

```

```

Enter a value to calculate the square root of: 25
The square root of 25 is 5.000000

```

```

Enter a value to calculate the square root of: -2

```

```

Square root of negative number not permitted
Please enter a non-negative value.

```

```

Enter a value to calculate the square root of: 2
The square root of 2 is 1.414214

```

**Ил. 13.7.** Тестирование пользовательского класса исключений (окончание)

## 13.9. Итоги

В этой главе вы узнали, как использовать исключения для обработки ошибок в приложении. Мы показали, что обработка исключений позволяет отделить код

обработки ошибок от «основной линии» выполнения программы. Вы узнали, что код, который может выдавать исключения, заключается в блок `try`, а блоки `catch` обрабатывают возможные исключения. В модели с завершением, применяемой в C#, после обработки исключения управление не возвращается в точку выдачи. Мы рассмотрели несколько важных классов иерархии .NET `Exception`, включая классы `Exception` (на основе которого создаются пользовательские классы исключений) и `SystemException`. Также вы узнали, как использовать блоки `finally` для освобождения ресурсов независимо от того, возникали исключения или нет, и как инициировать и перезапускать исключения командой `throw`. Мы рассмотрели команду `using`, используемую для автоматизации процесса освобождения ресурсов, свойства `Exception` для получения дополнительной информации об исключениях: `Message`, `StackTrace`, `InnerException` и метод `ToString`. В последней части главы описаны принципы создания пользовательских классов исключений.

# 14 Графический интерфейс и Windows Forms: часть 1

## 14.1. Введение

Графический интерфейс (GUI, Graphic User Interface) обеспечивает визуальное взаимодействие пользователя с программой, определяет ее оформление и поведение.

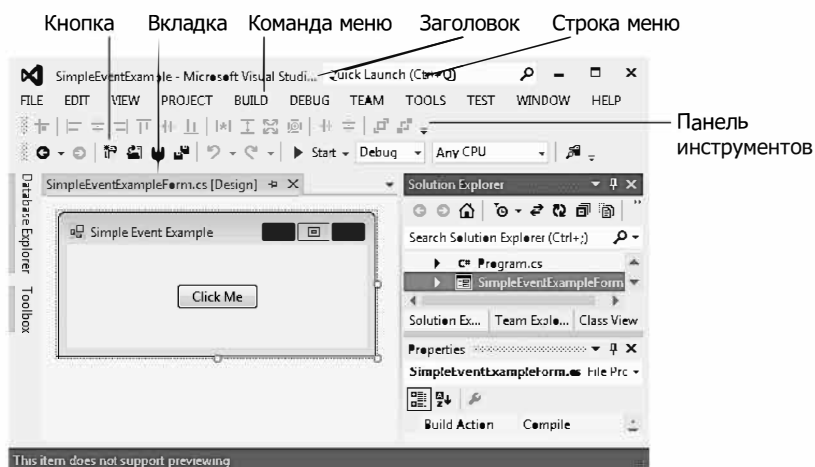


### КРАСИВО И УДОБНО 14.1

Соблюдение правил построения пользовательского интерфейса ускоряет освоение новых приложений, обладающих знакомым видом и поведением.

На ил. 14.1 изображено окно Visual Studio Express с различными элементами управления графического интерфейса. В верхней части находится строка меню с командами FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, WINDOW и HELP. Ниже расположена панель инструментов с кнопками, каждая из которых выполняет определенную операцию — например, создание нового или открытие существующего проекта. Еще ниже находится *вкладка* (tab), представляющая текущий открытый файл, — *представление с вкладками* дает возможность пользователю переключаться между открытыми файлами. Эти элементы управления образуют удобный интерфейс, через который вы общаетесь с IDE.

Графические интерфейсы строятся из *элементов управления* (которые также иногда называются *компонентами* и *виджетами*) — объектов, которые выводят информацию на экран или обеспечивают взаимодействие пользователя с приложением при помощи мыши, клавиатуры или других средств ввода (например, голосовых команд). Некоторые часто используемые элементы управления перечислены на ил. 14.2 — в этой главе и в главе 15 все они будут рассмотрены более подробно. В главе 15 описаны возможности и свойства других элементов графического интерфейса.



Ил. 14.1. Графический интерфейс окна Visual Studio 2012 for Windows Desktop

Элемент управления	Описание
Label	Надпись — используется для вывода графики или не редактируемого текста
Textbox	Текстовое поле — предоставляет возможность ввода с клавиатуры. Также может использоваться для вывода редактируемого или не редактируемого текста
Button	Кнопка — инициирует событие при нажатии
CheckBox	Флажок — определяет параметр, который может находиться в одном из двух состояний (установлен или снят)
ComboBox	Поле со списком — содержит раскрывающийся список вариантов, из которого пользователь может выбрать нужный вариант (щелчком на строке списка или вводом текста с клавиатуры)
ListBox	Список — предоставляет перечень, в котором пользователь может выбрать один или несколько вариантов
Panel	Панель — контейнер для размещения других элементов
NumericUpDown	Поле со счетчиком — позволяет выбрать числовое значение из диапазона

Ил. 14.2. Часто используемые элементы управления графического интерфейса

## 14.2. Windows Forms

Windows Forms — одна из библиотек, используемых для создания графических интерфейсов (в последующих главах вы также узнаете о библиотеках Windows 8 UI и Windows Presentation Foundation). *Форма* (Form) представляет собой графический элемент, который отображается на рабочем столе компьютера; это может быть

диалоговое окно, окно или окно MDI (см. главу 15). *Компонентом* (component) называется экземпляр класса, реализующего интерфейс `IComponent`, который определяет поведение компонента (например, способ его загрузки). *Элемент управления* (control) — такой, как `Button` или `Label`, — имеет графическое представление во время выполнения. Некоторые компоненты не имеют графического представления (например, класс `Timer` из пространства имен `System.Windows.Forms` — см. главу 15). Такие компоненты не видны во время выполнения.

На ил. 14.3 изображены компоненты и элементы управления Windows Forms с панели элементов (toolbox) C#. Элементы управления и компоненты разделены на категории по функциональности. Выбор категории `All Windows Forms` в верхней части панели элементов позволяет просмотреть все элементы управления и компоненты с других вкладок в одном списке (как на ил. 14.3). В этой и следующей главах будут рассмотрены многие из этих элементов управления и компонентов. Чтобы добавить элемент управления или компонент на форму, выделите его на панели элементов и перетащите мышью. Чтобы снять выделение с элемента управления или компонента, выберите значок `Pointer` на панели элементов (стрелка в начале списка). В режиме `Pointer` вы не сможете случайно добавить на форму новый элемент управления.



Ил. 14.3. Компоненты и элементы управления Windows Forms

Когда на экране находятся несколько окон, активное окно находится на переднем плане, а его заголовок выделен цветом. Чтобы окно стало активным, пользователь

щелкает в его границах. Говорят, что активное окно «имеет фокус ввода». Например, в Visual Studio активным окном становится панель элементов, когда вы выбираете один из ее значков, или окно свойств, когда вы редактируете свойства элемента управления.

Форма (Form) является контейнером для элементов управления и компонентов. При перетаскивании значков с панели элементов на форму Visual Studio генерирует код создания объекта и инициализации его основных свойств. Этот код обновляется при изменении свойств элемента управления или компонента в IDE. Удаление элемента управления или компонента с формы приводит к удалению соответствующего сгенерированного кода. IDE хранит сгенерированный код в отдельном файле с использованием *частичных классов* — классов, разбитых на несколько файлов и собираемых в один класс компилятором. Вы можете написать этот код самостоятельно, но гораздо проще поручить «черную работу» Visual Studio. Основные концепции визуального программирования изложены в главе 2. В этой и следующей главе мы используем визуальное программирование для построения более серьезных интерфейсов.

Все элементы управления и компоненты, представленные в этой главе, находятся в пространстве имен `System.Windows.Forms`. Чтобы создать приложение Windows Forms, разработчик обычно создает объект формы `Windows (Form)`, задает его свойства, добавляет на форму элементы, задает их свойства и реализует обработчики событий (методы), которые реагируют на события, генерируемые элементами. На ил. 14.4 перечислены часто используемые свойства Form, а также часто используемые методы и событие.

При создании элементов управления и обработчиков событий Visual Studio генерирует большую часть GUI-кода. В визуальном программировании IDE обеспечивает сопровождение этого кода, а разработчик пишет обработчики событий для определения действий, которые должны выполняться программой при наступлении того или иного события.

Свойства, методы и событие Form	Описание
Часто используемые свойства	
AcceptButton	Кнопка по умолчанию, которая срабатывает при нажатии клавиши Enter
AutoScroll	Логический признак (false по умолчанию) отображения полос прокрутки при необходимости
CancelButton	Кнопка, которая срабатывает при нажатии клавиши Escape
FormBorderStyle	Стиль границы формы (Sizable по умолчанию)
Font	Шрифт текста, выводимого на форме; также определяет шрифт по умолчанию для элементов, добавляемых на форму
Text	Текст в заголовке формы

**Ил. 14.4.** Часто используемые свойства, методы и событие Form (продолжение ↗)



Свойства, методы и событие Form	Описание
Часто используемые методы	
Close	Закрывает форму и освобождает все ресурсы (например, память, задействованную для содержимого формы). Закрытую форму нельзя открыть заново
Hide	Скрывает форму, но не уничтожает ее и не освобождает ее ресурсы
Show	Отображает скрытую форму
Часто используемое событие	
Load	Происходит перед отображением формы. События и обработка событий рассматриваются в следующем разделе

**Ил. 14.4.** Часто используемые свойства, методы и событие Form (окончание)

## 14.3. Обработка событий

Обычно пользователь взаимодействует с графическим интерфейсом приложения для выполнения операций, которые должны выполняться приложением. Например, когда вы пишете сообщение в почтовом клиенте, нажатие кнопки **Send** приказывает приложению отправить сообщение по заданным адресам электронной почты. Графический интерфейс *управляется событиями*. Когда пользователь взаимодействует с компонентом, событие заставляет программу выполнить операцию. К распространенным событиям, по которым приложение выполняет операцию, относятся нажатия кнопок, ввод текста в текстовых полях, выбор варианта из меню, закрытие окна и перемещение мыши. Со всеми элементами управления графического интерфейса связываются события; объекты других типов также могут связываться с событиями. Метод, выполняющий операцию в ответ на событие, называется *обработчиком события* (event handler), а общий процесс реагирования на события называется *обработкой событий*.

### 14.3.1. Простой графический интерфейс

Форма приложения на ил. 14.5 содержит кнопку (Button), нажатие которой открывает окно сообщения (MessageBox). Обратите внимание на объявление пространства имен в строке 6: оно вставляется для каждого созданного вами класса. В предыдущих примерах мы удаляли эти объявления, потому что они были лишними. Пространства имен предназначены для группировки взаимосвязанных классов. Полное имя каждого класса в действительности состоит из пространства имен, точки (.) и имени класса. Такая запись называется *полным именем класса*. В этом приложении можно использовать простое имя (без уточнения пространством

имен, то есть `SimpleEventExample`). Область действия каждого имени класса ограничивается пространством имен, в котором оно определено. Если вы захотите повторно использовать этот класс в другом приложении, используйте полное имя или включите директиву `using`, чтобы на класс можно было ссылаться по простому имени.

Пространства имен будут использоваться в главах 15 и 19. Если другое пространство имен также содержит одноименный класс, полное имя класса позволит различить классы в приложении и предотвратит *конфликты имен*.

```
1 // Ил. 14.5: SimpleEventExampleForm.cs
2 // Простой пример обработки событий.
3 using System;
4 using System.Windows.Forms;
5
6 namespace SimpleEventExample
7 {
8     // Форма с простым обработчиком события
9     public partial class SimpleEventExampleForm : Form
10    {
11        // Конструктор по умолчанию
12        public SimpleEventExampleForm()
13        {
14            InitializeComponent();
15        } // Конец конструктора
16
17        // Обработка события click кнопки clickButton
18        private void clickButton_Click( object sender, EventArgs e )
19        {
20            MessageBox.Show( "Button was clicked." );
21        } // Конец метода clickButton_Click
22    } // Конец класса SimpleEventExampleForm
23 } // Конец пространства имен SimpleEventExample
```



**Ил. 14.5.** Простой пример обработки события

### Переименование файла `Form1.cs`

Создайте форму и разместите на ней кнопку (см. главу 2). Начните с создания нового приложения Windows Forms, затем переименуйте файл `Form1.cs` в `SimpleEventExampleForm.cs` в окне `Solution Explorer`. Щелкните на форме в конструкторе форм, в окне свойств задайте свойству `Text` формы значение `"SimpleEventExample"`. Выберите свойство `Font` в окне свойств, щелкните на кнопке с многоточием (...) и выберите в открывшемся диалоговом окне значение `Segoe UI, 9pt`.

### Добавление кнопки на форму

Перетащите кнопку с панели элементов на форму. В окне свойств кнопки задайте свойству (Name) значение `clickButton`, а свойству `Text` — значение `ClickMe`. Обратите внимание: мы следуем схеме, в соответствии с которой имена переменных, создаваемых для элемента управления, завершаются обозначением типа (как, например, суффикс «Button» в имени переменной `clickButton`).

### Добавление обработчика события Click для кнопки

Когда пользователь щелкает на кнопке, наше приложение должно вывести окно сообщения (`MessageBox`). Для этого следует назначить обработчик события `Click` объекта `Button`. Если сделать двойной щелчок на кнопке, в программный код включается следующий пустой обработчик события:

```
private void clickButton_Click( object sender, EventArgs e )
{
}
```

IDE строит имена методов обработчиков событий по схеме `имяОбъекта_имяСобытия` (например, `clickButton_Click`). Обработчик `clickButton_Click` обрабатывает событие щелчка (`Click`) на элементе управления `clickButton`.

### Параметры обработчиков событий

Каждый обработчик события при вызове получает два параметра. Первый — обычно с именем `sender` — содержит ссылку на объект, сгенерировавший событие. Второй параметр содержит ссылку на объект `EventArgs` (или объект класса, производного от `EventArgs`), обычно с именем `e`. В этом объекте содержится дополнительная информация о возникшем событии. `EventArgs` — базовый класс для всех классов, представляющих информацию о событии.

### Вывод MessageBox

Чтобы вывести на экран объект `MessageBox` в ответ на событие, включите команду `MessageBox.Show( "Button was clicked." );`

в тело обработчика события. Полученный обработчик события находится в строках 18–21 на ил. 14.5. Если запустить приложение и щелкнуть на кнопке, на экране появляется окно `MessageBox` с текстом "Button was clicked".

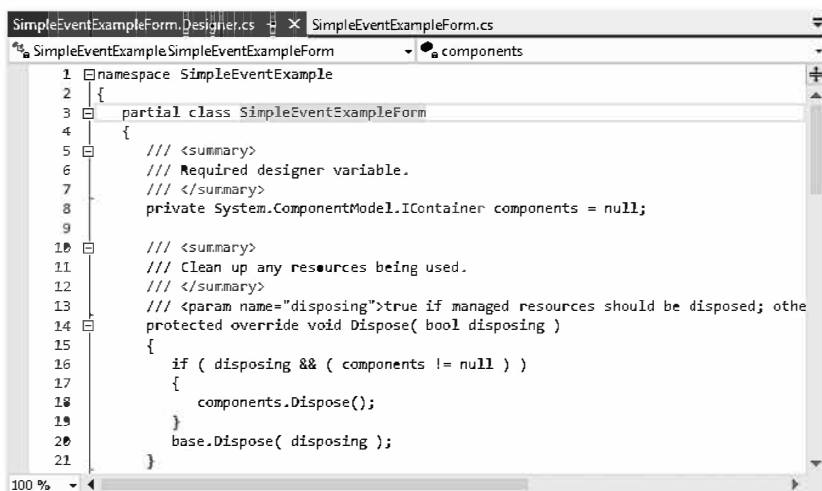
## 14.3.2. Автоматически генерируемый код графического интерфейса

Visual Studio помещает автоматически сгенерированный код графического интерфейса в файл `Designer.cs` формы (`SimpleEventExampleForm.Designer.cs` в данном примере). Чтобы открыть этот файл, раскройте в окне `Solution Explorer` узел файла, с которым вы работаете в настоящее время (`SimpleEventExampleForm.cs`), и сделайте двойной щелчок на имени файла, завершающемся `Designer.cs`. На ил. 14.6 и 14.7 показано

содержимое этого файла. IDE по умолчанию сворачивает код в строках 23–57 на ил. 14.7 — щелкните на значке + рядом со строкой 23, чтобы развернуть его (щелчок на значке — снова сворачивает код).

Теперь, после подробного рассмотрения классов и объектов, вам будет намного проще понять этот код. Так как код создается и поддерживается Visual Studio, обычно вам не приходится его просматривать. Более того, для построения приложений GUI понимать большую часть приведенного кода вообще не обязательно. Тем не менее мы присмотримся к нему поближе, чтобы вы поняли, как работают приложения с графическим интерфейсом.

Автоматически сгенерированный код, определяющий графический интерфейс, на самом деле является частью класса `Form` — в данном случае `SimpleEventExampleForm`. В строке 3 на ил. 14.6 (и в строке 9 на ил. 14.5) используется модификатор `partial`, который позволяет разбить этот класс на несколько файлов, включая файл с автоматически сгенерированным кодом и файлы, в которых хранится написанный вами код. В строке 59 на ил. 14.7 объявляется объект `clickButton`, созданный в режиме конструктора. Он объявляется как переменная экземпляра класса `SimpleEventExampleForm`. По умолчанию все объявления переменных экземпляров, созданные в окне конструктора C#, обладают модификатором доступа `private`. Код также включает метод `Dispose` для освобождения ресурсов (см. ил. 14.6, строки 14–21) и метод `InitializeComponent` (см. ил. 14.7, строки 29–55), код которого создает `Button`, а затем задает некоторые из свойств `Button` и `Form`. Значения свойств соответствуют значениям, заданным в окне свойств для каждого элемента управления. Visual Studio добавляет комментарии в сгенерированный код (см. строки 33–35). Строка 42 была сгенерирована при создании обработчика события `Click` кнопки.



Ил. 14.6. Первая половина файла с кодом, сгенерированным Visual Studio



**Ил. 14.7.** Вторая половина файла с кодом, сгенерированным Visual Studio

Метод `InitializeComponent` вызывается при создании формы и задает такие свойства, как заголовок формы, ее размер, размеры и текст элементов управления. Visual Studio также использует код этого метода для создания графического интерфейса, который вы видите в режиме конструктора. Изменение кода `InitializeComponent` может помешать правильному отображению графического интерфейса в Visual Studio.



#### КАК ИЗБЕЖАТЬ ОШИБОК 14.1

Код, сгенерированный при построении графического интерфейса в режиме конструктора, не предназначен для прямого редактирования (поэтому он и находится в отдельном файле). Модификация этого кода может привести к искажению графического интерфейса в режиме конструктора и нарушению работоспособности приложения. В режиме конструктора свойства элементов управления должны изменяться только в окне свойств.

### 14.3.3. Делегаты и механизм обработки событий

Элемент управления, генерирующий событие, называется *отправителем события*. Метод обработки события (обработчик) реагирует на конкретное событие, генерируемое элементом управления. При возникновении события отправитель вызывает свой обработчик для выполнения операции (то есть обработки события).

Механизм обработки событий .NET позволяет разработчикам выбирать имена методов обработки событий. Однако при этом каждый метод обработки событий должен объявить правильные параметры для получения информации об обрабатываемом событии. Так как имена могут выбираться программистом, отправитель события (например, `Button`) не может заранее знать, какой метод отреагирует на его события. Следовательно, нужен механизм определения того, какой метод обрабатывает некоторое событие.

#### Делегаты

Обработчики событий связываются с событиями элементов управления при помощи специальных объектов, называемых *делегатами*. В объявлении типа делегата указывается сигнатура метода — в обработке событий сигнатура задает возвращаемый тип и аргументы обработчика события. Элементы графического интерфейса имеют заранее определенных делегатов, соответствующих каждому генерируемому событию. Например, делегат события `Click` объекта `Button` относится к типу `EventHandler` (пространство имен `System`). В электронной справочной документации этот тип объявляется следующим образом:

```
public delegate void EventHandler( object sender, EventArgs e );
```

Ключевое слово `delegate` используется для объявления типа делегата `EventHandler`, способного хранить ссылку на метод, возвращающий `void` и получающий два параметра — типа `object` (отправитель события) и типа `EventArgs`. Если сравнить объявление делегата с первой строкой `clickButton_Click` (см. илл. 14.5, строка 18), вы увидите, что этот обработчик события возвращает `void` и получает параметры, заданные делегатом `EventHandler`. Приведенное выше объявление создает весь класс за вас, а компилятор берет на себя все технические подробности.

#### Обозначение метода, который должен вызываться делегатом

Отправитель события вызывает объект делегата как метод. Так как каждый обработчик события объявляется как делегат, обработчик события может просто вызвать подходящего делегата при возникновении события — `Button` по щелчку вызывает делегата `EventHandler`, соответствующего его событию `Click`. Задача делегата — вызвать соответствующий метод. Чтобы вызывался `clickButton_Clickmethod`, Visual Studio присваивает `clickButton_Click` делегату `ClickEventHandler` объекта `Button`, как показано в строке 42 ил. 14.7. Этот код добавляется Visual Studio при двойном щелчке на элементе `Button` в режиме конструктора. Выражение

```
new System.EventHandler(this.clickButton_Click);
```

создает объект делегата `EventHandler` и инициализирует его методом `clickButton_Click`. В строке 42 оператор `+=` используется для добавления `EventHandler` к делегату `Click` объекта `Button`. Это означает, что метод `clickButton_Click` должен вызываться тогда, когда пользователь щелкнет на объекте `Button`. Оператор `+=` перегружается классом делегата, который создается компилятором.

### Групповые делегаты

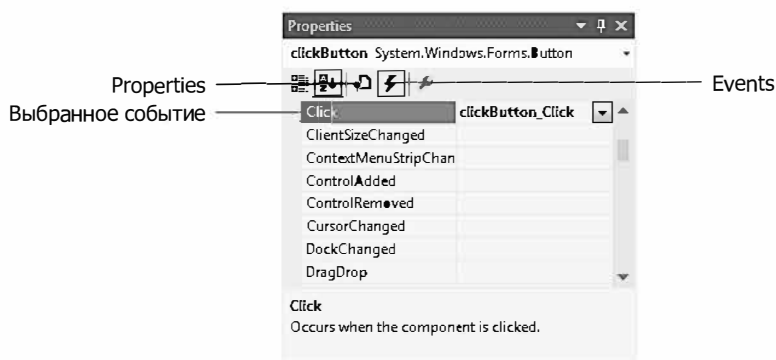
Разработчик может указать, что по событию должны вызываться несколько разных методов; для этого он добавляет других делегатов для события `Click` объекта `Button` командами, сходными с командой в строке 42 на ил. 14.7. Делегаты событий являются *групповыми* (multicast) — они представляют набор объектов делегатов с одинаковой сигнатурой. Групповые делегаты позволяют вызвать несколько методов в ответ на одно событие. При возникновении события отправитель вызывает *каждый* метод, на который ссылается групповой делегат. Делегаты событий являются производными от класса `MulticastDelegate`, производного от класса `Delegate` (оба класса принадлежат пространству имен `System`). В большинстве случаев для конкретного события элемента назначается только один обработчик события.

## 14.3.4. Другой способ создания обработчиков событий

В приложении с графическим интерфейсом на ил. 14.5 мы создали обработчик события двойным щелчком на элементе управления `Button`. Так для элемента управления создается обработчик для события по умолчанию (события, чаще всего используемого с этим элементом). Элементы могут генерировать много разных событий, каждое из которых может иметь собственный обработчик. Например, приложение также может предоставить обработчик события `MouseHover` элемента управления `Button`; это событие происходит, если задержать указатель мыши над кнопкой на короткий промежуток времени. Сейчас вы узнаете, как создать обработчик для события, не являющегося событием по умолчанию.

### Создание обработчиков событий в окне свойств

Дополнительные обработчики событий можно создать в окне свойств. Если выделить элемент управления на форме и щелкнуть на значке **Events** (значок с молнией на ил. 14.8) в окне свойств, в окне появляется список всех событий элемента управления. Двойной щелчок на имени события открывает существующий обработчик в редакторе или создает новый обработчик, если он еще не существует в коде. Также можно выделить событие, а затем выбрать в раскрывающемся списке справа существующий метод, который должен использоваться как обработчик выбранного события. В раскрывающемся списке перечислены методы класса `Form`, сигнатура которых позволяет использовать их в качестве обработчиков выбранного события. Чтобы вернуться к просмотру свойств элемента управления, щелкните на значке **Properties** (см. ил. 14.8).



**Ил. 14.8.** Просмотр событий элемента управления Button в окне свойств

Один метод может обрабатывать события нескольких элементов управления. Например, события `Click` трех элементов `Button` могут обрабатываться одним методом. Чтобы назначить обработчик для нескольких событий, выделите несколько элементов (перетаскиванием указателя мыши или щелчками с нажатой клавишей `Shift`) и выберите один метод на вкладке `Events` в окне свойств. Если при этом будет создан новый обработчик события, переименуйте его соответствующим образом. Также можно выделить каждый элемент управления по отдельности и назначить один метод обработчиком для каждого события.

### 14.3.5. Поиск информации о событиях

За информацией о событиях, выдаваемых каждым элементом управления, обращайтесь к документации Visual Studio. Выберите элемент управления в IDE и нажмите клавишу `F1`, чтобы вызвать электронную справку по элементу (ил. 14.9). На отображаемой веб-странице содержится основная информация о классе элемента управления. В левом столбце страницы находятся ссылки на дополнительную информацию о классе (конструктор, методы, свойства и события `Button`; содержимое списка зависит от класса). По каждой ссылке выводится информация о подмножестве членов класса. Щелкните на ссылке для получения списка событий, поддерживаемых элементом управления (`Button Events` в данном случае).

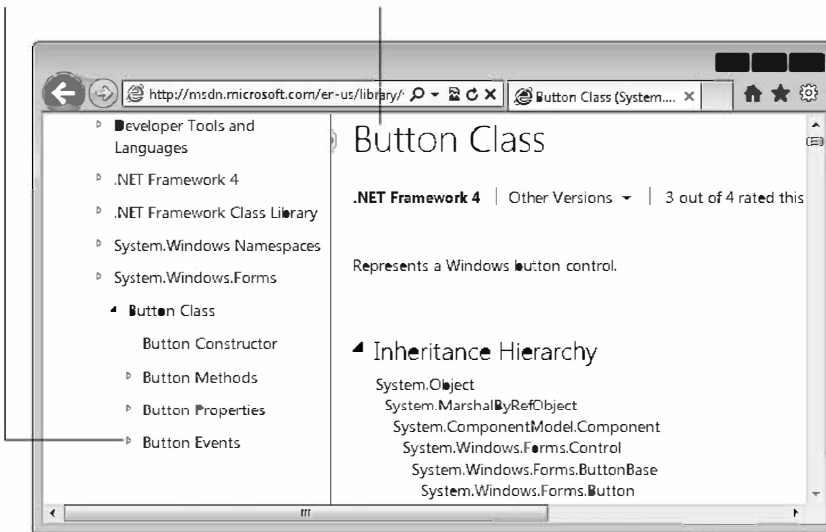
Щелкните на имени события, чтобы просмотреть его описание и примеры использования. Мы выбрали событие `Click` для вывода информации на ил. 14.10. Событие `Click` является членом класса `Control`, косвенного базового класса `Button`. В разделе `Remarks` страницы приводится расширенная информация о выбранном событии. Также можно воспользоваться окном `Object Browser` для поиска этой информации в пространстве имен `System.Windows.Forms`. В окне `Object Browser` выводятся только члены, изначально определенные в выбранном классе. Событие `Click` изначально определяется в классе `Control` и наследуется классом `Button`, поэтому для просмотра



описания события Click следует открыть в Object Browser документацию класса Control. Окно Object Browser более подробно рассматривается в разделе 10.12.

Ссылка на список поддерживаемых событий

Имя класса элемента управления

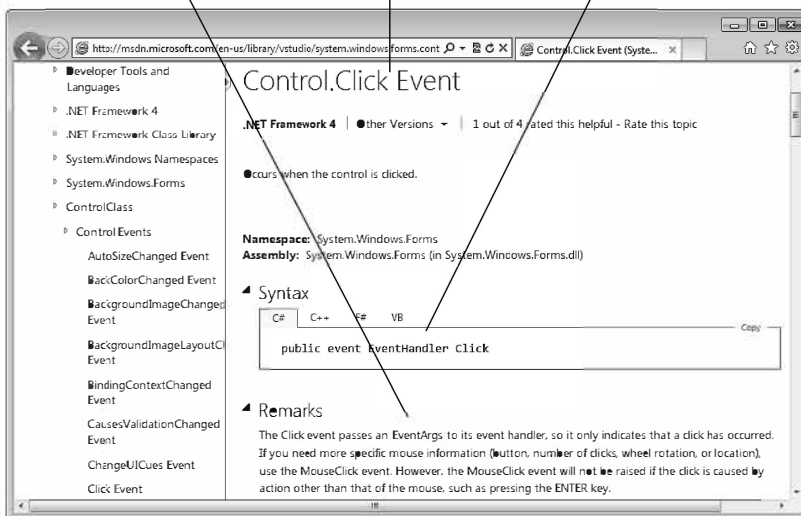


Ил. 14.9. Ссылка на список событий класса Button

Класс аргументов события

Имя события

Тип события



Ил. 14.10. Информация о событии Click

## 14.4. Свойства и макет элемента управления

В этом разделе представлены свойства, общие для многих элементов управления. Элементы управления наследуют от класса `Control` (пространство имен `System.Windows.Forms`). На ил. 14.11 перечислены некоторые свойства и методы класса `Control`, которые могут задаваться для многих элементов управления. Например, свойство `Text` определяет текст, выводимый на элементе управления. Местонахождение текста зависит от конкретного элемента: для формы текст выводится в заголовке, а для кнопки — непосредственно на поверхности.

Свойства и методы класса <code>Control</code>	Описание
Часто используемые свойства	
<code>BackColor</code>	Цвет фона
<code>BackgroundImage</code>	Фоновое изображение
<code>Enabled</code>	Признак доступности элемента управления (то есть возможности взаимодействия с ним со стороны пользователя). Недоступный (заблокированный) элемент обычно выделяется серым цветом
<code>Focused</code>	Признак наличия у элемента фокуса ввода (свойство доступно только во время выполнения)
<code>Font</code>	Шрифт, используемый для вывода текста элемента
<code>ForeColor</code>	Основной цвет элемента управления; обычно определяет цвет текста, определяемого свойством <code>Text</code>
<code>TabIndex</code>	Индекс элемента управления в последовательности перебора. При нажатии клавиши <code>Tab</code> фокус ввода передается между элементами в последовательности, которая определяется разработчиком
<code>TabStop</code>	Если свойство равно <code>true</code> , элемент управления может получить фокус при нажатии клавиши <code>Tab</code>
<code>Text</code>	Текст, связанный с элементом. Местонахождение и оформление текста зависит от типа элемента
<code>Visible</code>	Признак видимости элемента
Часто используемые методы	
<code>Hide</code>	Скрывает элемент управления (задает свойству <code>Visible</code> значение <code>false</code> )
<code>Select</code>	Передает элементу фокус
<code>Show</code>	Отображает элемент управления (задает свойству <code>Visible</code> значение <code>true</code> )

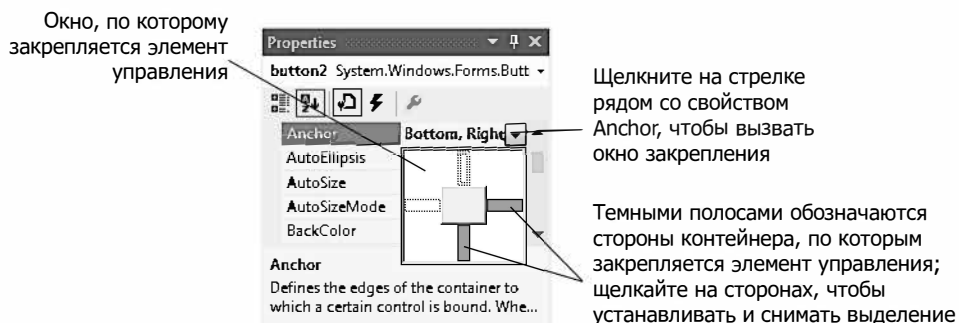
**Ил. 14.11.** Свойства и методы класса `Control`

Метод `Select` передает фокус элементу управления и делает его активным элементом управления. При нажатии клавиши `Tab` в выполняющемся приложении `Windows Forms` элементы управления получают фокус в порядке, задаваемом их свойством `TabIndex`. Свойство задается `Visual Studio` на основании порядка добавления элементов на форму, но его можно изменить командой `VIEW ► Tab Order`. Свойство `TabIndex` упрощает ввод информации во многих элементах управления — например, в наборе текстовых полей для имени, адреса, номера телефона и т. д. Пользователь заполняет одно поле, а затем быстро переходит к следующему элементу управления нажатием клавиши `Tab`.

Свойство `Enabled` определяет, может ли пользователь взаимодействовать с элементом управления для генерирования событий. Часто элементы управления блокируются из-за того, что некая возможность недоступна пользователю в конкретный момент времени. Например, текстовые редакторы часто блокируют команду вставки до тех пор, пока пользователь не скопирует фрагмент текста в буфер. Чаще всего текст заблокированного элемента выводится серым шрифтом (вместо черного). Элемент также можно скрыть без блокировки — для этого следует задать свойству `Visible` значение `false` или вызвать метод `hide`. В обоих случаях элемент управления продолжает существовать, но не отображается на форме.

### Закрепление и стыковка

Механизмы закрепления и стыковки применяются для определения макета, то есть размещения элементов управления внутри контейнера (например, `Form`). При *закреплении* (*anchoring*) элементы управления остаются на фиксированном расстоянии от сторон контейнера даже при изменении его размеров. Закрепление делает работу пользователя более удобной. Например, если пользователь привык, что элемент управления находится в определенном углу приложения, благодаря стыковке он *всегда* будет находиться в этом углу — даже если пользователь изменит размеры формы. *Стыковка* (*docking*) присоединяет элемент управления к контейнеру так, что он растягивается на всю сторону или заполняет всю площадь контейнера. Например, кнопка, пристыкованная к верхней стороне контейнера, будет растягиваться на всю верхнюю сторону контейнера независимо от его ширины. При изменении размеров окна (или другого родительского контейнера — например, `Panel`) закрепленные элементы управления перемещаются (и возможно, изменяются в размерах), чтобы расстояние от сторон, за которыми они закреплены, оставалось неизменным. По умолчанию большинство элементов закрепляется за левым верхним углом формы. Чтобы понаблюдать за эффектом закрепления элемента управления, создайте простое приложение `Windows Forms` с двумя кнопками. Закрепите одну кнопку за правой и нижней сторонами; для этого задайте свойство `Anchor`, как показано на ил. 14.12. Оставьте другому элементу управления вариант закрепления по умолчанию (`Top, Left`). Запустите приложение и увеличьте форму. Обратите внимание: кнопка, закрепленная за правым нижним углом, всегда сохраняет одинаковое расстояние от правого нижнего угла формы (ил. 14.13), но другая кнопка остается на исходном расстоянии от левого верхнего угла формы.

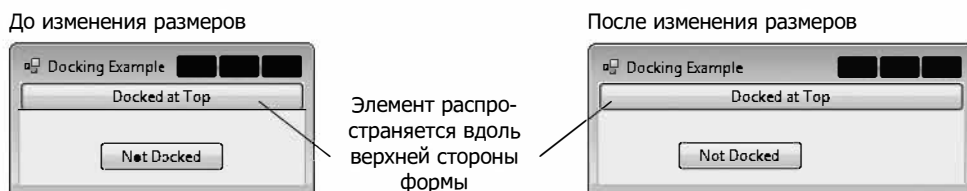


Ил. 14.12. Модификация свойства Anchor элемента управления



Ил. 14.13. Демонстрация использования Anchor

Иногда элемент управления должен распространяться на всю сторону формы даже при изменении размеров последней. Например, строка состояния должна оставаться у нижнего края формы. Стыковка позволяет элементу заполнить всю сторону (левую, правую, верхнюю или нижнюю) родительского контейнера или всю площадь контейнера. При изменении размеров родительского элемента пристыкованный элемент тоже изменяет размеры. На ил. 14.14 кнопка стыкуется у верхней стороны формы и автоматически распространяется на всю ее ширину. Форма имеет свойство `Padding`, определяющее расстояние между пристыкованными элементами управления и краями формы. Это свойство задает четверку значений (по одному для каждой стороны), по умолчанию каждое значение равно 0. Некоторые типичные свойства макета элементов управления представлены в таблице на ил. 14.15.



Ил. 14.14. Стыковка кнопки у верхнего края формы

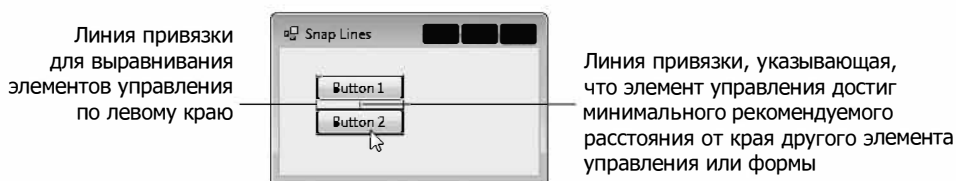
Свойства макета элемента управления	Описание
Anchor	Заставляет элемент управления оставаться на фиксированном расстоянии от сторон контейнера (даже при изменении размеров последнего)
Dock	Заставляет элемент управления распространяться вдоль стороны контейнера или заполнить оставшееся пространство контейнера
Padding	Задаёт расстояние между краями контейнера и пристыкованными элементами управления. Значение по умолчанию равно 0 (край элемента совмещается с краем формы)
Location	Задаёт местонахождение (набор координат) левого верхнего угла элемента управления относительно левого верхнего угла контейнера
Size	Задаёт размер элемента управления в пикселах в формате объекта Size со свойствами Width и Height
MinimumSize, MaximumSize	Обозначают соответственно минимальный и максимальный размер элемента управления

**Ил. 14.15.** Стыковка кнопки у верхнего края формы

Свойства `Anchor` и `Dock` объекта `Control` задаются относительно родительского контейнера `Control`, которым может быть форма или другой объект (например, `Panel` — см. раздел 14.6). Минимальный и максимальный размер формы (или другого элемента управления) может задаваться свойствами `MinimumSize` и `MaximumSize` соответственно. Оба свойства относятся к типу `Size`, свойства которого `Width` и `Height` задают ширину и высоту формы. Свойства `MinimumSize` и `MaximumSize` позволяют спроектировать макет графического интерфейса для заданного диапазона размеров. Пользователь не сможет сделать форму меньше размера, заданного свойством `MinimumSize`, или сделать ее больше размера, заданного свойством `MaximumSize`. Чтобы задать фиксированный размер формы (и запретить его изменение пользователем), задайте одинаковые значения минимального и максимального размера.

### Редактирование макета графического интерфейса в Visual Studio

Visual Studio помогает разработчику создавать макеты графического интерфейса. Во время перетаскивания элемента управления по форме появляются синие линии, упрощающие позиционирование элемента управления по отношению к другим элементам (ил. 14.16) и сторонам формы. Перетаскиваемый элемент как бы «прилипает» к сторонам других элементов. В Visual Studio также имеется меню **FORMAT** с командами изменения макета. Меню **FORMAT** отображается в IDE только при выделении одного или нескольких элементов в режиме конструктора. Если выделить несколько элементов управления, их можно выровнять командами подменю **FORMAT** ▶ **Align**. Меню **FORMAT** также позволяет изменить расстояние между элементами управления и выровнять элементы управления по центру формы.



Ил. 14.16. Линии выравнивания элементов управления

## 14.5. Надписи, текстовые поля и кнопки

*Надписи* используются для вывода текстовой информации (и возможно, графики) и определяются классом `Label`, производным от `Control`. Текст, выводимый в надписи, не может *напрямую* изменяться пользователем (но может быть изменен на программном уровне изменением свойства объекта `Label`). На ил. 14.17 перечислены часто используемые свойства `Label`.

Часто используемые свойства <code>Label</code>	Описание
<code>Font</code>	Шрифт текста надписи
<code>Text</code>	Текст надписи
<code>TextAlign</code>	Режим выравнивания текста в элементе управления — горизонтальный (по левому краю, по центру, по правому краю) и вертикальный (по верхнему краю, по центру, по нижнему краю). По умолчанию используется выравнивание по верхнему и левому краю

Ил. 14.17. Часто используемые свойства `Label`

*Текстовое поле* (класс `TextBox`) представляет собой область, в которой пользователь может вводить текст с клавиатуры (также возможен вывод текста программой). Если задать свойству текстового поля `UseSystemPasswordChar` значение `true`, текстовое поле работает в режиме ввода пароля и скрывает информацию, вводимую пользователем (вместо вводимых символов отображается символ-заменитель). Обе разновидности текстовых полей часто встречаются при входе в систему или на сайты — в обычном текстовом поле вводится имя пользователя, а в парольном текстовом поле — пароль. На ил. 14.18 перечислены часто используемые свойства и событие класса `TextBox`.

*Кнопка* (`Button`) — элемент управления, который нажимается пользователем для выполнения конкретной операции или выбора варианта. Как вы вскоре увидите, в программах могут использоваться разные типы кнопок, включая флажки и переключатели. Все классы кнопок наследуют от класса `ButtonBase` (пространство имен `System.Windows.Forms`), определяющего общие возможности кнопок. В этом разделе мы рассмотрим класс `Button`, который обычно используется для выдачи команд приложению. На ил. 14.19 перечислены часто используемые свойства и событие класса `Button`.

Часто используемые свойства и событие TextBox	Описание
Часто используемые свойства	
AcceptsReturn	Если свойство равно true в многострочном поле TextBox, нажатие Enter в TextBox создает новую строку. Если свойство равно false (значение по умолчанию), нажатие Enter приводит к тому же эффекту, что и нажатие кнопки по умолчанию на форме (кнопкой по умолчанию называется кнопка, назначенная свойству AcceptButton формы)
Multiline	Если свойство равно true, содержимое элемента TextBox может занимать несколько строк (по умолчанию используется значение false)
ReadOnly	Если свойство равно true, поле TextBox имеет серый фон, а его содержимое не может редактироваться (по умолчанию используется значение false)
ScrollBars	Для многострочных текстовых полей это свойство определяет режим отображения полос прокрутки: None (по умолчанию), Horizontal, Vertical или Both
Text	Текстовое содержимое TextBox
UseSystemPasswordChar	Если свойство равно true, используется режим ввода пароля (вместо вводимых символов отображается системный символ-заполнитель)
Часто используемое событие	
TextChanged	Генерируется при изменении текста в TextBox (то есть при добавлении или удалении символов). При двойном щелчке на элементе управления TextBox в режиме конструктора создается пустой обработчик этого события

**Ил. 14.18.** Часто используемые свойства и событие TextBox

Часто используемые свойства и событие Button	Описание
Часто используемые свойства	
Text	Текст, выводимый на поверхности кнопки
FlatStyle	Изменяет внешний вид кнопки: Flat (кнопка выводится без имитации объема), Rorip (кнопка выводится без имитации объема, пока пользователь не наведет на нее указатель мыши), Standard (трехмерная кнопка) или System (внешний вид кнопки определяется операционной системой)
Часто используемое событие	
Click	Генерируется при нажатии кнопки пользователем. При двойном щелчке на элементе управления Button в режиме конструктора создается пустой обработчик этого события

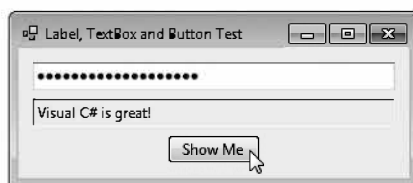
**Ил. 14.19.** Часто используемые свойства и событие Button

В программе на ил. 14.20 используются текстовое поле, кнопка и надпись. Пользователь вводит текст в режиме ввода пароля и нажимает кнопку, чтобы текст был выведен в надписи. Обычно введенный пароль отображаться не должен — режим ввода пароля нужен именно для того, чтобы скрыть текст, вводимый пользователем. При нажатии кнопки Show Me приложение читает текст, введенный в текстовом поле, и отображает его в надписи.

```

1  // Ил. 14.20: LabelTextBoxButtonTestForm.cs
2  // Использование TextBox, Label и Button,
3  // чтобы отображать скрытый текст при вводе пароля TextBox.
4  using System;
5  using System.Windows.Forms;
6
7  namespace LabelTextBoxButtonTest
8  {
9      // Форма с текстовым полем в режиме ввода пароля
10     // и надписью для вывода содержимого текстового поля
11     public partial class LabelTextBoxButtonTestForm : Form
12     {
13         // Конструктор по умолчанию
14         public LabelTextBoxButtonTestForm()
15         {
16             InitializeComponent();
17         } // Конец конструктора
18
19         // Вывод данных, введенных пользователем, в Label
20         private void displayPasswordButton_Click(
21             object sender, EventArgs e )
22         {
23             // Отображение текста, введенного пользователем
24             displayPasswordLabel.Text = inputPasswordTextBox.Text;
25         } // Конец метода displayPasswordButton_Click
26     } // Конец класса LabelTextBoxButtonTestForm
27 } // Конец пространства имен LabelTextBoxButtonTest

```



**Ил. 14.20.** Программа для вывода скрытого текста

Начните с создания графического интерфейса: перетащите элементы управления (TextBox, Button и Label) на форму. Разместите элементы управления, измените их имена в окне свойств (замените значения по умолчанию textBox1, button1 и label1 более содержательными именами displayPasswordLabel, displayPasswordButton и inputPasswordTextBox). Свойство (Name) в окне свойств позволяет изменить имя переменной элемента управления. Visual Studio создает необходимый код и размещает его в методе InitializeComponent частичного класса в файле LabelTextBoxButtonTestForm.Designer.cs.



Задайте свойству `Text` объекта `displayPasswordButton` значение `"Show Me"` и очистите свойство `Text` объекта `displayPasswordLabel`, чтобы оно было пустым в начале выполнения программы. Свойству `BorderStyle` объекта `displayPasswordLabel` задается значение `Fixed3D`, чтобы имитировать трехмерное оформление `Label`. Мы также задали свойству `TextAlign` значение `MiddleLeft`, чтобы текст надписи был выровнен по центру между верхним и нижним краем. Парольный символ `inputPasswordTextBox` определяется системными настройками пользователя.

Чтобы создать обработчик события для `displayPasswordButton`, мы сделали двойной щелчок на элементе управления в режиме конструктора. В тело обработчика добавлена строка 24. Когда пользователь щелкает на кнопке `Show Me` в приложении, строка 24 получает текст, введенный пользователем в `inputPasswordTextBox`, и выводит текст в `displayPasswordLabel`.

## 14.6. GroupBox и Panel

Контейнеры `GroupBox` и `Panel` предназначены для размещения элементов управления в графическом интерфейсе — обычно в них группируются элементы со сходной функциональностью или выполняющие общую функцию в графическом интерфейсе. Все элементы управления в `GroupBox` или `Panel` перемещаются вместе с перемещаемым контейнером `GroupBox` или `Panel`. Кроме того, контейнеры `GroupBox` и `Panel` могут использоваться для одновременного отображения или сокрытия набора элементов. Изменение свойства `Visible` контейнера изменяет видимость всех содержащихся в нем элементов управления.

Главное различие между этими двумя элементами управления заключается в том, что контейнер `GroupBox` может иметь заголовок и не поддерживает полосы прокрутки, а `Panel` может иметь полосы прокрутки и не содержит заголовка. Контейнер `GroupBox` по умолчанию имеет тонкую границу; `Panel` тоже можно настроить подобным образом при помощи свойства `BorderStyle`. На ил. 14.21–14.22 перечислены часто используемые свойства `GroupBox` и `Panel` соответственно.



### КРАСИВО И УДОБНО 14.2

Контейнеры `Panel` и `GroupBox` могут содержать другие контейнеры `Panel` и `GroupBox` для построения более сложных макетов.

Часто используемые свойства <code>GroupBox</code>	Описание
<code>Controls</code>	Набор элементов управления, содержащихся в <code>GroupBox</code>
<code>Text</code>	Текст заголовка, отображаемого в верхней части <code>GroupBox</code>

**Ил. 14.21.** Часто используемые свойства `GroupBox`

Часто используемые свойства Panel	Описание
AutoScroll	Признак отображения полос прокрутки в том случае, если размеры Panel слишком малы для отображения всего содержимого (по умолчанию используется значение false)
BorderStyle	Тип обрамления Panel. По умолчанию используется значение None; другие допустимые значения — Fixed3D и FixedSingle
Controls	Набор элементов управления, содержащихся в Panel

Ил. 14.22. Свойства Panel

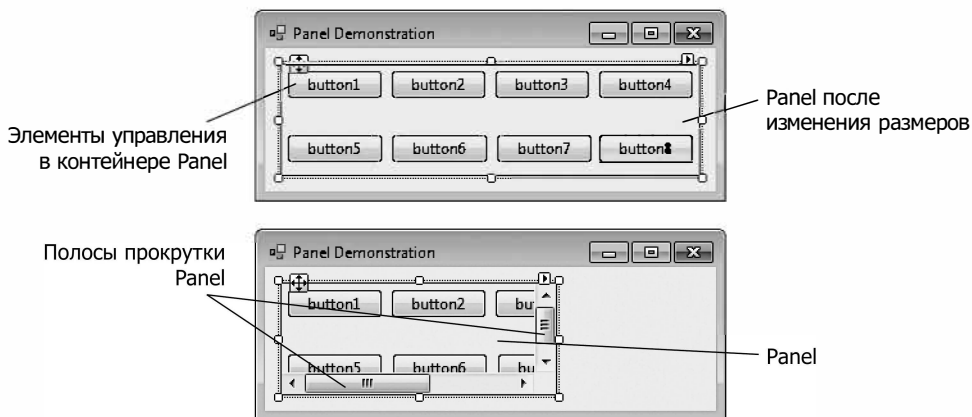


### КРАСИВО И УДОБНО 14.3

Закрепление и стыковка элементов управления в контейнерах GroupBox и Panel упрощают управление графическим интерфейсом. Элементы управления делятся на функциональные «группы», с которыми удобно работать.

Чтобы создать на форме контейнер GroupBox, перетащите его значок с панели элементов на форму. Затем перетащите другие элементы с панели элементов на GroupBox. Эти элементы добавляются в свойство Controls элемента GroupBox и становятся частью GroupBox. Свойство Text контейнера GroupBox определяет заголовок, выводимый у верхнего края GroupBox.

Чтобы создать на форме контейнер Panel, перетащите его значок с панели элементов на форму. Далее вы можете добавить другие элементы в контейнер, перетаскивая их с панели элементов на Panel. Чтобы разрешить использование полос прокрутки, задайте свойству AutoScroll объекта Panel значение true. Если при изменении размеров Panel не может вывести все содержимое, появляются полосы прокрутки (ил. 14.23). Полосы прокрутки могут использоваться для просмотра всех элементов управления в контейнере Panel — и в режиме конструктора, и во время выполнения. На ил. 14.23 свойству BorderStyle объекта Panel задано значение FixedSingle, чтобы контейнер был видим на форме.



Ил. 14.23. Создание контейнера Panel с полосами прокрутки



#### КРАСИВО И УДОБНО 14.4

Контейнеры Panel с полосами прокрутки делают графический интерфейс приложения более компактным.

В программе на ил. 14.24 контейнеры GroupBox и Panel используются для размещения кнопок. При щелчке на кнопках их обработчики событий изменяют текст, выводимый на кнопке.

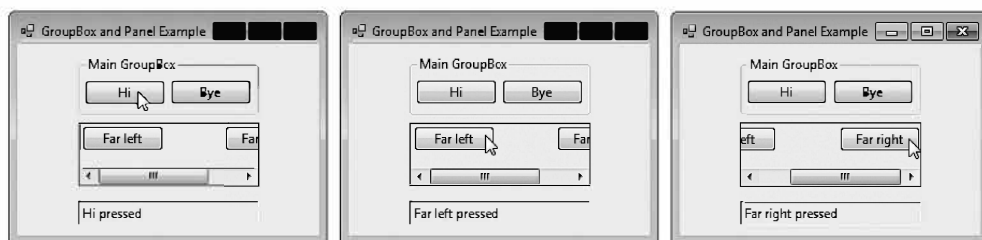
Группа mainGroupBox состоит из двух кнопок — hiButton (с текстом Hi) и byeButton (с текстом Bye). Контейнер Panel (с именем mainPanel) также содержит две кнопки: leftButton (с текстом Far Left) и rightButton (с текстом Far Right). У mainPanel свойство AutoScroll равно true, поэтому когда размеры содержимого Panel превышают размеры видимой области Panel, появляются полосы прокрутки.

```

1  // Ил. 14.24: GroupBoxPanelExampleForm.cs
2  // Использование контейнеров GroupBox и Panel для размещения кнопок.
3  using System;
4  using System.Windows.Forms;
5
6  namespace GroupBoxPanelExample
7  {
8      // Форма с контейнерами GroupBox и Panel
9      public partial class GroupBoxPanelExampleForm : Form
10     {
11         // Конструктор по умолчанию
12         public GroupBoxPanelExampleForm()
13         {
14             InitializeComponent();
15         } // Конец конструктора
16
17         // Обработчик события для кнопки Hi
18         private void hiButton_Click( object sender, EventArgs e )
19         {
20             messageLabel.Text = "Hi pressed"; // Изменение текста надписи
21         } // Конец метода hiButton_Click
22
23         // Обработчик события для кнопки Bye
24         private void byeButton_Click( object sender, EventArgs e )
25         {
26             messageLabel.Text = "Bye pressed"; // Изменение текста надписи
27         } // Конец метода byeButton_Click
28
29         // Обработчик события для кнопки Far Left
30         private void leftButton_Click( object sender, EventArgs e )
31         {
32             messageLabel.Text = "Far left pressed"; // Изменение текста надписи
33         } // Конец метода leftButton_Click
34
35         // Обработчик события для кнопки Far Right
36         private void rightButton_Click( object sender, EventArgs e )
37         {
38             messageLabel.Text = "Far right pressed"; // Изменение текста надписи
39         } // Конец метода rightButton_Click
40     } // Конец класса GroupBoxPanelExampleForm
41 } // Конец пространства имен GroupBoxPanelExample

```

**Ил. 14.24.** Использование контейнеров GroupBox и Panel для размещения кнопок (продолжение ↗)



**Ил. 14.24.** Использование контейнеров `GroupBox` и `Panel` для размещения кнопок (окончание)

Надпись (с именем `messageLabel`) изначально не содержит текста. Для добавления элементов управления в `mainGroupBox` и `mainPanel` Visual Studio вызывает метод `Add` свойства `Controls` каждого из контейнеров. Этот код размещается в частичном классе, находящемся в файле `GroupBoxPanelExample.Designer.cs`.

Обработчики событий четырех кнопок находятся в строках 18–39. Строки 20, 26, 32 и 38 изменяют текст `messageLabel` для обозначения кнопки, нажатой пользователем.

## 14.7. Флажки и переключатели

В C# поддерживаются две разновидности кнопок состояния, которые могут находиться в состоянии «вкл/выкл» или «истина/ложь» — флажки (`CheckBox`) и переключатели (`RadioButton`). Классы `CheckBox` и `RadioButton`, как и класс `Button`, являются производными от класса `ButtonBase`.

### Флажки

Флажок (`CheckBox`) представляет собой маленький квадратик, который может быть пустым или содержит пометку. Когда пользователь щелкает на флажке, чтобы установить его, в квадратике появляется пометка. Повторный щелчок на установленном флажке снимает пометку. Флажок также можно настроить для переключения между тремя состояниями (установленным, снятым и неопределенным); для этого его свойству `ThreeState` присваивается значение `true`. Любое количество флажков может одновременно находиться в установленном состоянии. Список часто используемых свойств и событий `CheckBox` приведен на ил. 14.25.

Свойства и события <code>CheckBox</code>	Описание
Часто используемые свойства	
Appearance	По умолчанию свойство содержит <code>Normal</code> , а флажок отображается в традиционном виде. Если задать свойству значение <code>Button</code> , то флажок будет отображаться в виде кнопки, которая выглядит нажатой при установке флажка

**Ил. 14.25.** Свойства и события `CheckBox` (продолжение ↗)

Свойства и события CheckBox	Описание
Checked	Свойство определяет, находится ли флажок в установленном (помеченном) или снятом (пустом) состоянии. Свойство возвращает тип bool. По умолчанию используется значение false (флажок не установлен)
CheckState	Свойство определяет текущее состояние флажка в виде значения из перечисления CheckState (Checked, Unchecked или Indeterminate). Значение Indeterminate используется для неопределенного состояния флажка (то есть не установленного и не снятого). Когда CheckState задается значение Indeterminate, флажок обычно окрашивается в серый цвет
Text	Текст, выводимый справа от флажка
ThreeState	Если это свойство равно true, флажок имеет три состояния — установленное, снятое и неопределенное. По умолчанию свойство равно false, а флажок имеет всего два состояния (снятое и установленное)
Часто используемые события	
CheckedChanged	Генерируется при изменении свойства Checked или CheckState; является событием по умолчанию класса CheckBox. При двойном щелчке на элементе управления CheckBox в режиме конструктора создается пустой обработчик этого события
CheckStateChanged	Генерируется при изменении свойства Checked или CheckState

**Ил. 14.25.** Свойства и события CheckBox (окончание)

В программе на ил. 14.26 пользователь может устанавливать флажки, чтобы изменять начертание шрифта надписи. Обработчик события одного флажка применяет полужирное начертание, а обработчик другого флажка применяет курсивное начертание. Если установлены оба флажка, текст надписи выводится полужирным курсивом. В исходном состоянии ни один из флажков не установлен.

```

1 // Ил. 14.26: CheckBoxTestForm.cs
2 // Использование флажков для изменения начертания шрифта.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6
7 namespace CheckBoxTest
8 {
9     // Форма содержит флажки для изменения текста.
10    public partial class CheckBoxTestForm : Form
11    {
12        // Конструктор по умолчанию
13        public CheckBoxTestForm()
14        {

```

**Ил. 14.26.** Использование флажков для изменения начертания шрифта (продолжение ➤)

```

15     InitializeComponent();
16 } // Конец конструктора
17
18 // Переключение полужирного начертания в зависимости
19 // от текущего состояния флажка
20 private void boldCheckBox_CheckedChanged(
21     object sender, EventArgs e )
22 {
23     outputLabel.Font = new Font( outputLabel.Font,
24         outputLabel.Font.Style ^ FontStyle.Bold );
25 } // Конец метода boldCheckBox_CheckedChanged
26
27 // Переключение курсивного начертания в зависимости
28 // от текущего состояния флажка
29 private void italicCheckBox_CheckedChanged(
30     object sender, EventArgs e )
31 {
32     outputLabel.Font = new Font( outputLabel.Font,
33         outputLabel.Font.Style ^ FontStyle.Italic );
34 } // Конец метода italicCheckBox_CheckedChanged
35 } // Конец класса CheckBoxTestForm
36 } // Конец пространства имен CheckBoxTest

```



**Ил. 14.26.** Использование флажков для изменения начертания шрифта (окончание)

У флажка `boldCheckBox` свойству `Text` задается значение `Bold`, а у флажка `italicCheckBox` — значение `Italic`. Свойству `Text` надписи `outputLabel` задается текст `Watch the font style change`. После создания элементов управления мы определяем их обработчики событий. Двойной щелчок на флажках в режиме конструктора создает пустые обработчики событий `CheckedChanged`.

Чтобы изменить начертание шрифта надписи, мы задаем ее свойству `Font` новый объект `Font` (строки 23–24 и 32–33). Класс `Font` определяется в пространстве имен `System.Drawing`. В аргументах используемого конструктора `Font` передается текущий шрифт и новый стиль начертания. Первый аргумент `outputLabel.Font` использует исходное имя и размер шрифта `outputLabel`. Начертание задается значениями из перечисления `FontStyle`: `Regular`, `Bold`, `Italic`, `Strikeout` и `Underline`. (Значение `Strikeout` выводит перечеркнутый текст.) Свойство `Style` объекта `Font` доступно только для чтения, поэтому оно может задаваться только при создании объекта `Font`.

### Объединение стилей начертания поразрядными операторами

Стили можно комбинировать при помощи *поразрядных операторов*, работающих с отдельными битами информации. Как говорилось в главе 1, все данные представляются на компьютере сочетаниями нулей и единиц. Каждый нуль или единица представляет состояние одного бита. Перечисление `FontStyle` (пространство имен `System.Drawing`) представляется набором битов, которые выбираются таким образом, чтобы элементы `FontStyle` можно было объединять в составные стили при помощи поразрядных операторов. Стили не являются взаимоисключающими, поэтому их добавление и удаление не влияет на другие элементы `FontStyle` в комбинациях. Для определения комбинаций используется логический оператор ИЛИ (`|`) или логический исключающий оператор ИЛИ (`^`). Когда логический оператор ИЛИ применяется к двум битам, если хотя бы один из двух битов содержит 1, то результат равен 1. Предположим, значение `FontStyle.Bold` представляется битами 01, а значение `FontStyle.Italic` — битами 10. При объединении стилей логическим оператором ИЛИ (`|`) мы получим биты 11.

```
01 = Bold
10 = Italic
--
11 = Bold и Italic
```

Логический оператор ИЛИ помогает строить комбинации стилей. Но что, если нам потребуется удалить стиль из комбинации, как на ил. 14.26? В этом поможет логический исключающий оператор ИЛИ: когда он применяется к двум битам и оба бита имеют одинаковые значения, то результат равен 0. Если биты различны, результат равен 1. Снова предположим, что значение `FontStyle.Bold` представляется битами 01, а значение `FontStyle.Italic` — битами 10. Применяя логический исключающий оператор ИЛИ (`^`), мы получаем биты 11.

```
01 = Bold
10 = Italic
--
11 = Bold и Italic
```

Теперь предположим, что из комбинации `FontStyle.Bold` и `FontStyle.Italic` нужно удалить значение `FontStyle.Bold`. Для этого достаточно применить к комбинации и `FontStyle.Bold` логический исключающий оператор ИЛИ:

```
11 = Bold и Italic
01 = Bold
--
10 = Italic
```

Мы рассмотрели очень простой пример. Преимущества применения поразрядных операторов становятся более очевидными, если учесть, что существуют пять значений `FontStyle` (`Bold`, `Italic`, `Regular`, `Strikeout` и `Underline`), образующих 16 комбинаций `FontStyle`. Использование поразрядных операторов сильно уменьшает объем кода, необходимого для проверки всех возможных комбинаций.

На ил. 14.26 необходимо задать `FontStyle` так, чтобы текст выводился полужирным шрифтом, если до этого он не имел полужирного начертания, и наоборот. В строке 24 для этого используется поразрядный исключающий оператор ИЛИ. Если

в значении `outputLabel.Font.Style` установлен бит полужирного начертания, то в полученном стиле его не будет. Если текст изначально выводится курсивом, то полученный текст будет полужирным и курсивным (вместо просто полужирного). То же относится и к стилю `FontStyle.Italic` в строке 33.

Если бы мы не использовали поразрядные операторы для комбинирования элементов `FontStyle`, нам пришлось бы проверять текущий стиль и изменять его соответствующим образом. В обработчике `boldCheckBox_CheckedChanged` придется проверить обычное начертание и сделать его полужирным, проверить полужирное начертание и сделать его обычным, проверить курсивное начертание и сделать его полужирным курсивом, проверить полужирное курсивное начертание и сделать его курсивным. Такое решение получается крайне громоздким, потому что для каждого добавляемого стиля количество комбинаций удваивается. При добавлении флажка подчеркивания придется проверять еще 8 дополнительных комбинаций, а при добавлении флажка для перечеркивания их станет уже 16.

## Переключатели

Переключатели (класс `RadioButton`), как и флажки, могут находиться в одном из двух состояний — установленном и снятом. Однако переключатели обычно объединяются в группы, в которых в любой момент времени может быть установлен только один переключатель. При установке одного переключателя в группе остальные переключатели снимаются. Таким образом, переключатели используются для представления наборов взаимоисключающих вариантов (то есть наборов, в которых одновременное выделение нескольких вариантов невозможно).



### КРАСИВО И УДОБНО 14.5

Используйте переключатели, если в группе может быть выбран только один вариант. Используйте флажки, если в группе можно выбрать несколько вариантов одновременно.

Все переключатели, добавленные в контейнер, образуют одну группу. Чтобы разделить переключатели на несколько групп, добавьте их в разные контейнеры (например, `GroupBox` или `Panel`). Часто используемые свойства и событие класса `RadioButton` перечислены на ил. 14.27.

Свойства и событие <code>RadioButton</code>	Описание
Часто используемые свойства	
<code>Checked</code>	Свойство определяет, находится ли переключатель в установленном состоянии
<code>Text</code>	Текст, выводимый рядом с переключателем
Часто используемое событие	
<code>CheckedChanged</code>	Генерируется при установлении или снятии переключателя. При двойном щелчке на элементе управления <code>RadioButton</code> в режиме конструктора создается пустой обработчик этого события

**Ил. 14.27.** Свойства и событие `CheckBox`





### АРХИТЕКТУРНОЕ РЕШЕНИЕ 14.1

Контейнеры Form, GroupBox и Panel используются для логической группировки переключателей. Переключатели в каждой группе являются взаимноисключающими по отношению друг к другу, но не к переключателям из других групп.

В программе на ил. 14.28 переключатели используются для выбора атрибутов MessageBox. После выбора конфигурации пользователь нажимает кнопку Display Button для вывода MessageBox. Надпись в левом нижнем углу содержит результат MessageBox (то есть какую кнопку нажал пользователь — Yes, No, Cancel и т. д.).

```

1  // Ил. 14.28: RadioButtonsTestForm.cs
2  // Использование переключателей для настройки атрибутов окна сообщения.
3  using System;
4  using System.Windows.Forms;
5
6  namespace RadioButtonsTest
7  {
8      // Пользователь выбирает один переключатель в каждой группе,
9      // чтобы настроить атрибуты окна сообщения.
10     public partial class RadioButtonsTestForm : Form
11     {
12         // Создание переменной для хранения настроек пользователя
13         private MessageBoxIcon iconType;
14         private MessageBoxButtons buttonType;
15
16         // Конструктор по умолчанию
17         public RadioButtonsTestForm()
18         {
19             InitializeComponent();
20         } // Конец конструктора
21
22         // Изменение кнопок в зависимости от настроек пользователя
23         private void buttonType_CheckedChanged(
24             object sender, EventArgs e )
25         {
26             if ( sender == okRadioButton ) // Вывод кнопки OK
27                 buttonType = MessageBoxButtons.OK;
28
29             // Вывод кнопок OK и Cancel
30             else if ( sender == okCancelRadioButton )
31                 buttonType = MessageBoxButtons.OKCancel;
32
33             // Вывод кнопок Abort, Retry и Ignore
34             else if ( sender == abortRetryIgnoreRadioButton )
35                 buttonType = MessageBoxButtons.AbortRetryIgnore;
36
37             // Вывод кнопок Yes, No и Cancel
38             else if ( sender == yesNoCancelRadioButton )
39                 buttonType = MessageBoxButtons.YesNoCancel;
40
41             // Вывод кнопок Yes и No
42             else if ( sender == yesNoRadioButton )
43                 buttonType = MessageBoxButtons.YesNo;

```

**Ил. 14.28.** Использование переключателей для настройки атрибутов окна сообщения (продолжение ↗)

```

44
45         // Последний вариант -- кнопки Retry и Cancel
46         else
47             buttonType = MessageBoxButtons.RetryCancel;
48     } // Конец метода buttonType_CheckedChanged
49
50     // Изменение значка на основании настроек sender
51     private void iconType_CheckedChanged( object sender, EventArgs e )
52     {
53         if ( sender == asteriskRadioButton ) // Звездочка
54             iconType = MessageBoxIcon.Asterisk;
55
56         // Значок ошибки
57         else if ( sender == errorRadioButton )
58             iconType = MessageBoxIcon.Error;
59
60         // Восклицательный знак
61         else if ( sender == exclamationRadioButton )
62             iconType = MessageBoxIcon.Exclamation;
63
64         // Рука
65         else if ( sender == handRadioButton )
66             iconType = MessageBoxIcon.Hand;
67
68         // Информация
69         else if ( sender == informationRadioButton )
70             iconType = MessageBoxIcon.Information;
71
72         // Вопросительный знак
73         else if ( sender == questionRadioButton )
74             iconType = MessageBoxIcon.Question;
75
76         // Знак "Стоп"
77         else if ( sender == stopRadioButton )
78             iconType = MessageBoxIcon.Stop;
79
80         // Последний вариант -- предупреждающий знак
81         else
82             iconType = MessageBoxIcon.Warning;
83     } // Конец метода iconType_CheckedChanged
84
85     // Вывод окна с информацией о кнопке, нажатой пользователем
86     private void displayButton_Click( object sender, EventArgs e )
87     {
88         // Отображение MessageBox и сохранение значения
89         // кнопки, нажатой пользователем
90         DialogResult result = MessageBox.Show(
91             "This is your Custom MessageBox.", "Custom MessageBox",
92             buttonType, iconType );
93
94         // check to see which Button was pressed in the MessageBox
95         // change text displayed accordingly
96         switch (result)
97         {
98             case DialogResult.OK:

```

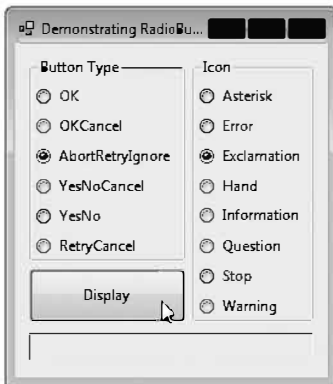
**Ил. 14.28.** Использование переключателей для настройки атрибутов окна сообщения (продолжение ↗)

```

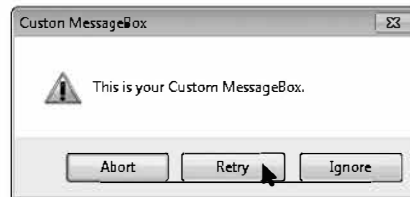
99         displayLabel.Text = "OK was pressed.";
100        break;
101    case DialogResult.Cancel:
102        displayLabel.Text = "Cancel was pressed.";
103        break;
104    case DialogResult.Abort:
105        displayLabel.Text = "Abort was pressed.";
106        break;
107    case DialogResult.Retry:
108        displayLabel.Text = "Retry was pressed.";
109        break;
110    case DialogResult.Ignore:
111        displayLabel.Text = "Ignore was pressed.";
112        break;
113    case DialogResult.Yes:
114        displayLabel.Text = "Yes was pressed.";
115        break;
116    case DialogResult.No:
117        displayLabel.Text = "No was pressed.";
118        break;
119    } // Конец switch
120    } // Конец метода displayButton_Click
121    } // Конец класса RadioButtonsTestForm
122 } // Конец пространства имен RadioButtonsTest

```

а) Графический интерфейс для тестирования переключателей



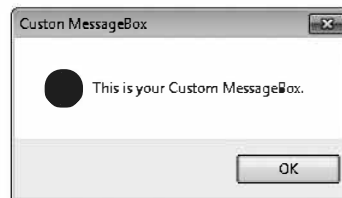
б) Окно типа AbortRetryIgnore



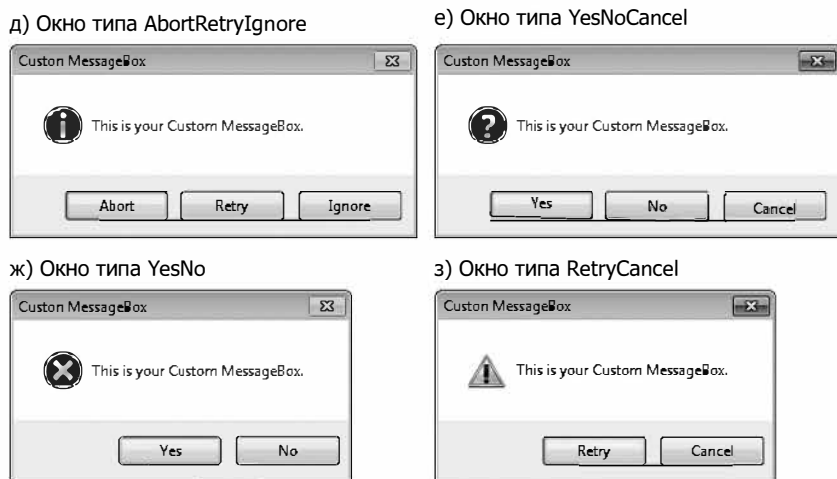
в) Окно типа OKCancel



г) Окно типа OK



**Ил. 14.28.** Использование переключателей для настройки атрибутов окна (продолжение ↗)



**Ил. 14.28.** Использование переключателей для настройки атрибутов окна (окончание)

Настройки пользователя сохраняются в переменных `iconType` и `buttonType` (объявляемых в строках 13–14). Объект `iconType` относится к типу `MessageBoxIcon` и может принимать значения `Asterisk`, `Error`, `Exclamation`, `Hand`, `Information`, `None`, `Question`, `Stop` и `Warning`. В примерах вывода представлены только значки `Error`, `Exclamation`, `Information` и `Question`.

Объект `buttonType` относится к типу `MessageBoxButton` и может принимать значения `AbortRetryIgnore`, `OK`, `OKCancel`, `RetryCancel`, `YesNo` и `YesNoCancel`. Имя определяет набор вариантов, выводимых в окне `MessageBox`. В примерах вывода показаны окна сообщений для всех значений перечисления `MessageBoxButtons`.

Мы создали два контейнера `GroupBox`, по одному для каждого набора значений; им назначены заголовки `Button Type` и `Icon`. В контейнерах содержатся переключатели, представляющие значения из перечислений, а свойства `Text` переключателей задаются соответствующим образом. Так как переключатели объединяются в логические группы, в каждом контейнере `GroupBox` может быть установлен только один переключатель. Также создается кнопка (`displayButton`) с текстом `Display`. Когда пользователь щелкает на кнопке, на экране появляется окно `MessageBox` с заданными атрибутами. В надписи (`displayLabel`) выводится информация о том, какая кнопка была нажата пользователем в `MessageBox`.

Обработчик события переключателей обрабатывает событие `CheckedChanged` каждого переключателя. При установке переключателя, содержащегося в контейнере `GroupBox` с текстом `Button Type`, соответствующий обработчик события инициализирует `buttonType` соответствующим значением. Обработка событий переключателей определяется в строках 23–48. Аналогичным образом, когда пользователь устанавливает переключатели из контейнера `Icon`, обработчик, связанный с этими событиями (строки 51–83), задает `iconType` соответствующее значение.

Обработчик события Click кнопки `displayButton` (строки 86–120) создает окно сообщения `MessageBox` (строки 90–92). Атрибуты `MessageBox` задаются значениями, хранящимися в `iconType` и `buttonType`. Когда пользователь щелкает на одной из кнопок `MessageBox`, приложению возвращается результат — значение из перечисления `DialogResult` (`Abort`, `Cancel`, `Ignore`, `No`, `None`, `OK`, `Retry` или `Yes`). Команда `switch` в строках 96–119 проверяет результат и задает значение `displayLabel.Text` соответствующим образом.

## 14.8. Графическое поле

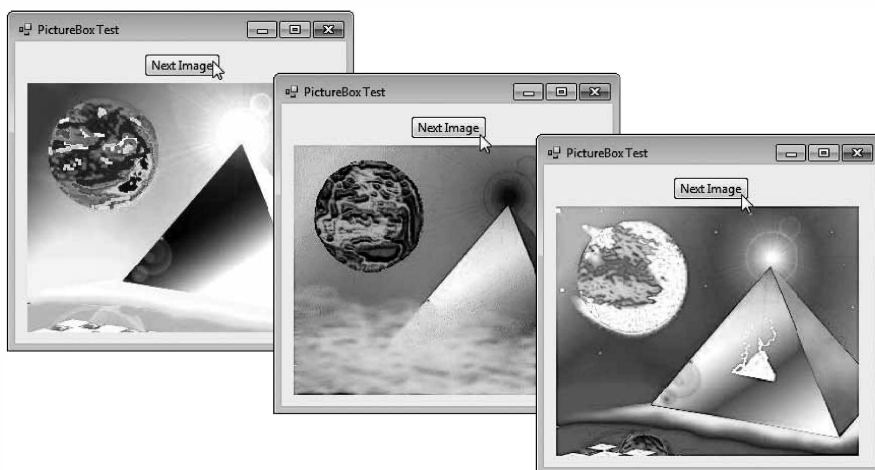
Графическое поле (`PictureBox`) используется для вывода изображений. Поддерживаются разные форматы графики: BMP, PNG (Portable Network Graphics), GIF (Graphics Interchange Format) и JPEG. Свойство `Image` элемента управления `PictureBox` задает выводимое изображение, а свойство `SizeMode` определяет режим вывода (`Normal`, `StretchImage`, `AutoSize`, `CenterImage` или `Zoom`). На ил. 14.29 описаны часто используемые свойства и событие `PictureBox`.

Свойства и событие <code>PictureBox</code>	Описание
Часто используемые свойства	
<code>Image</code>	Изображение, выводимое в <code>PictureBox</code>
<code>SizeMode</code>	Перечисление, управляющее размером и режимом позиционирования графики. В режиме <code>Normal</code> (используется по умолчанию) изображение размещается в левом верхнем углу <code>PictureBox</code> , а в режиме <code>CenterImage</code> — в центре. В этих двух режимах слишком большие изображения автоматически обрезаются. Режим <code>StretchImage</code> изменяет размеры изображения по размерам <code>PictureBox</code> . Режим <code>AutoSize</code> подгоняет размеры <code>PictureBox</code> под размеры изображения. В режиме <code>Zoom</code> размеры изображения изменяются в соответствии с размерами <code>PictureBox</code> , но без нарушения исходных пропорций
Часто используемое событие	
<code>Click</code>	Происходит при щелчке на элементе управления. При двойном щелчке на элементе управления <code>PictureBox</code> в режиме конструктора создается пустой обработчик этого события

**Ил. 14.29.** Свойства и событие `PictureBox`

На ил. 14.30 элемент управления `PictureBox` с именем `imagePictureBox` используется для вывода одного из трех изображений — `image0.bmp`, `image1.bmp` или `image2.bmp`. Эти изображения находятся в подкаталоге `Images` каталога примеров этой главы. Каждый раз, когда пользователь щелкает на кнопке `Next Image`, текущее изображение заменяется следующим. Если кнопка `Next Image` нажимается при выводе последнего изображения, снова открывается первое изображение.

```
1 // Ил. 14.30: PictureBoxTestForm.cs
2 // Использование элемента управления PictureBox для вывода графики.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6
7 namespace PictureBoxTest
8 {
9     // Форма для вывода разных изображений при щелчке на кнопке
10    public partial class PictureBoxTestForm : Form
11    {
12        private int imageNum = -1; // Определяет выводимое изображение
13
14        // Конструктор по умолчанию
15        public PictureBoxTestForm()
16        {
17            InitializeComponent();
18        } // Конец конструктора
19
20        // Изображение сменяется при каждом щелчке на кнопке Next Image
21        private void nextButton_Click( object sender, EventArgs e )
22        {
23            imageNum = ( imageNum + 1 ) % 3; // imageNum cycles from 0 to 2
24
25            // Загрузка изображения из ресурсов и помещение в PictureBox
26            imagePictureBox.Image = ( Image )
27            ( Properties.Resources.ResourceManager.GetObject(
28                string.Format( "image{0}", imageNum ) ) );
29        } // Конец метода nextButton_Click
30    } // Конец класса PictureBoxTestForm
31 } // Конец пространства имен PictureBoxTest
```



**Ил. 14.30.** Использование PictureBox для вывода графики

### Программное использование ресурсов

В этом примере изображения были включены в проект как *ресурсы*. Компилятор встроил их в исполняемый файл, чтобы приложение могло обращаться к изображениям через пространство имен `Properties` проекта. Встраивая графику в приложение, мы можем не беспокоиться о необходимости ее перемещения вместе с приложением на другие компьютеры.

При создании нового проекта используйте следующие действия для добавления графики в проект в виде ресурсов:

1. После создания проекта щелкните правой кнопкой мыши на узле `Properties` проекта в окне `Solution Explorer`. Выберите команду `Open`, чтобы открыть свойства проекта.
2. Перейдите на вкладку `Resources`.
3. Щелкните на стрелке рядом с `Add Resource` и выберите команду `Add Existing File...`, чтобы вызвать диалоговое окно `Add existing file to resources`.
4. Найдите файлы изображений, которые нужно включить в ресурсы приложения, и щелкните на кнопке `Open`. В папку `Images` примеров этой главы были включены три изображения.
5. Сохраните проект.

В окне `Solution Explorer` появляется папка с именем `Resources`. В дальнейшем мы будем использовать этот прием для большинства приложений, использующих графику.

Ресурсы проекта хранятся в классе `Resources` (из пространства имен `Properties` проекта). Класс `Resources` содержит объект `ResourceManager` для программного взаимодействия с ресурсами. Для обращения к изображению можно использовать метод `GetObject`, который получает в аргументе имя ресурса в том виде, в котором оно отображается на вкладке `Resources` (например, `"image0"`), и возвращает ресурс в формате `Object`. В строках 27–28 метод `GetObject` вызывается с результатом выражения

```
string.Format( "image{0}", imageNum )
```

Это выражение строит имя ресурса из слова `"image"` и индекса следующего изображения (`imageNum`, полученного ранее в строке 23). Чтобы присвоить объект свойству `Image` элемента управления `PictureBox` (строка 26), необходимо преобразовать его к типу `Image` (пространство имен `System.Drawing`).

Класс `Resources` также предоставляет прямой доступ к ресурсам в формате `Resources.имяРесурса`, где *имяРесурса* — имя, назначенное ресурсу при его создании. При использовании такого выражения возвращаемый ресурс уже относится к правильному типу. Например, выражение `Properties.Resources.image0` дает объект `Image`, представляющий первое изображение.

## 14.9. Подсказки

В главе 2 были продемонстрированы экранные *подсказки* (tool tips) — полезный текст, который появляется при наведении указателя мыши на элемент графического интерфейса. Подсказки, отображаемые в Visual Studio, помогают вам освоить функциональность IDE и напоминают, для чего нужна та или иная кнопка панели инструментов. Подсказки используются во многих популярных программах — например, в Microsoft Word. В этом разделе вы узнаете, как использовать компонент `ToolTip` для добавления поддержки подсказок в ваши приложения. На ил. 14.31 перечислены часто используемые свойства и событие класса `ToolTip`.

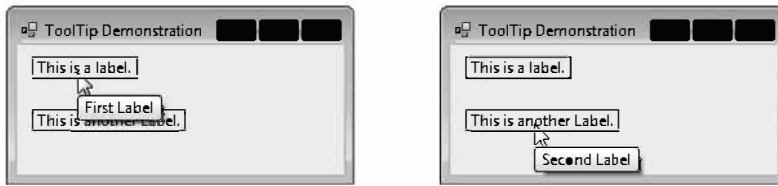
Свойства и событие <code>ToolTip</code>	Описание
Часто используемые свойства	
<code>AutoPopDelay</code>	Промежуток времени (в миллисекундах), в течение которого отображается подсказка при наведении указателя мыши на элемент управления
<code>InitialDelay</code>	Промежуток времени (в миллисекундах), после которого при наведении указателя мыши на элемент управления появляется подсказка
<code>ReshowDelay</code>	Промежуток времени (в миллисекундах), разделяющий появление двух разных подсказок (при перемещении указателя мыши между элементами управления)
Часто используемое событие	
<code>Draw</code>	Генерируется при отображении подсказки и позволяет программисту изменить ее внешний вид

**Ил. 14.31.** Свойства и событие `ToolTip`

Компонент `ToolTip`, добавленный с панели элементов, отображается в *области компонентов* (component tray) — области, находящейся под формой в режиме конструктора. После добавления компонента `ToolTip` на форму в окне свойств других элементов управления формы появляется новое свойство с именем, состоящим из строки `ToolTip on` и имени компонента `ToolTip`. Например, на нашей форме компоненту `ToolTip` присвоено имя `helpfulToolTip`, поэтому для определения текста подсказки элемента управления следует задать свойство `ToolTip on helpfulToolTip` этого элемента.

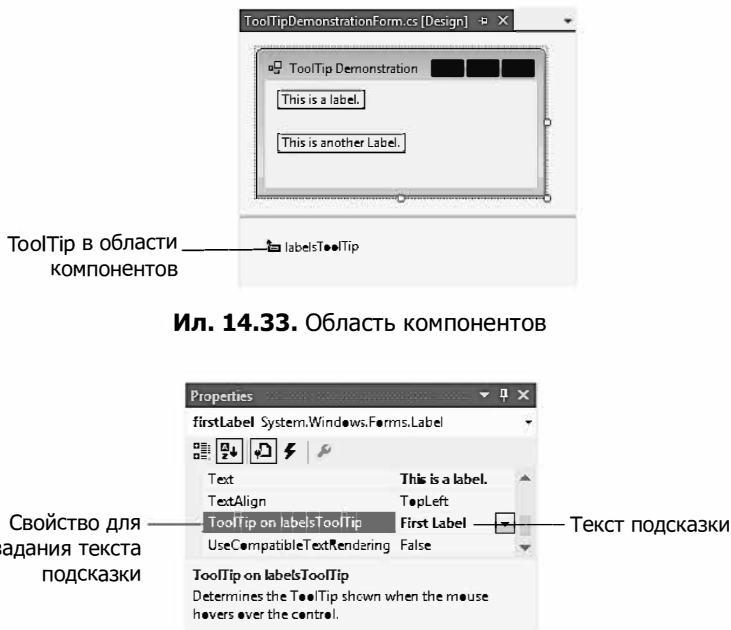
На ил. 14.32 показано, как работает компонент `ToolTip`. В нашем примере создана форма с двумя надписями, для которых выводятся разные подсказки. Чтобы пример лучше выглядел, мы задали свойству `BorderStyle` каждой надписи значение `FixedSingle`, при котором выводится сплошная граница. Так как обработка событий в этом примере не используется, мы не приводим код класса формы.





Ил. 14.32. Использование компонента ToolTip

В нашем примере компоненту ToolTip было присвоено имя `labelsToolTip`. На ил. 14.33 изображен компонент ToolTip в области компонентов. Мы задали тексту подсказки первой надписи значение "FirstLabel", а тексту подсказки второй надписи — значение "SecondLabel". На ил. 14.34 показано, как настраивается текст подсказки для первой надписи.



Ил. 14.33. Область компонентов

Ил. 14.34. Настройка текста подсказки для элемента управления

## 14.10. Поле со счетчиком

В некоторых ситуациях данные, вводимые пользователем, должны ограничиваться конкретным диапазоном числовых значений. Для этого используется элемент управления `NumericUpDown`. Он выглядит как текстовое поле, справа от которого расположены две маленькие кнопки со стрелками вверх и вниз. По умолчанию пользователь может вводить числовые значения так, как это делается в текстовом

поле, или же щелкать на кнопках со стрелками для увеличения/уменьшения текущего значения. Максимальное и минимальное значения в диапазоне задаются свойствами `Maximum` и `Minimum` соответственно (оба свойства относятся к типу `decimal`). Свойство `Increment` (также относящееся к типу `decimal`) определяет величину изменения текущего значения при щелчке на кнопке. Свойство `DecimalPlaces` определяет количество знаков в дробной части отображаемого числа. На ил. 14.35 представлены часто используемые свойства и событие `NumericUpDown`.

Свойства и событие <code>NumericUpDown</code>	Описание
Часто используемые свойства	
<code>DecimalPlaces</code>	Количество знаков в дробной части отображаемого числа
<code>Increment</code>	Величина изменения текущего числа в элементе управления при щелчках на кнопках со стрелками
<code>Maximum</code>	Максимальное значение в диапазоне элемента управления
<code>Minimum</code>	Минимальное значение в диапазоне элемента управления
<code>UpDownAlign</code>	Режим выравнивания кнопок со стрелками (кнопки могут отображаться слева или справа от элемента управления)
<code>Value</code>	Числовое значение, отображаемое в элементе
Часто используемое событие	
<code>ValueChanged</code>	Генерируется при изменении текущего значения в элементе управления; является событием по умолчанию класса <code>NumericUpDown</code>

**Ил. 14.35.** Свойства и событие `NumericUpDown`

Программа на ил. 14.36 демонстрирует использование элемента управления `NumericUpDown` в приложении для вычисления сложных процентов (по аналогии с приложением на ил. 6.6). В текстовых полях вводится исходная сумма и процентная ставка, а элемент `NumericUpDown` используется для ввода периода для вычисления процентов (в годах).

```

1  // Ил. 14.36: InterestCalculatorForm.cs
2  // Использование элемента управления NumericUpDown.
3  using System;
4  using System.Windows.Forms;
5
6  namespace NumericUpDownTest
7  {
8      public partial class InterestCalculatorForm : Form
9      {
10         // Конструктор по умолчанию
11         public InterestCalculatorForm()
12         {
13             InitializeComponent();

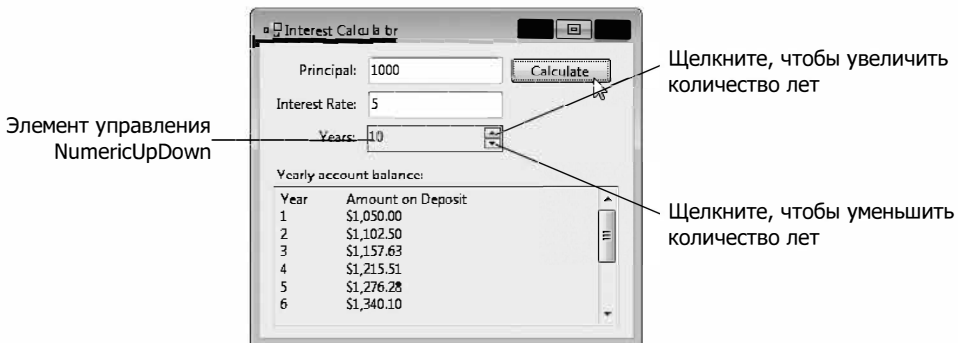
```

**Ил. 14.36.** Использование элемента управления `NumericUpDown` (продолжение ➤)

```

14     } // Конец конструктора
15
16     private void calculateButton_Click(
17         object sender, EventArgs e )
18     {
19         // Объявление переменных для хранения введенных данных
20         decimal principal; // Исходная сумма
21         double rate; // Процентная ставка
22         int year; // Количество лет
23         decimal amount; // Промежуточная сумма
24         string output; // Выходные данные
25
26         // Получение данных от пользователя
27         principal = Convert.ToDecimal( principalTextBox.Text );
28         rate = Convert.ToDouble( interestTextBox.Text );
29         year = Convert.ToInt32( yearUpDown.Value );
30
31         // Вывод заголовка выходных данных
32         output = "Year\tAmount on Deposit\r\n";
33
34         // Вычисление промежуточной суммы и присоединение ее к output
35         for ( int yearCounter = 1; yearCounter <= year; ++yearCounter )
36         {
37             amount = principal *
38                 ( ( decimal ) Math.Pow( ( 1 + rate / 100 ), yearCounter ) );
39             output += ( yearCounter + "\t" +
40                 string.Format( "{0:C}", amount ) + "\r\n" );
41         } // Конец for
42
43         displayTextBox.Text = output; // Вывод результата
44     } // Конец метода calculateButton_Click
45 } // Конец класса InterestCalculatorForm
46 } // Конец пространства имен NumericUpDownTest

```



**Ил. 14.36.** Использование элемента управления NumericUpDown (окончание)

У элемента управления NumericUpDown с именем yearUpDown свойство Minimum равно 1, а свойство Maximum равно 10. Свойству Increment оставлено значение 1, используемое по умолчанию. Эти настройки позволяют ввести количество лет в диапазоне от 1 до 10 с приращением 1. Если задать свойству Increment значение 0.5, то пользователь

сможет вводить такие значения, как 1.5 или 2.5. Если не изменять значение свойства `DecimalPlaces` (0 по умолчанию), то значения 1.5 и 2.5 будут отображаться в виде 2 и 3 соответственно. Также можно задать свойство `ReadOnly` равным `true`, чтобы пользователь не мог просто ввести нужное число в поле (и должен был использовать кнопки со стрелками для изменения текущего значения). По умолчанию свойство `ReadOnly` равно `false`. Выходные данные приложения выводятся в многострочном текстовом поле с вертикальной полосой прокрутки, чтобы пользователь мог просмотреть все результаты.

## 14.11. Обработка событий мыши

В этом разделе вы научитесь обрабатывать события мыши (такие, как щелчки и перемещения), генерируемые при взаимодействиях мыши с элементами управления. События мыши могут обрабатываться для любых элементов, производных от класса `System.Windows.Forms.Control`. Для большинства событий мыши информация о событии передается обработчику через объект класса `MouseEventArgs`, а делегатом, используемым для создания обработчиков, является `EventHandler`. Каждому обработчику таких событий должны передаваться два параметра: `object` и объект `MouseEventArgs`.

Класс `MouseEventArgs` содержит информацию, относящуюся к событию мыши, — координаты указателя мыши, нажатая кнопка (`Right`, `Left` или `Middle`) и количество щелчков кнопки. Координаты в объекте `MouseEventArgs` задаются относительно элемента управления, сгенерировавшего событие, — то есть точка (0,0) соответствует левому верхнему углу элемента управления, в котором произошло событие мыши. Часто используемые события мыши и аргументы событий перечислены в таблице на ил. 14.37.

События мыши и аргументы событий	Описание
События мыши с аргументом типа <code>EventArgs</code>	
<code>MouseEnter</code>	Указатель мыши входит в границы элемента управления
<code>MouseHover</code>	Указатель мыши задерживается на одном месте в границах элемента управления
<code>MouseLeave</code>	Указатель мыши выходит за границы элемента управления
События мыши с аргументом типа <code>MouseEventArgs</code>	
<code>MouseDown</code>	Кнопка мыши нажимается в области элемента управления
<code>MouseMove</code>	Указатель мыши перемещается в границах элемента управления

**Ил. 14.37.** События мыши и аргументы событий (продолжение ↗)

События мыши и аргументы событий	Описание
MouseDown	Кнопка мыши отпускается в границах элемента управления
MouseWheel	Колесо мыши перемещается при нахождении фокуса у элемента управления
Свойства класса MouseEventArgs	
Button	Нажатая кнопка мыши (Left, Right, Middle, None)
Clicks	Количество щелчков кнопкой мыши
X	Координата X точки возникновения события в элементе управления
Y	Координата Y точки возникновения события в элементе управления

**Ил. 14.37.** События мыши и аргументы событий (окончание)

В программе на ил. 14.38 события мыши используются для рисования на форме. Каждый раз, когда пользователь перетаскивает указатель мыши (то есть перемещает мышь с нажатой кнопкой), на форме рисуются кружки в точках возникновения событий во время операции перетаскивания.

```

1  // Ил. 14.38: PainterForm.cs
2  // Использование мыши для рисования на форме.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace Painter
8
9      // Создание формы - поверхности для рисования
10     public partial class PainterForm : Form
11     {
12         bool shouldPaint = false; // Признак режима рисования
13
14         // Конструктор по умолчанию
15         public PainterForm()
16         {
17             InitializeComponent();
18         } // Конец конструктора
19
20         // Рисование при нажатой кнопке мыши
21         private void PainterForm_MouseDown(
22             object sender, MouseEventArgs e )
23         {
24             // Режим рисования включается
25             shouldPaint = true;
26         } // Конец метода PainterForm_MouseDown
27
28         // При отпускании кнопки мыши рисование прекращается
29         private void PainterForm_MouseUp( object sender, MouseEventArgs e )

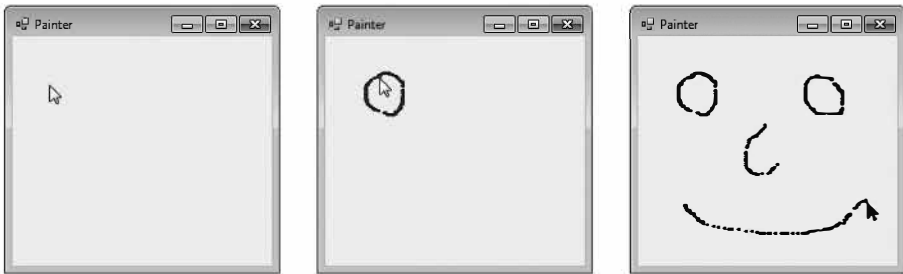
```

**Ил. 14.38.** Использование мыши для рисования на форме (продолжение ↗)

```

30     {
31         // Режим рисования отключается
32         shouldPaint = false;
33     } // Конец метода PainterForm_MouseUp
34
35     // Рисование круга при перемещении мыши с нажатой кнопкой
36     private void PainterForm_MouseMove(
37         object sender, MouseEventArgs e )
38     {
39         if ( shouldPaint ) // Проверяем, нажата ли кнопка мыши
40         {
41             // Рисование круга в точке нахождения указателя мыши
42             using ( Graphics graphics = CreateGraphics() )
43             {
44                 graphics.FillEllipse(
45                     new SolidBrush( Color.BlueViolet ), e.X, e.Y, 4, 4 );
46             } // Конец using; вызывается graphics.Dispose()
47         } // Конец if
48     } // Конец метода PainterForm_MouseMove
49 } // Конец класса PainterForm
50 } // Конец пространства имен Painter

```



**Ил. 14.38.** Использование мыши для рисования на форме (окончание)

В строке 12 объявляется переменная `shouldPaint`, которая определяет, активен ли режим рисования на форме. Программа должна рисовать только при нажатой кнопке мыши. Когда пользователь щелкает или удерживает нажатой кнопку мыши, система генерирует событие `MouseDown`, а обработчик события (строки 21–26) задает `shouldPaint` значение `true`. Когда пользователь отпускает кнопку мыши, система генерирует событие `MouseUp`, в обработчике которого `PainterForm_MouseUp` переменной `shouldPaint` присваивается `false` (строки 29–33), а программа прекращает рисование. В отличие от событий `MouseMove`, происходящих непрерывно при перемещении мыши, система генерирует событие `MouseDown` только при первом *нажатии* кнопки мыши, и событие `MouseUp` — только при ее *отпускании*.

Каждый раз, когда мышь перемещается по элементу управления, происходит событие `MouseMove` этого элемента. В обработчике `PainterForm_MouseMove` (строки 36–48) программа рисует только в том случае, если переменная `shouldPaint` равна `true` (то есть кнопка мыши нажата). В команде `using` строка 42 вызывает унаследованный от `Form` метод `CreateGraphics` для создания объекта `Graphics`, позволяющего программе рисовать на форме. Класс `Graphics` предоставляет методы для рисования

разных фигур. Например, в строках 44–45 метод `FillEllipse` используется для рисования круга. В первом параметре `FillEllipse` в данном случае передается объект класса `SolidBrush`, который задает цвет заливки формы. Цвет передается в аргументе конструктора класса `SolidBrush`. Тип `Color` содержит многочисленные, заранее определенные цветовые константы (в нашем примере выбирается значение `Color.BlueViolet`). `FillEllipse` рисует эллипс в ограничивающем прямоугольнике, который задается координатами  $x$  и  $y$  левого верхнего угла, шириной и высотой — последние четыре аргумента метода. Координаты  $x$  и  $y$  представляют местонахождение события мыши; они могут быть получены из аргументов события мыши (`e.x` и `e.y`). Чтобы нарисовать круг, мы задаем равные значения ширины и высоты ограничивающего прямоугольника — в нашем примере они равны 4 пикселям. Классы `Graphics`, `SolidBrush` и `Color` являются частью пространства имен `System.Drawing`. Как говорилось в главе 13, команда `using` автоматически вызывает `Dispose` для объекта, созданного в круглых скобках за ключевым словом `using`. Это важно, потому что объекты `Graphics` являются ограниченным ресурсом. Вызов `Dispose` для объекта `Graphics` обеспечивает возвращение его ресурсов системе для повторного использования.

## 14.12. Обработка событий клавиатуры

События клавиатуры происходят при нажатии и отпускании клавиш. Такие события могут обрабатываться для любых элементов управления, наследующих от `System.Windows.Forms.Control`. Три важнейших события — `KeyPress`, `KeyUp` и `KeyDown`. Событие `KeyPress` происходит при нажатии клавиши, представляющей ASCII-символ. Конкретная клавиша определяется при помощи свойства `KeyChar` аргумента `KeyPressEventArgs` обработчика события.

Событие `KeyPress` не указывает, были ли нажаты клавиши-модификаторы (`Shift`, `Alt` и `Ctrl`) при нажатии события клавиатуры. Если эта информация для вас важна, используйте события `KeyUp` или `KeyDown`. Аргумент `KeyEventArgs` каждого из этих событий содержит информацию о клавишах-модификаторах. На ил. 14.39 приведена основная информация о событиях клавиатуры. Некоторые свойства возвращают значения из перечисления `Keys` с константами, определяющими различные клавиши на клавиатуре. Как и перечисление `FontStyle` (раздел 14.7), перечисление `Keys` представляется набором битов, так что константы перечисления могут объединяться поразрядными операторами для определения одновременного нажатия *нескольких клавиш*.

События клавиатуры и аргументы событий	Описание
События клавиатуры с аргументом типа <code>KeyEventArgs</code>	
<code>KeyDown</code>	Генерируется при исходном нажатии клавиши
<code>KeyUp</code>	Генерируется при отпускании клавиши

**Ил. 14.39.** События клавиатуры и аргументы событий (продолжение ↗)

События клавиатуры и аргументы событий	Описание
Событие клавиатуры с аргументом типа <code>KeyPressEventArgs</code>	
<code>KeyPress</code>	Генерируется при нажатии клавиши (после <code>KeyDown</code> , но до <code>KeyUp</code> )
Свойства класса <code>KeyPressEventArgs</code>	
<code>KeyChar</code>	Возвращает ASCII-символ для нажатой клавиши
Свойство класса <code>KeyEventArgs</code>	
<code>Alt</code>	Указывает, была ли нажата клавиша <code>Alt</code>
<code>Control</code>	Указывает, была ли нажата клавиша <code>Ctrl</code>
<code>Shift</code>	Указывает, была ли нажата клавиша <code>Shift</code>
<code>KeyCode</code>	Возвращает код клавиши в виде значения из перечисления <code>Keys</code> (без информации о состоянии клавиш-модификаторов). Используется для проверки нажатия конкретных клавиш
<code>KeyData</code>	Возвращает код клавиши вместе с данными состояния модификаторов в виде значения из перечисления <code>Keys</code> . Свойство содержит полную информацию о нажатой клавише
<code>KeyValue</code>	Возвращает код клавиши в виде значения <code>int</code> (вместо значения из перечисления <code>Keys</code> ). Свойство используется для получения числового представления нажатой клавиши. Значение <code>int</code> называется кодом виртуальной клавиши <code>Windows</code>
<code>Modifiers</code>	Возвращает значение <code>Keys</code> , обозначающее состояние нажатых клавиш-модификаторов ( <code>Alt</code> , <code>Ctrl</code> и <code>Shift</code> ). Свойство используется только для определения информации о клавишах-модификаторах

**Ил. 14.39.** События клавиатуры и аргументы событий (окончание)

В программе на ил. 14.40 продемонстрировано использование обработчиков событий клавиатуры для вывода информации о нажатой клавише. Программа представляет собой форму с двумя надписями: на одной выводится информация о нажатой клавише, а на другой — информация о состоянии клавиш-модификаторов.

```

1 // Ил. 14.40: KeyDemo.cs
2 // Вывод информации о нажатой клавише.
3 using System;
4 using System.Windows.Forms;
5
6 namespace KeyDemo
7 {
8     // Форма для вывода информации о нажатых клавишах

```

**Ил. 14.40.** Демонстрация событий клавиатуры (продолжение ↗)

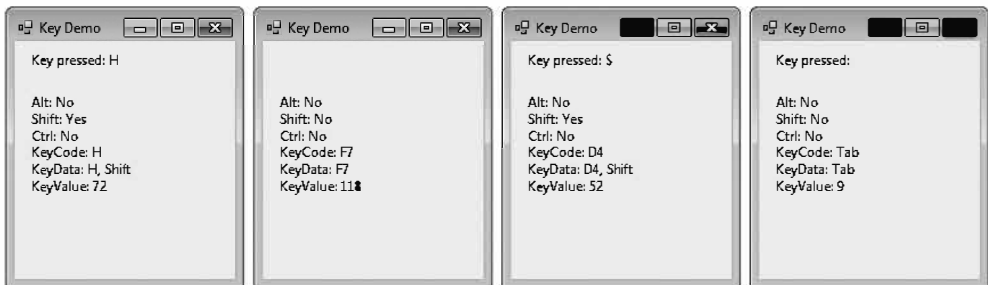


```

9  public partial class KeyDemo : Form
10 {
11     // Конструктор по умолчанию
12     public KeyDemo()
13     {
14         InitializeComponent();
15     } // Конец конструктора
16
17     // Вывод нажатого символа с использованием KeyChar
18     private void KeyDemo_KeyPress(
19         object sender, KeyPressEventArgs e )
20     {
21         charLabel.Text = "Key pressed: " + e.KeyChar;
22     } // Конец метода KeyDemo_KeyPress
23
24     // Вывод состояния модификаторов и различных свойств
25     private void KeyDemo_KeyDown( object sender, KeyEventArgs e )
26     {
27         keyInfoLabel.Text =
28             "Alt: " + ( e.Alt ? "Yes" : "No" ) + '\n' +
29             "Shift: " + ( e.Shift ? "Yes" : "No" ) + '\n' +
30             "Ctrl: " + ( e.Control ? "Yes" : "No" ) + '\n' +
31             "KeyCode: " + e.KeyCode + '\n' +
32             "KeyData: " + e.KeyData + '\n' +
33             "KeyValue: " + e.KeyValue;
34     } // Конец метода KeyDemo_KeyDown
35
36     // Сброс содержимого надписей при отпускании клавиши
37     private void KeyDemo_KeyUp( object sender, KeyEventArgs e )
38     {
39         charLabel.Text = "";
40         keyInfoLabel.Text = "";
41     } // Конец метода KeyDemo_KeyUp
42 } // Конец класса KeyDemo
43 } // Конец пространства имен KeyDemo

```

а) Нажата клавиша H      б) Нажата клавиша F7      в) Нажата клавиша \$      г) Нажата клавиша Tab



**Ил. 14.40.** Демонстрация событий клавиатуры (окончание)

В поле `charLabel` выводится символьный эквивалент нажатой клавиши, а в поле `keyInfoLabel` — информация, относящаяся к нажатой клавише. Поскольку события `KeyDown` и `KeyPress` передают разную информацию, форма (`KeyDemo`) обрабатывает оба события.

Обработчик события `KeyPress` (строки 18–22) использует свойство `KeyChar` объекта `KeyPressEventArgs`. Нажатая клавиша возвращается в виде значения `char`, которое отображается в `charLabel` (строка 21). Если нажатая клавиша не является ASCII-символом, то событие `KeyPress` не происходит, а в `charLabel` никакой текст не отображается. ASCII представляет собой стандартный формат кодирования букв, цифр, знаков препинания и других символов. ASCII не поддерживает ни функциональные клавиши (такие, как F1), ни клавиши-модификаторы (Alt, Ctrl и Shift).

Обработчик `KeyDown` (строки 25–34) выводит информацию из объекта `KeyEventArgs`. Обработчик проверяет состояние клавиш Alt, Shift и Ctrl по свойствам `Alt`, `Shift` и `Control`, каждое из которых возвращает логическое значение: `true`, если соответствующая клавиша нажата, `false` — в противном случае. Затем обработчик события выводит значения свойств `KeyCode`, `KeyData` и `KeyValue`.

Свойство `KeyCode` возвращает значение из перечисления `Keys` (строка 31). Оно возвращает нажатую клавишу, но не предоставляет никакой информации о клавишах-модификаторах. Таким образом, и прописная, и строчная буква «а» представляются как клавиша A.

Свойство `KeyData` (строка 32) также возвращает значение из перечисления `Keys`, но это свойство включает данные о клавишах-модификаторах. Таким образом, при вводе символа «А» информация `KeyData` сообщает о нажатии клавиши A и клавиши Shift. Наконец, свойство `KeyValue` (строка 33) возвращает значение `int`, представляющее нажатую клавишу, — *код клавиши*. Это значение пригодится при проверке клавиш, не входящих в кодировку ASCII (например, F12).

Обработчик события `KeyUp` (строки 37–41) стирает содержимое обеих надписей при отпускании клавиши. Как видно из результатов приложения, клавиши, не входящие в ASCII, в `charLabel` не отображаются, потому что событие `KeyPress` для них не генерируется. Например, в `charLabel` при нажатии клавиш F7 или Tab не отображается никакой текст (см. ил. 14.40, б и г). Однако событие `KeyDown` при этом все равно генерируется, и в `keyInfoLabel` выводится информация о нажатой клавише. Для проверки нажатий конкретных клавиш можно использовать перечисление `Keys`, сравнивая `KeyCode` нажатой клавиши со значениями из `Keys`.



## АРХИТЕКТУРНОЕ РЕШЕНИЕ 14.2

Чтобы элемент управления реагировал на нажатие конкретной клавиши (например, Enter), обработайте событие клавиатуры и проверьте нажатие нужной клавиши. Чтобы при нажатии клавиши Enter на форме происходил щелчок на кнопке, задайте свойство `AcceptButton` формы.

По умолчанию событие клавиатуры обрабатывается элементом управления, которому в настоящий момент принадлежит фокус ввода. Иногда бывает удобнее доверить обработку этих событий форме. Для этого свойству `KeyPreview` формы задается значение `true`, в результате чего форма получает события клавиатуры до их передачи другому элементу управления. Например, нажатие клавиши вызовет обработчик `KeyPress` формы — при том, что фокус принадлежит элементу внутри формы, а не самой форме.

## 14.13. Итоги

В этой главе читатель познакомился с некоторыми часто используемыми элементами управления. Мы подробно рассмотрели механизм обработки событий и научились создавать обработчики событий. Вы узнали, как использовать делегатов для связывания обработчиков событий с событиями конкретных элементов и как использовать свойства элементов управления и Visual Studio для определения макета графического интерфейса. Далее рассматривались конкретные основные элементы управления, начиная с надписей, кнопок и текстовых полей. Вы узнали, как использовать контейнеры `GroupBox` и `Panel` для группировки элементов. Мы рассмотрели флажки и переключатели — специализированные кнопки, позволяющие пользователям выбирать варианты из предложенного списка. Были представлены примеры вывода изображений в графических полях, отображения подсказок с использованием компонентов `ToolTip` и задания диапазонов числовых значений с использованием элементов управления `NumericUpDown`. В завершающей части главы рассматривалась обработка событий мыши и клавиатуры.

# 15 Графический интерфейс и Windows Forms: часть 2

## 15.1. Введение

В этой главе мы продолжим изучать графические интерфейсы Windows Forms. Глава начинается с меню, предназначенных для отображения логически упорядоченных команд (или вариантов). Вы научитесь создавать меню при помощи инструментов, предоставляемых Visual Studio. Затем будут рассмотрены возможности ввода и отображения даты и времени с использованием элементов управления `MonthCalendar` и `DateTimePicker`. Мы также рассмотрим `LinkLabel` — мощные компоненты, позволяющие перейти к одному из нескольких возможных мест (например, файлу на текущей машине или веб-странице) простым щелчком кнопки мыши.

Также будут рассмотрены возможности работы со списками в элементах управления `ListBox` и объединения флажков в `CheckedListBox`. Мы займемся созданием раскрывающихся списков с использованием элементов управления `ComboBox` и иерархического отображения данных в элементах `TreeView`. Далее будут представлены два других важных компонента графических интерфейсов — вкладки и окна с многодокументным интерфейсом MDI. С их помощью можно строить реальные приложения со сложными интерфейсами.

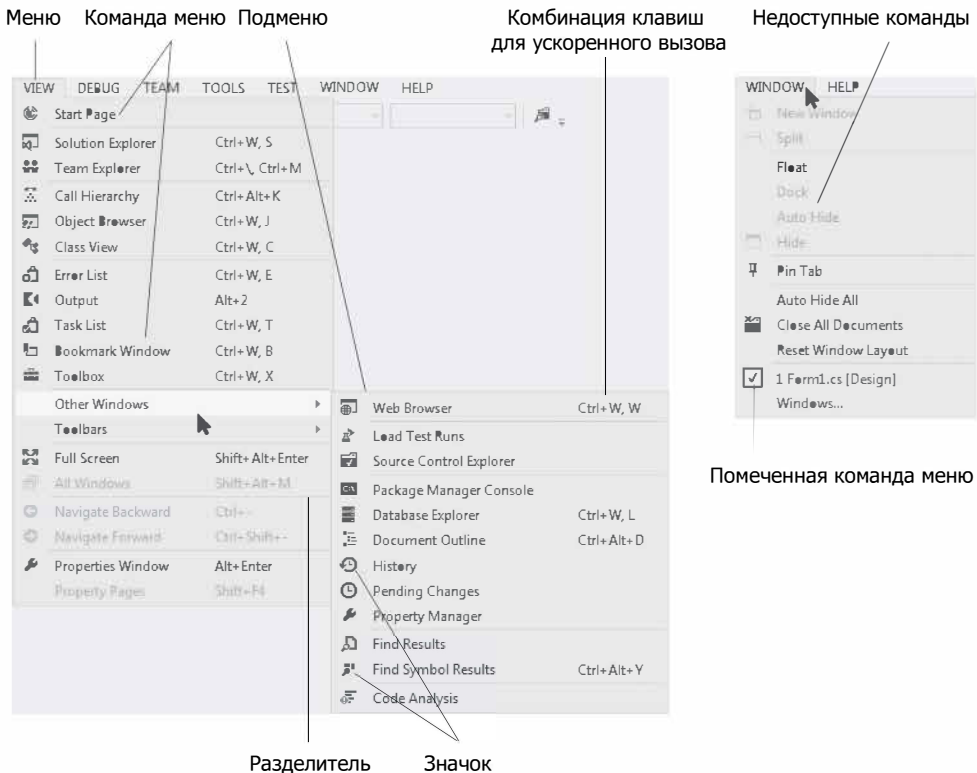
Visual Studio предоставляет много разных компонентов, часть из которых описана в этой (и предыдущей) главах. Вы также можете разрабатывать пользовательские элементы управления и добавлять их на панель элементов, как будет показано в последнем примере этой главы. Приемы и инструменты, описанные в этой главе, закладывают основу для создания более серьезных графических интерфейсов и пользовательских элементов управления.

## 15.2. Меню

*Меню* предназначены для группировки взаимосвязанных команд в приложениях Windows Forms. Хотя выбор команд зависит от конкретной программы, некоторые команды — такие, как *Open* и *Save* — присутствуют во многих приложениях. Меню являются неотъемлемой частью графических интерфейсов, потому что они позволяют группировать команды без загромождения интерфейса.

На ил. 15.1 изображено меню Visual C# IDE с *командами меню* (также называемыми пунктами меню) и *подменю* (меню внутри меню). Меню верхнего уровня выводится в левой части иллюстрации, а подменю и команды меню выводятся справа. Меню, содержащее команду меню, называется *родительским меню* этой команды. Команда меню, содержащая подменю, считается родителем для этого подменю.

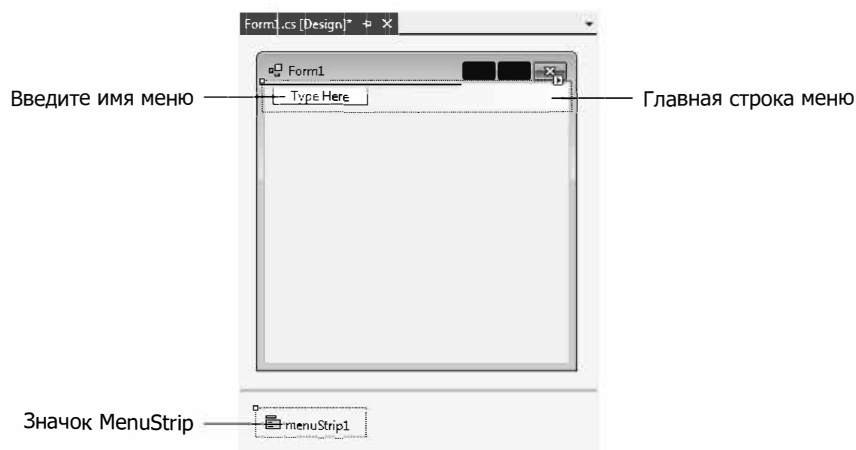
Меню могут назначаться *Alt-комбинации* клавиш, состоящие из клавиши Alt и буквы, подчеркнутой в названии меню, — например, комбинация Alt+F обычно открывает меню File. Командам меню тоже могут назначаться комбинации клавиш для ускоренного вызова (комбинации Ctrl, Shift, Alt, F1, F2, букв и т. д.). Некоторые команды меню выводятся с пометками, которые обычно означают, что несколько команд меню могут быть выбраны одновременно.



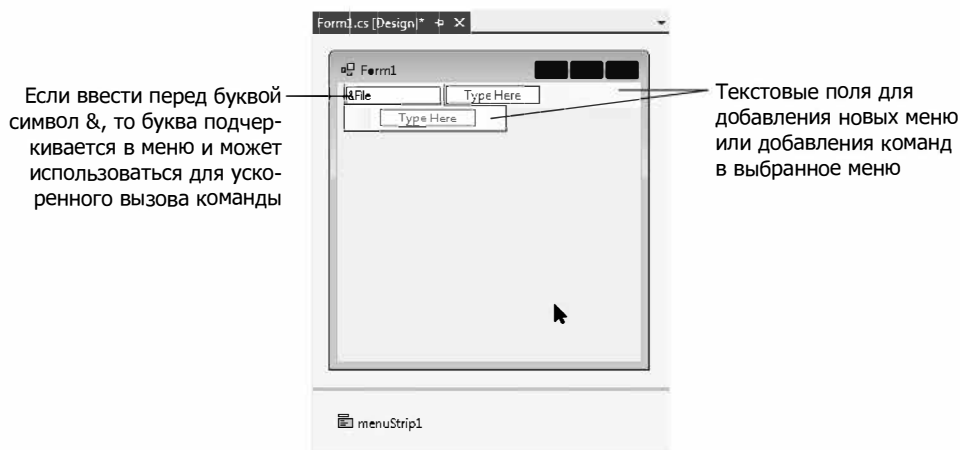
**Ил. 15.1.** Меню, подменю и команды меню

Чтобы создать меню, откройте панель элементов и перетащите элемент управления `MenuStrip` на форму. В верхней части формы (под заголовком) отображается строка меню, а в области компонентов появляется значок `MenuStrip`. Чтобы выбрать компонент `MenuStrip`, щелкните на значке. Открывается режим конструктора для создания и редактирования меню вашего приложения. Меню, как и другие элементы управления, обладают свойствами и событиями, с которыми можно работать в окне свойств.

Чтобы добавить в меню команды, щелкните на текстовом поле `Type Here` (ил. 15.2) и введите имя команды меню. При этом в меню включается новый компонент типа `ToolStripMenuItem`. После нажатия клавиши `Enter` имя команды добавляется в меню, а на экране появляются новые поля `Type Here`, позволяющие добавлять элементы под исходной командой меню или в стороне от нее (ил. 15.3).

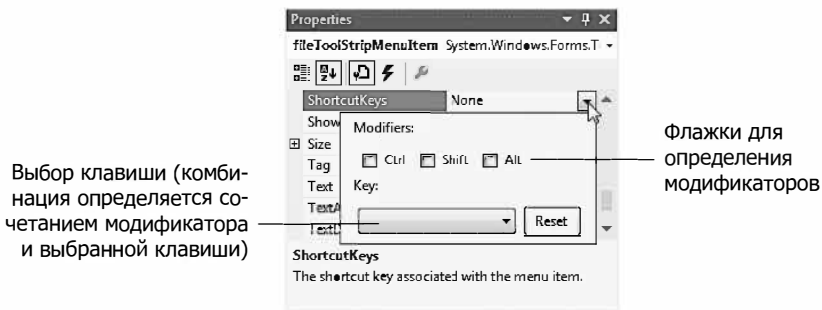


**Ил. 15.2.** Редактирование меню в Visual Studio



**Ил. 15.3.** Добавление компонентов `ToolStripMenuItem` в `MenuStrip`

Чтобы создать комбинацию клавиш для ускоренного вызова команд меню, введите & перед нужным символом. Например, чтобы создать команду меню File с подчеркнутой буквой F, введите строку &File. Чтобы символ & выводился в меню, введите &&. Другие сочетания клавиш ускоренного доступа (например, Ctrl + F9) назначаются командам меню при помощи свойства `ShortcutKeys` соответствующего компонента `ToolStripMenuItem`. Для этого щелкните на стрелке рядом со свойством `ShortcutKeys` в окне свойств. В открывшемся окне (ил. 15.4) выберите комбинацию клавиш при помощи флажков и раскрывающегося списка. Когда это будет сделано, щелкните в любой точке экрана. Чтобы скрыть комбинации клавиш из меню, задайте свойству `ShowShortcutKeys` значение `false`; свойство `ShortcutKeyDisplayString` позволяет настроить формат отображения комбинаций клавиш в командах меню.



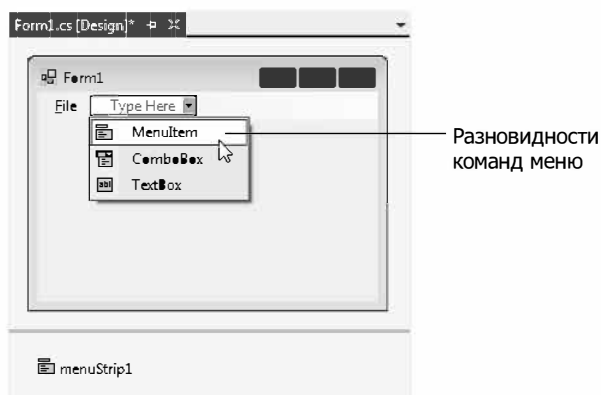
**Ил. 15.4.** Определение комбинаций клавиш для команд меню



### КРАСИВО И УДОБНО 15.1

Кнопкам можно назначать комбинации клавиш. Для этого введите & непосредственно перед нужным символом в тексте кнопки. Чтобы активизировать кнопку комбинацией клавиш во время выполнения приложения, нажмите Alt с подчеркнутым символом. Если подчеркивания не видны во время выполнения, нажмите клавишу Alt.

Чтобы удалить команду меню, выделите ее мышью и нажмите клавишу Delete. Для группировки команд меню используются разделители; чтобы вставить разделитель, щелкните правой кнопкой мыши в меню и выберите команду Insert ► Separator или введите «-» вместо текста команды меню. Возможно, вы заметили, что при добавлении команд в режиме конструктора перед вводом текста новой команды появляется раскрывающийся список. Щелчок на кнопке со стрелкой (ил. 15.5) позволяет выбрать тип добавляемой команды — `MenuItem` (тип `ToolStripMenuItem`, используется по умолчанию), `ComboBox` (тип `ToolStripComboBox`) или `TextBox` (тип `ToolStripTextBox`). Мы сосредоточимся на компонентах `ToolStripMenuItem`. [Примечание: для команд меню, не находящихся на верхнем уровне, в этом списке появляется четвертая команда для вставки разделителя.]



Ил. 15.5. Параметры команд меню

При выборе компонента `ToolStripMenuItem` генерируется событие `Click`. Чтобы создать пустой обработчик `Click`, сделайте двойной щелчок на команде меню в режиме конструктора. Чаще всего при обработке этих событий выводятся диалоговые окна и задаются свойства. Часто используемые свойства меню и событие перечислены на ил. 15.6.



### КРАСИВО И УДОБНО 15.2

После имен команд меню, требующих ввода дополнительной информации от пользователя (как правило, в диалоговом окне), принято ставить многоточие (...) — например, `Save As...`. За именами команд меню, немедленно выполняющих свое действие без ввода дополнительной информации (например, `Save`), многоточие не ставится.

Свойства и событие <code>MenuStrip</code> и <code>ToolStripMenuItem</code>	Описание
<b>Свойство <code>MenuStrip</code></b>	
<code>RightToLeft</code>	Текст выводится справа налево (для языков с соответствующей письменностью)
<b>Свойства <code>ToolStripMenuItem</code></b>	
<code>Checked</code>	Указывает, находится ли команда меню в помеченном состоянии. По умолчанию используется значение <code>false</code> (команда не помечена)
<code>CheckOnClick</code>	Указывает, должна ли устанавливаться/сниматься пометка команды меню при щелчке
<code>ShortcutKeyDisplayString</code>	Задаёт текст, который должен отображаться рядом с командой для ускоренного вызова. Если значение пусто, отображаются названия клавиш; в противном случае выводится заданный текст

Ил. 15.6. Свойства и событие `MenuStrip` и `ToolStripMenuItem` (продолжение ↗)



Свойства и событие MenuStrip и ToolStripMenuItem	Описание
ShortcutKeys	Задаёт комбинацию клавиш для команды меню (например, нажатие <Ctrl>+F9 эквивалентно щелчку на команде меню)
ShowShortcutKeys	Указывает, должна ли выводиться комбинация клавиш рядом с командой меню. По умолчанию используется значение true, при котором комбинация выводится
Text	Задаёт текст команды меню. Чтобы создать Alt-комбинацию клавиш для ускоренного вызова, поставьте перед символом знак & (например, &File для подчеркивания F в команде File)
Часто используемое событие ToolStripMenuItem	
Click	Генерируется при щелчке на команде или использовании комбинации клавиш. При двойном щелчке на меню в режиме конструктора создается пустой обработчик этого события

**Ил. 15.6.** Свойства и событие MenuStrip и ToolStripMenuItem (окончание)

Класс MenuTestForm (ил. 15.7) создает простое меню на форме. Форма содержит меню верхнего уровня File с командами About (выводит окно MessageBox) и Exit (завершает выполнение программы). Программа также включает меню Format с командами, изменяющими формат текста надписи. Меню Format содержит подменю Color и Font для изменения цвета и шрифта текста надписи.

### Создание графического интерфейса

Начните с перетаскивания компонента MenuStrip с панели элементов на форму. Используйте режим конструктора для создания структуры меню, показанной в примерах вывода. Меню File (fileToolStripMenuItem) содержит команды About (aboutToolStripMenuItem) и Exit (exitToolStripMenuItem); меню Format (formatToolStripMenuItem) содержит два подменю. Первое подменю, Color (colorToolStripMenuItem), содержит команды Black (blackToolStripMenuItem), Blue (blueToolStripMenuItem), Red (redToolStripMenuItem) и Green (greenToolStripMenuItem). Второе подменю, Font (fontToolStripMenuItem), содержит команды Times New Roman (timesToolStripMenuItem), Courier (courierToolStripMenuItem), Comic Sans (comicToolStripMenuItem), разделитель (dashToolStripMenuItem), Bold (boldToolStripMenuItem) и Italic (italicToolStripMenuItem).

```

1 // Ил. 15.7: MenuTestForm.cs
2 // Использование меню для изменения цвета и начертания шрифта.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6
7 namespace MenuTest

```

**Ил. 15.7.** Меню для изменения цвета и начертания шрифта (продолжение ↗)

```
8  {
9  // Форма содержит меню для изменения цвета и начертания
10 // шрифта текста, выводимого в надписи
11 public partial class MenuTestForm : Form
12 {
13     // Конструктор
14     public MenuTestForm()
15     {
16         InitializeComponent();
17     } // Конец конструктора
18
19     // При выборе команды About выбирается MessageBox
20     private void aboutToolStripMenuItem_Click(
21         object sender, EventArgs e )
22     {
23         MessageBox.Show( "This is an example\nof using menus.", "About",
24             MessageBoxButtons.OK, MessageBoxIcon.Information );
25     } // Конец метода aboutToolStripMenuItem_Click
26
27     // При выборе команды Exit программа завершается
28     private void exitToolStripMenuItem_Click(
29         object sender, EventArgs e )
30     {
31         Application.Exit();
32     } // Конец метода exitToolStripMenuItem_Click
33
34     // Сброс пометок для команд меню Color
35     private void ClearColor()
36     {
37         // Сброс всех пометок
38         blackToolStripMenuItem.Checked = false;
39         blueToolStripMenuItem.Checked = false;
40         redToolStripMenuItem.Checked = false;
41         greenToolStripMenuItem.Checked = false;
42     } // Конец метода ClearColor
43
44     // Обновление состояния меню и переход на черный цвет текста
45     private void blackToolStripMenuItem_Click(
46         object sender, EventArgs e )
47     {
48         // Сброс пометок команд Color
49         ClearColor();
50
51         // Выбор черного цвета
52         displayLabel.ForeColor = Color.Black;
53         blackToolStripMenuItem.Checked = true;
54     } // Конец метода blackToolStripMenuItem_Click
55
56     // Обновление состояния меню и переход на синий цвет текста
57     private void blueToolStripMenuItem_Click(
58         object sender, EventArgs e )
59     {
60         // Сброс пометок команд Color
```

**Ил. 15.7.** Меню для изменения цвета текста и шрифта (продолжение )

```

61         ClearColor();
62
63         // Выбор синего цвета
64         displayLabel.ForeColor = Color.Blue;
65         blueToolStripMenuItem.Checked = true;
66     } // Конец метода blueToolStripMenuItem_Click
67
68     // Обновление состояния меню и переход на красный цвет текста
69     private void redToolStripMenuItem_Click(
70         object sender, EventArgs e )
71     {
72         // Сброс пометок команд Color
73         ClearColor();
74
75         // Выбор красного цвета
76         displayLabel.ForeColor = Color.Red;
77         redToolStripMenuItem.Checked = true;
78     } // Конец метода redToolStripMenuItem_Click
79
80     // Обновление состояния меню и переход на зеленый цвет текста
81     private void greenToolStripMenuItem_Click(
82         object sender, EventArgs e )
83     {
84         // Сброс пометок команд Color
85         ClearColor();
86
87         // Выбор зеленого цвета
88         displayLabel.ForeColor = Color.Green;
89         greenToolStripMenuItem.Checked = true;
90     } // Конец метода greenToolStripMenuItem_Click
91
92     // Сброс пометок команд Font
93     private void ClearFont()
94     {
95         // Сброс всех пометок
96         timesToolStripMenuItem.Checked = false;
97         courierToolStripMenuItem.Checked = false;
98         comicToolStripMenuItem.Checked = false;
99     } // Конец метода ClearFont
100
101     // Обновление состояния меню и выбор шрифта Times New Roman
102     private void timesToolStripMenuItem_Click(
103         object sender, EventArgs e )
104     {
105         // Сброс пометок команд Font
106         ClearFont();
107
108         // Назначение шрифта Times New Roman
109         timesToolStripMenuItem.Checked = true;
110         displayLabel.Font = new Font( "Times New Roman", 14,
111             displayLabel.Font.Style );
112     } // Конец метода timesToolStripMenuItem_Click
113
114     // Обновление состояния меню и выбор шрифта Courier

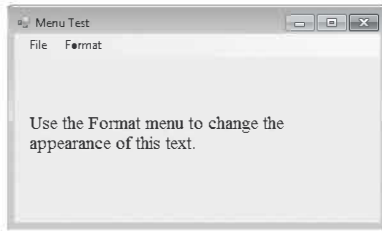
```

**Ил. 15.7.** Меню для изменения цвета текста и шрифта (продолжение ➤)

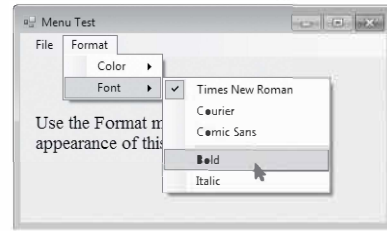
```
115     private void courierToolStripMenuItem_Click(  
116         object sender, EventArgs e )  
117     {  
118         // Сброс пометок команд Font  
119         ClearFont();  
120  
121         // Назначение шрифта Courier  
122         courierToolStripMenuItem.Checked = true;  
123         displayLabel.Font = new Font( "Courier", 14,  
124             displayLabel.Font.Style );  
125     } // Конец метода courierToolStripMenuItem_Click  
126  
127     // Обновление состояния меню и выбор шрифта Comic Sans MS  
128     private void comicToolStripMenuItem_Click(  
129         object sender, EventArgs e )  
130     {  
131         // Сброс пометок команд Font  
132         ClearFont();  
133  
134         // Назначение шрифта Comic Sans  
135         comicToolStripMenuItem.Checked = true;  
136         displayLabel.Font = new Font( "Comic Sans MS", 14,  
137             displayLabel.Font.Style );  
138     } // Конец метода comicToolStripMenuItem_Click  
139  
140     // Переключение пометки и полужирного начертания  
141     private void boldToolStripMenuItem_Click(  
142         object sender, EventArgs e )  
143     {  
144         // Переключение пометки  
145         boldToolStripMenuItem.Checked = !boldToolStripMenuItem.Checked;  
146  
147         // Переключение полужирного начертания с сохранением остальных стилей  
148         displayLabel.Font = new Font( displayLabel.Font,  
149             displayLabel.Font.Style ^ FontStyle.Bold );  
150     } // Конец метода boldToolStripMenuItem_Click  
151  
152     // Переключение пометки и курсивного начертания  
153     private void italicToolStripMenuItem_Click(  
154         object sender, EventArgs e )  
155     {  
156         // Переключение пометки  
157         italicToolStripMenuItem.Checked =  
158             !italicToolStripMenuItem.Checked;  
159  
160         // Переключение курсивного начертания с сохранением остальных стилей  
161         displayLabel.Font = new Font( displayLabel.Font,  
162             displayLabel.Font.Style ^ FontStyle.Italic );  
163     } // Конец метода italicToolStripMenuItem_Click  
164 } // Конец класса MenuTestForm  
165 } // Конец пространства имен MenuTest
```

**Ил. 15.7.** Меню для изменения цвета текста и шрифта (продолжение ↗)

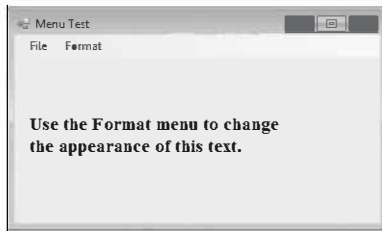
а) Исходное состояние



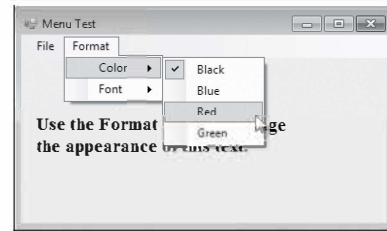
б) Выбор команды меню Bold



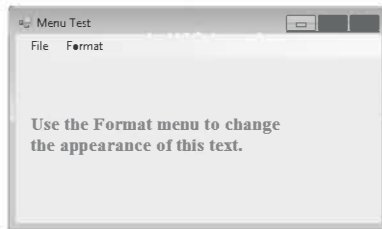
в) Приложение после выбора полужирного шрифта



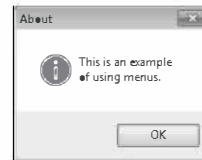
г) Выбор команды меню Red



д) Приложение после выбора красного цвета текста



е) Диалоговое окно, выводимое командой File ► About

**Ил. 15.7.** Меню для изменения цвета текста и шрифта (окончание)

### Обработчик событий Click команд меню About и Exit

При щелчке на команде меню About в меню File выводится окно сообщения (строки 20–25). Команда меню Exit закрывает приложение статическим методом Exit класса Application (строка 31). Статические методы класса Application управляют выполнением программы; метод Exit завершает приложение.

### События подменю Color

Команды меню Color (Black, Blue, Red и Green) являются взаимоисключающими — в любой момент времени пользователь может выбрать только одну из них (вскоре вы увидите, как это было сделано). Чтобы показать, что команда из меню Color выбрана, мы задаем ее свойству Checked значение true. Слева от команды меню появляется пометка.

С каждой командой меню Color связывается обработчик события Click. Для команды Black это метод с именем blackToolStripMenuItem\_Click (строки 45–54).

Обработчиками событий других команд меню Color — Blue, Red и Green — являются методы `blueToolStripMenuItem_Click` (строки 57–66), `redToolStripMenuItem_Click` (строки 69–78) и `greenToolStripMenuItem_Click` (строки 81–90) соответственно. Так как команды меню Color должны быть взаимоисключающими, каждый обработчик события вызывает метод `ClearColor` (строки 35–42) перед тем, как задавать соответствующему свойству `Checked` значение `true`. Метод `ClearColor` задает свойству `Checked` каждого компонента `ToolStripMenuItem` из меню Color значение `false`, делая невозможной одновременную пометку нескольких команд меню. В режиме конструктора мы задаем свойству `Checked` команды меню Black значение `true`, потому что при запуске программы текст на форме должен быть черным.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 15.1

Компонент `MenuStrip` не следит за состоянием взаимоисключающих команд, даже если свойство `Checked` истинно. Вы должны самостоятельно запрограммировать это поведение.

### События подменю Font

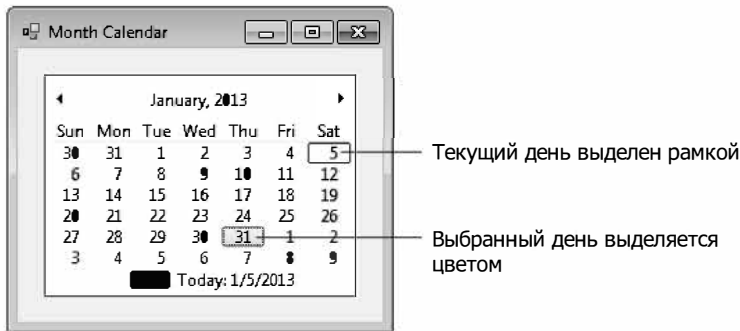
Меню Font содержит три команды для выбора шрифтов (Courier, Times New Roman и Comic Sans) и две команды для выбора начертания (Bold и Italic). Мы добавили разделитель, чтобы показать, что это две разные группы команд. Объект Font в любой момент времени может задать только один шрифт, но начертания могут задаваться одновременно (то есть шрифт может быть полужирным и курсивным одновременно). Рядом с командами меню Font будут отображаться пометки; как и в случае с меню Color, мы должны обеспечить взаимоисключающую пометку этих команд в обработчиках событий.

Обработчиками событий команд меню Times New Roman, Courier и Comic Sans являются методы `timesToolStripMenuItem_Click` (строки 102–112), `courierToolStripMenuItem_Click` (строки 115–125) и `comicToolStripMenuItem_Click` (строки 128–138) соответственно. Эти обработчики похожи на обработчики команд Color. В каждом из них свойства `Checked` всех команд меню Font сбрасываются вызовом метода `ClearFont` (строки 93–99), после чего свойству `Checked` команды, инициировавшей событие, задается значение `true`; так обеспечивается взаимоисключаемость команд меню Font. В режиме конструктора мы изначально задаем свойство `Checked` команды Times New Roman равным `true`, потому что этот шрифт должен использоваться для вывода исходного текста на форме. Обработчики событий команд Bold и Italic (строки 141–163) используют поразрядный исключающий оператор ИЛИ (^) для комбинирования стилей (см. главу 14).

## 15.3. Элемент управления MonthCalendar

Во многих приложениях выполняются вычисления с датой и временем. .NET Framework предоставляет два элемента для получения информации даты и времени — `MonthCalendar` и `DateTimePicker` (см. раздел 15.4).

Элемент управления MonthCalendar (ил. 15.8) выводит на форме календарь на месяц. Пользователь может выбрать дату из текущего месяца или перейти к другому месяцу при помощи кнопок со стрелками. Выбранная дата выделяется цветом. Чтобы выбрать несколько дат в календаре, пользователь щелкает на них, удерживая клавишу Shift. Событием по умолчанию для этого элемента управления является событие `DateChanged`, которое генерируется при выборе новой даты. Свойства позволяют изменить внешний вид календаря, количество одновременно выбираемых дат, а также диапазон дат, которые могут быть выбраны пользователем. Основные свойства и часто используемое событие MonthCalendar перечислены на ил. 15.9.



**Ил. 15.8.** Элемент управления MonthCalendar

Свойства и событие MonthCalendar	Описание
Свойства MonthCalendar	
<code>FirstDayOfWeek</code>	Первый день каждой недели, отображаемой в календаре
<code>MaxDate</code>	Последняя дата, которая может быть выбрана пользователем
<code>MaxSelectionCount</code>	Максимальное количество одновременно выбираемых дат
<code>MinDate</code>	Первая дата, которая может быть выбрана пользователем
<code>MonthlyBoldedDates</code>	Массив дат, которые будут выделяться в календаре жирным шрифтом
<code>SelectionEnd</code>	Последняя из дат, выбранных пользователем
<code>SelectionRange</code>	Даты, выбранные пользователем
<code>SelectionStart</code>	Первая из дат, выбранных пользователем
Часто используемое событие MonthCalendar	
<code>DateChanged</code>	Генерируется при выборе даты в календаре

**Ил. 15.9.** Свойства и событие MonthCalendar

## 15.4. Элемент управления DateTimePicker

Элемент управления `DateTimePicker` (см. результат программы на ил. 15.11) похож на `MonthCalendar`, но он выводит календарь при нажатии кнопки со стрелкой. Элемент `DateTimePicker` может использоваться для получения от пользователя информации о дате и времени. В свойстве `Value` элемента управления `DateTimePicker` хранится объект `DateTime`, который всегда содержит информацию как даты, так и времени. Для получения от объекта `DateTime` информации даты используется свойство `Date`, а для получения информации времени — свойство `TimeOfDay`.

Элемент управления `DateTimePicker` также предоставляет больше возможностей для настройки, чем `MonthCalendar`, — он поддерживает больше свойств для изменения оформления и поведения раскрывающегося календаря. Свойство `Format` задает возможности выбора с использованием значений из перечисления `DateTimePickerFormat`. Перечисление содержит значения `Long` (вывод даты в длинном формате вида `Thursday, July 10, 2013`), `Short` (вывод даты в коротком формате вида `7/10/2013`), `Time` (вывод времени в формате вида `5:31:02 PM`) и `Custom` (формат определяется пользователем). Со значением `Custom` оформление вывода `DateTimePicker` задается свойством `CustomFormat`. Событием по умолчанию данного элемента управления является `ValueChanged`, которое происходит при изменении выбранного значения (даты или времени). Свойства и часто используемое событие `DateTimePicker` представлены на ил. 15.10.

Свойства и событие <code>DateTimePicker</code>	Описание
Свойства <code>DateTimePicker</code>	
<code>CalendarForeColor</code>	Цвет текста в календаре
<code>CalendarMonthBackground</code>	Цвет фона в календаре
<code>CustomFormat</code>	Форматная строка для даты и/или времени, отображаемых в элементе управления
<code>Format</code>	Формат даты и/или времени, отображаемых в элементе управления
<code>MaxDate</code>	Последняя дата, которая может быть выбрана пользователем
<code>MinDate</code>	Первая дата, которая может быть выбрана пользователем
<code>ShowCheckBox</code>	Признак отображения флажка слева от выбранной даты/времени
<code>ShowUpDown</code>	Признак отображения кнопок со стрелками вверх/вниз. Кнопки удобны при использовании <code>DateTimePicker</code> для выбора времени (увеличение/уменьшение часов, минут и секунд)
<code>Value</code>	Дата, выбранная пользователем
Часто используемое событие <code>DateTimePicker</code>	
<code>ValueChanged</code>	Генерируется при изменении свойства <code>Value</code> , в том числе и при выборе новой даты/времени пользователем

Ил. 15.10. Свойства и событие `DateTimePicker`



В листинге на ил. 15.11 продемонстрировано использование DateTimePicker для выбора времени отправки товара. Эта функциональность используется многими компаниями — например, службы проката DVD указывают день отправки и примерное время доставки заказчику. Пользователь выбирает день отправки, в приложении выводится примерная дата получения — через два дня после отправки, три дня с воскресеньем (почта по воскресеньям не доставляется).

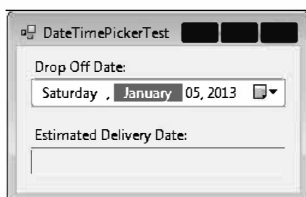
```

1  // Ил. 15.11: DateTimePickerForm.cs
2  // Использование элемента управления DateTimePicker.
3  using System;
4  using System.Windows.Forms;
5
6  namespace DateTimePickerTest
7  {
8      // Форма позволяет выбрать дату отправки с использованием элемента
9      // DateTimePicker и выводит примерную дату доставки.
10     public partial class DateTimePickerForm : Form
11     {
12         // Конструктор
13         public DateTimePickerForm()
14         {
15             InitializeComponent();
16         } // Конец конструктора
17
18         private void dateTimePickerDropOff_ValueChanged(
19             object sender, EventArgs e )
20         {
21             DateTime dropOffDate = dateTimePickerDropOff.Value;
22
23             // Если пересылка приходится на воскресенье, прибавить день
24             if ( dropOffDate.DayOfWeek == DayOfWeek.Friday ||
25                 dropOffDate.DayOfWeek == DayOfWeek.Saturday ||
26                 dropOffDate.DayOfWeek == DayOfWeek.Sunday )
27
28                 // Три дня на доставку
29                 outputLabel.Text =
30                     dropOffDate.AddDays( 3 ).ToLongDateString();
31             else
32                 // Иначе примерный срок доставки составляет два дня
33                 outputLabel.Text =
34                     dropOffDate.AddDays( 2 ).ToLongDateString();
35         } // Конец метода dateTimePickerDropOff_ValueChanged
36
37         private void DateTimePickerForm_Load( object sender, EventArgs e )
38         {
39             // Пользователь не может выбирать дни в прошлом.
40             dateTimePickerDropOff.MinDate = DateTime.Today;
41
42             // Пользователь может выбирать будущие дни в интервале до года.
43             dateTimePickerDropOff.MaxDate = DateTime.Today.AddYears( 1 );
44         } // Конец метода DateTimePickerForm_Load
45     } // Конец класса DateTimePickerForm
46 } // Конец пространства имен DateTimePickerTest

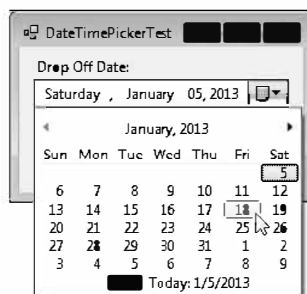
```

**Ил. 15.11.** Использование DateTimePicker (продолжение ☞)

а) При запуске приложения выводится текущая дата



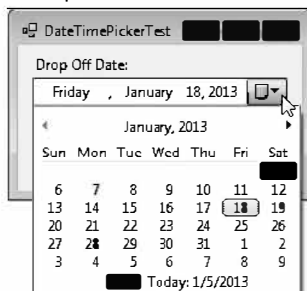
б) Выбор даты отправки



в) Приложение после выбора даты отправки



г) Приложение с текущей и выбранной датой



**Ил. 15.11.** Использование DateTimePicker (окончание)

Свойство `Format` объекта `DateTimePicker` (`dropOffDateTimePicker`) задано равным `Long`, чтобы пользователь мог выбрать дату в приложении. При выборе даты происходит событие `ValueChanged`. Обработчик этого события (строки 18–35) сначала получает выбранную дату из свойства `Value` элемента `DateTimePicker` (строка 21). Строки 24–26 используют свойство `DayOfWeek` структуры `DateTime` для определения дня недели, на который приходится выбранная дата. Дни недели представляются элементами перечисления `DayOfWeek`. Строки 29–30 и 33–34 используют метод `AddDays` класса `DateTime` для увеличения даты на два или три дня соответственно. Полученная дата выводится в формате `Long` с использованием метода `ToLongDateString`.

В этом приложении пользователь не должен выбирать дату отправки, предшествующую текущему дню, или дату, удаленную более чем на год в будущее. Для этого мы инициализируем свойства `MinDate` и `MaxDate` элемента управления `DateTimePicker` при загрузке формы (строки 40 и 43). Свойство `Today` возвращает текущий день, а метод `AddYears` (с аргументом 1) вычисляет дату, которая наступит через год.

Присмотримся к результатам повнимательнее. В начале своей работы приложение выводит текущую дату (ил. 15.11, а). На ил. 15.11, б выбрано 18-е января. На ил. 15.11, в выводится примерная дата доставки 21 января. На ил. 15.11, г видно, что после выбора 18-е января выделяется в календаре.

## 15.5. Элемент управления LinkLabel

Элемент управления `LinkLabel` предназначен для отображения ссылок на другие ресурсы — например, файлы или веб-страницы (ил. 15.12). `LinkLabel` выглядит как подчеркнутый текст (синий шрифт по умолчанию). При наведении на ссылку указатель мыши принимает форму руки, как в случае гиперссылок на веб-страницах. Цвет ссылки может меняться в зависимости от того, посещалась ли она ранее и является ли активной (указатель мыши находится над ссылкой). При щелчке на `LinkLabel` генерируется событие `LinkClicked` (ил. 15.13). Класс `LinkLabel` является производным от `Label` и поэтому наследует всю функциональность `Label`.



**Ил. 15.12.** Элемент управления `LinkLabel` в работающей программе



### КРАСИВО И УДОБНО 15.3

Чтобы показать, что по щелчку на ссылке происходит переход к другому ресурсу (например, веб-странице), рекомендуется использовать `LinkLabel`, хотя другие элементы управления могут решать аналогичные задачи.

Свойства и событие <code>LinkLabel</code>	Описание
Часто используемые свойства	
<code>ActiveLinkColor</code>	Цвет активной ссылки в процессе щелчка. Цвет по умолчанию (обычно красный) задается системой
<code>LinkArea</code>	Свойство определяет, какая часть текста <code>LinkLabel</code> является частью ссылки
<code>LinkBehavior</code>	Поведение ссылки (например, ее внешний вид при наведении указателя мыши)
<code>LinkColor</code>	Исходный цвет ссылки до перехода по ней. Цвет по умолчанию (обычно синий) задается системой
<code>LinkVisited</code>	Если свойство истинно, то ссылка отображается так, словно она уже посещалась (ее цвет заменяется цветом, заданным свойством <code>VisitedLinkColor</code> ). Значение по умолчанию равно <code>false</code>
<code>Text</code>	Текст элемента управления
<code>UseMnemonic</code>	Если свойство истинно, то символ <code>&amp;</code> в свойстве <code>Text</code> интерпретируется как признак комбинации ускоренного вызова (по аналогии с Alt-комбинациями меню)
<code>VisitedLinkColor</code>	Цвет посещенной ссылки. Цвет по умолчанию (обычно фиолетовый) задается системой

**Ил. 15.13.** Свойства и событие `LinkLabel` (продолжение ➤)

Свойства и событие LinkLabel	Описание
Часто используемое событие (аргументы LinkLabelLinkClickedEventArgs)	
LinkClicked	Генерируется при щелчке на ссылке. Является событием по умолчанию при двойном щелчке на элементе управления в режиме конструктора

**Ил. 15.13.** Свойства и событие LinkLabel (окончание)

Класс LinkLabelTestForm (ил. 15.14) использует три элемента LinkLabel для создания ссылок на диск C:, веб-сайт Deitel ([www.deitel.com](http://www.deitel.com)) и приложение Блокнот. Свойства Text объектов cDriveLinkLabel, deitelLinkLabel и notepadLinkLabel описывают назначение каждой ссылки.

```

1  // Ил. 15.14: LinkLabelTestForm.cs
2  // Создание гиперссылок с использованием элементов управления LinkLabel.
3  using System;
4  using System.Windows.Forms;
5
6  namespace LinkLabelTest
7  {
8      // Форма использует объекты LinkLabel для просмотра диска C:\,
9      // загрузки веб-страницы и запуска Блокнота.
10     public partial class LinkLabelTestForm : Form
11     {
12         // Конструктор
13         public LinkLabelTestForm()
14         {
15             InitializeComponent();
16         } // Конец конструктора
17
18         // Просмотр диска C:\
19         private void cDriveLinkLabel_LinkClicked( object sender,
20             LinkLabelLinkClickedEventArgs e )
21         {
22             // Изменение LinkColor после щелчка
23             driveLinkLabel.LinkVisited = true;
24
25             System.Diagnostics.Process.Start( @"C:\" );
26         } // Конец метода cDriveLinkLabel_LinkClicked
27
28         // Загрузка www.deitel.com в браузере
29         private void deitelLinkLabel_LinkClicked( object sender,
30             LinkLabelLinkClickedEventArgs e )
31         {
32             // Изменение LinkColor после щелчка
33             deitelLinkLabel.LinkVisited = true;
34
35             System.Diagnostics.Process.Start( "http://www.deitel.com" );
36         } // Конец метода deitelLinkLabel_LinkClicked

```

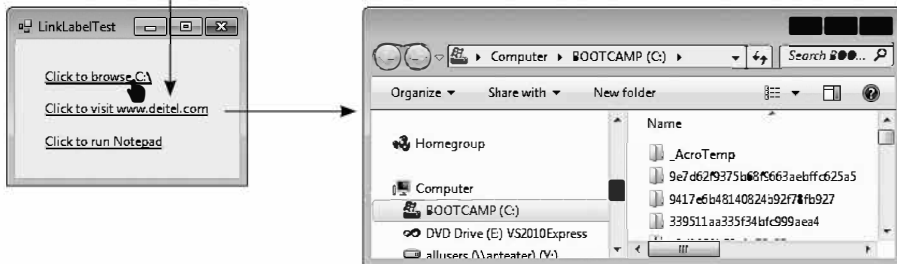
**Ил. 15.14.** Использование элементов управления LinkLabel для создания ссылок на диск, веб-страницу и приложение (продолжение ↗)

```

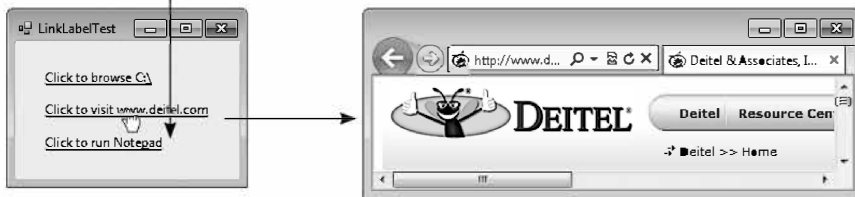
37
38 // Запуск приложения Notepad
39 private void notepadLinkLabel_LinkClicked( object sender,
40     LinkLabelLinkClickedEventArgs e )
41 {
42     // Изменение LinkColor после щелчка
43     notepadLinkLabel.LinkVisited = true;
44
45     // Программа запускается так, словно она запущена из меню
46     // "Выполнить"; полный путь не нужен.
47     System.Diagnostics.Process.Start( "notepad" );
48 } // Конец метода method notepadLinkLabel_LinkClicked
49 } // Конец класса LinkLabelTestForm
50 } // Конец пространства имен LinkLabelTest

```

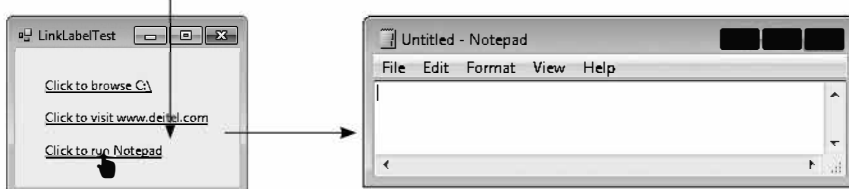
Щелкните на первом элементе управления LinkLabel, чтобы просмотреть содержимое диска C:



Щелкните на втором элементе управления LinkLabel, чтобы перейти на сайт Deitel



Щелкните на третьем элементе управления LinkLabel, чтобы открыть Блокнот



**Ил. 15.14.** Использование элементов управления LinkLabel для создания ссылок на диск, веб-страницу и приложение (окончание)

Обработчики событий элементов управления `LinkLabel` вызывают метод `Start` класса `Process` (пространство имен `System.Diagnostics`), позволяющий выполнять другие программы, загружать документы или веб-сайты из приложения. Метод `Start` получает один аргумент (открываемый файл) или два аргумента (запускаемое приложение и его аргументы командной строки). Аргументы метода `Start` могут задаваться в той же форме, в которой они вводятся для команды Windows **Выполнить** (Пуск ► Выполнить). Для приложений, известных Windows, полные имена не нужны, а расширения файлов часто могут опускаться. Чтобы открыть файл типа, известного системе Windows (и который она умеет обрабатывать), просто укажите полное имя файла. Например, при передаче методу файла `.docx` система Windows открывает его в Microsoft Word (или другой программе, зарегистрированной для открытия файлов `.docx`). Операционная система Windows должна иметь возможность использовать приложение, связанное с расширением заданного файла, для открытия файла.

Обработчик события `LinkClicked` элемента управления `cDriveLinkLabel` просматривает содержимое диска C: (строки 19–26). Строка 23 задает свойству `LinkVisited` значение `true`; при этом цвет ссылки меняется с синего на фиолетовый (цвета `LinkVisited` могут настраиваться в окне свойств Visual Studio). Затем обработчик события передает значение `@ "C:\\"` методу `Start` (строка 25), который открывает окно Проводника Windows. Символ `@` перед `"C:\\"` означает, что все символы строки должны интерпретироваться буквально. Таким образом, обратная косая черта в строке не считается первым символом служебной последовательности. Буквальная интерпретация символов упрощает пути к каталогам, поскольку символы `\` в путях не нужно заменять последовательностями `\\`.

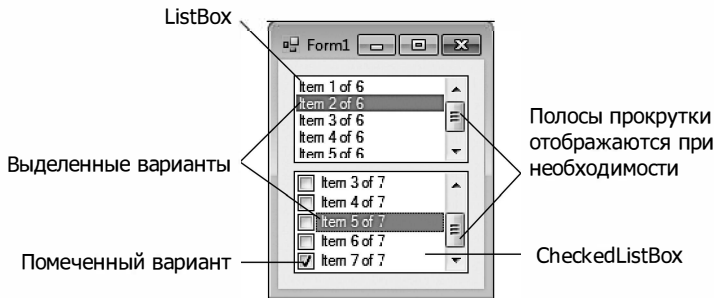
Обработчик события `LinkClicked` элемента управления `deitelLinkLabel` (строки 29–36) открывает веб-страницу `www.deitel.com` в браузере по умолчанию. Для этого адрес веб-страницы передается методу `Start` (строка 35), а веб-страница открывается в новом окне или на вкладке браузера. В строке 33 свойству `LinkVisited` задается значение `true`.

Обработчик события `LinkClicked` элемента управления `notepadLinkLabel` (строки 39–48) открывает приложение Блокнот (Notepad). В строке 43 свойству `LinkVisited` задается значение `true`, чтобы ссылка выглядела как посещенная. В строке 47 метод `Start` вызывается с аргументом `"notepad"`, что приводит к запуску `notepad.exe`. В строке 47 ни полный путь, ни расширение `.exe` не являются обязательными — Windows автоматически распознает в аргументе, переданном методу `Start`, исполняемый файл.

## 15.6. Элемент управления `ListBox`

Элемент управления `ListBox` предназначен для просмотра и выделения вариантов из списка. Объекты `ListBox` относятся к *статическим* объектам графического интерфейса; это означает, что пользователь не может напрямую редактировать содержимое списка. Приложение может предоставить пользователю текстовые

поля и кнопки для ввода вариантов, добавляемых в список, но непосредственное добавление должно выполняться в коде. Элемент управления `CheckedListBox` (раздел 15.7) расширяет `ListBox`, добавляя флажок рядом с каждым вариантом в списке. Пользователь может устанавливать пометки рядом с несколькими вариантами сразу, как при работе с наборами флажков. (Пользователь также может выделить несколько элементов `ListBox` при помощи свойства `SelectionMode`, которое будет описано далее.) На ил. 15.15 изображены элементы управления `ListBox` и `CheckedListBox`. Если содержимое списка не помещается в видимой области, в обоих элементах управления появляются полосы прокрутки.



**Ил. 15.15.** Элементы управления `ListBox` и `CheckBox` на форме

На ил. 15.16 перечислены часто используемые свойства и методы `ListBox`, а также часто используемое событие. Свойство `SelectionMode` определяет количество вариантов, которые могут выбираться пользователем. Допустимые значения свойства — `None`, `One`, `MultiSimple` и `MultiExtended` (из перечисления `SelectionMode`); различия между ними объясняются в таблице на ил. 15.16. Событие `SelectedIndexChanged` происходит при выборе нового варианта пользователем.

Свойства, методы и событие <code>ListBox</code>	Описание
Часто используемые свойства	
<code>Items</code>	Коллекция вариантов <code>ListBox</code>
<code>MultiColumn</code>	Указывает, может ли список отображаться в несколько столбцов (в этом режиме вертикальные полосы прокрутки не отображаются)
<code>SelectedIndex</code>	Возвращает индекс выделенного варианта. Если в списке нет выделенных вариантов, свойство возвращает <code>-1</code> . Если пользователь выбрал несколько вариантов, свойство возвращает только один из их индексов. При выделении нескольких вариантов следует использовать свойство <code>SelectedIndices</code>
<code>SelectedIndices</code>	Возвращает коллекцию индексов всех выделенных вариантов

**Ил. 15.16.** Свойства, методы и событие `ListBox` (продолжение ↗)

Свойства, методы и событие <code>ListBox</code>	Описание
<code>SelectedItem</code>	Возвращает ссылку на выделенный вариант. Если выделено несколько вариантов, свойство может вернуть любой из индексов
<code>SelectedItems</code>	Возвращает коллекцию выделенных вариантов
<code>SelectionMode</code>	Определяет количество вариантов, которые могут быть выделены, и способ выделения нескольких элементов. Допустимые значения: <code>None</code> , <code>One</code> (по умолчанию), <code>MultiSimple</code> (множественное выделение разрешено) и <code>MultiExtended</code> (поддерживается множественное выделение с использованием клавиш управления курсором, щелчков мышью и клавиш <code>Shift</code> и <code>Ctrl</code> )
<code>Sorted</code>	Указывает, сортируется ли содержимое списка в алфавитном порядке. Если свойству задано значение <code>true</code> , то список сортируется. По умолчанию используется значение <code>false</code>
Часто используемые методы	
<code>ClearSelected</code>	Снимает выделение со всех вариантов
<code>GetSelected</code>	Возвращает <code>true</code> , если элемент с указанным индексом выделен
Часто используемое событие	
<code>SelectedIndexChanged</code>	Генерируется при изменении выделенного индекса. Является событием по умолчанию при двойном щелчке на элементе управления в режиме конструктора

Ил. 15.16. Свойства, методы и событие `ListBox` (окончание)

Оба элемента управления, `ListBox` и `CheckedListBox`, содержат свойства `Items`, `SelectedItem` и `SelectedIndex`. Свойство `Items` возвращает коллекцию вариантов списка. Как правило, для операций со списками объектов в среде .NET используются коллекции. Многие компоненты графического интерфейса .NET (например, `ListBox`) используют коллекции для предоставления доступа к спискам внутренних объектов — например, вариантам `ListBox`. Коллекции более подробно рассматриваются в главе 21. Коллекция, возвращенная свойством `Items`, представляется объектом типа `ListBox.ObjectCollection`. Свойство `SelectedItem` возвращает текущий выделенный вариант `ListBox`. Если пользователь может выделить несколько вариантов, используйте коллекцию `SelectedItems` для получения всех выделенных вариантов в формате `ListBox.SelectedObjectCollection`. Свойство `SelectedIndex` возвращает индекс выделенного варианта — если выделенных вариантов может быть несколько, используйте свойство `SelectedIndices`, которое возвращает `ListBox.SelectedIndexCollection`. Если выделенные варианты отсутствуют, свойство `SelectedIndex` возвращает `-1`. Метод `GetSelected` получает индекс и возвращает `true`, если выделен заданный вариант.



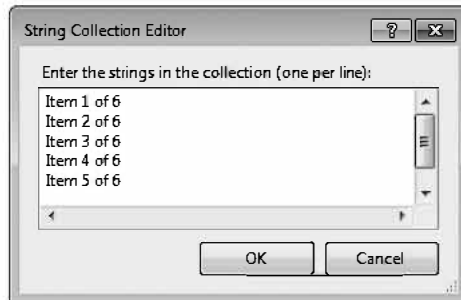
### Добавление вариантов в ListBox и CheckedListBox

Чтобы добавить новые варианты в элемент управления ListBox или CheckedListBox, необходимо добавить объекты в его коллекцию Items. Это можно сделать вызовом метода Add для добавления строки в коллекцию Items элемента управления ListBox или CheckedListBox. Например, следующая команда добавляет строку myListItem в элемент управления ListBox с именем myListBox:

```
myListBox.Items.Add( myListItem );
```

Чтобы добавить несколько объектов, следует либо многократно вызвать метод Add, либо вызвать метод AddRange для добавления массива объектов. Каждый из классов ListBox и CheckedListBox вызывает метод ToString переданного объекта для определения надписи, представляющей соответствующий объект в списке. Это позволяет добавлять в ListBox и CheckedListBox разные объекты, которые позднее возвращаются в свойствах SelectedItem и SelectedItems.

Также возможно визуальное добавление вариантов в ListBox и CheckedListBox в свойстве Items в окне свойств. Кнопка с многоточием открывает окно String Collection Editor, в котором имеется текстовая область для добавления вариантов; каждый вариант выводится в отдельной строке (ил. 15.17). Затем Visual Studio включает в метод InitializeComponent код добавления этих строк в коллекцию Items.



Ил. 15.17. Окно String Collection Editor

Программа на ил. 15.18 использует класс ListBoxTestForm для добавления и удаления вариантов из списка displayListBox. В текстовом поле inputTextBox пользователь вводит новый вариант. При нажатии кнопки Add Button новый вариант появляется в displayListBox. Аналогичным образом, когда пользователь выделяет вариант и щелкает на кнопке Remove, вариант удаляется. При щелчке Clear удаляет все варианты в displayListBox. Кнопка Exit завершает приложение.

Обработчик события addButton\_Click (строки 20–24) вызывает метод Add коллекции Items в ListBox. При вызове метода передается строка, добавляемая в displayListBox. В данном случае используется строка, введенная пользователем в inputTextBox (строка 22). После добавления варианта свойство inputTextBox.Text очищается (строка 23).

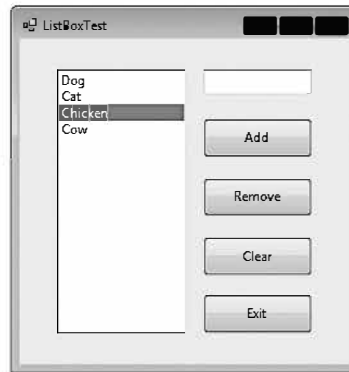
```
1 // Ил. 15.18: ListBoxTestForm.cs
2 // Программа добавления и удаления вариантов ListBox
3 using System;
4 using System.Windows.Forms;
5
6 namespace ListBoxTest
7 {
8     // Форма с текстовым полем и кнопками для добавления
9     // и удаления вариантов ListBox.
10    public partial class ListBoxTestForm : Form
11    {
12        // Конструктор
13        public ListBoxTestForm()
14        {
15            InitializeComponent();
16        } // Конец конструктора
17
18        // Добавление нового варианта в ListBox (текст берется
19        // из текстового поля, которое затем очищается)
20        private void addButton_Click( object sender, EventArgs e )
21        {
22            displayListBox.Items.Add( inputTextBox.Text );
23            inputTextBox.Clear();
24        } // Конец метода addButton_Click
25
26        // Удаление варианта (если есть выделенный вариант)
27        private void removeButton_Click( object sender, EventArgs e )
28        {
29            // Проверяем наличие выделенного варианта; удаляем, если есть
30            if ( displayListBox.SelectedIndex != -1 )
31                displayListBox.Items.RemoveAt(
32                    displayListBox.SelectedIndex );
33        } // Конец метода removeButton_Click
34
35        // Очистка списка вариантов в ListBox
36        private void clearButton_Click( object sender, EventArgs e )
37        {
38            displayListBox.Items.Clear();
39        } // Конец метода clearButton_Click
40
41        // Выход из приложения
42        private void exitButton_Click( object sender, EventArgs e )
43        {
44            Application.Exit();
45        } // Конец метода exitButton_Click
46    } // Конец класса ListBoxTestForm
47 } // Конец пространства имен ListBoxTest
```

**Ил. 15.18.** Программа для добавления и удаления вариантов ListBox  
(продолжение ↗)

а) Приложение после добавления вариантов Dog, Cat и Chicken, но до добавления Cow



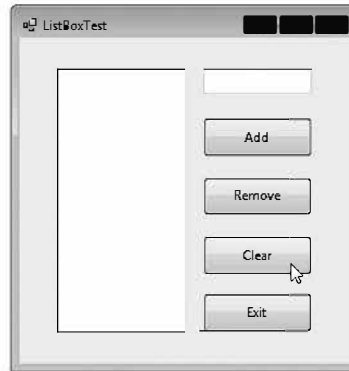
б) Приложение после добавления Cow, но до удаления Chicken



в) Приложение после удаления Chicken



г) Приложение после очистки списка ListBox



**Ил. 15.18.** Программа для добавления и удаления вариантов ListBox (окончание)

Обработчик события `removeButton_Click` (строки 27–33) использует метод `RemoveAt` для удаления варианта из списка. Обработчик события `removeButton_Click` сначала использует свойство `SelectedIndex` для определения выбранного индекса. Если значение `SelectedIndex` не равно `-1` (то есть в списке присутствует выделенный вариант), строки 31–32 удаляют вариант, соответствующий индексу выделения. Обработчик события `clearButton_Click` (строки 36–39) вызывает метод `Clear` коллекции `Items` (строка 38), удаляющий все содержимое `displayListBox`. Наконец, обработчик события `exitButton_Click` (строки 42–45) завершает приложение вызовом метода `Application.Exit` (строка 44).

## 15.7. Элемент управления `CheckedListBox`

Элемент управления `CheckedListBox` является производным от `ListBox`; рядом с каждым вариантом в списке отображается флажок (`CheckBox`). Варианты добавляются методами `Add` и `AddRange` или в окне `String Collection Editor`. Элемент управления `CheckedListBox` допускает пометку нескольких вариантов, но для выделения вариантов устанавливаются более жесткие ограничения. Для свойства `SelectionMode` допустимы только значения `None` и `One`. В режиме `One` разрешается одиночное выделение, тогда как в режиме `None` выделение запрещено. Так как вариант должен быть выделен для пометки, если вы хотите разрешить пользователю выделение вариантов, свойству `SelectionMode` необходимо задать значение `One`. Таким образом, переключение свойства `SelectionMode` между значениями `One` и `None` фактически означает выбор между возможностью и невозможностью пометки элементов списка. Часто используемые свойства, метод и событие `CheckedListBox` перечислены на ил. 15.19.



### ТИПИЧНАЯ ОШИБКА 15.1

При попытке задать свойству `SelectionMode` значение `MultiSimple` или `MultiExtended` в окне свойств `CheckedListBox` среда разработки выводит сообщение об ошибке. Если эти значения задаются на программном уровне, происходит ошибка времени выполнения.

Каждый раз, когда пользователь устанавливает или снимает пометку варианта в `CheckedListBox`, происходит событие `ItemCheck`. Свойства объекта аргументов события `CurrentValue` и `NewValue` возвращают значения `CheckState` для текущего и нового состояния варианта соответственно. Сравнение этих значений позволяет определить, была ли установлена или снята пометка варианта `CheckedListBox`. В элементе управления `CheckedListBox` присутствуют свойства `SelectedItems` и `SelectedIndices` (унаследованные от класса `ListBox`), но к ним также добавляются свойства `CheckedItems` и `CheckedIndices`, возвращающие информацию о помеченных вариантах и индексах.

Свойства, метод и событие <code>CheckedListBox</code>	Описание
Часто используемые свойства	
<code>CheckedItems</code>	Свойство доступно только во время выполнения. Возвращает коллекцию помеченных вариантов в виде <code>CheckedListBox.CheckedItemCollection</code> . Не путайте с выделенным вариантом, который имеет другой цвет, но не обязательно помечен. В любой момент времени в <code>CheckedListBox</code> может быть не более одного выделенного варианта
<code>CheckedIndices</code>	Свойство доступно только во время выполнения. Возвращает индексы всех помеченных вариантов в виде коллекции <code>CheckedListBox.CheckedIndexCollection</code>

**Ил. 15.19.** Свойства, методы и событие `CheckedListBox` (продолжение ↗)

Свойства, метод и событие <code>CheckedListBox</code>	Описание
<code>CheckOnClick</code>	Если свойство истинно и пользователь щелкает на варианте, то происходит одновременное установление/снятие выделения и установление/снятие пометки. По умолчанию свойство равно <code>false</code> ; это означает, что пользователь должен щелкнуть на варианте, а затем сделать повторный щелчок для установления или снятия пометки
<code>SelectionMode</code>	Определяет возможность выделения и пометки вариантов. Допустимые значения: <code>One</code> (используется по умолчанию; возможна пометка нескольких элементов) и <code>None</code> (пометка запрещена)
Часто используемый метод	
<code>GetItemChecked</code>	Получает индекс и возвращает <code>true</code> , если соответствующий вариант помечен
Часто используемое событие	
<code>ItemCheck</code>	Генерируется при установлении или снятии пометки варианта
Свойства <code>ItemCheckEventArgs</code>	
<code>CurrentValue</code>	Указывает, помечен или нет текущий вариант. Допустимые значения: <code>Checked</code> , <code>Unchecked</code> и <code>Indeterminate</code>
<code>Index</code>	Возвращает индекс измененного варианта (начиная с нуля)
<code>NewValue</code>	Задаёт новое состояние варианта

**Ил. 15.19.** Свойства, методы и событие `CheckedListBox` (окончание)

В листинге на ил. 15.20 класс `CheckedListBoxTestForm` использует элементы управления `CheckedListBox` и `ListBox` для отображения книг, выбранных пользователем из списка. `CheckedListBox` даёт возможность выбрать сразу несколько книг. В окне `String Collection Editor` были добавлены варианты для книг Deitel по разным темам: C, C++, Java™, Internet & WWW, VB 2012, Visual C++ и Visual C# 2012. В элементе управления `ListBox` (с именем `displayListBox`) выводятся книги, выбранные пользователем. На снимках результатов элемент управления `CheckedListBox` располагается слева, а `ListBox` — справа.

Когда пользователь устанавливает или снимает пометку в `itemCheckedListBox`, происходит событие `ItemCheck`, по которому выполняется обработчик события `itemCheckedListBox_ItemCheck` (строки 19–31). Команда `if...else` (строки 27–30) определяет, была ли установлена или снята пометка варианта в `CheckedListBox`. В строке 27 свойство `NewValue` используется для определения того, был ли вариант помечен (`CheckState.Checked`). Если пользователь пометил вариант, то строка 28 добавляет его в `ListBoxdisplayListBox`. Если пользователь снял пометку с варианта, строка 30 удаляет соответствующий вариант из `displayListBox`. Чтобы создать этот обработчик события, следует выделить `CheckedListBox` в режиме конструктора, просмотреть свойства элемента управления в окне свойств и сделать двойной щелчок

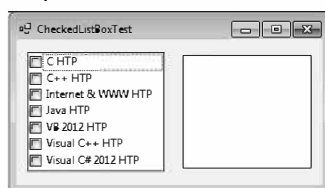
на событии `ItemCheck`. Событием по умолчанию для `CheckedListBox` является событие `SelectedIndexChanged`.

```

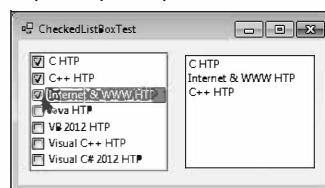
1 // Ил. 15.20: CheckedListBoxTestForm.cs
2 // Использование CheckedListBox и ListBox в приложении.
3 using System;
4 using System.Windows.Forms;
5
6 namespace CheckedListBoxTest
7 {
8     // Форма использует для добавления вариантов в ListBox.
9     public partial class CheckedListBoxTestForm : Form
10     {
11         // Конструктор
12         public CheckedListBoxTestForm()
13         {
14             InitializeComponent();
15         } // Конец конструктора
16
17         // Вариант, для которого устанавливается/снимается пометка,
18         // добавляется/удаляется из ListBox.
19         private void itemCheckedListBox_ItemCheck(
20             object sender, ItemCheckEventArgs e )
21         {
22             // Получение ссылки на выделенный вариант.
23             string item = itemCheckedListBox.SelectedItem.ToString();
24
25             // Если вариант помечен, добавить его в ListBox;
26             // в противном случае удалить его из ListBox.
27             if ( e.NewValue == CheckState.Checked )
28                 displayListBox.Items.Add( item );
29             else
30                 displayListBox.Items.Remove( item );
31         } // Конец метода itemCheckedListBox_ItemCheck
32     } // Конец класса CheckedListBoxTestForm
33 } // Конец пространства имен CheckedListBoxTest

```

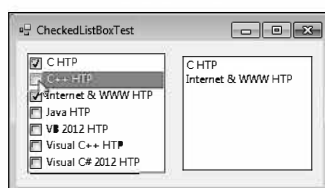
а) Приложение в момент запуска



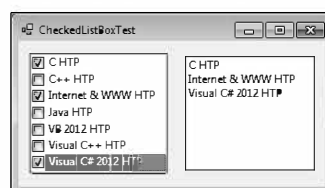
б) Приложение после пометки первых трех вариантов



в) Приложение после снятия пометки с варианта C++ HTP



г) Приложение после пометки варианта Visual C# 2012 HTP

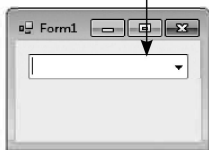


**Ил. 15.20.** Использование `CheckedListBox` и `ListBox` в приложении

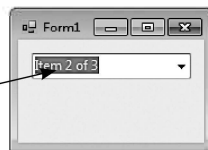
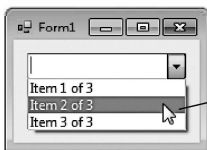
## 15.8. Элемент управления ComboBox

Элемент управления ComboBox (поле со списком) объединяет функциональность текстового поля и раскрывающегося списка — он содержит список, из которого пользователь может выбрать значение. Как правило, ComboBox выглядит как текстовое поле, справа от которого располагается кнопка со стрелкой. По умолчанию пользователь может ввести текст в поле или щелкнуть на кнопке, чтобы открыть список заранее определенных вариантов. Если пользователь выбирает вариант из списка, этот вариант отображается в текстовом поле. Если все содержимое не помещается в списке, появляется полоса прокрутки. Максимальное количество вариантов, которые могут отображаться в раскрывающемся списке одновременно, задается свойством `MaxDropDownItems`. На ил. 15.21 изображен элемент управления ComboBox в трех разных состояниях.

Щелкните на кнопке со стрелкой, чтобы открыть список вариантов



Выбор варианта из раскрывающегося списка изменяет содержимое текстового поля



**Ил. 15.21.** Использование ComboBox

Как и в случае с элементом управления ListBox, объекты можно добавлять в коллекцию `Items` на программном уровне, с использованием методов `Add` и `AddRange`, или визуально, в окне String Collection Editor. На ил. 15.22 перечислены часто используемые свойства и событие класса ComboBox.



### КРАСИВО И УДОБНО 15.4

Элемент управления ComboBox помогает сэкономить место в графическом интерфейсе. Его недостаток заключается в том, что, в отличие от ListBox, для просмотра всех доступных вариантов пользователь должен открыть раскрывающийся список.

Свойства и событие ComboBox	Описание
Часто используемые свойства	
DropDownStyle	Определяет тип ComboBox. Значение Simple означает, что текстовая часть допускает редактирование, а список виден всегда. Значение DropDown (используется по умолчанию) означает, что текстовая часть допускает редактирование, а для просмотра списка пользователь должен щелкнуть на кнопке со стрелкой. Значение DropDownList означает, что текстовая часть не редактируется, а список открывается щелчком на кнопке

**Ил. 15.22.** Свойства и событие ComboBox (продолжение ↗)

Свойства и событие ComboBox	Описание
Items	Коллекция вариантов элемента управления ComboBox
MaxDropDownItems	Задаёт максимальное количество вариантов (от 1 до 100), которые могут отображаться в раскрывающемся списке. Если количество вариантов превышает максимально допустимое, появляется полоса прокрутки
SelectedIndex	Возвращает индекс выделенного варианта или -1, если выделение отсутствует
SelectedItem	Возвращает ссылку на выделенный вариант
Sorted	Указывает, сортируется ли содержимое списка в алфавитном порядке. Если свойству задано значение true, то список сортируется. По умолчанию используется значение false
Часто используемое событие	
SelectedIndexChanged	Генерируется при изменении выделенного индекса (например, при выделении другого варианта). Является событием по умолчанию при двойном щелчке на элементе управления в режиме конструктора

**Ил. 15.22.** Свойства и событие ComboBox (окончание)

Свойство `DropDownStyle` определяет тип `ComboBox` и представляется значением из перечисления `ComboBoxStyle`, которое содержит значения `Simple`, `DropDown` и `DropDownList`. В режиме `Simple` кнопка со стрелкой отсутствует; вместо этого рядом с элементом отображается полоса прокрутки, позволяющая пользователю выбрать нужный вариант из списка. Пользователь также может ввести свой вариант в текстовом поле. В режиме `DropDown` (используется по умолчанию) список открывается при щелчке на кнопке со стрелкой (или нажатии клавиши  $\downarrow$ ). Пользователь может ввести новый вариант в текстовом поле. В последнем режиме `DropDownList` имеется раскрывающийся список, но пользователь не может вводить текст в текстовом поле.

Элемент управления `ComboBox` обладает свойствами `Items` (коллекция), `SelectedItem` и `SelectedIndex`, сходными с аналогичными свойствами `ListBox`. В `ComboBox` можно выделить не более одного варианта. Если выделение отсутствует, то значение `SelectedIndex` равно -1. При изменении выделения происходит событие `SelectedIndexChanged`.

Класс `ComboBoxTestForm` (ил. 15.23) предлагает пользователю выбрать фигуру, которая будет нарисована на форме, — круг, эллипс, квадрат или сектор (как с заполнением, так и без) — в списке `ComboBox`. Редактирование `ComboBox` в этом примере запрещено, поэтому пользователь не сможет ввести нужный вариант в текстовом поле.



### КРАСИВО И УДОБНО 15.5

Разрешайте редактирование для списковых элементов управления (таких, как `ComboBox`) только в том случае, если приложение рассчитано на ввод новых вариантов пользователем. В противном случае пользователь может ввести значение, неподходящее для целей вашего приложения.



```
1 // Ил. 15.23: ComboBoxTestForm.cs
2 // Использование ComboBox для выбора фигуры.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6
7 namespace ComboBoxTest
8 {
9     // Использование ComboBox для выбора фигуры, рисуемой на форме.
10    public partial class ComboBoxTestForm : Form
11    {
12        // Конструктор
13        public ComboBoxTestForm()
14        {
15            InitializeComponent();
16        } // Конец конструктора
17
18        // Получение индекса и рисование выбранной фигуры
19        private void imageComboBox_SelectedIndexChanged(
20            object sender, EventArgs e )
21        {
22            // Создание объектов Graphics, Pen и SolidBrush
23            Graphics myGraphics = base.CreateGraphics();
24
25            // Создание объекта Pen с цветом DarkRed
26            Pen myPen = new Pen( Color.DarkRed );
27
28            // Создание объекта SolidBrush с цветом DarkRed
29            SolidBrush mySolidBrush = new SolidBrush( Color.DarkRed );
30
31            // Очистка области вывода и заполнение ее белым цветом.
32            myGraphics.Clear( Color.White );
33
34            // Определение индекса и рисование фигуры
35            switch ( imageComboBox.SelectedIndex )
36            {
37                case 0: // Выбран круг
38                    myGraphics.DrawEllipse( myPen, 50, 50, 150, 150 );
39                    break;
40                case 1: // Выбран прямоугольник
41                    myGraphics.DrawRectangle( myPen, 50, 50, 150, 150 );
42                    break;
43                case 2: // Выбран эллипс
44                    myGraphics.DrawEllipse( myPen, 50, 85, 150, 115 );
45                    break;
46                case 3: // Выбран сектор
47                    myGraphics.DrawPie( myPen, 50, 50, 150, 150, 0, 45 );
48                    break;
49                case 4: // Выбран заполненный круг
50                    myGraphics.FillEllipse( mySolidBrush, 50, 50, 150, 150 );
51                    break;
52                case 5: // Выбран заполненный прямоугольник
53                    myGraphics.FillRectangle( mySolidBrush, 50, 50, 150,
54                        150 );
```

**Ил. 15.23.** Использование ComboBox для рисования фигуры (продолжение ➤)

```

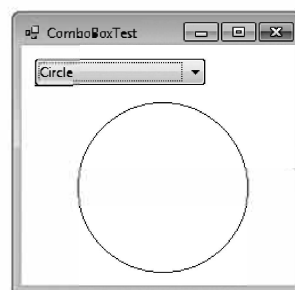
55         break;
56     case 6: // Выбран заполненный эллипс
57         myGraphics.FillEllipse( mySolidBrush, 50, 85, 150, 115 );
58         break;
59     case 7: // Выбран заполненный сектор
60         myGraphics.FillPie( mySolidBrush, 50, 50, 150, 150, 0,
61             45 );
62         break;
63     } // Конец switch
64
65     myGraphics.Dispose(); // Освобождение объекта Graphics
66 } // Конец метода imageComboBox_SelectedIndexChanged
67 } // Конец класса ComboBoxTestForm
68 } // Конец пространства имен ComboBoxTest

```

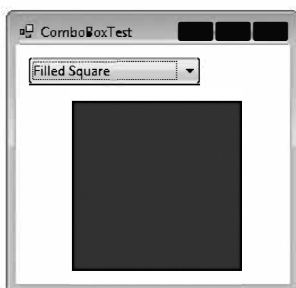
а) Приложение  
в момент  
запуска



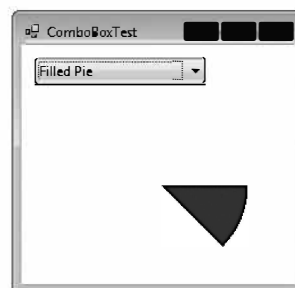
б) Приложение  
после выбора  
варианта Circle  
в ComboBox



в) Приложение  
после выбора  
варианта  
Filled Square  
в ComboBox



г) Приложение  
после выбора  
варианта Filled Pie  
в ComboBox



**Ил. 15.23.** Использование ComboBox для рисования фигуры (окончание)

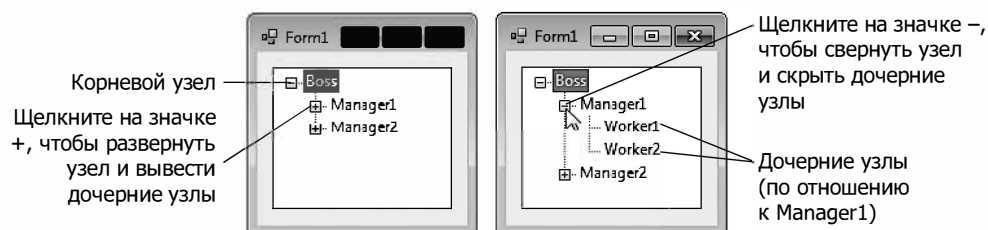
После создания элемента управления ComboBox с именем `imageComboBox` запретите его редактирование, задав свойству `DropDownStyle` значение `DropDownList` в окне свойств. Затем добавьте варианты `Circle`, `Square`, `Ellipse`, `Pie`, `Filled Circle`, `Filled Square`, `FilledEllipse` и `Filled Pie` в коллекцию `Items` в окне `String Collection Editor`. Когда пользователь выбирает вариант в `imageComboBox`, происходит событие `SelectedIndexChanged` и выполняется обработчик события `imageComboBox_SelectedIndexChanged` (строки 19–66). В строках 23–29 создаются объекты `Graphics`, `Pen` и `SolidBrush`, используемые для рисования на форме. Объект `Graphics` (строка 23) позволяет рисовать на компоненте пером или кистью с использованием одного из методов

Graphics. Объект Pen (строка 26) используется методами DrawEllipse, DrawRectangle и DrawPie (строки 38, 41, 44 и 47) для рисования контуров фигур. Объект SolidBrush (строка 29) используется методами FillEllipse, FillRectangle и FillPie (строки 50, 53–54, 57 и 60–61) для заполнения соответствующих контуров. Строка 32 окрашивает всю форму в белый цвет методом Clear класса Graphics.

Приложение рисует фигуру, определяемую индексом выделенного элемента. Команда switch (строки 35–63) использует свойство imageComboBox.SelectedIndex для определения того, какой вариант был выделен пользователем. Метод DrawEllipse класса Graphics (строка 38) получает объект Pen, координаты левого верхнего угла, ширину и высоту ограничивающего прямоугольника (то есть прямоугольной области), в котором будет отображаться эллипс. Начало координат находится в левом верхнем углу формы; координата  $x$  увеличивается слева направо, а координата  $y$  — сверху вниз. В строке 38 рисуется круг — частный случай эллипса с одинаковой шириной и высотой. В строке 44 рисуется эллипс с разными значениями ширины и высоты. Метод DrawRectangle класса Graphics (строка 41) получает объект Pen, координаты левого верхнего угла и ширину с высотой прямоугольника. Метод DrawPie (строка 47) рисует сектор как часть эллипса, ограничиваемого заданным прямоугольником. Метод DrawPie получает объект Pen, координаты левого верхнего угла прямоугольника, его ширину и высоту, начальный угол (в градусах) и угловой размер дуги (в градусах). Значения углов возрастают по часовой стрелке. Методы FillEllipse (строки 50 и 57), FillRectangle (строки 53–54) и FillPie (строки 60–61) похожи на свои аналоги для рисования незаполненных фигур, не считая того, что вместо Pen им передается объект Brush (например, SolidBrush). Некоторые из нарисованных фигур изображены на снимках к ил. 15.23.

## 15.9. Элемент управления TreeView

Элемент управления TreeView отображает иерархическое дерево с узлами. Традиционно узлы представляются объектами, которые содержат значения и могут ссылаться на другие узлы. Родительский узел содержит дочерние узлы, а дочерние узлы могут быть родителями для других узлов. Два дочерних узла с общим родителем называются *родственными* (sibling). Дерево представляет собой совокупность узлов, обычно упорядоченных в иерархическую структуру. Первым родительским узлом дерева является корневой узел (TreeView может иметь несколько корней). Например, файловая система компьютера может быть представлена в виде дерева. Каталог верхнего уровня (обычно C:) является корнем, каждая папка верхнего уровня на диске C: — дочерним узлом, а каждая вложенная папка может содержать собственные вложенные папки. Элементы управления TreeView полезны для отображения иерархической информации — например, только что упоминавшейся файловой структуры. Узлы и деревья более подробно рассматриваются в главе 19. На ил. 15.24 изображена форма с элементом управления TreeView.



Ил. 15.24. Элемент управления TreeView с деревом

Родительские узлы можно открывать и закрывать, щелкая на квадратике со значком + или -. У узлов, не имеющих дочерних узлов, этих квадратиков нет.

Узлы TreeView представляются экземплярами класса `TreeNode`. Каждый объект `TreeNode` содержит коллекцию `Nodes` (тип `TreeNodeCollection`), которая содержит другие — *дочерние* — объекты `TreeNode`. Свойство `Parent` возвращает ссылку на родительский узел (или `null` для корневого узла). На ил. 15.25 и 15.26 перечислены часто используемые свойства TreeView и TreeNode, методы TreeNode и часто используемое событие TreeView.

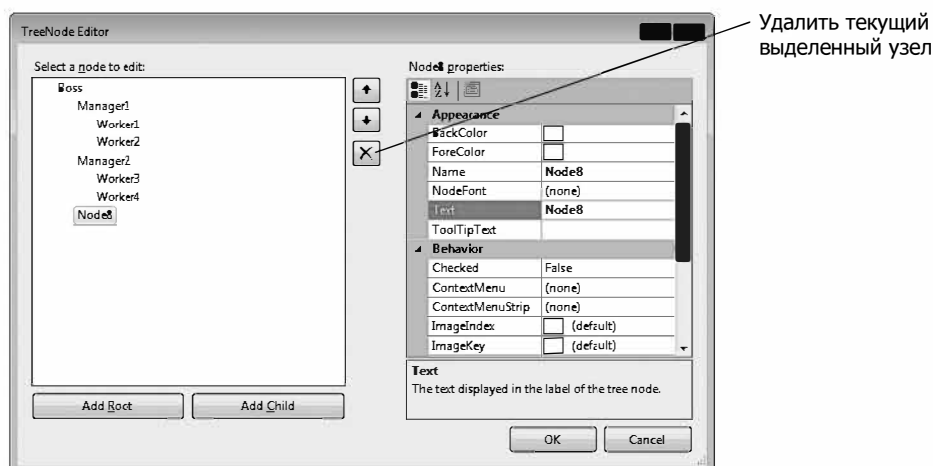
Свойства и событие TreeView	Описание
Часто используемые свойства	
<code>CheckBoxes</code>	Указывает, должны ли отображаться флажки рядом с узлами. Со значением <code>true</code> флажки отображаются. По умолчанию используется значение <code>false</code>
<code>ImageList</code>	Задаёт объект <code>ImageList</code> со значками узлов. Объект <code>ImageList</code> представляет собой коллекцию, содержащую объекты <code>Image</code>
<code>Nodes</code>	Возвращает коллекцию <code>TreeNode</code> элемента управления в виде объекта <code>TreeNodeCollection</code> с методами <code>Add</code> (добавление объекта <code>TreeNode</code> ), <code>Clear</code> (удаление всей коллекции) и <code>Remove</code> (удаление заданного узла). Удаление родительского узла приводит к удалению всех его дочерних узлов
<code>SelectedNode</code>	Выделенный узел
Часто используемое событие	
<code>AfterSelect</code>	Генерируется после изменения выделенного узла. Является событием по умолчанию при двойном щелчке на элементе управления в режиме конструктора

Ил. 15.25. Свойства и событие TreeView

Свойства и методы TreeNode	Описание
Часто используемые свойства	
Checked	Указывает, помечен ли узел TreeNode (в родительском объекте TreeView свойство CheckBoxes должно содержать true)
FirstNode	Задаёт первый узел в коллекции Nodes (то есть первый дочерний узел дерева)
FullPath	Обозначает путь к узлу от корня дерева
ImageIndex	Задаёт индекс изображения, выводимого при отсутствии выделения узла, из коллекции ImageList объекта TreeView
LastNode	Задаёт последний узел в коллекции Nodes (то есть последний дочерний узел дерева)
NextNode	Следующий родственный узел
Nodes	Коллекция объектов TreeNode, содержащихся в текущем узле (то есть все дочерние узлы текущего узла), с методами Add (добавление объекта TreeNode), Clear (удаление всей коллекции) и Remove (удаление заданного узла). Удаление родительского узла приводит к удалению всех его дочерних узлов
PrevNode	Предыдущий родственный узел
SelectedImageIndex	Задаёт индекс изображения, выводимого при выделении узла, из коллекции ImageList объекта TreeView
Text	Задаёт текст TreeNode
Часто используемые методы	
Collapse	Закрывает узел
Expand	Раскрывает узел
ExpandAll	Открывает все дочерние узлы текущего узла
GetNodeCount	Возвращает количество дочерних узлов

**Ил. 15.26.** Свойства и методы TreeNode

Чтобы добавить узлы в TreeView визуальными средствами, щелкните на кнопке с многоточием рядом со свойством Nodes в окне свойств. На экране появляется окно **TreeNode Editor** (ил. 15.27) с пустым деревом, представляющим TreeView. В окне имеются кнопки для создания корневого узла и добавления/удаления узлов. Справа выводятся свойства текущего узла; здесь можно изменить его имя.



Ил. 15.27. Окно TreeNode Editor

Чтобы добавить узлы на программном уровне, сначала создайте корневой узел. Создайте новый объект `TreeNode`, передайте ему отображаемую строку. Затем вызовите метод `Add`, чтобы добавить новый узел `TreeNode` в коллекцию `Nodes` объекта `TreeView`. Например, команда для добавления корневого узла выглядит так:

```
myTreeView.Nodes.Add( new TreeNode( rootLabel ) );
```

где *myTreeView* — элемент управления `TreeView`, в который добавляются узлы, а *rootLabel* — текст, выводимый в *myTreeView*. Чтобы включить в корневой узел дочерние узлы, добавьте новые объекты `TreeNode` в коллекцию `Nodes`. Корневой узел `TreeView` выбирается выражением

```
myTreeView.Nodes[ myIndex ]
```

где *myIndex* — индекс корневого узла в коллекции `Nodes` элемента управления *myTreeView*. Добавление узлов в дочерние узлы происходит так же, как добавление корневых узлов в *myTreeView*. Чтобы добавить дочерний узел в корневой узел с индексом *myIndex*, используйте команду вида

```
myTreeView.Nodes[ myIndex ].Nodes.Add( new TreeNode( ChildLabel ) );
```

Класс `TreeViewDirectoryStructureForm` (ил. 15.28) использует элемент управления `TreeView` для вывода содержимого каталога, выбранного пользователем. Для определения каталога используется текстовое поле и кнопка. Сначала пользователь вводит полный путь к каталогу, а затем щелкает на кнопке, чтобы назначить заданный каталог корневым узлом `TreeView`. Каждый подкаталог этого каталога становится дочерним узлом. Общий макет похож на тот, который используется в Проводнике Windows. Папки можно раскрывать и закрывать при помощи квадратиков со значками, расположенными слева.

Когда пользователь щелкает на кнопке `enterButton`, все узлы `directoryTreeView` стираются (строка 68). Затем, если заданный каталог существует (строка 73), введенный в `inputTextBox` путь используется для создания корневого узла. Строка 76 добавляет

каталог в `directoryTreeView` как корневой узел, а в строках 79–80 вызывается метод `PopulateTreeView` (строки 21–62), получающий каталог (строку) и родительский узел. Метод `PopulateTreeView` создает дочерние узлы для подкаталогов каталога, полученного в аргументе.

```

1  // Ил. 15.28: TreeViewDirectoryStructureForm.cs
2  // Вывод структуры каталогов в элементе управления TreeView.
3  using System;
4  using System.Windows.Forms;
5  using System.IO;
6
7  namespace TreeViewDirectoryStructure
8  {
9      // Форма использует TreeView для вывода структуры каталогов
10     public partial class TreeViewDirectoryStructureForm : Form
11     {
12         string substringDirectory; // Последняя часть полного имени
13
14         // Конструктор
15         public TreeViewDirectoryStructureForm()
16         {
17             InitializeComponent();
18         } // Конец конструктора
19
20         // Заполнение текущего узла данными подкаталогов
21         public void PopulateTreeView(
22             string directoryValue, TreeNode parentNode )
23         {
24             // Массив подкаталогов текущего каталога
25             string[] directoryArray =
26                 Directory.GetDirectories( directoryValue );
27
28             // Заполнение текущего узла данными подкаталогов
29             try
30             {
31                 // Проверяем наличие подкаталогов
32                 if ( directoryArray.Length != 0 )
33                 {
34                     // Для каждого подкаталога создать новый объект TreeNode,
35                     // добавить его как дочерний для текущего узла
36                     // и рекурсивно заполнить дочерние узлы
37                     foreach ( string directory in directoryArray )
38                     {
39                         // Получение последней части полного имени
40                         // из полного пути вызовом метода
41                         // GetFileNameWithoutExtension класса Path
42                         substringDirectory =
43                             Path.GetFileNameWithoutExtension( directory );
44
45                         // Создание объекта TreeNode для текущего каталога
46                         TreeNode myNode = new TreeNode( substringDirectory );
47
48                         // Добавление текущего каталога в родительский узел
49                         parentNode.Nodes.Add( myNode );
50
51                         // Рекурсивное заполнение всех подкаталогов
52                         PopulateTreeView( directory, myNode );

```

**Ил. 15.28.** Вывод содержимого каталогов в элементе управления TreeView (продолжение ↗)

```

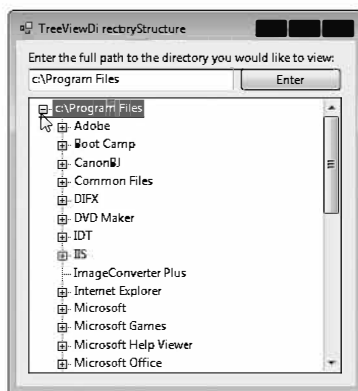
53         } // Конец foreach
54     } // Конец if
55 } // Конец try
56
57 // Перехват исключения
58 catch ( UnauthorizedAccessException )
59 {
60     parentNode.Nodes.Add( "Access denied" );
61 } // Конец catch
62 } // Конец метода PopulateTreeView
63
64 // Обработчик события Click кнопки enterButton
65 private void enterButton_Click( object sender, EventArgs e )
66 {
67     // Очистка всех узлов
68     directoryTreeView.Nodes.Clear();
69
70     // Проверка существования каталога, заданного пользователем.
71     // Если каталог существует, заполняем TreeView, а если нет -
72     // выводим окно с сообщением об ошибке.
73     if ( Directory.Exists( inputTextBox.Text ) )
74     {
75         // Включение полного пути в directoryTreeView
76         directoryTreeView.Nodes.Add( inputTextBox.Text );
77
78         // Добавление подкаталогов
79         PopulateTreeView(
80             inputTextBox.Text, directoryTreeView.Nodes[ 0 ] );
81     }
82     // Вывод сообщения об ошибке, если каталог не найден
83     else
84         MessageBox.Show( inputTextBox.Text + " could not be found.",
85             "Directory Not Found", MessageBoxButtons.OK,
86             MessageBoxIcon.Error );
87 } // Конец метода enterButton_Click
88 } // Конец класса TreeViewDirectoryStructureForm
89 } // Конец пространства имен TreeViewDirectoryStructure

```

а) Приложение после ввода пути



б) Приложение после нажатия кнопки для вывода содержимого каталога

**Ил. 15.28.** Вывод содержимого каталогов в элементе управления TreeView (окончание)



Метод `PopulateTreeView` (строки 21–62) получает список подкаталогов при помощи метода `GetDirectories` класса `Directory` (пространство имен `System.IO`) в строках 25–26. Метод `GetDirectories` получает строку (текущий каталог) и возвращает массив строк (подкаталоги). Если каталог недоступен по соображениям безопасности, выдается исключение `UnauthorizedAccessException`. В строках 58–61 это исключение перехватывается, и вместо данных подкаталогов добавляется узел с текстом «Access denied».

Если каталог содержит подкаталоги, в строках 42–43 метод `GetFileNameWithoutExtension` класса `Path` сокращает полное имя до имени каталога. Класс `Path` предоставляет функции для работы со строками путей классов или каталогов. Затем для каждой строки в `directoryArray` создается новый дочерний узел (строка 46). Мы используем метод `Add` (строка 49) для включения каждого дочернего узла в родительский узел. Метод `PopulateTreeView` вызывается рекурсивно для каждого подкаталога (строка 52), который последовательно заполняет `TreeView` всей структурой каталогов. С нашим рекурсивным алгоритмом исходная загрузка больших каталогов может сопровождаться задержкой, но после добавления имен папок в соответствующую коллекцию `Nodes` они будут открываться и закрываться без задержек. В следующем разделе будет представлен альтернативный алгоритм решения этой проблемы.

## 15.10. Элемент управления ListView

Элемент управления `ListView` отчасти похож на `ListBox`: он тоже отображает список, в котором пользователь может выбрать один или несколько вариантов (пример использования `ListView` представлен на ил. 15.31). Класс `ListView` более универсален и позволяет выводить данные в более разнообразных форматах. Например, `ListView` может выводить значки рядом с вариантами списка (под управлением свойств `SmallImageList`, `LargeImageList` и `StateImageList`) и выводить дополнительную информацию о вариантах. Свойство `MultiSelect` (типа `bool`) определяет, разрешено ли выделение нескольких вариантов. Если задать свойству `CheckBoxes` (типа `bool`) значение `true`, рядом с вариантами списка будут отображаться флажки, а элемент управления `ListView` внешне будет напоминать `CheckedListBox`. Свойство `View` определяет макет `ListBox`. Свойство `Activation` определяет способ выделения вариантов списка. Дополнительная информация об этих свойствах и событии `ItemActivate` представлена на ил. 15.29.

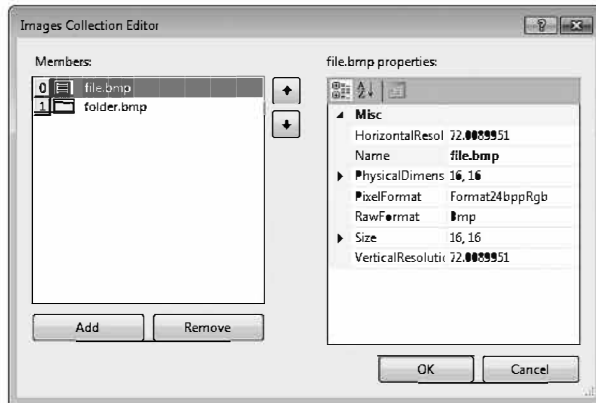
Элемент управления `ListView` позволяет определить значки, которые будут выводиться рядом с вариантами списка. Для отображения значков потребуется компонент `ImageList`. Чтобы создать его, перетащите компонент на форму с панели элементов. Затем выберите свойство `Images` в окне свойств, чтобы вызвать окно `Image Collection Editor` (ил. 15.30). В нем можно выбрать изображения для включения в коллекцию `ImageList`, содержащую массив объектов `Image`. При этом изображения встраиваются в приложение (по аналогии с ресурсами), чтобы их не приходилось распространять отдельно с приложением, — но при этом они не являются частью проекта. В нашем примере изображения включаются в `ImageList` на программном уровне. После

создания пустой коллекции `ImageList` включите изображения файлов и папок (из примеров этой главы) в проект в виде ресурсов. Затем задайте свойству `SmallImageList` объекта `ListView` новый объект `ImageList`. Свойство `SmallImageList` задает список изображений для мелких значков, а свойство `LargeImageList` задает список изображений для крупных значков. Каждый вариант `ListView` представляется объектом типа `ListViewItem`. Чтобы выбрать значок варианта `ListView`, следует задать соответствующий индекс свойству `ImageIndex` объекта варианта.

Свойства и события <code>ListView</code>	Описание
Часто используемые свойства	
<code>Activation</code>	Определяет способ активизации вариантов списка. Свойству задается значение из перечисления <code>ItemActivation</code> . Допустимые значения: <code>OneClick</code> (активизация одним щелчком), <code>TwoClick</code> (активизация двойным щелчком, вариант изменяет цвет при выделении) и <code>Standard</code> (используется по умолчанию; активизация двойным щелчком, вариант не изменяет цвет)
<code>CheckBoxes</code>	Указывает, должны ли выводиться флажки рядом с вариантами списка. По умолчанию используется значение <code>false</code>
<code>LargeImageList</code>	Определяет коллекцию <code>ImageList</code> с крупными значками
<code>Items</code>	Возвращает коллекцию <code>ListViewItems</code> элемента управления
<code>MultiSelect</code>	Определяет, разрешено ли множественное выделение. По умолчанию используется значение <code>true</code> (множественное выделение разрешено)
<code>SelectedItems</code>	Возвращает коллекцию выделенных вариантов в формате <code>ListView.SelectedItemsCollection</code>
<code>SmallImageList</code>	Определяет коллекцию <code>ImageList</code> с мелкими значками
<code>View</code>	Определяет внешний вид <code>ListViewItems</code> . Допустимые значения: <code>LargeIcon</code> (используется по умолчанию; крупные значки, варианты могут выводиться в несколько столбцов), <code>SmallIcon</code> (мелкие значки, варианты могут выводиться в несколько столбцов), <code>List</code> (мелкие значки, варианты выводятся в один столбец), <code>Details</code> (аналог <code>List</code> , но с возможностью вывода многостолбцовой информации о вариантах) и <code>Tile</code> (крупные значки, информация выводится справа от значка)
Часто используемые события	
<code>Click</code>	Генерируется при щелчке на варианте; является событием по умолчанию
<code>ImageActivate</code>	Генерируется при активизации вариантов <code>ListView</code> (одиночным или двойным щелчком). Не содержит информации о том, какой из вариантов активизирован, — для ее получения следует использовать <code>SelectedItems</code> или <code>SelectedIndices</code>

**Ил. 15.29.** Свойства и события `ListView`

Класс `ListViewTestForm` (ил. 15.31) отображает в элементе управления `ListView` файлы и папки вместе с мелкими значками, представляющими каждый файл или папку. Если файл или папка недоступны из-за настроек разрешений, выводится окно сообщения `MessageBox`. Программа анализирует содержимое каталога в процессе просмотра (в отличие от индексирования всего диска сразу).



**Ил. 15.30.** Окно Image Collection Editor для компонента `ImageList`

### Метод `ListViewTestForm_Load`

Метод `ListViewTestForm_Load` (строки 114–123) обрабатывает событие `Load` формы. При загрузке приложения значки папок и файлов включаются в коллекцию `Images` объекта `fileFolderImageList` (строки 117–118). Так как свойству `SmallImageList` объекта `ListView` задан объект `ImageList`, элемент управления `ListView` будет отображать эти изображения как значки каждого варианта. Значок папки был добавлен первым, поэтому он имеет индекс 0, а значку файла соответствует индекс 1. Приложение также загружает в `ListView` свой домашний каталог (полученный в строке 14) во время загрузки (строка 121) и выводит путь к каталогу (строка 122).

```

1  // Ил. 15.31: ListViewTestForm.cs
2  // Вывод каталогов и их содержимого в ListView.
3  using System;
4  using System.Windows.Forms;
5  using System.IO;
6
7  namespace ListViewTest
8  {
9      // Форма с элементом управления ListView для вывода
10     // папок и файлов в каталоге.
11     public partial class ListViewTestForm : Form
12     {
13         // Сохранение текущего каталога
14         string currentDirectory = Directory.GetCurrentDirectory();

```

**Ил. 15.31.** Файлы и папки `ListView` (продолжение ↗)

```

15
16 // Конструктор
17 public ListViewTestForm()
18 {
19     InitializeComponent();
20 } // Конец конструктора
21
22 // Просмотр каталога или подъем на один уровень
23 private void browserListView_Click( object sender, EventArgs e )
24 {
25     // Проверяем, что в списке есть выделенные варианты
26     if ( browserListView.SelectedItems.Count != 0 )
27     {
28         // Если выбран первый вариант, подняться на один уровень
29         if ( browserListView.Items[ 0 ].Selected )
30         {
31             // Создание объекта DirectoryInfo для каталога
32             DirectoryInfo directoryObject =
33                 new DirectoryInfo( currentDirectory );
34
35             // Если у каталога есть родитель, загрузить его
36             if ( directoryObject.Parent != null )
37             {
38                 LoadFilesInDirectory(
39                     directoryObject.Parent.FullName );
40             } // Конец if
41         } // Конец if
42
43         // В списке выделен каталог или файл
44         else
45         {
46             // Выбранный каталог или файл
47             string chosen = browserListView.SelectedItems[ 0 ].Text;
48
49             // Если выделен каталог, загрузить его содержимое
50             if ( Directory.Exists(
51                 Path.Combine( currentDirectory, chosen ) ) )
52             {
53                 LoadFilesInDirectory(
54                     Path.Combine( currentDirectory, chosen ) );
55             } // Конец if
56         } // Конец else
57
58         // Обновление displayLabel
59         displayLabel.Text = currentDirectory;
60     } // Конец if
61 } // Конец метода browserListView_Click
62
63 // Вывод файлов/подкаталогов текущего каталога
64 public void LoadFilesInDirectory( string currentDirectoryValue )
65 {
66     // Загрузка и вывод информации каталога
67     try
68     {
69         // Очистка ListView и ввод первого варианта в списке

```

**Ил. 15.31.** Файлы и папки ListView (продолжение ☞)

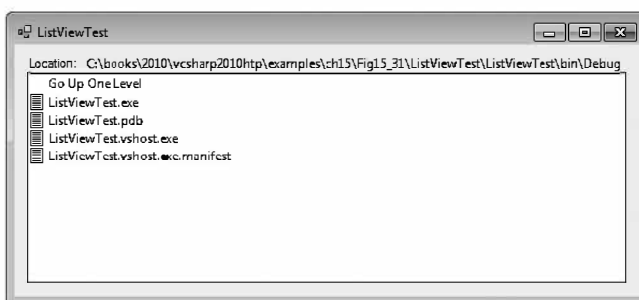
```

70     browserListView.Items.Clear();
71     browserListView.Items.Add( "Go Up One Level" );
72
73     // Обновление текущего каталога
74     currentDirectory = currentDirectoryValue;
75     DirectoryInfo newCurrentDirectory =
76         new DirectoryInfo( currentDirectory );
77
78     // Создание массивов файлов и каталогов
79     DirectoryInfo[] directoryArray =
80         newCurrentDirectory.GetDirectories();
81     FileInfo[] fileArray = newCurrentDirectory.GetFiles();
82
83     // Добавление имен каталогов в ListView
84     foreach ( DirectoryInfo dir in directoryArray )
85     {
86         // Добавление каталога в ListView
87         ListViewItem newDirectoryItem =
88             browserListView.Items.Add( dir.Name );
89
90         newDirectoryItem.ImageIndex = 0; // Изображение для каталога
91     } // Конец foreach
92
93     // Добавление имен файлов в ListView
94     foreach ( FileInfo file in fileArray )
95     {
96         // Добавление файла в ListView
97         ListViewItem newFileItem =
98             browserListView.Items.Add( file.Name );
99
100        newFileItem.ImageIndex = 1; // Изображение для файла
101    } // Конец foreach
102 } // Конец try
103
104 // Отказано в доступе
105 catch ( UnauthorizedAccessException )
106 {
107     MessageBox.Show( "Warning: Some files may not be " +
108         "visible due to permission settings",
109         "Attention", 0, MessageBoxIcon.Warning );
110 } // Конец catch
111 } // Конец метода LoadFilesInDirectory
112
113 // Обработка события загрузки формы
114 private void ListViewTestForm_Load( object sender, EventArgs e )
115 {
116     // Добавление значков в ImageList
117     fileFolderImageList.Images.Add( Properties.Resources.folder );
118     fileFolderImageList.Images.Add( Properties.Resources.file );
119
120     // Загрузка текущего каталога в browserListView
121     LoadFilesInDirectory( currentDirectory );
122     displayLabel.Text = currentDirectory;
123 } // Конец метода ListViewTestForm_Load
124 } // Конец класса ListViewTestForm
125 } // Конец пространства имен ListViewTest

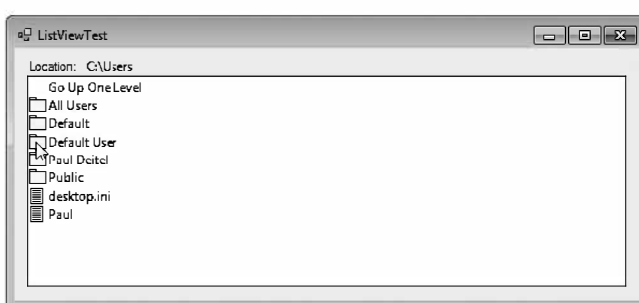
```

**Ил. 15.31.** Файлы и папки ListView (продолжение ↗)

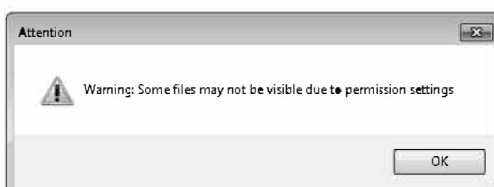
- а) Приложение с содержимым папки по умолчанию



- б) Приложение с содержимым каталога C:\Users



- в) Диалоговое окно, выводимое при попытке обращения к каталогу, для которого пользователь не имеет прав доступа



**Ил. 15.31.** Файлы и папки ListView (окончание)

## Метод LoadFilesInDirectory

Метод `LoadFilesInDirectory` (строки 64–111) заполняет `browserListView` данными переданного каталога (`currentDirectoryValue`). Он очищает элемент управления `browserListView` и добавляет в него вариант "Go Up One Level". Когда пользователь щелкает на этом варианте, программа пытается подняться на один уровень (см. далее). Затем метод создает объект `DirectoryInfo`, инициализированный строкой `currentDirectory` (строки 75–76). Если пользователь не имеет разрешений для просмотра каталога, происходит исключение (которое перехватывается в строке 105). Метод `LoadFilesInDirectory` несколько отличается от метода `PopulateTreeView` из предыдущей программы (см. ил. 15.28): вместо загрузки всех папок на жестком диске метод `LoadFilesInDirectory` загружает только папки в текущем каталоге.

Класс `DirectoryInfo` (пространство имен `System.IO`) предоставляет средства для просмотра структуры каталогов и работы с ней. Метод `GetDirectories` (строка 80) возвращает массив объектов `DirectoryInfo` с подкаталогами текущего каталога. Аналогичный метод `GetFiles` (строка 81) возвращает массив объектов класса `FileInfo` с информацией о файлах текущего каталога. Свойство `Name` (и класса `DirectoryInfo`,

и класса `FileInfo`) содержит только имя каталога или файла (например, `temp` вместо `C:\myfolder\temp`). Для обращения к полному имени используется свойство `FullName`.

В строках 84–91 и 94–101 перебираются подкаталоги и файлы текущего каталога, которые добавляются в `browserListView`. В строках 90 и 100 задаются свойства `ImageIndex` вновь созданных элементов. Для каталогов значком назначается изображение каталога (индекс 0), а для файлов — изображение файла (индекс 1).

### Метод `browserListView_Click`

Метод `browserListView_Click` (строки 23–61) вызывается при щелчке на элементе управления `browserListView`. Строка 26 проверяет наличие выделенных вариантов. Если выделение имеется, строка 29 проверяет, был ли выделен первый вариант в `browserListView`. Первым вариантом `browserListView` всегда является команда перехода на верхний уровень. В строках 32–33 создается объект `DirectoryInfo` для текущего каталога. Строка 36 проверяет свойство `Parent`, чтобы убедиться в том, что пользователь не находится в корне дерева каталогов. Свойство `Parent` возвращает родительский каталог в виде объекта `DirectoryInfo`; если родительский каталог не существует, `Parent` возвращает `null`. Если родительский каталог существует, то в строках 38–39 полное имя родительского каталога передается `LoadFilesInDirectory`.

Если пользователь не выбрал первый вариант `browserListView`, то строки 44–56 позволяют пользователю продолжить перемещение по структуре каталогов. В строке 47 создается строка `chosen`, которой присваивается текст выделенного варианта (первый элемент коллекции `SelectedItem`s). Строки 50–51 определяют, был ли выделен действительный каталог (вместо файла). При помощи метода `Combine` класса `Path` программа объединяет строки `currentDirectory` и `chosen` для образования нового пути. Метод `Combine` автоматически добавляет обратную косую черту (`\`) между двумя фрагментами, если это необходимо. Значение передается методу `Exists` класса `Directory`. Метод `Exists` возвращает `true`, если его параметр `string` представляет действительный каталог; в этом случае программа передает строку методу `LoadFilesInDirectory` (строки 53–54). Наконец, надпись `displayLabel` обновляется данными нового каталога (строка 59).

Программа загружается быстро, потому что она индексирует файлы только в текущем каталоге. При загрузке нового каталога возможна небольшая задержка. Кроме того, для отображения изменений в структуре каталогов достаточно выполнить простую перезагрузку каталога. В предыдущей программе (см. ил. 15.28) возможна довольно заметная исходная задержка, поскольку она загружает всю структуру каталогов. Подобные альтернативы типичны для программирования.



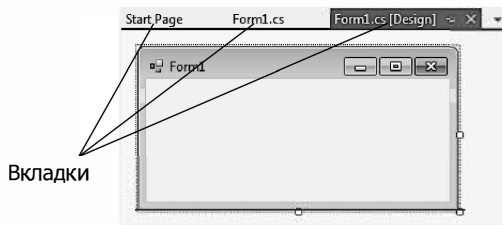
### АРХИТЕКТУРНОЕ РЕШЕНИЕ 15.2

Если ваше приложение рассчитано на выполнение в течение длительного времени, стоит отдать предпочтение большой исходной задержке с высоким быстродействием во время работы программы. Для приложений с малым временем выполнения разработчики часто предпочитают быструю начальную загрузку с небольшими задержками после каждой операции.

## 15.11. Элемент управления TabControl

Элемент управления `TabControl` создает окна с вкладками наподобие тех, которые используются в Visual Studio (ил. 15.32). Вкладки позволяют разместить больше информации на форме и организовать логическую группировку данных. Объект `TabControl` содержит объекты `TabPage`, которые, как и объекты `Panel` и `GroupBox`, могут содержать другие элементы управления. Сначала вы добавляете элементы управления в объекты `TabPage`, а затем добавляете `TabPage` в `TabControl`. В любой момент времени отображается только один объект `TabPage`. Добавление объектов в `TabPage` и `TabControl` выполняется следующими командами:

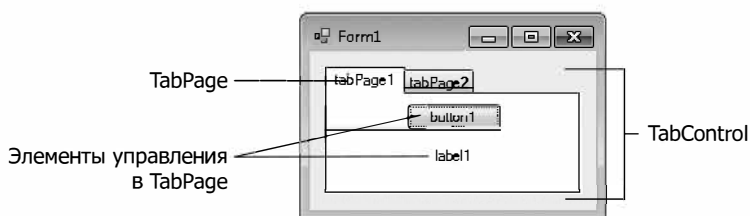
```
myTabPage.Controls.Add( myControl );  
myTabControl.TabPages.Add( myTabPage );
```



Ил. 15.32. Окно со вкладками в Visual Studio

Эти команды вызывают метод `Add` коллекции `Controls` и метод `Add` коллекции `TabPages`. В приведенном примере объект `TabControl` с именем `myControl` добавляется в объект `TabPage` с именем `myTabPage`, после чего `myTabPage` добавляется в `myTabControl`. Также можно воспользоваться методом `AddRange` для добавления массива `TabPages` или элементов управления в `TabControl` или `TabPage` соответственно.

На ил. 15.33 изображен пример элемента управления `TabControl`.

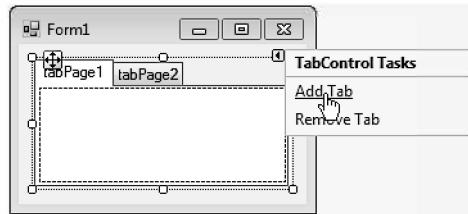


Ил. 15.33. Элемент управления `TabControl` с несколькими объектами `TabPage`

Объекты `TabControl` также можно добавлять визуально, перетаскивая их с формы в режиме конструктора. Чтобы добавить объект `TabPage` в режиме конструктора, щелкните в верхней части `TabControl`, откройте меню быстрых действий и выберите команду `Add Tab` (ил. 15.34). Также можно щелкнуть на свойстве `TabPages` в окне свойств и добавить вкладки в открывшемся диалоговом окне. Чтобы изменить надпись на вкладке, задайте свойство `Text` объекта `TabPage`. Щелчок на вкладке



выделяет объект `TabControl` — чтобы выделить `TabPage`, щелкните на области элемента под вкладками. Также можно добавить элементы управления в `TabPage`, перетаскивая их с панели элементов. Чтобы просматривать разные объекты `TabPage`, щелкайте на соответствующих вкладках (в режиме конструктора или во время выполнения). Часто используемые свойства и событие класса `TabControl` перечислены на ил. 15.35.



**Ил. 15.34.** `TabPage`s добавляется к `TabControl`

Свойства и событие <code>TabControl</code>	Описание
Часто используемые свойства	
<code>ImageList</code>	Изображения, выводимые на вкладках
<code>ItemSize</code>	Размер вкладки
<code>Multiline</code>	Указывает, допускается ли вывод вкладок в несколько строк
<code>SelectedIndex</code>	Индекс выделенного объекта <code>TabPage</code>
<code>SelectedTab</code>	Выделенный объект <code>TabPage</code>
<code>TabCount</code>	Возвращает количество вкладок
<code>TabPage</code> s	Возвращает коллекцию объектов <code>TabPage</code> , содержащихся в <code>TabControl</code> , в виде <code>TabControl.TabPageCollection</code>
Часто используемое событие	
<code>SelectedIndexChanged</code>	Генерируется при изменении <code>SelectedIndex</code> (то есть при выделении другого объекта <code>TabPage</code> )

**Ил. 15.35.** Свойства и событие `TabControl`

Каждый объект `TabPage` генерирует событие `Click` при щелчке на своей вкладке. Обработчики этих событий можно создать двойным щелчком на теле `TabPage`.

Класс `UsingTabsForm` (ил. 15.36) использует элемент управления `TabControl` для вывода различных атрибутов текста надписи (`Color`, `Size` и `Message`). На последней вкладке выводится сообщение об использовании `TabControl`.

```

1 // Ил. 15.36: UsingTabsForm.cs
2 // Использование TabControl для отображения атрибутов шрифта.
3 using System;
4 using System.Drawing;
5 using System.Windows.Forms;
6
7 namespace UsingTabs
8 {
9     // Форма использует вкладки и переключатели для настройки шрифта.
10     public partial class UsingTabsForm : Form

```

**Ил. 15.36.** Использование `TabControl` для настройки шрифта (продолжение ➤)

```
11 {
12     // Конструктор
13     public UsingTabsForm()
14     {
15         InitializeComponent();
16     } // Конец конструктора
17
18     // Обработчик события переключателя Black
19     private void blackRadioButton_CheckedChanged(
20         object sender, EventArgs e )
21     {
22         displayLabel.ForeColor = Color.Black; // Выбор черного цвета
23     } // Конец метода blackRadioButton_CheckedChanged
24
25     // Обработчик события переключателя Red
26     private void redRadioButton_CheckedChanged(
27         object sender, EventArgs e )
28     {
29         displayLabel.ForeColor = Color.Red; // Выбор красного цвета
30     } // Конец метода redRadioButton_CheckedChanged
31
32     // Обработчик события переключателя Green
33     private void greenRadioButton_CheckedChanged(
34         object sender, EventArgs e )
35     {
36         displayLabel.ForeColor = Color.Green; // Выбор зеленого цвета
37     } // Конец метода greenRadioButton_CheckedChanged
38
39     // Обработчик события переключателя 12 point
40     private void size12RadioButton_CheckedChanged(
41         object sender, EventArgs e )
42     {
43         // Выбор размера шрифта 12 пунктов
44         displayLabel.Font = new Font( displayLabel.Font.Name, 12 );
45     } // Конец метода size12RadioButton_CheckedChanged
46
47     // Обработчик события переключателя 16 point
48     private void size16RadioButton_CheckedChanged(
49         object sender, EventArgs e )
50     {
51         // Выбор размера шрифта 16 пунктов
52         displayLabel.Font = new Font( displayLabel.Font.Name, 16 );
53     } // Конец метода size16RadioButton_CheckedChanged
54
55     // Обработчик события переключателя 20 point
56     private void size20RadioButton_CheckedChanged(
57         object sender, EventArgs e )
58     {
59         // Выбор размера шрифта 20 пунктов
60         displayLabel.Font = new Font( displayLabel.Font.Name, 20 );
61     } // Конец метода size20RadioButton_CheckedChanged
62
63     // Обработчик события переключателя Hello!
64     private void helloRadioButton_CheckedChanged(
```

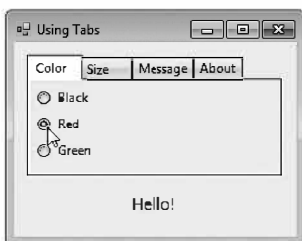
**Ил. 15.36.** Использование TabControl для настройки шрифта (продолжение ↗)

```

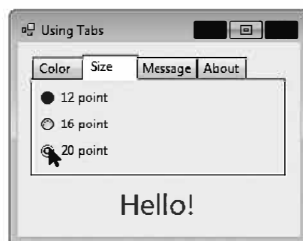
65     object sender, EventArgs e )
66     {
67         displayLabel.Text = "Hello!"; // Вывод текста "Hello!"
68     } // Конец метода helloRadioButton_CheckedChanged
69
70     // Обработчик события переключателя Goodbye!
71     private void goodbyeRadioButton_CheckedChanged(
72         object sender, EventArgs e )
73     {
74         displayLabel.Text = "Goodbye!"; // Вывод текста "Goodbye!"
75     } // Конец метода goodbyeRadioButton_CheckedChanged
76 } // Конец класса UsingTabsForm
77 } // Конец пространства имен UsingTabs

```

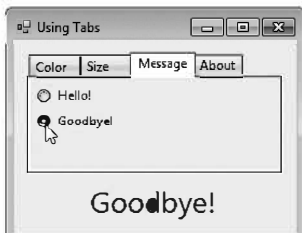
а) Установка переключателя Red на вкладке Color



б) Установка переключателя 20 point на вкладке Size



в) Установка переключателя Goodbye! на вкладке Message



г) Переход на вкладку Above



**Ил. 15.36.** Использование TabControl для настройки шрифта (окончание)

Объекты `textOptionsTabControl`, `colorTabPage`, `sizeTabPage`, `messageTabPage` и `aboutTabPage` создаются в режиме конструктора (см. ранее). Вкладка `colorTabPage` содержит три переключателя для черного (`blackRadioButton`), красного (`redRadioButton`) и зеленого цвета (`greenRadioButton`). Эта вкладка изображена на ил. 15.36, а. Обработчик события `CheckedChanged` каждого переключателя обновляет цвет текста `displayLabel` (строки 22, 29 и 36). Вкладка `sizeTabPage` (ил. 15.36, б) содержит три переключателя для размеров шрифта 12 (`size12RadioButton`), 16 (`size16RadioButton`) и 20 пунктов (`size20RadioButton`); изменение размера шрифта происходит в строках 44, 52 и 60 соответственно. Вкладка `messageTabPage` (ил. 15.36, в) содержит два переключателя для сообщений *Hello!* (`helloRadioButton`) и *Goodbye!* (`goodbyeRadioButton`). Эти переключатели определяют текст `displayLabel` (строки 67 и 74 соответственно). Вкладка `aboutTabPage` (ил. 15.36, г) содержит надпись (`messageLabel`) с кратким описанием `TabControl`.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 15.3

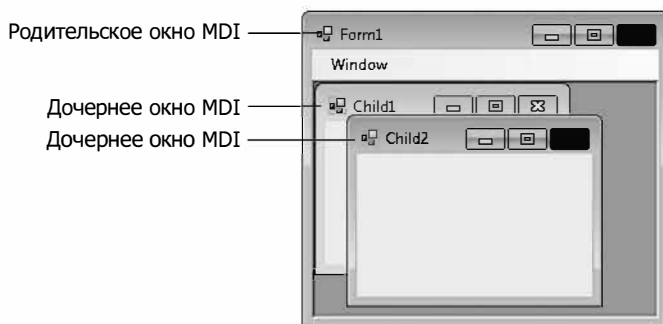
Элемент `TabPage` может выполнять функции контейнера для одной логической группы переключателей, обеспечивая их взаимоисключающую установку. Чтобы разместить на одной вкладке `TabPage` несколько групп переключателей, сгруппируйте их в контейнерах `Panel` или `GroupBox` на этой вкладке.

## 15.12. Окна MDI

В предыдущих главах мы создавали только приложения с *однодокументным* интерфейсом (SDI, Single Document Interface). В таких программах в любой момент времени открыто только одно окно, или *документ*. Как правило, SDI-программы обладают ограниченными возможностями — как, например, стандартные приложения Paint и Блокнот. Для работы с несколькими документами пользователь должен запустить несколько экземпляров приложения.

Многие современные приложения используют *многодокументный* интерфейс MDI (Multiple Document Interface), позволяющий работать сразу с несколькими документами, — как, например, программы из пакета Microsoft Office. Программы с интерфейсом MDI обычно обладают расширенными возможностями — так, Paint Shop Pro и Photoshop по возможностям редактирования изображений превосходят Paint.

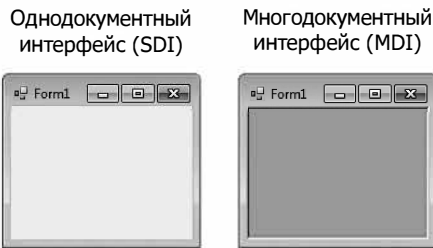
Главное окно многодокументной программы называется *родительским окном*, а все внутренние окна называются *дочерними окнами*. Хотя MDI-приложение может иметь много дочерних окон, родительское окно всегда только одно. Кроме того, в любой момент времени активным может быть только одно дочернее окно. Дочерние окна не могут сами быть родительскими и не могут перемещаться за границу своего родителя. В остальном (в отношении закрытия, изменения размеров и т. д.) дочернее окно ничем не отличается от любого другого окна. Функциональность дочернего окна может отличаться от функциональности других дочерних окон. Например, в одном дочернем окне пользователь может редактировать графику, в другом — вводить текст, а в третьем наблюдать за сетевым трафиком; при этом все окна принадлежат одному MDI-родителю. На ил. 15.37 изображен пример MDI-приложения с двумя дочерними окнами.



**Ил. 15.37.** MDI-приложение с родительским и дочерними окнами

Чтобы создать форму MDI, создайте новую форму и задайте ее свойству `IsMdiContainer` значение `true`. Внешний вид формы изменяется, как показано на ил. 15.38. Затем создайте класс дочерней формы. Для этого щелкните правой кнопкой мыши на проекте в окне *Solution Explorer*, выберите команду **Project** ► **Add Windows Form...** и введите имя файла. Отредактируйте форму так, как потребуется. Чтобы добавить дочернюю форму в родительскую, необходимо создать новый объект дочерней формы, задать его свойству `MdiParent` объект родительской формы, а затем вызвать метод `Show` дочерней формы. В обобщенном виде добавление дочерней формы реализуется следующим образом:

```
ChildFormClass childForm = New ChildFormClass();
childForm.MdiParent = parentForm;
childForm.Show();
```



**Ил. 15.38.** Формы с интерфейсом SDI и MDI

В большинстве случаев родительская форма создает дочернюю, поэтому в *parentForm* используется `this`. Код создания дочерней формы обычно находится в обработчике события, который создает новое окно в ответ на действие пользователя. Для создания новых дочерних окон чаще всего используются команды меню (например, **File** ► **New** ► **Window**).

Свойство `MdiChildren` класса `Form` возвращает массив ссылок на дочерние формы. Например, эта информация может использоваться для проверки состояния всех дочерних окон (допустим, проверки того, что находящаяся в них информация была сохранена перед закрытием родительского окна). Свойство `ActiveMdiChild` возвращает ссылку на активное дочернее окно; при отсутствии активных дочерних окон возвращается `null`. Другие возможности окон MDI перечислены на ил. 15.39.

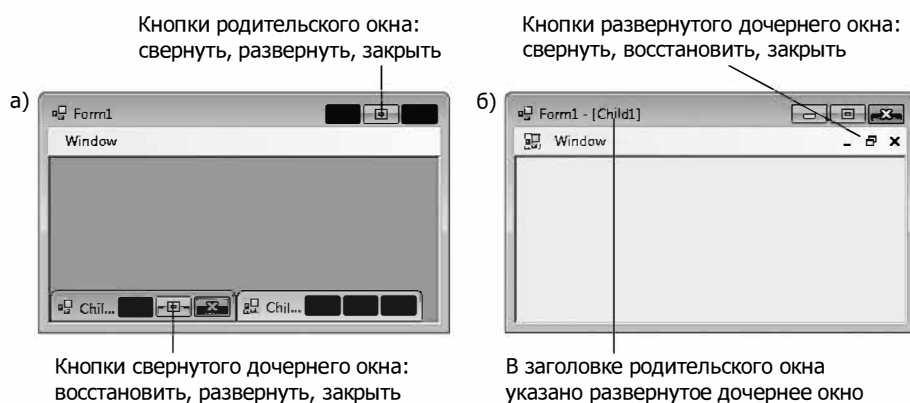
Свойства, метод и событие форм MDI	Описание
Часто используемые свойства дочерних окон MDI	
<code>IsMdiChild</code>	Если значение свойства истинно, то форма является дочерней формой MDI (доступно только для чтения)
<code>MdiParent</code>	Задаёт родительскую форму MDI для дочерней формы

**Ил. 15.39.** Свойства, метод и событие родительских и дочерних окон MDI (продолжение ↗)

Свойства, метод и событие форм MDI	Описание
Часто используемые свойства родительских окон MDI	
ActiveMdiChild	Возвращает текущую активную дочернюю форму MDI (null, если ни одна дочерняя форма не активна)
IsMdiContainer	Если значение истинно, то форма может быть родителем MDI. По умолчанию используется значение false
MdiChildren	Возвращает список дочерних форм MDI в массиве объектов Form
Часто используемый метод	
LayoutMdi	Определяет способ размещения дочерних форм в родительской форме MDI. В параметре передается значение из перечисления MDILayout с допустимыми значениями ArrangeIcons, Cascade, TileHorizontal и TileVertical. Разные способы размещения изображены на илл. 15.42
Часто используемое событие	
MdiChildActivate	Генерируется при закрытии или активизации дочерней формы MDI

**Ил. 15.39.** Свойства, метод и событие родительских и дочерних окон MDI (окончание)

Дочерние окна можно сворачивать, разворачивать и закрывать в родительском окне независимо друг от друга. На ил. 15.40 представлены два варианта макета: с двумя свернутыми дочерними окнами и с развернутым дочерним окном. При сворачивании или закрытии родительского окна дочерние окна также сворачиваются или закрываются.



**Ил. 15.40.** Свернутое или развернутое дочернее окно

Обратите внимание за заголовок окна на ил. 15.40, б; в нем отображается текст **Form1 - [Child1]**. Когда дочернее окно разворачивается, текст из его заголовка вставляется

в заголовок родительского окна. Когда дочернее окно находится в свернутом или развернутом состоянии, в его заголовке выводится кнопка, которая возвращает дочернее окно к предыдущему размеру (до того, как окно было свернуто или развернуто).

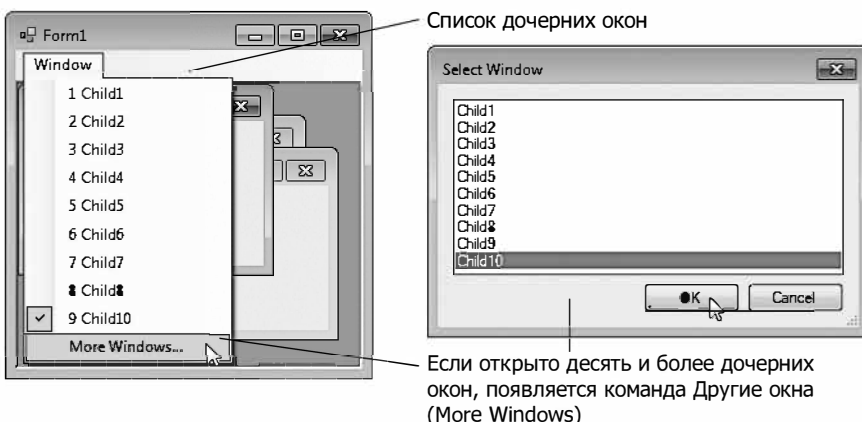
В C# имеется свойство для отслеживания дочерних окон, открытых в контейнере MDI. Свойство `MdiWindowListItem` компонента `MenuStrip` указывает, какое меню (если оно имеется) выводит список открытых дочерних окон, чтобы пользователь мог вывести нужное окно на передний план. При открытии нового дочернего окна соответствующая запись добавляется в конец списка (ил. 15.41). Если открыто десять и более дочерних окон, в список включается команда Другие окна (More Windows), которая позволяет выбрать окно из списка в диалоговом окне.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 15.1

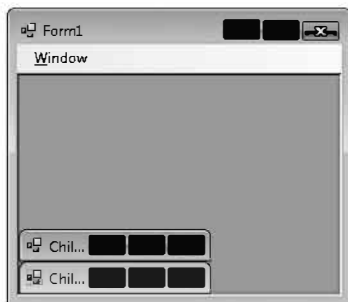
Создавая MDI-приложение, включайте в них меню для вывода списка открытых дочерних окон. Пользователь сможет быстро перейти к нужному дочернему окну вместо того, чтобы разыскивать его в родительском окне.

Контейнеры MDI позволяют управлять размещением своих дочерних окон. Дочерние окна MDI-приложения размещаются вызовом метода `LayoutMdi` родительской формы. Метод `LayoutMdi` получает значение из перечисления `MdiLayout`, состоящего из значений `ArrangeIcons`, `Cascade`, `TileHorizontal` и `TileVertical`. При *мозаичном* размещении родительское окно заполняется без перекрытия; такие окна могут размещаться по горизонтали (`TileHorizontal`) или по вертикали (`TileVertical`). В режиме *каскадного* заполнения (`Cascade`) окна перекрываются — все они имеют одинаковый размер и отображают заголовок, если это возможно. В режиме `ArrangeIcons` для всех свернутых дочерних окон отображаются значки. Если свернутые окна разбросаны по родительскому окну, в режиме `ArrangeIcons` они аккуратно выстраиваются в левом нижнем углу родительского окна. На ил. 15.42 продемонстрированы разные значения из перечисления `MdiLayout`.

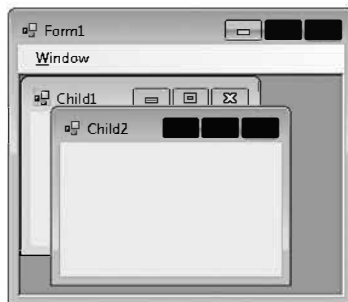


Ил. 15.41. Пример свойства `MdiWindowListItem` компонента `MenuStrip`

a) ArrangeIcons



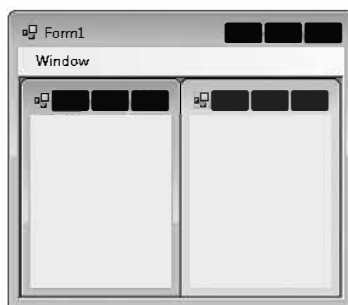
б) Cascade



в) TileHorizontal



г) TileVertical

**Ил. 15.42.** Значения из перечисления MdiLayout

Класс UsingMDIForm (ил. 15.43) демонстрирует использование окон MDI. Он открывает три экземпляра дочерней формы ChildForm (ил. 15.44), каждый из которых содержит элемент управления PictureBox с изображением. Родительская форма MDI содержит меню с командами создания и размещения дочерних форм.

### Родительская форма MDI

На ил. 15.43 представлен класс UsingMDIForm — родительская форма MDI-приложения. Эта форма, которая создается в самом начале работы, содержит два меню верхнего уровня. Первое меню File (fileToolStripMenuItem) содержит команду Exit (exitToolStripMenuItem) и подменю New (newToolStripMenuItem) с командами для всех типов дочерних окон. Второе меню Window (windowToolStripMenuItem) предоставляет команды для размещения дочерних окон MDI, а также список дочерних окон MDI.

```

1 // Ил. 15.43: UsingMDIForm.cs
2 // Использование родительских и дочерних окон MDI.
3 using System;
4 using System.Windows.Forms;
5
6 namespace UsingMDI

```

**Ил. 15.43.** Класс родительского окна MDI (продолжение ↗)



```

7 {
8     // Форма для демонстрации использования окон MDI.
9     public partial class UsingMDIForm : Form
10    {
11        // Конструктор
12        public UsingMDIForm()
13        {
14            InitializeComponent();
15        } // Конец конструктора
16
17        // Создание окна Lavender Flowers
18        private void lavenderToolStripMenuItem_Click(
19            object sender, EventArgs e )
20        {
21            // Создание нового дочернего окна
22            ChildForm child = new ChildForm(
23                "Lavender Flowers", "lavenderflowers" );
24            child.MdiParent = this; // Назначение родителя
25            child.Show(); // Отображение дочернего окна
26        } // Конец метода lavenderToolStripMenuItem_Click
27
28        // Создание окна Purple Flowers
29        private void purpleToolStripMenuItem_Click(
30            object sender, EventArgs e )
31        {
32            // Создание нового дочернего окна
33            ChildForm child = new ChildForm(
34                "Purple Flowers", "purpleflowers" );
35            child.MdiParent = this; // Назначение родителя
36            child.Show(); // Отображение дочернего окна
37        } // Конец метода purpleToolStripMenuItem_Click
38
39        // Создание окна Yellow Flowers
40        private void yellowToolStripMenuItem_Click(
41            object sender, EventArgs e )
42        {
43            // Создание нового дочернего окна
44            Child child = new ChildForm(
45                "Yellow Flowers", "yellowflowers" );
46            child.MdiParent = this; // Назначение родителя
47            child.Show(); // Отображение дочернего окна
48        } // Конец метода yellowToolStripMenuItem_Click
49
50        // Выход из приложения
51        private void exitToolStripMenuItem_Click(
52            object sender, EventArgs e )
53        {
54            Application.Exit();
55        } // Конец метода exitToolStripMenuItem_Click
56
57        // Выбор каскадного размещения
58        private void cascadeToolStripMenuItem_Click(
59            object sender, EventArgs e )

```

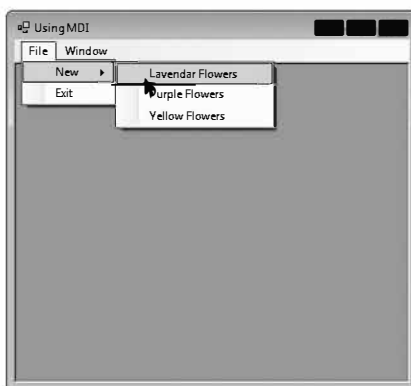
**Ил. 15.43.** Класс родительского окна MDI (продолжение ↗)

```

60     {
61         this.LayoutMdi( MdiLayout.Cascade );
62     } // Конец метода cascadeToolStripMenuItem_Click
63
64     // Выбор горизонтального мозаичного размещения
65     private void tileHorizontalToolStripMenuItem_Click(
66         object sender, EventArgs e )
67     {
68         this.LayoutMdi( MdiLayout.TileHorizontal );
69     } // Конец метода tileHorizontalToolStripMenuItem_Click
70
71     // Выбор вертикального мозаичного размещения
72     private void tileVerticalToolStripMenuItem_Click(
73         object sender, EventArgs e )
74     {
75         this.LayoutMdi( MdiLayout.TileVertical );
76     } // Конец метода tileVerticalToolStripMenuItem_Click
77 } // Конец класса UsingMDIForm
78 } // Конец пространства имен UsingMDI

```

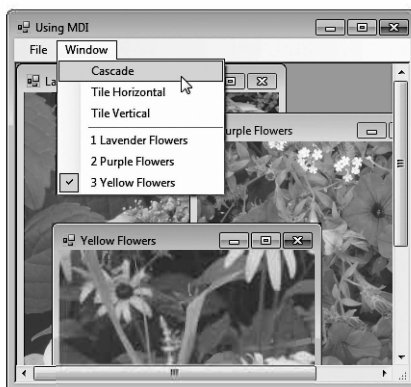
а) Выбор команды меню Lavender Flowers



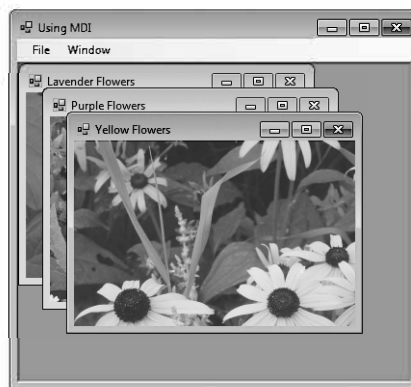
б) Приложение с окном Lavender Flowers



в) Выбор каскадного размещения



г) Каскадные окна в окне MDI



Ил. 15.43. Класс родительского окна MDI (окончание)

В окне свойств мы задаем свойству `IsMdiContainer` формы значение `true`, в результате чего форма становится родителем MDI. Кроме того, свойству `MdiWindowListItem` компонента `MenuStrip` задается значение `windowToolStripMenuItem`, чтобы в меню `Window` выводился список дочерних окон MDI.

С командой меню `Cascade` (`cascadeToolStripMenuItem`) связан обработчик события (`cascadeToolStripMenuItem_Click`, строки 58–62) для каскадного размещения дочерних окон. Обработчик события вызывает метод `LayoutMdi` с аргументом `Cascade` из перечисления `MdiLayout` (строка 61).

С командой меню `Tile Horizontal` (`tileHorizontalToolStripMenuItem`) связан обработчик события (`tileHorizontalToolStripMenuItem_Click`, строки 65–69), который размещает дочерние окна по горизонтали. Обработчик события вызывает метод `LayoutMdi` с аргументом `TileHorizontal` из перечисления `MdiLayout` (строка 68).

Наконец, с командой меню `Tile Vertical` (`tileVerticalToolStripMenuItem`) связывается обработчик события (`tileVerticalToolStripMenuItem_Click`, строки 72–76), который размещает дочерние окна по вертикали. Обработчик события вызывает метод `LayoutMdi` с аргументом `TileVertical` из перечисления `MdiLayout` (строка 75).

### Дочерняя форма MDI

На этой стадии приложение еще не готово — необходимо определить класс дочерней формы MDI. Для этого щелкните правой кнопкой мыши на проекте в окне `Solution Explorer` и выберите команду `Add ► Windows Form`. Введите в открывшемся диалоговом окне имя нового класса `ChildForm` (ил. 15.44) и добавьте на форму `ChildForm` элемент управления `PictureBox` (`displayPictureBox`). В конструкторе `ChildForm` строка 16 задает текст заголовка. В строках 19–21 изображение загружается из ресурсов приложения, преобразуется в `Image` и задается свойству `Image` объекта `displayPictureBox`. Используемые изображения находятся в папке `Images` каталога примеров этой главы.

После определения класса дочерней формы MDI родительская форма (ил. 15.43) может создавать новые дочерние окна. Обработчики событий в строках 18–48 создают новую дочернюю форму, соответствующую выбранной команде меню. В строках 22–23, 33–34 и 44–45 создаются новые экземпляры `ChildForm`. Строки 24, 35 и 46 задают свойству `MdiParent` каждой дочерней формы ссылку на родительскую форму. В строках 25, 36 и 47 дочерняя форма отображается на экране вызовом метода `Show`.

```

1  // Ил. 15.44: ChildForm.cs
2  // Класс дочернего окна MDI.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace UsingMDI
8  {
9      public partial class ChildForm : Form
10     {
11         public ChildForm( string title, string resourceName )

```

**Ил. 15.44.** Дочерняя форма MDI (продолжение ⌘)

```
12     {
13         // Необходимо для поддержки режима конструктора Windows Form
14         InitializeComponent();
15
16         Text = title; // Текст заголовка
17
18         // set image to display in PictureBox
19         displayPictureBox.Image =
20             ( Image ) ( Properties.Resources.ResourceManager.GetObject(
21                 resourceName );
22     } // Конец конструктора
23 } // Конец класса ChildForm
24 } // Конец пространства имен UsingMDI
```

**Ил. 15.44.** Дочерняя форма MDI (окончание)

## 15.13. Визуальное наследование

В главе 11 вы узнали, как наследование используется для создания новых классов на базе существующих классов. Мы также использовали наследование для создания форм с графическим интерфейсом, объявляя новые классы форм производными от класса `System.Windows.Forms.Form`. Класс производной формы содержит всю функциональность базового класса формы, включая все свойства, методы, переменные и элементы управления базового класса. Кроме того, он наследует от базового класса все визуальные аспекты: размеры, расположение компонентов, расстояния между компонентами, цвета и шрифты. Это называется *визуальным наследованием*.

Визуальное наследование помогает достичь визуального единства оформления приложений. Например, можно определить базовую форму с логотипом продукта, конкретным цветом фона, заранее определенной строкой меню и другими элементами. Далее базовая форма используется для создания производных форм, обеспечивая единство оформления и фирменного стиля. Также можно создавать элементы управления, производные от других элементов управления.

### Создание базовой формы

Класс `VisualInheritanceBaseForm` (ил. 15.45) объявляется производным от `Form`. Графический интерфейс приложения состоит из двух надписей с текстом *Bugs, Bugs, Bugs* и *Copyright 2014, by Deitel & Associates, Inc.*, а также кнопки с текстом *Learn More*. Когда пользователь нажимает кнопку *Learn More*, вызывается метод `learnMoreButton_Click` (строки 18–24), который открывает окно сообщения с дополнительной информацией.

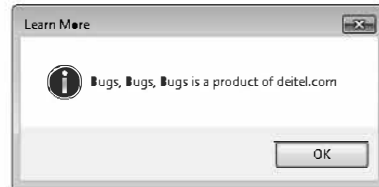
```
1 // Ил. 15.45: VisualInheritanceBaseForm.cs
2 // Базовая форма для визуального наследования.
3 using System;
4 using System.Windows.Forms;
```

**Ил. 15.45.** Класс `VisualInheritanceBaseForm`, производный от класса `Form`, содержит кнопку *Learn More* (продолжение ↗)

```

5
6 namespace VisualInheritanceBase
7 {
8     // Базовая форма для демонстрации визуального наследования
9     public partial class VisualInheritanceBaseForm : Form
10    {
11        // Конструктор
12        public VisualInheritanceForm()
13        {
14            InitializeComponent();
15        } // Конец конструктора
16
17        // Вывод окна MessageBox при щелчке на кнопке
18        private void learnMoreButton_Click( object sender, EventArgs e )
19        {
20            MessageBox.Show(
21                "Bugs, Bugs, Bugs is a product of deitel.com",
22                "Learn More", MessageBoxButtons.OK,
23                MessageBoxIcon.Information );
24        } // Конец метода learnMoreButton_Click
25    } // Конец класса VisualInheritanceBaseForm
26 } // Конец пространства имен VisualInheritanceBase

```



**Ил. 15.45.** Класс VisualInheritanceBaseForm, производный от класса Form, содержит кнопку Learn More (окончание)

### Последовательность действий по объявлению и использованию класса

Чтобы форма (или любой другой класс) могла использоваться в других приложениях, ее необходимо включить в библиотеку классов. Чтобы создать класс, предназначенный для повторного использования, выполните следующие действия:

1. Объявите открытый класс. Если класс не является открытым, он может использоваться только другими классами из той же сборки (то есть откомпилированных в том же DLL- или EXE-файле).
2. Выберите пространство имен и добавьте объявление пространства в файл исходного кода с объявлением своего класса.
3. Откомпилируйте класс в библиотеку классов.
4. Включите ссылку на библиотеку классов в приложение.

Рассмотрим каждый из этих шагов в контексте нашего примера.

### Шаг 1. Создание открытого класса

На этом шаге мы будем использовать открытый класс `VisualInheritanceBaseForm`, объявленный в листинге на ил. 15.45. По умолчанию каждый новый класс формы, созданный вами, объявляется открытым.

### Шаг 2. Добавление пространства имен

На шаге 2 будет использовано объявление пространства имен, автоматически созданное IDE. По умолчанию каждый новый класс помещается в пространство имен, имя которого совпадает с именем проекта. Вы видели, что практически во всех рассмотренных примерах классы из существующих библиотек (например, .NET Framework Class Library) могут импортироваться в приложение C#. Каждый класс принадлежит пространству имен, содержащему набор взаимосвязанных классов. С повышением сложности приложений пространства имен помогают справиться со сложностью компонентов. Библиотеки классов и пространства имен тоже упрощают повторное использование кода — приложения могут добавлять классы из других пространств имен (как это делалось в большинстве наших примеров). В предыдущих главах объявления пространств имен удалялись из листингов, так как в них не было необходимости.

Включение класса в объявление пространства имен означает, что класс является частью указанного пространства. Имя пространства является частью полного имени класса, так что класс `VisualInheritanceTestForm` на самом деле называется `VisualInheritanceBase.VisualInheritanceBaseForm`. В своих приложениях вы либо используете это полное имя, либо пишете директиву `using` и используете простое имя класса (`VisualInheritanceBaseForm`). Если другое пространство имен содержит класс с таким же именем, полное имя класса поможет вам различить классы и предотвратит конфликт имен.

### Шаг 3. Компиляция библиотеки классов

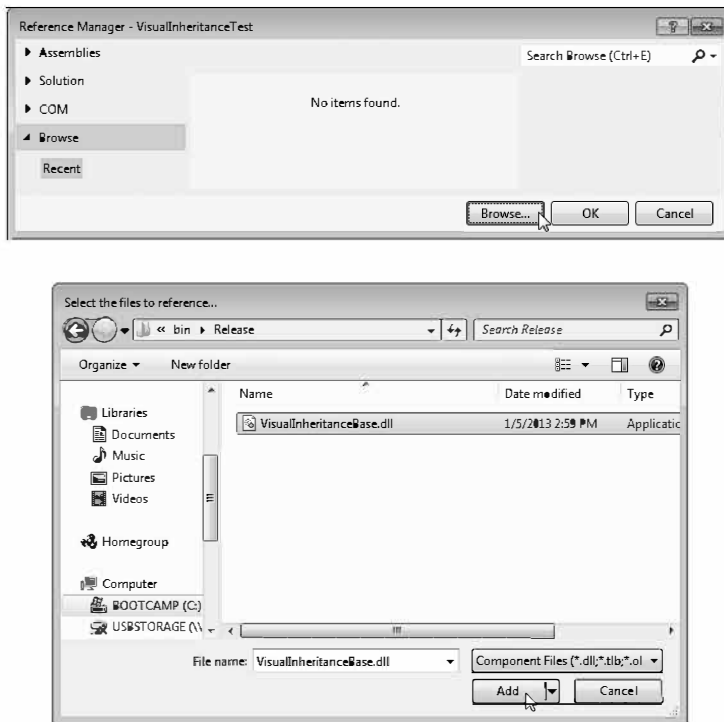
Чтобы другие формы могли наследовать от `VisualInheritanceForm`, необходимо упаковать `VisualInheritanceForm` в библиотеку классов и откомпилировать ее в файл `.dll`. В таких файлах, называемых *библиотеками динамической компоновки* (dynamically linked library), упаковываются классы, на которые можно ссылаться из других приложений. Щелкните правой кнопкой мыши на имени проекта в окне `Solution Explorer` и выберите команду `Properties`, перейдите на вкладку `Application`. В раскрывающемся списке `Output` выберите вместо `Windows Application` вариант `Class Library`. В результате построения проекта будет создан файл `.dll`. Проект также можно настроить как библиотеку классов при создании — для этого выберите шаблон `Class Library` в диалоговом окне `New Project`. [*Примечание:* библиотека классов не может выполняться как автономное приложение. Экранники, представленные на ил. 15.45, были сделаны до изменения типа проекта.]

### Шаг 4. Включение ссылки на библиотеку классов

После того как класс будет откомпилирован и сохранен в файле библиотеки классов, разработчик может использовать библиотеку в любом приложении; для этого необходимо лишь сообщить среде разработки местонахождение файла библиотеки. Чтобы организовать визуальное наследование от `VisualInheritanceBaseForm`, сначала

создайте новое приложение Windows. Щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer и выберите в открывшемся контекстном меню команду Add Reference. На экране появляется диалоговое окно со списком библиотеки классов .NET Framework. Некоторые библиотеки классов (например, библиотека с пространством имен System) используются настолько часто, что IDE включает их в приложения автоматически. С библиотеками из списка этого не происходит.

В диалоговом окне Reference Manager щелкните на ссылке Browse, затем на кнопке Browse.... При построении библиотеки классов Visual C# помещает файл .dll в папку bin\Debug или bin\Release проекта в зависимости от того, какой вариант выбран в раскрывающемся списке Solution Configurations на панели инструментов IDE — Debug или Release. На вкладке Browse перейдите к каталогу с файлом библиотеки классов, созданному на шаге 3 (ил. 15.46). Выберите файл .dll и щелкните на кнопке OK.



**Ил. 15.46.** Поиск файла DLL в диалоговом окне Reference Manager

### Шаг 5. Создание класса, производного от базовой формы

Откройте файл, определяющий графический интерфейс нового приложения, и измените строку с определением класса: в ней должно быть указано, что форма приложения наследует от класса VisualInheritanceBaseForm. Строка с объявлением класса должна выглядеть так:

```
public partial class VisualInheritanceTestForm :
    VisualInheritanceBaseForm
```

Если пространство имен `VisualInheritanceBase` не указано в директиве `using`, необходимо использовать полное имя класса `VisualInheritanceBase.VisualInheritanceBaseForm`. В режиме конструктора на форме нового приложения должны отображаться элементы управления, унаследованные от базовой формы (ил. 15.47). Теперь на форму можно добавить новые компоненты.



**Ил. 15.47.** Форма для демонстрации визуального наследования

### Класс `VisualInheritanceTestForm`

Класс `VisualInheritanceTestForm` (ил. 15.48) является производным от `VisualInheritanceBaseForm`. Выходные данные демонстрируют функциональность программы. Компоненты, их размещение и функциональность базового класса `VisualInheritanceBaseForm` (см. ил. 15.45) наследуются классом `VisualInheritanceTestForm`. К ним добавляется кнопка с текстом `About this Program`. При нажатии кнопки вызывается метод `aboutButton_Click` (строки 19–25). Этот метод выводит другое окно сообщения с другим текстом (строки 21–24).

```

1  // Ил. 15.48: VisualInheritanceTestForm.cs
2  // Создание производных форм с использованием визуального наследования.
3  using System;
4  using System.Windows.Forms;
5
6  namespace VisualInheritanceTest
7  {
8      // Производная форма, созданная посредством визуального наследования
9      public partial class VisualInheritanceTestForm :
10         VisualInheritanceBase.VisualInheritanceBaseForm
11     {
12         // Конструктор
13         public VisualInheritanceTestForm()
14         {
15             InitializeComponent();
16         } // Конец конструктора
17
18         // При щелчке на кнопке открывается окно сообщения
19         private void aboutButton_Click(object sender, EventArgs e)
20         {
21             MessageBox.Show(
22                 "This program was created by Deitel & Associates.",

```

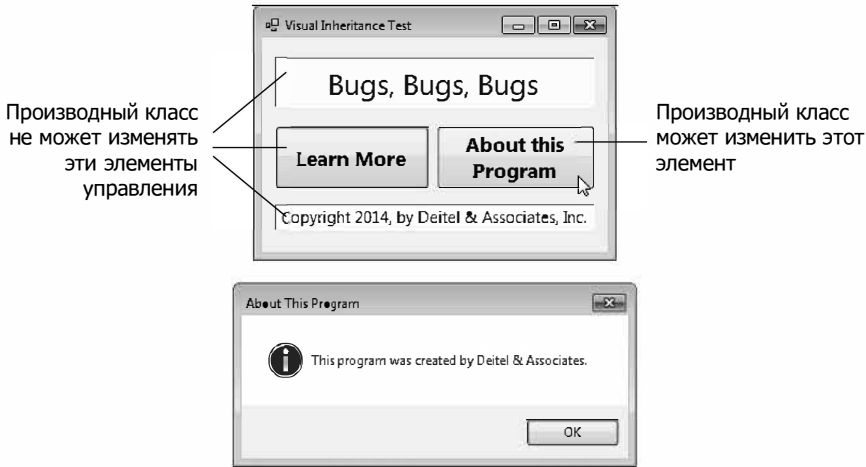
**Ил. 15.48.** Класс `VisualInheritanceTestForm`, производный от класса `VisualInheritanceBaseForm`, содержит дополнительную кнопку (продолжение ↗)



```

23         "About This Program", MessageBoxButtons.OK,
24         MessageBoxIcon.Information );
25     } // Конец метода aboutButton_Click
26 } // Конец класса VisualInheritanceTestForm
27 } // Конец пространства имен VisualInheritanceTest

```



**Ил. 15.48.** Класс `VisualInheritanceTestForm`, производный от класса `VisualInheritanceBaseForm`, содержит дополнительную кнопку (окончание)

Когда пользователь щелкает на кнопке `Learn More`, событие обрабатывается обработчиком базового класса `learnMoreButton_Click`. Так как `VisualInheritanceBaseForm` использует модификатор доступа `private` для объявления своих элементов управления, `VisualInheritanceTestForm` не может изменять унаследованные элементы на визуальном или программном уровне. IDE отображает в левом верхнем углу визуального унаследованного элемента управления значок, который сообщает, что элемент управления унаследован и не может изменяться.

## 15.14. Пользовательские элементы управления

.NET Framework позволяет создавать пользовательские элементы управления. Они размещаются на панели элементов и добавляются на формы, контейнеры `Panel` и `GroupBox` так же, как мы добавляем кнопки, надписи и другие заранее определенные элементы управления. Самый простой способ создания пользовательских элементов управления — объявление класса, производного от существующего элемента управления (например, `Label`). Этот способ хорошо подходит для расширения функциональности существующих элементов вместо замены их новым элементом управления, предоставляющим нужные функции. Например, вы можете создать новую разновидность `Label`, которая ведет себя как обычная надпись, но выглядит иначе. Для этого разработчик определяет класс, производный от `Label`, и переопределяет метод `OnPaint`.

## Метод OnPaint

Все элементы управления поддерживают метод `OnPaint`, вызываемый системой при перерисовке компонента (например, при изменении его размеров). Метод получает объект `PaintEventArgs` с графической информацией — свойство `Graphics` содержит графический объект, используемый для перерисовки, и свойство `ClipRectangle` определяет ограничивающий прямоугольник элемента управления. Когда система инициирует событие `Paint` для перерисовки элемента управления, последний перехватывает событие и вызывает свой метод `OnPaint`. Перед выполнением кода пользовательской прорисовки переопределенная реализация `OnPaint` должна вызывать метод `OnPaint` базового класса. Тем самым обеспечивается гарантированное выполнение исходного кода прорисовки наряду с кодом, определенным вами в классе пользовательского элемента управления.

## Создание новых элементов управления

Для создания новых элементов управления на базе существующих элементов используется класс `UserControl`. Например, программист может создать пользовательский элемент управления из элементов `Button`, `Label` и `TextBox`, связанных общей функциональностью (скажем, кнопка задает текст надписи по данным, содержащимся в текстовом поле). Объект `UserControl` выполняет функции контейнера для добавленных в него элементов управления. `UserControl` содержит существующие элементы, но не определяет способ их прорисовки. Для управления внешним видом каждого составляющего элемента можно организовать обработку события `Paint` каждого элемента управления или переопределить `OnPaint`. И обработчик события `Paint`, и метод `OnPaint` получают аргумент `PaintEventArgs`, который может использоваться для рисования (линии, прямоугольники и т. д.) на составляющих элементах управления.

Также существует другой путь: программа может создать элемент управления «с нуля» наследованием от класса `Control`. Класс не определяет никакого конкретного поведения; этим должен заниматься программист. Вместо этого класс `Control` обеспечивает функциональность, свойственную всем элементам управления (например, события и маркеры изменения размеров). Метод `OnPaint` должен содержать вызов метода `OnPaint` базового класса, который вызывает обработчики событий `Paint`. Вы добавляете код, который осуществляет пользовательскую прорисовку в переопределенном методе `OnPaint`. Этот способ обеспечивает наибольшую гибкость, но требует самого значительного планирования. Краткая сводка всех трех способов приведена на ил. 15.49.

Способы создания пользовательских элементов управления и свойства <code>PaintEventArgs</code>	Описание
Способы создания	
Наследование от элементов управления Windows Forms	Используется для добавления функциональности к существующим элементам управления. Если вы переопределяете метод <code>OnPaint</code> , вызовите метод <code>OnPaint</code> базового класса. Возможно только дополнение внешнего вида исходного элемента управления, а не его радикальная переработка

**Ил. 15.49.** Создание пользовательских элементов управления (продолжение ➤)

Способы создания пользовательских элементов управления и свойства PaintEventArgs	Описание
Создание UserControl	Объект UserControl создается из нескольких существующих элементов управления (с объединением их функциональности). Код прорисовки размещается в обработчике события Paint или в переопределенном методе OnPaint
Наследование от класса Control	Определение нового элемента управления «с нуля». Разработчик переопределяет метод OnPaint, затем вызывает метод OnPaint базового класса и включает методы прорисовки. Этот способ позволяет полностью определить внешний вид и функциональность элемента управления
Свойства PaintEventArgs	
Graphics	Объект, используемый для рисования на поверхности элемента управления
ClipRectangle	Задаёт прямоугольник, определяющий границы элемента управления

**Ил. 15.49.** Создание пользовательских элементов управления (окончание)

### Элемент управления Clock

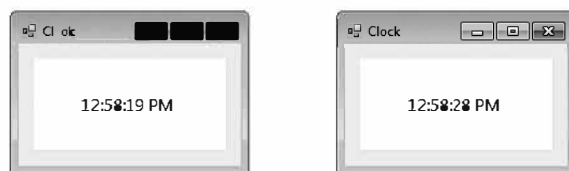
На ил. 15.50 представлена реализация элемента управления цифровых часов. Этот объект UserControl состоит из надписи (Label) и таймера (Timer) — каждый раз, когда Timer инициирует событие (один раз в секунду в нашем примере), содержимое Label обновляется текущим временем.

```

1 // Ил. 15.50: ClockUserControl.cs
2 // Пользовательский элемент управления с таймером и надписью.
3 using System;
4 using System.Windows.Forms;
5
6 namespace ClockExample
7 {
8     // UserControl для вывода текущего времени
9     public partial class ClockUserControl : UserControl
10    {
11        // Конструктор
12        public ClockUserControl()
13        {
14            InitializeComponent();
15        } // Конец конструктора
16
17        // Обновление Label с каждым сигналом таймера
18        private void clockTimer_Tick(object sender, EventArgs e)
19        {
20            // Получение текущего времени (Now) и преобразование в строку
21            displayLabel.Text = DateTime.Now.ToString();
22        } // Конец метода clockTimer_Tick
23    } // Конец класса ClockUserControl
24 } // Конец пространства имен ClockExample

```

**Ил. 15.50.** Реализация часов на базе UserControl (продолжение ➞)



**Ил. 15.50.** Реализация часов на базе UserControl (окончание)

## Таймеры

Таймеры (пространство имен `System.Windows.Forms`) представляют собой невизуальные компоненты, генерирующие события `Tick` с заданным интервалом. Этот интервал задается свойством `Interval` объекта `Timer`, определяющим количество миллисекунд (тысячных долей секунды) между событиями. По умолчанию таймеры не работают и не генерируют события.

## Добавление пользовательского элемента управления

В этом приложении используется пользовательский элемент управления (`ClockUserControl`) и форма, на которой он размещается. Создайте Windows-приложение, затем создайте класс, производный от `UserControl`, командой **Project ▸ Add User Control**. На экране появляется диалоговое окно для выбора типа добавляемого компонента — в нем уже выбран тип пользовательских элементов управления. Введите имя файла (и класса) `ClockUserControl`. Наш пустой элемент управления `ClockUserControl` отображается в виде серого прямоугольника.

## Проектирование пользовательского элемента управления

Вы работаете с новым элементом управления почти так же, как с формами `Windows`, — на него тоже можно добавлять элементы управления с панели элементов и задавать свойства в окне свойств. Однако вместо создания приложения вы просто создаете новый элемент управления, состоящий из других элементов. Добавьте на `UserControl` надпись (`Label` с именем `displayLabel`) и таймер (`Timer` с именем `clockTimer`). Установите интервал таймера равным 1000 миллисекундам и запрограммируйте вывод текста `displayLabel` в каждом событии `Tick` (строки 18–22). Чтобы таймер генерировал события, его свойству `Enabled` необходимо задать значение `true` в окне свойств.

Структура `DateTime` (пространство имен `System`) содержит свойство `Now`, которое возвращает текущее время. Метод `ToLongTimeString` преобразует `Now` в строку с часами, минутами и секундами (и возможно, индикатором АМ/ПМ в зависимости от локального контекста). Эта строка используется для назначения времени в `displayLabel` (строка 21).

Созданный элемент управления появляется на панели элементов в категории *ИмяПроекта* `Components`, где *ИмяПроекта* — имя вашего проекта. Возможно, для

появления нового значка на панели элементов вам придется переключиться на форму вашего приложения. Чтобы использовать элемент управления, просто перетащите его на форму и запустите приложение Windows. Мы назначили объекту `ClockUserControl` белый фон, чтобы он выделялся на форме. На ил. 15.50 показан результат работы приложения с элементом управления `ClockUserControl`. В приложении обработчиков событий нет, поэтому мы приводим только код `ClockUserControl`.

### Распространение пользовательских элементов управления

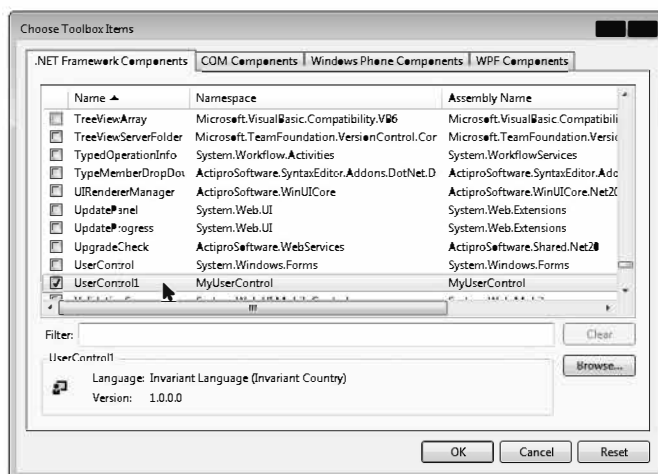
Visual Studio позволяет передавать пользовательские элементы управления другим разработчикам. Чтобы создать реализацию `UserControl`, которая может экспортироваться в другие решения, выполните следующие действия:

1. Создайте новый проект **Class Library**.
2. Удалите файл `Class1.cs`, автоматически включенный в приложение.
3. Щелкните правой кнопкой мыши на проекте в окне **Solution Explorer** и выберите команду **Add ► User Control**. В открывшемся диалоговом окне введите имя файла пользовательского элемента управления и щелкните на кнопке **Add**.
4. В проекте добавьте в `UserControl` элементы управления и функциональность (ил. 15.51).



**Ил. 15.51.** Создание пользовательского элемента управления

5. Постройте проект. Visual Studio создает для `UserControl` файл `.dll` в выходном каталоге (`bin/Release` или `bin/Release`). Этот файл не является исполняемым; библиотеки классов используются для определения классов, используемых в других приложениях.
6. Создайте Windows-приложение.
7. В созданном приложении щелкните правой кнопкой мыши на панели элементов и выберите команду **Choose Items**. В открывшемся диалоговом окне **Choose Toolbox Items** щелкните на кнопке **Browse**. Найдите файл `.dll` в библиотеке классов, созданной на этапах 1–5. Выбранный элемент управления появляется в диалоговом окне **Choose Toolbox Items** (ил. 15.52). Если он еще не выделен, установите флажок. Щелкните на кнопке **OK**, чтобы добавить элемент управления на панель элементов. Теперь его можно добавлять на форму, как любой другой элемент управления.



**Ил. 15.52.** Добавление пользовательского элемента управления на панель элементов

## 15.15. Итоги

Многие современные коммерческие приложения обладают удобным графическим интерфейсом, поэтому умение создавать сложные графические интерфейсы стало важным аспектом работы любого программиста. В среде разработки Visual Studio графические интерфейсы строятся легко и быстро. В главах 14 и 15 были представлены основные приемы программирования графических интерфейсов Windows Forms. В главе 15 вы научились создавать меню, предоставляющие пользователю простой доступ к функциональности приложения. Мы рассмотрели элементы управления `DateTimePicker` и `MonthCalendar`, предназначенные для ввода даты и времени, а также элементы управления `LinkLabel` для создания ссылок на приложения и веб-страницы. Далее были представлены элементы управления для отображения списковой информации — `ListBox`, `CheckedListBox` и `ListView`. Мы использовали элемент управления `ComboBox` для создания раскрывающихся списков, а элемент управления `TreeView` — для отображения данных в иерархической форме. Затем были представлены сложные графические интерфейсы со вкладками и с поддержкой MDI. Глава завершается демонстрацией визуального наследования и создания пользовательских элементов управления. В главе 16 мы займемся обработкой строковых и символьных данных.

# 16 Строки и символы

## 16.1. Введение

В этой главе представлены средства .NET Framework Class Library по работе со строками и с символами. Эти средства могут применяться в текстовых редакторах, системах форматирования текста, системах электронной верстки и других программах, работающих с текстом. В предыдущих главах были описаны простейшие возможности обработки строк, а сейчас мы займемся подробным рассмотрением средств работы с текстом класса `string` и типа `char` из пространства имен `System`, а также класса `StringBuilder` из пространства имен `System.Text`.

Глава начинается с общих сведений о символах и строках, символьных константах и строковых литералах, с примерами использования многочисленных конструкторов и методов класса `string`. Эти примеры показывают, как определить длину строки, скопировать строку, обратиться к отдельным символам, выполнять поиск в строках, извлекать подстроки, сравнивать строки, выполнять конкатенацию, заменять символы в строках и преобразовывать строки к верхнему или нижнему регистру.

Затем будет представлен класс `StringBuilder`, предназначенный для динамического построения строк. Мы рассмотрим возможности класса `StringBuilder` по определению и заданию размера объекта `StringBuilder`, а также присоединению, вставке, удалению и замене символов в объекте `StringBuilder`. Также будут представлены методы проверки символов структуры `Char`, при помощи которых программа может определить, является ли символ цифрой, буквой, символом нижнего регистра, символом верхнего регистра, знаком препинания или символом, отличным от знака препинания. Эти методы удобны для проверки отдельных символов в пользовательском вводе. Кроме того, тип `Char` предоставляет методы для преобразования символа к верхнему или нижнему регистру.

## 16.2. Основы работы с символами и строками

*Символы* являются основными структурными элементами исходного кода C#. Любая программа состоит из символов, последовательность которых интерпретируется

компилятором как инструкции по выполнению некоторой операции. Программа также может содержать *символьные константы* — символы, представленные в виде целочисленного значения (кода символа). Например, целочисленное значение 122 соответствует символьной константе «z», а значение 10 — служебному символу новой строки «\n». Символьные константы задаются в соответствии с кодировкой Юникод — международным набором символов, который содержит намного больше знаков и букв, чем кодировка ASCII.

*Строка* (string) представляет собой последовательность символов, которая рассматривается как единое целое. Такими символами могут быть прописные и строчные буквы, цифры, различные специальные знаки: +, -, \*, /, \$ и т. д. Строка представляется объектом класса string из пространства имен System<sup>1</sup>. Строковые литералы, также называемые строковыми константами, записываются в виде последовательности символов, заключенной в кавычки:

```
"John Q. Doe"  
"9999 Main Street"  
"Waltham, Massachusetts"  
"(201) 555-1212"
```

В объявлении ссылке string может присваиваться строковый литерал. Объявление `string color = "blue";`

инициализирует переменную color типа string ссылкой на объект строкового литерала "blue".



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 16.1

Если объект строкового литерала несколько раз используется в приложении, то во всех местах программы, где этот литерал используется, будут содержаться ссылки на одну копию литерала. Такое совместное использование одного объекта (с целью экономии памяти) возможно, потому что объекты строковых литералов неявно являются константами.

### Буквальные строки

В некоторых строках (например, в именах файлов) неоднократно встречаются символы \ (обратная косая черта). Чтобы избежать включения в строку лишних символов \, можно запретить интерпретацию служебных последовательностей и интерпретировать все символы в строке буквально. Для этого перед литералом ставится символ @, создающий так называемую *буквальную строку* (verbatim string). Символы \ в кавычках, следующие за @, не считаются признаками служебных последовательностей. Буквальные строки часто упрощают программирование и чтение кода. Возьмем строку "C:\MyFolder\MySubFolder\MyFile.txt" со следующей командой присваивания:

```
string file = "C:\\MyFolder\\MySubFolder\\MyFile.txt";
```

<sup>1</sup> В C# ключевое слово string является синонимом для класса String.



С использованием синтаксиса буквальных строк команда присваивания может выглядеть так:

```
string file = @"C:\MyFolder\MySubFolder\MyFile.txt";
```

Такое решение также позволяет распространять строковые литералы на несколько строк с сохранением всех новых строк, пробелов и табуляций.

## 16.3. Конструкторы string

Класс `string` предоставляет восемь конструкторов для разных способов инициализации строк. Три из них продемонстрированы на ил. 16.1.

```
1 // Ил. 16.1: StringConstructor.cs
2 // Конструкторы класса string.
3 using System;
4
5 class StringConstructor
6 {
7     public static void Main( string[] args )
8     {
9         // Инициализация string
10        char[] characterArray =
11        { 'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y' };
12        string originalString = "Welcome to C# programming!";
13        string string1 = originalString;
14        string string2 = newstring( characterArray );
15        string string3 = newstring( characterArray, 6, 3 );
16        string string4 = newstring( 'C', 5 );
17
18        Console.WriteLine( "string1 = " + "\"" + string1 + "\"\n" +
19        "string2 = " + "\"" + string2 + "\"\n" +
20        "string3 = " + "\"" + string3 + "\"\n" +
21        "string4 = " + "\"" + string4 + "\"\n" );
22    } // Конец Main
23 } // Конец класса StringConstructor
```

```
string1 = "Welcome to C# programming!"
string2 = "birth day"
string3 = "day"
string4 = "CCCCC"
```

**Ил. 16.1.** Конструкторы string

В строках 10–11 выделяется память для массива `characterArray`, содержащего девять символов. В строках 12–16 объявляются строковые переменные `originalString`, `string1`, `string2`, `string3` и `string4`. В строке 12 строковый литерал `"Welcome to C# programming!"` присваивается переменной `originalString`. В строке 13 в переменную `string1` заносится ссылка на тот же литерал.

В строке 14 переменной `string2` присваивается новая строка; при этом используется конструктор `string`, в аргументе которого передается массив символов. Созданная строка содержит копию символов массива.

Строка 15 присваивает `string3` новую строку с использованием конструктора, получающего массив `char` и два аргумента `int`. Второй аргумент задает начальный индекс (смещение), с которого начинается копирование символов массива. Третий аргумент задает количество символов, копируемых от заданной позиции. Новая строка содержит копию символов массива. Если при заданном смещении и длине происходит выход за границу массива, выдается исключение `ArgumentOutOfRangeException`.

В строке 16 переменной `string4` присваивается новая строка, созданная конструктором, в аргументах которого передается символ и значение `int`, определяющее количество повторений этого символа в строке.



### АРХИТЕКТУРНОЕ РЕШЕНИЕ 16.1

В большинстве случаев копировать существующую строку не обязательно. Все строки являются неизменяемыми — их символьное содержимое не может изменяться после создания. Кроме того, если на объект `string` (или вообще на любой другой объект) существует несколько ссылок, этот объект не может быть уничтожен уборщиком мусора.

## 16.4. Индексатор `string`, свойство `Length` и метод `CopyTo`

В приложении на ил. 16.2 продемонстрировано использование индексатора `string`, упрощающего выборку любого символа в строке, и свойства `Length`, возвращающего длину строки. Метод `CopyTo` копирует заданное количество символов из строки в массив символов.

```

1 // Ил. 16.2: StringMethods.cs
2 // Использование индексатора, свойства Length и метода CopyTo
3 // класса string.
4 using System;
5
6 class StringMethods
7 {
8     public static void Main( string[] args )
9     {
10         string string1 = "hello there";
11         char[] characterArray = new char[ 5 ];
12
13         // Вывод значения string1
14         Console.WriteLine( "string1: \"" + string1 + "\"" );
15
16         // Проверка свойства Length
17         Console.WriteLine( "Length of string1: " + string1.Length );
18
19         // Перебор символов string1 и вывод в обратном порядке
20         Console.Write( "The string reversed is: " );

```

**Ил. 16.2.** Индексатор `string`, свойство `Length` и метод `CopyTo` (продолжение ↗)

```

21
22     for ( int i = string1.Length - 1; i >= 0; --i )
23         Console.Write( string1[ i ] );
24
25     // Копирование символов из string1 в characterArray
26     string1.CopyTo( 0, characterArray, 0, characterArray.Length );
27     Console.Write( "\nThe character array is: " );
28
29     for ( int i = 0; i < characterArray.Length; ++i )
30         Console.Write( characterArray[ i ] );
31
32     Console.WriteLine( "\n" );
33 } // конец Main
34 } // Конец класса StringMethods

```

```

string1: "hello there"
Length of string1: 11
The string reversed is: ereht olleh
The character array is: hello

```

**Ил. 16.2.** Индексатор string, свойство Length и метод CopyTo (окончание)

Приложение определяет длину строки, выводит ее символы в обратном порядке и копирует последовательность символов из строки в символьный массив. В строке 17 свойство Length используется для определения количества символов в string1. Строки, как и массивы, всегда знают свой размер.

В строках 22–23 символы string1 выводятся в обратном порядке с использованием индексатора. Индексатор интерпретирует объект string как массив символов и возвращает каждый символ в некоторой позиции строки. Он получает целочисленный аргумент и возвращает символ, находящийся в заданной позиции. Как и в случае с массивами, считается, что первый элемент строки находится в позиции 0.

**ТИПИЧНАЯ ОШИБКА 16.1**

При попытке обратиться к символу за границей строки выдается исключение `IndexOutOfRangeException`.

В строке 26 метод CopyTo копирует символы string1 в символьный массив characterArray. Первый аргумент метода CopyTo содержит индекс, с которого метод начинает копирование символов в строку. Второй аргумент задает массив, в который копируются символы. Третий аргумент содержит индекс начальной позиции, с которой метод начинает размещать скопированные символы в массив. В последнем аргументе передается количество символов, копируемых из строки. В строках 29–30 содержимое массива выводится по одному символу.

## 16.5. Сравнение строк

В следующих двух примерах представлены различные способы сравнения строк. Чтобы понять, как одна строка может быть «больше» или «меньше» другой,


представьте процесс упорядочения фамилий по алфавиту. Разумеется, фамилия "Jones" будет предшествовать фамилии "Smith", потому что первая буква "Jones" предшествует первой букве "Smith". Алфавит — не просто набор букв, а упорядоченный список символов, в котором каждая буква находится в определенной позиции. Компьютеры могут упорядочивать символы по алфавиту, потому что во внутреннем представлении символов используются числовые коды, которые тоже упорядочены по алфавиту, так что код 'a', например, меньше кода 'b'.

### Сравнение строк методами Equals, CompareTo и оператором ==

Класс `string` предоставляет несколько способов сравнения строк. В приложении на ил. 16.3 продемонстрировано использование метода `Equals`, метода `CompareTo` и оператора проверки равенства (`==`).

Условие в строке 21 использует метод `Equals` для проверки равенства `string1` и строкового литерала `"hello"`. Метод `Equals` (унаследованный от `object` и переопределенный в `string`) проверяет две строки на идентичность содержимого. Если содержимое совпадает, метод возвращает `true`; в противном случае возвращается `false`. В данном случае результат равен `true`, поскольку `string1` ссылается на объект строкового литерала `"hello"`. Метод `Equals` использует правила сортировки, определяемые текущим культурным контекстом вашей системы. Сравнение `"hello"` с `"HELLO"` вернет `false`, потому что буквы нижнего регистра отличаются от соответствующих букв верхнего регистра.

```
1 // Ил. 16.3: StringComparison.cs
2 // Сравнение строк
3 using System;
4
5 class StringComparison
6 {
7     public static void Main( string[] args )
8     {
9         string string1 = "hello";
10        string string2 = "good bye";
11        string string3 = "Happy Birthday";
12        string string4 = "happy birthday";
13
14        // Вывод значений четырех строк
15        Console.WriteLine( "string1 = \"" + string1 + "\"" +
16            "\"\nstring2 = \"" + string2 + "\"" +
17            "\"\nstring3 = \"" + string3 + "\"" +
18            "\"\nstring4 = \"" + string4 + "\"\n" );
19
20        // Проверка равенства с методом Equals
21        if ( string1.Equals( "hello" ) )
22            Console.WriteLine( "string1 equals \"hello\"" );
23        else
24            Console.WriteLine( "string1 does not equal \"hello\"" );
25
26        // Проверка равенства с оператором ==
27        if ( string1 == "hello" )
```

**Ил. 16.3.** Примеры сравнения объектов `string` (продолжение )

```

28     Console.WriteLine( "string1 equals \"hello\"" );
29     else
30         Console.WriteLine( "string1 does not equal \"hello\"" );
31
32     // Проверка равенства с учетом регистра case
33     if ( string.Equals( string3, string4 ) ) // Статический метод
34         Console.WriteLine( "string3 equals string4" );
35     else
36         Console.WriteLine( "string3 does not equal string4" );
37
38     // Демонстрация CompareTo
39     Console.WriteLine( "\nstring1.CompareTo( string2 ) is " +
40         string1.CompareTo( string2 ) + "\n" +
41         "string2.CompareTo( string1 ) is " +
42         string2.CompareTo( string1 ) + "\n" +
43         "string1.CompareTo( string1 ) is " +
44         string1.CompareTo( string1 ) + "\n" +
45         "string3.CompareTo( string4 ) is " +
46         string3.CompareTo( string4 ) + "\n" +
47         "string4.CompareTo( string3 ) is " +
48         string4.CompareTo( string3 ) + "\n\n" );
49 } // Конец Main
50 } // Конец класса StringComparison

```

```

string1 = "hello"
string2 = "good bye"
string3 = "Happy Birthday"
string4 = "happy birthday"

string1 equals "hello"
string1 equals "hello"
string3 does not equal string4

string1.CompareTo( string2 ) is 1
string2.CompareTo( string1 ) is -1
string1.CompareTo( string1 ) is 0
string3.CompareTo( string4 ) is 1
string4.CompareTo( string3 ) is -1

```

**Ил. 16.3.** Примеры сравнения объектов string (окончание)

Условие в строке 27 использует перегруженный оператор проверки равенства (==) для сравнения string1 с литералом "hello". В C# оператор проверки равенства сравнивает содержимое двух строк. Таким образом, условие if будет истинным, потому что значения string1 и "hello" равны.

В строке 33 проверяется равенство string3 и string4; результат демонстрирует, что сравнения действительно производятся с учетом регистра. Здесь статический метод Equals используется для сравнения двух строк. Значение "HappyBirthday" не равно "happy birthday", поэтому условие if ложно и программа выводит сообщение "string3 does not equal string4" (строка 36).

В строках 40–48 метод CompareTo класса string используется для сравнения строк. Метод CompareTo возвращает 0, если строки равны; отрицательное значение, если

строка, для которой вызывается `CompareTo`, меньше строки, переданной в аргументе; или положительное значение, если строка, для которой вызывается `CompareTo`, больше аргумента.

Обратите внимание: `CompareTo` считает, что `string3` больше `string4`. Между этими двумя строками есть только одно различие: `string3` содержит две буквы верхнего регистра в позициях, в которых `string4` содержит буквы нижнего регистра, — код буквы в верхнем регистре меньше кода соответствующей буквы в нижнем регистре (например, букве 'A' соответствует код 65, а букве 'a' — код 97).

### Проверка начала или конца строки

На ил. 16.4 показано, как проверить, что экземпляр `string` начинается или заканчивается заданной подстрокой. Для такой проверки используется метод `StartsWith` или `EndsWith` соответственно. Метод `Main` класса `StringStartEnd` определяет массив с именем `strings`, содержащий строки "started", "starting", "ended" и "ending". Оставшаяся часть кода `Main` проверяет наличие конкретного набора символов в начале или конце элементов массива.

В строке 13 используется метод `StartsWith`, получающий строковый аргумент. Условие `if` проверяет, начинается ли строка с индексом `i` с символов "st". Если результат проверки положителен, то метод возвращает `true` и значение `strings[i]` выводится с соответствующим сообщением.

В строке 21 метод `EndsWith` проверяет, заканчивается ли строка с индексом `i` символами "ed". Как и в предыдущем случае, при положительном результате выводится значение `strings[i]` с сообщением.

```
1  // Ил. 16.4: StringStartEnd.cs
2  // Использование методов StartsWith и EndsWith.
3  using System;
4
5  class StringStartEnd
6  {
7      public static void Main( string[] args )
8      {
9          string[] strings = { "started", "starting", "ended", "ending" };
10
11          // Проверяем, начинается ли каждая строка в массиве с "st"
12          for ( int i = 0; i < strings.Length; i++ )
13              if ( strings[ i ].StartsWith( "st" ) )
14                  Console.WriteLine( "\"" + strings[ i ] + "\"" +
15                      " starts with \"st\"" );
16
17          Console.WriteLine();
18
19          // Проверяем, заканчивается ли каждая строка в массиве на "ed"
20          for ( int i = 0; i < strings.Length; i++ )
21              if ( strings[ i ].EndsWith( "ed" ) )
22                  Console.WriteLine( "\"" + strings[ i ] + "\"" +
23                      " ends with \"ed\"" );
```

**Ил. 16.4.** Методы `StartsWith` и `EndsWith` (продолжение ➤)

```

24
25     Console.WriteLine();
26 } // Конец Main
27 } // Конец класса StringStartEnd

```

```

"started" starts with "st"
"starting" starts with "st"

"started" ends with "ed"
"ended" ends with "ed"

```

**Ил. 16.4.** Методы `StartsWith` и `EndsWith` (окончание)

## 16.6. Поиск символов и подстрок

Задача поиска символа или набора символов в строке встречается во многих приложениях. Например, в любом текстовом редакторе поддерживается поиск в документе. В приложении на ил. 16.5 продемонстрированы некоторые из многочисленных версий методов `IndexOf`, `IndexOfAny`, `LastIndexOf` и `LastIndexOfAny`, предназначенных для поиска заданного символа или подстроки. Все варианты поиска в этом примере выполняются в строке `letters` (инициализированной значением `"abcdefghijklmabcdefghijklm"`) в методе `Main` класса `StringIndexMethods`.

В строках 14, 16 и 18 метод `IndexOf` используется для поиска первого вхождения символа или подстроки в объекте `string`. Если поиск оказывается успешным, `IndexOf` возвращает индекс заданного символа в строке; в противном случае `IndexOf` возвращает `-1`. Выражение в строке 16 использует версию метода `IndexOf` с двумя аргументами — искомым символом и начальным индексом, с которого должен начаться поиск. Метод не анализирует никакие символы, предшествующие этому индексу (в данном случае 1). Выражение в строке 18 использует другую версию метода `IndexOf`, которая получает три аргумента — искомый символ, индекс начала поиска и размер области поиска.

```

1  // Ил. 16.5: StringIndexMethods.cs
2  // Методы поиска символов и подстрок.
3  using System;
4
5  class StringIndexMethods
6  {
7      public static void Main( string[] args )
8      {
9          string letters = "abcdefghijklmabcdefghijklm";
10         char[] searchLetters = { 'c', 'a', '$' };
11
12         // Проверка IndexOf для поиска символа в строке
13         Console.WriteLine( "First 'c' is located at index " +
14             letters.IndexOf( 'c' ) );
15         Console.WriteLine( "First 'a' starting at 1 is located at index " +
16             letters.IndexOf( 'a', 1 ) );

```

**Ил. 16.5.** Поиск символов и подстрок в объектах `string` (продолжение ↗)

```

17 Console.WriteLine( "First '$' in the 5 positions starting at 3 " +
18     "is located at index " + letters.IndexOf( '$', 3, 5 ) );
19
20 // Проверка LastIndexOf для поиска символа в строке
21 Console.WriteLine( "\nLast 'c' is located at index " +
22     letters.LastIndexOf( 'c' ) );
23 Console.WriteLine( "Last 'a' up to position 25 is located at " +
24     "index " + letters.LastIndexOf( 'a', 25 ) );
25 Console.WriteLine( "Last '$' in the 5 positions ending at 15 " +
26     "is located at index " + letters.LastIndexOf( '$', 15, 5 ) );
27
28 // Проверка IndexOf для поиска подстроки
29 Console.WriteLine( "\nFirst \"def\" is located at index " +
30     letters.IndexOf( "def" ) );
31 Console.WriteLine( "First \"def\" starting at 7 is located at " +
32     "index " + letters.IndexOf( "def", 7 ) );
33 Console.WriteLine( "First \"hello\" in the 15 positions " +
34     "starting at 5 is located at index " +
35     letters.IndexOf( "hello", 5, 15 ) );
36
37 // Проверка LastIndexOf для поиска подстроки
38 Console.WriteLine( "\nLast \"def\" is located at index " +
39     letters.LastIndexOf( "def" ) );
40 Console.WriteLine( "Last \"def\" up to position 25 is located " +
41     "at index " + letters.LastIndexOf( "def", 25 ) );
42 Console.WriteLine( "Last \"hello\" in the 15 positions " +
43     "ending at 20 is located at index " +
44     letters.LastIndexOf( "hello", 20, 15 ) );
45
46 // Проверка IndexOfAny для поиска первого вхождения символов в массиве
47 Console.WriteLine( "\nFirst 'c', 'a' or '$' is " +
48     "located at index " + letters.IndexOfAny( searchLetters ) );
49 Console.WriteLine( "First 'c', 'a' or '$' starting at 7 is " +
50     "located at index " + letters.IndexOfAny( searchLetters, 7 ) );
51 Console.WriteLine( "First 'c', 'a' or '$' in the 5 positions " +
52     "starting at 7 is located at index " +
53     letters.IndexOfAny( searchLetters, 7, 5 ) );
54
55 // Проверка LastIndexOfAny для поиска последнего вхождения
56 // символов в массиве.
57 Console.WriteLine( "\nLast 'c', 'a' or '$' is " +
58     "located at index " + letters.LastIndexOfAny( searchLetters ) );
59 Console.WriteLine( "Last 'c', 'a' or '$' up to position 1 is " +
60     "located at index " +
61     letters.LastIndexOfAny( searchLetters, 1 ) );
62 Console.WriteLine( "Last 'c', 'a' or '$' in the 5 positions " +
63     "ending at 25 is located at index " +
64     letters.LastIndexOfAny( searchLetters, 25, 5 ) );
65 } // Конец Main
66 } // Конец класса StringIndexMethods

```

```

First 'c' is located at index 2
First 'a' starting at 1 is located at index 13
First '$' in the 5 positions starting at 3 is located at index -1

```

**Ил. 16.5.** Поиск символов и подстрок в объектах string (продолжение ☞)



```

Last 'c' is located at index 15
Last 'a' up to position 25 is located at index 13
Last '$' in the 5 positions ending at 15 is located at index -1

First "def" is located at index 3
First "def" starting at 7 is located at index 16
First "hello" in the 15 positions starting at 5 is located at index -1

Last "def" is located at index 16
Last "def" up to position 25 is located at index 16
Last "hello" in the 15 positions ending at 20 is located at index -1

First 'c', 'a' or '$' is located at index 0
First 'c', 'a' or '$' starting at 7 is located at index 13
First 'c', 'a' or '$' in the 5 positions starting at 7 is located at index -1

Last 'c', 'a' or '$' is located at index 15
Last 'c', 'a' or '$' up to position 1 is located at index 0
Last 'c', 'a' or '$' in the 5 positions ending at 25 is located at index -1

```

### Ил. 16.5. Поиск символов и подстрок в объектах string (окончание)

В строках 22, 24 и 26 метод `LastIndexOf` используется для поиска последнего вхождения символа в строке. Метод `LastIndexOf` выполняет поиск от конца строки к началу. Если символ удастся найти, `LastIndexOf` возвращает индекс заданного символа; в противном случае `LastIndexOf` возвращает `-1`. Существуют три версии метода `LastIndexOf`. Выражение в строке 22 использует версию, в аргументе которой передается искомый символ. Выражение в строке 24 использует версию с двумя аргументами — искомым символом и максимальным индексом, с которого должен начаться поиск в обратном направлении. В выражении в строке 26 используется третья версия метода `LastIndexOf` с тремя аргументами — искомым символом, начальным индексом, от которого поиск ведется в обратном направлении, и длиной участка строки (количеством символов), в котором ведется поиск.

В строках 29–44 используются версии `IndexOf` и `LastIndexOf`, которым в первом аргументе передается `string` вместо символа. Эти версии работают аналогично описанным выше, за исключением того, что они ищут последовательности символов (или подстроки), заданные аргументом `string`.

В строках 47–64 используются методы `IndexOfAny` и `LastIndexOfAny`, которым в первом аргументе передается массив символов. Эти версии методов тоже работают идентично описанным выше, но возвращают индекс первого вхождения любого из символов в аргументе-массиве.



### ТИПИЧНАЯ ОШИБКА 16.2

В перегруженных методах `LastIndexOf` и `LastIndexOfAny`, получающих три параметра, второй аргумент должен быть больше третьего либо равен ему. На первый взгляд это выглядит противоестественно, но вспомните, что поиск ведется от конца строки к началу.

## 16.7. Извлечение подстроки

Класс `string` предоставляет два метода `Substring`, которые создают новую строку копированием части существующей строки. Оба метода возвращают новый объект `string`. Использование обоих методов продемонстрировано в приложении на ил. 16.6.

```
1 // Ил. 16.6: SubString.cs
2 // Использование метода Substring.
3 using System;
4
5 class SubString
6 {
7     public static void Main( string[] args )
8     {
9         string letters = "abcdefghijklmabcdefghijklm";
10
11         // Вызов метода Substring с одним параметром
12         Console.WriteLine( "Substring from index 20 to end is \"\" +
13             letters.Substring( 20 ) + "\"\" );
14
15         // Вызов метода Substring с двумя параметрами
16         Console.WriteLine( "Substring from index 0 of length 6 is \"\" +
17             letters.Substring( 0, 6 ) + "\"\" );
18     } // Конец метода Main
19 } // Конец класса SubString
```

```
Substring from index 20 to end is "hijklm"
Substring from index 0 of length 6 is "abcdef"
```

**Ил. 16.6.** Подстроки, выделенные из строк

Команда в строке 13 использует метод `Substring` с одним аргументом `int`. Аргумент задает индекс, начиная с которого метод копирует символы в исходную строку. Возвращаемая подстрока содержит копию символов от начального индекса до конца строки. Если индекс, заданный в аргументе, выходит за границы строки, программа выдает исключение `ArgumentOutOfRangeException`.

Вторая версия метода `Substring` (строка 17) получает два аргумента `int`. Первый аргумент задает начальный индекс, с которого метод копирует символы из исходной строки. Второй аргумент задает длину копируемой подстроки. Возвращаемая подстрока содержит копию заданных символов из исходной строки. Если заданная длина подстроки слишком велика (то есть вызов пытается включить в подстроку символы за концом исходной строки), выдается исключение `ArgumentOutOfRangeException`.

## 16.8. Конкатенация

Оператор `+` — не единственный способ выполнения конкатенации. Статический метод `Concat` класса `string` (ил. 16.7) возвращает новый объект `string`, содержащий объединенные символы двух исходных строк. В строке 16 следующего примера

символы `string2` присоединяются к копии `string1` методом `Concat`. Команда в строке 16 не изменяет содержимого исходных строк.

```

1 // Ил. 16.7: SubConcatenation.cs
2 // Использование метода Concat класса string.
3 using System;
4
5 class StringConcatenation
6 {
7     public static void Main( string[] args )
8     {
9         string string1 = "Happy ";
10        string string2 = "Birthday";
11
12        Console.WriteLine( "string1 = \"" + string1 + "\"\n" +
13            "string2 = \"" + string2 + "\"" );
14        Console.WriteLine(
15            "\nResult of string.Concat( string1, string2 ) = " +
16            string.Concat( string1, string2 ) );
17        Console.WriteLine( "string1 after concatenation = " + string1 );
18    } // Конец Main
19 } // Конец класса StringConcatenation

```

```

string1 = "Happy "
string2 = "Birthday"

```

```

Result of string.Concat( string1, string2 ) = Happy Birthday
string1 after concatenation = Happy

```

**Ил. 16.7.** Статический метод `Concat`

## 16.9. Другие методы string

Класс `string` предоставляет ряд методов, возвращающих модифицированные копии строк. Использование методов `Replace`, `ToLower`, `ToUpper` и `Trim` продемонстрировано в приложении на ил. 16.8.

```

1 // Ил. 16.8: StringMethods2.cs
2 // Использование статических методов Replace, ToLower,
3 // ToUpper, Trim и ToString.
4 using System;
5
6 class StringMethods2
7 {
8     public static void Main( string[] args )
9     {
10        string string1 = "cheers!";
11        string string2 = "GOOD BYE ";
12        string string3 = " spaces ";
13
14        Console.WriteLine( "string1 = \"" + string1 + "\"\n" +

```

**Ил. 16.8.** Методы `Replace`, `ToLower`, `ToUpper` и `Trim` (продолжение ↗)

```

15     "string2 = \"" + string2 + "\"\n" +
16     "string3 = \"" + string3 + "\"" );
17
18     // Вызов метода Replace
19     Console.WriteLine(
20         "\nReplacing \"e\" with \"E\" in string1: \"" +
21         string1.Replace( 'e', 'E' ) + "\"" );
22
23     // Вызов методов ToLower и ToUpper
24     Console.WriteLine( "\nstring1.ToUpper() = \"" +
25         string1.ToUpper() + "\"\nstring2.ToLower() = \"" +
26         string2.ToLower() + "\"" );
27
28     // Вызов метода Trim
29     Console.WriteLine( "\nstring3 after trim = \"" +
30         string3.Trim() + "\"" );
31
32     Console.WriteLine( "\nstring1 = \"" + string1 + "\"" );
33 } // Конец Main
34 } // Конец класса StringMethods2

```

```

string1 = "cheers!"
string2 = "GOOD BYE "
string3 = " spaces "

Replacing "e" with "E" in string1: "chEers!"

string1.ToUpper() = "CHEERS!"
string2.ToLower() = "good bye "

string3 after trim = "spaces"

string1 = "cheers!"

```

**Ил. 16.8.** Методы Replace, ToLower, ToUpper и Trim (окончание)

В строке 21 метод `Replace` возвращает новую строку, полученную из `string1` заменой каждого вхождения символа 'e' на 'E'. Метод `Replace` получает два аргумента: искомым символом и другим символом, которым заменяются все вхождения первого аргумента. Исходная строка остается неизменной. Если вхождения первого аргумента не обнаружены, метод возвращает исходную строку. Перегруженная версия этого метода позволяет передать в аргументах два объекта `string`.

Метод `ToUpper` создает новый объект `string` (строка 25), в котором все буквы нижнего регистра из `string1` заменены их эквивалентами в верхнем регистре. Содержимое исходной строки при этом не изменяется. Если исходная строка не содержит символов, которые должны быть преобразованы, то возвращается исходная строка. В строке 26 метод `ToLower` возвращает новую строку, в которой все буквы верхнего регистра из `string2` заменены своими эквивалентами в нижнем регистре. Исходная строка при этом не изменяется. Как и в случае с `ToUpper`, при отсутствии символов, преобразуемых к нижнему регистру, метод `ToLower` возвращает исходную строку.

В строке 30 метод `Trim` используется для удаления всех пропусков в начале и конце строки. Этот метод особенно удобен для обработки данных, введенных пользователем (например, в текстовом поле). Другая версия метода `Trim` получает символьный массив и возвращает копию строки, у которой в начале и конце удалены все символы, встречающиеся в аргументе-массиве.

## 16.10. Класс `StringBuilder`

Класс `string` предоставляет широкие возможности обработки строк, однако содержимое объекта `string` изменяться не может. Так, операция конкатенации фактически создает новые строки — оператор `+=` создает новую строку и присваивает ссылку на нее переменной в левой части оператора `+=`.

В следующих разделах рассматриваются средства класса `StringBuilder` (пространство имен `System.Text`), предназначенного для создания и обработки динамических строковых данных — то есть *изменяемых строк*. Объект `StringBuilder` может хранить некоторое количество символов, определяемое его емкостью. При заполнении `StringBuilder` емкость объекта автоматически увеличивается, чтобы в нем помещались дополнительные символы. Как вы увидите, для выполнения конкатенации наряду с операторами `+` и `+=` могут использоваться члены класса `StringBuilder` (такие, как `Append` и `AppendFormat`). Класс `StringBuilder` особенно полезен при работе с многочисленными строками, и он гораздо эффективнее создания отдельных неизменяемых строк.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 16.2

Объекты класса `string` являются неизменяемыми (то есть константными), тогда как объекты класса `StringBuilder` могут изменяться. Компилятор `C#` может включать некоторые оптимизации, связанные с работой со строками (такие, как использование одного объекта `string` с множественными ссылками), потому что он знает, что эти объекты не изменятся.

Класс `StringBuilder` предоставляет шесть перегруженных конструкторов. Три из них продемонстрированы в классе `StringBuilderConstructor` (ил. 16.9).

В строке 10 конструктор `StringBuilder` без параметров используется для создания объекта `StringBuilder`, который не содержит символов и имеет исходную емкость, зависящую от реализации. В строке 11 конструктор `StringBuilder` с аргументом `int` создает объект `StringBuilder` без символов, емкость которого определяется аргументом `int` (например, 10). В строке 12 используется конструктор `StringBuilder` с аргументом `string`, который создает объект `StringBuilder` с символами аргумента, — при этом исходная емкость может отличаться от размера `string`. В строках 14–16 метод `ToString` класса `StringBuilder` неявно используется для получения строкового представления содержимого `StringBuilder`.

```
1 // Ил. 16.9: StringBuilderConstructor.cs
2 // Демонстрация конструкторов класса StringBuilder.
3 using System;
4 using System.Text;
5
6 class StringBuilderConstructor
7 {
8     public static void Main( string[] args )
9     {
10         StringBuilder buffer1 = new StringBuilder();
11         StringBuilder buffer2 = new StringBuilder( 10 );
12         StringBuilder buffer3 = new StringBuilder( "hello" );
13
14         Console.WriteLine( "buffer1 = \"" + buffer1 + "\"" );
15         Console.WriteLine( "buffer2 = \"" + buffer2 + "\"" );
16         Console.WriteLine( "buffer3 = \"" + buffer3 + "\"" );
17     } // Конец Main
18 } // Конец класса StringBuilderConstructor
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

**Ил. 16.9.** Конструкторы класса `StringBuilder`

## 16.11. Свойства `Length` и `Capacity`, метод `EnsureCapacity` и индексатор класса `StringBuilder`

Класс `StringBuilder` предоставляет свойства `Length` и `Capacity` для получения текущего количества символов в `StringBuilder` и количества символов, которые могут храниться в `StringBuilder` без выделения дополнительной памяти. Эти свойства также могут использоваться для увеличения и уменьшения длины или емкости `StringBuilder`. Метод `EnsureCapacity` позволяет сократить количество операций увеличения емкости `StringBuilder`. Он гарантирует, что емкость `StringBuilder` не меньше заданной величины. Использование этих методов и свойств продемонстрировано в программе на ил. 16.10.

```
1 // Ил. 16.10: StringBuilderFeatures.cs
2 // Демонстрация некоторых возможностей класса StringBuilder.
3 using System;
4 using System.Text;
5
6 class StringBuilderFeatures
7 {
8     public static void Main( string[] args )
9     {
10         StringBuilder buffer =
11             new StringBuilder( "Hello, how are you?" );
```

**Ил. 16.10.** Операции с размером `StringBuilder` (продолжение )

```
12
13 // Использование свойств Length и Capacity
14 Console.WriteLine( "buffer = " + buffer +
15     "\nLength = " + buffer.Length +
16     "\nCapacity = " + buffer.Capacity );
17
18 buffer.EnsureCapacity( 75 ); // Задать емкость не ниже 75
19 Console.WriteLine( "\nNew capacity = " +
20     buffer.Capacity );
21
22 // Усечение StringBuilder посредством задания Length
23 buffer.Length = 10;
24 Console.Write( "\nNew length = " +
25     buffer.Length + "\nbuffer = " );
26
27 // Использование индекатора StringBuilder
28 for ( int i = 0; i < buffer.Length; ++i )
29     Console.Write( buffer[ i ] );
30
31 Console.WriteLine( "\n" );
32 } // Конец Main
33 } // Конец класса StringBuilderFeatures
```

```
buffer = Hello, how are you?
Length = 19
Capacity = 19
New capacity = 75
New length = 10
buffer = Hello, how
```

#### Ил. 16.10. Операции с размером StringBuilder (окончание)

Программа содержит один объект `StringBuilder` с именем `buffer`. В строках 10–11 используется конструктор `StringBuilder`, который получает строковый аргумент для создания экземпляра `StringBuilder` и его инициализации значением "Hello, how are you?". В строках 14–16 выводится содержимое, длина и емкость `StringBuilder`.

В строке 18 емкость `StringBuilder` увеличивается до 75 символов (минимум). Если в объект `StringBuilder` будут добавлены новые символы, превышающие его текущую емкость, то объект расширяется так, как если бы для него был вызван метод `EnsureCapacity`.

В строке 23 свойство `Length` используется для задания длины `StringBuilder`, равной 10, — свойство `Capacity` при этом не изменяется. Если заданная длина меньше текущего количества символов в `StringBuilder`, то содержимое `StringBuilder` усекается до заданной длины. Если заданная длина больше текущего количества символов в `StringBuilder`, к `StringBuilder` присоединяются нуль-символы (`'\0'`) до тех пор, пока общее количество символов в `StringBuilder` не сравняется с заданной длиной.

## 16.12. Методы Append и AppendFormat класса StringBuilder

Класс `StringBuilder` предоставляет 19 перегруженных методов `Append`, которые добавляют различные типы значений в конец `StringBuilder`. В Framework Class Library включены версии для всех простых типов и символьных массивов, `string` и `object`. (Напомним, что метод `ToString` выдает строковое представление произвольного объекта.) Каждый метод получает аргумент, преобразует его в `string` и присоединяет его к `StringBuilder`. Использование методов `Append` продемонстрировано на ил. 16.11.

```
1 // Ил. 16.11: StringBuilderAppend.cs
2 // Использование методов Append класса StringBuilder.
3 using System;
4 using System.Text;
5
6 class StringBuilderAppend
7 {
8     public static void Main( string[] args )
9     {
10         object objectValue = "hello";
11         string stringValue = "good bye";
12         char[] characterArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
13         bool booleanValue = true;
14         char characterValue = 'Z';
15         int integerValue = 7;
16         long longValue = 1000000;
17         float floatValue = 2.5F; // Суффикс F - признак типа float
18         double doubleValue = 33.333;
19         StringBuilder buffer = new StringBuilder();
20
21         // Использование метода Append для присоединения данных к буферу
22         buffer.Append( objectValue );
23         buffer.Append( " " );
24         buffer.Append( stringValue );
25         buffer.Append( " " );
26         buffer.Append( characterArray );
27         buffer.Append( " " );
28         buffer.Append( characterArray, 0, 3 );
29         buffer.Append( " " );
30         buffer.Append( booleanValue );
31         buffer.Append( " " );
32         buffer.Append( characterValue );
33         buffer.Append( " " );
34         buffer.Append( integerValue );
35         buffer.Append( " " );
36         buffer.Append( longValue );
37         buffer.Append( " " );
38         buffer.Append( floatValue );
39         buffer.Append( " " );
40         buffer.Append( doubleValue );
```

**Ил. 16.11.** Методы `Append` класса `StringBuilder` (продолжение ↗)



```

41
42     Console.WriteLine( "buffer = " + buffer.ToString() + "\n" );
43 } // Конец Main
44 } // Конец класса StringBuilderAppend

```

```
buffer = hello good bye abcdef abc True Z 7 1000000 2.5 33.333
```

### Ил. 16.11. Методы Append класса StringBuilder (окончание)

В строках 22–40 десять разных перегруженных методов Append используются для присоединения строковых представлений объектов, созданных в строках 10–18, в конец текста `StringBuilder`. Класс `StringBuilder` также предоставляет метод `AppendFormat`, который преобразует строку к заданному формату и присоединяет ее к `StringBuilder`. Пример использования этого метода представлен на ил. 16.12.

```

1 // Ил. 16.12: StringBuilderAppendFormat.cs
2 // Использование метода AppendFormat.
3 using System;
4 using System.Text;
5
6 class StringBuilderAppendFormat
7 {
8     public static void Main( string[] args )
9     {
10         StringBuilder buffer = new StringBuilder();
11
12         // Отформатированная строка
13         string string1 = "This {0} costs: {1:C}.\n";
14
15         // Массив для аргументов string1
16         object[] objectArray = new object[ 2 ];
17
18         objectArray[ 0 ] = "car";
19         objectArray[ 1 ] = 1234.56;
20
21         // Присоединение к буферу отформатированной строки
22         buffer.AppendFormat( string1, objectArray );
23
24         // Отформатированная строка
25         string string2 = "Number:{0:d3}.\n" +
26             "Number right aligned with spaces:{0, 4}.\n" +
27             "Number left aligned with spaces:{0, -4}.";
28
29         // Присоединение к буферу отформатированной строки с аргументом
30         buffer.AppendFormat( string2, 5 );
31
32         // Вывод отформатированных строк
33         Console.WriteLine( buffer.ToString() );
34     } // Конец Main
35 } // Конец класса StringBuilderAppendFormat

```

```

This car costs: $1,234.56.
Number:005.
Number right aligned with spaces: 5.
Number left aligned with spaces:5 .

```

### Ил. 16.12. Метод AppendFormat класса StringBuilder

В строке 13 создается объект `string` с форматными данными. Текст в фигурных скобках описывает форматирование некоторых данных. Форматные элементы определяются по схеме `{X[, Y][: ФорматнаяСтрока]}`, где *X* — количество форматируемых аргументов (начиная с нуля); *Y* — необязательный аргумент (положительный или отрицательный), определяющий количество символов в результате. Если длина полученной строки меньше *Y*, то она дополняется пробелами. С положительным значением строка выравнивается по правому краю, а с отрицательным — по левому. Необязательный элемент *ФорматнаяСтрока* применяет к аргументу определенный формат (денежный, экспоненциальный и т. д.). В данном случае запись `{0}` означает, что выводится первый аргумент. Запись `{1:c}` означает, что второй аргумент должен форматироваться как денежная сумма.

В строке 22 используется версия `AppendFormat` с двумя параметрами — строкой, задающей формат, и массивом объектов, которые служат аргументами форматной строки. Аргумент с обозначением `{0}` находится в массиве объектов с индексом 0.

В строках 25–27 определяется другая строка, используемая для форматирования. Первый формат `{0:d3}` означает, что первый аргумент форматируется как целое число из трех цифр; если число содержит менее трех цифр, оно дополняется начальными нулями до нужной длины. Следующий формат `{0,4}` указывает, что отформатированная строка состоит из четырех символов и выравнивается по правому краю. Третий формат `{0,-4}` указывает, что отформатированная строка должна быть выровнена по левому краю.

В строке 30 используется версия `AppendFormat` с двумя параметрами — объектом `string`, содержащим описание формата, и объектом, к которому формат применяется (в нашем примере это число 5). На ил. 16.12 представлен результат применения двух версий `AppendFormat` с соответствующими аргументами.

## 16.13. Методы Insert, Remove и Replace класса StringBuilder

Класс `StringBuilder` предоставляет 18 перегруженных методов `Insert` для разных типов данных, вставляемых в произвольную позицию `StringBuilder`. Класс предоставляет версии для всех простых типов, а также символьных массивов, `string` и `object`. Каждый метод преобразует свой второй аргумент в строку и вставляет результат в `StringBuilder` перед символом в позиции, заданной первым аргументом. Индекс, заданный первым аргументом, должен быть больше либо равен 0 и меньше длины `StringBuilder`; в противном случае выдается исключение `ArgumentOutOfRangeException`.

Класс `StringBuilder` также предоставляет метод `Remove` для удаления произвольной части `StringBuilder`. При вызове `Remove` передаются два аргумента — индекс, с которого начинается удаление, и количество удаляемых символов. Сумма начального индекса и количества удаляемых символов должна быть меньше длины `StringBuilder`; в противном случае программа выдает исключение `ArgumentOutOfRangeException`. Использование методов `Insert` и `Remove` продемонстрировано на ил. 16.13.

```
1 // Ил. 16.13: StringBuilderInsertRemove.cs
2 // Использование методов Insert и Remove
3 // класса StringBuilder.
4 using System;
5 using System.Text;
6
7 class StringBuilderInsertRemove
8 {
9     public static void Main( string[] args )
10    {
11        object objectValue = "hello";
12        string stringValue = "good bye";
13        char[] characterArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
14        bool booleanValue = true;
15        char characterValue = 'K';
16        int integerValue = 7;
17        long longValue = 10000000;
18        float floatValue = 2.5F; // Суффикс F - признак типа float
19        double doubleValue = 33.333;
20        StringBuilder buffer = new StringBuilder();
21
22        // Вставка значений в буфер
23        buffer.Insert( 0, objectValue );
24        buffer.Insert( 0, " " );
25        buffer.Insert( 0, stringValue );
26        buffer.Insert( 0, " " );
27        buffer.Insert( 0, characterArray );
28        buffer.Insert( 0, " " );
29        buffer.Insert( 0, booleanValue );
30        buffer.Insert( 0, " " );
31        buffer.Insert( 0, characterValue );
32        buffer.Insert( 0, " " );
33        buffer.Insert( 0, integerValue );
34        buffer.Insert( 0, " " );
35        buffer.Insert( 0, longValue );
36        buffer.Insert( 0, " " );
37        buffer.Insert( 0, floatValue );
38        buffer.Insert( 0, " " );
39        buffer.Insert( 0, doubleValue );
40        buffer.Insert( 0, " " );
41
42        Console.WriteLine( "buffer after Inserts: \n" + buffer + "\n" );
43
44        buffer.Remove( 10, 1 ); // Удаление 2 в 2.5
45        buffer.Remove( 4, 4 ); // Удаление .333 в 33.333
46
47        Console.WriteLine( "buffer after Removes:\n" + buffer );
48    } // Конец Main
49 } // Конец класса StringBuilderInsertRemove
```

```
buffer after Inserts:
33.333 2.5 10000000 7 K True abcdef good bye hello
buffer after Removes:
33 .5 10000000 7 K True abcdef good bye hello
```

**Ил. 16.13.** Вставка и удаление текста из StringBuilder

Другой полезный метод класса `StringBuilder` — `Replace` — ищет заданную подстроку или символ и заменяет их другой подстрокой или символом. Использование этого метода продемонстрировано на ил. 16.14.

```

1 // Ил. 16.14: StringBuilderReplace.cs
2 // Использование метода Replace.
3 using System;
4 using System.Text;
5
6 class StringBuilderReplace
7 {
8     public static void Main( string[] args )
9     {
10         StringBuilder builder1 =
11             new StringBuilder( "Happy Birthday Jane" );
12         StringBuilder builder2 =
13             new StringBuilder( "goodbye greg" );
14
15         Console.WriteLine( "Before replacements:\n" +
16             builder1.ToString() + "\n" + builder2.ToString() );
17
18         builder1.Replace( "Jane", "Greg" );
19         builder2.Replace( 'g', 'G', 0, 5 );
20
21         Console.WriteLine( "\nAfter replacements:\n" +
22             builder1.ToString() + "\n" + builder2.ToString() );
23     } // Конец Main
24 } // Конец класса StringBuilderReplace

```

```

Before Replacements:
Happy Birthday Jane
good bye greg

```

```

After replacements:
Happy Birthday Greg
Goodbye greg

```

**Ил. 16.14.** Замена текста в `StringBuilder`

В строке 18 метод `Replace` используется для замены всех вхождений подстроки "Jane" в `builder1` подстрокой "Greg". Другая перегруженная версия этого метода получает два символа и заменяет каждое вхождение первого символа вторым. В строке 19 используется перегруженная версия `Replace` с четырьмя параметрами, из которых первые два могут быть символами или строками, а вторые два относятся к типу `int`. Метод заменяет все вхождения первого символа вторым символом (или первой строки — второй), начиная с индекса, заданного первым параметром `int`, на протяжении длины, заданной вторым параметром `int`. Таким образом, в данном примере `Replace` просматривает всего пять символов, начиная с символа с индексом 0. Как видно из результатов, эта версия `Replace` заменяет `g` на `G` в слове "good", но не в "greg". Дело в том, что вхождения «`g`» в "greg" не входят в диапазон, определяемый аргументами `int` (между индексами 0 и 4).

## 16.14. Методы Char

В C# существует концепция *структур*, отдаленно похожих на классы. Структуры представляют значимые типы. Они, как и классы, могут обладать методами и свойствами и могут использовать модификаторы доступа `public` и `private`. Кроме того, для обращения к членам структур используется оператор «точка» (`.`).

Простые типы в действительности являются синонимами для структурных типов. Например, тип `int` определяется структурой `System.Int32`, тип `long` — структурой `System.Int64` и т. д. Все структурные типы являются производными от класса `ValueType`, производного от `object`. Кроме того, все структурные типы неявно запечатаны, поэтому они не поддерживают виртуальные или абстрактные методы, а их члены не могут объявляться с модификаторами `protected` или `protected internal`.

В структуре `System.Char`<sup>1</sup>, представляющей символы, многие методы являются статическими, получают как минимум один символьный аргумент и выполняют с ним проверку или некоторую обработку. В следующем примере будет представлено несколько таких методов. На ил. 16.15 продемонстрированы статические методы, которые проверяют символы на принадлежность к определенной категории, а также статические методы для преобразования регистра символов.

```

1 // Ил. 16.15: StaticCharMethods.cs
2 // Статические методы проверки символов и преобразования
3 // регистра из структуры Char
4 using System;
5
6 class StaticCharMethods
7 {
8     static void Main( string[] args )
9     {
10         Console.Write( "Enter a character: " );
11         char character = Convert.ToChar( Console.ReadLine() );
12
13         Console.WriteLine( "is digit: {0}", Char.IsDigit( character ) );
14         Console.WriteLine( "is letter: {0}", Char.IsLetter( character ) );
15         Console.WriteLine( "is letter or digit: {0}",
16             Char.IsLetterOrDigit( character ) );
17         Console.WriteLine( "is lower case: {0}",
18             Char.IsLower( character ) );
19         Console.WriteLine( "is upper case: {0}",
20             Char.IsUpper( character ) );

```

**Ил. 16.15.** Статические методы проверки символов  
и преобразования регистра (продолжение ↗)

<sup>1</sup> Подобно тому как ключевое слово `string` является синонимом для класса `String`, ключевое слово `char` является синонимом для структуры `Char`. В тексте книги имя `Char` будет использоваться при вызове статических методов структуры `Char`, а во всех остальных случаях будет использоваться термин `char`.

```
21 Console.WriteLine( "to upper case: {0}",
22     Char.ToUpper( character ) );
23 Console.WriteLine( "to lower case: {0}",
24     Char.ToLower( character ) );
25 Console.WriteLine( "is punctuation: {0}",
26     Char.IsPunctuation( character ) );
27 Console.WriteLine( "is symbol: {0}", Char.IsSymbol( character ) );
28 } // Конец Main
29 } // Конец класса StaticCharMethods
```

```
Enter a character: A
is digit: False
is letter: True
is letter or digit: True
is lower case: False
is upper case: True
to upper case: A
to lower case: a
is punctuation: False
is symbol: False
```

```
Enter a character: 8
is digit: True
is letter: False
is letter or digit: True
is lower case: False
is upper case: False
to upper case: 8
to lower case: 8
is punctuation: False
is symbol: False
```

```
Enter a character: @
is digit: False
is letter: False
is letter or digit: False
is lower case: False
is upper case: False
to upper case: @
to lower case: @
is punctuation: True
is symbol: False
```

```
Enter a character: m
is digit: False
is letter: True
is letter or digit: True
is lower case: True
is upper case: False
to upper case: M
to lower case: m
is punctuation: False
is symbol: False
```

**Ил. 16.15.** Статические методы проверки символов  
и преобразования регистра (продолжение ☞)

```

Enter a character: +
is digit: False
is letter: False
is letter or digit: False
is lower case: False
is upper case: False
to upper case: +
to lower case: +
is punctuation: False
is symbol: True

```

**Ил. 16.15.** Статические методы проверки символов и преобразования регистра (окончание)

Символ, введенный пользователем, анализируется в строках 13–27. В строке 13 метод `IsDigit` проверяет, является ли введенный символ цифрой. Для цифр метод возвращает `true`; в остальных случаях возвращается `false` (обратите внимание: значения `bool` выводятся с прописной буквы). В строке 14 метод `IsLetter` проверяет, является ли символ буквой. В строке 16 метод `IsLetterOrDigit` проверяет, является ли введенный символ буквой или цифрой.

Вызов метода `IsLower` в строке 18 проверяет, является ли символ буквой нижнего регистра. В строке 20 метод `IsUpper` проверяет, является ли символ буквой верхнего регистра. В строке 22 метод `ToUpper` преобразует символ в эквивалентный символ верхнего регистра. Если эквивалент в верхнем регистре существует, метод возвращает преобразованный символ; в противном случае возвращается исходный аргумент. В строке 24 метод `ToLower` преобразует символ в эквивалентный символ нижнего регистра. Если эквивалент в нижнем регистре существует, метод возвращает преобразованный символ; в противном случае возвращается исходный аргумент.

В строке 26 метод `IsPunctuation` проверяет, является ли символ знаком препинания — "!", ":", ")" и т. д. Наконец, в строке 27 метод `IsSymbol` проверяет, является ли символ специальным знаком (например, "+", "=" или "^").

Структурный тип `Char` также содержит другие методы, не показанные в этом примере. Многие статические методы аналогичны приведенным — например, `IsWhiteSpace` проверяет, является ли символ пропуском (например, символом новой строки, табуляцией или пробелом). Структура также содержит открытые методы экземпляров; многие из них (такие, как `ToString` и `Equals`) уже встречались в других классах. В частности, в эту группу входит метод `CompareTo`, используемый для сравнения двух символов.

## 16.15. Итоги

В этой главе были описаны средства библиотеки `Framework Class Library` для работы со строками и символами. Глава начинается с представления основных принципов реализации строк и символов. Вы научились определять длину строки, копировать

строки, обращаться к отдельным символам, проводить поиск, извлекать подстроки из больших строк, сравнивать строки, выполнять конкатенацию, заменять символы в строках и преобразовывать символы к верхнему или нижнему регистру.

Вторая часть главы посвящена классу `StringBuilder`, предназначенному для динамического построения строк. Вы узнали, как определять и задавать размер объекта `StringBuilder` и как выполнять присоединение, вставку, удаление и замену символов в объекте `StringBuilder`. Далее были представлены методы проверки символов типа `Char`, при помощи которых программа может проверить, является ли символ цифрой, буквой, буквой верхнего или нижнего регистра, знаком препинания или символом, отличным от знака препинания, а также методы преобразования символов к верхнему или нижнему регистру.



# 17 Файлы и потоки

## 17.1. Введение

Переменные и массивы предназначены для *временного* хранения данных — при выходе локальной переменной «из области действия» или завершении программы данные теряются. С другой стороны, файлы (и базы данных, которые будут рассматриваться в главе 22) предназначены для долгосрочного хранения больших объемов данных даже после завершения программы, создавшей эти данные. Файлы с данными хранятся на *внешних запоминающих устройствах*: магнитных и оптических дисках, флеш-памяти и магнитных лентах. В этой главе вы научитесь создавать, обновлять и обрабатывать файлы данных в программах C#.

Глава открывается обзором иерархии данных, от уровня битов до файлов. Затем будут рассмотрены некоторые классы для работы с файлами из Framework Class Library. Приведенные примеры покажут, как получить информацию о файлах и каталогах на вашем компьютере. В оставшейся части главы рассматриваются операции чтения и записи текстовых и двоичных файлов.

## 17.2. Иерархия данных

В конечном счете все данные, обрабатываемые компьютером, представляют собой последовательности 0 и 1. Это связано с простотой и экономичностью построения электронных устройств, которые могут находиться в двух стабильных состояниях — одно состояние представляет 0, а другое 1. По сути, все впечатляющие достижения современных компьютеров основаны на простейших манипуляциях с нулями и единицами.

### Биты

Наименьший блок данных, с которым может работать компьютер, называется *битом*. Бит может принимать одно из двух значений: 0 или 1. Электронные устройства компьютера могут выполнять простейшие манипуляции с битами: проверка значения, присваивание и инвертирование (замена 1 на 0 или 0 на 1).

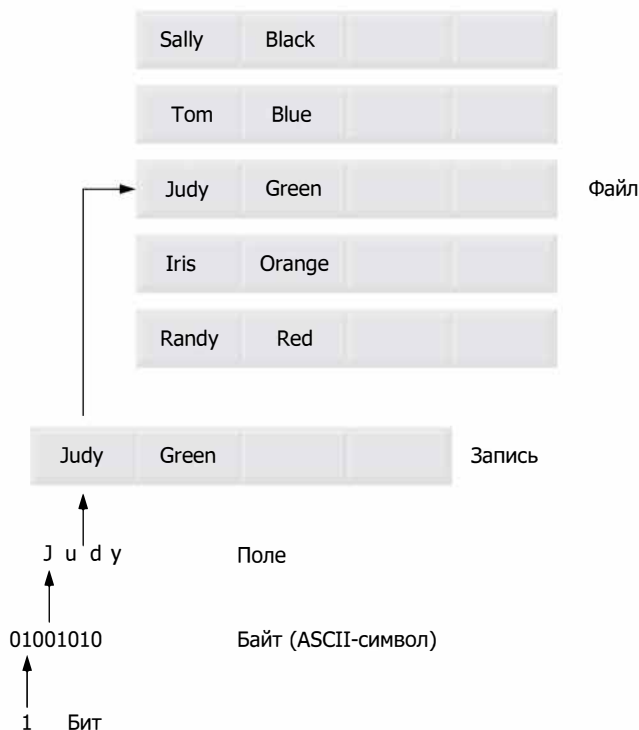
## Символы

Программировать на уровне битов неудобно — человеку проще работать с данными в форме десятичных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8 и 9), букв и специальных знаков (\$, @, %, &, \*, (, ), -, +, «, :, ?, / и т. д.), которые называются *символами*. Совокупность всех символов, используемых для написания программ и представления данных на конкретном компьютере, называется *набором символов* данного компьютера. Так как компьютеры могут работать только с 0 и 1, каждый символ в наборе символов представляется последовательностью 0 и 1. *Байты* состоят из 8 бит. В C# используется набор символов Юникод ([www.unicode.org](http://www.unicode.org)). Программисты создают программы и блоки данных из символов; компьютеры обрабатывают эти символы как последовательности битов.

## Поля

Подобно тому как символы состоят из битов, поля состоят из символов. *Поле* представляет собой группу символов, передающую некоторую информацию. Например, поле из символов верхнего и нижнего регистра может представлять имя человека.

Блоки данных, обрабатываемых компьютером, образуют иерархию (ил. 17.1), в которой данные укрупняются и усложняются с переходом от битов к символам, полям и более сложным составным данным.



Ил. 17.1. Иерархия данных

## Записи и файлы

Как правило, запись (которая может представляться классом) состоит из нескольких взаимосвязанных полей. Например, в системе расчета зарплаты запись конкретного работника может включать следующие поля:

- ☐ табельный номер работника;
- ☐ имя;
- ☐ адрес;
- ☐ почасовая ставка;
- ☐ налоговые вычеты;
- ☐ годовой заработок;
- ☐ сумма удержанного налога.

В приведенном примере каждое поле в наборе относится к одному работнику. *Файл* представляет собой группу взаимосвязанных записей<sup>1</sup>. Так, файл с платежной ведомостью компании обычно содержит одну запись для каждого работника. Платежная ведомость небольшой компании может состоять из 22 записей, а в большой компании количество записей может достигать 100 000. Ничто не мешает компании иметь много файлов с миллионами, миллиардами и даже триллионами символов.

## Ключ записи

Для упрощения выборки конкретных записей из файла по крайней мере одно поле в каждой записи выбирается в качестве *ключа*, который идентифицирует запись как относящуюся к конкретному человеку или сущности и отличает ее от остальных. Например, в записи платежной ведомости в качестве ключа обычно используется табельный номер сотрудника.

## Последовательные файлы

Существует много способов организации записей в файлах. В одном из распространенных вариантов — *последовательном файле* — записи хранятся в порядке значений ключевого поля. В первой записи файла хранится работник с наименьшим табельным номером, а в последующих записях табельные номера последовательно возрастают.

## Базы данных

В большинстве коммерческих приложений данные хранятся во множестве разных файлов. Например, компания может создать файлы платежной ведомости, дебиторской задолженности, кредиторской задолженности, данных складского учета и т. д.

<sup>1</sup> В общем случае файл содержит произвольные данные в произвольных форматах. В некоторых операционных системах файл рассматривается как простая совокупность байтов, а любая организация байтов в файле (такая, как упорядочение данных по записям) является представлением, которое определяется прикладным программистом.

Взаимосвязанные данные часто хранятся в базах данных. Совокупность программ, предназначенных для создания и управления базами данных, называется *системой управления базами данных*, или *СУБД*. Базы данных рассматриваются в главе 22.

## 17.3. Файлы и потоки

В C# каждый файл рассматривается как последовательный *поток* байтов (ил. 17.2). Каждый файл завершается либо маркером конца файла, либо определенной комбинацией байтов, хранящейся в административной структуре данных, которая поддерживается системой. При открытии файла создается объект, с которым связывается поток данных. При выполнении консольного приложения исполнительная среда создает три объекта потоков, для обращения к которым используются свойства `Console.Out`, `Console.In` и `Console.Error`. Эти объекты используют потоки для упрощения взаимодействия программы с конкретным файлом или устройством. `Console.In` относится к объекту стандартного входного потока, при помощи которого программа получает данные, введенные с клавиатуры. `Console.Out` относится к объекту стандартного потока вывода, позволяющего программе выводить данные на экран. `Console.Error` относится к объекту стандартного потока ошибок, при помощи которого программа выводит сообщения об ошибках на экран. Мы уже использовали `Console.Out` и `Console.In` в консольных приложениях; методы `Write` и `WriteLine` класса `Console` используют `Console.Out` для выполнения вывода, а методы `Read` и `ReadLine` класса `Console` используют `Console.In` для выполнения ввода.



**Ил. 17.2.** Представление файла из  $n$  байтов в C#

В Framework Class Library включено много классов для обработки файлов. Пространство имен `System.IO` содержит классы потоков — такие, как `StreamReader` (для ввода текста из файла), `StreamWriter` (для вывода текста в файл) и `FileStream` (для ввода и вывода из файла). Эти классы потоков являются производными от абстрактных классов `TextReader`, `TextWriter` и `Stream` соответственно. Свойства `Console.In` и `Console.Out` относятся к типам `TextReader` и `TextWriter` соответственно. Для инициализации свойств `Console.In` и `Console.Out` класса `Console` система создает объекты классов, производных от `TextReader` и `TextWriter`.

Абстрактный класс `Stream` обеспечивает функциональность представления потоков в виде байтов. Классы `FileStream`, `MemoryStream` и `BufferedStream` (все они принадлежат к пространству имен `System.IO`) являются производными от класса `Stream`. Класс `FileStream` может использоваться для записи и чтения данных из файлов.

Класс `MemoryStream` обеспечивает прямую передачу данных в память и из нее — эти операции выполняются намного быстрее, чем чтение и запись на внешние

устройства. Класс `BufferedStream` использует буферизацию для чтения и записи данных в поток. *Буферизация* представляет собой механизм повышения производительности ввода-вывода, при которой результаты операций вывода направляются в специальную область памяти — буфер; размер этой области позволяет хранить данные многих операций вывода. Затем непосредственная передача данных на выходное устройство выполняется одной большой физической операцией вывода при заполнении буфера. Операции вывода, результаты которых направляются в выходной буфер в памяти, часто называются *логическими* операциями вывода. Буферизация также способна ускорить операции ввода: изначально в буфер загружается больше данных, чем необходимо, чтобы последующие операции чтения получали данные из быстрой памяти, а не с медленного внешнего устройства.

В этой главе мы используем важнейшие классы потоков для создания последовательных файлов и выполнения операций с ними.

## 17.4. Классы File и Directory

Информация хранится в файлах, которые распределяются по *каталогам* (также называемым *папками*). Классы `File` и `Directory` позволяют программам работать с файлами и каталогами на дисках. Класс `File` способен получать информацию о файлах; он может использоваться для открытия файлов для чтения или записи. Выполнение операций чтения и записи с файлами будет рассмотрено ниже.

На ил. 17.3 представлены некоторые статические методы класса `File` для обработки и получения информации о файлах. Некоторые из этих методов продемонстрированы в приложении на ил. 17.5.

Статический метод	Описание
<code>AppendText</code>	Возвращает объект <code>StreamWriter</code> , который присоединяет текст к существующему файлу или создает новый файл
<code>Copy</code>	Копирует файл в новый файл
<code>Create</code>	Создает файл и возвращает связанный с ним объект <code>FileStream</code>
<code>CreateText</code>	Создает текстовый файл и возвращает связанный с ним объект <code>StreamWriter</code>
<code>Delete</code>	Удаляет заданный файл
<code>Exists</code>	Возвращает <code>true</code> , если заданный файл существует, или <code>false</code> в противном случае
<code>GetCreationTime</code>	Возвращает объект <code>DateTime</code> , представляющий время создания файла
<code>GetLastAccessTime</code>	Возвращает объект <code>DateTime</code> , представляющий время последнего обращения к файлу
<code>GetLastWriteTime</code>	Возвращает объект <code>DateTime</code> , представляющий время последней модификации файла

**Ил. 17.3.** Статические методы класса `File` (неполный список) (продолжение ↗)

Статический метод	Описание
Move	Перемещает файл в заданное место
Open	Возвращает объект FileStream, связанный с файлом и обладающий заданными разрешениями чтения/записи
OpenRead	Возвращает объект FileStream, доступный только для чтения и связанный с заданным файлом
OpenText	Возвращает объект StreamReader, связанный с заданным файлом
OpenWrite	Возвращает объект FileStream для записи, связанный с заданным файлом

**Ил. 17.3.** Статические методы класса File (неполный список) (окончание)

Класс Directory предоставляет средства для работы с каталогами. На ил. 17.4 перечислены некоторые статические методы класса Directory для работы с каталогами; некоторые из них также продемонстрированы в приложении на ил. 17.5. Объект DirectoryInfo, возвращаемый методом CreateDirectory, содержит информацию о каталоге. Большую часть информации, содержащейся в классе DirectoryInfo, также можно получить при помощи методов класса Directory.

Статический метод	Описание
CreateDirectory	Создает каталог и возвращает связанный с ним объект DirectoryInfo
Delete	Удаляет заданный каталог
Exists	Возвращает true, если заданный каталог существует, или false в противном случае
GetDirectories	Возвращает массив string с именами подкаталогов в заданном каталоге
GetFiles	Возвращает массив string с именами файлов в заданном каталоге
GetCreationTime	Возвращает объект DateTime, представляющий время создания каталога
GetLastAccessTime	Возвращает объект DateTime, представляющий время последнего обращения к каталогу
GetLastWriteTime	Возвращает объект DateTime, представляющий время последней модификации каталога
Move	Перемещает каталог в заданное место

**Ил. 17.4.** Статические методы класса Directory

## Использование классов File и Directory

Класс FileTestForm (ил. 17.5) использует методы File и Directory для получения информации о файлах и каталогах. Форма содержит текстовое поле inputTextBox, в котором пользователь вводит имя файла или каталога. Для каждой клавиши, нажатой пользователем во время ввода данных в текстовом поле, программа вызывает метод inputTextBox\_KeyDown (строки 19–75). Если пользователь нажимает клавишу Enter (строка 22), этот метод выводит содержимое файла или каталога в зависимости

от введенного текста. (Если пользователь не нажал клавишу Enter, метод возвращает управление без вывода данных.) В строке 28 метод `Exists` класса `File` проверяет, является ли введенный пользователем текст именем существующего файла. Если проверка дает положительный результат, в строке 31 вызывается закрытый метод `GetInformation` (строки 79–97), который вызывает методы `GetCreationTime` (строка 88), `GetLastWriteTime` (строка 92) и `GetLastAccessTime` (строка 96) класса `File` для получения информации о файле. При возврате управления методом `GetInformation` строка 38 создает экземпляр `StreamReader` для чтения текста из файла. Конструктор `StreamReader` получает строку с именем и путь к открываемому файлу. В строке 40 вызывается метод `ReadToEnd` класса `StreamReader`; он читает все содержимое файла в формате `string` и присоединяет прочитанные данные к `outputTextBox`. После того как файл будет прочитан, блок `using` завершается и уничтожает соответствующий объект, что приводит к закрытию файла.

```

1 // Ил. 17.5: FileTestForm.cs
2 // Использование классов File и Directory.
3 using System;
4 using System.Windows.Forms;
5 using System.IO;
6
7 namespace FileTest
8 {
9     // Вывод содержимого файлов и каталогов
10    public partial class FileTestForm : Form
11    {
12        // Конструктор без параметров
13        public FileTestForm()
14        {
15            InitializeComponent();
16        } // Конец конструктора
17
18        // Метод вызывается при нажатии клавиши пользователем
19        private void inputTextBox_KeyDown( object sender, KeyEventArgs e )
20        {
21            // Проверка нажатия клавиши Enter
22            if ( e.KeyCode == Keys.Enter )
23            {
24                // Получение файла или каталога, заданного пользователем
25                string fileName = inputTextBox.Text;
26
27                // Проверка соответствия fileName файлу
28                if ( File.Exists( fileName ) )
29                {
30                    // Получение даты создания файла, модификации и т. д.
31                    GetInformation( fileName );
32                    StreamReader stream = null; // Объявление StreamReader
33
34                    // Вывод содержимого файла через StreamReader
35                    try
36                    {
37                        // Получение объекта StreamReader и содержимого файла
38                        using ( stream = new StreamReader( fileName ) )

```

**Ил. 17.5.** Использование классов `File` и `Directory` (продолжение ↗)

```

39         {
40             outputTextBox.AppendText( stream.ReadToEnd() );
41         } // Конец using
42     } // Конец try
43     catch ( IOException )
44     {
45         MessageBox.Show( "Error reading from file",
46             "File Error", MessageBoxButtons.OK,
47             MessageBoxIcon.Error );
48     } // Конец catch
49 } // Конец if
50 // Проверка соответствия fileName каталогу
51 else if ( Directory.Exists( fileName ) )
52 {
53     // Получение даты создания каталога,
54     // даты изменения и т. д.
55     GetInformation( fileName );
56
57     // Получение содержимого заданного каталога
58     string[] directoryList =
59         Directory.GetDirectories( fileName );
60
61     outputTextBox.AppendText( "Directory contents:\n" );
62
63     // Вывод содержимого directoryList
64     foreach ( var directory in directoryList )
65         outputTextBox.AppendText( directory + "\n" );
66 } // Конец else if
67 else
68 {
69     // Сообщить пользователю, что файл/каталог не существует
70     MessageBox.Show( inputTextBox.Text +
71         " does not exist", "File Error",
72         MessageBoxButtons.OK, MessageBoxIcon.Error );
73 } // Конец else
74 } // Конец if
75 } // Конец метода inputTextBox_KeyDown
76
77 // Получение информации о файле или каталоге
78 // и вывод ее в outputTextBox
79 private void GetInformation( string fileName )
80 {
81     outputTextBox.Clear();
82
83     // Вывод сообщения о том, что файл/каталог существует
84     outputTextBox.AppendText( fileName + " exists\n" );
85
86     // Вывод времени создания файла или каталога
87     outputTextBox.AppendText( "Created: " +
88         File.GetCreationTime( fileName ) + "\n" );
89
90     // Вывод времени последней модификации файла или каталога
91     outputTextBox.AppendText( "Last modified: " +
92         File.GetLastWriteTime( fileName ) + "\n" );

```

**Ил. 17.5.** Использование классов File и Directory (продолжение ↗)

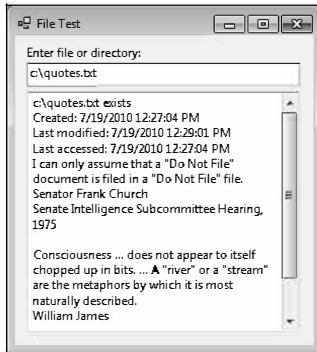


```

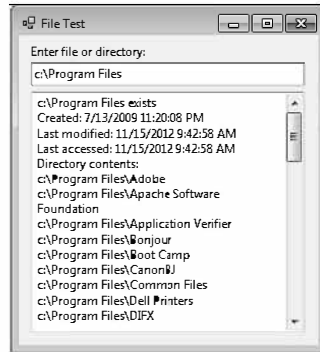
93
94     // Вывод времени последнего обращения к файлу или каталогу
95     outputTextBox.AppendText( "Last accessed: " +
96         File.GetLastAccessTime( fileName ) + "\n" );
97 } // Конец метода GetInformation
98 } // Конец класса FileTestForm
99 } // Конец пространства имен FileTest

```

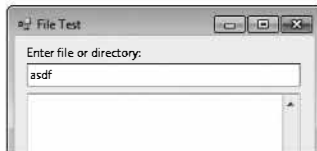
а) Просмотр содержимого файла quotes.txt



б) Просмотр всех файлов в каталоге C:\Program Files\



в) Пользователь ввел недействительное имя



г) Вывод сообщения об ошибке



**Ил. 17.5.** Использование классов File и Directory (окончание)

Если строка 28 определяет, что введенный текст не является именем файла, строка 51 проверяет, является ли он именем каталога, при помощи метода `Exists` класса `Directory`. Если пользователь указал существующий каталог, в строке 55 вызывается метод `GetInformation` для получения информации каталога. В строке 59 вызывается метод `GetDirectories` класса `Directory` для получения массива с элементами `string`, содержащими имена подкаталогов в заданном каталоге. В строках 64–65 выводятся все элементы массива. Обратите внимание: если строка 51 определяет, что введенный пользователем текст не является именем каталога, в строках 70–72 приложение сообщает (при помощи `MessageBox`), что файл или каталог с заданным именем не существует.

### Поиск в каталогах с использованием LINQ

Рассмотрим другой пример работы с файлами и каталогами. Класс `LINQToFileDirectoryForm` (ил. 17.6) использует LINQ с классами `File`, `Path` и `Directory` для определения количества файлов каждого типа в заданном каталоге. Обнаружив

файл с расширением *.bak* (то есть файл резервной копии), программа выводит окно сообщения с предложением удалить этот файл, после чего реагирует на решение пользователя. В этом примере также используется технология LINQ to Objects для удаления файлов с резервными копиями.

Когда пользователь щелкает на кнопке Search Directory, программа вызывает метод `searchButton_Click` (строки 25–65), который проводит рекурсивный поиск в каталоге, заданном пользователем. Если пользователь ввел текст, в строке 29 вызывается метод `Exists` объекта `Directory` для проверки того, соответствует ли этот текст имени каталога. Если соответствие не найдено, строки 32–33 сообщают пользователю об ошибке.

```
1  // Ил. 17.6: LINQToFileDirectoryForm.cs
2  // Использование LINQ для поиска в каталогах и определения типов файлов.
3  using System;
4  using System.Collections.Generic;
5  using System.Linq;
6  using System.Windows.Forms;
7  using System.IO;
8
9  namespace LINQToFileDirectory
10 {
11     public partial class LINQToFileDirectoryForm : Form
12     {
13         string currentDirectory; // Каталог для поиска
14
15         // Найденные расширения и количество вхождений каждого
16         Dictionary<string, int> found = new Dictionary<string, int>();
17
18         // Конструктор без параметров
19         public LINQToFileDirectoryForm()
20         {
21             InitializeComponent();
22         } // Конец конструктора
23
24         // Обработка события Click кнопки Search Directory
25         private void searchButton_Click( object sender, EventArgs e )
26         {
27             // Проверка существования заданного пути
28             if ( pathTextBox.Text != string.Empty &&
29                 !Directory.Exists( pathTextBox.Text ) )
30             {
31                 // Если каталог не существует, сообщить об ошибке
32                 MessageBox.Show( "Invalid Directory", "Error",
33                     MessageBoxButtons.OK, MessageBoxIcon.Error );
34             } // Конец if
35             else
36             {
37                 // Если каталог не задан, использовать текущий каталог.
38                 if ( pathTextBox.Text == string.Empty )
39                     currentDirectory = Directory.GetCurrentDirectory();
```

**Ил. 17.6.** Использование LINQ для поиска каталогов и определения типов файлов (продолжение ⚡)

```

40         else
41             currentDirectory = pathTextBox.Text;
42
43         directoryTextBox.Text = currentDirectory; // Вывод каталога
44
45         // Очистка текстовых полей
46         pathTextBox.Clear();
47         resultsTextBox.Clear();
48
49         SearchDirectory( currentDirectory ); // Поиск по каталогу
50
51         // Предложить пользователю удалить файлы .bak
52         CleanDirectory( currentDirectory );
53
54         // Обобщение и вывод результатов
55         foreach ( var current in found.Keys )
56         {
57             // Вывод количества файлов с текущим расширением
58             resultsTextBox.AppendText( string.Format(
59                 "* Found {0} {1} files.\r\n",
60                 found[ current ], current ) );
61         } // Конец foreach
62
63         found.Clear(); // Очистка результатов для нового поиска
64     } // Конец else
65 } // Конец метода searchButton_Click
66
67 // Поиск по каталогу средствами LINQ
68 private void SearchDirectory( string folder )
69 {
70     // Файлы, находящиеся в каталоге
71     string[] files = Directory.GetFiles( folder );
72
73     // Подкаталоги, находящиеся в каталоге
74     string[] directories = Directory.GetDirectories( folder );
75
76     // Поиск всех расширений файлов в каталоге
77     var extensions =
78         ( from file in files
79           select Path.GetExtension( file ) ).Distinct();
80
81     // Подсчет файлов с каждым расширением
82     foreach ( var extension in extensions )
83     {
84
85         // Получить количество файлов с заданным расширением
86         var extensionCount =
87             ( from file in files
88               where Path.GetExtension( file ) == extension
89               select file ).Count();
90
91
92         // Если словарь уже содержит ключ для расширения

```

**Ил. 17.6.** Использование LINQ для поиска каталогов  
и определения типов файлов (продолжение ↗)

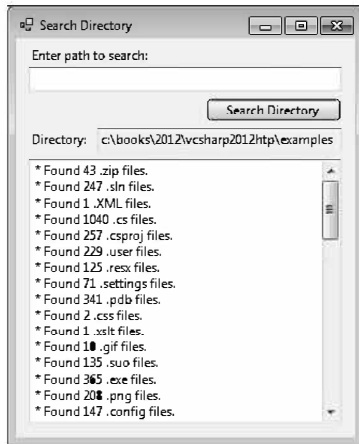
```

93         if ( found.ContainsKey( extension ) )
94             found[ extension ] += extensionCount; // Обновить счетчик
95         else
96             found.Add( extension, extensionCount ); // Новый счетчик
97     } // Конец foreach
98
99     // Рекурсивный вызов для поиска по подкаталогам
100    foreach ( var subdirectory in directories )
101        SearchDirectory( subdirectory );
102 } // Конец метода SearchDirectory
103
104 // Удаление файлов резервных копий (.bak)
105 private void CleanDirectory( string folder )
106 {
107     // Файлы, находящиеся в каталоге
108     string[] files = Directory.GetFiles( folder );
109
110     // Подкаталоги, находящиеся в каталоге
111     string[] directories = Directory.GetDirectories( folder );
112
113     // Выбор всех файлов резервных копий в каталоге
114     var backupFiles =
115         from file in files
116         where Path.GetExtension( file ) == ".bak"
117         select file;
118
119     // Перебор всех файлов резервных копий (.bak)
120     foreach ( var backup in backupFiles )
121     {
122         DialogResult result = MessageBox.Show( "Found backup file " +
123             Path.GetFileName( backup ) + ". Delete?", "Delete Backup",
124             MessageBoxButtons.YesNo, MessageBoxIcon.Question );
125
126         // Если пользователь выбрал кнопку 'yes', удалить файл
127         if ( result == DialogResult.Yes )
128         {
129             File.Delete( backup ); // Удаление файла
130             --found[ ".bak" ]; // Уменьшение счетчика в словаре
131
132             // Если файлов .bak не осталось, удалить ключ из словаря
133             if ( found[ ".bak" ] == 0 )
134                 found.Remove( ".bak" );
135         } // Конец if
136     } // Конец foreach
137
138     // Рекурсивный вызов для очистки подкаталогов
139     foreach ( var subdirectory in directories )
140         CleanDirectory( subdirectory );
141 } // Конец метода CleanDirectory
142 } // Конец класса LINQToFileDirectoryForm
143 } // Конец пространства имен LINQToFileDirectory

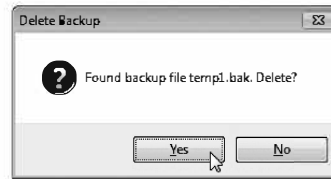
```

**Ил. 17.6.** Использование LINQ для поиска каталогов  
и определения типов файлов (продолжение ☞)

а) Приложение после входа в каталог и нажатия кнопки Search Directory



б) Диалоговое окно для подтверждения удаления файла .bak



Ил. 17.6. Использование LINQ для поиска каталогов и определения типов файлов (окончание)

### Метод SearchDirectory

В строках 38–41 загружается содержимое текущего каталога (если пользователь не ввел путь) или заданного каталога. В строке 49 имя каталога передается рекурсивному методу SearchDirectory (строки 68–102). В строке 71 вызывается метод GetFiles класса Directory для получения массива с именами файлов в заданном каталоге. Строка 74 вызывает метод GetDirectories класса Directory для получения массива с именами подкаталогов в заданном каталоге.

В строках 78–79 технология LINQ используется для получения *различающихся* расширений файлов в массиве files. Метод GetExtension класса Path получает расширение для заданного имени файла. Для каждого расширения, возвращенного запросом LINQ, в строках 82–97 определяется количество его вхождений в массиве files. Запрос LINQ в строках 87–89 сравнивает каждое расширение из массива files с текущим обрабатываемым расширением (строка 89). Все совпадения включаются в результат. Затем метод LINQ Count используется для определения общего количества файлов, соответствующих текущему расширению.

Класс LINQToFileDirectoryForm использует объект Dictionary (объявленный в строке 16) для хранения расширений и количеств имен файлов, имеющих указанное расширение. Класс Dictionary (пространство имен System.Collections.Generic) определяет *словарь* — коллекцию пар «ключ-значение», в которой каждому ключу соответствует некоторое значение. Класс Dictionary является обобщенным классом, как и класс List (см. раздел 9.4). В строке 16 указано, что словарь found содержит пары string и int, представляющие соответственно расширения и количество файлов с этими расширениями. В строке 93 метод ContainsKey класса Dictionary используется

для проверки того, было ли указанное расширение ранее включено в словарь. Если метод возвращает `true`, строка 94 прибавляет значение `extensionCount`, вычисленное в строках 88–90, к общему количеству вхождений данного расширения, хранимому в словаре. В противном случае в строке 96 метод `Add` класса `Dictionary` используется для вставки в словарь новой пары «ключ-значение» для нового расширения и его счетчика `extensionCount`. В строках 100–101 метод `SearchDirectory` рекурсивно вызывается для каждого подкаталога в текущем каталоге.

### Метод `CleanDirectory`

Когда метод `SearchDirectory` возвращает управление, в строке 52 вызывается метод `CleanDirectory` (строки 105–141) для поиска всех файлов с расширением `.bak`. Строки 108 и 111 получают список имен файлов и каталогов в текущем каталоге соответственно. Запрос LINQ в строках 115–117 ищет в текущем каталоге все файлы с расширением `.bak`. В строках 120–136 приложение перебирает результаты и предлагает пользователю подтвердить удаление каждого файла. Если пользователь нажимает кнопку `Yes` в диалоговом окне, в строке 129 метод `Delete` класса `File` удаляет файл с диска, а строка 130 уменьшает общее количество файлов `.bak` на 1. Если количество оставшихся файлов `.bak` равно 0, в строке 134 метод `Remove` класса `Dictionary` используется для удаления пары «ключ-значение» для файлов `.bak` из словаря. В строках 139–140 метод `CleanDirectory` рекурсивно вызывается для каждого подкаталога в текущем каталоге. После того как каждый подкаталог будет проверен на наличие файлов `.bak`, метод `CleanDirectory` возвращает управление, и в строках 55–61 выводится сводка расширений файлов и количества файлов с каждым расширением. В строке 55 свойство `Keys` класса `Dictionary` используется для получения всех ключей. В строке 60 индексатор `Dictionary` получает значение, соответствующее текущему ключу. Наконец, в строке 63 метод `Clear` класса `Dictionary` используется для удаления содержимого словаря.

## 17.5. Создание текстового файла с последовательным доступом

Средства работы с файлами в C# не определяют никакой структуры файлов, поэтому файлы C# должны структурироваться в соответствии с потребностями вашего приложения. В нескольких следующих примерах для реализации концепции записи будут использоваться специальные символы.

### Класс `BankUIForm`

Следующие примеры демонстрируют принципы обработки файлов в приложении, управляющем банковскими счетами. Эти программы обладают похожими пользовательскими интерфейсами, поэтому мы создали класс `BankUIForm` (ил. 17.7), инкапсулирующий графический интерфейс базового класса (см. ил. 17.7). Класс `BankUIForm` (из проекта `BankLibrary` в примерах этой главы) содержит четыре надписи и четыре текстовых поля. Методы `ClearTextboxes` (строки 28–40), `SetTextBoxValues`

(строки 43–64) и `GetTextBoxValues` (строки 67–78) соответственно стирают, задают и читают содержимое текстовых полей.

```
1 // Ил. 17.7: BankUIForm.cs
2 // Форма, используемая в примерах этой главы.
3 using System;
4 using System.Windows.Forms;
5
6 namespace BankLibrary
7 {
8     public partial class BankUIForm : Form
9     {
10         protected int TextBoxCount = 4; // Число текстовых полей на форме
11
12         // Константы перечисления задают индексы текстовых полей
13         public enum TextBoxIndices
14         {
15             ACCOUNT,
16             FIRST,
17             LAST,
18             BALANCE
19         } // Конец enum
20
21         // Конструктор без параметров
22         public BankUIForm()
23         {
24             InitializeComponent();
25         } // Конец конструктора
26
27         // Очистка всех текстовых полей
28         public void ClearTextBoxes()
29         {
30             // Перебор всех элементов управления на форме
31             foreach ( Control guiControl in Controls )
32             {
33                 // Проверяем, является ли элемент управления текстовым полем
34                 if ( guiControl is TextBox )
35                 {
36                     // Очистка текстового поля
37                     ( ( TextBox ) guiControl ).Clear();
38                 } // Конец if
39             } // Конец foreach
40         } // Конец метода ClearTextBoxes
41
42         // Заполнение текстовых полей данными из массива строк
43         public void SetTextBoxValues( string[] values )
44         {
45             // Проверка правильности длины массива
46             if ( values.Length != TextBoxCount )
47             {
48                 // Если длина неправильная, выдать исключение
49                 throw ( new ArgumentException( "There must be " +
50                     ( TextBoxCount ) + " strings in the array" ) );
51             } // Конец if
52             // Если длина прошла проверку, присвоить данные из массива
```

**Ил. 17.7.** Базовый класс графического интерфейса примеров работы с файлами (продолжение ↗)

```
53         else
54         {
55             // Заполнить текстовые поля значениями элементов массива
56             accountTextBox.Text =
57                 values[ ( int ) TextBoxIndices.ACCOUNT ];
58             firstNameTextBox.Text =
59                 values[ ( int ) TextBoxIndices.FIRST ];
60             lastNameTextBox.Text = values[ ( int ) TextBoxIndices.LAST ];
61             balanceTextBox.Text =
62                 values[ ( int ) TextBoxIndices.BALANCE ];
63         } // Конец else
64     } // Конец метода SetTextBoxValues
65
66     // Метод возвращает значения TextBox в виде массива string
67     public string[] GetTextBoxValues()
68     {
69         string[] values = new string[ TextBoxCount ];
70
71         // Копирование содержимого полей в массив string
72         values[ ( int ) TextBoxIndices.ACCOUNT ] = accountTextBox.Text;
73         values[ ( int ) TextBoxIndices.FIRST ] = firstNameTextBox.Text;
74         values[ ( int ) TextBoxIndices.LAST ] = lastNameTextBox.Text;
75         values[ ( int ) TextBoxIndices.BALANCE ] = balanceTextBox.Text;
76
77         return values;
78     } // Конец метода GetTextBoxValues
79 } // Конец класса BankUIForm
80 } // Конец пространства имен BankLibrary
```



**Ил. 17.7.** Базовый класс графического интерфейса примеров работы с файлами (окончание)

Применение визуального наследования (см. ил. 15.13) позволяет расширить этот класс для создания графического интерфейса нескольких ближайших примеров. Не забудьте, что для повторного использования класса `BankUIForm` необходимо откомпилировать его в библиотеку классов, а затем добавить ссылку на нее в каждый проект, в котором она будет использоваться. Эта библиотека включена в архив примеров кода этой главы. Возможно, вам придется заново добавить ссылки на библиотеку в примеры при копировании их в вашу систему, потому что, скорее всего, в вашей системе библиотека будет находиться в другом каталоге.



## Класс Record

На ил. 17.8 представлен класс `Record`, используемый в листингах на ил. 17.9, 17.11 и 17.12 для хранения информации в каждой записи, читаемой или записываемой в файл. Этот класс также входит в DLL-библиотеку `BankLibrary`, поэтому он находится в одном проекте с классом `BankUIForm`.

```
1 // Ил. 17.8: Record.cs
2 // Класс, представляющий запись данных.
3
4 namespace BankLibrary
5 {
6     public class Record
7     {
8         // Автоматически реализуемое свойство Account
9         public int Account { get; set; }
10
11         // Автоматически реализуемое свойство FirstName
12         public string FirstName { get; set; }
13
14         // Автоматически реализуемое свойство LastName
15         public string LastName { get; set; }
16
17         // Автоматически реализуемое свойство Balance
18         public decimal Balance { get; set; }
19
20         // Конструктор без параметров присваивает полям значения по умолчанию
21         public Record()
22             : this( 0, string.Empty, string.Empty, 0M )
23         {
24         } // Конец конструктора
25
26         // Перегруженный конструктор присваивает полям значения параметров
27         public Record( int accountValue, string firstNameValue,
28             string lastNameValue, decimal balanceValue )
29         {
30             Account = accountValue;
31             FirstName = firstNameValue;
32             LastName = lastNameValue;
33             Balance = balanceValue;
34         } // Конец класса
35     } // Конец класса Record
36 } // Конец пространства имен BankLibrary
```

**Ил. 17.8.** Класс записи в файле последовательного доступа

Класс `Record` содержит автоматически реализуемые свойства для переменных экземпляров `Account`, `FirstName`, `LastName` и `Balance` (строки 9–18), которые в совокупности представляют всю информацию записи. Конструктор без параметров (строки 21–24) задает значения этих полей, вызывая конструктор с четырьмя аргументами (строки 27–34). Последний присваивает значения полям: номеру счета присваивается 0, имени и фамилии — `string.Empty`, а балансу — `0.0M`.

### Использование потока символов для создания файла вывода

Класс `CreateFileForm` (ил. 17.9) использует экземпляры класса `Record` для создания файла последовательного доступа, который может использоваться системой учета дебиторской задолженности. Для каждого клиента программа получает номер счета, имя и фамилию клиента и баланс (то есть сумму, которую клиент задолжал компании за ранее предоставленные товары и услуги). Данные, полученные от каждого клиента, образуют запись данного клиента. В нашем приложении номер счета используется как ключ записи — данные в файлах создаются и поддерживаются в порядке счетов. Программа предполагает, что пользователь вводит записи упорядоченными по номерам счетов. Конечно, полноценная система учета должна поддерживать возможность сортировки, чтобы пользователь мог вводить записи в произвольном порядке.

```
1  // Ил. 17.9: CreateFileForm.cs
2  // Создание файла последовательного доступа
3  using System;
4  using System.Windows.Forms;
5  using System.IO;
6  using BankLibrary;
7
8  namespace CreateFile
9  {
10 public partial class CreateFileForm : BankUIForm
11 {
12     private StreamWriter fileWriter; // Для записи данных в файл
13
14     // Конструктор без параметров
15     public CreateFileForm()
16     {
17         InitializeComponent();
18     } // Конец конструктора
19
20     // Обработчик события кнопки Save
21     private void saveButton_Click( object sender, EventArgs e )
22     {
23         // Создание и вывод диалогового окна для сохранения файла
24         DialogResult result; // Результат SaveFileDialog
25         string fileName;     // Имя файла с данными
26
27         using ( SaveFileDialog fileChooser = new SaveFileDialog() )
28         {
29             fileChooser.CheckFileExists = false; // Разрешить создание
30             result = fileChooser.ShowDialog();
31             fileName = fileChooser.FileName;     // Имя сохраняемого файла
32         } // Конец using
33
34         // Убедиться, что пользователь нажал кнопку "ОК"
35         if ( result == DialogResult.OK )
36         {
37             // Если указан недействительный файл, сообщить об ошибке
38             if ( fileName == string.Empty )
```

**Ил. 17.9.** Создание файла последовательного доступа  
и вывод информации (продолжение ⇨)

```
39         MessageBox.Show( "Invalid File Name", "Error",
40             MessageBoxButtons.OK, MessageBoxIcon.Error );
41     else
42     {
43         // Если указан действительный файл, сохранить данные
44         try
45         {
46             // Открыть файл с доступом для записи
47             FileStream output = new FileStream( fileName,
48                 FileMode.OpenOrCreate, FileAccess.Write );
49
50             // Файл, в который записываются данные
51             StreamWriter writer = new StreamWriter( output );
52
53             // Заблокировать кнопку Save, разрешить доступ к Enter
54             saveButton.Enabled = false;
55             enterButton.Enabled = true;
56         } // Конец try
57         // Обработка ошибок при открытии файла
58         catch ( IOException )
59         {
60             // Оповестить пользователя, если файл не существует
61             MessageBox.Show( "Error opening file", "Error",
62                 MessageBoxButtons.OK, MessageBoxIcon.Error );
63         } // Конец catch
64     } // Конец else
65 } // Конец if
66 } // Конец метода saveButton_Click
67
68 // Обработчик события enterButton Click
69 private void enterButton_Click( object sender, EventArgs e )
70 {
71     // Сохранение данных текстовых полей в массиве
72     string[] values = GetTextBoxValues();
73
74     // Запись со значениями текстовых полей для вывода
75     Record record = new Record();
76
77     // Проверить, есть ли данные в поле с номером счета
78     if ( values[ ( int ) TextBoxIndices.ACCOUNT ] != string.Empty )
79     {
80         // Сохранение данных полей в объекте Record и вывод
81         try
82         {
83             // Получение номера счета из текстового поля
84             int accountNumber = Int32.Parse(
85                 values[ ( int ) TextBoxIndices.ACCOUNT ] );
86
87             // Проверка данных accountNumber
88             if ( accountNumber > 0 )
89             {
90                 // Сохранение данных в полях Record
91                 record.Account = accountNumber;
```

**Ил. 17.9.** Создание файла последовательного доступа  
и вывод информации (продолжение ↗)

```

92         record.FirstName = values[ ( int )
93             TextBoxIndices.FIRST ];
94         record.LastName = values[ ( int )
95             TextBoxIndices.LAST ];
96         record.Balance = Decimal.Parse(
97             values[ ( int ) TextBoxIndices.BALANCE ] );
98
99         // Запись Record в файл с разделением полей запятыми
100        fileWriter.WriteLine(
101            record.Account + "," + record.FirstName + "," +
102            record.LastName + "," + record.Balance );
103    } // Конец if
104    else
105    {
106        // Сообщить о недействительном номере счета
107        MessageBox.Show( "Invalid Account Number", "Error",
108            MessageBoxButtons.OK, MessageBoxIcon.Error );
109    } // Конец else
110 } // Конец try
111 // Сообщить об ошибке при операции вывода
112 catch ( IOException )
113 {
114     MessageBox.Show( "Error Writing to File", "Error",
115         MessageBoxButtons.OK, MessageBoxIcon.Error );
116 } // Конец catch
117 // Сообщить об ошибке в формате параметров
118 catch ( FormatException )
119 {
120     MessageBox.Show( "Invalid Format", "Error",
121         MessageBoxButtons.OK, MessageBoxIcon.Error );
122 } // Конец catch
123 } // Конец if
124
125 ClearTextBoxes(); // Очистить текстовые поля
126 } // Конец метода enterButton_Click
127
128 // Обработчик exitButton Click
129 private void exitButton_Click( object sender, EventArgs e )
130 {
131     // Проверить, существует ли файл
132     if ( fileWriter != null )
133     {
134         try
135         {
136             // Закрыть StreamWriter и используемый файл
137             fileWriter.Close();
138         } // Конец try
139         // Сообщить об ошибке при закрытии файла
140         catch ( IOException )
141         {
142             MessageBox.Show( "Cannot close file", "Error",
143                 MessageBoxButtons.OK, MessageBoxIcon.Error );
144         } // Конец catch
145     } // Конец if

```

**Ил. 17.9.** Создание файла последовательного доступа  
и вывод информации (продолжение ↗)

```

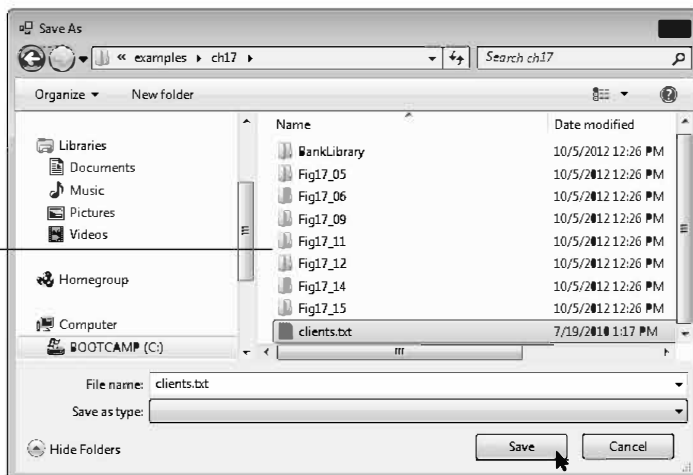
146
147         Application.Exit();
148     } // Конец метода exitButton_Click
149 } // Конец класса CreateFileForm
150 } // Конец пространства имен CreateFile

```

а) Графический интерфейс приложения с тремя дополнительными элементами управления

б) Диалоговое окно сохранения файла

Файлы и каталоги



в) Счет 100 с именем Nancy Brown и балансом -25.54

**Ил. 17.9.** Создание файла последовательного доступа и вывод информации (окончание)

Класс `CreateFileForm` либо создает, либо открывает файл (в зависимости от того, существует ли он), а затем позволяет пользователю вывести в него записи. Директива `using` в строке 6 позволяет использовать классы пространства имен `BankLibrary`; это пространство имен содержит класс `BankUIForm`, производным от которого является класс `CreateFileForm` (строка 10). Графический интерфейс класса `CreateFileForm` дополняет интерфейс `BankUIForm` кнопками `Save As`, `Enter` и `Exit`.

### Метод `saveButton_Click`

Когда пользователь щелкает на кнопке `Save As`, программа вызывает обработчик события `saveButton_Click` (строки 21–66). Строка 27 создает экземпляр объекта класса `SaveFileDialog` (пространство имен `System.Windows.Forms`). Размещение этого объекта в команде `using` (строки 27–32) гарантирует, что метод `Dispose` диалогового окна будет вызван для освобождения его ресурсов сразу же после того, как программа получит данные от пользователя. Объекты `SaveFileDialog` используются для выбора файлов (ил. 17.9, б). Строка 29 указывает, что диалоговое окно не должно проверять, существует ли файл с именем, заданным пользователем (впрочем, это поведение используется по умолчанию).

В строке 30 вызывается метод `ShowDialog` класса `SaveFileDialog` для отображения диалогового окна. Объект `SaveFileDialog` не позволяет пользователю взаимодействовать с другим окном программы до того, как пользователь закроет `SaveFileDialog` кнопкой `Save` или `Cancel` (такие диалоговые окна называются *модальными*). Пользователь выбирает диск, каталог и имя файла и нажимает кнопку `Save`. Метод `ShowDialog` возвращает значение `DialogResult`, указывающее, какая кнопка (`Save` или `Cancel`) была нажата пользователем для закрытия диалогового окна. Возвращаемое значение присваивается переменной `result` (строка 30). В строке 31 имя файла из диалогового окна сохраняется в переменной. Строка 35 проверяет, была ли нажата кнопка `OK`; для этого значение `result` сравнивается с `DialogResult.OK`. Если значения равны, метод `saveButton_Click` продолжает работу.

Один из способов открытия файла для работы с текстом основан на создании объекта класса `FileStream`; это происходит в строках 47–48 нашего примера. Используемый конструктор `FileStream` получает три аргумента — строку с путем и именем открываемого файла; константу, описывающую режим открытия файла; и константу, описывающую разрешения доступа к файлу. Константа `FileMode.OpenOrCreate` (строка 48) означает, что объект `FileStream` должен открыть файл, если он существует, или создать файл, если он не существует. Обратите внимание: объект `StreamWriter` перезаписывает содержимое существующего файла. Если вы хотите сохранить исходное содержимое файла, используйте режим `FileMode.Append`. Существуют и другие константы `FileMode`, описывающие режим открытия файлов; они будут кратко описаны при использовании в примерах.

Константа `FileAccess.Write` указывает, что программа может выполнять с объектом `FileStream` только операции записи. В третьем параметре конструктора также могут передаваться две другие константы — `FileAccess.Read` (доступ только для чтения) и `FileAccess.ReadWrite` (доступ для чтения/записи). Строка 58

перехватывает исключение `IOException`, возникающее при ошибке открытия файла или создания `StreamWriter`; обработка исключения сводится к выдаче сообщения об ошибке (строки 61–62). Если выполнение прошло без исключений, файл открыт для записи.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 17.1

При открытии файлов используйте значения из перечисления `FileAccess` для управления доступом к этим файлам.



### ТИПИЧНАЯ ОШИБКА 17.1

Если вы забудете открыть файл, прежде чем использовать его в программе, это приведет к логической ошибке.

## Метод `enterButton_Click`

Заполнив текстовые поля, пользователь щелкает на кнопке `Enter`; при этом вызывается обработчик `enterButton_Click` (строки 69–126), который должен сохранить введенные данные в файл, заданном пользователем. Если пользователь ввел действительный номер счета (целое число больше нуля), строки 91–97 сохраняют значения текстовых полей в объекте типа `Record` (созданном в строке 75). Если пользователь ввел в одном из текстовых полей недействительные данные (например, нецифровые символы в поле баланса), программа выдает исключение `FormatException`. Блок `catch` в строках 118–122 обрабатывает такие исключения, сообщая пользователю об ошибке форматирования (при помощи `MessageBox`).

Если пользователь ввел действительные данные, в строках 100–102 сформированная запись выводится в файл вызовом метода `WriteLine` объекта `StreamWriter`, созданного в строке 51. Метод `WriteLine` выводит в файл последовательность символов. Объект `StreamWriter` конструируется с аргументом `FileStream`, определяющим файл, в который `StreamWriter` будет выводить текст. Класс `StreamWriter` (как и большинство классов, обсуждаемых в этой главе) принадлежит пространству имен `System.IO`.

## Метод `exitButton_Click`

При щелчке на кнопке `Exit` выполняется метод `exitButton_Click` (строки 129–148). Строка 137 закрывает объект `StreamWriter`, который автоматически закрывает `FileStream`, после чего строка 147 завершает программу. Обратите внимание: метод `Close` вызывается в блоке `try`. Если файл или поток не удастся нормально закрыть, метод `Close` выдает исключение `IOException`. В этом случае важно оповестить пользователя о том, что информация в файле или потоке может быть повреждена.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 17.1

Явно закрывайте все файлы, с которыми программа завершила работу. Это снизит затраты ресурсов в программах, которые продолжают долго работать после того, как они завершат работу с конкретным файлом. Практика явного закрытия файлов также делает программный код более понятным.



## ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 17.2

Освобождение неиспользуемых ресурсов делает их доступными для использования в других программах. Тем самым вы повышаете эффективность использования ресурсов.

### Тестовые данные

Для тестирования программы использовались тестовые данные, представленные на ил. 17.10. Способ хранения записей данных в файле в программе не отражен. Чтобы убедиться в том, что файл был создан успешно, в следующем разделе мы создадим программу для чтения и вывода файла. Так как данные хранятся в текстовом формате, содержимое файла можно просмотреть в любом текстовом редакторе.

Номер счета	Имя	Фамилия	Баланс
100	Nancy	Brown	-25.54
200	Stacey	Dunn	314.33
300	Doug	Barker	0.00
400	Dave	Smith	258.34
500	Sam	Stone	34.98

**Ил. 17.10.** Тестовые данные для программы на илл. 17.9

## 17.6. Чтение данных из текстового файла с последовательным доступом

В предыдущем разделе было показано, как создать файл в приложении с последовательным доступом к данным. В этом разделе рассматривается операция последовательного чтения данных из файла.

Класс `ReadSequentialAccessFileForm` (ил. 17.11) читает записи из файла, созданного в программе на ил. 17.9, и выводит содержимое каждой записи. Код в целом похож на код класса на ил. 17.9, поэтому мы рассмотрим только специфические особенности этого приложения.

```
1 // Ил. 17.11: ReadSequentialAccessFileForm.cs
2 // Чтение файла с последовательным доступом.
3 using System;
4 using System.Windows.Forms;
5 using System.IO;
6 using BankLibrary;
7
8 namespace ReadSequentialAccessFile
9 {
10     public partial class ReadSequentialAccessFileForm : BankUIForm
```

**Ил. 17.11.** Чтение файла с последовательным доступом (продолжение ↗)



```
11 {
12     private StreamReader fileReader; // Для чтения из текстового файла
13
14     // Конструктор без параметров
15     public ReadSequentialAccessFileForm()
16     {
17         InitializeComponent();
18     } // Конец конструктора
19
20     // Вызывается при нажатии кнопки Open
21     private void openButton_Click( object sender, EventArgs e )
22     {
23         // Создание и вывод диалогового окна открытия файла
24         DialogResult result; // Результат OpenFileDialog
25         string fileName;     // Имя файла с данными
26
27         using ( OpenFileDialog fileChooser = new OpenFileDialog() )
28         {
29             result = fileChooser.ShowDialog();
30             fileName = fileChooser.FileName; // Получение введенного имени
31         } // Конец using
32
33         // Проверяем, что пользователь нажал кнопку "OK"
34         if ( result == DialogResult.OK )
35         {
36             ClearTextBoxes();
37
38             // Если задан недействительный файл, сообщить об ошибке
39             if ( fileName == string.Empty )
40                 MessageBox.Show( "Invalid File Name", "Error",
41                                 MessageBoxButtons.OK, MessageBoxIcon.Error );
42             else
43             {
44                 try
45                 {
46                     // Создание объекта FileStream для чтения файла
47                     FileStream input = new FileStream(
48                         fileName, FileMode.Open, FileAccess.Read );
49
50                     // Задание файла, из которого читаются данные
51                     fileReader = new StreamReader( input );
52
53                     openButton.Enabled = false; // Кнопка Open File блокируется
54                     nextButton.Enabled = true;  // Кнопка Next Record доступна
55                 } // Конец try
56                 catch ( IOException )
57                 {
58                     MessageBox.Show( "Error reading from file",
59                                     "File Error", MessageBoxButtons.OK,
60                                     MessageBoxIcon.Error );
61                 } // Конец catch
62             } // Конец else
63         } // Конец if
64     } // Конец метода openButton_Click
```

**Ил. 17.11.** Чтение файла с последовательным доступом (продолжение ➤)

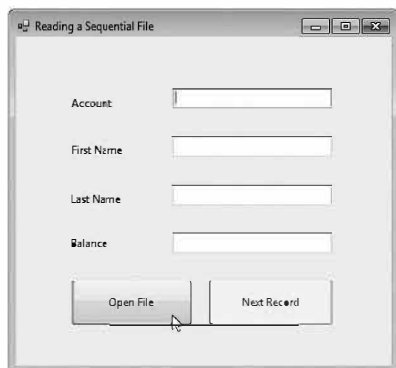
```

65
66 // Вызывается при нажатии кнопки Next
67 private void nextButton_Click( object sender, EventArgs e )
68 {
69     try
70     {
71         // Получение следующей записи из файла
72         string inputRecord = fileReader.ReadLine();
73         string[] inputFields; // Для хранения отдельных блоков данных
74
75         if ( inputRecord != null )
76         {
77             inputFields = inputRecord.Split( ',' );
78
79             Record record = new Record(
80                 Convert.ToInt32( inputFields[ 0 ] ), inputFields[ 1 ],
81                 inputFields[ 2 ],
82                 Convert.ToDecimal( inputFields[ 3 ] ) );
83
84             // Копирование значений из массива в текстовые поля
85             SetTextBoxValues( inputFields );
86         } // Конец if
87         else
88         {
89             // Закрытие объекта StreamReader и используемого файла
90             fileReader.Close();
91             openButton.Enabled = true; // Кнопка Open File доступна
92             nextButton.Enabled = false; // Кнопка Next Record блокируется
93             ClearTextBoxes();
94
95             // Сообщить пользователю об отсутствии записей в файле
96             MessageBox.Show( "No more records in file", string.Empty,
97                 MessageBoxButtons.OK, MessageBoxIcon.Information );
98         } // Конец else
99     } // Конец try
100     catch ( IOException )
101     {
102         MessageBox.Show( "Error Reading from File", "Error",
103             MessageBoxButtons.OK, MessageBoxIcon.Error );
104     } // Конец catch
105 } // Конец метода nextButton_Click
106 } // Конец класса ReadSequentialAccessFileForm
107 } // Конец пространства имен ReadSequentialAccessFile

```

**Ил. 17.11.** Чтение файла с последовательным доступом (продолжение ↗)

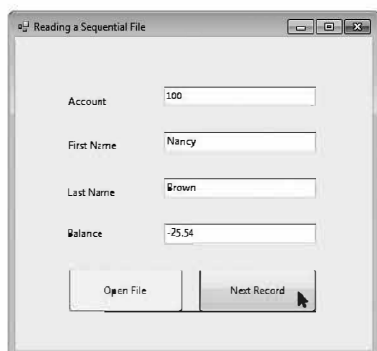
а) Графический интерфейс BankUI с кнопкой Open File



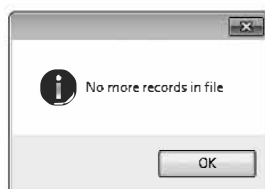
б) Окно OpenFileDialog



в) Чтение данных счета 100



г) Когда все записи будут прочитаны, приложение выводит сообщение для пользователя



**Ил. 17.11.** Чтение файла с последовательным доступом (окончание)

### Метод `openButton_Click`

Когда пользователь щелкает на кнопке Open File, программа вызывает обработчик `openButton_Click` (строки 21–64). В строке 27 создается объект `OpenFileDialog`, а в строке 29 его метод `ShowDialog` вызывается для отображения диалогового окна (ил. 17.11, б). Поведение и графический интерфейс диалоговых окон сохранения и открытия файлов идентичен, отличается только заголовок. Если пользователь выбирает действительное имя файла, в строках 47–48 создается объект `FileStream`, который присваивается ссылке `input`. Мы передаем константу `FileMode.Open` во втором аргументе конструктора `FileStream`, чтобы указать, что объект `FileStream` должен открывать файл, если тот существует, или выдавать исключение `FileNotFoundException` для несуществующего файла. (В нашем примере конструктор `FileStream` не выдает исключение `FileNotFoundException`, потому что окно `OpenFileDialog` настроено на проверку существования файла.) В предыдущем примере (см. ил. 17.9) для вывода текста в файл использовался объект `FileStream` с доступом только для чтения.

В текущем примере (см. ил. 17.11) мы ограничиваемся доступом только для чтения, для чего в третьем аргументе конструктора `FileStream` передается константа `FileAccess.Read`. Объект `FileStream` используется для создания `StreamReader` в строке 51. Объект `FileStream` задает файл, из которого объект `StreamReader` будет читать текст.



### КАК ИЗБЕЖАТЬ ОШИБОК 17.1

Если содержимое файла не должно изменяться, открывайте файл в режиме `FileAccess.Read`; тем самым вы предотвратите случайные модификации.

### Метод `nextButton_Click`

Когда пользователь щелкает на кнопке `Next Record`, программа вызывает обработчик события `nextButton_Click` (строки 67–105), который читает следующую запись из файла, указанного пользователем. (Для просмотра первой записи пользователь должен щелкнуть на кнопке `Next Record` после открытия файла.) В строке 72 вызывается метод `ReadLine` объекта `StreamReader` для чтения следующей записи. Если во время чтения происходит ошибка, выдается исключение `IOException` (перехватываемое в строке 100), а приложение сообщает о проблеме пользователю (строки 102–103). В противном случае строка 75 определяет, вернул ли метод `ReadLine` значение `null` (то есть в файле не осталось больше текста). Если данные были прочитаны, в строке 77 используется метод `Split` класса `string` для разбиения прочитанного потока символов на блоки, представляющие свойства `Record`, — второй аргумент указывает, что блоки ограничиваются запятыми в файле. Для сохранения этих свойств вызывается конструктор объекта `Record`, в аргументах которого передаются значения свойств (строки 79–82). В строке 85 элементы управления `TextBox` заполняются значениями `Record`. Если `ReadLine` вернет `null`, программа закрывает объект `StreamReader` (строка 90), что приводит к автоматическому закрытию объекта `FileStream`, а затем сообщает пользователю о том, что записей не осталось (строки 96–97).

## 17.7. Запрос информации о счетах

Чтобы выполнить последовательное чтение данных из файла, программы обычно начинают читать с начала файла и двигаются вперед, пока не будут обнаружены нужные данные. Иногда в процессе выполнения программы файл требуется последовательно просмотреть несколько раз (начиная с начала файла). Объект `FileStream` может переместить свой указатель текущей позиции (который содержит номер следующего байта, читаемого или записываемого в файл) в любую позицию в файле. При открытии объекта `FileStream` его указатель текущей позиции устанавливается в позицию 0 (то есть в начало файла).

Следующая программа основана на концепциях, представленных на ил. 17.11. Класс `CreditInquiryForm` (ил. 17.12) позволяет кредитному менеджеру провести поиск и вывести информацию по счетам клиентов с кредитовым балансом (то есть клиентов, которым компания задолжала), нулевым балансом (то есть клиентов,

которые не должны компании) и дебетовым балансом (то есть клиентов, которые задолжали компании за ранее полученные товары и услуги). Для вывода информации в программе будет использоваться элемент управления `RichTextBox`. Эти элементы управления по своим возможностям превосходят обычные текстовые поля — например, они предоставляют метод `Find` для поиска строк и метод `LoadFile` для вывода содержимого файла. Оба класса, `RichTextBox` и `TextBox`, являются производными от абстрактного класса `System.Windows.Forms.TextBoxBase`. В данном случае мы выбрали элемент управления `RichTextBox`, потому что он по умолчанию выводит многострочный текст, тогда как обычный элемент управления `TextBox` выводит текст в одну строку. Также можно было переключить `TextBox` в многострочный режим, задав свойству `Multiline` значение `true`.

Кнопки на форме программы позволяют кредитному менеджеру выбрать нужную информацию. Кнопка `Open File` открывает файл с данными. Кнопка `Credit Balances` выводит список счетов с положительным (кредитовым) балансом, кнопка `Debit Balances` выводит список счетов с дебетовым балансом, а кнопка `Zero Balances` выводит список счетов с нулевым балансом. Кнопка `Done` завершает работу приложения.

```

1 // Ил. 17.12: CreditInquiryForm.cs
2 // Последовательное чтение файла и вывод данных в зависимости
3 // от типа счета ( кредитовый, дебетовый или нулевой баланс ).
4 using System;
5 using System.Windows.Forms;
6 using System.IO;
7 using BankLibrary;
8
9 namespace CreditInquiry
10 {
11     public partial class CreditInquiryForm : Form
12     {
13         private FileStream input; // Обеспечивает связь с файлом
14         private StreamReader fileReader; // Для чтения из текстового файла
15
16         // Имя файла с данными кредитовых, дебетовых и нулевых балансов
17         private string fileName;
18
19         // Конструктор без параметров
20         public CreditInquiryForm()
21         {
22             InitializeComponent();
23         } // Конец конструктора
24
25         // Вызывается при нажатии кнопки Open File
26         private void openButton_Click( object sender, EventArgs e )
27         {
28             // Создание диалогового окна открытия файла
29             DialogResult result;
30
31             using ( OpenFileDialog fileChooser = new OpenFileDialog() )

```

**Ил. 17.12.** Программа для получения информации о счетах (продолжение ↗)

```

32     {
33         result = fileChooser.ShowDialog();
34         fileName = fileChooser.FileName;
35     } // Конец using
36
37     // Выйти из обработчика, если нажата кнопка Cancel
38     if ( result == DialogResult.OK )
39     {
40         // Если задан недействительный файл, сообщить об ошибке
41         if ( fileName == string.Empty )
42             MessageBox.Show( "Invalid File Name", "Error",
43                             MessageBoxButtons.OK, MessageBoxIcon.Error );
44         else
45         {
46             // Создание объекта FileStream для чтения файла
47             input = new FileStream( fileName,
48                                   FileMode.Open, FileAccess.Read );
49
50             // Задание файла, из которого читаются данные
51             fileReader = new StreamReader( input );
52
53             // Доступны все кнопки, кроме Open File
54             openButton.Enabled = false;
55             creditButton.Enabled = true;
56             debitButton.Enabled = true;
57             zeroButton.Enabled = true;
58         } // Конец else
59     } // Конец if
60 } // Конец метода openButton_Click
61
62 // Вызывается при нажатии кнопки Credit Balances,
63 // Debit Balances или Zero Balances
64 private void getBalances_Click( object sender, System.EventArgs e )
65 {
66     // Явное преобразование sender к типу кнопки
67     Button senderButton = ( Button ) sender;
68
69     // Получение текста с информацией о типе счетов
70     string accountType = senderButton.Text;
71
72     // Чтение и вывод информации из файла
73     try
74     {
75         // Возврат к началу файла
76         input.Seek( 0, SeekOrigin.Begin );
77
78         displayTextBox.Text = "The accounts are:\r\n";
79
80         // Перемещение до конца файла
81         while ( true )
82         {
83             string[] inputFields; // Для отдельных полей данных
84             Record record;        // Для хранения записи при чтении
85             decimal balance;      // Для хранения баланса каждой записи

```

**Ил. 17.12.** Программа для получения информации о счетах (продолжение ↗)

```

86
87 // Получение следующей записи из файла
88 string inputRecord = fileReader.ReadLine();
89
90 // В конце файла выйти из метода
91 if ( inputRecord == null )
92     return;
93
94 inputFields = inputRecord.Split( ',' ); // Разбор данных
95
96 // Создание объекта Record по входным данным
97 record = new Record(
98     Convert.ToInt32( inputFields[ 0 ] ), inputFields[ 1 ],
99     inputFields[ 2 ], Convert.ToDecimal(inputFields[ 3 ]));
100
101 // Сохранение последнего поля записи в переменной balance
102 balance = record.Balance;
103
104 // Проверяем, нужно ли выводить баланс
105 if ( ShouldDisplay( balance, accountType ) )
106 {
107     // Вывод записи
108     string output = record.Account + "\t" +
109         record.FirstName + "\t" + record.LastName + "\t";
110
111     // Вывод баланса в формате денежной суммы
112     output += String.Format( "{0:F}", balance ) + "\r\n";
113
114     // Копирование output для вывода на экране
115     displayTextBox.AppendText( output );
116 } // Конец if
117 } // Конец while
118 } // Конец try
119 // Обработка исключений при ошибке чтения файла
120 catch ( IOException )
121 {
122     MessageBox.Show( "Cannot Read File", "Error",
123         MessageBoxButtons.OK, MessageBoxIcon.Error );
124 } // Конец catch
125 } // Конец метода getBalances_Click
126
127 // Метод проверяет, нужно ли выводить заданную запись
128 private bool ShouldDisplay( decimal balance, string accountType )
129 {
130     if ( balance > 0M )
131     {
132         // Вывод кредитовых балансов
133         if ( accountType == "Credit Balances" )
134             return true;
135     } // Конец if
136     else if ( balance < 0M )
137     {
138         // Вывод дебетовых балансов
139         if ( accountType == "Debit Balances" )

```

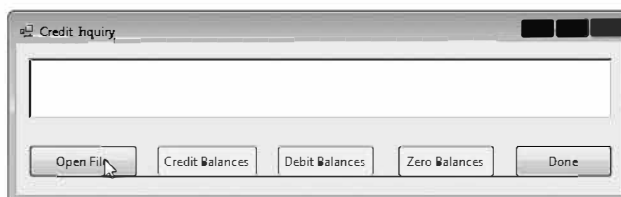
**Ил. 17.12.** Программа для получения информации о счетах (продолжение ↗)

```

140         return true;
141     } // Конец else if
142     else // balance == 0
143     {
144         // Вывод нулевых балансов
145         if ( accountType == "Zero Balances" )
146             return true;
147     } // Конец else
148
149     return false;
150 } // Конец метода ShouldDisplay
151
152 // Обработчик нажатия кнопки Done
153 private void doneButton_Click( object sender, EventArgs e )
154 {
155     if ( input != null )
156     {
157         // Закрытие файла и StreamReader
158         try
159         {
160             // Закрыть StreamReader и используемый файл
161             fileReader.Close();
162         } // Конец try
163         // Обработать исключение, если FileStream не существует
164         catch ( IOException )
165         {
166             // Сообщить об ошибке при закрытии файла
167             MessageBox.Show( "Cannot close file", "Error",
168                             MessageBoxButtons.OK, MessageBoxIcon.Error );
169         } // Конец catch
170     } // Конец if
171
172     Application.Exit();
173 } // Конец метода doneButton_Click
174 } // Конец класса CreditInquiryForm
175 } // Конец пространства имен CreditInquiry

```

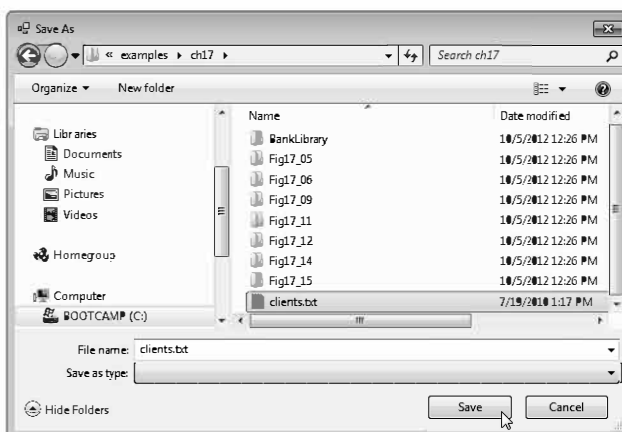
а) Форма приложения  
при запуске



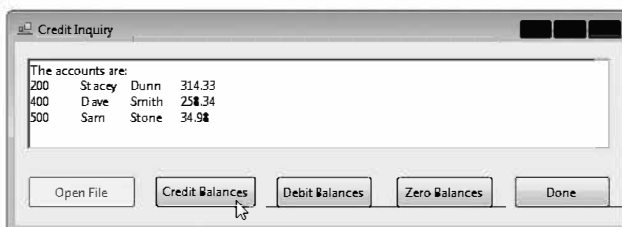
**Ил. 17.12.** Программа для получения информации о счетах (продолжение ↗)



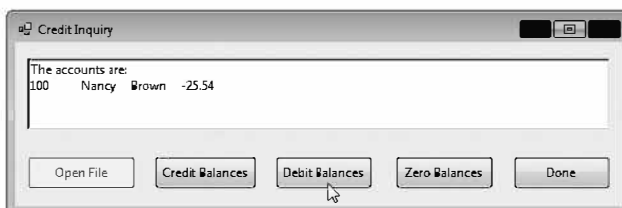
## 6) Открытие файла clients.txt



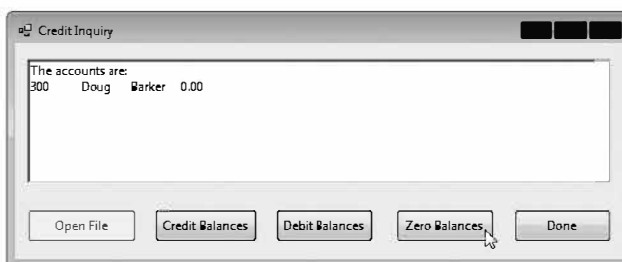
## в) Вывод счетов с кредитовыми балансами



## г) Вывод счетов с дебетовыми балансами



## д) Вывод счетов с нулевыми балансами

**Ил. 17.12.** Программа для получения информации о счетах (окончание)

Когда пользователь щелкает на кнопке **Open File**, программа вызывает обработчик события `openButton_Click` (строки 26–60). В строке 31 создается объект `OpenFileDialog`, а в строке 33 вызывается его метод `ShowDialog` для отображения диалогового окна, в котором пользователь выбирает открываемый файл. В строках 47–48 создается объект `FileStream` с доступом только для чтения; он присваивается ссылке

input. В строке 51 создается объект `StreamReader`, который используется для чтения текста из `FileStream`.

Когда пользователь щелкает на кнопке `Credit Balances`, `Debit Balances` или `Zero Balances`, программа вызывает метод `getBalances_Click` (строки 64–125). В строке 67 параметр `sender` (ссылка object на элемент управления, сгенерировавший событие) преобразуется в объект `Button`. В строке 70 извлекается текст кнопки `Button`, по которому программа определяет тип выводимых счетов. В строке 76 метод `Seek` класса `FileStream` возвращает указатель текущей позиции в начало файла. Метод `Seek` позволяет задать новую позицию смещением от начала файла, его конца или текущей позиции. Часть файла, от которой отсчитывается смещение, выбирается константами перечисления `SeekOrigin`. В нашем примере поток смещается на 0 байт от начала файла (`SeekOrigin.Begin`). В строках 81–117 определяется цикл `while`, который использует закрытый метод `ShouldDisplay` (строки 128–150) для проверки, нужно ли выводить каждую запись в файл. Цикл `while` получает данные, многократно вызывая метод `ReadLine` класса `StreamReader` (строка 88) и разбивая текст на блоки (строка 94), которые используются для инициализации объекта `record` (строки 97–99). Строка 91 определяет, достиг ли указатель текущей позиции конца файла; в этом случае вызов `ReadLine` возвращает `null`. Когда это произойдет, программа вернет управление из метода `getBalances_Click` (строка 92).

## 17.8. Сериализация

В разделе 17.5 было показано, как записать отдельные поля объекта `Record` в текстовый файл, а в разделе 17.6 вы увидели, как прочитать эти поля из файла и поместить их значения в объект `Record` в памяти. В приведенных примерах информация одной записи была объединена в объекте `Record`. При записи переменных экземпляров `Record` в файл на диске терялась часть информации — например, тип каждого значения. Так, при чтении значения "3" из файла программа не могла определить, было ли это значение сохранено из переменной типа `int`, `string` или `decimal`. На диске хранятся только данные, но не информация типов. Если бы программа, читающая данные, «знала», какому типу эти данные соответствуют, она могла бы прочитать их прямо в объект указанного типа. Например, в коде с ил. 17.11 мы знали, что программа читает значение `int` (номер счета), за которым следуют два значения `string` (имя и фамилия) и `decimal` (баланс). Мы также знаем, что эти значения разделены запятыми, а каждая логическая строка в файле содержит только одну запись. Это позволяло нам разобрать прочитанные данные и преобразовать номер счета в значение `int`, а баланс — в значение `decimal`. Иногда бывает проще и удобнее читать и записывать в файл целые объекты. В C# для таких задач существует механизм *сериализации*. Сериализованный объект представляется последовательностью байтов, которая содержит данные объекта, а также информацию о типе объекта и типе данных, хранящихся в объекте. После того как сериализованный объект будет записан в файл, его можно прочитать из этого файла и *десериализовать* — то есть воссоздать объект в памяти по информации о типе и байтам, представляющим объект и его данные.

Класс `BinaryFormatter` (пространство имен `System.Runtime.Serialization.Formatters.Binary`) предоставляет средства для чтения и записи целых объектов в поток. Метод `Serialize` класса `BinaryFormatter` записывает представление объекта в файл. Метод `Deserialize` класса `BinaryFormatter` читает это представление из файла и воссоздает исходный объект. Оба метода выдают исключение `SerializationException` при возникновении ошибки в ходе сериализации или десериализации. Обоим методам в параметре должен передаваться объект `Stream` (например, `FileStream`), чтобы объект `BinaryFormatter` мог обратиться к правильному потоку.

В разделах 17.9–17.10 мы займемся созданием и обработкой файлов последовательного доступа с использованием сериализации объектов. Сериализация выполняется на основе байтовых потоков, поэтому последовательные файлы, с которыми мы будем работать, будут *двоичными файлами*. Содержимое таких файлов непонятно рядовому пользователю, поэтому мы напомним отдельное приложение для чтения и вывода сериализованных объектов.

## 17.9. Создание файла последовательного доступа с использованием сериализации

Начнем с создания и записи сериализованных объектов в файл последовательного доступа. В этом разделе используется большая часть кода из раздела 17.5, поэтому мы сосредоточимся на новых возможностях.

### Определение класса `RecordSerializable`

Изменим класс `Record` (см. ил. 17.8), чтобы объекты этого класса могли сериализоваться. Класс `RecordSerializable` (из проекта `BankLibrary` — ил. 17.13) помечен атрибутом `[Serializable]` (строка 7), который сообщает CLR, что объекты `RecordSerializable` могут участвовать в сериализации. Классы, представляющие сериализуемые типы, должны либо включать этот атрибут в свои объявления, либо реализовать интерфейс `ISerializable`.

```
1 // Ил. 17.13: RecordSerializable.cs
2 // Сериализуемый класс, представляющий запись данных.
3 using System;
4
5 namespace BankLibrary
6 {
7     [Serializable]
8     public class RecordSerializable
9     {
10         // Автоматически реализуемое свойство Account
11         public int Account { get; set; }
12
13         // Автоматически реализуемое свойство FirstName
14         public string FirstName { get; set; }
```

**Ил. 17.13.** Класс `RecordSerializable` (продолжение ↗)

```
15
16 // Автоматически реализуемое свойство LastName
17 public string LastName { get; set; }
18
19 // Автоматически реализуемое свойство Balance
20 public decimal Balance { get; set; }
21
22 // Конструктор по умолчанию заполняет поля значениями по умолчанию
23 public RecordSerializable()
24     : this( 0, string.Empty, string.Empty, 0M )
25 {
26     } // Конец конструктора
27
28 // Перегруженный конструктор заполняет поля значениями параметров
29 public RecordSerializable( int accountValue, string firstNameValue,
30     string lastNameValue, decimal balanceValue )
31 {
32     Account = accountValue;
33     FirstName = firstNameValue;
34     LastName = lastNameValue;
35     Balance = balanceValue;
36 } // Конец конструктора
37 } // Конец класса RecordSerializable
38 } // Конец пространства имен BankLibrary
```

#### Ил. 17.13. Класс RecordSerializable (окончание)

В классе, помеченном атрибутом [Serializable] или реализующем интерфейс ISerializable, каждая переменная экземпляра тоже должна поддерживать сериализацию. Все переменные простых типов и string поддерживают сериализацию. Для переменных ссылочных типов возможность сериализации проверяется по объявлению класса (и возможно, его базовых классов). По умолчанию массивы поддерживают сериализацию, но если массив содержит ссылки на другие объекты, такие объекты могут поддерживать или не поддерживать сериализацию.

#### Использование файла SerializationStreamtoCreateanOutput

На следующем шаге мы применим сериализацию создания файла с последовательным доступом (ил. 17.14). Для тестирования этой программы будут использоваться данные из таблицы на ил. 17.10. Так как снимки экрана не отличаются от приведенных на ил. 17.9, они здесь не приводятся. В строке 15 создается объект BinaryFormatter для записи сериализованных объектов. В строках 53–54 открывается объект FileStream, в который программа записывает сериализованные объекты. Аргумент string, передаваемый конструктору FileStream, представляет имя открываемого файла и путь к нему. Он определяет файл, в который будут записываться сериализованные объекты.

Программа предполагает, что данные введены правильно и по порядку номеров записей. Обработчик события enterButton\_Click (строки 72–127) выполняет операцию вывода. В строке 78 создается объект RecordSerializable, который заполняется данными в строках 94–100. В строке 103 метод Serialize вызывается для вывода объекта RecordSerializable в выходной файл. Метод Serialize получает объект

`FileStream` в первом аргументе, чтобы объект `BinaryFormatter` мог записать свой второй аргумент в правильный файл. Для записи всего объекта достаточно одной команды. Если во время сериализации происходит ошибка, возникает исключение `SerializationException` — оно перехватывается в строках 113–117.

В примере запуска программы на ил. 17.14 вводится информация о пяти счетах — та же, которая была приведена на ил. 17.10. Программа не показывает, как данные выглядят в файле. Не забывайте, что мы используем двоичные файлы, которые не могут напрямую читаться пользователем. Чтобы убедиться в том, что файл был создан успешно, в следующем разделе будет приведена программа для чтения содержимого файла.

```
1  // Ил. 17.14: CreateFileForm.cs
2  // Создание файла с использованием сериализации.
3  using System;
4  using System.Windows.Forms;
5  using System.IO;
6  using System.Runtime.Serialization.Formatters.Binary;
7  using System.Runtime.Serialization;
8  using BankLibrary;
9
10 namespace CreateFile
11 {
12     public partial class CreateFileForm : BankUIForm
13     {
14         // Объект сериализации RecordSerializables в двоичном формате
15         private BinaryFormatter formatter = new BinaryFormatter();
16         private FileStream output; // Поток для записи в файл
17
18         // Конструктор без параметров
19         public CreateFileForm()
20         {
21             InitializeComponent();
22         } // Конец конструктора
23
24         // Обработчик saveButton_Click
25         private void saveButton_Click( object sender, EventArgs e )
26         {
27             // Создание и вывод диалогового окна для сохранения файла
28             DialogResult result;
29             string fileName; // Имя файла для сохранения данных
30
31             using ( SaveFileDialog fileChooser = new SaveFileDialog() )
32             {
33                 fileChooser.CheckFileExists = false; // Разрешить создание
34
35                 // Получение результата диалогового окна
36                 result = fileChooser.ShowDialog();
37                 fileName = fileChooser.FileName; // Имя сохраняемого файла
38             } // Конец using
39
40             // Убедиться, что пользователь нажал кнопку "ОК"
41             if ( result == DialogResult.OK )
```

**Ил. 17.14.** Создание файла посредством сериализации (продолжение ↗)

```

42         {
43             // Если указан недействительный файл, сообщить об ошибке
44             if ( fileName == string.Empty )
45                 MessageBox.Show( "Invalid File Name", "Error",
46                                 MessageBoxButtons.OK, MessageBoxIcon.Error );
47         }
48         else
49         {
50             // Если указан действительный файл, сохранить данные
51             try
52             {
53                 // Открыть файл с доступом для записи
54                 output = new FileStream( fileName,
55                                         FileMode.OpenOrCreate, FileAccess.Write );
56
57                 // Заблокировать кнопку Save, разрешить доступ к Enter
58                 saveButton.Enabled = false;
59                 enterButton.Enabled = true;
60             } // Конец try
61             // Обработка ошибок при открытии файла
62             catch ( IOException )
63             {
64                 // Сообщить об ошибке открытия файла
65                 MessageBox.Show( "Error opening file", "Error",
66                                 MessageBoxButtons.OK, MessageBoxIcon.Error );
67             } // Конец catch
68         } // Конец else
69     } // Конец if
70 } // Конец метода saveButton_Click
71
72 // Обработчик события enterButton Click
73 private void enterButton_Click( object sender, EventArgs e )
74 {
75     // Сохранение данных текстовых полей в массиве
76     string[] values = GetTextBoxValues();
77
78     // Объект RecordSerializable со значениями текстовых полей
79     RecordSerializable record = new RecordSerializable();
80
81     // Проверить, есть ли данные в поле с номером счета
82     if ( values[ ( int ) TextBoxIndices.ACCOUNT ] != string.Empty )
83     {
84         // Сохранение данных в объекте RecordSerializable и вывод
85         try
86         {
87             // Получение номера счета из текстового поля
88             int accountNumber = Int32.Parse(
89                 values[ ( int ) TextBoxIndices.ACCOUNT ] );
90
91             // Проверка данных accountNumber
92             if ( accountNumber > 0 )
93             {
94                 // Сохранение данных в полях RecordSerializable
95                 record.Account = accountNumber;
96                 record.FirstName = values[ ( int )
97                     TextBoxIndices.FIRST ];
98                 record.LastName = values[ ( int )

```

**Ил. 17.14.** Создание файла посредством сериализации (продолжение )

```
198         TextBoxIndices.LAST ];
199         record.Balance = Decimal.Parse( values[
200             ( int ) TextBoxIndices.BALANCE ] );
201
202         // Вывод в FileStream (сериализация объекта)
203         formatter.Serialize( output, record );
204     } // Конец if
205     else
206     {
207         // Сообщить о недействительном номере счета
208         MessageBox.Show( "Invalid Account Number", "Error",
209             MessageBoxButtons.OK, MessageBoxIcon.Error );
210     } // Конец else
211 } // Конец try
212 // Сообщить об ошибке сериализации
213 catch ( SerializationException )
214 {
215     MessageBox.Show( "Error Writing to File", "Error",
216         MessageBoxButtons.OK, MessageBoxIcon.Error );
217 } // end catch
218 // Сообщить об ошибке в формате параметров
219 catch ( FormatException )
220 {
221     MessageBox.Show( "Invalid Format", "Error",
222         MessageBoxButtons.OK, MessageBoxIcon.Error );
223 } // Конец catch
224 } // Конец if
225
226 ClearTextBoxes(); // Очистить текстовые поля
227 } // Конец метода enterButton_Click
228
229 // Обработчик события exitButton_Click
230 private void exitButton_Click( object sender, EventArgs e )
231 {
232     // Проверить, существует ли файл
233     if ( output != null )
234     {
235         // Закрыть файл
236         try
237         {
238             output.Close(); // Закрыть FileStream
239         } // Конец try
240         // Сообщить об ошибке при закрытии файла
241         catch ( IOException )
242         {
243             MessageBox.Show( "Cannot close file", "Error",
244                 MessageBoxButtons.OK, MessageBoxIcon.Error );
245         } // Конец catch
246     } // Конец if
247
248     Application.Exit();
249 } // Конец метода exitButton_Click
250 } // Конец класса CreateFileForm
251 } // Конец пространства имен CreateFile
```

**Ил. 17.14.** Создание файла посредством сериализации (окончание)

## 17.10. Чтение и десериализация данных из двоичного формата

В предыдущем разделе было показано, как создать файл с последовательным доступом, в который записываются сериализованные объекты. В этом разделе рассматривается процесс последовательного чтения сериализованных объектов из файла.

Программа на ил. 17.15 читает и выводит содержимое файла `clients.ser`, созданного кодом на ил. 17.14. Так как снимки экрана не отличаются от приведенных на ил. 17.11, они здесь не приводятся. В строке 15 создается объект `BinaryFormatter`, который будет использоваться для чтения объектов. Программа открывает файл для ввода, создавая объект `FileStream` (строки 49–50). Имя открываемого файла передается в первом аргументе конструктора `FileStream`.

Программа читает объекты из файла в обработчике события `nextButton_Click` (строки 59–92). Мы используем метод `Deserialize` (объекта `BinaryFormatter`, созданного в строке 15) для чтения данных (строки 65–66). Обратите внимание на преобразование результата `Deserialize` к типу `RecordSerializable` (строка 66) — это необходимо, потому что `Deserialize` возвращает ссылку типа `object`, а нам нужно обращаться к свойствам, принадлежащим классу `RecordSerializable`. Если в процессе десериализации произойдет ошибка или будет достигнут конец файла, выдается исключение `SerializationException`, а объект `FileStream` закрывается (строка 82).

```
1 // Ил. 17.15: ReadSequentialAccessFileForm.cs
2 // Чтение из файла с использованием сериализации.
3 using System;
4 using System.Windows.Forms;
5 using System.IO;
6 using System.Runtime.Serialization.Formatters.Binary;
7 using System.Runtime.Serialization;
8 using BankLibrary;
9
10 namespace ReadSequentialAccessFile
11 {
12     public partial class ReadSequentialAccessFileForm : BankUIForm
13     {
14         // Объект для десериализации RecordSerializable в двоичном формате
15         private BinaryFormatter reader = new BinaryFormatter();
16         private FileStream input; // Поток для чтения из файла
17
18         // Конструктор без параметров
19         public ReadSequentialAccessFileForm()
20         {
21             InitializeComponent();
22         } // Конец конструктора
23
24         // Вызывается при нажатии кнопки Open
25         private void openButton_Click( object sender, EventArgs e )
```

**Ил. 17.15.** Чтение из файла посредством сериализации (продолжение ↗)



```
26     {
27         // Создание и вывод диалогового окна открытия файла
28         DialogResult result; // Результат OpenFileDialog
29         string fileName;    // Имя файла с данными
30
31         using ( OpenFileDialog fileChooser = new OpenFileDialog() )
32         {
33             result = fileChooser.ShowDialog();
34             fileName = fileChooser.FileName; // Получение введенного имени
35         } // Конец using
36
37         // Проверяем, что пользователь нажал кнопку "OK"
38         if ( result == DialogResult.OK )
39         {
40             ClearTextBoxes();
41
42             // Если задан недействительный файл, сообщить об ошибке
43             if ( fileName == string.Empty )
44                 MessageBox.Show( "Invalid File Name", "Error",
45                                 MessageBoxButtons.OK, MessageBoxIcon.Error );
46             else
47             {
48                 // Создание объекта FileStream для чтения файла
49                 input = new FileStream(
50                     fileName, FileMode.Open, FileAccess.Read );
51
52                 openButton.Enabled = false; // Кнопка Open File блокируется
53                 nextButton.Enabled = true;  // Кнопка Next Record доступна
54             } // Конец else
55         } // Конец if
56     } // Конец метода openButton_Click
57
58     // Вызывается при нажатии кнопки Next
59     private void nextButton_Click( object sender, EventArgs e )
60     {
61         // Десериализация RecordSerializable и заполнение текстовых полей
62         try
63         {
64             // Получение следующего объекта RecordSerializable из файла
65             RecordSerializable record =
66                 ( RecordSerializable ) reader.Deserialize( input );
67
68             // Сохранение значений RecordSerializable во временном массиве
69             string[] values = new string[] {
70                 record.Account.ToString(),
71                 record.FirstName.ToString(),
72                 record.LastName.ToString(),
73                 record.Balance.ToString()
74             };
75
76             // Копирование значений из массива в текстовые поля
77             SetTextBoxValues( values );
78         } // Конец try
```

**Ил. 17.15.** Чтение из файла посредством сериализации (продолжение ☞)

```
79         // Обработка отсутствия объектов RecordSerializable в файле
80         catch ( SerializationException )
81         {
82             input.Close(); // Закрытие FileStream
83             openButton.Enabled = true; // Кнопка Open File доступна
84             nextButton.Enabled = false; // Кнопка Next Record блокируется
85
86             ClearTextBoxes();
87
88             // Сообщить об отсутствии RecordSerializable в файле
89             MessageBox.Show( "No more records in file", string.Empty,
90                             MessageBoxButtons.OK, MessageBoxIcon.Information );
91         } // Конец catch
92     } // Конец метода nextButton_Click
93 } // Конец класса ReadSequentialAccessFileForm
94 } // Конец пространства имен ReadSequentialAccessFile
```

**Ил. 17.15.** Чтение из файла посредством сериализации (окончание)

## 17.11. Итоги

Из этой главы вы узнали, как организовать долгосрочное хранение данных в файлах. Как было сказано в начале главы, данные хранятся в виде 0 и 1, и комбинации этих значений образуют байты, поля, записи и файлы. Были рассмотрены некоторые классы из пространства имен `System.IO`. Класс `File` предназначен для работы с файлами, а классы `Directory` и `DirectoryInfo` — для работы с каталогами. Далее были продемонстрированы принципы работы с записями в текстовых файлах с последовательным доступом. Глава завершается описанием различий между хранением данных в текстовых файлах и сериализацией объектов и рассмотрением механизма сериализации для сохранения целых объектов и их загрузки из файлов.

# 18 Сортировка и поиск

## 18.1. Введение

В процессе *поиска* данных приложение определяет, присутствует ли значение (называемое *ключом* поиска) в наборе данных, и если присутствует — находит его местоположение. Два популярных алгоритма поиска — простой линейный поиск и более быстрый, но и более сложный, бинарный поиск. В процессе сортировки данные размещаются в определенном порядке, определяемом одним или несколькими ключами сортировки. Список имен может сортироваться по алфавиту, список банковских счетов — по номеру счета и т. д. В этой главе представлены два простых алгоритма сортировки: сортировка выбором и сортировка вставкой, а также более эффективный, но более сложный алгоритм сортировки слиянием. На ил. 18.1 приведена краткая сводка алгоритмов поиска и сортировки, рассмотренных в книге.

Глава	Алгоритм	Раздел
Алгоритмы поиска		
18	Линейный поиск	Раздел 18.2.1
	Бинарный поиск	Раздел 18.2.2
21	Метод <code>BinarySearch</code> класса <code>Array</code>	Ил. 21.3
	Метод <code>Contains</code> классов <code>List&lt;T&gt;</code> и <code>Stack&lt;T&gt;</code>	Ил. 21.4
	Метод <code>ContainsKey</code> класса <code>Dictionary&lt;K, T&gt;</code>	Ил. 21.7
Алгоритмы сортировки		
18	Сортировка выбором	Раздел 18.3.1
	Сортировка вставкой	Раздел 18.3.2
	Рекурсивная сортировка слиянием	Раздел 18.3.3
18, 21	Методы сортировки классов <code>Array</code> и <code>List&lt;T&gt;</code>	Ил. 18.4, 21.3–21.4

**Ил. 18.1.** Средства сортировки и поиска, представленные в книге

## 18.2. Алгоритмы поиска

Поиск телефонного номера, посещение веб-сайта, поиск определения слова в словаре — все эти операции связаны с поиском в значительных объемах данных. В следующих двух разделах рассматриваются два стандартных алгоритма поиска — один легко программируется, но обладает относительно низкой эффективностью, а другой относительно эффективен, но создает больше проблем с программированием.

### 18.2.1. Линейный поиск

Алгоритм линейного поиска последовательно просматривает каждый элемент массива. Если ключ поиска не соответствует элементу массива, алгоритм переходит к следующему элементу, а при достижении конца массива сообщает пользователю о том, что ключ не найден. Если ключ будет обнаружен, алгоритм проверяет каждый элемент, пока не найдет подходящий, и возвращает индекс этого элемента.

Для примера рассмотрим массив, содержащий следующие элементы:

34 56 2 10 77 51 93 30 5 52

Допустим, метод ищет значение 51. При использовании линейного алгоритма поиска метод сначала проверяет, совпадает ли 34 с ключом поиска. Совпадения нет, поэтому алгоритм проверяет, совпадает ли 56 с ключом поиска. Последовательное перемещение по массиву продолжается; метод проверяет 2, затем 10 и 77. При достижении значения 51, совпадающего с ключом поиска, метод возвращает индекс 5 — местонахождение 51 в массиве. Если после проверки всех элементов метод определяет, что ключ не совпадает ни с одним элементом в массиве, он возвращает -1. При наличии дубликатов в массиве линейный поиск возвращает индекс первого элемента, совпадающего с ключом поиска.

На ил. 18.2 приведено объявление класса `LinearArray`. Класс объявляет закрытую переменную экземпляра `data` (массив с элементами `int`) и статический объект `Random` с именем `generator` для заполнения массива случайными целыми числами. При создании объекта класса `LinearArray` конструктор (строки 12–19) создает массив `data` и инициализирует его случайными числами в диапазоне 10–99.

```
1 // Ил. 18.2: LinearArray.cs
2 // Класс содержит массив случайных целых чисел и метод
3 // для последовательного поиска в этом массиве.
4 using System;
5
6 public class LinearArray
7 {
8     private int[] data; // Массив значений
9     private static Random generator = new Random();
```

**Ил. 18.2.** Класс с массивом случайных целых чисел  
и методом для последовательного поиска в этом массиве (продолжение ➤)

```

10
11 // Создание массива заданного размера и заполнение его случайными числами
12 public LinearArray( int size )
13 {
14     data = newint[ size ]; // Выделение памяти для массива
15
16     // Заполнение массива случайными целыми числами в диапазоне 10-99
17     for ( int i = 0; i < size; ++i )
18         data[ i ] = generator.Next( 10, 100 );
19 } // Конец конструктора LinearArray
20
21 // Линейный поиск в данных
22 public int LinearSearch( int searchKey )
23 {
24     // Последовательный перебор массива
25     for ( int index = 0; index < data.Length; ++index )
26         if ( data[ index ] == searchKey )
27             return index; // Возвращает индекс найденного совпадения
28
29     return -1; // Искомое значение не найдено
30 } // Конец метода LinearSearch
31
32 // Метод для вывода значений из массива
33 public override string ToString()
34 {
35     string temporary = string.Empty;
36
37     // Перебор элементов массива
38     foreach ( int element in data )
39         temporary += element + " ";
40
41     temporary += "\n"; // Добавление символа новой строки
42     return temporary;
43 } // Конец метода ToString
44 } // Конец класса LinearArray

```

**Ил. 18.2.** Класс с массивом случайных целых чисел  
и методом для последовательного поиска в этом массиве (окончание)

В строках 22–30 выполняется линейный поиск. Ключ поиска передается в параметре `searchKey`. В строках 25–27 перебираются элементы массива. В строке 26 каждый элемент массива сравнивается с `searchKey`. Если значения равны, то строка 27 возвращает индекс элемента. Если цикл завершается, а значение так и остается найденным, то строка 29 возвращает `-1`. В строках 33–43 объявляется метод `ToString`, который возвращает строковое представление массива.

На ил. 18.3 создается объект типа `LinearArray` с именем `searchArray`, который содержит массив из 10 целых чисел (строка 13) и поддерживает поиск элементов с заданным значением. В строках 17–18 пользователь вводит ключ поиска, который сохраняется в `searchInt`. Строки 21–37 выполняются в цикле до тех пор, пока пользователь не введет специальное значение `-1`. В массиве хранятся целые числа от 10 до 99 (строка 18 на ил. 18.2). В строке 24 вызов метода `LinearSearch` проверяет,

находится ли значение `searchInt` в массиве. Если поиск `searchInt` проходит успешно, `LinearSearch` возвращает позицию элемента, которую метод выводит в строках 27–29. Если значение `searchInt` не обнаружено в массиве, `LinearSearch` возвращает `-1` и метод оповещает пользователя о неудаче (строки 31–32). В строках 35–36 пользователь вводит следующее целое число.

### Эффективность линейного поиска

Все алгоритмы поиска предназначены для решения одной задачи — поиска элемента, совпадающего с заданным ключом (если такой элемент существует). Однако существует много характеристик, по которым поисковые алгоритмы отличаются друг от друга. Главной характеристикой является объем работы, необходимой для завершения поиска. Для его представления часто используется метрика «большого *O*» — оценка времени выполнения алгоритма в худшем случае. Для алгоритмов поиска и сортировки она особенно сильно зависит от количества элементов в наборе данных и используемого алгоритма.

```

1 // Ил. 18.3: LinearSearchTest.cs
2 // Последовательный поиск элемента в массиве.
3 using System;
4
5 public class LinearSearchTest
6 {
7     public static void Main( string[] args )
8     {
9         int searchInt; // Ключ поиска
10        int position; // Местонахождение ключа в массиве
11
12        // Создание и вывод массива
13        LinearArray searchArray = new LinearArray( 10 );
14        Console.WriteLine( searchArray ); // Вывод массива
15
16        // Получение первого значения от пользователя
17        Console.Write( "Please enter an integer value (-1 to quit): " );
18        searchInt = Convert.ToInt32( Console.ReadLine() );
19
20        // Ввод искомого значения в цикле; -1 завершает приложение
21        while ( searchInt != -1 )
22        {
23            // Выполнение линейного поиска
24            position = searchArray.LinearSearch( searchInt );
25
26            if ( position != -1 ) // Значение найдено
27                Console.WriteLine(
28                    "The integer {0} was found in position {1}.\\n",
29                    searchInt, position );
30            else // Значение не найдено
31                Console.WriteLine( "The integer {0} was not found.\\n",
32                    searchInt );
33
34            // Получение следующего значения int

```

**Ил. 18.3.** Последовательный поиск в массиве (продолжение ➤)

```

35         Console.Write( "Please enter an integer value (-1 to quit): " );
36         searchInt = Convert.ToInt32( Console.ReadLine() );
37     } // Конец while
38 } // Конец Main
39 } // Конец класса LinearSearchTest

```

```
64 90 84 62 28 68 55 27 78 73
```

```

Please enter an integer value (-1 to quit): 78
The integer 78 was found in position 8.

```

```

Please enter an integer value (-1 to quit): 64
The integer 64 was found in position 0.

```

```

Please enter an integer value (-1 to quit): 65
The integer 65 was not found.

```

```
Please enter an integer value (-1 to quit): -1
```

### Ил. 18.3. Последовательный поиск в массиве (окончание)

#### Постоянная вычислительная сложность

Предположим, алгоритм должен проверять, равен ли первый элемент массива второму. Если массив состоит из 10 элементов, алгоритм потребует одного сравнения. Если массив состоит из 1000 элементов, количество сравнений не изменится. Такой алгоритм полностью независим от количества элементов в массиве; говорят, что он характеризуется *постоянной вычислительной сложностью*, которая в записи «большого  $O$ » представляется в виде  $O(1)$ . Алгоритм с вычислительной сложностью  $O(1)$  не обязательно ограничивается одним сравнением.  $O(1)$  означает лишь то, что количество сравнений неизменно — оно не увеличивается с ростом количества элементов в массиве. Алгоритм, который проверяет, равен ли первый элемент массива любому из следующих трех элементов, тоже обладает вычислительной сложностью  $O(1)$ , хотя и требует трех сравнений вместо одного.

#### Линейная вычислительная сложность

Алгоритм, который проверяет, равен ли первый элемент массива любому другому элементу, требует минимум  $n - 1$  сравнений, где  $n$  — количество элементов в массиве. Если массив содержит 10 элементов, алгоритм требует до 9 сравнений. Если массив содержит 1000 элементов, алгоритм требует до 999 сравнений. С ростом  $n$  эта часть выражения начинает доминировать, а вычитание 1 становится несущественным. Запись «большого  $O$ » предназначена для выделения доминирующих факторов и подавления факторов, которые теряют значимость с ростом  $n$ . По этой причине алгоритм, требующий до  $n - 1$  сравнений (как описанный ранее), обозначается записью  $O(n)$  — то есть обладает линейной вычислительной сложностью.  $O(n)$  обычно читается «порядка  $n$ ».

#### Квадратичная вычислительная сложность

Предположим, алгоритм должен проверять, повторяется ли значение элемента в массиве. Для этого он сравнивает первый элемент со всеми элементами массива,

второй элемент — со всеми элементами, кроме первого (с которым он уже сравнивался), третий — со всеми элементами, кроме первых двух и т. д. В итоге алгоритм должен выполнить  $(n - 1) + (n - 2) + \dots + 2 + 1$ , или  $n^2/2 - n/2$  сравнений. С ростом  $n$  составляющая  $n^2$  становится доминирующей, а составляющая  $n$  теряет значимость. И снова запись «большого  $O$ » выделяет фактор  $n^2$ , оставляя  $n^2/2$ . Но как мы вскоре увидим, постоянные множители в этой записи принято исключать.

Запись «большого  $O$ » отражает рост времени выполнения алгоритма относительно количества обработанных элементов. Допустим, алгоритму нужно выполнить  $n^2$  сравнений. С четырьмя элементами ему потребуются 16 сравнений, а с восьмью — 64 сравнения. При таком алгоритме с удвоением количества элементов число сравнений возрастает в 4 раза. Рассмотрим аналогичный алгоритм с  $n^2/2$  сравнениями. С четырьмя элементами алгоритму потребуются 8 сравнений, а с восьмью — 32 сравнения. И снова с удвоением количества элементов число сравнений возрастает в 4 раза. У обоих алгоритмов вычислительная сложность растет со скоростью квадрата  $n$ , поэтому константа игнорируется и оба алгоритма характеризуются записью  $O(n^2)$ ; это называется *квадратичным временем выполнения*, а запись читается «порядка  $n$ -квадрат».

При малых значениях  $n$  алгоритмы  $O(n^2)$  (на современных персональных компьютерах, выполняющих миллиарды операций в секунду) не оказывают заметного влияния на скорость выполнения. Но с ростом  $n$  быстродействие начинает ощутимо снижаться. Алгоритм  $O(n^2)$ , работающий с массивом из миллиона элементов, должен выполнить триллион «операций» (каждая из которых может состоять из нескольких машинных команд), что может занять несколько часов. А для массива из миллиарда элементов количество операций достигнет квинтиллиона — числа настолько огромного, что выполнение алгоритма может занять десятилетия! Как вы вскоре увидите, алгоритмы  $O(n^2)$  легко реализуются. Однако существуют другие алгоритмы, с более благоприятными характеристиками вычислительной сложности; их реализация обычно требует большего труда и хитроумия, но это того стоит — особенно с увеличением  $n$ .

### Вычислительная сложность линейного поиска

Алгоритмы линейного поиска выполняются за время  $O(n)$ . В худшем случае алгоритм должен проверить каждый элемент массива, чтобы узнать, присутствует ли среди них искомое значение. Если размер массива удваивается, то и количество необходимых сравнений возрастает вдвое. Линейный поиск обеспечивает выдающееся быстродействие, если элемент, соответствующий ключу поиска, находится в начале массива. Однако нас интересуют алгоритмы, которые обеспечивают (в среднем) хорошее быстродействие любых операций поиска, включая те, у которых элемент с совпадающим значением ключа находится ближе к концу массива.

Линейный поиск программируется проще всех остальных алгоритмов, но он может уступать другим алгоритмам по скорости. Если приложению приходится часто выполнять поиск в больших массивах, лучше реализовать другой, более эффективный алгоритм — например, алгоритм бинарного поиска, представленный в следующем разделе.





### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 18.1

Простейшие алгоритмы иногда оказываются недостаточно эффективными. У них есть свои достоинства — простота реализации, тестирования и отладки, но для достижения максимального быстродействия приходится применять более сложные алгоритмы.

## 18.2.2. Бинарный поиск

Алгоритм бинарного поиска эффективнее алгоритма линейного поиска, однако он требует предварительной сортировки массива. На первой итерации алгоритма проверяется средний элемент массива. Если он совпадает с ключом сортировки, выполнение алгоритма завершается. Если ключ поиска меньше среднего элемента (предполагается, что массив отсортирован по возрастанию), значит, он не может находиться во второй половине массива, поэтому алгоритм продолжает работать только с первой половиной массива (то есть от первого элемента до среднего, не включая его). Если ключ поиска больше среднего элемента, значит, он не может находиться в первой половине массива, поэтому алгоритм продолжает работать только со второй половиной массива (то есть от среднего элемента до последнего). При каждой итерации проверяется среднее значение в оставшейся части массива, называемой *подмассивом*. Подмассив может не содержать ни одного элемента, а может включать весь массив. Если ключ поиска не совпадает с элементом, алгоритм исключает половину оставшихся элементов. В конечном итоге алгоритм либо находит элемент, совпадающий с ключом поиска, либо сокращает подмассив до нулевого размера.

В качестве примера рассмотрим отсортированный массив из 15 элементов:

2 3 5 10 27 30 34 51 56 65 77 81 82 93 99

Поиск ведется по ключу 65. Приложение, реализующее алгоритм бинарного поиска, сначала проверяет, совпадает ли значение 51 с ключом поиска (потому что 51 — средний элемент массива). Ключ поиска (65) больше 51, поэтому значение 51 «отбрасывается» (то есть исключается из дальнейшего рассмотрения) вместе с первой половиной массива (все элементы, меньшие 51). Затем алгоритм проверяет, совпадает ли значение 81 (средний элемент оставшейся части массива) с ключом поиска. Ключ (65) меньше 81, поэтому значение 81 отбрасывается вместе с элементами, большими 81. После всего двух проверок алгоритм сократил количество проверяемых значений до 3 (56, 65 и 77). Затем алгоритм проверяет значение 65 (которое совпадает с ключом поиска) и возвращает индекс элемента массива, содержащего 65. Для поиска ключа среди элементов массива алгоритму хватило всего трех сравнений. С алгоритмом линейного поиска для этого потребовалось бы 10 сравнений. [Примечание: в приведенном примере мы использовали массив с 15 элементами, чтобы в нем всегда был очевиден средний элемент. При четном количестве элементов середина массива находится между двумя элементами. В нашей реализации будет выбираться больший из двух элементов.]


### Реализация бинарного поиска

На ил. 18.4 приведен класс `BinaryArray`. Как и `LinearArray`, он содержит закрытую переменную экземпляра `data` (массив с элементами `int`), статический объект `Random` с именем `generator` для заполнения массива случайно сгенерированными целыми числами, конструктор, метод поиска (`BinarySearch`), метод `RemainingElements` (создает строку с элементами массива, в которых будет проводиться дальнейший поиск) и `ToString`. В строках 12–21 объявляется конструктор. После инициализации массива случайными целыми числами от 10–99 (строки 17–18) в строке 20 вызывается метод `Array.Sort` для массива `data`. Статический метод `Sort` класса `Array` сортирует элементы массива по возрастанию. Не забывайте, что алгоритм бинарного поиска работает только с отсортированными массивами.

```

1 // Ил. 18.4: BinaryArray.cs
2 // Класс содержит массив случайных целых чисел и метод
3 // для бинарного поиска в этом массиве.
4 using System;
5
6 public class BinaryArray
7 {
8     private int[] data; // Массив значений
9     private static Random generator = new Random();
10
11     // Создание массива и его заполнение случайными числами
12     public BinaryArray( int size )
13     {
14         data = new int[ size ]; // Выделение памяти для массива
15
16         // Заполнение массива случайными числами в диапазоне 10-99
17         for ( int i = 0; i < size; ++i )
18             data[ i ] = generator.Next( 10, 100 );
19
20         Array.Sort( data );
21     } // Конец конструктора BinaryArray
22
23     // Бинарный поиск в данных
24     public int BinarySearch( int searchElement )
25     {
26         int low = 0; // Нижняя граница области поиска
27         int high = data.Length - 1; // Верхняя граница области поиска
28         int middle = ( low + high + 1 ) / 2; // Средний элемент
29         int location = -1; // Возвращаемое значение; -1, если ключ не найден
30
31         do // Цикл поиска элемента массива
32         {
33             // Вывод оставшихся элементов массива
34             Console.Write( RemainingElements( low, high ) );
35
36             // Вывод пробелов для выравнивания
37             for ( int i = 0; i < middle; ++i )
38                 Console.Write( " " );

```

**Ил. 18.4.** Класс с массивом случайных целых чисел и методом для бинарного поиска в этом массиве (продолжение )

```

39
40     Console.WriteLine( " * " ); // Обозначение текущей середины
41
42     // Если элемент найден в середине диапазона
43     if ( searchElement == data[ middle ] )
44         location = middle; // Ключ найден в текущей середине
45
46     // Средний элемент слишком велик
47     else if ( searchElement < data[ middle ] )
48         high = middle - 1; // Исключить верхнюю половину
49     else // Средний элемент слишком мал
50         low = middle + 1; // Исключить нижнюю половину
51
52     middle = ( low + high + 1 ) / 2; // Вычисление середины
53 } while ( ( low <= high ) && ( location == -1 ) );
54
55 return location; // Вернуть позицию ключа поиска
56 } // Конец метода BinarySearch
57
58 // Метод для вывода диапазона элементов массива
59 public string RemainingElements( int low, int high )
60 {
61     string temporary = string.Empty;
62
63     // Вывод пробелов для выравнивания
64     for ( int i = 0; i < low; ++i )
65         temporary += " ";
66
67     // Вывод элементов, оставшихся в массиве
68     for ( int i = low; i <= high; ++i )
69         temporary += data[ i ] + " ";
70
71     temporary += "\n";
72     return temporary;
73 } // Конец метода RemainingElements
74
75 // Метод вывода значений из массива
76 public override string ToString()
77 {
78     return RemainingElements( 0, data.Length - 1 );
79 } // Конец метода ToString
80 } // Конец класса BinaryArray

```

**Ил. 18.4.** Класс с массивом случайных целых чисел и методом для бинарного поиска в этом массиве (окончание)

В строках 24–56 объявляется метод `BinarySearch`. Ключ поиска передается в параметре `searchElement` (строка 24). В строках 26–28 вычисляется индекс нижней границы `low`, индекс верхней границы `high` и индекс середины части массива, в которой в настоящее время проводится поиск `middle`. В начале метода значение `low` равно 0, значение `high` — длине массива минус 1, а значение `middle` — их среднему арифметическому. В строке 29 индекс искомого элемента `location` инициализируется значением `-1` — значением, которое будет возвращено в случае неудачного поиска. В строках 31–53 цикл выполняется до тех пор, пока значение `low` не станет

больше `high` (это происходит, если элемент не найден), или значение `location` не станет отличным от `-1` (означает, что ключ поиска найден). Строка 43 проверяет, равно ли значение среднего элемента `searchElement`. Если они равны, строка 44 присваивает переменной `location` значение `middle`. Тогда цикл завершается, а значение `location` возвращается вызывающей стороне. Каждая итерация цикла проверяет одно значение (строка 43) и исключает половину оставшихся значений в массиве (строка 48 или 50).

В строках 22–40 на ил. 18.5 цикл выполняется до тех пор, пока пользователь не введет `-1`. При вводе любого другого числа приложение выполняет бинарный поиск, проверяя, присутствует ли введенное число в элементе массива. Первая строка результатов приложения содержит массив целых чисел, упорядоченный по возрастанию. Когда пользователь приказывает приложению найти значение `72`, приложение сначала проверяет средний элемент (\* в примере вывода на ил. 18.5), который равен `52`. Ключ поиска больше `52`, поэтому приложение исключает из рассмотрения первую половину массива и проверяет средний элемент второй половины. Ключ поиска меньше `82`, поэтому приложение исключает из рассмотрения вторую половину подмассива, оставляя только три элемента. Наконец, приложение проверяет значение `72` (совпадающее с ключом поиска) и возвращает индекс `9`.

```
1 // Ил. 18.5: BinarySearchTest.cs
2 // Бинарный поиск элемента в массиве.
3 using System;
4
5 public class BinarySearchTest
6 {
7     public static void Main( string[] args )
8     {
9         int searchInt; // Ключ поиска
10        int position; // Местонахождение ключа в массиве
11
12        // Создание и вывод массива
13        BinaryArray searchArray = new BinaryArray( 15 );
14        Console.WriteLine( searchArray );
15
16        // Получение первого значения от пользователя
17        Console.Write( "Please enter an integer value (-1 to quit): " );
18        searchInt = Convert.ToInt32( Console.ReadLine() );
19        Console.WriteLine();
20
21        // Ввод искомого значения в цикле; -1 завершает приложение
22        while ( searchInt != -1 )
23        {
24            // Выполнение бинарного поиска
25            position = searchArray.BinarySearch( searchInt );
26
27            // Значение не найдено
28            if ( position == -1 )
29                Console.WriteLine( "The integer {0} was not found.\n",
30                                searchInt );
```

**Ил. 18.5.** Поиск элемента в массиве с использованием бинарного поиска (продолжение ↗)

```

31         else
32             Console.WriteLine(
33                 "The integer {0} was found in position {1}.\n",
34                 searchInt, position);
35
36         // Получение следующего значения int
37         Console.Write( "Please enter an integer value (-1 to quit): " );
38         searchInt = Convert.ToInt32( Console.ReadLine() );
39         Console.WriteLine();
40     } // Конец while
41 } // Конец Main
42 } // Конец класса BinarySearchTest

```

```

12 17 22 25 30 39 40 52 56 72 76 82 84 91 93

Please enter an integer value (-1 to quit): 72

12 17 22 25 30 39 40 52 56 72 76 82 84 91 93
                        *
                        56 72 76 82 84 91 93
                              *
                              56 72 76
                                    *
                                    The integer 72 was found in position 9.

Please enter an integer value (-1 to quit): 13

12 17 22 25 30 39 40 52 56 72 76 82 84 91 93
                        *

12 17 22 25 30 39 40
                *

12 17 22
        *

12
*
The integer 13 was not found.

Please enter an integer value (-1 to quit): -1

```

**Ил. 18.5.** Поиск элемента в массиве с использованием бинарного поиска (окончание)

### Эффективность бинарного поиска

В худшем случае поиск в отсортированном массиве с 1023 элементами при бинарном поиске потребует около 10 сравнений. Многократное деление 1023 на 2 (потому что после каждого сравнения половина массива исключается из рассмотрения) с округлением вниз (так как средний элемент тоже исключается) дает значения 511, 255, 127, 63, 31, 15, 7, 3, 1 и 0. Число 1023 ( $2^{10} - 1$ ) всего после 10 делений на 2 уменьшилось до 0 (признак того, что элементов для проверки не осталось). Деление на 2 эквивалентно одному сравнению в алгоритме бинарного поиска. Таким образом, для поиска ключа в массиве с 1 048 575 ( $2^{20} - 1$ ) элементами достаточно всего 20 сравнений, а в массиве с миллиардом элементов (что менее  $2^{30} - 1$ ) для поиска ключа потребуется не более 30 сравнений. Это огромный выигрыш по сравнению с линейным поиском: в массиве с миллиардом элементов при линейном поиске потребуется около 500 миллионов

сравнений, тогда как с бинарным поиском хватит всего 30! Максимальное количество сравнений, необходимых при бинарном поиске в произвольном отсортированном массиве, равно двоичному логарифму от количества элементов в массиве, то есть  $\log_2 n$ . Все логарифмы растут приблизительно с одинаковой скоростью, поэтому в записи «большого O» основание логарифма можно опустить. Итак, в записи «большого O» для вычислительной сложности бинарного поиска используется обозначение  $O(\log n)$ , а сама сложность называется *логарифмической*.

## 18.3. Алгоритмы сортировки

Сортировка данных (то есть размещение данных в определенном порядке — например, по возрастанию или убыванию) является одним из важнейших применений компьютерных технологий. Банк сортирует чеки по номеру счета, чтобы подготовить выписки по счетам к концу месяца. Телефонные компании сортируют свои справочники сначала по фамилии, затем по имени, чтобы упростить поиск телефонных номеров. Практически в любой организации возникает задача сортировки данных — часто в огромных объемах. Сортировка данных — нетривиальная задача, требующая значительных вычислительных ресурсов, и в этой области велись серьезные исследования.

Важно понять, что при сортировке конечный результат — отсортированный массив — остается одним и тем же независимо от того, какой алгоритм использовался для сортировки. Выбор алгоритма влияет только на вычислительную сложность и затраты памяти. В оставшейся части главы представлены три распространенных алгоритма сортировки. Первые два — сортировка выбором и сортировка вставкой — легко реализуются, но обладают низкой эффективностью. Последний алгоритм — сортировка слиянием — работает намного быстрее, но отличается большей сложностью реализации. Мы сосредоточимся на сортировке массивов данных простых типов, а именно `int`. Также возможна сортировка массивов объектов — эта тема будет рассмотрена в главе 21.

### 18.3.1. Сортировка выбором

Алгоритм сортировки выбором прост, но неэффективен. На первой итерации алгоритм выбирает наименьший элемент массива и меняет его местами с первым элементом. На второй итерации наименьший из оставшихся элементов меняется местами со вторым элементом. Алгоритм продолжает работу до тех пор, пока последняя итерация не выберет элемент, второй по величине, и при необходимости не поменяет его местами с предпоследним элементом, после чего наибольший элемент останется в последней позиции. После итерации с номером  $i$  наименьшие  $i$  элементов массива будут отсортированы по возрастанию в первых  $i$  позициях массива. В качестве примера рассмотрим массив

34 56 4 10 77 51 93 30 5 52

Приложение, реализующее сортировку выбором, сначала находит в этом массиве наименьший элемент (4), который имеет индекс 2 (то есть находится в позиции 3). Приложение меняет местами 4 и 34, в результате чего массив принимает вид

```
4 56 34 10 77 51 93 30 5 52
```

Затем приложение находит наименьшее значение среди остальных элементов (все элементы, кроме 4); это значение 5 в элементе с индексом 8. Элементы 5 и 56 меняются местами, а массив принимает вид

```
4 5 34 10 77 51 93 30 56 52
```

На третьей итерации приложение находит следующее наименьшее значение (10) и меняет его местами с 34.

```
4 5 10 34 77 51 93 30 56 52
```

Процесс продолжается до тех пор, пока массив не будет полностью отсортирован.

```
4 5 10 30 34 51 52 56 77 93
```

После первой итерации наименьший элемент находится в первой позиции. После второй итерации два наименьших элемента располагаются по порядку в первых двух позициях. После третьей итерации три наименьших элемента располагаются по порядку в первых трех позициях.

На ил. 18.6 приведен класс `SelectionSort`, который объявляет закрытую переменную экземпляра `data` (массив с элементами `int`) и статический объект `Random` с именем `generator` для заполнения массива случайными целыми числами. При создании объекта класса `SelectionSort` конструктор (строки 12–19) создает массив `data` и инициализирует его случайными числами в диапазоне 10–99.

```
1 // Ил. 18.6: SelectionSort.cs
2 // Класс создает массив, заполненный случайными целыми числами.
3 // Предоставляет метод для выполнения сортировки выбором.
4 using System;
5
6 public class SelectionSort
7 {
8     private int[] data; // Массив значений
9     private static Random generator = new Random();
10
11     // Создание массива заданного размера и заполнение его случайными числами
12     public SelectionSort( int size )
13     {
14         data = newint[ size ]; // Выделение памяти для массива
15
16         // Заполнение массива случайными целыми числами в диапазоне 10-99
17         for ( int i = 0; i < size; ++i )
18             data[ i ] = generator.Next( 10, 100 );
19     } // Конец конструктора SelectionSort
```

**Ил. 18.6.** Класс с массивом случайных целых чисел и методом для выполнения сортировки выбором (продолжение ↗)

```

20
21 // Упорядочение массива методом сортировки выбором
22 public void Sort()
23 {
24     int smallest; // Индекс наименьшего элемента
25
26     // Перебор по data.Length - 1 элементам
27     for ( int i = 0; i < data.Length - 1; ++i )
28     {
29         smallest = i; // Первый индекс оставшегося массива
30
31         // Перебор для определения индекса наименьшего элемента
32         for ( int index = i + 1; index < data.Length; ++index )
33             if ( data[ index ] < data[ smallest ] )
34                 smallest = index;
35
36         Swap( i, smallest ); // Поменять местами с наименьшим элементом
37         PrintPass( i + 1, smallest ); // Вывести номер итерации
38     } // Конец внешнего цикла for
39 } // Конец метода Sort
40
41 // Вспомогательный метод для перестановки значений двух элементов
42 public void Swap( int first, int second )
43 {
44     int temporary = data[ first ]; // Сохранение first в temporary
45     data[ first ] = data[ second ]; // Замена first на second
46     data[ second ] = temporary; // Перемещение temporary в second
47 } // Конец метода Swap
48
49 // Вывод итерации алгоритма
50 public void PrintPass( int pass, int index )
51 {
52     Console.Write( "after pass {0}: ", pass );
53
54     // Вывод элементов до выбранного
55     for ( int i = 0; i < index; ++i )
56         Console.Write( data[ i ] + " " );
57
58     Console.Write( data[ index ] + "* " ); // Обозначение перестановки
59
60     // Завершение вывода массива
61     for ( int i = index + 1; i < data.Length; ++i )
62         Console.Write( data[ i ] + " " );
63
64     Console.Write( "\n " ); // Для выравнивания
65
66     // Обозначение отсортированной части массива
67     for( int j = 0; j < pass; ++j )
68         Console.Write( "-- " );
69     Console.WriteLine( "\n" ); // Пропуск строки
70 } // Конец метода PrintPass

```

**Ил. 18.6.** Класс с массивом случайных целых чисел и методом для выполнения сортировки выбором (продолжение ↗)



```

71
72 // Метод для вывода значений из массива
73 public override string ToString()
74 {
75     string temporary = string.Empty;
76
77     // Перебор массива
78     foreach ( int element in data )
79         temporary += element + " ";
80
81     temporary += "\n"; // Добавление символа новой строки
82     return temporary;
83 } // Конец метода ToString
84 } // Конец класса SelectionSort

```

**Ил. 18.6.** Класс с массивом случайных целых чисел и методом для выполнения сортировки выбором (окончание)

В строках 22–39 объявляется метод `Sort`. В строке 24 объявляется переменная `smallest`, в которой будет храниться индекс наименьшего элемента в оставшейся части массива. Строки 27–38 выполняются в цикле `data.Length-1` раз. Строка 29 инициализирует индекс наименьшего элемента индексом текущего элемента. Строки 32–34 перебирают оставшиеся элементы в массиве. Для каждого из этих элементов строка 33 сравнивает его значение со значением наименьшего известного элемента `smallest`. Если текущий элемент меньше `smallest`, то в строке 34 `smallest` присваивается индекс текущего элемента. При завершении цикла `smallest` будет содержать индекс наименьшего элемента в оставшемся массиве. Строка 36 вызывает метод `Swap` (строки 42–47) для перемещения наименьшего из оставшихся элементов на следующую позицию в массиве.

Строка 10 на ил. 18.7 создает объект `SelectionSort` с 10 элементами. В строке 13 метод `ToString` неявно вызывается для вывода несортированного объекта. В строке 15 вызывается метод `Sort` (см. ил. 8.6, строки 22–39), который применяет к массиву сортировку выбором. Затем в строках 17–18 выводится отсортированный объект. В выходных данных часть массива, отсортированная после каждого прохода, обозначается дефисами (см. ил. 18.6, строки 67–68). Звездочка размещается рядом с позицией элемента, который поменялся местами с наименьшим элементом на текущем проходе. При каждой итерации были переставлены элемент, помеченный звездочкой, и элемент над крайними правыми дефисами.

```

1 // Ил. 18.7: SelectionSortTest.cs
2 // Тестирование класса сортировки выбором.
3 using System;
4
5 public class SelectionSortTest
6 {
7     public static void Main( string[] args )
8     {
9         // Создание объекта для выполнения сортировки выбором.

```

**Ил. 18.7.** Тестирование реализации сортировки выбором (продолжение ➤)

```

10     SelectionSort sortArray = new SelectionSort( 10 );
11
12     Console.WriteLine( "Unsorted array:" );
13     Console.WriteLine( sortArray ); // Вывод несортированного массива
14
15     sortArray.Sort(); // Сортировка массива
16
17     Console.WriteLine( "Sorted array:" );
18     Console.WriteLine( sortArray ); // Вывод отсортированного массива
19 } // Конец Main
20 } // Конец класса SelectionSortTest

```

```

Unsorted array:
86 97 83 45 19 31 86 13 57 61
after pass 1: 13 97 83 45 19 31 86 86* 57 61
--
after pass 2: 13 19 83 45 97* 31 86 86 57 61
-- --
after pass 3: 13 19 31 45 97 83* 86 86 57 61
-- -- --
after pass 4: 13 19 31 45* 97 83 86 86 57 61
-- -- -- --
after pass 5: 13 19 31 45 57 83 86 86 97* 61
-- -- -- -- --
after pass 6: 13 19 31 45 57 61 86 86 97 83*
-- -- -- -- --
after pass 7: 13 19 31 45 57 61 83 86 97 86*
-- -- -- -- --
after pass 8: 13 19 31 45 57 61 83 86* 97 86
-- -- -- -- --
after pass 9: 13 19 31 45 57 61 83 86 86 97*
-- -- -- -- --

Sorted array:
13 19 31 45 57 61 83 86 86 97

```

**Ил. 18.7.** Тестирование реализации сортировки выбором (окончание)

### Эффективность сортировки выбором

Алгоритм сортировки выбором выполняется за время  $O(n^2)$ . Метод Sort в строках 22–39 на ил. 18.6, реализующий алгоритм сортировки выбором, содержит вложенные циклы for. Внешний цикл for (строки 27–38) перебирает первые  $n-1$  элементов массива и переставляет наименьший из оставшихся элементов с его позицией в порядке сортировки. Внутренний цикл for (строки 32–34) перебирает все элементы оставшегося несортированного массива в поисках наименьшего. Цикл выполняется  $n-1$  раз при первой итерации внешнего цикла,  $n-2$  раз при второй итерации, затем  $n-3$ , ..., 3, 2, 1. В общей сложности внутренний цикл будет выполнен  $n(n-1)/2$ , или  $(n^2-n)/2$ . В записи «большого  $O$ » менее значимые факторы опускаются, а константы игнорируются; в итоге мы получаем  $O(n^2)$ .

### 18.3.2. Сортировка вставкой

Сортировка вставкой — еще один простой, хотя и неэффективный алгоритм сортировки. Первая итерация проверяет второй элемент массива, и если тот меньше первого — меняет их местами. Вторая итерация проверяет третий элемент и вставляет его в правильную позицию по отношению к первым двум элементам (с перемещением их в случае необходимости), так что все три элемента располагаются по порядку. На  $i$ -й итерации алгоритма первые  $i$  элементов исходного массива хранятся в отсортированном порядке.

Следующий массив идентичен тому, который мы использовали при обсуждении сортировки выбором:

```
34 56 4 10 77 51 93 30 5 52
```

Приложение, реализующее алгоритм сортировки вставкой, сначала проверяет первые два элемента массива, 34 и 56. Эти элементы уже хранятся в нужном порядке, поэтому выполнение продолжается (если бы элементы хранились не по порядку, то они поменялись бы местами).

На следующей итерации проверяется третье значение 4. Оно меньше 56, поэтому приложение сохраняет 4 во временной переменной и перемещает 56 на один элемент вправо. Затем приложение сравнивает 4 с 34 и определяет, что 4 меньше, поэтому 34 перемещается на один элемент вправо. Приложение достигло начала массива, поэтому значение 4 помещается в позицию с нулевым индексом. На этой стадии массив выглядит так:

```
4 34 56 10 77 51 93 30 5 52
```

На следующей итерации приложение сохраняет значение 10 во временной переменной. Затем приложение сравнивает 10 с 56 и перемещает 56 на один элемент вправо, потому что 56 больше 10. Затем 10 сравнивается с 34, и 34 перемещается на один элемент вправо. Сравнивая 10 с 4, приложение видит, что 10 больше 4, и помещает 10 в элемент с индексом 1. Массив принимает вид:

```
4 10 34 56 77 51 93 30 5 52
```

При использовании этого алгоритма на  $i$ -й итерации первые  $i$  элементов исходного массива отсортированы, но их позиции еще могут измениться — если в массиве где-то обнаружатся меньшие значения.

На ил. 18.8 представлен класс `InsertionSort`. В строках 22–46 объявляется метод `Sort`. В строке 24 объявляется переменная `insert` для элемента, вставляемого с перемещением всех остальных элементов. В строках 27–45 перебираются `data.Length - 1` элементов массива. При каждой итерации строка 30 сохраняет в переменной `insert` значение элемента, который будет вставляться в отсортированную часть массива. Строка 33 объявляет и инициализирует переменную `moveItem`, в которой хранится место вставки элемента. Цикл в строках 36–41 ищет позицию для вставки элемента. Цикл завершается либо при достижении начала массива, либо при достижении

элемента, меньшего вставляемого значения. В строке 39 элемент перемещается вправо, а в строке 40 уменьшается позиция вставки следующего элемента. После завершения цикла строка 43 вставляет элемент на место. Код на ил. 18.9 почти не отличается от кода на ил. 18.7, не считая того, что он создает и использует объект InsertionSort. В выходных данных часть массива, отсортированная после каждого прохода, обозначается дефисами (см. ил. 18.8, строки 66–67). Звездочка размещается рядом с элементом, который был вставлен на место в текущей итерации.

```

1 // Ил. 18.8: InsertionSort.cs
2 // Класс создает массив, заполненный случайными целыми числами.
3 // Предоставляет метод для выполнения сортировки вставкой.
4 using System;
5
6 public class InsertionSort
7 {
8     private int[] data; // Массив значений
9     private static Random generator = new Random();
10
11     // Создание массива заданного размера и заполнение его случайными числами
12     public InsertionSort( int size )
13     {
14         data = new int[ size ]; // Выделение памяти для массива
15
16         // Заполнение массива случайными целыми числами в диапазоне 10-99
17         for ( int i = 0; i < size; ++i )
18             data[ i ] = generator.Next( 10, 100 );
19     } // Конец конструктора InsertionSort
20
21     // Упорядочение массива методом сортировки вставкой
22     public void Sort()
23     {
24         int insert; // Временная переменная для вставляемого элемента
25
26         // Перебор по data.Length - 1 элементам
27         for ( int next = 1; next < data.Length; ++next )
28         {
29             // Сохранение значения в текущем элементе
30             insert = data[ next ];
31
32             // Инициализация позиции для размещения элемента
33             int moveItem = next;
34
35             // Поиск места для размещения текущего элемента
36             while ( moveItem > 0 && data[ moveItem - 1 ] > insert )
37             {
38                 // Сдвиг элемента вправо на одну позицию
39                 data[ moveItem ] = data[ moveItem - 1 ];
40                 moveItem--;
41             } // Конец while
42
43             data[ moveItem ] = insert; // Размещение вставленного элемента
44             PrintPass( next, moveItem ); // Вывод итерации алгоритма

```

**Ил. 18.8.** Класс с массивом случайных целых чисел  
и методом для выполнения сортировки вставкой (продолжение ↗)

```

45     } // Конец for
46 } // Конец метода Sort
47
48 // Вывод итерации
49 public void PrintPass( int pass, int index )
50 {
51     Console.Write( "after pass {0}: ", pass );
52
53     // Вывод элементов до переставленного
54     for ( int i = 0; i < index; ++i )
55         Console.Write( data[ i ] + " " );
56
57     Console.Write( data[ index ] + "*" ); // Обозначение перестановки
58
59     // Завершение вывода массива
60     for ( int i = index + 1; i < data.Length; ++i )
61         Console.Write( data[ i ] + " " );
62
63     Console.WriteLine( "\n " ); // Для выравнивания
64
65     // Обозначение отсортированной части массива
66     for( int i = 0; i <= pass; ++i )
67         Console.Write( "-- " );
68     Console.WriteLine( "\n" ); // Пропуск строки при выводе
69 } // Конец метода PrintPass
70
71 // Метод для вывода значений из массива
72 public override string ToString()
73 {
74     string temporary = string.Empty;
75
76     // Перебор элементов массива
77     foreach ( int element in data )
78         temporary += element + " ";
79
80     temporary += "\n"; // Добавление символа новой строки
81     return temporary;
82 } // Конец метода ToString
83 } // Конец класса InsertionSort

```

**Ил. 18.8.** Класс с массивом случайных целых чисел  
и методом для выполнения сортировки вставкой (окончание)

```

1 // Ил. 18.9: InsertionSortTest.cs
2 // Тестирование класса, выполняющего сортировку вставкой.
3 using System;
4
5 public class InsertionSortTest
6 {
7     public static void Main( string[] args )
8     {
9         // Создание объекта для выполнения сортировки вставкой
10         InsertionSort sortArray = new InsertionSort( 10 );

```

**Ил. 18.9.** Тестирование реализации сортировки вставкой (продолжение ↗)

```

11
12     Console.WriteLine( "Unsorted array:" );
13     Console.WriteLine( sortArray ); // Вывод несортированного массива
14
15     sortArray.Sort(); // Сортировка массива
16
17     Console.WriteLine( "Sorted array:" );
18     Console.WriteLine( sortArray ); // Вывод сортированного массива
19 } // Конец Main
20 } // Конец класса InsertionSortTest

```

```

Unsorted array:
12 27 36 28 33 92 11 93 59 62
after pass 1: 12 27* 36 28 33 92 11 93 59 62
-- --
after pass 2: 12 27 36* 28 33 92 11 93 59 62
-- -- --
after pass 3: 12 27 28* 36 33 92 11 93 59 62
-- -- -- --
after pass 4: 12 27 28 33* 36 92 11 93 59 62
-- -- -- -- --
after pass 5: 12 27 28 33 36 92* 11 93 59 62
-- -- -- -- --
after pass 6: 11* 12 27 28 33 36 92 93 59 62
-- -- -- -- --
after pass 7: 11 12 27 28 33 36 92 93* 59 62
-- -- -- -- --
after pass 8: 11 12 27 28 33 36 59* 92 93 62
-- -- -- -- --
after pass 9: 11 12 27 28 33 36 59 62* 92 93
-- -- -- -- --

Sorted array:
11 12 27 28 33 36 59 62 92 93

```

**Ил. 18.9.** Тестирование реализации сортировки вставкой (окончание)

### Эффективность сортировки вставкой

Алгоритм сортировки вставкой также выполняется за время  $O(n^2)$ . Как и в случае сортировки выбором, реализация сортировки вставкой (строки 22–46 на ил. 18.8) содержит вложенные циклы. Цикл `for` (строки 27–45) выполняется `data.Length-1` раз; при каждой итерации элемент вставляется в соответствующую позицию среди элементов, отсортированных до настоящего момента. В контексте нашего приложения выражение `data.Length-1` эквивалентно  $n-1$  (так как `data.Length` — размер массива). Цикл `while` (строки 36–41) выполняет перебор среди предшествующих элементов массива. В худшем случае этот цикл `while` потребует  $n-1$  сравнения. Каждый отдельный цикл выполняется за время  $O(n)$ . В записи «большого  $O$ » вложенные циклы означают умножение количеств итераций каждого цикла. Для каждой итерации внешнего цикла выполняется определенное количество итераций внутреннего цикла. В нашем алгоритме для каждой из  $O(n)$  итераций внешнего цикла будет выполнено  $O(n)$  итераций внутреннего цикла. Умножение этих значений в записи «большого  $O$ » дает вычислительную сложность  $O(n^2)$ .

### 18.3.3. Сортировка слиянием

Алгоритм сортировки слиянием отличается большей эффективностью, но зато он концептуально сложнее алгоритмов сортировки выбором и вставкой. Алгоритм сортировки слиянием основан на разбиении массива на два подмассива одинакового размера; каждый подмассив сортируется, после чего они сливаются в один больший массив. С нечетным количеством элементов алгоритм создает два подмассива, один из которых содержит на один элемент больше другого.

В нашем примере используется рекурсивная реализация сортировки слиянием. Базовый случай — массив из одного элемента. Конечно, такой массив всегда отсортирован, поэтому сортировка слиянием немедленно возвращает управление для вызова с одноэлементным массивом. Остальные массивы разбиваются на два фрагмента приблизительно одинаковой длины, которые рекурсивно сортируются и сливаются в один больший отсортированный массив.

Предположим, алгоритм уже отсортировал меньшие массивы, в результате чего были созданы отсортированные массивы А:

4 10 34 56 77

и В:

5 30 51 52 93

Алгоритм сортировки слиянием объединяет эти два массива в один отсортированный массив большего размера. Наименьшим элементом А является 4 (элемент А с нулевым индексом). Наименьшим элементом В является 5 (элемент В с нулевым индексом). Для определения наименьшего элемента в объединенном массиве алгоритм сравнивает 4 и 5. Значение из массива А меньше, поэтому 4 становится первым элементом объединенного массива. Затем алгоритм сравнивает 10 (второй элемент массива А) с 5 (первым элементом массива В). Значение из массива В меньше, поэтому 5 становится вторым элементом большого массива. Далее алгоритм сравнивает 10 с 30, значение 10 становится третьим элементом в массиве и т. д.

В строках 22–25 на ил. 18.10 объявляется метод Sort. В строке 24 метод SortArray вызывается с аргументами 0 и data.Length-1 — начальным и конечным индексами сортируемого массива. Эти значения сообщают методу SortArray, что он должен работать со всем массивом.

```

1 // Ил. 18.10: MergeSort.cs
2 // Класс создает массив, заполненный случайными целыми числами.
3 // Предоставляет метод для выполнения сортировки слиянием.
4 using System;
5
6 public class MergeSort
7 {
8     private int[] data; // Массив значений
9     private static Random generator = new Random();

```

**Ил. 18.10.** Класс с массивом случайных целых чисел и методом для выполнения сортировки слиянием (продолжение ↗)

```

10
11 // Создание массива заданного размера и заполнение его случайными числами
12 public MergeSort( int size )
13 {
14     data = new int[ size ]; // Выделение памяти для массива
15
16     // Заполнение массива случайными целыми числами в диапазоне 10-99
17     for ( int i = 0; i < size; ++i )
18         data[ i ] = generator.Next( 10, 100 );
19 } // Конец конструктора MergeSort
20
21 // Вызов рекурсивного метода SortArray для запуска сортировки слиянием
22 public void Sort()
23 {
24     SortArray( 0, data.Length - 1 ); // Сортировка всего массива
25 } // Конец метода Sort
26
27 // Разбиение массива, сортировка подмассивов и их слияние
28 private void SortArray( int low, int high )
29 {
30     // Проверка базового случая; размер массива равен 1
31     if ( ( high - low ) >= 1 ) // Если не базовый случай
32     {
33         int middle1 = ( low + high ) / 2; // Вычисление середины массива
34         int middle2 = middle1 + 1; // Вычисление следующего элемента
35
36         // Вывод разбиения
37         Console.WriteLine( "split: " + Subarray( low, high ) );
38         Console.WriteLine( " " + Subarray( low, middle1 ) );
39         Console.WriteLine( " " + Subarray( middle2, high ) );
40         Console.WriteLine();
41         // Массив разбивается надвое; каждая половина
42         // сортируется рекурсивным вызовом
43         SortArray( low, middle1 ); // Первая половина массива
44         SortArray( middle2, high ); // Вторая половина массива
45
46         // Слияние двух отсортированных подмассивов
47         Merge( low, middle1, middle2, high );
48     } // Конец if
49 } // Конец метода SortArray
50
51 // Слияние двух отсортированных подмассивов
52 private void Merge( int left, int middle1, int middle2, int right )
53 {
54     int leftIndex = left; // Индекс в левом подмассиве
55     int rightIndex = middle2; // Индекс в правом подмассиве
56     int combinedIndex = left; // Индекс во временном рабочем массиве
57     int[] combined = new int[ data.Length ]; // Рабочий массив
58
59     // Вывод двух подмассивов перед слиянием
60     Console.WriteLine( "merge: " + Subarray( left, middle1 ) );
61     Console.WriteLine( " " + Subarray( middle2, right ) );

```

**Ил. 18.10.** Класс с массивом случайных целых чисел и методом для выполнения сортировки слиянием (продолжение ☞)



```

62
63 // Слияние массивов до достижения конца любого из них
64 while ( leftIndex <= middle1 && rightIndex <= right )
65 {
66     // Размещение меньшего из двух текущих элементов в массиве
67     // результата и переход к следующей позиции в массивах
68     if ( data[ leftIndex ] <= data[ rightIndex ] )
69         combined[ combinedIndex++ ] = data[ leftIndex++ ];
70     else
71         combined[ combinedIndex++ ] = data[ rightIndex++ ];
72 } // Конец while
73
74 // Если левый массив пуст
75 if ( leftIndex == middle2 )
76     // Копирование остатка правого массива
77     while ( rightIndex <= right )
78         combined[ combinedIndex++ ] = data[ rightIndex++ ];
79 else // Правый массив пуст
80     // Копирование остатка левого массива
81     while ( leftIndex <= middle1 )
82         combined[ combinedIndex++ ] = data[ leftIndex++ ];
83
84 // Копирование значений обратно в исходный массив
85 for ( int i = left; i <= right; ++i )
86     data[ i ] = combined[ i ];
87
88 // Вывод массива после слияния
89 Console.WriteLine( " " + Subarray( left, right ) );
90 Console.WriteLine();
91 } // Конец метода Merge
92
93 // Метод для вывода подмассива
94 public string Subarray( int low, int high )
95 {
96     string temporary = string.Empty;
97
98     // Вывод пробелов для выравнивания
99     for ( int i = 0; i < low; ++i )
100         temporary += " ";
101
102     // Вывод элементов, оставшихся в массиве
103     for ( int i = low; i <= high; ++i )
104         temporary += " " + data[ i ];
105
106     return temporary;
107 } // Конец метода Subarray
108
109 // Метод для вывода значений из массива
110 public override string ToString()
111 {
112     return Subarray( 0, data.Length - 1 );
113 } // Конец метода ToString
114 } // Конец класса MergeSort

```

**Ил. 18.10.** Класс с массивом случайных целых чисел и методом для выполнения сортировки слиянием (окончание)

Метод `SortArray` объявляется в строках 28–49. Строка 31 проверяет базовый случай. Если размер массива равен 1, то массив уже отсортирован, поэтому метод просто возвращает управление. Если размер массива больше 1, то метод разбивает массив надвое и рекурсивно вызывает метод `SortArray` для сортировки двух подмассивов и их слияния. Строка 43 рекурсивно вызывает метод `SortArray` для первой половины массива, а строка 44 рекурсивно вызывает метод `SortArray` для второй половины. При возврате управления из этих двух методов каждая половина массива отсортирована. Строка 47 вызывает метод `Merge` (строки 52–91) для двух половин массива, чтобы объединить два отсортированных подмассива в один большой отсортированный массив.

Строки 64–72 метода `Merge` выполняются в цикле до тех пор, пока приложение не достигнет конца одного из подмассивов. Строка 68 проверяет, какой из элементов в начале массива меньше. Если элемент левого массива меньше элемента правого подмассива или равен ему, то в строке 69 он помещается в позицию объединенного массива. Если элемент правого массива меньше, то он помещается в объединенный массив в строке 71. После завершения цикла `while` (строка 72) один полный подмассив находится в объединенном массиве, но другой подмассив все еще содержит данные. Строка 75 проверяет, достигнут ли конец левого массива. При достижении конца массива строки 77–78 заполняют объединенный массив оставшимися элементами правого массива. Если левый массив еще не закончился, значит, закончился правый массив, и строки 81–82 заполняют объединенный массив оставшимися элементами левого массива. Наконец, строки 85–86 копируют объединенный массив в исходный. В листинге на ил. 18.11 представлен пример создания и использования объекта `MergeSort`. В результатах приложения показаны разбиения и слияния, выполняемые в ходе работы алгоритма, с ходом сортировки на каждом шаге.

```
1 // Ил. 18.11: MergeSortTest.cs
2 // Тестирование класса сортировки слиянием.
3 using System;
4
5 public class MergeSortTest
6 {
7     public static void Main( string[] args )
8     {
9         // Создание объекта для выполнения сортировки слиянием
10        MergeSort sortArray = new MergeSort( 10 );
11
12        // Вывод несортированного массива
13        Console.WriteLine( "Unsorted: {0}\n\n", sortArray );
14
15        sortArray.Sort(); // Сортировка массива
16
17        // Вывод отсортированного массива
18        Console.WriteLine( "Sorted: {0}", sortArray );
19    } // Конец Main
20 } // Конец класса MergeSortTest
```

**Ил. 18.11.** Тестирование реализации сортировки слиянием (продолжение ➤)

```

Unsorted: 36 38 81 93 85 72 31 11 33 74

split: 36 38 81 93 85 72 31 11 33 74
      36 38 81 93 85
                72 31 11 33 74

split: 36 38 81 93 85
      36 38 81
            93 85

split: 36 38 81
      36 38
            81

split: 36 38
      36
            38

merge: 36
       38
      36 38
merge: 36 38
       81
      36 38 81

split:          93 85
          93
              85
merge:          93
              85
          85 93

merge: 36 38 81
       85 93
      36 38 81 85 93

split:          72 31 11 33 74
          72 31 11
                  33 74

split:          72 31 11
          72 31
                  11

split:          72 31
          72
              31

merge:          72
              31
          31 72

merge:          31 72
              11
          11 31 72

split:          33 74
              33
                  74

```

**Ил. 18.11.** Тестирование реализации сортировки слиянием (продолжение ↗)

```

merge:                33
                        74
                        33 74
merge:             11 31 72
                        33 74
                        11 31 33 72 74
merge:  36 38 81 85 93
                        11 31 33 72 74
                        11 31 33 36 38 72 74 81 85 93

Sorted: 11 31 33 36 38 72 74 81 85 93

```

**Ил. 18.11.** Тестирование реализации сортировки слиянием (окончание)

### Эффективность сортировки слиянием

Сортировка слиянием значительно превосходит по эффективности сортировку выбором и вставкой при обработке больших наборов данных. Возьмем первый (нерекурсивный) вызов метода `SortArray`. Он порождает два рекурсивных вызова метода `SortArray` с подмассивами, размер каждого из которых равен примерно половине исходного массива, и один вызов метода `Merge`. Вызов `Merge` требует в худшем случае  $n - 1$  сравнения для заполнения исходного массива, то есть обладает вычислительной сложностью  $O(n)$ . (Вспомните, что каждый элемент массива выбирается сравнением одного элемента из каждого подмассива.) Два вызова метода `SortArray` порождают четыре рекурсивных вызова `SortArray`, каждый из которых работает примерно с четвертью исходного массива, и два вызова `Merge`. Каждый из двух вызовов `Merge` требует в худшем случае  $n/2 - 1$  сравнения, а общее количество сравнений составляет  $(n/2 - 1) + (n/2 - 1) = n - 2$ , что соответствует сложности  $O(n)$ . С продолжением процесса каждый вызов `SortArray` генерирует два дополнительных вызова `SortArray` и вызов `Merge`, пока алгоритм не разобьет массив на подмассивы с одним элементом. На каждом уровне для слияния подмассивов требуются  $O(n)$  сравнений. Каждый уровень уменьшает размер массивов вдвое, так что удвоение размера массива потребует одного уровня, увеличение в четыре раза — двух уровней, и т. д. Это логарифмическая закономерность, а количество уровней равно  $\log_2 n$  уровней. В итоге мы приходим к вычислительной сложности  $O(n \log n)$ .

## 18.4. Сводка эффективности алгоритмов поиска и сортировки

На ил. 18.12 представлена сводка алгоритмов сортировки и поиска, описанных в книге, с указанием вычислительной сложности каждого алгоритма. В таблице на ил. 18.13 перечислены разные варианты сложности алгоритмов с примерами значений  $n$ , демонстрирующими различия в скорости роста сложности.

Алгоритм	Раздел	Вычислительная сложность
Алгоритмы поиска		
Линейный поиск	Раздел 18.2.1	$O(n)$
Бинарный поиск	Раздел 18.2.2	$O(\log n)$
Алгоритмы сортировки		
Сортировка выбором	Раздел 18.3.1	$O(n^2)$
Сортировка вставкой	Раздел 18.3.2	$O(n^2)$
Рекурсивная сортировка слиянием	Раздел 18.3.3	$O(n \log n)$

**Ил. 18.12.** Вычислительная сложность алгоритмов сортировки и поиска

n	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	16
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10 000
1000	3	1000	3000	$10^6$
1 000 000	6	1 000 000	6 000 000	$10^{12}$
1 000 000 000	9	1 000 000 000	9 000 000 000	$10^{18}$

**Ил. 18.13.** Количество сравнений для разных вариантов сложности

## 18.5. Итоги

Из этой главы вы узнали, как проводится поиск в массивах и как решается задача сортировки массива для размещения его элементов в определенном порядке. Мы рассмотрели линейный и бинарный поиск, а также сортировку выбором, вставкой и слиянием. Линейный поиск работает с любыми наборами данных, а бинарный поиск требует предварительной сортировки данных. Также в этой главе было показано, что простейшие алгоритмы сортировки и поиска могут обладать низким быстродействием. Для сравнения эффективности рассматриваемых алгоритмов используется запись «большого O». В следующей главе будут рассмотрены динамические структуры данных, которые могут расширяться или сокращаться во время выполнения.

# 19 Структуры данных

## 19.1. Введение

В этой главе продолжится наш курс алгоритмов и структур данных. Большинство структур данных, рассмотренных до настоящего момента, обладало фиксированным размером — как, например, одно- и двумерные массивы. В главе 9 была представлена коллекция `List<T>` с динамически изменяемым размером. В этой главе мы продолжим знакомство с динамическими структурами данных, размер которых уменьшается и увеличивается во время выполнения. *Связанные списки* представляют наборы «сцепленных» данных — пользователь может выполнять вставку и удаление в любой позиции связанного списка. *Стеки* играют важную роль в работе компиляторов и операционных систем; операции вставки и удаления могут выполняться только с одного конца — на вершине стека. В *очередях* вставка осуществляется в конце, а удаление — в начале. *Бинарные деревья* упрощают скоростной поиск и сортировку данных и эффективное удаление дубликатов; они используются для представления файловых систем и компиляция выражений на машинный язык. Эти структуры данных также находят немало других интересных применений.

В этой главе будут рассмотрены все эти структуры данных, а также написаны программы для их создания и обработки. Классы, наследование и композиция, применяемые при создании и упаковке этих структур данных, упростят повторное использование и сопровождение. В главе 20 будет представлен механизм *обобщенных типов* для объявления структур данных, автоматически адаптируемых к данным произвольного типа. В главе 21 рассматриваются заранее определенные классы коллекций C#, реализующие различные структуры данных.

Примеры этой главы представляют собой вполне реальные программы, которые могут пригодиться в учебных курсах более высокого уровня и в коммерческих приложениях. Основной упор делается на операции со ссылками.

## 19.2. Структуры простых типов, упаковка и распаковка

В структурах данных, описанных в этой главе, хранятся ссылки на объекты. Однако как вы вскоре увидите, в структурах данных могут храниться значения как простых,

так и ссылочных типов. В этом разделе описаны механизмы, позволяющие работать со значениями простых типов как с объектами.

### Структуры простых типов

У каждого простого типа имеется соответствующая структура в пространстве имен `System`. Это структуры `Boolean`, `Byte`, `SByte`, `Char`, `Decimal`, `Double`, `Single`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64` и `UInt64`. Типы, объявляемые с ключевым словом `struct`, называются *значимыми типами*.

Простые типы в действительности являются синонимами для соответствующих структур, так что переменная простого типа может объявляться как с ключевым словом этого простого типа, так и с именем структуры — например, `int` и `Int32` являются взаимозаменяемыми. Методы, связанные с простым типом, находятся в соответствующей структуре (например, метод `Parse`, преобразующий `string` в `int`, находится в структуре `Int32`). За информацией о методах, доступных для работы со значениями структурного типа, обращайтесь к документации этого типа.

### Упаковка и распаковка

Простые типы и другие структуры являются производными от класса `ValueType` в пространстве имен `System`. Класс `ValueType` является производным от класса `object`. Таким образом, каждый простой тип может быть присвоен переменной `object`; это преобразование, называемое *упаковкой* (`boxing`), позволяет использовать простые типы там, где ожидается тип `object`. При упаковке значение простого типа копируется в `object`, чтобы с ним можно было работать как с `object`. Упаковка может выполняться либо явно, либо неявно с использованием следующих команд:

```
int i = 5; // Создание значения int
object object1 = ( object ) i; // Явная упаковка значения int
object object2 = i; // Неявная упаковка значения int
```

После выполнения предыдущего кода `object1` и `object2` ссылаются на два разных объекта, каждый из которых содержит копию значения переменной `i` типа `int`.

*Распаковка* (`unboxing`) применяется для явного преобразования ссылки на `object` в значение простого типа:

```
int int1 = ( int ) object1; // Явная распаковка значения int
```

Попытка явной распаковки ссылки на `object`, которая не ссылается на правильное значение простого типа, приводит к исключению `InvalidCastException`.

В главах 20 и 21 рассматриваются обобщенные типы и коллекции `C#`; вы узнаете, как создавать структуры данных конкретных значимых типов, для работы с которыми не нужны преобразования упаковки и распаковки.

## 19.3. Самоотносимые классы

*Самоотносимый* (`self-referential`) класс содержит ссылочную переменную, которая ссылается на объект того же класса. Например, объявление класса на ил. 19.1

определяет основу самоотносимого класса с именем `Node`. Объявляемый тип содержит два свойства: `intData` и `Next`. Свойство `Next` содержит ссылку на `Node` — объект того же типа, в котором это свойство объявлено; отсюда и название «самоотносимый класс».

```

1 // Ил. 19.1: Fig19_01.cs
2 // Объявление класса Node.
3 class Node
4 {
5     public int Data { get; set; } // Для целочисленных данных
6     public Node Next { get; set; } // Для ссылки на следующий объект Node
7
8     public Node( int dataValue )
9     {
10         Data = dataValue;
11     } // Конец конструктора
12 } // Конец класса Node

```

**Ил. 19.1.** Объявление класса `Node`

Самоотносимые объекты связываются друг с другом ссылками, образуя полезные структуры данных: списки, очереди, стеки и деревья. На ил. 19.2 представлены два объекта, образующих связанный список. Обратная косая черта (представляющая `null`-ссылку) во втором объекте всего лишь означает, что ссылка не указывает на другой объект; она не имеет отношения к символу `\` в языке `C#`. Как правило, `null`-ссылка обозначает конец структуры данных.



#### ТИПИЧНАЯ ОШИБКА 19.1

Отличная от `null` ссылка в последнем узле структуры является логической ошибкой.



**Ил. 19.2.** Связывание объектов самоотносимого класса

Создание и поддержание динамических структур данных требует динамического управления памятью, то есть выделения дополнительной памяти для хранения новых узлов и ее освобождения, когда надобность в ней отпадет. Как было показано в разделе 10.8, программы `C#` не выполняют явного освобождения динамически выделенной памяти — вместо этого среда `CLR` выполняет автоматическую уборку мусора.

Оператор `new` занимает центральное место в динамическом выделении памяти. В операнде `new` передается тип объекта, для которого выполняется динамическое выделение памяти. Оператор возвращает ссылку на объект указанного типа. Например, команда

```
Node nodeToAdd = new Node( 10 );
```

выделяет блок памяти, необходимый для хранения объекта `Node`, инициализирует его и присваивает ссылку на этот объект переменной `nodeToAdd`. Если выделить



память не удалось, `new` выдает исключение `OutOfMemoryException`. В аргументе конструктора 10 передаются данные объекта `Node`.

В следующих разделах рассматриваются списки, стеки, очереди и деревья. При создании и сопровождении этих структур данных используется динамическое выделение памяти и самоотносимые классы.

## 19.4. Связанные списки

Связанный список представляет собой линейную коллекцию (то есть последовательность) объектов самоотносимого класса, называемых *узлами* (*nodes*), связанных по ссылкам. Программа обращается к связанному списку по ссылке на первый узел. Переход к каждому следующему узлу осуществляется по ссылке, хранящейся в предыдущем узле. Ссылка в последнем узле списка инициализируется значением `null`, отмечающим конец списка. Данные сохраняются в связанном списке динамически, то есть узлы создаются по мере необходимости. Узел может содержать данные любого типа, в том числе и ссылки на объекты других классов. Стеки и очереди тоже относятся к линейным структурам данных; более того, они могут рассматриваться как специализированные реализации связанных списков. Деревья являются нелинейными структурами данных.

Списки данных можно хранить в массивах, но связанные списки обладают некоторыми преимуществами. Связанный список хорошо подходит в тех ситуациях, в которых количество элементов данных неизвестно заранее. В отличие от размеров связанных списков, размеры традиционных массивов `C#` не могут изменяться, потому что размер массива фиксируется в момент его создания. Традиционный массив может переполниться, но связанные списки переполняются только в том случае, если в системе недостаточно памяти для удовлетворения запроса на динамическое выделение памяти.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 19.1

Массив можно объявить с размером, заведомо превышающим количество реально хранимых элементов, но это приведет к неэффективному использованию памяти. Связанные списки в таких ситуациях более эффективно расходуют память, потому что они могут расширяться и сокращаться во время выполнения.

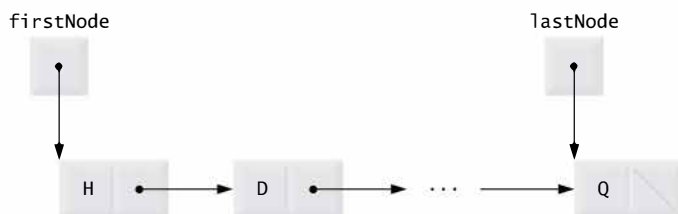
Программист может поддерживать элементы связанного списка в порядке сортировки, для чего каждый новый элемент вставляется в подходящую позицию списка (поиск правильной позиции вставки занимает некоторое время). Перемещать существующие элементы списка при этом не нужно.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 19.2

Элементы массива хранятся в непрерывной области памяти, чтобы программа могла мгновенно обратиться к любому элементу — адрес элемента вычисляется по его индексу. Связанные списки не предоставляют прямого доступа к своим элементам; чтобы обратиться к узлу связанного списка, необходимо перейти к нему по ссылкам от начала списка.

В общем случае узлы связанных списков не занимают непрерывную область памяти. На ил. 19.3 изображен связанный список с несколькими узлами.



**Ил. 19.3.** Графическое представление связанного списка



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 19.3

Использование связанных структур данных и динамического выделения памяти (вместо массивов) может сэкономить память. Однако помните, что для хранения ссылок тоже необходима память, а динамическое выделение памяти связано с затратами на вызовы методов.

## Реализация связанного списка

На ил. 19.4–19.5 экземпляр класса `List` используется для работы со списком объектов. Метод `Main` класса `ListTest` (см. ил. 19.5) создает список объектов, вставляет объекты в начало списка методом `InsertAtFront` класса `List`, вставляет объекты в конец списка методом `InsertAtBack` класса `List`, удаляет объекты в начале списка методом `RemoveFromFront` класса `List` и удаляет объекты в конце списка методом `RemoveFromBack` класса `List`. После каждой операции вставки или удаления программа вызывает метод `Display` класса `List` для вывода текущего содержимого списка. Если программа делает попытку удалить элемент из пустого списка, происходит исключение `EmptyListException`. Подробное описание кода приводится ниже.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 19.4

Затраты времени на вставку и удаление в отсортированном массиве могут быть довольно значительными — все элементы, находящиеся за вставляемым или удаляемым элементом, должны сдвигаться соответствующим образом.

Программа состоит из четырех классов — `ListNode` (см. ил. 19.4, строки 8–30), `List` (строки 33–147), `EmptyListException` (строки 150–172) и `ListTest` (см. ил. 19.5). Классы на ил. 19.4 образуют библиотеку для работы со связанными списками (определяемую в пространстве имен `LinkedListLibrary`), которая может использоваться повторно в дальнейших примерах. Поместите код с ил. 19.4 в отдельный проект библиотеки классов (см. раздел 15.13).

```
1 // Ил. 19.4: LinkedListLibrary.cs
2 // Объявления классов ListNode, List и EmptyListException.
3 using System;
```

**Ил. 19.4.** Объявления классов `ListNode`, `List` и `EmptyListException` (продолжение ➤)

```

4
5 namespace LinkedListLibrary
6 {
7     // Класс для представления узла списка
8     class ListNode
9     {
10         // Автоматически реализуемое свойство Data (только для чтения)
11         public object Data { get; private set; }
12
13         // Автоматически реализуемое свойство Next
14         public ListNode Next { get; set; }
15
16         // Конструктор для создания объекта ListNode со ссылкой
17         // на dataValue, который является последним узлом в списке
18         public ListNode( object dataValue )
19             : this( dataValue, null )
20         {
21         } // Конец конструктора по умолчанию
22
23         // Конструктор для создания объекта ListNode со ссылками
24         // на dataValue и на следующий объект ListNode в списке
25         public ListNode( object dataValue, ListNode nextNode )
26         {
27             Data = dataValue;
28             Next = nextNode;
29         } // Конец конструктора
30     } // Конец класса ListNode
31
32     // Объявление класса List
33     public class List
34     {
35         private ListNode firstNode;
36         private ListNode lastNode;
37         private string name; // Строковое имя (например, "list")
38
39         // Конструирование пустого объекта List с заданным именем
40         public List( string listName )
41         {
42             name = listName;
43             firstNode = lastNode = null;
44         } // Конец конструктора
45
46         // Конструирование пустого объекта List с именем "list"
47         public List()
48             : this( "list" )
49         {
50         } // Конец конструктора по умолчанию
51
52         // Вставка объектов в начале списка. Если список пуст,
53         // firstNode и lastNode содержат ссылку на один и тот же объект.
54         // В противном случае firstNode содержит ссылку на новый узел.
55         public void InsertAtFront( object insertItem )
56         {
57             if ( IsEmpty() )

```

**Ил. 19.4.** Объявления классов ListNode, List и EmptyListException (продолжение ↗)

```

58         firstNode = lastNode = new ListNode( insertItem );
59     else
60         firstNode = new ListNode( insertItem, firstNode );
61 } // Конец метода InsertAtFront
62
63 // Вставка объектов в конце списка. Если список пуст, firstNode
64 // и lastNode содержат ссылку на один и тот же объект. В противном
65 // случае свойство lastNode.Next содержит ссылку на новый узел.
66 public void InsertAtBack( object insertItem )
67 {
68     if ( IsEmpty() )
69         firstNode = lastNode = new ListNode( insertItem );
70     else
71         lastNode = lastNode.Next = new ListNode( insertItem );
72 } // Конец метода InsertAtBack
73
74 // Удаление первого узла из списка
75 public object RemoveFromFront()
76 {
77     if ( IsEmpty() )
78         throw new EmptyListException( name );
79
80     object removeItem = firstNode.Data; // Сохранение данных
81
82     // Сброс ссылок firstNode и lastNode
83     if ( firstNode == lastNode )
84         firstNode = lastNode = null;
85     else
86         firstNode = firstNode.Next;
87
88     return removeItem; // Возвращение удаленных данных
89 } // Конец метода RemoveFromFront
90
91 // Удаление последнего узла из списка
92 public object RemoveFromBack()
93 {
94     if ( IsEmpty() )
95         throw new EmptyListException( name );
96
97     object removeItem = lastNode.Data; // Сохранение данных
98
99     // Сброс ссылок firstNode и lastNode
100    if ( firstNode == lastNode )
101        firstNode = lastNode = null;
102    else
103    {
104        ListNode current = firstNode;
105
106        // Выполнять, пока значение current.Next отлично от lastNode
107        while ( current.Next != lastNode )
108            current = current.Next; // Переход к следующему узлу
109
110        // current становится новым значением lastNode
111        lastNode = current;

```

**Ил. 19.4.** Объявления классов ListNode, List и EmptyListException (продолжение ↗)

```

112         current.Next = null;
113     } // Конец else
114
115     return removeItem; // Возвращение удаленных данных
116 } // Конец метода RemoveFromBack
117
118 // Метод возвращает true, если список пуст
119 public bool IsEmpty()
120 {
121     return firstNode == null;
122 } // Конец метода IsEmpty
123
124 // Вывод содержимого списка
125 public void Display()
126 {
127     if ( IsEmpty() )
128     {
129         Console.WriteLine( "Empty " + name );
130     } // Конец if
131     else
132     {
133         Console.Write( "The " + name + " is: " );
134
135         ListNode current = firstNode;
136
137         // Вывод данных текущего узла в цикле до конца списка
138         while ( current != null )
139         {
140             Console.Write( current.Data + " " );
141             current = current.Next;
142         } // Конец while
143
144         Console.WriteLine( "\n" );
145     } // Конец else
146 } // Конец метода Display
147 } // Конец класса List
148
149 // Объявление класса EmptyListException
150 public class EmptyListException : Exception
151 {
152     // Конструктор без параметров
153     public EmptyListException()
154         : base( "The list is empty" )
155     {
156         // Пустой конструктор
157     } // Конец конструктора EmptyListException
158
159     // Конструктор с одним параметром
160     public EmptyListException( string name )
161         : base( "The " + name + " is empty" )
162     {
163         // Пустой конструктор
164     } // Конец конструктора EmptyListException

```

**Ил. 19.4.** Объявления классов ListNode, List и EmptyListException (продолжение ↗)

```
165
166     // Конструктор с двумя параметрами
167     public EmptyListException( string exception, Exception inner )
168         : base( exception, inner )
169     {
170         // Пустой конструктор
171     } // Конец конструктора EmptyListException
172 } // Конец класса EmptyListException
173 } // Конец пространства имен LinkedListLibrary
```

**Ил. 19.4.** Объявления классов `ListNode`, `List` и `EmptyListException` (окончание)

### Класс `ListNode`

В каждом объекте `List` инкапсулируется связанный список объектов `ListNode`. Класс `ListNode` (см. ил. 19.4, строки 8–30) содержит два свойства: `Data` и `Next`. Свойство `Data` может ссылаться на произвольный объект. [*Примечание:* как правило, структура данных содержит данные только *одного* типа или данные любых типов, производных от общего базового типа.] В нашем примере используются данные разных типов, производных от `object`; это показывает, что класс `List` может хранить данные любого типа. В `Next` хранится ссылка на следующий объект `ListNode` в связанном списке. Конструкторы `ListNode` (строки 18–21 и 25–29) инициализируют объект `ListNode`, который будет помещен в конец связанного списка или перед конкретным объектом `ListNode` в списке.

### Класс `List`

Класс `List` (строки 33–147) содержит закрытые переменные экземпляров `firstNode` (ссылка на первый объект `ListNode` в списке) и `lastNode` (ссылка на последний объект `ListNode` в списке). Конструкторы (строки 40–44 и 47–50) инициализируют обе ссылки значениями `null` и позволяют задать имя объекта `List` для вывода. Основные методы класса `List` — `InsertAtFront` (строки 55–61), `InsertAtBack` (строки 66–72), `RemoveFromFront` (строки 75–89) и `RemoveFromBack` (строки 92–116). Метод `IsEmpty` (строки 119–122) является предикатным методом, который проверяет, пуст ли список (то есть ссылка на первый узел списка равна `null`). Как правило, предикатные методы проверяют некоторое условие и не изменяют объект, для которого они вызываются. Если список пуст, то метод `IsEmpty` возвращает `true`; в противном случае возвращается `false`. Метод `Display` (строки 125–146) выводит содержимое списка. Методы класса `List` подробно описаны после ил. 19.5.

### Класс `EmptyListException`

Класс `EmptyListException` (строки 150–172) определяет исключение, которое будет использоваться для обозначения некорректных операций с пустым объектом `List`.

### Класс `ListTest`

Класс `ListTest` (см. ил. 19.5) использует библиотеку для создания связанного списка и работы с ним. [*Примечание:* не забудьте добавить в проект на ил. 19.5 ссылку на библиотеку, содержащую классы на ил. 19.4. Если вы используете готовый пример, возможно, вам придется обновить эту ссылку.] В строке 11 создается новый

объект `List`, который присваивается переменной `list`. В строках 14–17 создаются данные, добавляемые в список. В строках 20–27 методы вставки используются для включения данных в список; после каждой вставки метод `Display` выводит текущее содержимое списка. Значения переменных простых типов неявно упаковываются в строках 20, 22 и 24, где программа ожидает получить ссылки на объект. Код в блоке `try` (строки 33–50) удаляет объекты методами класса `List`, выводит каждый удаленный объект и выводит список после каждого удаления. При попытке удаления объекта из пустого списка блок `catch` в строках 51–54 перехватывает `EmptyListException` и выводит сообщение об ошибке.

```

1 // Ил. 19.5: ListTest.cs
2 // Тестирование класса List.
3 using System;
4 using LinkedListLibrary;
5
6 // Класс для тестирования функциональности класса List.
7 class ListTest
8 {
9     public static void Main( string[] args )
10    {
11        List list = new List(); // Создание контейнера List
12
13        // Создание данных для хранения в списке
14        bool aBoolean = true;
15        char aCharacter = '$';
16        int anInteger = 34567;
17        string aString = "hello";
18
19        // Использование методов вставки
20        list.InsertAtFront( aBoolean );
21        list.Display();
22        list.InsertAtFront( aCharacter );
23        list.Display();
24        list.InsertAtBack( anInteger );
25        list.Display();
26        list.InsertAtBack( aString );
27        list.Display();
28
29        // Использование методов удаления
30        object removedObject;
31
32        // Удаление данных и вывод списка после каждого удаления
33        try
34        {
35            removedObject = list.RemoveFromFront();
36            Console.WriteLine( removedObject + " removed" );
37            list.Display();
38
39            removedObject = list.RemoveFromFront();
40            Console.WriteLine( removedObject + " removed" );
41            list.Display();
42
43            removedObject = list.RemoveFromBack();

```

**Ил. 19.5.** Тестирование класса `List` (продолжение ➞)

```
44         Console.WriteLine( removedObject + " removed" );
45         list.Display();
46
47         removedObject = list.RemoveFromBack();
48         Console.WriteLine( removedObject + " removed" );
49         list.Display();
50     } // Конец try
51     catch ( EmptyListException emptyListException )
52     {
53         Console.Error.WriteLine( "\n" + emptyListException );
54     } // Конец catch
55 } // Конец Main
56 } // Конец класса ListTest
```

```
The list is: True
The list is: $ True
The list is: $ True 34567
The list is: $ True 34567 hello
$ removed
The list is: True 34567 hello
True removed
The list is: 34567 hello
hello removed
The list is: 34567
34567 removed
Empty list
```

**Ил. 19.5.** Тестирование класса List (окончание)

### Метод InsertAtFront

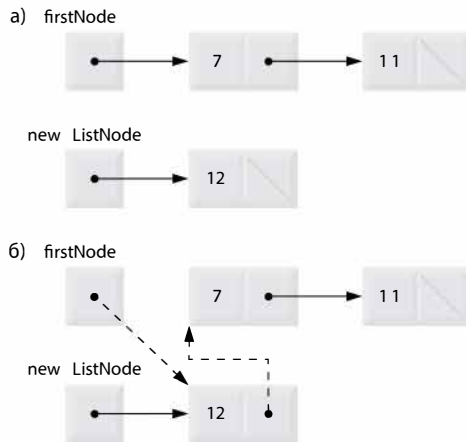
На нескольких следующих страницах подробно рассматриваются методы класса List. Метод InsertAtFront (см. ил. 19.4, строки 55–61) включает новый узел в начало списка. Его выполнение состоит из трех шагов:

1. Вызов IsEmpty проверяет, пуст ли список (строка 57).
2. Если список пуст, firstNode и lastNode присваиваются ссылки на новый объект ListNode, инициализированный insertItem (строка 58). Конструктор ListNode в строках 18–21 вызывает конструктор ListNode в строках 25–29, который задает свойству Data ссылку на объект, переданный в первом аргументе, и сбрасывает ссылку в свойстве Next в null.
3. Если список не пуст, новый узел «подключается» к списку; для этого firstNode задается ссылка на новый объект ListNode, инициализированный значениями insertItem и firstNode (строка 60). При выполнении конструктора ListNode (строки 25–29) свойству Data задается ссылка на объект, переданный в первом



аргументе, а вставка выполняется заданием `Next` ссылки на объект `ListNode`, переданный во втором аргументе.

На ил. 19.6, *а* изображены список и новый узел во время операции `InsertAtFront` перед включением нового узла в список. Пунктирные линии и стрелки в части *б* обозначают шаг 3 операции `InsertAtFront`, в результате которого узел со значением 12 становится новым началом списка.



**Ил. 19.6.** Операция `InsertAtFront`



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 19.5

После того как точка вставки нового элемента в отсортированный связанный список будет найдена, добавление элемента в список выполняется быстро — для этого достаточно изменить всего две ссылки. Все существующие узлы остаются в памяти на своих местах.

### Метод `InsertAtBack`

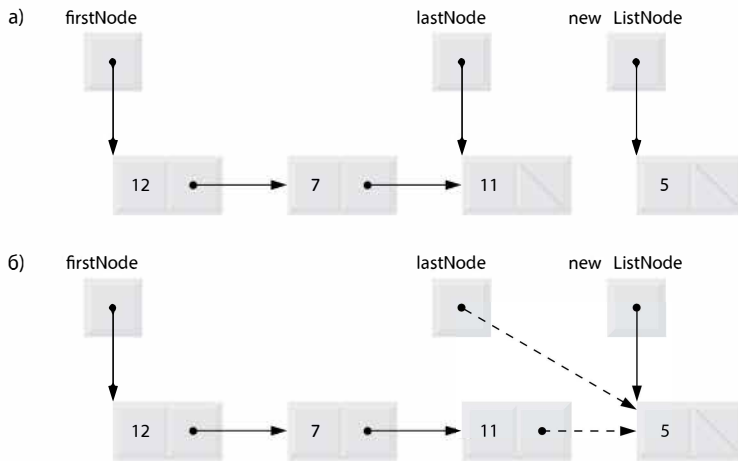
Метод `InsertAtBack` (см. ил. 19.4, строки 66–72) добавляет новый узел в конец списка.

Метод состоит из трех шагов:

1. Вызов `IsEmpty` проверяет, пуст ли список (строка 68).
2. Если список пуст, `firstNode` и `lastNode` присваиваются ссылки на новый объект `ListNode`, инициализированный `insertItem` (строки 68–69). Конструктор `ListNode` в строках 18–21 вызывает конструктор `ListNode` в строках 25–29, который задает свойству `Data` ссылку на объект, переданный в первом аргументе, и задает ссылке в свойстве `Next` значение `null`.
3. Если список не пуст, новый узел добавляется в список; для этого `lastNode` и `lastNode.Next` задается ссылка на новый объект `ListNode`, инициализированный значением `insertItem` (строка 71). При выполнении конструктора `ListNode` (строки 18–21) вызывается конструктор в строках 25–29, который задает

свойству `Data` ссылку на объект, переданный в первом аргументе, а ссылке `Next` присваивает `null`.

На ил. 19.7, *а* изображены список и новый узел во время операции `InsertAtBack`, перед включением нового узла в список. Пунктирные линии и стрелки в части *б* обозначают шаг 3 операции `InsertAtBack`, в результате которого в конец непустого списка добавляется новый узел.



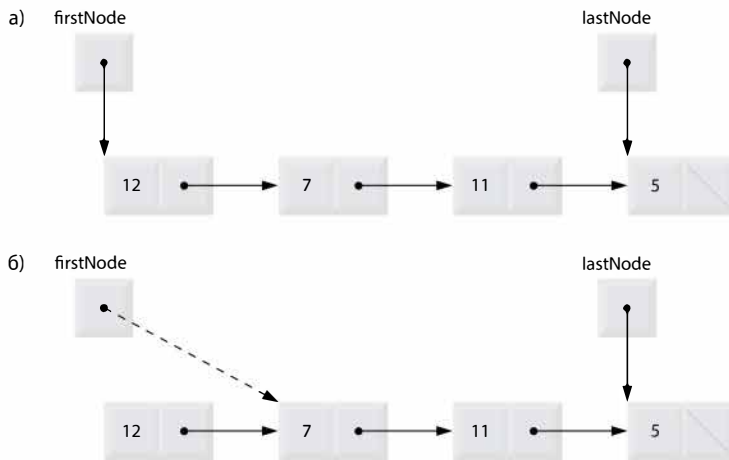
Ил. 19.7. Операция `InsertAtBack`

### Метод `RemoveFromFront`

Метод `RemoveFromFront` (см. ил. 19.4, строки 75–89) удаляет первый узел из списка и возвращает ссылку на данные удаленного узла. При попытке удаления узла из пустого списка метод выдает исключение `EmptyListException` (строка 78). После проверки пустого списка удаление первого узла состоит из четырех шагов:

1. Значение `firstNode.Data` (данные, удаляемые из списка) присваивается переменной `removeItem` (строка 80).
2. Если `firstNode` и `lastNode` ссылаются на *один* объект, то список содержит только один элемент, поэтому метод присваивает `firstNode` и `lastNode` значение `null` (строка 84) для удаления узла из списка (в результате чего список остается пустым).
3. Если список содержит более одного узла, метод оставляет ссылку `lastNode` без изменений и присваивает `firstNode` значение `firstNode.Next` (строка 86). Таким образом, `firstNode` ссылается на узел, который ранее был вторым в списке.
4. Метод возвращает ссылку на `removeItem` (строка 88).

На ил. 19.8, *а* изображен список до операции удаления. Пунктирные линии и стрелки в части *б* обозначают операции со ссылками.

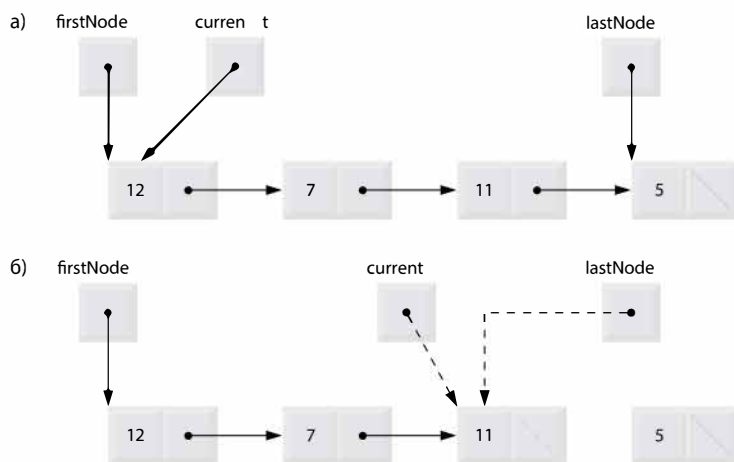
Ил. 19.8. Операция `RemoveFromFront`

### Метод `RemoveFromBack`

Метод `RemoveFromBack` (см. ил. 19.4, строки 92–116) удаляет последний узел из списка и возвращает ссылку на данные удаленного узла. При попытке удаления узла из пустого списка метод выдает исключение `EmptyListException` (строка 95). Метод состоит из следующих шагов:

1. Значение `lastNode.Data` (данные, удаляемые из списка) присваивается переменной `removeItem` (строка 97).
2. Если `firstNode` и `lastNode` ссылаются на *один* объект (строка 100), то список содержит только один элемент, поэтому метод присваивает `firstNode` и `lastNode` значение `null` (строка 101) для удаления узла из списка (в результате чего список остается пустым).
3. Если список содержит более одного узла, метод создает переменную `ListNode` с именем `current` и присваивает ей `firstNode` (строка 104).
4. Метод последовательно перебирает узлы в `current`, пока переменная не будет ссылаться на узел, предшествующий последнему. В цикле `while` (строки 107–108) `current` присваивается значение `current.Next`, пока оно не станет равным `lastNode`.
5. Когда предпоследний узел будет обнаружен, значение `current` присваивается `lastNode` (строка 111) для обновления информации о последнем узле в списке.
6. Свойству `current.Next` задается значение `null` (строка 112), чтобы последний узел был исключен из списка, а список завершался текущим узлом.
7. Метод возвращает ссылку на `removeItem` (строка 115).

На ил. 19.9, *a* изображен список до операции удаления. Пунктирные линии и стрелки в части *b* обозначают операции со ссылками.



Ил. 19.9. Операция RemoveFromBack

## Метод Display

Метод `Display` (см. ил. 19.4, строки 125–146) сначала проверяет, пуст ли список (строка 127). Если список пуст, `Display` выводит строку, состоящую из "Empty" и имени списка, после чего возвращает управление вызывающему методу. В противном случае `Display` выводит данные списка: строку, состоящую из префикса "The ", имени списка и строки "is: ". Затем строка 135 создает переменную `List-Node` с именем `current` и инициализирует ее значением `firstNode`. Пока переменная `current` не равна `null`, в списке еще остаются узлы. Соответственно метод выводит `current.Data` (строка 140), после чего присваивает `current` значение `current.Next` (строка 141) для перехода к следующему узлу в списке.

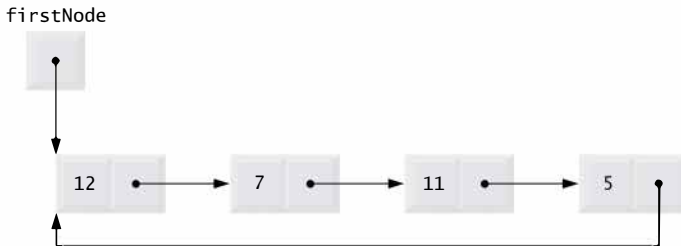
## Линейные и циклические односвязные и двусвязные списки

Связанные списки, которые рассматривались до настоящего момента, были односвязными — они начинались со ссылки на первый узел, а каждый узел содержал ссылку на следующий узел в последовательности. Такой список завершается узлом, ссылка которого на следующий узел содержит `null`. Перемещение по односвязному списку возможно только в одном направлении.

Циклический односвязный список (ил. 19.10) начинается со ссылки на первый узел. Каждый узел содержит ссылку на следующий узел, но вместо `null`-ссылки «последний» узел содержит ссылку на первый узел; круг замыкается.

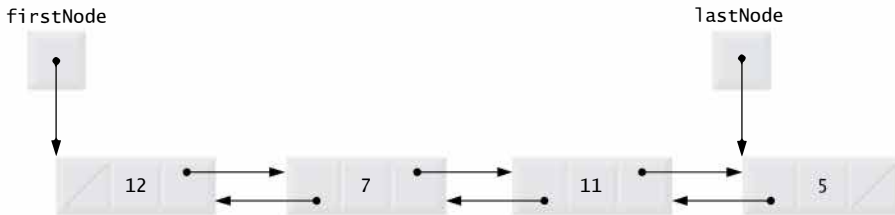
Двусвязный список (ил. 19.11) поддерживает перемещение как в прямом, так и в обратном направлении. Такой список часто реализуется с двумя «начальными ссылками» — одна указывает на первый элемент списка для перемещения в прямом направлении, а другая указывает на последний элемент для перемещения в обратном направлении. Каждый узел содержит прямую ссылку на следующий узел и обратную ссылку на предыдущий узел. Например, если список содержит

телефонный справочник, упорядоченный по алфавиту, поиск фамилии в начале алфавита лучше начать с начала списка. С другой стороны, поиск фамилий, начинающихся с последних букв, эффективнее начинать с конца.

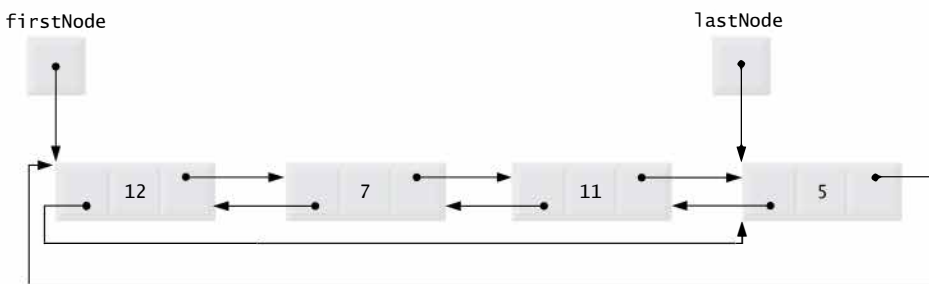


**Ил. 19.10.** Циклический односвязный список

В циклическом двусвязном списке (ил. 19.12) прямая ссылка последнего узла указывает на первый узел, а обратная ссылка последнего узла — на первый; таким образом круг замыкается.



**Ил. 19.11.** Двусвязный список



**Ил. 19.12.** Циклический двусвязный список

## 19.5. Стеки

*Стек* может рассматриваться как ограниченная разновидность связанного списка — операции вставки и удаления узлов выполняются только от начала списка.

Основные операции со стеком — *занесение* и *извлечение* элементов. Операция занесения в стек (push) добавляет новый узел на вершину стека. Операция извлечения (pop) удаляет узел с вершины стека и возвращает данные извлеченного узла.

Стеки имеют много интересных практических применений. Например, при вызове метода в программе вызванный метод должен каким-то образом передать управление на сторону вызова; для этого адрес возврата заносится в стек вызова. Если в программе происходит серия вызовов, последовательные адреса возврата заносятся в стек в порядке LIFO («последним пришел, первым вышел»), чтобы каждый метод мог вернуть управление вызывающей стороне. Стеки поддерживают рекурсивные вызовы точно так же, как и традиционные нерекурсивные вызовы.

Пространство имен `System.Collections` содержит класс `Stack`, реализующий стек с возможностью расширения и сокращения в процессе выполнения.

В нашем следующем примере тесная связь между списками и стеками используется для реализации класса стека на базе класса списка. Мы продемонстрируем две формы повторного использования: сначала класс стека будет реализован посредством наследования от класса `List` (см. ил. 19.4). Затем функционально эквивалентный класс стека будет реализован на базе композиции, для чего объект `List` будет включен в закрытую переменную класса стека.

### Реализация стека на базе наследования

На ил. 19.13 класс стека создается посредством наследования от класса `List` на ил. 19.4 (строка 8 на ил. 19.13). Класс стека должен содержать методы `Push`, `Pop`, `IsEmpty` и `Display`. В сущности, это методы `InsertAtFront`, `RemoveFromFront`, `IsEmpty` и `Display` класса `List`. Конечно, класс `List` содержит и другие методы (такие, как `InsertAtBack` и `RemoveFromBack`), которые не должны быть доступны через открытый интерфейс стека. Однако важно помнить, что все методы открытого интерфейса класса `List` также являются открытыми методами производного класса `StackInheritance` (см. ил. 19.13).

```
1 // Ил. 19.13: StackInheritanceLibrary.cs
2 // Реализация стека наследованием от класса List.
3 using LinkedListLibrary;
4
5 namespace StackInheritanceLibrary
6 {
7     // Класс StackInheritance наследует функциональность класса List.
8     public class StackInheritance : List
9     {
10         // Вызов конструктора List с именем "stack"
11         public StackInheritance()
12             : base( "stack" )
13         {
14             } // Конец конструктора
15
16         // Размещение dataValue на вершине стека посредством
17         // вставки dataValue в начало связанного списка
18         public void Push( object dataValue )
```

**Ил. 19.13.** Реализация стека наследованием от класса `List` (продолжение ↗)

```

19     {
20         InsertAtFront( dataValue );
21     } // Конец метода Push
22
23     // Удаление элемента на вершине стека посредством удаления
24     // элемента в начале связанного списка
25     public object Pop()
26     {
27         return RemoveFromFront();
28     } // Конец метода Pop
29 } // Конец класса StackInheritance
30 } // Конец пространства имен StackInheritanceLibrary

```

**Ил. 19.13.** Реализация стека наследованием от класса List (окончание)

Реализация каждого метода `StackInheritance` вызывает соответствующий метод `List` — метод `Push` вызывает `InsertAtFront`, метод `Pop` вызывает `RemoveFromFront`. Класс `StackInheritance` не определяет методы `IsEmpty` и `Display`, потому что `StackInheritance` наследует эти методы от класса `List` в своем открытом интерфейсе. Класс `StackInheritance` использует пространство имен `LinkedListLibrary` (см. ил. 19.4); следовательно, библиотека классов, определяющая `StackInheritance`, должна содержать ссылку на библиотеку `LinkedListLibrary`.

Метод `Main` класса `StackInheritanceTest` (ил. 19.14) использует класс `StackInheritance` для создания стека объектов с именем `stack` (строка 12). В строках 15–18 определяются четыре значения, которые будут заноситься в стек и извлекаться из него. Программа заносит в стек (строки 21, 23, 25 и 27) переменные `bool (true)`, `char ('$')`, `int (34567)` и `string ("hello")`. Бесконечный цикл `while` (строки 33–38) извлекает элементы из стека. Если стек пуст, метод `Pop` выдает исключение `EmptyListException`, а программа выводит трассировку стека на момент возникновения исключения. Программа использует метод `Display` (унаследованный `StackInheritance` от класса `List`) для вывода содержимого стека после каждой операции. Класс `StackInheritanceTest` использует пространство имен `LinkedListLibrary` (см. ил. 19.4) и пространство имен `StackInheritanceLibrary` (см. ил. 19.13); это означает, что в решение класса `StackInheritanceTest` должны быть включены ссылки на обе библиотеки.

```

1 // Ил. 19.14: StackInheritanceTest.cs
2 // Тестирование класса StackInheritance.
3 using System;
4 using StackInheritanceLibrary;
5 using LinkedListLibrary;
6
7 // Демонстрация функциональности класса StackInheritance
8 class StackInheritanceTest
9 {
10     public static void Main( string[] args )
11     {
12         StackInheritance stack = new StackInheritance();
13
14         // Создание объектов для сохранения в стеке
15         bool aBoolean = true;

```

**Ил. 19.14.** Тестирование класса `StackInheritance` (продолжение ➤)

```

16     char aCharacter = '$';
17     int anInteger = 34567;
18     string aString = "hello";
19
20     // Использование метода Push для добавления элементов в стек
21     stack.Push( aBoolean );
22     stack.Display();
23     stack.Push( aCharacter );
24     stack.Display();
25     stack.Push( anInteger );
26     stack.Display();
27     stack.Push( aString );
28     stack.Display();
29
30     // Извлечение элементов из стека
31     try
32     {
33         while ( true )
34         {
35             object removedObject = stack.Pop();
36             Console.WriteLine( removedObject + " popped" );
37             stack.Display();
38         } // Конец while
39     } // Конец try
40     catch ( EmptyListException emptyListException )
41     {
42         // Если произойдет исключение, вывести трассировку стека
43         Console.Error.WriteLine( emptyListException.StackTrace );
44     } // Конец catch
45 } // Конец Main
46 } // Конец класса StackInheritanceTest

```

The stack is: True

The stack is: \$ True

The stack is: 34567 \$ True

The stack is: hello 34567 \$ True

hello popped

The stack is: 34567 \$ True

34567 popped

The stack is: \$ True

\$ popped

The stack is: True

True popped

Empty stack

at LinkedListLibrary.List.RemoveFromFront()  
in C:\examples\ch21\Fig21\_04\LinkedListLibrary\  
LinkedListLibrary\LinkedListLibrary.cs:line 78

**Ил. 19.14.** Тестирование класса StackInheritance (продолжение ☞)



```

at StackInheritanceLibrary.StackInheritance.Pop()
in C:\examples\ch21\Fig21_13\StackInheritanceLibrary\
StackInheritanceLibrary\StackInheritance.cs:line 27
at StackInheritanceTest.Main(String[] args)
in C:\examples\ch21\Fig21_14\StackInheritanceTest\
StackInheritanceTest\StackInheritanceTest.cs:line 35

```

#### Ил. 19.14. Тестирование класса StackInheritance (окончание)

#### Класс стека, содержащий ссылку на List

Другой способ реализации класса стека основан на повторном использовании класса списка посредством композиции. Класс на ил. 19.15 использует закрытый объект класса List (строка 10) в объявлении класса StackComposition. Композиция позволяет скрыть методы класса List, которые не должны присутствовать в открытом интерфейсе стека; для этого в открытый интерфейс включаются только необходимые методы List. Класс реализует каждый метод стека, поручая его работу соответствующему методу List. Методы StackComposition вызывают методы класса List: InsertAtFront, RemoveFromFront, IsEmpty и Display. В этом примере класс StackCompositionTest не приводится, потому что он отличается от предыдущего только измененным именем класса стека (StackComposition вместо StackInheritance).

```

1 // Ил. 19.15: StackCompositionLibrary.cs
2 // Композиционное объявление StackComposition с внутренним объектом List.
3 using LinkedListLibrary;
4
5 namespace StackCompositionLibrary
6 {
7     // Класс StackComposition инкапсулирует функциональность List
8     public class StackComposition
9     {
10         private List stack;
11
12         // Конструирование пустого стека
13         public StackComposition()
14         {
15             stack = new List( "stack" );
16         } // Конец конструктора
17
18         // Добавление объекта в стек
19         public void Push( object dataValue )
20         {
21             stack.InsertAtFront( dataValue );
22         } // Конец метода Push
23
24         // Удаление объекта из стека
25         public object Pop()
26         {
27             return stack.RemoveFromFront();
28         } // Конец метода Pop

```

**Ил. 19.15.** Класс StackComposition инкапсулирует функциональность класса List (продолжение ↗)

```
29
30     // Проверка пустого стека
31     public bool IsEmpty()
32     {
33         return stack.IsEmpty();
34     } // Конец метода IsEmpty
35
36     // Вывод содержимого стека
37     public void Display()
38     {
39         stack.Display();
40     } // Конец метода Display
41 } // Конец класса StackComposition
42 } // Конец пространства имен StackCompositionLibrary
```

**Ил. 19.15.** Класс `StackComposition` инкапсулирует функциональность класса `List` (окончание)

## 19.6. Очереди

Другая часто используемая структура — очередь — напоминает очереди в магазинах; кассир первым обслуживает покупателя, стоящего в начале очереди. Другие покупатели встают в конец очереди и ожидают обслуживания. Узлы очереди удаляются только от начала и вставляются только в конец очереди. По этой причине очередь называется структурой данных FIFO (First In, First Out, то есть «первым пришел, первым обслужен»).

Очереди также часто применяются в компьютерных системах. В однопроцессорной системе в любой момент времени может выполняться только одно приложение. Каждое приложение, которому требуется процессорное время, помещается в очередь. Приложение, находящееся в начале очереди, следующим получает процессорное время. Чтобы получить доступ к процессору, все приложения постепенно перемещаются из конца в начало очереди.

Очереди также используются в системах печати. Например, один принтер может совместно использоваться всеми пользователями сети. Пользователи отправляют задания на печать, даже если принтер в настоящий момент занят. Задания печати помещаются в очередь до того момента, когда принтер освободится. Программа, называемая *стюлером*, управляет очередью и следит за тем, чтобы после завершения одного задания печати следующее задание было отправлено на принтер.

Информационные пакеты в компьютерных сетях также ставятся в очереди. Каждый пакет, прибывающий на сетевой узел, должен быть передан следующему узлу на пути к своей точке назначения. Узел-маршрутизатор передает пакеты последовательно, так что дополнительные пакеты должны ждать в очереди до того момента, когда маршрутизатор сможет обработать их.

### Класс `Queue`, производный от `List`

Код на ил. 19.16 создает класс очереди посредством наследования от класса списка. Класс `QueueInheritance` (см. ил. 19.16) содержит методы `Enqueue`, `Dequeue`, `IsEmpty`

и `Display`. В сущности, это методы `InsertAtBack`, `RemoveFromFront`, `IsEmpty` и `Display` класса `List`. Конечно, класс `List` содержит и другие методы (такие, как `InsertAtFront` и `RemoveFromBack`), которые не должны быть доступны в открытом интерфейсе очереди. Однако важно помнить, что все методы открытого интерфейса класса `List` также являются открытыми методами производного класса `QueueInheritance`.

Реализация каждого метода `QueueInheritance` вызывает соответствующий метод `List` — метод `Enqueue` вызывает `InsertAtBack`, метод `Dequeue` вызывает `RemoveFromFront`. Вызовы `IsEmpty` и `Display` вызывают версии базового класса, которые класс `QueueInheritance` наследует от класса `List` в своем открытом интерфейсе. Класс `QueueInheritance` использует пространство имен `LinkedListLibrary` (см. ил. 19.4); следовательно, библиотека классов, определяющая `QueueInheritance`, должна содержать ссылку на библиотеку `LinkedListLibrary`.

```

1 // Ил. 19.16: QueueInheritanceLibrary.cs
2 // Реализация очереди наследованием от класса List.
3 using LinkedListLibrary;
4
5 namespace QueueInheritanceLibrary
6 {
7     // Класс QueueInheritance наследует функциональность класса List.
8     public class QueueInheritance : List
9     {
10         // Вызов конструктора List с именем "queue"
11         public QueueInheritance()
12             : base( "queue" )
13         {
14         } // Конец конструктора
15
16         // Размещение dataValue в конце очереди посредством
17         // вставки dataValue в конец связанного списка
18         public void Enqueue( object dataValue )
19         {
20             InsertAtBack( dataValue );
21         } // Конец метода Enqueue
22
23         // Удаление элемента в начале очереди посредством удаления
24         // элемента в начале связанного списка
25         public object Dequeue()
26         {
27             return RemoveFromFront();
28         } // Конец метода Dequeue
29     } // Конец класса QueueInheritance
30 } // Конец пространства имен QueueInheritanceLibrary

```

**Ил. 19.16.** Реализация очереди наследованием от класса `List`

Метод `Main` класса `QueueInheritanceTest` (ил. 19.17) создает объект `QueueInheritance` с именем `queue`. В строках 15–18 определяются четыре значения, которые будут заноситься в очередь и извлекаться из нее. Программа ставит в очередь (строки 21, 23, 25 и 27) переменные `bool` (`true`), `char` (`'$'`), `int` (`34567`) и `string` (`"hello"`). Класс `QueueInheritanceTest` использует пространство имен `LinkedListLibrary`

и пространство имен `QueueInheritanceLibrary`; это означает, что в решение класса `QueueInheritanceTest` должны быть включены ссылки на обе библиотеки.

```

1 // Ил. 19.17: QueueTest.cs
2 // Тестирование класса QueueInheritance.
3 using System;
4 using QueueInheritanceLibrary;
5 using LinkedListLibrary;
6
7 // Демонстрация функциональности класса QueueInheritance
8 class QueueTest
9 {
10     public static void Main( string[] args )
11     {
12         QueueInheritance queue = new QueueInheritance();
13
14         // Создание объектов для сохранения в очереди
15         bool aBoolean = true;
16         char aCharacter = '$';
17         int anInteger = 34567;
18         string aString = "hello";
19
20         // Использование метода Enqueue для добавления элементов в очередь
21         queue.Enqueue( aBoolean );
22         queue.Display();
23         queue.Enqueue( aCharacter );
24         queue.Display();
25         queue.Enqueue( anInteger );
26         queue.Display();
27         queue.Enqueue( aString );
28         queue.Display();
29
30         // Использование метода Dequeue для извлечения элементов из очереди
31         object removedObject = null;
32
33         // Извлечение элементов из очереди
34         try
35         {
36             while ( true )
37             {
38                 removedObject = queue.Dequeue();
39                 Console.WriteLine( removedObject + " dequeued" );
40                 queue.Display();
41             } // Конец while
42         } // Конец try
43         catch ( EmptyListException emptyListException )
44         {
45             // Если произойдет исключение, вывести трассировку стека
46             Console.Error.WriteLine( emptyListException.StackTrace );
47         } // Конец catch
48     } // Конец Main
49 } // Конец класса QueueTest

```

The queue is: True

**Ил. 19.17.** Тестирование класса `QueueInheritance` (продолжение ↗)

```

The queue is: True $
The queue is: True $ 34567
The queue is: True $ 34567 hello
True dequeued
The queue is: $ 34567 hello
$ dequeued
The queue is: 34567 hello
34567 dequeued
The queue is: hello
hello dequeued
Empty queue
at LinkedListLibrary.List.RemoveFromFront()
in C:\examples\ch21\Fig21_04\LinkedListLibrary\
LinkedListLibrary\LinkedListLibrary.cs:line 78
at QueueInheritanceLibrary.QueueInheritance.Dequeue()
in C:\examples\ch21\Fig21_16\QueueInheritanceLibrary\
QueueInheritanceLibrary\QueueInheritance.cs:line 28
at QueueTest.Main(String[] args)
in C:\examples\ch21\Fig21_17\QueueTest\
QueueTest\QueueTest.cs:line 38

```

**Ил. 19.17.** Тестирование класса QueueInheritance (окончание)

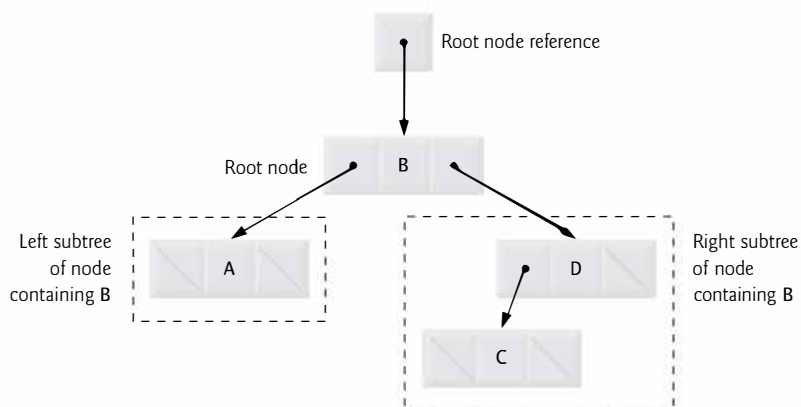
Бесконечный цикл `while` (строки 36–41) выводит элементы из очереди в порядке FIFO. Если в очереди не осталось ни одного объекта, метод `Dequeue` выдает исключение `EmptyListException`, а программа выводит трассировку стека на момент возникновения исключения. Программа использует метод `Display` (унаследованный от класса `List`) для вывода содержимого очереди после каждой операции.

## 19.7. Деревья

Связанные списки, стеки и очереди относятся к линейным структурам данных. Дерево представляет собой нелинейную двумерную структуру данных со специальными свойствами. В узлах деревьев хранятся две ссылки и более.

### Основная терминология

В бинарном дереве (ил. 19.18) каждый узел содержит две ссылки (нуль, одна или обе могут содержать `null`). Корневой узел является первым узлом дерева. Каждая ссылка в корневом узле ссылается на дочерний узел, который является первым узлом соответствующего поддерева, левого или правого. Дочерние узлы конкретного узла называются *родственными* узлами. Узел, не имеющий дочерних узлов, называется *листовым* (или просто «листом»). Специалисты по информатике обычно рисуют деревья сверху вниз начиная от корня — в направлении, обратном порядку роста большинства деревьев в природе.



**Ил. 19.18.** Графическое представление бинарного дерева

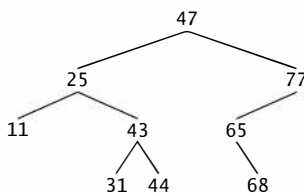


### ТИПИЧНАЯ ОШИБКА 19.2

Не забывайте присваивать null ссылкам в листовых узлах дерева, это довольно распространенная логическая ошибка.

## Бинарное дерево поиска

В следующем примере мы создадим особую разновидность бинарного дерева — так называемое *бинарное дерево поиска*. У бинарного дерева поиска (которое не содержит дубликатов в значениях узлов) есть отличительная особенность: значения в любом левом поддереве меньше значения родительского узла поддерева, а значения в любом правом поддереве больше значения родительского узла поддерева. На ил. 19.19 изображено бинарное дерево поиска с 9 целочисленными значениями. Форма бинарного дерева поиска, соответствующего набору данных, может зависеть от порядка вставки значений в дерево.



**Ил. 19.19.** Бинарное дерево поиска с 9 значениями

### 19.7.1. Бинарное дерево поиска с целыми значениями

Приложение на ил. 19.20 и 19.21 создает бинарное дерево поиска с целыми числами и обходит его (то есть перебирает все его узлы) тремя способами — рекурсивным симметричным, прямым и обратным. Программа генерирует 10 случайных чисел

и вставляет каждое из них в дерево. На ил. 19.20 определяется класс `Tree` в пространстве имен `BinaryTreeLibrary` (для повторного использования). На ил. 19.21 определяется класс `TreeTest` для демонстрации функциональности класса `Tree`. Метод `Main` класса `TreeTest` создает экземпляр пустого объекта `Tree`, после чего генерирует 10 случайных чисел и вставляет их в бинарное дерево вызовом метода `InsertNode` класса `Tree`. Затем программа перебирает узлы дерева тремя способами — прямым, симметричным и обратным (эти способы более подробно рассматриваются далее).

```

1  // Ил. 19.20: BinaryTreeLibrary.cs
2  // Объявление класса TreeNode и класса Tree.
3  using System;
4
5  namespace BinaryTreeLibrary
6  {
7      // Объявление класса TreeNode
8      class TreeNode
9      {
10         // Автоматически реализуемое свойство LeftNode
11         public TreeNode LeftNode { get; set; }
12
13         // Автоматически реализуемое свойство Data
14         public int Data { get; set; }
15
16         // Автоматически реализуемое свойство RightNode
17         public TreeNode RightNode { get; set; }
18
19         // Инициализация Data и формирование листового узла
20         public TreeNode( int nodeData )
21         {
22             Data = nodeData;
23             LeftNode = RightNode = null; // У узла нет дочерних узлов
24         } // Конец конструктора
25
26         // Вставка объекта TreeNode в дерево, содержащее узлы;
27         // дубликаты игнорируются.
28         public void Insert( int insertValue )
29         {
30             if ( insertValue < Data ) // Вставка в левое поддерево
31             {
32                 // Вставка нового объекта TreeNode
33                 if ( LeftNode == null )
34                     LeftNode = new TreeNode( insertValue );
35                 else // Продолжить обход левого поддерева
36                     LeftNode.Insert( insertValue );
37             } // Конец if
38             else if ( insertValue > Data ) // Вставка в правое поддерево
39             {
40                 // Вставка нового объекта TreeNode
41                 if ( RightNode == null )
42                     RightNode = new TreeNode( insertValue );
43                 else // Продолжить обход правого поддерева
44                     RightNode.Insert( insertValue );
45             } // Конец else if

```

**Ил. 19.20.** Объявление классов `TreeNode` и `Tree` (продолжение ↗)

```

46     } // Конец метода Insert
47 } // Конец класса TreeNode
48
49 // Объявление класса Tree
50 public class Tree
51 {
52     private TreeNode root;
53
54     // Конструирование пустого дерева с целочисленными данными
55     public Tree()
56     {
57         root = null;
58     } // Конец конструктора
59
60     // Вставка нового узла в бинарное дерево поиска.
61     // Если ссылка на корневой узел равна null, создать корневой узел.
62     // В противном случае вызывается метод Insert класса TreeNode.
63     public void InsertNode( int insertValue )
64     {
65         if ( root == null )
66             root = new TreeNode( insertValue );
67         else
68             root.Insert( insertValue );
69     } // Конец метода InsertNode
70
71     // Запуск прямого обхода
72     public void PreorderTraversal()
73     {
74         PreorderHelper( root );
75     } // Конец метода PreorderTraversal
76
77     // Рекурсивный метод для выполнения прямого обхода
78     private void PreorderHelper( TreeNode node )
79     {
80         if ( node != null )
81         {
82             // Вывод данных узла
83             Console.Write( node.Data + " " );
84
85             // Обход левого поддерева
86             PreorderHelper( node.LeftNode );
87
88             // Обход правого поддерева
89             PreorderHelper( node.RightNode );
90         } // Конец if
91     } // Конец метода PreorderHelper
92
93     // Запуск симметричного обхода
94     public void InorderTraversal()
95     {
96         InorderHelper( root );
97     } // Конец метода InorderTraversal
98
99     // Рекурсивный метод для выполнения симметричного обхода
100    private void InorderHelper( TreeNode node )

```

**Ил. 19.20.** Объявление классов `TreeNode` и `Tree` (продолжение ↗)



```

101     {
102         if ( node != null )
103         {
104             // Обход левого поддерева
105             InorderHelper( node.LeftNode );
106
107             // Вывод данных узла
108             Console.Write( node.Data + " " );
109
110             // Обход правого поддерева
111             InorderHelper( node.RightNode );
112         } // Конец if
113     } // Конец метода InorderHelper
114
115     // Запуск обратного обхода
116     public void PostorderTraversal()
117     {
118         PostorderHelper( root );
119     } // Конец метода PostorderTraversal
120
121     // Рекурсивный метод для выполнения обратного обхода
122     private void PostorderHelper( TreeNode node )
123     {
124         if ( node != null )
125         {
126             // Обход левого поддерева
127             PostorderHelper( node.LeftNode );
128
129             // Обход правого поддерева
130             PostorderHelper( node.RightNode );
131
132             // Вывод данных узла
133             Console.Write( node.Data + " " );
134         } // Конец if
135     } // Конец метода PostorderHelper
136 } // Конец класса Tree
137 } // Конец пространства имен BinaryTreeLibrary

```

**Ил. 19.20.** Объявление классов `TreeNode` и `Tree` (окончание)

```

1 // Ил. 19.21: TreeTest.cs
2 // Тестирование класса Tree с бинарным деревом.
3 using System;
4 using BinaryTreeLibrary;
5
6 // Объявление класса TreeTest
7 public class TreeTest
8 {
9     // Тестирование класса Tree
10     public static void Main( string[] args )
11     {
12         Tree tree = new Tree();
13         int insertValue;

```

**Ил. 19.21.** Тестирование класса `Tree` с бинарным деревом (продолжение ↗)

```

14
15 Console.WriteLine( "Inserting values: " );
16 Random random = new Random();
17
18 // Вставка 10 случайных целых чисел в дерево
19 for ( int i = 1; i <= 10; i++ )
20 {
21     insertValue = random.Next( 100 );
22     Console.Write( insertValue + " " );
23
24     tree.InsertNode( insertValue );
25 } // Конец for
26
27 // выполнить прямой обход дерева
28 Console.WriteLine( "\n\nPreorder traversal" );
29 tree.PreorderTraversal();
30
31 // Выполнить симметричный обход дерева
32 Console.WriteLine( "\n\nInorder traversal" );
33 tree.InorderTraversal();
34
35 // Выполнить обратный обход дерева
36 Console.WriteLine( "\n\nPostorder traversal" );
37 tree.PostorderTraversal();
38 Console.WriteLine();
39 } // Конец Main
40 } // Конец класса TreeTest

```

```

Inserting values:
39 69 94 47 50 72 55 41 97 73

Preorder traversal
39 69 47 41 50 55 94 72 73 97

Inorder traversal
39 41 47 50 55 69 72 73 94 97

Postorder traversal
41 55 50 47 73 72 97 94 69 39

```

#### Ил. 19.21. Тестирование класса Tree с бинарным деревом (окончание)

Класс `TreeNode` (строки 8–47 на ил. 19.20) представляет собой самоотносимый класс с тремя свойствами: `LeftNode` и `RightNode` типа `TreeNode` и `Data` типа `int`. Изначально каждый объект `TreeNode` представляет листовой узел, поэтому конструктор (строки 20–24) инициализирует ссылки `LeftNode` и `RightNode` значением `null`. Метод `Insert` класса `TreeNode` (строки 28–46) будет рассмотрен ниже.

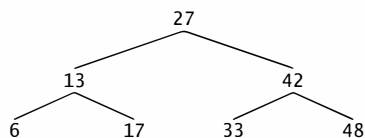
Класс `Tree` (строки 50–136) работает с объектами класса `TreeNode`. Закрытая переменная `root` (строка 52) содержит ссылку на корневой узел дерева. Открытый метод `InsertNode` (строки 63–69) вставляет новый узел в дерево, а открытые методы `PreorderTraversal` (строки 72–75), `InorderTraversal` (строки 94–97) и `PostorderTraversal` (строки 116–119) начинают обход дерева. Каждый из этих методов вызывает отдельный вспомогательный рекурсивный метод для выполнения

операций обхода с внутренним представлением дерева. Конструктор `Tree` (строки 55–58) инициализирует `root` значением `null`; это указывает на то, что в исходном состоянии дерево не содержит узлов.

Метод `InsertNode` (строки 63–69) сначала проверяет, пусто ли дерево. Если дерево не содержит узлов, то строка 66 создает новый объект `TreeNode`, инициализирует узел целым числом, вставленным в дерево, и присваивает новый узел `root`. Если дерево не пусто, то `InsertNode` вызывает метод `Insert` класса `TreeNode` (строки 28–46), который рекурсивно определяет позицию нового узла в дереве и вставляет узел в эту позицию. В бинарном дереве поиска узел может вставляться *только как листовой*.

Метод `Insert` класса `TreeNode` сравнивает вставляемое значение со значением, хранимым в данных корневого узла. Если вставляемое значение меньше данных корневого узла, программа проверяет, пусто ли левое поддерево (строка 33). Если оно пусто, то строка 34 создает новый узел `TreeNode`, инициализирует его вставляемым целым числом и присваивает объект нового узла ссылке `LeftNode`. В противном случае строка 36 рекурсивно вызывает `Insert` для левого поддерева, чтобы повторить процедуру вставки для левого поддерева. Если вставляемое значение больше данных корневого узла, то программа проверяет, пусто ли правое поддерево (строка 41). Если оно пусто, то строка 42 создает новый узел `TreeNode`, инициализирует его вставляемым целым числом и присваивает объект нового узла ссылке `RightNode`. В противном случае строка 44 рекурсивно вызывает `Insert` для правого поддерева, чтобы повторить процедуру вставки для правого поддерева. Если вставляемое значение совпадает со значением существующего узла, оно игнорируется.

Методы `InorderTraversal`, `PreorderTraversal` и `PostorderTraversal` вызывают вспомогательные методы `InorderHelper` (строки 100–113), `PreorderHelper` (строки 78–91) и `PostorderHelper` (строки 122–135) соответственно для обхода дерева и вывода значений узлов. Вспомогательные методы класса `Tree` нужны для того, чтобы программист мог запустить обход без предварительного получения ссылки на корневой узел и последующего вызова рекурсивного метода с этой ссылкой. Методы `InorderTraversal`, `PreorderTraversal` и `PostorderTraversal` просто передают значение закрытой переменной `root` соответствующему вспомогательному методу для запуска обхода дерева. В следующем описании используется бинарное дерево поиска, изображенное на ил. 19.22.



Ил. 19.22. Бинарное дерево поиска

### Алгоритм симметричного обхода

Метод `InorderHelper` (строки 100–113) определяет последовательность действий при симметричном обходе.

1. Если аргумент равен `null`, дерево не обрабатывается.
2. Обойти левое поддерево вызовом `InorderHelper` (строка 105).
3. Обработать значение в узле (строка 108).
4. Обойти правое поддерево вызовом `InorderHelper` (строка 111).

При симметричном обходе значение узла обрабатывается только после обработки значений левого поддерева. Для дерева на ил. 19.22 порядок обхода будет таким:

6 13 17 27 33 42 48

При симметричном обходе бинарного дерева поиска значения узлов выводятся упорядоченными по возрастанию. В процессе создания бинарного дерева поиска (в сочетании с симметричным обходом) данные сортируются, а этот процесс называется *сортировкой бинарного дерева*.

### Алгоритм прямого обхода

Метод `PreorderHelper` (строки 78–91) определяет последовательность действий при прямом обходе:

1. Если аргумент равен `null`, дерево не обрабатывается.
2. Обработать значение в узле (строка 83).
3. Обойти левое поддерево вызовом `PreorderHelper` (строка 86).
4. Обойти правое поддерево вызовом `PreorderHelper` (строка 89).

При прямом обходе значение каждого узла обрабатывается при его посещении. После обработки значения в заданном узле сначала обрабатываются значения левого поддерева, а затем значения правого поддерева. Для дерева на ил. 19.22 порядок прямого обхода будет таким:

27 13 6 17 42 33 48

### Алгоритм обратного обхода

Метод `PostorderHelper` (строки 122–135) определяет последовательность действий при обратном обходе:

1. Если аргумент равен `null`, дерево не обрабатывается.
2. Обойти левое поддерево вызовом `PostorderHelper` (строка 127).
3. Обойти правое поддерево вызовом `PostorderHelper` (строка 130).
4. Обработать значение в узле (строка 133).

При прямом обходе значение каждого узла обрабатывается после обработки значений всех его дочерних узлов. Для дерева на ил. 19.22 порядок обратного обхода будет таким:

6 17 13 33 48 42 27

### Исключение дубликатов

Бинарное дерево поиска обеспечивает автоматическое исключение дубликатов. При построении дерева операция вставки распознает попытки вставки дубликатов, потому что при каждом сравнении дубликат следует тем же правилам «перейти налево» или «перейти направо», что и исходное значение. Таким образом, операция вставки в конечном итоге сравнивает дубликат со значением узла, содержащим то же значение. На этой стадии операция вставки может просто отбросить дубликат.

Поиск в бинарном дереве значения, совпадающего с ключом, выполняется быстро — особенно для *плотно упакованных* бинарных деревьев поиска. В плотно упакованном бинарном дереве каждый уровень содержит примерно вдвое больше элементов, чем предыдущий. На ил. 19.22 изображено плотно упакованное бинарное дерево. Плотное упакованное бинарное дерево поиска с  $n$  элементами содержит минимум  $\log_2 n$  уровней. Для такого дерева поиска совпадения (или определение его отсутствия) требуется не более  $\log_2 n$  сравнений. Поиск в (плотно упакованном) бинарном дереве поиска из 1000 элементов требует максимум 10 сравнений, потому что  $2^{10} > 1000$ . При поиске в (плотно упакованном) бинарном дереве поиска с 1 000 000 элементов требуется не более 20 сравнений, потому что  $2^{20} > 1\,000\,000$ .

## 19.7.2. Бинарное дерево поиска для объектов IComparable

Реализация бинарного дерева в разделе 19.7.1 хорошо работает, когда все данные относятся к типу `int`. Предположим, вы хотите работать с бинарным деревом с элементами `double`. Классы `TreeNode` и `Tree` придется переименовать и адаптировать для работы с `double`. Таким образом, для каждого типа данных придется создавать собственную версию классов `TreeNode` и `Tree`. Это приводит к увеличению объема кода и усложнению его сопровождения.

В идеале нам хотелось бы определить функциональность бинарного дерева, а затем использовать ее для многих типов. В таких языках, как C#, существуют средства унифицированной работы с объектами любых типов, позволяющие проектировать более гибкие структуры данных. В C# для этой цели используется механизм *обобщенных типов* (generics), описанный в главе 20.

В нашем следующем примере мы воспользуемся полиморфными средствами C# для реализации классов `TreeNode` и `Tree`, работающих с объектами любого типа, реализующего интерфейс `IComparable` (пространство имен `System`). Для этого принципиально необходимо иметь возможность сравнения объектов, хранящихся в бинарном дереве поиска, чтобы определить путь к точке вставки нового узла. Классы, реализующие `IComparable`, определяют метод `CompareTo` для сравнения объекта, для которого вызывается метод, с объектом, передаваемым в аргументе. Метод возвращает отрицательное значение `int`, если объект вызова меньше объекта-аргумента, нуль в случае равенства объектов и положительное значение, если объект вызова

больше объекта-аргумента. Кроме того, объект вызова и объект-аргумент должны относиться к одному типу данных; в противном случае метод выдает исключение `ArgumentException`.

В листингах на ил. 19.23–19.24 программа из раздела 19.7.1 расширяется для работы с объектами `IComparable`. Единственное ограничение новых версий классов `TreeNode` и `Tree` заключается в том, что объект `Tree` может содержать объекты только одного типа (то есть только `string` или только `double`). Если программа попытается вставить в `Tree` объекты разных типов, происходит исключение `ArgumentException`. Чтобы сделать возможной обработку объектов `IComparable`, мы изменили всего пять строк кода класса `TreeNode` (строки 14, 20, 28, 30 и 38 на ил. 19.23) и одну строку кода класса `Tree` (строка 63). Не считая строк 30 и 38, все остальные изменения сводятся к простой замене `int` на `IComparable`.

В строках 30 и 38 для сравнения вставленного значения со значением заданного узла ранее использовались операторы `<` и `>`. В этих строках теперь сравниваются объекты `IComparable` с использованием метода `CompareTo` интерфейса, после чего программа проверяет возвращаемое значение метода: меньше нуля (объект вызова меньше объекта-аргумента) или больше нуля (объект вызова больше объекта-аргумента). [*Примечание:* если бы этот класс был написан с использованием обобщенных типов, то тип данных — `int` или `IComparable` — можно было бы заменить во время компиляции любым типом, реализующим необходимые операторы и методы.]

```

1  // Ил. 19.23: BinaryTreeLibrary2.cs
2  // Объявление класса TreeNode и класса Tree.
3  using System;
4
5  namespace BinaryTreeLibrary2
6  {
7      // Объявление класса TreeNode
8      class TreeNode
9      {
10         // Автоматически реализуемое свойство LeftNode
11         public TreeNode LeftNode { get; set; }
12
13         // Автоматически реализуемое свойство Data
14         public IComparable Data { get; set; }
15
16         // Автоматически реализуемое свойство RightNode
17         public TreeNode RightNode { get; set; }
18
19         // Инициализация Data и формирование листового узла
20         public TreeNode( IComparable nodeData )
21         {
22             Data = nodeData;
23             LeftNode = RightNode = null; // У узла нет дочерних узлов
24         } // Конец конструктора
25
26         // Вставка объекта TreeNode в дерево, содержащее узлы;
27         // дубликаты игнорируются.

```

**Ил. 19.23.** Объявление классов `TreeNode` и `Tree` (продолжение ↗)

```

28     public void Insert( IComparable insertValue )
29     {
30         if ( insertValue.CompareTo(Data) < 0 ) // Вставка в левое поддереву
31         {
32             // insert new TreeNode
33             if ( LeftNode == null )
34                 LeftNode = new TreeNode( insertValue );
35             else // continue traversing left subtree
36                 LeftNode.Insert( insertValue );
37         } // end if
38         else if ( insertValue.CompareTo( Data ) > 0 ) // В правое поддереву
39         {
40             // Вставка нового объекта TreeNode
41             if ( RightNode == null )
42                 RightNode = new TreeNode( insertValue );
43             else // continue traversing right subtree
44                 RightNode.Insert( insertValue );
45         } // Конец else if
46     } // Конец метода Insert
47 } // Конец класса TreeNode
48
49 // Объявление класса Tree
50 public class Tree
51 {
52     private TreeNode root;
53
54     // Конструирование пустого дерева с объектами IComparable
55     public Tree()
56     {
57         root = null;
58     } // Конец конструктора
59
60     // Вставка нового узла в бинарное дерево поиска.
61     // Если ссылка на корневой узел равна null, создать корневой узел.
62     // В противном случае вызывается метод Insert класса TreeNode.
63     public void InsertNode( IComparable insertValue )
64     {
65         if ( root == null )
66             root = new TreeNode( insertValue );
67         else
68             root.Insert( insertValue );
69     } // Конец метода InsertNode
70
71     // Запуск прямого обхода
72     public void PreorderTraversal()
73     {
74         PreorderHelper( root );
75     } // Конец метода PreorderTraversal
76
77     // Рекурсивный метод для выполнения прямого обхода
78     private void PreorderHelper( TreeNode node )
79     {
80         if ( node != null )
81         {
82             // Вывод данных узла

```

**Ил. 19.23.** Объявление классов TreeNode и Tree (продолжение ↗)

```

83         Console.Write( node.Data + " " );
84
85         // Обход левого поддерева
86         PreorderHelper( node.LeftNode );
87
88         // Обход правого поддерева
89         PreorderHelper( node.RightNode );
90     } // Конец if
91 } // Конец метода PreorderHelper
92
93 // Запуск симметричного обхода
94 public void InorderTraversal()
95 {
96     InorderHelper( root );
97 } // Конец метода InorderTraversal
98
99 // Рекурсивный метод для выполнения симметричного обхода
100 private void InorderHelper( TreeNode node )
101 {
102     if ( node != null )
103     {
104         // Обход левого поддерева
105         InorderHelper( node.LeftNode );
106
107         // Вывод данного узла
108         Console.Write( node.Data + " " );
109
110         // Обход правого поддерева
111         InorderHelper( node.RightNode );
112     } // Конец if
113 } // Конец метода InorderHelper
114
115 // Запуск обратного обхода
116 public void PostorderTraversal()
117 {
118     PostorderHelper( root );
119 } // Конец метода PostorderTraversal
120
121 // Рекурсивный метод для выполнения обратного обхода
122 private void PostorderHelper( TreeNode node )
123 {
124     if ( node != null )
125     {
126         // Обход левого поддерева
127         PostorderHelper( node.LeftNode );
128
129         // Обход правого поддерева
130         PostorderHelper( node.RightNode );
131
132         // Вывод данных узла
133         Console.Write( node.Data + " " );
134     } // Конец if
135 } // Конец метода PostorderHelper
136 } // Конец класса class Tree
137 } // Конец пространства имен BinaryTreeLibrary

```

**Ил. 19.23.** Объявление классов `TreeNode` и `Tree` (окончание)



Класс `TreeTest` (ил. 19.24) создает три объекта `Tree` для хранения значений `int`, `double` и `string`; все эти типы в .NET Framework определяются как реализующие `Comparable`. Программа заполняет дерево значениями из массивов `intArray` (строка 12), `doubleArray` (строка 13) и `stringArray` (строки 14–15) соответственно.

```

1 // Ил. 19.24: TreeTest.cs
2 // Тестирование класса Tree с объектами Comparable.
3 using System;
4 using BinaryTreeLibrary2;
5
6 // Объявление класса TreeTest
7 public class TreeTest
8 {
9     // Тестирование класса Tree
10    public static void Main( string[] args )
11    {
12        int[] intArray = { 8, 2, 4, 3, 1, 7, 5, 6 };
13        double[] doubleArray = { 8.8, 2.2, 4.4, 3.3, 1.1, 7.7, 5.5, 6.6 };
14        string[] stringArray = { "eight", "two", "four",
15                                "three", "one", "seven", "five", "six" };
16
17        // Создание дерева int
18        Tree intTree = new Tree();
19        PopulateTree( intArray, intTree, "intTree" );
20        TraverseTree( intTree, "intTree" );
21
22        // Создание дерева double
23        Tree doubleTree = new Tree();
24        PopulateTree( doubleArray, doubleTree, "doubleTree" );
25        TraverseTree( doubleTree, "doubleTree" );
26
27        // Создание дерева string
28        Tree stringTree = new Tree();
29        PopulateTree( stringArray, stringTree, "stringTree" );
30        TraverseTree( stringTree, "stringTree" );
31    } // Конец Main
32
33    // Заполнение дерева элементами массива
34    private static void PopulateTree( Array array, Tree tree, string name )
35    {
36        Console.WriteLine( "\n\nInserting into " + name + ":" );
37
38        foreach ( Comparable data in array )
39        {
40            Console.Write( data + " " );
41            tree.InsertNode( data );
42        } // Конец foreach
43    } // Конец метода PopulateTree
44
45    // Обходы дерева
46    private static void TraverseTree( Tree tree, string treeType )
47    {
48        // Выполнить прямой обход дерева
49        Console.WriteLine( "\n\nPreorder traversal of " + treeType );

```

**Ил. 19.24.** Тестирование класса `Tree` с объектами `Comparable` (продолжение ↗)

```

50     tree.PreorderTraversal();
51
52     // Выполнить симметричный обход дерева
53     Console.WriteLine( "\n\nInorder traversal of " + treeType );
54     tree.InorderTraversal();
55
56     // Выполнить обратный обход дерева
57     Console.WriteLine( "\n\nPostorder traversal of " + treeType );
58     tree.PostorderTraversal();
59 } // Конец метода TraverseTree
60 } // Конец класса TreeTest

```

Inserting into intTree:

8 2 4 3 1 7 5 6

Preorder traversal of intTree

8 2 1 4 3 7 5 6

Inorder traversal of intTree

1 2 3 4 5 6 7 8

Postorder traversal of intTree

1 3 6 5 7 4 2 8

Inserting into doubleTree:

8.8 2.2 4.4 3.3 1.1 7.7 5.5 6.6

Preorder traversal of doubleTree

8.8 2.2 1.1 4.4 3.3 7.7 5.5 6.6

Inorder traversal of doubleTree

1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8

Postorder traversal of doubleTree

1.1 3.3 6.6 5.5 7.7 4.4 2.2 8.8

Inserting into stringTree:

eight two four three one seven five six

Preorder traversal of stringTree

eight two four five three one seven six

Inorder traversal of stringTree

eight five four one seven six three two

Postorder traversal of stringTree

five six seven one three four two eight

#### Ил. 19.24. Тестирование класса Tree с объектами IComparable (окончание)

Метод `PopulateTree` (строки 34–43) получает в аргументах массив со значениями-инициализаторами, объект `Tree`, в котором будут размещаться элементы массива, и строку с именем `Tree`, после чего все элементы массива вставляются в дерево.

Метод `TraverseTree` (строки 46–59) получает объект `Tree` и строку с именем `Tree`, после чего выводит данные прямого, симметричного и обратного обхода дерева. При симметричном обходе каждого объекта `Tree` данные выводятся в отсортированном порядке независимо от типа данных, хранящихся в дереве. Наша полиморфная реализация класса `Tree` вызывает метод `CompareTo` соответствующего типа данных для определения пути к позиции вставки каждого значения с использованием стандартных правил вставки в бинарном дереве поиска. Также обратите внимание на алфавитный порядок обхода дерева `string`.

## 19.8. Итоги

Из этой главы вы узнали, что простые типы представляются структурами значимых типов, но благодаря преобразованиям упаковки и распаковки они могут использоваться в любых местах программы, в которых могут использоваться `object`. Затем были рассмотрены связанные списки — коллекции элементов данных, объединенных ссылками в «цепочки». Программа может выполнять операции вставки и удаления в любой позиции связанного списка (хотя наша реализация выполняла операции вставки и удаления только от концов списка). Как было показано далее, структуры данных стека и очереди представляют собой версии списков с ограниченной функциональностью. У стеков операции вставки и удаления выполняются только в начале, а у очередей вставка выполняется в конце, а удаление в начале. Глава завершается описанием структуры данных бинарного дерева. Мы рассмотрели бинарные деревья поиска, предназначенные для быстрого поиска и сортировки данных, а также эффективного удаления дубликатов.

# 20 Обобщенные типы

## 20.1. Введение

В главе 19 были представлены структуры данных, которые сохраняли ссылки на объекты и работали с ними. В этих структурах данных можно хранить любые разновидности `object`, однако у такой реализации есть недостаток, с которым мы сталкиваемся при получении данных из коллекции. Как правило, приложение работает с конкретными типами объектов, поэтому ссылки на `object`, полученные из коллекции, обычно приходится преобразовывать к соответствующему типу, чтобы приложение могло правильно обработать объекты. Кроме того, данные значимых типов (например, `int` и `double`) приходится упаковывать для того, чтобы с ними можно было работать через ссылки на `object`; упаковка снижает производительность обработки таких данных. И что самое важное, обработка всех данных в формате `object` ограничивает возможности компилятора C# по проверке типов.

Разработчик может легко создать структуры данных, которые работают с произвольными типами данных как с `object` (как было сделано в главе 19), но любые несоответствия типов лучше обнаруживать во время компиляции — такая проверка называется *безопасностью типов во время компиляции*. Например, если реализация стека предназначена для хранения только значений `int`, при попытке занесения в нее `string` компилятор должен сообщить об ошибке. Кроме того, метод `Sort` должен сравнивать элементы, которые заведомо относятся к одному типу. Если создать версии класса `Stack` и метода `Sort` для конкретного типа, компилятор C# безусловно сможет обеспечить безопасность типов во время компиляции. Однако это означает, что нам придется создать несколько копий одного базового кода. В этой главе рассматривается механизм *обобщенных типов* (generics), который позволяет разработчику создавать обобщенные программные конструкции (см. ранее). Обобщенные методы позволяют определить целое семейство взаимосвязанных методов в одном объявлении; обобщенные классы определяют семейство взаимосвязанных классов. Аналогичным образом одно объявление обобщенного интерфейса описывает семейство взаимосвязанных интерфейсов. Обобщенные типы обеспечивают безопасность типов во время компиляции. [Примечание: также возможна реализация обобщенных структур и делегатов.] До настоящего момента в книге использовались обобщенные типы `List` (глава 9) и `Dictionary` (глава 17).

Вы можете написать обобщенный метод для сортировки массива объектов, затем вызвать его по отдельности для массива `int`, массива `double`, массива `string` и т. д. для сортировки каждого из разнотипных массивов. Компилятор выполняет проверку типов и следит за тем, чтобы массив, переданный обобщенному методу сортировки, содержал только элементы правильного типа. Можно написать один обобщенный класс `Stack` для работы со стеком объектов, а затем создать экземпляры `Stack` для стека с элементами `int`, стека с элементами `double`, стека с элементами `string` и т. д. Компилятор обеспечивает проверку типов и следит за тем, чтобы в стеке хранились только элементы правильного типа.

В этой главе представлены примеры обобщенных методов и классов. Также в ней рассматриваются отношения между обобщениями и другими функциональными аспектами C# — например, перегрузкой. В главе 21 будут описаны обобщенные и необобщенные классы коллекций .NET Framework; коллекцией называется структура данных, предназначенная для хранения группы взаимосвязанных объектов или значений. Классы коллекций .NET Framework используют обобщенные типы для того, чтобы разработчик мог задать точный тип объектов, хранящихся в конкретной коллекции.

## 20.2. Причины использования обобщенных методов

Перегруженные методы часто используются для выполнения сходных операций с разными типами данных. Чтобы понять, для чего нужны обобщенные методы, начнем с примера (ил. 20.1), содержащего три перегруженных метода `DisplayArray` (строки 23–29, строки 32–38 и строки 41–47). Эти методы выводят элементы массивов `int`, `double` и `char` соответственно. Вскоре мы реализуем эту программу более компактно и элегантно с использованием одного обобщенного метода.

```

1 // Ил. 20.1: OverloadedMethods.cs
2 // Использование перегруженных методов для вывода массивов разных типов.
3 using System;
4
5 class OverloadedMethods
6 {
7     static void Main( string[] args )
8     {
9         // Создание массивов с элементами int, double и char
10        int[] intArray = { 1, 2, 3, 4, 5, 6 };
11        double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
12        char[] charArray = { 'H', 'E', 'L', 'L', 'O' };
13
14        Console.WriteLine( "Array intArray contains:" );
15        DisplayArray( intArray ); // В аргументе передается массив int
16        Console.WriteLine( "Array doubleArray contains:" );

```

**Ил. 20.1.** Использование перегруженных методов для вывода массивов разных типов (продолжение ↗)

```
17     DisplayArray( doubleArray ); // В аргументе передается массив double
18     Console.WriteLine( "Array charArray contains:" );
19     DisplayArray( charArray ); // В аргументе передается массив char
20 } // Конец Main
21
22 // Вывод массива int
23 private static void DisplayArray( int[] inputArray )
24 {
25     foreach ( int element in inputArray )
26         Console.Write( element + " " );
27
28     Console.WriteLine( "\n" );
29 } // Конец метода DisplayArray
30
31 // Вывод массива double
32 private static void DisplayArray( double[] inputArray )
33 {
34     foreach ( double element in inputArray )
35         Console.Write( element + " " );
36
37     Console.WriteLine( "\n" );
38 } // Конец метода DisplayArray
39
40 // Вывод массива char
41 private static void DisplayArray( char[] inputArray )
42 {
43     foreach ( char element in inputArray )
44         Console.Write( element + " " );
45
46     Console.WriteLine( "\n" );
47 } // Конец метода DisplayArray
48 } // Конец класса OverloadedMethods
```

Array intArray contains:

1 2 3 4 5 6

Array doubleArray contains:

1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array charArray contains:

H E L L O

**Ил. 20.1.** Использование перегруженных методов для вывода массивов разных типов (окончание)

Программа начинается с объявления и инициализации трех массивов — массива `int` с именем `intArray` с шестью элементами (строка 10), массива `double` с именем `doubleArray` с семью элементами (строка 11) и массива `char` с именем `charArray` и пятью элементами (строка 12). Содержимое всех трех массивов выводится в строках 14–19.

Встречая вызов метода, компилятор пытается найти объявление метода с таким же именем и набором параметров, соответствующим типам аргументов в вызове метода. В нашем примере каждый вызов `DisplayArray` точно соответствует одному

из объявлений метода `DisplayArray`. Например, в строке 15 `DisplayArray` вызывается с аргументом `intArray`. Во время компиляции компилятор определяет тип аргумента `intArray` (то есть `int[]`), ищет метод с именем `DisplayArray` и одним параметром `int[]`, находит его в строках 23–29 и организует вызов этого метода. Аналогичным образом при обнаружении вызова `DisplayArray` в строке 17 компилятор определяет тип аргумента `doubleArray` (то есть `double[]`), ищет метод с именем `DisplayArray` и одним параметром `double[]`, находит его в строках 32–38 и организует вызов этого метода. Наконец, для вызова `DisplayArray` в строке 19 компилятор определяет тип аргумента `charArray` (то есть `char[]`), ищет метод с именем `DisplayArray` и одним параметром `char[]`, находит его в строках 41–47 и организует вызов этого метода.

Внимательно просмотрите каждый метод `DisplayArray`. Обратите внимание: тип элементов массива (`int`, `double` или `char`) встречается в каждом методе в двух местах: в заголовке метода (строки 23, 32 и 41) и в заголовке команды `foreach` (строки 25, 34 и 43). Если заменить тип элемента в каждом методе обобщенным именем (например, `T` — от «type», то есть «тип»), все три метода будут выглядеть так, как показано на ил. 20.2. Заменяя конкретный тип элементов массива в каждом из трех методов одним «обобщенным параметром типа», мы сможем объявить один метод `DisplayArray`, который сможет выводить элементы произвольного массива. Однако метод на ил. 20.2 не откомпилируется из-за неправильного синтаксиса. Правильный синтаксис обобщенного метода `DisplayArray` показан на ил. 20.3.

```
1 private static void DisplayArray( T[] inputArray )
2 {
3     foreach ( T element in inputArray )
4         Console.Write( element + " " );
5
6     Console.WriteLine( "\n" );
7 } // Конец метода DisplayArray
```

**Ил. 20.2.** Метод `DisplayArray`, в котором конкретные имена типов заменены обобщенным именем `T`. Внимание: этот код не откомпилируется!

## 20.3. Реализация обобщенного метода

Если операции, выполняемые несколькими перегруженными методами, идентичны для всех типов аргументов, перегруженные методы можно более компактно и удобно запрограммировать с использованием обобщенного метода. Вы можете написать одно объявление обобщенного метода, который будет вызываться в разное время с аргументами разных типов. По типам аргументов, переданных обобщенному методу, компилятор обеспечит соответствующую обработку каждого вызова.

На ил. 20.3 приложение на ил. 20.1 заново реализуется с использованием обобщенного метода `DisplayArray` (строки 24–30). Обратите внимание: вызовы метода `DisplayArray` в строках 16, 18 и 20 идентичны вызовам на ил. 20.1, результаты выполнения двух приложений идентичны, а код на ил. 20.3 на 17 строк короче кода на ил. 20.1. Как показано на ил. 20.3, механизм обобщения позволяет создать

и протестировать код один раз, а потом заново использовать его для разных типов данных. Он хорошо демонстрирует выразительную мощь обобщений.

```
1 // Ил. 20.3: GenericMethod.cs
2 // Использование обобщенного метода для вывода массивов разных типов.
3 using System;
4 using System.Collections.Generic;
5
6 class GenericMethod
7 {
8     public static void Main( string[] args )
9     {
10         // Создание массивов с элементами int, double и char
11         int[] intArray = { 1, 2, 3, 4, 5, 6 };
12         double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
13         char[] charArray = { 'H', 'E', 'L', 'L', 'O' };
14
15         Console.WriteLine( "Array intArray contains:" );
16         DisplayArray( intArray ); // В аргументе передается массив int
17         Console.WriteLine( "Array doubleArray contains:" );
18         DisplayArray( doubleArray ); // В аргументе передается массив double
19         Console.WriteLine( "Array charArray contains:" );
20         DisplayArray( charArray ); // В аргументе передается массив char
21     } // Конец Main
22
23     // Вывод массива произвольного типа
24     private static void DisplayArray< T >( T[] inputArray )
25     {
26         foreach ( T element in inputArray )
27             Console.Write( element + " " );
28
29         Console.WriteLine( "\n" );
30     } // Конец метода DisplayArray
31 } // Конец класса GenericMethod
```

Array intArray contains:

1 2 3 4 5 6

Array doubleArray contains:

1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array charArray contains:

H E L L O

### Ил. 20.3. Использование обобщенного метода для вывода массивов разных типов

В строке 24 начинается объявление метода `DisplayArray`. Все объявления обобщенных методов содержат список параметров-типов, заключенный в угловые скобки (`<T>` в нашем примере) после имени метода. Каждый список состоит из одного или нескольких параметров-типов, разделенных запятыми. Параметр-тип представляет собой идентификатор, используемый вместо имени конкретного типа. Параметр-тип может использоваться для объявления возвращаемого типа, типов параметров и типов локальных переменных в объявлении обобщенного метода; параметры-типы используются как «заместители» типов аргументов, представляющих типы данных, передаваемых обобщенному методу.



Тело обобщенного метода объявляется так же, как тело любого другого метода. Имена параметров-типов в объявлении метода должны соответствовать именам из списка. Например, в строке 26 переменная `element` в цикле `foreach` объявляется с типом `T`, соответствующим параметру-типу (`T`), объявленному в строке 24. Кроме того, параметр-тип может объявляться в списке параметров-типов только один раз, но в списке параметров метода может встречаться неоднократно. Имена параметров-типов не обязаны быть уникальными между обобщенными методами.



### ТИПИЧНАЯ ОШИБКА 20.1

Если вы забудете включить список параметров-типов в объявлении обобщенного метода, компилятор не опознает имена параметров-типов, встречающиеся в методе. Это приведет к ошибкам компиляции.

В списке параметров-типов `DisplayArray` (строка 24) параметр-тип `T` объявляется как «заместитель» для типа элементов массива, выводимого методом `DisplayArray`. Обратите внимание: `T` указывается в списке параметров как тип элемента массива (строка 24). В заголовке команды `foreach` (строка 26) `T` также используется как тип элемента. Это те же два места, в которых у перегруженных методов `DisplayArray` на ил. 20.1 указывался тип элемента `int`, `double` или `char`. Остальной код `DisplayArray` идентичен версии на ил. 20.1.



### ПРИЕМЫ ПРОГРАММИРОВАНИЯ 20.1

В качестве идентификаторов параметров-типов рекомендуется использовать отдельные прописные буквы. Как правило, параметру-типу, представляющему тип элемента массива (или другой коллекции), назначается идентификатор `E` (от «element») или `T` (от «type»).

Программа на ил. 20.3, как и на ил. 20.1, начинается с объявления и инициализации массива `int` с именем `intArray` с шестью элементами (строка 11), массива `double` с именем `doubleArray` с семью элементами (строка 12) и массива `char` с именем `charArray` и пятью элементами (строка 13). Содержимое всех трех массивов выводится вызовом `DisplayArray` (строки 16, 18 и 20) — сначала с аргументом `intArray`, затем с аргументом `doubleArray` и наконец с аргументом `charArray`.

Когда компилятор обнаруживает вызов метода (как в строке 16), он анализирует множество методов (обобщенных и необобщенных), которые могут соответствовать этому вызову, в поисках наиболее подходящего варианта. Если подходящий метод не найден или найдено сразу несколько кандидатов, компилятор выдает сообщение об ошибке. Если вы не уверены в том, какой из методов будет вызван, полную информацию о разрешении вызовов методов можно найти в разделе 14.5.5.1 документа ECMA C# Language Specification:

[www.ecma-international.org/publications/standards/Ecma-334.htm](http://www.ecma-international.org/publications/standards/Ecma-334.htm)

или в разделе 7.5.3 документа Microsoft C# Language Specification 4:

[bit.ly/CSharp4Spec](http://bit.ly/CSharp4Spec)

Для строки 16 компилятор определяет, что оптимальное совпадение достигается при замене параметра-типа `T` в строках 24 и 25 объявления метода `DisplayArray` типом элементов аргумента `intArray` (то есть `int`). Компилятор генерирует вызов `DisplayArray` с типом `int`, заменяющим параметра-тип `T`. Процесс повторяется для вызовов метода `DisplayArray` в строках 18 и 20.



### ТИПИЧНАЯ ОШИБКА 20.2

Если компилятор не может найти одно обобщенное или необобщенное объявление, оптимально подходящее для вызова метода, или если оптимальных кандидатов несколько, происходит ошибка компиляции.

Также можно явно указать аргументы-типы для обозначения конкретного типа, который должен использоваться при вызове обобщенной функции. Например, строку 16 можно записать в следующем виде:

```
DisplayArray< int >( intArray ); // В аргументе передается массив int
```

В этом вызове метода явно указан аргумент-тип (`int`), который должен использоваться для замены параметра-типа `T` в строках 24 и 26 объявления метода `DisplayArray`. Для каждой переменной, объявленной с параметром-типом, компилятор также определяет, разрешены ли операции, выполняемые с такой переменной, для всех типов, которые может принимать параметр-тип. Единственной операцией, выполняемой с элементами массивов в данном примере, является вывод строкового представления элементов. В строке 27 для каждого элемента значимого типа выполняется неявное преобразование упаковки, и для каждого элемента выполняется неявный вызов `ToString`. Поскольку все объекты поддерживают метод `ToString`, компилятор заключает, что в строке 27 с каждым элементом массива выполняется действительная операция.

Объявление обобщенного метода `DisplayArray` на ил. 20.3 снимает необходимость в перегруженных методах на ил. 20.1; в результате такой замены мы экономим 17 строк кода и создаем метод, пригодный для повторного использования, способный выводить строковые представления элементов любого одномерного массива, а не только массива `int`, `double` или `char`.

## 20.4. Ограничения типов

В этом разделе будет представлен обобщенный метод `Maximum`, который определяет и возвращает наибольший из трех своих аргументов (относящихся к одному типу). Обобщенный метод в этом примере использует параметр-тип для объявления типа как возвращаемого значения метода, так и его параметров. Обычно для сравнения и определения наибольшего значения используется оператор `>`, однако этот оператор не перегружен для каждого типа, встроенного в `Framework Class Library` или определяемого посредством расширения этих типов. Обобщенный код ограничивается выполнением операций, которые гарантированно работают для всех возможных

типов. Таким образом, выражение вида *переменная1*<*переменная2* допустимо только при одном условии: если компилятор может быть уверен в том, что оператор < поддерживается для всех типов, которые когда-либо будут использоваться с обобщенным кодом. Аналогичным образом вызов метода для переменной обобщенного типа возможен, только если компилятор точно знает, что этот метод будет поддерживаться всеми типами, которые будут использоваться с обобщенным кодом.

### Интерфейс `Comparable<T>`

Два объекта одного типа могут сравниваться, если этот тип реализует обобщенный интерфейс `Comparable<T>` (из пространства имен `System`). Преимущество реализации интерфейса `Comparable<T>` заключается в том, что объекты `Comparable<T>` могут использоваться с методами сортировки и поиска классов из пространства имен `System.Collections.Generic` — эти методы будут рассматриваться в главе 21. Все структуры `Framework Class Library`, соответствующие простым типам, реализуют этот интерфейс. Например, простому типу `double` соответствует структура `Double`, а простому типу `int` соответствует структура `Int32` — и `Double` и `Int32` реализуют интерфейс `Comparable<T>`. Типы, реализующие `Comparable<T>`, должны объявить метод `CompareTo` для сравнения объектов. Например, два значения типа `int` — `int1` и `int2` — можно сравнить следующим выражением:

```
int1.CompareTo( int2 )
```

Метод `CompareTo` должен вернуть 0, если объекты равны, отрицательное целое число, если `int1` меньше `int2`, или положительное целое число, если `int1` больше `int2`. Программист, который объявляет тип, реализующий `Comparable<T>`, должен определить метод `CompareTo` так, чтобы он сравнивал содержимое двух объектов этого типа и возвращал соответствующий результат.

### Определение ограничений типов

Объекты `Comparable` могут сравниваться, но по умолчанию они не могут использоваться с обобщенным кодом, потому что не все типы реализуют интерфейс `Comparable<T>`. Однако мы можем ограничить типы, используемые с обобщенным методом или классом, и потребовать, чтобы они соответствовали заданным требованиям. На ил. 20.4 объявляется метод `Maximum` (строки 20–34) с ограничением типа, которое требует, чтобы каждый из аргументов метода относился к типу `Comparable<T>`. Это важное ограничение, потому что сравнение поддерживается не для всех объектов. С другой стороны, все объекты `Comparable<T>` гарантированно содержат метод `CompareTo`, который может использоваться в методе `Maximum` для определения наибольшего из трех аргументов.

```
1 // Ил. 20.4: MaximumTest.cs
2 // Обобщенный метод Maximum возвращает наибольший из трех объектов.
3 using System;
4
5 class MaximumTest
6 {
```

**Ил. 20.4.** Обобщенный метод `Maximum` возвращает наибольший из трех объектов (продолжение ➤)

```
7 public static void Main( string[] args )
8 {
9     Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
10         3, 4, 5, Maximum( 3, 4, 5 ) );
11     Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
12         6.6, 8.8, 7.7, Maximum( 6.6, 8.8, 7.7 ) );
13     Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
14         "pear", "apple", "orange",
15         Maximum( "pear", "apple", "orange" ) );
16 } // Конец Main
17
18 // Обобщенная функция для определения наибольшего
19 // из объектов IComparable
20 private static T Maximum< T >( T x, T y, T z )
21     where T : IComparable< T >
22 {
23     T max = x; // Изначально считаем, что x - наибольший объект
24
25     // Сравнить y с max
26     if ( y.CompareTo( max ) > 0 )
27         max = y; // y - наибольшее значение
28
29     // Сравнить z с max
30     if ( z.CompareTo( max ) > 0 )
31         max = z; // z - наибольшее значение
32
33     return max; // Возвращает наибольший объект
34 } // Конец метода Maximum
35 } // Конец класса MaximumTest
```

Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear

**Ил. 20.4.** Обобщенный метод `Maximum` возвращает наибольший из трех объектов (окончание)

Обобщенный метод `Maximum` использует параметр-тип `T` как тип возвращаемого значения метода (строка 20), как тип параметров метода `x`, `y` и `z` (строка 20) и как тип локальной переменной `max` (строка 23). Секция `where` метода `Maximum` (после списка параметров в строке 21) определяет ограничение типа для параметра-типа `T`. В данном случае условие `where T : IComparable<T>` означает, что методу должен передаваться аргумент-тип, реализующий интерфейс `IComparable<T>`. Если ограничение типа не задано, по умолчанию используется ограничение `object`.

В C# существует несколько разновидностей ограничений типов. *Ограничение класса* означает, что аргумент должен быть объектом конкретного базового класса или одного из его субклассов. *Ограничение интерфейса* означает, что класс аргумента должен реализовать конкретный интерфейс. Ограничение типа в строке 21 является ограничением интерфейса, поскольку `IComparable<T>` является интерфейсом. Вы можете указать, что аргумент-тип должен быть ссылочным или значимым типом,

при помощи ограничения ссылочного типа (`class`) или значимого типа (`struct`) соответственно. Наконец, также можно указать ограничение конструктора `new()`, указывающее, что обобщенный код может использовать оператор `new` для создания новых объектов типа, представляемого параметром-типом. Если параметр-тип задается с ограничением конструктора, класс аргумента должен предоставлять открытый конструктор без параметров или конструктор по умолчанию, который гарантирует, что объекты класса могут создаваться без передачи аргументов конструктору; в противном случае происходит ошибка компиляции.

Для параметра-типа можно задать несколько ограничений. Для этого ограничения просто перечисляются через запятую в условии `where`. Ограничения класса, ссылочного типа или значимого типа должны указываться на первом месте — причем для каждого параметра-типа может задаваться только одна из этих разновидностей ограничений. Затем перечисляются ограничения интерфейсов (если они есть), и в последнюю очередь указываются ограничения конструкторов (если они есть).

### Анализ кода

Метод `Maximum` сначала предполагает, что его первый аргумент (`x`) является наибольшим, и присваивает его локальной переменной `max` (строка 23). Затем команда `if` в строках 26–27 проверяет, больше ли `y` значения `max` (метод `CompareTo` объекта `y` вызывается при помощи выражения `y.CompareTo(max)`). Если значение `y` больше `max`, то оно присваивается переменной `max` (строка 27). Аналогичным образом команда в строках 30–31 проверяет, больше ли `z` значения `max`. Если `z` больше, то строка 31 присваивает `z` переменной `max`, после чего строка 33 возвращает `max` вызывающей стороне.

В методе `Main` (строки 7–16) строка 10 вызывает `Maximum` с целыми значениями 3, 4 и 5. Обобщенный метод `Maximum` подходит для такого вызова, но его аргументы должны реализовать интерфейс `IComparable<T>`. Тип `int` является синонимом для структуры `Int32`, реализующей интерфейс `IComparable<int>`. Таким образом, `int` (и другие простые типы) является действительным аргументом для метода `Maximum`.

В строке 12 методу `Maximum` передаются три аргумента `double`. И снова такой вызов допустим, потому что `double` является синонимом для структуры `Double`, которая реализует `IComparable<double>`. В строке 14 методу `Maximum` передаются три строки, которые также являются объектами `IComparable<string>`. Мы намеренно поместили наибольшее значение в разных позициях при разных вызовах метода (строки 10, 12 и 15), чтобы показать, что обобщенный метод всегда находит наибольшее значение независимо от его позиции в списке аргументов и вычисленного аргумента-типа.

## 20.5. Перегрузка обобщенных методов

Обобщенные методы могут перегружаться. Каждый перегруженный метод должен иметь уникальную сигнатуру (см. главу 7). Класс может предоставить два и более обобщенных метода с одним именем, но разными параметрами. Например, можно предоставить вторую версию обобщенного метода `DisplayArray` (см. ил. 20.3)

с дополнительными параметрами `lowIndex` и `highIndex`, определяющими выводимую часть массива (см. приложение 20.8).

Обобщенный метод может перегружаться необобщенными методами с тем же именем. Когда компилятор встречает вызов метода, он ищет объявление метода, которое лучше всего соответствует имени метода и типам аргументов, заданным при вызове. Например, обобщенный метод `DisplayArray` на ил. 20.3 может быть перегружен версией, специализированной для типа `string`, которая выводит данные в табличном формате. Если компилятор не может сопоставить вызов метода с объявлением обобщенного или необобщенного метода или если наличие нескольких кандидатов создает неоднозначность, компилятор выдает сообщение об ошибке.

## 20.6. Обобщенные классы

Концепция структуры данных (например, стека), содержащей элементы данных, может существовать независимо от типа элементов, с которыми он работает. Обобщенный класс предоставляет средства для описания класса без привязки к типу. Далее программа может создавать версии обобщенного класса, специализированные для конкретных типов. При этом расширяются возможности повторного использования программного кода: в обобщенном классе используется простое, компактное обозначение конкретных типов, которые будут использоваться вместо параметров-типов. Компилятор обеспечивает безопасность типов во время компиляции, а исполнительная система заменяет параметры-типы аргументами-типами, чтобы ваш клиентский код мог взаимодействовать с обобщенным классом. Например, обобщенный класс `Stack` может быть основой для создания многих классов `Stack` («`Stack` с элементами `double`», «`Stack` с элементами `int`», «`Stack` с элементами `char`», «`Stack` с элементами `Employee`»). На ил. 20.5 представлено обобщенное объявление класса `Stack`. Не путайте его с классом `Stack` из пространства имен `System.Collections.Generics`. Объявление обобщенного класса похоже на объявление необобщенного класса, за исключением того, что за именем класса следует список параметров-типов (строка 5) и возможно, одно или несколько ограничений параметра-типа. Параметр-тип `T` представляет тип элементов, с которыми работает `Stack`. Как и в случае с обобщенными методами, список параметров-типов обобщенного класса состоит из одного или нескольких параметров-типов, разделенных запятыми. Параметр-тип `T` используется в объявлении класса `Stack` (см. ил. 20.5) для представления типа элемента. Класс `Stack` объявляет переменную `elements` как массив типа `T` (строка 8). В этом массиве (созданном в строке 21) будут храниться элементы `Stack`. [*Примечание:* в данном примере `Stack` реализуется как массив. Как вы видели в главе 19, стеки также часто реализуются как ограниченные версии связанных списков.]

```
1 // Ил. 20.5: Stack.cs
2 // Обобщенный класс Stack.
3 using System;
```

**Ил. 20.5.** Обобщенный класс `Stack` (продолжение ↗)

```

4
5 class Stack< T >
6 {
7     private int top;          // Ссылка на верхний элемент стека
8     private T[] elements;    // Массив для хранения элементов стека
9
10    // Конструктор без параметров создает стек с размером по умолчанию.
11    public Stack()
12        : this( 10 ) // Размер по умолчанию
13    {
14        // Вызывает конструктор в строке 18 для проведения инициализации
15    } // Конец конструктора
16
17    // Конструктор создает стек с заданным количеством элементов
18    public Stack( int stackSize )
19    {
20        if ( stackSize > 0 ) // Проверка stackSize
21            elements = new T[ stackSize ]; // Создание stackSize элементов
22        else
23            throw new ArgumentException( "Stack size must be positive." );
24
25        top = -1; // Изначально стек пуст
26    } // Конец конструктора
27
28    // Занесение элемента в стек; в случае ошибки
29    // выдается исключение FullStackException
30    public void Push( T pushValue )
31    {
32        if ( top == elements.Length - 1 ) // Стек заполнен
33            throw new FullStackException( string.Format(
34                "Stack is full, cannot push {0}", pushValue ) );
35
36        ++top; // Увеличение top
37        elements[ top ] = pushValue; // pushValue заносится в стек
38    } // Конец метода Push
39
40    // Метод возвращает верхний элемент, если стек не пуст;
41    // в противном случае выдается EmptyStackException
42    public T Pop()
43    {
44        if ( top == -1 ) // Стек пуст
45            throw new EmptyStackException( "Stack is empty, cannot pop" );
46
47        --top; // Уменьшение top
48        return elements[ top + 1 ]; // Вернуть верхнее значение
49    } // Конец метода Pop
50 } // Конец класса Stack

```

**Ил. 20.5.** Обобщенный класс Stack (окончание)

## Конструкторы Stack

Класс Stack содержит два конструктора. Конструктор без параметров (строки 11–15) передает размер стека по умолчанию (10) конструктору с одним аргументом, используя синтаксис this (строка 12) для вызова другого конструктора в том же классе. Конструктор с одним аргументом (строки 18–26) проверяет аргумент stackSize

и создает массив с заданным размером `stackSize`, если он больше 0 (или выдает исключение в противном случае).

### Метод `Push`

Метод `Push` (строки 30–38) сначала определяет, не пытается ли программа занести элемент в заполненный стек. Если стек полон, то в строках 33–34 выдается исключение `FullStackException` (объявленное на ил. 20.6). Если стек не заполнен, то строка 36 увеличивает счетчик `top` для обозначения новой позиции вершины, после чего строка 37 размещает аргумент в указанной позиции массива `elements`.

### Метод `Pop`

Метод `Pop` (строки 42–49) сначала определяет, не совершается ли попытка извлечения элемента из пустого стека. В таком случае строка 45 выдает исключение `EmptyStackException` (объявленное на ил. 20.7). В противном случае строка 47 уменьшает счетчик `top`, чтобы обозначить новую позицию вершины стека, а строка 48 возвращает исходный верхний элемент стека.

Каждый из классов `FullStackException` (см. ил. 20.6) и `EmptyStackException` (см. ил. 20.7) предоставляет конструктор без параметров, конструктор с одним аргументом для классов исключений (см. раздел 13.8) и конструктор с двумя аргументами для создания нового исключения на базе существующего. Конструктор без параметров назначает сообщение об ошибке по умолчанию, а два других конструктора назначают пользовательские сообщения об ошибках.

```
1 // Ил. 20.6: FullStackException.cs
2 // Исключение FullStackException сообщает о переполнении стека.
3 using System;
4
5 class FullStackException : Exception
6 {
7     // Конструктор без параметров
8     public FullStackException() : base( "Stack is full" )
9     {
10         // Пустой конструктор
11     } // Конец конструктора FullStackException
12
13     // Конструктор с одним параметром
14     public FullStackException( string exception ) : base( exception )
15     {
16         // Пустой конструктор
17     } // Конец конструктора FullStackException
18
19     // Конструктор с двумя параметрами
20     public FullStackException( string exception, Exception inner )
21         : base( exception, inner )
22     {
23         // Пустой конструктор
24     } // Конец конструктора FullStackException
25 } // Конец класса FullStackException
```

**Ил. 20.6.** Исключение `FullStackException` сообщает о переполнении стека



```

1 // Ил. 20.7: EmptyStackException.cs
2 // Исключение EmptyStackException сообщает о пустом стеке.
3 using System;
4
5 class EmptyStackException : Exception
6 {
7     // Конструктор без параметров
8     public EmptyStackException() : base( "Stack is empty" )
9     {
10         // Пустой конструктор
11     } // Конец конструктора EmptyStackException
12
13     // Конструктор с одним параметром
14     public EmptyStackException( string exception ) : base( exception )
15     {
16         // Пустой конструктор
17     } // Конец конструктора EmptyStackException
18
19     // Конструктор с двумя параметрами
20     public EmptyStackException( string exception, Exception inner )
21         : base( exception, inner )
22     {
23         // Пустой конструктор
24     } // Конец конструктора EmptyStackException
25 } // Конец класса EmptyStackException

```

**Ил. 20.7.** Исключение FullStackException сообщает о пустом стеке

Как и в случае с обобщенными методами, при компиляции обобщенного класса компилятор выполняет проверку типов к параметрам-типам класса, чтобы проверить возможность их использования с кодом обобщенного класса. Ограничения определяют операции, которые могут выполняться с параметрами-типами. Исполнительная система заменяет параметры-типы с конкретными типами во время выполнения. Для класса Stack (см. ил. 20.5) ограничение типа не задано, поэтому используется ограничение типа по умолчанию (object). Областью действия параметра-типа обобщенного класса является весь класс.

Теперь рассмотрим приложение (ил. 20.8), в котором используется обобщенный класс Stack. В строках 13–14 объявляются переменные типов Stack<double> и Stack<int>; типы double и int — аргументы-типы Stack. Компилятор заменяет параметры-типы в обобщенном классе, чтобы компилятор мог выполнить проверку типов. Метод Main создает объекты doubleStack с размером 5 (строка 18) и intStack с размером 10 (строка 19), после чего вызывает методы TestPushDouble (строки 28–48), TestPopDouble (строки 51–73), TestPushInt (строки 76–96) и TestPopInt (строки 99–121) для работы с двумя стеками.

```

1 // Ил. 20.8: StackTest.cs
2 // Тестирование обобщенного класса Stack.
3 using System;
4
5 class StackTest

```

**Ил. 20.8.** Тестирование обобщенного класса Stack (продолжение ↗)

```

6  {
7  // Создание массивов с элементами double и int
8  private static double[] doubleElements =
9      new double[]{ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
10 private static int[] intElements =
11     new int[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
12
13 private static Stack< double > doubleStack; // Стек с элементами double
14 private static Stack< int > intStack;       // Стек с элементами int
15
16 public static void Main( string[] args )
17 {
18     doubleStack = new Stack< double >( 5 ); // Стек с элементами double
19     intStack = new Stack< int >( 10 );      // Стек с элементами int
20
21     TestPushDouble(); // Занесение double в doubleStack
22     TestPopDouble();  // Извлечение double из doubleStack
23     TestPushInt();    // Занесение int в intStack
24     TestPopInt();     // Извлечение int из intStack
25 } // Конец Main
26
27 // Тестирование метода Push с doubleStack
28 private static void TestPushDouble()
29 {
30     // Занесение элементов в стек
31     try
32     {
33         Console.WriteLine( "\nPushing elements onto doubleStack" );
34
35         // Занесение элементов в стек
36         foreach ( var element in doubleElements )
37         {
38             Console.Write( "{0:F1} ", element );
39             doubleStack.Push( element ); // Занесение в doubleStack
40         } // Конец foreach
41     } // Конец try
42     catch ( FullStackException exception )
43     {
44         Console.Error.WriteLine();
45         Console.Error.WriteLine( "Message: " + exception.Message );
46         Console.Error.WriteLine( exception.StackTrace );
47     } // Конец catch
48 } // Конец метода TestPushDouble
49
50 // Тестирование метода Pop с doubleStack
51 private static void TestPopDouble()
52 {
53     // Извлечение элементов из стека
54     try
55     {
56         Console.WriteLine( "\nPopping elements from doubleStack" );
57
58         double popValue; // Сохранение элемента, извлеченного из стека

```

**Ил. 20.8.** Тестирование обобщенного класса Stack (продолжение ↗)

```

59
60         // Удаление всех элементов из стека
61         while ( true )
62         {
63             popValue = doubleStack.Pop(); // Извлечение из doubleStack
64             Console.Write( "{0:F1} ", popValue );
65         } // Конец while
66     } // Конец try
67     catch ( EmptyStackException exception )
68     {
69         Console.Error.WriteLine();
70         Console.Error.WriteLine( "Message: " + exception.Message );
71         Console.Error.WriteLine( exception.StackTrace );
72     } // Конец catch
73 } // Конец метода TestPopDouble
74
75 // Тестирование метода Push с intStack
76 private static void TestPushInt()
77 {
78     // Занесение элементов в стек
79     try
80     {
81         Console.WriteLine( "\nPushing elements onto intStack" );
82
83         // Занесение элементов в стек
84         foreach ( var element in intElements )
85         {
86             Console.Write( "{0} ", element );
87             intStack.Push( element ); // Занесение в intStack
88         } // Конец foreach
89     } // Конец try
90     catch ( FullStackException exception )
91     {
92         Console.Error.WriteLine();
93         Console.Error.WriteLine( "Message: " + exception.Message );
94         Console.Error.WriteLine( exception.StackTrace );
95     } // Конец catch
96 } // Конец метода TestPushInt
97
98 // Тестирование метода Pop с intStack
99 private static void TestPopInt()
100 {
101     // Извлечение элементов из стека
102     try
103     {
104         Console.WriteLine( "\nPopping elements from intStack" );
105
106         int popValue; // Сохранение элемента, извлеченного из стека
107
108         // Удаление всех элементов из стека
109         while ( true )
110         {
111             popValue = intStack.Pop(); // Извлечение из intStack

```

**Ил. 20.8.** Тестирование обобщенного класса Stack (продолжение ↗)

```

112         Console.Write( "{0} ", popValue );
113     } // Конец while
114 } // Конец try
115 catch ( EmptyStackException exception )
116 {
117     Console.Error.WriteLine();
118     Console.Error.WriteLine( "Message: " + exception.Message );
119     Console.Error.WriteLine( exception.StackTrace );
120 } // Конец catch
121 } // Конец метода TestPopInt
122 } // Конец класса StackTest

```

```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5 6.6
Message: Stack is full, cannot push 6.6
at Stack`1.Push(T pushValue) in
c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 36
at StackTest.TestPushDouble() in
c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 39

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Message: Stack is empty, cannot pop
at Stack`1.Pop() in
c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 47
at StackTest.TestPopDouble() in
c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 63

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10 11
Message: Stack is full, cannot push 11
at Stack`1.Push(T pushValue) in
c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 36
at StackTest.TestPushInt() in
c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 87

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Message: Stack is empty, cannot pop
at Stack`1.Pop() in
c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 47
at StackTest.TestPopInt() in
c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 111

```

**Ил. 20.8.** Тестирование обобщенного класса Stack (окончание)

### Метод TestPushDouble

Метод TestPushDouble (строки 28–48) вызывает метод Push для занесения значений 1.1, 2.2, 3.3, 4.4 и 5.5, хранимых в массиве doubleElements, в doubleStack. Команда foreach завершается, когда тестовая программа пытается занести шестое значение в объект doubleStack (в котором нет свободного места, потому что в doubleStack могут храниться только пять элементов). В этом случае метод выдает исключение FullStackException (см. ил. 20.6), сообщающее о заполнении стека. В строках 42–47

это исключение перехватывается, а программа выводит сообщение и информацию трассировки стека. Трассировка стека сообщает о возникшем исключении и показывает, что исключение было сгенерировано методом `Push` класса `Stack` в строке 36 файла `Stack.cs` (см. ил. 20.5). Из трассировки также видно, что метод `Push` был вызван методом `TestPushDouble` класса `StackTest` в строке 39 файла `StackTest.cs`. По этой информации можно определить методы, находившиеся в стеке вызова методов на момент возникновения исключения. Так как программа перехватывает исключение, исполнительная среда `C#` считает, что исключение было обработано, и программа продолжает выполнение.

### Метод `TestPopDouble`

Метод `TestPopDouble` (строки 51–73) вызывает метод `Pop` класса `Stack` в бесконечном цикле `while`, последовательно удаляя все значения из стека. Обратите внимание: значения извлекаются в порядке, обратном порядку их занесения (LIFO), — разумеется, это отличительная особенность стеков. Цикл `while` (строки 61–65) продолжает выполняться до тех пор, пока стек не опустеет. Исключение `EmptyStackException` происходит при попытке извлечения элемента из пустого стека. Управление передается блоку `catch` (строки 67–72), где оно обрабатывается, а программа может продолжить выполнение. Когда тестовая программа пытается извлечь из стека шестое значение, объект `doubleStack` пуст, поэтому метод `Pop` выдает исключение `EmptyStackException`.

### Методы `TestPushInt` и `TestPopInt`

Метод `TestPushInt` (строки 76–96) вызывает метод `Push` класса `Stack` для занесения значений в `intStack` до тех пор, пока стек не заполнится. Метод `TestPopInt` (строки 99–121) вызывает метод `Pop` класса `Stack` для удаления значений из `intStack`, пока оно не опустеет. Как и в предыдущем случае, значения извлекаются в порядке, противоположном порядку их занесения в стек.

### Создание обобщенных методов для тестирования класса `Stack<T>`

Обратите внимание: код методов `TestPushDouble` и `TestPushInt` для занесения значений в `Stack<double>` и `Stack<int>` почти идентичен, как и код `TestPopDouble` и `TestPopInt` для извлечения значений из `Stack<double>` и `Stack<int>` соответственно. Это обстоятельство также открывает возможность для использования обобщенных методов. На ил. 20.9 объявляется обобщенный метод `TestPush` (строки 33–54), который выполняет те же действия, что и методы `TestPushDouble` и `TestPushInt` на ил. 20.8, — то есть вызывает `Push` для занесения значений в `Stack<T>`. Аналогичный обобщенный метод `TestPop` (строки 57–79) выполняет те же действия, что и методы `TestPopDouble` и `TestPopInt` на ил. 20.8, — то есть вызывает `Pop` для извлечения значений из `Stack<T>`.

```
1 // Ил. 20.9: StackTest.cs
2 // Тестирование обобщенного класса Stack.
3 using System;
4 using System.Collections.Generic;
```

**Ил. 20.9.** Тестирование обобщенного класса `Stack` (продолжение ↗)

```

5
6 class StackTest
7 {
8     // Создание массивов с элементами double и int
9     private static double[] doubleElements =
10         new double[] { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
11     private static int[] intElements =
12         new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
13
14     private static Stack< double > doubleStack; // Стек с элементами double
15     private static Stack< int > intStack;       // Стек с элементами int
16
17     public static void Main( string[] args )
18     {
19         doubleStack = new Stack< double >( 5 ); // Стек с элементами double
20         intStack = new Stack< int >( 10 );      // Стек с элементами int
21
22         // Занесение значений double в doubleStack
23         TestPush( "doubleStack", doubleStack, doubleElements );
24         // Извлечение значений double из doubleStack
25         TestPop( "doubleStack", doubleStack );
26         // Занесение значений int в intStack
27         TestPush( "intStack", intStack, intElements );
28         // Извлечение значений int из intStack
29         TestPop( "intStack", intStack );
30     } // Конец Main
31
32     // Тестирование метода Push
33     private static void TestPush< T >( string name, Stack< T > stack,
34         IEnumerable< T > elements )
35     {
36         // Занесение содержимого elements в стек
37         try
38         {
39             Console.WriteLine( "\nPushing elements onto " + name );
40
41             // Занесение элементов в стек
42             foreach ( var element in elements )
43             {
44                 Console.Write( "{0} ", element );
45                 stack.Push( element ); // Занести в стек
46             } // Конец foreach
47         } // Конец try
48         catch ( FullStackException exception )
49         {
50             Console.Error.WriteLine();
51             Console.Error.WriteLine( "Message: " + exception.Message );
52             Console.Error.WriteLine( exception.StackTrace );
53         } // Конец catch
54     } // Конец метода TestPush
55
56     // Тестирование метода Pop
57     private static void TestPop< T >( string name, Stack< T > stack )

```

**Ил. 20.9.** Тестирование обобщенного класса Stack (продолжение ↗)

```

58     {
59         // Извлечение элементов из стека
60         try
61         {
62             Console.WriteLine( "\nPopping elements from " + name );
63
64             T popValue; // Сохранение элемента, извлеченного из стека
65
66             // Удаление всех элементов из стека
67             while ( true )
68             {
69                 popValue = stack.Pop(); // Извлечение из стека
70                 Console.Write( "{0} ", popValue );
71             } // Конец while
72         } // Конец try
73         catch ( EmptyStackException exception )
74         {
75             Console.Error.WriteLine();
76             Console.Error.WriteLine( "Message: " + exception.Message );
77             Console.Error.WriteLine( exception.StackTrace );
78         } // Конец catch
79     } // Конец TestPop
80 } // Конец класса StackTest

```

```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5 6.6
Message: Stack is full, cannot push 6.6
at Stack`1.Push(T pushValue)
in c:\examples\ch22\Fig22_09\Stack\Stack\Stack.cs:line 36
at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1 elements)
in c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 45

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Message: Stack is empty, cannot pop
at Stack`1.Pop() in c:\examples\ch22\Fig22_09\Stack\Stack\Stack.cs:line 47
at StackTest.TestPop[T](String name, Stack`1 stack) in
c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 69

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10 11
Message: Stack is full, cannot push 11
at Stack`1.Push(T pushValue) in
c:\examples\ch22\Fig22_09\Stack\Stack\Stack.cs:line 36
at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1 elements)
in c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 45

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Message: Stack is empty, cannot pop
at Stack`1.Pop() in c:\examples\ch22\Fig22_09\Stack\Stack\Stack.cs:line 47
at StackTest.TestPop[T](String name, Stack`1 stack) in
c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 69

```

### Ил. 20.9. Тестирование обобщенного класса Stack (окончание)

Метод `Main` (см. ил. 20.9, строки 17–30) создает объекты `Stack<double>` (строка 19) и `Stack<int>` (строка 20). В строках 23–29 вызываются обобщенные методы `TestPush` и `TestPop` для тестирования объектов `Stack`.

В обобщенном методе `TestPush` (строки 33–54) параметр-тип `T` (заданный в строке 33) используется для представления типа данных, хранящихся в `Stack`. Обобщенный метод получает три аргумента — строку, представляющую имя объекта `Stack` для вывода, объект типа `Stack<T>` и объект `IEnumerable<T>` с элементами, которые будут занесены в `Stack<T>` вызовом `Push`. Компилятор следит за соответствием типа `Stack` и типа элементов, которые будут заноситься в `Stack` при вызове `Push` (аргумент-тип обобщенного метода). Обобщенный метод `TestPop` (строки 57–79) получает два аргумента: строку, представляющую имя объекта `Stack` для вывода, и объект типа `Stack<T>`.

## 20.7. Итоги

Эта глава посвящена механизму обобщений, обеспечивающему безопасность типов во время компиляции за счет проверки возможных несоответствий типов. Вы узнали, что компилятор разрешит компиляцию обобщенного кода только в том случае, если все операции, выполняемые с параметрами-типами в обобщенном коде, поддерживаются для всех типов, которые могут использоваться с обобщенным кодом. Также вы научились объявлять обобщенные методы и классы с использованием параметров-типов и использовать ограничение типа для определения требований к параметру-типу — ключевому компоненту безопасности типов во время компиляции. Мы рассмотрели несколько разновидностей ограничений типов: ограничения ссылочных типов, ограничения значимых типов, ограничения классов, ограничения интерфейсов и ограничения конструкторов. Также была рассмотрена реализация множественных ограничений типов для параметра-типа. В завершение главы вы узнали, что обобщения расширяют возможности повторного использования кода.



# 21 Коллекции

## 21.1. Введение

В главе 19 рассматривалась тема создания структур данных и работы с ними. Материал излагался на «низком уровне»: мы вручную создавали каждый элемент структуры данных, используя динамическое выделение памяти с оператором `new`, и изменяли структуры данных, выполняя прямые манипуляции с элементами и ссылками на них. В большинстве приложений создавать такие структуры данных не обязательно — вместо них можно воспользоваться готовыми классами структур данных из .NET Framework. Такие классы называются *классами коллекций*, потому что они предназначены для хранения коллекций данных. Экземпляр такого класса содержит набор элементов. В качестве примера коллекции можно привести песни, хранимые на вашем компьютере, имена игроков вашей любимой спортивной команды и т. д.

Классы коллекций позволяют разработчику хранить наборы данных с использованием готовых структур данных, не заботясь о подробностях их реализации. Это хороший пример повторного использования кода: классы коллекций ускоряют работу программистов и повышают быстродействие программ за счет повышения скорости выполнения и снижения затрат памяти. В этой главе рассматриваются интерфейсы коллекций с описанием возможностей каждого типа, классов реализации и перечислителей для «обхода» коллекций.

.NET Framework предоставляет несколько пространств имен, относящихся к коллекциям. Пространство имен `System.Collections` содержит коллекции, в которых хранятся ссылки на объекты. Мы упоминаем их здесь, потому что в отрасли все еще существует большой объем унаследованного кода, в котором используются эти коллекции. В большинстве новых приложений следует использовать коллекции из пространства имен `System.Collections.Generic`, содержащего обобщенные классы (такие, как упоминавшиеся ранее `List<T>` и `Dictionary<K, V>`) для хранения данных конкретных типов. Пространство имен `System.Collections.Concurrent` содержит так называемые *поточно-безопасные* коллекции для использования в многопоточных приложениях. Пространство имен `System.Collections.Specialized` содержит подборку коллекций для поддержки конкретных типов, таких как `string` и биты. Дополнительную информацию об этом пространстве имен можно найти по адресу [msdn.microsoft.com/en-us/library/system.collections.specialized.aspx](http://msdn.microsoft.com/en-us/library/system.collections.specialized.aspx). Коллекции этих

пространств имен предоставляют стандартизированные компоненты, предназначенные для повторного использования; вам не нужно писать собственные классы коллекций. Эти классы коллекций написаны с расчетом на широкое повторное использование, оптимизированы для быстрого выполнения и эффективного использования памяти.

## 21.2. Обзор коллекций

Все классы коллекций .NET Framework реализуют тот или иной набор интерфейсов. В этих интерфейсах объявляются операции, которые должны выполняться с разными типами коллекций. Некоторые интерфейсы коллекций .NET Framework перечислены на ил. 21.1. Все интерфейсы на ил. 21.1 объявлены в пространстве имен `System.Collections`, и у них существуют обобщенные аналоги в пространстве имен `System.Collections.Generic`. В программной среде предоставлены стандартные реализации этих интерфейсов. Программисты также могут предоставить реализации, подходящие для их требований.

Интерфейс	Описание
<code>ICollection</code>	Интерфейс, от которого наследуют интерфейсы <code>IList</code> и <code>IDictionary</code> . Содержит свойство <code>Count</code> для определения размера коллекции и метод <code>CopyTo</code> для копирования содержимого коллекции в традиционный массив
<code>IList</code>	Упорядоченная коллекция, с которой можно работать как с массивом. Предоставляет индексатор для обращения к элементам по целочисленному индексу. Также содержит методы поиска и изменения коллекции, включая <code>Add</code> , <code>Remove</code> , <code>Contains</code> и <code>IndexOf</code>
<code>IDictionary</code>	Коллекция значений, индексируемых произвольным объектом-«ключом». Предоставляет индексатор для обращения к элементам по индексу (тип <code>object</code> ) и методы для изменения коллекции (например, <code>Add</code> и <code>Remove</code> ). Свойство <code>Keys</code> содержит объекты, используемые в качестве индексов, а свойство <code>Values</code> — все объекты, хранящиеся в коллекции
<code>IEnumerable</code>	Объект поддерживает перечисление. Интерфейс содержит ровно один метод <code>GetEnumerator</code> , который возвращает объект <code>IEnumerator</code> (см. раздел 21.3). <code>ICollection</code> расширяет <code>IEnumerable</code> , поэтому все классы коллекций прямо или косвенно реализуют <code>IEnumerable</code>

**Ил. 21.1.** Некоторые интерфейсы коллекций

### Пространства имен `System.Collections` и `System.Collections.Specialized`

В ранних версиях C# в .NET Framework классы коллекций в основном содержались в пространствах имен `System.Collections` и `System.Collections.Specialized`. Эти классы хранили ссылки на `object` и работали с ними. В таких коллекциях можно хранить произвольные объекты, но один из недостатков хранения ссылок на `object` проявляется при получении данных из коллекции. Приложения обычно работают с конкретными типами объектов, поэтому ссылки на `object`, полученные из коллекции, обычно приходится преобразовывать к нужному типу.

### Пространство имен System.Collections.Generic

.NET Framework также включает пространство имен `System.Collections.Generic`, в котором используются обобщения, рассмотренные в главе 20. Многие классы этого пространства имен представляют собой обобщенные аналоги классов из пространства имен `System.Collections`. Это означает, что разработчик может задать конкретный тип, который будет храниться в коллекции. При этом он пользуется преимуществами проверки типов во время компиляции — компилятор следит за тем, чтобы с коллекцией использовались правильные типы, а при нарушении выдает ошибки во время компиляции. Кроме того, после задания типа, хранящегося в коллекции, прочитанные из нее элементы будут относиться к правильному типу. Это избавляет от необходимости выполнения явных преобразований типов, которые могут выдать во время выполнения исключение `InvalidCastException`, если объект, на который указывает ссылка, относится к неправильному типу. Заодно программа избавляется от затрат ресурсов, связанных с явным преобразованием, что делает ее более эффективной. Обобщенные коллекции особенно эффективны при хранении структур, поскольку они избавлены от затрат, связанных с упаковкой и распаковкой.

В этой главе представлены классы коллекций `Array`, `ArrayList`, `Stack`, `Hashtable`, обобщенные классы `SortedDictionary` и `LinkedList`, а также встроенные средства для работы с массивами. Пространство имен `System.Collections` предоставляет ряд других структур данных, включая `BitArray` (коллекцию значений `true/false`), `Queue` и `SortedList` (коллекция пар «ключ-значение», отсортированных по ключу, с возможностью выборки по ключу или по индексу).

В таблице на ил. 21.2 представлена сводка многих классов коллекций. Также будет рассмотрен интерфейс `IEnumerator`. Классы коллекций могут создавать *перечислители* (`enumerators`), которые позволяют программистам перебирать элементы коллекций. И хотя перечислители различаются по реализации, все они реализуют интерфейс `IEnumerator`, чтобы с ними можно было работать полиморфно. Как вы вскоре увидите, команда `foreach` представляет собой всего лишь удобную запись для использования перечислителя. Следующий раздел начнется с описания перечислителей и средств работы с массивами. Классы коллекций прямо или косвенно реализуют `ICollection` и `IEnumerable` (или их обобщенные эквиваленты `ICollection<T>` и `IEnumerable<T>` для обобщенных коллекций).

Класс	Реализует	Описание
Пространство имен System		
<code>Array</code>	<code>IList</code>	Базовый класс для всех традиционных массивов. См. раздел 21.3
Пространство имен System.Collections		
<code>ArrayList</code>	<code>IList</code>	Имитирует традиционные массивы, но поддерживает расширение или сокращение в соответствии с количеством элементов. См. раздел 21.4.1

**Ил. 21.2.** Некоторые классы коллекций .NET Framework (продолжение ☞)

Класс	Реализует	Описание
BitArray	ICollection	Массив для bool с эффективным расходом памяти
Hashtable	IDictionary	Неупорядоченная коллекция пар «ключ-значение» с возможностью обращения к элементам по ключу. См. раздел 21.4.3
Queue	ICollection	Очередь. См. раздел 19.6
SortedList	IDictionary	Коллекция пар «ключ-значение», отсортированная по ключу с возможностью обращения к элементам по ключу или по индексу
Stack	ICollection	Стек. См. раздел 21.4.2
Пространство имен System.Collections.Generic		
Dictionary<K, V>	IDictionary<K, V>	Обобщенная неупорядоченная коллекция пар «ключ-значение» с возможностью обращения к элементам по ключу. См. раздел 17.4
LinkedList<T>	ICollection<T>	Двусвязный список. См. раздел 21.5.2
List<T>	IList<T>	Обобщенная версия ArrayList. См. раздел 9.4
Queue<T>	ICollection<T>	Обобщенная версия Queue
SortedDictionary<K, V>	IDictionary<K, V>	Версия Dictionary, сортирующая данные по ключу с формированием бинарного дерева. См. раздел 21.5.1
SortedList<K, V>	IDictionary<K, V>	Обобщенная версия SortedList
Stack<T>	ICollection<T>	Обобщенная версия Stack. См. раздел 21.4.2

**Ил. 21.2.** Некоторые классы коллекций .NET Framework (окончание)

## 21.3. Класс Array и перечислители

В главе 8 были представлены базовые средства работы с массивами. Все массивы неявно наследуют от абстрактного базового класса `Array` (пространство имен `System`); этот класс определяет свойство `Length`, задающее количество элементов в массиве. Кроме того, класс `Array` предоставляет статические методы с алгоритмами обработки массивов. Обычно в классе `Array` эти методы перегружаются — например, метод `Reverse` может переставить в обратном порядке как элементы всего массива, так и элементы в заданном диапазоне. Полный список статических методов класса `Array` находится по адресу:

[msdn.microsoft.com/en-us/library/system.array.aspx](http://msdn.microsoft.com/en-us/library/system.array.aspx)

На ил. 21.3 представлены некоторые статические методы класса `Array`.

```

1 // Ил. 21.3: UsingArray.cs
2 // Статические методы класса Array для традиционных операций с массивами
3 using System;
4 using System.Collections;
```

**Ил. 21.3.** Класс `Array` предназначен для выполнения традиционных операций с массивами (продолжение ↗)

```
5
6 // Демонстрация алгоритмов класса Array
7 public class UsingArray
8 {
9     private static int[] intValues = { 1, 2, 3, 4, 5, 6 };
10    private static double[] doubleValues = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11    private static int[] intValuesCopy;
12
13    // Метод Main демонстрирует использование методов класса Array
14    public static void Main( string[] args )
15    {
16        intValuesCopy = new int[ intValues.Length ]; // Нули по умолчанию
17
18        Console.WriteLine( "Initial array values:\n" );
19        PrintArrays(); // Вывод исходного содержимого массива
20
21        // Сортировка значений doubleValue
22        Array.Sort( doubleValues );
23
24        // Копирование intValues в intValuesCopy
25        Array.Copy( intValues, intValuesCopy, intValues.Length );
26
27        Console.WriteLine( "\nArray values after Sort and Copy:\n" );
28        PrintArrays(); // output array contents
29        Console.WriteLine();
30
31        // Поиск значения 5 в intValues
32        int result = Array.BinarySearch( intValues, 5 );
33        if ( result >= 0 )
34            Console.WriteLine( "5 found at element {0} in intValues",
35                               result );
36        else
37            Console.WriteLine( "5 not found in intValues" );
38
39        // Поиск значения 8763 в intValues
40        result = Array.BinarySearch( intValues, 8763 );
41        if ( result >= 0 )
42            Console.WriteLine( "8763 found at element {0} in intValues",
43                               result );
44        else
45            Console.WriteLine( "8763 not found in intValues" );
46    } // Конец Main
47
48    // Вывод содержимого массива с использованием перечислителей
49    private static void PrintArrays()
50    {
51        Console.Write( "doubleValues: " );
52
53        // Перебор массива double с использованием перечислителя
54        IEnumerator enumerator = doubleValues.GetEnumerator();
55
56        while ( enumerator.MoveNext() )
57            Console.Write( enumerator.Current + " " );
```

**Ил. 21.3.** Класс Array предназначен для выполнения традиционных операций с массивами (продолжение ↗)

```
58
59     Console.Write( "\nintValues: " );
60
61     // Перебор массива int с использованием перечислителя
62     enumerator = intValues.GetEnumerator();
63
64     while ( enumerator.MoveNext() )
65         Console.Write( enumerator.Current + " " );
66
67     Console.Write( "\nintValuesCopy: " );
68
69     // Перебор второго массива int с использованием foreach
70     foreach ( var element in intValuesCopy )
71         Console.Write( element + " " );
72
73     Console.WriteLine();
74 } // Конец метода PrintArrays
75 } // Конец класса UsingArray
```

Initial array values:

```
doubleValues: 8.4 9.3 0.2 7.9 3.4
intValues: 1 2 3 4 5 6
intValuesCopy: 0 0 0 0 0 0
```

Array values after Sort and Copy:

```
doubleValues: 0.2 3.4 7.9 8.4 9.3
intValues: 1 2 3 4 5 6
intValuesCopy: 1 2 3 4 5 6
```

```
5 found at element 4 in intValues
8763 not found in intValues
```

**Ил. 21.3.** Класс `Array` предназначен для выполнения традиционных операций с массивами (окончание)

Директивы `using` в строках 3–4 включают пространства имен `System` (для классов `Array` и `Console`) и `System.Collections` (для интерфейса `IEnumerator`, который будет описан ниже). Ссылки на сборки этих пространств имен неявно включаются в каждое приложение, поэтому добавлять новые ссылки в файл проекта не обязательно.

Тестовый класс объявляет три статические переменные массивов (строки 9–11). Первые две строки инициализируют `intValues` и `doubleValues` массивами `int` и `double` соответственно. Статическая переменная `intValuesCopy` предназначена для демонстрации метода `Copy` объекта `Array`, поэтому она сохраняет значение по умолчанию `null` — ссылка еще не указывает на массив.

В строке 16 `intValuesCopy` инициализируется массивом `int` с такой же длиной, как у массива `intValues`. В строке 19 вызывается метод `PrintArrays` (строки 49–74) для вывода исходного содержимого всех трех массивов. Метод `PrintArrays` будет рассмотрен позднее. Из вывода на ил. 21.3 мы видим, что каждый элемент массива `intValuesCopy` инициализирован значением по умолчанию 0.

**Метод Sort класса Array**

В строке 22 статический метод `Sort` класса `Array` используется для сортировки массива `doubleValues`. При возвращении управления этим методом массив содержит исходные элементы, отсортированные по возрастанию. Элементы массива должны реализовать интерфейс `Comparable`.

**Метод Copy класса Array**

В строке 25 статический метод `Copy` класса `Array` используется для копирования элементов из массива `intValues` в массив `intValuesCopy`. В первом аргументе содержится копируемый массив (`intValues`), во втором — приемный массив (`intValuesCopy`), а в третьем — значение `int`, представляющее количество копируемых элементов (в нашем примере `intValues.Length` задает все элементы).

**Массив BinarySearch класса Array**

В строках 32 и 40 вызывается статический метод `BinarySearch` класса `Array` для выполнения бинарного поиска в массиве `intValues`. Метод `BinarySearch` получает отсортированный массив, в котором проводится поиск, и ключ для поиска. Метод возвращает индекс в массиве, по которому находится ключ (или отрицательное число, если ключ не найден). Предполагается, что `BinarySearch` получает отсортированный массив. Для несортированных массивов поведение метода непредсказуемо. Бинарный поиск подробно рассматривается в главе 18.

**Метод GetEnumerator класса Array и интерфейс IEnumerator**

Метод `PrintArrays` (строки 49–74) использует методы класса `Array` для перебора элементов массива. Метод `GetEnumerator` (строка 54) получает перечислитель для массива `doubleValues`. Как говорилось ранее, класс `Array` реализует интерфейс `IEnumerable`. Все массивы неявно наследуют от `Array`, поэтому типы массивов `int[]` и `double[]` реализуют метод `GetEnumerator` интерфейса `IEnumerable`, который возвращает перечислитель для перебора коллекции. Интерфейс `IEnumerator` (реализуемый всеми перечислителями) определяет методы `MoveNext` и `Reset`, а также свойство `Current`. Метод `MoveNext` перемещает перечислитель к следующему элементу коллекции; первый вызов `MoveNext` устанавливает перечислитель на первый элемент. `MoveNext` возвращает `true`, если коллекция содержит хотя бы один элемент; в противном случае возвращается `false`. Метод `Reset` устанавливает перечислитель *перед* первым элементом коллекции. Методы `MoveNext` и `Reset` выдают исключение `InvalidOperationException`, если содержимое коллекции как-либо изменяется после создания перечислителя. Свойство `Current` возвращает объект, находящийся в текущей позиции коллекции.

**ТИПИЧНАЯ ОШИБКА 21.1**

Если коллекция изменяется после создания перечислителя для коллекции, перечислитель немедленно становится недействительным — все последующие вызовы методов для перечислителя выдают исключение `InvalidOperationException`.

Перечислитель, возвращенный методом `GetEnumerator` в строке 54, изначально устанавливается перед первым элементом массива `doubleValues`. Затем, когда в строке 56 вызывается метод `MoveNext` для первой итерации цикла `while`, перечислитель перемещается к первому элементу `doubleValues`. Команда `while` в строках 56–57 перебирает все элементы до того момента, когда перечислитель выйдет за границу `doubleValues` и `MoveNext` вернет `false`. В каждой итерации свойство `Current` используется для получения и вывода текущего элемента массива. В строках 62–65 перебираются элементы массива `intValues`.

Обратите внимание: метод `PrintArrays` вызывается дважды (строки 19 и 28), поэтому метод `GetEnumerator` вызывается дважды для `doubleValues`. Метод `GetEnumerator` (строки 54 и 62) всегда возвращает перечислитель, установленный перед первым элементом. Также обратите внимание на то, что свойство `Current` интерфейса `IEnumerator` доступно только для чтения. Перечислители не могут использоваться для изменения содержимого коллекции — только для получения содержимого.

### Перебор коллекции в цикле `foreach`

В строках 70–71 команда `foreach` используется для перебора элементов коллекции. Команда `foreach` неявно получает перечислитель методом `GetEnumerator` и использует метод `MoveNext` перечислителя и свойство `Current` для перебора коллекции, как мы сделали явно в строках 54–57. По этой причине команда `foreach` может использоваться для перебора любой коллекции, реализующей интерфейс `IEnumerable`, — не только массивов. Эта функциональность будет продемонстрирована в следующем разделе при описании класса `ArrayList`.

### Методы `Clear`, `CreateInstance`, `IndexOf`, `LastIndexOf` и `Reverse` класса `Array`

В число статических методов `Array` также входят методы `Clear` (для присваивания элементам в заданном диапазоне 0, `false` или `null` — в зависимости от обстоятельств), `CreateInstance` (для создания нового массива заданного типа), `IndexOf` (для поиска первого вхождения объекта в массиве или части массива), `LastIndexOf` (для поиска последнего вхождения объекта в массиве или части массива) и `Reverse` (для перестановки содержимого массива или части массива в обратном порядке).

## 21.4. Необобщенные коллекции

Пространство имен `System.Collections` из .NET Framework Class Library является основным хранилищем необобщенных коллекций. Эти классы предоставляют стандартные реализации многих структур данных, описанных в главе 19, в коллекциях с хранением ссылок на тип `object`. В этом разделе будут представлены классы `ArrayList`, `Stack` и `Hashtable`.



### 21.4.1. Класс ArrayList

В большинстве языков программирования традиционные массивы имеют фиксированный размер — приложение не может изменять его в соответствии с требованиями приложения в процессе работы. В некоторых приложениях ограничение фиксированного размера создает проблемы. Разработчику приходится выбирать между массивами фиксированного размера, достаточного для максимального количества элементов, которые могут понадобиться приложению, и динамическими структурами данных, способными расширяться и сокращаться в соответствии с изменяющимися потребностями приложения на стадии выполнения.

Класс ArrayList из .NET Framework моделирует функциональность традиционных массивов и поддерживает динамическое изменение размера коллекции методами класса. В любое время ArrayList содержит определенное количество элементов, меньшее либо равное его емкости — количеству элементов, зарезервированному для ArrayList. Приложение может изменять емкость при помощи свойства Capacity класса ArrayList. В новых приложениях следует использовать обобщенный класс List<T> (обобщенную версию ArrayList), представленный в главе 9.



#### **ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 21.1**

Как и в случае со связанными списками, вставка дополнительных элементов в объект ArrayList, текущий размер которого меньше его емкости, является быстрой операцией.



#### **ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 21.2**

Операция вставки элемента в объект ArrayList, который приходится расширять для размещения нового элемента, выполняется медленно. Переполнение объекта ArrayList требует повторного выделения памяти и копирования в него существующих значений.



#### **ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 21.3**

При нехватке памяти используйте метод TrimToSize класса ArrayList, усекающий ArrayList до фактического размера; таким образом оптимизируется расходование памяти ArrayList. Будьте внимательны — если приложению потребуется вставить в коллекцию дополнительные элементы, вставка будет выполняться медленнее, потому что потребует динамического расширения ArrayList (усечение не оставляет свободного места для вставки).

В объекте ArrayList хранятся ссылки на объекты. Все классы являются производными от класса object, что позволяет хранить в ArrayList объекты любого типа. На ил. 21.4 перечислены некоторые полезные методы и свойства класса ArrayList.

Метод или свойство	Описание
Add	Добавляет объект в конец ArrayList и возвращает индекс (значение int), по которому был размещен добавленный объект
Capacity	Свойство для чтения и записи количества элементов, зарезервированного для ArrayList
Clear	Удаляет все объекты из ArrayList
Contains	Возвращает true, если заданный объект находится в ArrayList; в противном случае возвращается false
IndexOf	Возвращает индекс первого вхождения заданного объекта в ArrayList
Insert	Вставляет объект в позицию с заданным индексом
Remove	Удаляет первое вхождение заданного объекта
RemoveAt	Удаляет объект в позиции с заданным индексом
RemoveRange	Удаляет заданное количество элементов начиная с заданного индекса
Sort	Сортирует ArrayList; элементы должны реализовать IComparable, или приложение должно использовать перегруженную версию Sort, получающую IComparer
TrimToSize	Задаёт ёмкость ArrayList по количеству элементов, в настоящий момент хранящихся в коллекции (Count)

**Ил. 21.4.** Некоторые методы и свойства класса ArrayList

На ил. 21.5 представлен класс ArrayList и некоторые из его методов. Класс ArrayList принадлежит пространству имен System.Collections (строка 4). В строках 8–11 объявляются два массива с элементами string (colors и removeColors), которые будут использоваться для заполнения двух объектов ArrayList. Как говорилось в разделе 10.10, константы должны быть инициализированы во время компиляции, но переменные, доступные только для чтения, могут инициализироваться во время выполнения. Массивы представляют собой объекты, создаваемые во время выполнения, поэтому, чтобы сделать colors и removeColors неизменяемыми, необходимо объявить их с ключевым словом readonly (а не const). При запуске приложения создается объект ArrayList с исходной ёмкостью в один элемент, который сохраняется в списке переменных (строка 16). Команда foreach в строках 19–20 добавляет пять элементов массива colors в список при помощи метода Add класса ArrayList, поэтому список расширяется для хранения этих новых элементов. В строке 24 перегруженный конструктор ArrayList используется для создания нового объекта ArrayList, инициализированного содержимым массива removeColors, который затем присваивается переменной removeList. Конструктор может инициализировать содержимое ArrayList элементами переданного ему объекта ICollection. Такой конструктор определен для многих классов коллекций. Обратите внимание: вызов конструктора в строке 24 делает то же, что и строки 19–20.

В строке 27 вызывается метод DisplayInformation (строки 37–54) для вывода содержимого списка. Этот метод перебирает элементы ArrayList командой foreach.

Как упоминалось в разделе 21.3, команда `foreach` представляет собой удобную сокращенную запись для вызова метода `GetEnumerator` класса `ArrayList` и использования перечислителя для перебора элементов коллекции. Кроме того, строка 40 определяет, что переменная-итератор относится к типу `object`, потому что класс `ArrayList` не является обобщенным и в нем хранятся ссылки на `object`.

```

1 // Ил. 21.5: ArrayListTest.cs
2 // Использование класса ArrayList.
3 using System;
4 using System.Collections;
5
6 public class ArrayListTest
7 {
8     private static readonly string[] colors =
9         { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
10    private static readonly string[] removeColors =
11        { "RED", "WHITE", "BLUE" };
12
13    // Создание ArrayList, добавление элементов colors и обработка
14    public static void Main( string[] args )
15    {
16        ArrayList list = new ArrayList( 1 ); // Исходная емкость равна 1
17
18        // Добавление элементов массива colors в список ArrayList
19        foreach ( var color in colors )
20            list.Add( color ); // Добавление элемента в ArrayList
21
22        // Добавление элементов массива removeColors в removeList
23        // с использованием конструктора ArrayList
24        ArrayList removeList = new ArrayList( removeColors );
25
26        Console.WriteLine( "ArrayList: " );
27        DisplayInformation( list ); // Вывод списка
28
29        // Исключение цветов, входящих в removeList, из списка ArrayList
30        RemoveColors( list, removeList );
31
32        Console.WriteLine( "\nArrayList after calling RemoveColors: " );
33        DisplayInformation( list ); // Вывод содержимого списка
34    } // Конец Main
35
36    // Вывод информации о содержимом ArrayList
37    private static void DisplayInformation( ArrayList arrayList )
38    {
39        // Перебор элементов конструкцией foreach
40        foreach ( var element in arrayList )
41            Console.Write( "{0} ", element ); // Вызывает ToString
42
43        // Вывод размера и емкости
44        Console.WriteLine( "\nSize = {0}; Capacity = {1}",
45            arrayList.Count, arrayList.Capacity );
46
47        int index = arrayList.IndexOf( "BLUE" );

```

**Ил. 21.5.** Использование класса `ArrayList` (продолжение ↗)

```

48
49     if ( index != -1 )
50         Console.WriteLine( "The array list contains BLUE at index {0}.",
51             index );
52     else
53         Console.WriteLine( "The array list does not contain BLUE." );
54 } // Конец метода DisplayInformation
55
56 // Удаление значений, содержащихся в secondList, из firstList
57 private static void RemoveColors( ArrayList firstList,
58     ArrayList secondList )
59 {
60     // Перебор элементов второго объекта ArrayList (как для массива)
61     for ( int count = 0; count < secondList.Count; ++count )
62         firstList.Remove( secondList[ count ] );
63 } // Конец метода RemoveColors
64 } // Конец класса ArrayListTest

```

```

ArrayList:
MAGENTA RED WHITE BLUE CYAN
Size = 5; Capacity = 8
The array list contains BLUE at index 3.

```

```

ArrayList after calling RemoveColors:
MAGENTA CYAN
Size = 2; Capacity = 8
The array list does not contain BLUE.

```

**Ил. 21.5.** Использование класса ArrayList (окончание)

### Методы Count и Capacity

Мы используем свойства Count и Capacity (строки 45) для вывода текущего количества и максимального количества элементов, которое может храниться без выделения дополнительной памяти ArrayList. Из выходных данных на ил. 21.5 видно, что ArrayList имеет емкость 8.

### Методы IndexOf и Contains

В строке 47 вызывается метод IndexOf для определения позиции строки "BLUE" в arrayList и сохранения результата в локальной переменной index. Если элемент не найден, метод IndexOf возвращает -1. Команда if в строках 49–53 сравнивает значение index с -1, чтобы определить, содержит ли arrayList строку "BLUE". Если строка найдена, мы выводим ее индекс. У ArrayList также имеется метод Contains, который просто возвращает true, если объект присутствует в ArrayList, и false в противном случае. Метод Contains лучше использовать в том случае, если вас не интересует индекс элемента.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 21.4

Методы IndexOf и Contains класса ArrayList выполняют линейный поиск; для больших объектов ArrayList эта операция обходится достаточно дорого. Если объект ArrayList отсортирован, используйте метод BinarySearch класса ArrayList для выполнения более эффективного поиска. Метод BinarySearch возвращает индекс элемента или отрицательное число, если элемент не найден.

## Метод Remove

После того как метод `DisplayInformation` вернет управление, мы вызываем метод `RemoveColors` (строки 57–63) с двумя объектами `ArrayList`. Команда `for` в строках 61–62 перебирает элементы коллекции `secondList`. В строке 62 для обращения к элементу `ArrayList` используется индексатор — для этого за именем ссылки на `ArrayList` в квадратных скобках (`[]`) указывается индекс нужного элемента. Если индекс не находится в диапазоне от 0 до текущего количества элементов в `ArrayList` (задаваемого свойством `Count` объекта `ArrayList`), выдается исключение `ArgumentOutOfRangeException`. Мы используем индексатор для получения очередного элемента `secondList`, после чего удаляем каждый из них из `firstList` методом `Remove`. Этот метод выполняет линейный поиск и удаляет (только) первое вхождение заданного объекта. Все последующие элементы сдвигаются к началу `ArrayList`, заполняя освободившуюся позицию.

После вызова `RemoveColors` строка 33 снова выводит содержимое списка; результат подтверждает, что элементы `removeList` действительно были удалены.

## 21.4.2. Класс Stack

Класс `Stack` реализует структуру данных стека и предоставляет большую часть тех возможностей, которую мы определили в своей реализации из раздела 19.5. Тестовое приложение на ил. 19.14 демонстрирует использование созданной нами структуры данных `StackInheritance`. На ил. 21.6 приведена адаптированная версия ил. 19.14, демонстрирующая использование класса `Stack` из .NET Framework. В новых приложениях, в которых потребуется функциональность стека, следует использовать обобщенный класс `Stack<T>`.

```

1 // Ил. 21.6: StackTest.cs
2 // Тестирование класса Stack.
3 using System;
4 using System.Collections;
5
6 public class StackTest
7 {
8     public static void Main( string[] args )
9     {
10         Stack stack = new Stack(); // Создание пустого объекта Stack
11
12         // Создание объектов для сохранения в стеке
13         bool aBoolean = true;
14         char aCharacter = '$';
15         int anInteger = 34567;
16         string aString = "hello";
17
18         // Использование метода Push для добавления элементов в стек
19         stack.Push( aBoolean );
20         PrintStack( stack );
21         stack.Push( aCharacter );
22         PrintStack( stack );

```

**Ил. 21.6.** Демонстрация класса `Stack` (продолжение ↗)

```

23     stack.Push( anInteger );
24     PrintStack( stack );
25     stack.Push( aString );
26     PrintStack( stack );
27
28     // Проверка верхнего элемента стека
29     Console.WriteLine( "The top element of the stack is {0}\n",
30         stack.Peek() );
31
32     // Извлечение элементов из стека
33     try
34     {
35         while ( true )
36         {
37             object removedObject = stack.Pop();
38             Console.WriteLine( removedObject + " popped" );
39             PrintStack( stack );
40         } // Конец while
41     } // Конец try
42     catch ( InvalidOperationException exception )
43     {
44         // Если произойдет исключение, вывести трассировку стека
45         Console.Error.WriteLine( exception );
46     } // Конец catch
47 } // Конец Main
48
49 // display the contents of a stack
50 private static void PrintStack( Stack stack )
51 {
52     if ( stack.Count == 0 )
53         Console.WriteLine( "stack is empty\n" ); // the stack is empty
54     else
55     {
56         Console.Write( "The stack is: " );
57
58         // Перебор элементов стека командой foreach
59         foreach ( var element in stack )
60             Console.Write( "{0} ", element ); // Вызывает ToString
61
62         Console.WriteLine( "\n" );
63     } // Конец else
64 } // Конец метода PrintStack
65 } // Конец класса StackTest

```

The stack is: True

The stack is: \$ True

The stack is: 34567 \$ True

The stack is: hello 34567 \$ True  
The top element of the stack is hello

hello popped  
The stack is: 34567 \$ True

**Ил. 21.6.** Демонстрация класса Stack (продолжение ↗)

```

34567 popped
The stack is: $ True

$ popped
The stack is: True

True popped
stack is empty

System.InvalidOperationException: Stack empty.
  at System.Collections.Stack.Pop()
  at StackTest.Main(String[] args) in
    c:\examples\ch23\fig23_06\StackTest\StackTest.cs:line 37

```

### **Ил. 21.6.** Демонстрация класса Stack (окончание)

Директива `using` в строке 4 позволяет использовать класс `Stack` с неуточненным именем из пространства имен `System.Collections`. Строка 10 создает объект `Stack`. Как и следовало ожидать, класс `Stack` содержит методы `Push` и `Pop` для выполнения основных операций со стеком.

#### **Метод Push**

Метод `Push` получает аргумент `object` и вставляет его на вершину `Stack`. Если во время выполнения операции `Push` количество элементов в `Stack` (свойство `Count`) равно емкости, объект `Stack` расширяется, чтобы в нем могло храниться больше объектов. В строках 19–26 метод `Push` используется для добавления в стек четырех элементов (`bool`, `char`, `int` и `string`); после каждой операции `Push` вызывается метод `PrintStack` (строки 50–64) для вывода содержимого стека. Помните, что необобщенный класс `Stack` может хранить только ссылки на `object`, поэтому все элементы значимых типов — `bool`, `char` и `int` — неявно проходят упаковку перед добавлением в `Stack`. (Пространство имен `System.Collections.Generic` предоставляет обобщенный класс `Stack`, который поддерживает многие методы и свойства, представленные на ил. 21.6; эта версия избавляет от затрат ресурсов на упаковку и распаковку простых типов.)

#### **Метод PrintStack**

Метод `PrintStack` (строки 50–64) использует свойство `Count` класса `Stack` (реализованное для выполнения контракта интерфейса `ICollection`) для получения количества элементов в стеке. Если стек не пуст (то есть значение `Count` отлично от 0), команда `foreach` перебирает элементы и выводит содержимое стека, неявно вызывая метод `ToString` для каждого элемента.

Команда `foreach` неявно вызывает метод `GetEnumerator` класса `Stack`; программа также может *явно* вызвать этот метод, чтобы организовать перебор с использованием перечислителя.

#### **Метод Peek**

Метод `Peek` возвращает значение верхнего элемента стека, но не извлекает элемент из стека. Вызов `Peek` в строке 30 возвращает верхний объект `Stack`, после чего

программа выводит этот объект с неявным вызовом метода `ToString` для объекта. Если метод `Peek` вызывается при пустом стеке, выдается исключение `InvalidOperationException`. Блок обработки исключения в нашем случае не нужен, потому что мы знаем, что стек не пуст.

### Метод `Pop`

Метод `Pop` вызывается без аргументов — он извлекает и возвращает объект, находящийся на вершине стека. Программа в бесконечном цикле (строки 35–40) извлекает объекты из стека и выводит их, пока стек не опустеет. При вызове `Pop` для пустого стека выдается исключение `InvalidOperationException`. Блок `catch` (строки 42–46) выводит информацию об исключении, неявно вызывая метод `ToString` объекта `InvalidOperationException` для получения сообщения об ошибке и трассировки стека.



#### ТИПИЧНАЯ ОШИБКА 21.2

Попытка вызова `Peek` или `Pop` для пустого стека (объекта `Stack`, у которого свойство `Count` равно 0) приводит к выдаче исключения `InvalidOperationException`.

### Метод `Contains`

Класс `Stack` также содержит метод `Contains`, не представленный на ил. 21.6. Этот метод возвращает `true`, если заданный объект содержится в `Stack`, и `false` в противном случае.

## 21.4.3. Класс `Hashtable`

Приложение, создающее объекты новых или существующих типов, должно эффективно управлять этими объектами — в частности, выполнять операции сортировки и выборки. Сортировка и выборка информации из массива выполняется эффективно, если часть данных напрямую совпадает с ключом, а сами ключи уникальны и обладают высокой плотностью. Если программа хранит данные 100 работников с номерами социального страхования, состоящими из 9 цифр и вы хотите организовать хранение и выборку данных работников с использованием номера соцстрахования в качестве ключа, в общем случае для этого потребуется массив с 1 000 000 000 элементов, потому что существует 1 000 000 000 уникальных чисел из 9 цифр. Массив такого размера обеспечит высокую эффективность сохранения и выборки записей работников с использованием номера социального страхования в качестве индекса массива, но приведет к огромным затратам памяти.

Эта проблема встречается во многих приложениях — ключи либо относятся к неправильному типу (то есть не является неотрицательным целым числом), либо имеют правильный тип, но разбросаны в слишком большом диапазоне.

### Хеширование

В таких ситуациях нужен механизм быстрого преобразования ключей (например, номеров социального страхования и складских номеров деталей) в уникальные



индексы массивов. Когда приложению потребуется сохранить данные, этот механизм должен быстро преобразовать ключ в индекс, по которому будет сохранена запись в массиве. Выборка осуществляется аналогичным образом — располагая ключом, связанным с записью данных, приложение просто применяет к ключу преобразование, получает индекс данных в массиве и производит их выборку.

Эта схема лежит в основе *хеширования*, при котором данные хранятся в структуре данных, называемой хеш-таблицей. Значение, полученное в результате обработки индекса, — *хеш-код* — не имеет никакого реального смысла помимо своей единственной задачи: сохранения и выборки одной конкретной записи данных.

### Коллизии

В описанной схеме возможны *коллизии* (то есть преобразование двух разных ключей в один элемент массива). Так как две разные записи данных не могут находиться в одном месте, необходимо найти альтернативное хранилище для всех последующих записей, хешируемых по заданному индексу массива. Одно из возможных решений — повторное хеширование, то есть повторное применение преобразования к ключу для нахождения следующей ячейки-кандидата в массиве). Процесс хеширования проектируется таким образом, чтобы нужную ячейку можно было найти после небольшого числа хеширований.

Другой механизм использует один хеш-код для поиска первой ячейки-кандидата. Если ячейка занята, алгоритм последовательно просматривает следующие ячейки, пока не найдет свободную. Выборка выполняется аналогично; ключ хешируется один раз, а содержимое полученной ячейки проверяется на соответствие искомым данным. Если данные найдены, то поиск считается завершенным, а если нет — алгоритм последовательно просматривает следующие ячейки до нахождения нужных данных.

Самое популярное решение проблемы коллизий в хеш-таблицах заключается в создании «гнезд»; чаще всего *гнездо* представляет собой связанный список всех пар «ключ-значение», хешируемых в данную ячейку. Именно это решение реализовано в классе `Hashtable` среды .NET Framework.

### Коэффициент загрузки

Производительность схем хеширования зависит от *коэффициента загрузки* — отношения количества объектов, хранящихся в хеш-таблице, к общему количеству ячеек. С увеличением коэффициента загрузки повышается вероятность коллизий.



#### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 21.5

Коэффициент загрузки хеш-таблицы дает классический пример компромисса между скоростью и затратами памяти: повышение коэффициента загрузки обеспечивает более эффективное расходование памяти, но приложение начинает работать медленнее из-за повышения частоты коллизий. Снижение коэффициента загрузки способствует повышению скорости из-за сокращения числа коллизий, но память расходуется менее эффективно, потому что большая часть хеш-таблицы остается пустой.

Хеширование часто применяется на практике, поэтому .NET Framework предоставляет класс `Hashtable`, позволяющий программисту легко реализовать хеширование в своих приложениях. Эта концепция играет очень важную роль в объектно-ориентированном программировании: классы инкапсулируют и скрывают сложность (то есть подробности реализации) и предоставляют удобные интерфейсы для разработчика. Разработка классов, способных качественно решать эту задачу, — один из самых ценных навыков в области объектно-ориентированного программирования.


### Хеш-функция

*Хеш-функция* выполняет вычисления, которые определяют, в какой позиции хеш-таблицы должны находиться данные. Она применяется к ключу в паре «ключ-значение». Класс `Hashtable` может получать в качестве ключа любой объект, поэтому в классе `object` определяется метод `GetHashCode`, наследуемый всеми объектами. Большинство классов, объекты которых должны использоваться в качестве ключей в хеш-таблицах, переопределяют этот метод для эффективного вычисления хеш-кода для конкретного типа. Например, класс `string` при вычислении хеш-кода учитывает текущее содержимое строки.

### Использование `Hashtable`

В приложении на ил. 21.7 класс `Hashtable` используется для подсчета вхождений каждого слова в строке. В новых приложениях вместо `Hashtable` следует использовать обобщенный класс `Dictionary<K, V>` (см. раздел 17.4).

```
1 // Ил. 21.7: HashtableTest.cs
2 // Приложение подсчитывает вхождения каждого слова в строке
3 // и сохраняет результаты в хеш-таблице.
4 using System;
5 using System.Text.RegularExpressions;
6 using System.Collections;
7
8 public class HashtableTest
9 {
10     public static void Main( string[] args )
11     {
12         // Создание хеш-таблицы на основании введенных данных
13         Hashtable table = CollectWords();
14
15         // Вывод содержимого хеш-таблицы
16         DisplayHashtable( table );
17     } // Конец Main
18
19     // Создание хеш-таблицы по данным, введенным пользователем
20     private static Hashtable CollectWords()
21     {
22         Hashtable table = new Hashtable(); // Создание новой хеш-таблицы
23
24         Console.WriteLine( "Enter a string: " ); // Запрос данных
25         string input = Console.ReadLine();      // Чтение данных
```

**Ил. 21.7.** Приложение подсчитывает вхождения каждого слова в `string` и сохраняет результат в хеш-таблице (продолжение )

```

26
27 // Разбиение введенного текста на лексемы
28 string[] words = Regex.Split( input, @"\s+" );
29
30 // Обработка слов
31 foreach ( var word in words )
32 {
33     string wordKey = word.ToLower(); // Слово в нижнем регистре
34
35     // Если слово присутствует в хеш-таблице
36     if ( table.ContainsKey( wordKey ) )
37     {
38         table[ wordKey ] = ( ( int ) table[ wordKey ] ) + 1;
39     } // Конец if
40     else
41     {
42         // Включение в хеш-таблицу нового слова со счетчиком 1
43         table.Add( wordKey, 1 );
44     } // Конец foreach
45
46     return table;
47 } // Конец метода CollectWords
48
49 // Вывод содержимого хеш-таблицы
50 private static void DisplayHashtable( Hashtable table )
51 {
52     Console.WriteLine( "\nHashtable contains:\n{0,-12}{1,-12}",
53         "Key:", "Value:" );
54
55     // Генерирование вывода для каждого ключа в хеш-таблице
56     // перебором свойства Keys командой foreach.
57     foreach ( var key in table.Keys )
58     {
59         Console.WriteLine( "{0,-12}{1,-12}", key, table[ key ] );
60     }
61     Console.WriteLine( "\nsize: {0}", table.Count );
62 } // Конец метода DisplayHashtable
63 } // Конец класса HashtableTest

```

```

Enter a string:
As idle as a painted ship upon a painted ocean

```

```

Hashtable contains:
Key:    Value:
ocean   1
a        2
as       2
ship     1
upon     1
painted  2
idle     1

size: 7

```

**Ил. 21.7.** Приложение подсчитывает вхождения каждого слова в string и сохраняет результат в хеш-таблице (окончание)

В строках 4–6 содержатся директивы `using` для пространств имен `System` (для класса `Console`), `System.Text.RegularExpressions` (для класса `Regex`) и `System.Collections` (для класса `Hashtable`). Класс `HashtableTest` объявляет три статических метода. Метод `CollectWords` (строки 20–46) вводит строку и возвращает объект `Hashtable`, значения которого определяют количество вхождений каждого слова (само слово используется в качестве ключа). Метод `DisplayHashtable` (строки 49–60) выводит переданный объект `Hashtable` по столбцам. Метод `Main` (строки 10–17) просто вызывает `CollectWords` (строка 13) и передает объект `Hashtable`, возвращенный `CollectWords`, методу `DisplayHashtable` в строке 16.

### Метод `CollectWords`

Метод `CollectWords` (строки 20–46) начинается с инициализации локальной переменной `table` новым объектом `Hashtable` (строка 22), у которого коэффициент загрузки по умолчанию равен 1.0. Когда `Hashtable` достигает заданного коэффициента загрузки, емкость хеш-таблицы автоматически увеличивается (эта подробность реализации остается скрытой от клиентов класса). В строках 24–25 пользователь по запросу вводит строку. В строке 28 статический метод `Split` класса `Regex` используется для разбиения строки по пропуску. Так создается массив «слов», который сохраняется в локальной переменной `words`.

### Методы `ContainsKey` и `Add` класса `Hashtable`

В строках 31–43 перебираются все элементы массива `words`. Каждое слово преобразуется к нижнему регистру методом `ToLower` класса `string`, после чего сохраняется в переменной `wordKey` (строка 33). Затем строка 36 вызывает метод `ContainsKey` класса `Hashtable` для проверки присутствия слова в хеш-таблице (если слово уже встречалось в строке ранее). Если `Hashtable` не содержит значения для указанного слова, строка 42 использует метод `Add` класса `Hashtable` для создания нового элемента в хеш-таблице; его ключом является слово, преобразованное к нижнему регистру, а значением — объект со значением 1. При передаче `Add` целочисленного значения 1 выполняется автоматическая упаковка, потому что в хеш-таблице и ключ и значение сохраняются по ссылкам на тип `object`.



#### ТИПИЧНАЯ ОШИБКА 21.3

Попытка вызова метода `Add` для добавления ключа, уже существующего в хеш-таблице, приводит к исключению `ArgumentException`.

### Индексатор `Hashtable`

Если слово уже представлено ключом в хеш-таблице, то строка 38 использует индексатор `Hashtable` для получения и присваивания значения, связанного с ключом (счетчика вхождений слова) в хеш-таблице. Сначала мы преобразуем значение, полученное `get`-методом доступа, из `object` в `int`. При этом значение распаковывается, чтобы его можно было увеличить на 1. Затем `set`-метод индексатора присваивает значению, связанному с ключом, увеличенный счетчик, который также проходит неявную упаковку для сохранения в хеш-таблице.

Вызов `get`-метода индексатора `Hashtable` с ключом, отсутствующим в таблице, возвращает ссылку `null`. Использование `set`-метода с несуществующим ключом приводит к созданию нового элемента, как если бы в программе был вызван метод `Add`.

### Метод `DisplayHashtable`

В строке 45 хеш-таблица возвращается методу `Main`, который передает ее методу `DisplayHashtable` (строки 49–60), который выводит все элементы. Этот метод использует свойство `Keys`, доступное только для чтения (строка 56), для получения объекта `ICollection` со всеми ключами. Поскольку `ICollection` расширяет `IEnumerable`, коллекцию можно использовать в команде `foreach` (строки 56–57) для перебора ключей хеш-таблицы. Цикл выводит каждый ключ и значение в хеш-таблице с использованием переменной-итератора и `get`-метода таблицы. Каждый ключ и значение выводятся в поле ширины -12. Отрицательная ширина указывает на то, что вывод выравнивается по левому краю поля. Хеш-таблица не отсортирована, так что пары «ключ-значение» не выводятся в каком-то конкретном порядке. В строке 59 свойство `Count` объекта `Hashtable` используется для получения количества пар «ключ-значение» в хеш-таблице.

### `DictionaryEntry` и `IDictionary`

В строках 56–57 также можно было использовать команду `foreach` с самим объектом `Hashtable` вместо свойства `Keys`. При использовании команды `foreach` с объектом `Hashtable` переменная-итератор будет относиться к типу `DictionaryEntry`. Перечислитель `Hashtable` (или любого другого класса, реализующего `IDictionary`) использует структуру `DictionaryEntry` для хранения пар «ключ-значение». Эта структура предоставляет свойства `Key` и `Value` для получения ключа и значения текущего элемента. Если ключи вас не интересуют, класс `Hashtable` также предоставляет доступное только для чтения свойство `Values`, которое возвращает объект `ICollection` со всеми значениями, хранимыми в `Hashtable`. Это свойство можно использовать для перебора всех значений, хранящихся в `Hashtable`, без учета позиции их хранения.

### Недостатки необобщенных коллекций

В приложении для подсчета слов на ил. 21.7 объект `Hashtable` хранит ключ и данные по ссылкам на `object`, хотя должны храниться только ключи `string` и значения `int`. В результате код становится более громоздким: например, в строке 38 приходится распаковывать и упаковывать данные `int`, хранящиеся в `Hashtable`, при каждом увеличении счетчика для конкретного ключа, а это неэффективно. Аналогичная проблема существует в строке 56 — переменная-итератор команды `foreach` представляет собой ссылку на `object`. Чтобы использовать какие-либо методы, специфические для класса `string`, необходимо выполнить понижающее преобразование.

Такие преобразования могут приводить к коварным ошибкам. Предположим, вы решили сделать код ил. 21.7 более понятным и использовать `set`-метод индексатора вместо метода `Add` для добавления пары «ключ-значение» в строке 42, но случайно ввели следующую команду:

```
table[ wordKey ] = wordKey; // Инициализируется значением 1
```

Эта команда создает новый элемент с ключом `string` и значением `string` (вместо значения `int`, равного 1). Хотя приложение откомпилируется, понятно, что оно будет работать неправильно. Если слово встречается дважды, строка 38 попытается преобразовать строку в `int`, а во время выполнения будет выдано исключение `InvalidCastException`. Ошибка во время выполнения сообщит, что проблема находится в строке 38, в которой произошло исключение, а не в строке 42. Это дополнительно усложнит поиск и исправление ошибки, особенно в больших приложениях, в которых исключение может произойти в другом файле — и даже в другой сборке.

## 21.5. Обобщенные коллекции

Пространство имен `System.Collections.Generic` содержит обобщенные классы, позволяющие создавать коллекции заданных типов. Как было показано на ил. 21.2, многие из них представляют собой обобщенные аналоги необобщенных коллекций. Пара классов реализует новые структуры данных. Далее будут представлены обобщенные коллекции `SortedDictionary` и `LinkedList`.

### 21.5.1. Обобщенный класс `SortedDictionary`

*Словарем* называется коллекция пар «ключ-значение». Хеш-таблица — один из способов реализации словаря. .NET Framework предоставляет несколько реализаций словарей (и обобщенных, и необобщенных); все они реализуют интерфейс `IDictionary` (см. ил. 21.1). Приложение на ил. 21.8 представляет собой измененную версию ил. 21.7 с использованием обобщенного класса `SortedDictionary`. Обобщенный класс `SortedDictionary` не использует хеш-таблицу, а хранит пары «ключ-значение» в бинарном дереве поиска (см. раздел 19.7). Как подсказывает имя класса, элементы `SortedDictionary` сортируются в дереве по ключу. Так как ключ реализует обобщенный интерфейс `IComparable<T>`, класс `SortedDictionary` использует результаты метода `CompareTo` интерфейса `IComparable<T>` для сортировки ключей. Обратите внимание: несмотря на эти подробности реализации, при работе с классами `Hashtable` и `SortedDictionary` мы используем одни и те же открытые методы, свойства и индексаторы. Собственно, если не считать синтаксиса обобщений, ил. 21.8 почти не отличается от ил. 21.7. В этом проявляется элегантность объектно-ориентированного программирования.

```
1 // Ил. 21.8: SortedDictionaryTest.cs
2 // Приложение подсчитывает вхождения каждого слова в строке
3 // и сохраняет результаты в обобщенном отсортированном словаре.
4 using System;
5 using System.Text.RegularExpressions;
6 using System.Collections.Generic;
```

**Ил. 21.8.** Приложение подсчитывает вхождения каждого слова в `string` и сохраняет результат в отсортированном словаре (продолжение ↗)

```

7
8 public class SortedDictionaryTest
9 {
10     public static void Main( string[] args )
11     {
12         // Создание отсортированного словаря на основании введенных данных
13         SortedDictionary< string, int > dictionary = CollectWords();
14
15         // Вывод содержимого отсортированного словаря
16         DisplayDictionary( dictionary );
17     } // Конец Main
18
19     // Создание отсортированного словаря на основании введенных данных
20     private static SortedDictionary< string, int > CollectWords()
21     {
22         // Создание нового отсортированного словаря
23         SortedDictionary< string, int > dictionary =
24             new SortedDictionary< string, int >();
25
26         Console.WriteLine( "Enter a string: " ); // Запрос данных
27         string input = Console.ReadLine();       // Чтение данных
28
29         // Разбиение введенного текста на лексемы
30         string[] words = Regex.Split( input, @"\s+" );
31
32         // Обработка слов
33         foreach ( var word in words )
34         {
35             string wordKey = word.ToLower(); // Слово в нижнем регистре
36
37             // Если слово присутствует в словаре
38             if ( dictionary.ContainsKey( wordKey ) )
39             {
40                 ++dictionary[ wordKey ];
41             } // Конец if
42             else
43             {
44                 // Включение в словарь нового слова со счетчиком 1
45                 dictionary.Add( wordKey, 1 );
46             } // Конец foreach
47
48             return dictionary;
49         } // Конец метода CollectWords
50
51         // Вывод содержимого словаря
52         private static void DisplayDictionary< K, V >(
53             SortedDictionary< K, V > dictionary )
54         {
55             Console.WriteLine( "\nSorted dictionary contains:\n{0,-12}{1,-12}",
56                 "Key:", "Value:" );
57
58             // Генерирование вывода для каждого ключа в отсортированном
59             // словаре перебором свойства Keys командой foreach.
60             foreach ( K key in dictionary.Keys )
61                 Console.WriteLine( "{0,-12}{1,-12}", key, dictionary[ key ] );

```

**Ил. 21.8.** Приложение подсчитывает вхождения каждого слова в string и сохраняет результат в отсортированном словаре (продолжение ☞)

```
61
62     Console.WriteLine( "\nsize: {0}", dictionary.Count );
63 } // Конец метода DisplayDictionary
64 } // Конец класса SortedDictionaryTest
```

```
Enter a string:
We few, we happy few, we band of brothers

Sorted dictionary contains:
Key:      Value:
band      1
brothers  1
few,      2
happy     1
of        1
we        3
size: 6
```

**Ил. 21.8.** Приложение подсчитывает вхождения каждого слова в string и сохраняет результат в отсортированном словаре (окончание)

Строка 6 содержит директиву using для пространства имен System.Collections.Generic, в котором находится класс SortedDictionary. Обобщенный класс SortedDictionary получает два аргумента-типа — первый задает тип ключа (string), а второй — тип значения (int). Чтобы создать словарь с ключами string, ассоциированными со значениями int, мы просто заменили имя Hashtable в строке 13 и строках 23–24 именем SortedDictionary<string, int>. Теперь компилятор может обнаружить попытку сохранения в словаре объекта неправильного типа и сообщить о ней разработчику. Кроме того, поскольку компилятор знает, что структура данных содержит значения int, ему уже не приходится выполнять понижающее преобразование в строке 40. Это позволяет использовать в строке 40 более компактный синтаксис инкремента (++). Код, полученный в результате этих изменений, обеспечивает проверку безопасности типов во время компиляции.

Статический метод DisplayDictionary (строки 51–63) был модифицирован с учетом обобщения: он получает параметры-типы K и V. Эти параметры, используемые в строке 52, показывают, что DisplayDictionary получает объект SortedDictionary с ключами типа K и значениями типа V. Параметр-тип K снова используется в строке 59 как тип ключа итерации. Это превосходный пример повторного использования кода: если позднее мы захотим изменить приложение, чтобы оно подсчитывало количество вхождений каждого символа в строке, метод DisplayDictionary может получать аргумент типа SortedDictionary<char, int> без каких-либо изменений. Теперь выводимые пары «ключ-значение» упорядочиваются по ключу, как показано на ил. 21.8.



### ПОВЫШАЕМ ЭФФЕКТИВНОСТЬ 21.6

Так как класс SortedDictionary хранит свои элементы отсортированными в бинарном дереве, операция получения или вставки пары «ключ-значение» выполняется за время  $O(\log n)$ , то есть относительно быстро по сравнению с линейным поиском и последующей вставкой.



**ТИПИЧНАЯ ОШИБКА 21.4**

При вызове `get`-метода индексатора `SortedDictionary` с ключом, не существующим в коллекции, возникает исключение `KeyNotFoundException`. Его поведение отличается от поведения `get`-метода индексатора `Hashtable`, который вернет `null`.

## 21.5.2. Обобщенный класс `LinkedList`

Обобщенный класс `LinkedList` представляет собой двусвязный список с возможностью перемещения в прямом и обратном направлении по узлам обобщенного класса `LinkedListNode`. Каждый узел содержит свойство `Value` и свойства `Previous` и `Next`, доступные только для чтения. Тип свойства `Value` соответствует единственному параметру-типу `LinkedList`. Свойство `Previous` возвращает ссылку на предыдущий узел связанного списка (или `null` для первого узла в списке). Аналогичным образом свойство `Next` возвращает ссылку на следующий узел связанного списка (или `null` для последнего узла в списке). Некоторые операции со связанными списками представлены на ил. 21.9.

```

1  // Ил. 21.9: LinkedListTest.cs
2  // Использование LinkedList.
3  using System;
4  using System.Collections.Generic;
5
6  public class LinkedListTest
7  {
8      private static readonly string[] colors = { "black", "yellow",
9          "green", "blue", "violet", "silver" };
10     private static readonly string[] colors2 = { "gold", "white",
11         "brown", "blue", "gray" };
12
13     // Создание и обработка объектов LinkedList
14     public static void Main( string[] args )
15     {
16         LinkedList< string > list1 = new LinkedList< string >();
17
18         // Добавление элементов в первый связанный список
19         foreach ( var color in colors )
20             list1.AddLast( color );
21
22         // Добавление элементов во второй связанный список через конструктор
23         LinkedList< string > list2 = new LinkedList< string >( colors2 );
24
25         Concatenate( list1, list2 ); // Присоединение list2 к list1
26         PrintList( list1 );          // Вывод элементов list1
27
28         Console.WriteLine( "\nConverting strings in list1 to uppercase\n" );
29         ToUppercaseStrings( list1 ); // Преобразование к верхнему регистру
30         PrintList( list1 );          // Вывод элементов list1

```

**Ил. 21.9.** Использование `LinkedList` (продолжение ➤)

```

31
32     Console.WriteLine( "\nDeleting strings between BLACK and BROWN\n" );
33     RemoveItemsBetween( list1, "BLACK", "BROWN" );
34
35     PrintList( list1 );           // Вывод элементов list1
36     PrintReversedList( list1 ); // Вывод списка в обратном порядке
37 } // Конец Main
38
39 // Вывод содержимого списка
40 private static void PrintList< T >( LinkedList< T > list )
41 {
42     Console.WriteLine( "Linked list: " );
43
44     foreach ( T value in list )
45         Console.Write( "{0} ", value );
46
47     Console.WriteLine();
48 } // Конец метода PrintList
49
50 // Присоединение второго списка в конец первого
51 private static void Concatenate< T >( LinkedList< T > list1,
52     LinkedList< T > list2 )
53 {
54     // Конкатенация списков выполняется копированием
55     // значений элементов из второго списка в первый
56     foreach ( T value in list2 )
57         list1.AddLast( value ); // Добавить новый узел
58 } // Конец метода Concatenate
59
60 // Поиск объектов string и преобразование их к верхнему регистру
61 private static void ToUppercaseStrings( LinkedList< string > list )
62 {
63     // Перебор списка по узлам
64     LinkedListNode< string > currentNode = list.First;
65
66     while ( currentNode != null )
67     {
68         string color = currentNode.Value; // Получить значение в узле
69         currentNode.Value = color.ToUpper(); // В верхнем регистре
70
71         currentNode = currentNode.Next; // Получить следующий узел
72     } // Конец while
73 } // Конец метода ToUppercaseStrings
74
75 // Удаление элементов между двумя заданными позициями
76 private static void RemoveItemsBetween< T >( LinkedList< T > list,
77     T startItem, T endItem )
78 {
79     // Получение узлов, соответствующих начальной конечной позиции
80     LinkedListNode< T > currentNode = list.Find( startItem );
81     LinkedListNode< T > endNode = list.Find( endItem );
82
83     // Удаление элементов от начального до последнего
84     // или до конца связанного списка

```

**Ил. 21.9.** Использование LinkedList (продолжение ↗)

```

85     while ( ( currentNode.Next != null ) &&
86             ( currentNode.Next != endNode ) )
87     {
88         list.Remove( currentNode.Next ); // Удалить следующий узел
89     } // Конец while
90 } // Конец метода RemoveItemsBetween
91
92 // Вывод списка в обратном порядке
93 private static void PrintReversedList< T >( LinkedList< T > list )
94 {
95     Console.WriteLine( "Reversed List:" );
96
97     // Перебор списка по узлам
98     LinkedListNode< T > currentNode = list.Last;
99
100    while ( currentNode != null )
101    {
102        Console.Write( "{0} ", currentNode.Value );
103        currentNode = currentNode.Previous; // Получить предыдущий узел
104    } // Конец while
105
106    Console.WriteLine();
107 } // Конец метода PrintReversedList
108 } // Конец класса LinkedListTest

```

```

Linked list:
black yellow green blue violet silver gold white brown blue gray

Converting strings in list1 to uppercase

Linked list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY

Deleting strings between BLACK and BROWN

Linked list:
BLACK BROWN BLUE GRAY
Reversed List:
GRAY BLUE BROWN BLACK

```

### Ил. 21.9. Использование LinkedList (окончание)

Директива `using` в строке 4 позволяет использовать класс `LinkedList` без уточнения имени. В строках 16–23 создаются объекты `LinkedLists` с данными `string`, которые заполняются содержимым массивов `colors` и `colors2` соответственно. Обобщенный класс `LinkedList` имеет один параметр-тип, для которого в данном примере передается аргумент-тип `string` (строки 16 и 23).

### Методы `AddLast` и `AddFirst` класса `LinkedList`

В приложении продемонстрированы два способа заполнения списков. В строках 19–20 список `list1` заполняется командой `foreach` и методом `AddLast`. Метод `AddLast` создает новый объект `LinkedListNode` (со значением, доступным в свойстве `Value`) и присоединяет его в конец списка. Также существует метод `AddFirst`, который

вставляет узел в начало списка. В строке 23 вызывается конструктор, получающий параметр `IEnumerable<string>`. Все массивы неявно наследуют от обобщенных интерфейсов `IList` и `IEnumerable` с аргументом-типом, соответствующим типу массива, так что массив `colors2` реализует `IEnumerable<string>`. Параметр-тип этого обобщенного интерфейса `IEnumerable` соответствует параметру-типу обобщенного объекта `LinkedList`. Вызов конструктора копирует содержимое массива `colors2` в `list2`.

### Методы для тестирования класса `LinkedList`

В строке 25 вызывается обобщенный метод `Concatenate` (строки 51–58), присоединяющий все элементы `list2` в конец `list1`. Строка 26 вызывает метод `PrintList` (строки 40–48) для вывода содержимого `list1`. Строка 29 вызывает метод `ToUppercaseStrings` (строки 61–73) для преобразования каждого элемента `string` к верхнему регистру, после чего в строке 30 метод `PrintList` вызывается снова для вывода измененных строк.

В строке 33 вызывается метод `RemoveItemsBetween` (строки 76–90) для удаления элементов между значениями "BLACK" и "BROWN" (не включая граничные элементы). В строке 35 список выводится повторно, после чего строка 36 вызывает метод `PrintReversedList` (строки 93–107) для вывода списка в обратном порядке.

### Обобщенный метод `Concatenate`

Обобщенный метод `Concatenate` (строки 51–58) перебирает `list2` командой `foreach` и вызывает метод `AddLast` для присоединения каждого значения в конец `list1`. Перечислитель класса `LinkedList` перебирает значения узлов, а не сами узлы, поэтому переменная-итератор имеет тип `T`. Обратите внимание: при этом для каждого узла из `list2` в `list1` создается новый узел. Один объект `LinkedListNode` не может принадлежать более чем одному списку `LinkedList`. Если вы хотите, чтобы одни данные принадлежали сразу нескольким объектам `LinkedList`, необходимо скопировать узел в каждом списке для предотвращения исключений `InvalidOperationException`.

### Обобщенный метод `PrintList` и метод `ToUppercaseStrings`

Обобщенный метод `PrintList` (строки 40–48) похожим образом использует команду `foreach` для перебора и вывода значений `LinkedList`. Метод `ToUppercaseStrings` (строки 61–73) получает связанный список `string` и преобразует каждое значение `string` к верхнему регистру. Этот метод заменяет объекты `string`, хранящиеся в списке, поэтому мы не можем использовать перечислитель (с командой `foreach`), как в двух предыдущих методах. Вместо этого мы получаем первый объект `LinkedListNode` из свойства `First` (строка 64) и используем команду `while` для перебора списка в цикле (строки 66–72). Каждая итерация команды `while` получает и обновляет содержимое `currentNode` через свойство `Value`, используя метод `ToUpper` класса `string` для создания версии `color` в верхнем регистре. В конце каждой итерации текущим делается следующий узел списка, для чего `currentNode` присваивается узел из его собственного свойства `Next` (строка 71). Свойство `Next` последнего узла в списке содержит `null`, поэтому при переходе `while` за границу списка цикл завершается.

### Метод `ToUppercaseStrings`

Объявлять метод `ToUppercaseStrings` обобщенным не имеет смысла, потому что он применяет к значениям в узлах методы, специфические для `string`. Методам `PrintList` (строки 40–48) и `Concatenate` (строки 51–58) не нужно использовать методы, относящиеся к `string`, поэтому их можно объявить с обобщенными параметрами-типами для расширения возможностей повторного использования кода.

### Обобщенный метод `RemoveItemsBetween`

Обобщенный метод `RemoveItemsBetween` (строки 76–90) удаляет диапазон элементов между двумя узлами. В строках 80–81 два «граничных» узла определяются при помощи метода `Find`. Этот метод выполняет линейный поиск по списку и возвращает первый узел, содержащий значение, равное переданному аргументу. Если значение не найдено, метод `Find` возвращает `null`. Узел перед началом диапазона сохраняется в локальной переменной `currentNode`, а узел после конца диапазона — в переменной `endNode`.

Строки 85–89 удаляют все элементы между `currentNode` и `endNode`. При каждой итерации цикла узел, следующий за `currentNode`, удаляется методом `Remove` (строка 88). Метод `Remove` получает `LinkedListNode`, «вырезает» этот узел из `LinkedList` и приводит в порядок ссылки соседних узлов. После вызова `Remove` свойство `Next` объекта `currentNode` содержит узел, следующий за удаленным, а в свойстве `Previous` хранится `currentNode`. Команда `while` продолжает выполняться в цикле до того момента, пока между `currentNode` и `endNode` не останется ни одного узла или пока `currentNode` не станет последним узлом в списке. (Также существует перегруженная версия метода `Remove`, которая выполняет линейный поиск заданного значения и удаляет из списка первый узел, в котором оно содержится.)

### Метод `PrintReversedList`

Метод `PrintReversedList` (строки 93–107) выводит список в обратном порядке, используя «ручной» перебор узлов. Строка 98 получает последний элемент списка из свойства `Last` и сохраняет его в `currentNode`. Команда `while` в строках 100–104 перебирает список в обратном порядке, перемещая ссылку `currentNode` к предыдущему узлу в конце каждой итерации; при выходе за начало списка цикл завершается. Обратите внимание на сходство этого кода с кодом в строках 64–72, который перебирает список от начала к концу.

## 21.6. Ковариантность и контрвариантность для обобщенных типов

C# поддерживает ковариантность и контрвариантность обобщенных интерфейсов и типов делегатов. Мы рассмотрим эти концепции на примере массивов, которые всегда были ковариантными и контрвариантными в C#.

### Ковариантность массивов

Вспомните иерархию классов `Employee` из раздела 12.5, которая состояла из базового класса `Employee` и производных классов `SalariedEmployee`, `CommissionEmployee` и `BasePlusCommissionEmployee`. Допустим, в программе присутствуют следующие объявления:

```
SalariedEmployee[] salariedEmployees = {  
    new SalariedEmployee( "Bob", "Blue", "111-11-1111", 800M ),  
    new SalariedEmployee( "Rachel", "Red", "222-22-2222", 1234M ) };  
Employee[] employees;
```

Такая программа может содержать следующие команды:

```
employees = salariedEmployees;
```

Хотя тип массива `SalariedEmployee[]` не наследует напрямую от типа массива `Employee[]`, предыдущая команда допустима, потому что класс `SalariedEmployee` является производным от `Employee`.

Также представьте следующий метод, который выводит строковое представление каждого объекта `Employee` из своего параметра-массива `employees`:

```
void PrintEmployees( Employee[] employees )
```

Этот метод можно вызвать для массива `SalariedEmployees`, как в следующей команде:

```
PrintEmployees( salariedEmployees );
```

Этот метод правильно выведет строковое представление всех объектов `SalariedEmployee` из аргумента-массива. Другим примером ковариантности служит присваивание массива с типом производного класса переменной массива с типом базового класса.

### Ковариантность в обобщенных типах

Ковариантность также работает с некоторыми обобщенными типами интерфейсов и делегатов, включая интерфейс `IEnumerable<T>`, реализуемый массивами и обобщенными коллекциями. Используя приведенное ранее объявление массива `salariedEmployees`, рассмотрим следующую команду:

```
IEnumerable< Employee > employees = salariedEmployees;
```

До Visual C# 2010 эта команда вызывала ошибку компиляции. Теперь интерфейс `IEnumerable<T>` стал ковариантным, поэтому такая команда разрешена. Если внести следующие изменения в метод `PrintEmployees`:

```
void PrintEmployees( IEnumerable< Employee > employees )
```

то его можно будет вызывать с массивом объектов `SalariedEmployee`, потому что этот массив реализует интерфейс `IEnumerable<SalariedEmployee>`, `SalariedEmployee` является частным случаем `Employee`, а интерфейс `IEnumerable<T>` ковариантен. Такая ковариантность работает только со ссылочными типами, связанными отношениями в иерархиях классов.

### Контрвариантность в массивах

Ранее было показано, что массив типов производного класса (`salariedEmployees`) можно присвоить переменной-массиву типа базового класса (`employees`). Теперь возьмем следующую команду, которая всегда работала в C#:

```
SalariedEmployee[] salariedEmployees2 =  
( SalariedEmployee[] ) employees;
```

Из приведенных выше команд мы знаем, что переменная `employees` (массив с элементами `Employee`) в настоящее время ссылается на массив `SalariedEmployees`. Использование оператора преобразования типа для присваивания `employees` (массива с элементами типа базового класса) переменной `salariedEmployees2` (массива с элементами типа производного класса) является примером контрвариантности. Если `employees` не является массивом `SalariedEmployees`, преобразование типа завершится неудачей во время выполнения.

### Контрвариантность в обобщенных типах

Чтобы понять, как работает контрвариантность в обобщенных типах, рассмотрим коллекцию `SortedSet` с элементами `SalariedEmployees`. Класс `SortedSet<T>` хранит множество объектов в отсортированном порядке (дубликаты запрещены). Объекты, помещенные в `SortedSet`, должны реализовать интерфейс `Comparable<T>`. Объекты классов, не реализующих этот интерфейс, можно сравнивать при помощи объекта, реализующего интерфейс `IComparer<T>`. Метод `Compare` этого интерфейса сравнивает два своих аргумента и возвращает 0, если они равны, отрицательное целое число, если первый объект меньше второго, и положительное целое число, если первый объект больше второго.

Классы иерархии `Employee` не реализуют `Comparable<T>`. Допустим, мы хотим отсортировать объекты `Employee` по номеру социального страхования. Для сравнения двух объектов `Employee` реализуется следующий класс:

```
class EmployeeComparer : IComparer< Employee >  
{  
    int IComparer< Employee >.Compare( Employee a, Employee b)  
    {  
        return a.SocialSecurityNumber.CompareTo(  
            b.SocialSecurityNumber );  
    } // Конец метода Compare  
} // Конец класса EmployeeComparer
```

Метод `Compare` возвращает результат сравнения двух номеров социального страхования `Employee` с использованием метода `CompareTo` класса `string`.

А теперь рассмотрим следующую команду создания `SortedSet`:

```
SortedSet< SalariedEmployee > set =  
    new SortedSet< SalariedEmployee >( new EmployeeComparer() );
```

Если аргумент-тип не реализует `Comparable<T>`, вы должны предоставить подходящий объект `IComparer<T>` для сравнения объектов, которые будут помещаться

в `SortedSet`. Так как мы создаем коллекцию `SortedSet` с элементами `SalariedEmployee`, компилятор ожидает, что объект `IComparer<T>` реализует `IComparer<SalariedEmployee>`. Вместо этого мы предоставляем объект, реализующий `IComparer<Employee>`. Компилятор позволяет предоставить реализацию `IComparer` для типа базового класса там, где ожидается реализация `IComparer` для типа производного класса, потому что интерфейс `IComparer<T>` поддерживает контрвариантность.

### Веб-ресурсы

Список ковариантных и контрвариантных типов интерфейсов находится по адресу [msdn.microsoft.com/en-us/library/dd799517.aspx#VariantList](http://msdn.microsoft.com/en-us/library/dd799517.aspx#VariantList)

Вы также можете создавать собственные вариантыные типы. За дополнительной информацией обращайтесь по адресу

[msdn.microsoft.com/en-us/library/dd997386.aspx](http://msdn.microsoft.com/en-us/library/dd997386.aspx)

## 21.7. Итоги

В этой главе представлены классы коллекций .NET Framework. Мы рассмотрели иерархию интерфейсов, реализуемых многими классами коллекций. Вы научились использовать класс `Array` для выполнения манипуляций с массивами и узнали, что пространства имен `System.Collections` и `System.Collections.Generic` содержат много необобщенных и обобщенных классов коллекций соответственно. Мы рассмотрели необобщенные классы `ArrayList`, `Stack` и `Hashtable`, а также обобщенные классы `SortedDictionary` и `LinkedList`. При этом структуры данных были рассмотрены более подробно: мы обсудили динамически расширяемые коллекции, схемы хеширования и две реализации словаря. Вы узнали о преимуществах обобщенных коллекций перед их необобщенными аналогами, научились использовать перечислители для обхода этих структур данных и получения их содержимого. Мы рассмотрели применение команды `foreach` для многих классов Framework Class Library и выяснили, что работа команды основана на создании «скрытых» перечислителей для перебора коллекций.



# 22

# Базы данных и LINQ

## 22.1. Введение

*База данных* представляет собой упорядоченный набор данных. *Система управления базами данных* (СУБД) предоставляет средства для хранения, упорядочения, выборки и изменения данных. Большинство современных СУБД работает с реляционными базами данных, в которых данные упорядочиваются в таблицы, состоящие из строк и столбцов.

Некоторые популярные коммерческие СУБД — Microsoft SQL Server, Oracle, Sybase и IBM DB2. СУБД с открытым кодом PostgreSQL и MySQL могут загружаться и свободно использоваться всеми желающими. В этой главе используется бесплатная СУБД Microsoft SQL Server Express, устанавливаемая в составе Visual Studio. Ее также можно отдельно загрузить с сайта Microsoft ([www.microsoft.com/express/sql](http://www.microsoft.com/express/sql)).

### SQL Server Express

SQL Server Express предоставляет многие возможности полного (коммерческого) продукта Microsoft SQL Server, но обладает рядом ограничений — например, максимальный размер базы данных составляет 10 Гбайт. Файл базы данных SQL Server Express легко переносится в полную версию SQL Server — мы сделали это на своем сайте *deitel.com*, когда наша база данных стала слишком велика для SQL Server Express. Дополнительная информация о версиях SQL Server доступна по адресу [bit.ly/SQLServerEditions](http://bit.ly/SQLServerEditions).

Версия SQL Server Express, входящая в поставку Visual Studio Express 2012 for Windows Desktop, называется SQL Server Express 2012 LocalDB. Она предназначена для разработки и тестирования приложений на вашем компьютере.

### Язык SQL

Язык SQL (Structured Query Language) является международным стандартом выполнения запросов к реляционным базам данных (то есть запросов на получение информации, удовлетворяющей заданному критерию) и обработки данных. Программы, обращавшиеся к реляционным данным, традиционно передавали СУБД запросы SQL в виде строк, после чего обрабатывали результаты.

### LINQ to Entities и ADO.NET Entity Framework

Логическим расширением запроса и обработки данных в базах данных является выполнение тех же операций с произвольными источниками данных — массивами, коллекциями (например, коллекцией `Items` элемента управления `ListBox`) и файлами. В главе 9 была представлена технология LINQ to Objects, использовавшаяся для обработки данных в массивах. LINQ to Entities позволяет работать с данными, хранящимися в реляционных базах данных, — в нашем случае базах данных SQL Server Express. Как и в случае с LINQ to Objects, IDE предоставляет поддержку IntelliSense для запросов LINQ to Entities.

Среда ADO.NET Entity Framework (часто обозначаемая сокращением EF) позволяет приложениям взаимодействовать с данными в разных формах, включая данные, хранимые в реляционных базах данных. Разработчик использует ADO.NET Entity Framework и Visual Studio для создания так называемой *модели данных сущностей*, представляющей базу данных, после чего при помощи LINQ to Entities работает с объектами модели сущностей. И хотя в этой главе мы будем работать с базами данных SQL Server Express, ADO.NET Entity Framework поддерживает большинство популярных систем управления базами данных. ADO.NET Entity Framework незаметно для разработчика генерирует команды SQL для взаимодействия с базой данных.

В этой главе мы сначала рассмотрим общие концепции реляционных баз данных, после чего реализуем несколько приложений с использованием ADO.NET Entity Framework, LINQ to Entities и средств IDE для работы с базами данных. В следующих главах будут представлены реальные приложения для баз данных и LINQ to Entities — например, гостевая книга и книжный магазин на базе веб-технологий. Базы данных занимают центральное место в большинстве приложений промышленного уровня.

### LINQ to SQL и LINQ to Entities

В предыдущем издании книги рассматривалась технология LINQ to SQL. В 2008 году компания Microsoft прекратила дальнейшую разработку LINQ to SQL в пользу новых и более мощных технологий LINQ to Entities и ADO.NET Entity Framework.

## 22.2. Реляционные базы данных

В реляционной базе данных информация упорядочивается в таблицы. На ил. 22.1 приведен пример таблицы `Employees`, которая может использоваться в системе учета кадров. В таблице хранятся атрибуты работников. Таблица состоит из строк (также называемых *записями*) и столбцов (также называемых *полями*), в которых хранятся значения. В нашем примере таблица состоит из шести строк (по одной на работника) и пяти столбцов (по одному на атрибут). В атрибутах хранятся идентификатор работника, имя, отдел, зарплата и местонахождение работника. Столбец `ID` каждой записи содержит первичный ключ таблицы — столбец (или группу столбцов) с уникальным значением, которое не может повторяться в других столбцах. Уникальность гарантирует, что каждое значение первичного ключа

может использоваться для однозначной идентификации одной строки. Первичный ключ, состоящий из двух и более столбцов, называется *составным ключом*. LINQ to Entities требует, чтобы каждая таблица содержала первичный ключ для поддержки обновления данных в таблицах. Строки на ил. 22.1 выводятся по возрастанию первичного ключа, однако они также могут выводиться в порядке убывания или вообще без какого-либо определенного порядка.

Таблица Employees

	ID	Name	Department	Salary	Location
	23603	Jones	413	1100	New Jersey
	24568	Kerwin	413	2000	New Jersey
Строка {	34589	Larson	642	1800	Los Angeles
	35761	Myers	611	1400	Orlando
	47132	Neumann	413	9000	New Jersey
	78321	Stephens	611	8500	Orlando
	Первичный ключ		Столбец		

**Ил. 22.1.** Таблица Employees с тестовыми данными

Каждый столбец представляет атрибут данных. Значения некоторых столбцов могут повторяться в разных записях. Например, в трех разных строках столбец `Department` таблицы `Employees` содержит число 413, которое показывает, что все эти работники принадлежат одному отделу.

### Выбор подмножеств данных

Технология LINQ to Entities может использоваться для определения запросов, выбирающих подмножество данных из таблицы. Например, программа может выбрать из таблицы `Employees` данные для получения результатов запроса с информацией о местонахождении каждого отдела, упорядоченных по возрастанию кодов отделов (ил. 22.2).

Department	Location
413	New Jersey
611	Orlando
642	Los Angeles

**Ил. 22.2.** Информация о разделах и их местонахождении из таблицы `Employees` (без повторений)

## 22.3. База данных Books

Теперь мы рассмотрим простую базу данных `Books`, в которой хранится информация о некоторых книгах Deitel. Начнем с обзора таблиц базы данных. Совокупность таблиц базы данных, их полей и отношений между ними называется *схемой* базы данных. ADO.NET Entity Framework использует схему базы данных для

определения классов, обеспечивающих взаимодействие с базой данных. В разделах 22.5–22.7 приведены примеры работы с базой данных Books. Файл базы данных Books.mdf включен в примеры этой главы. Файлы баз данных SQL Server имеют расширение .mdf.

### Таблица Authors базы данных Books

База данных состоит из трех таблиц: Authors, Titles и AuthorISBN. Таблица Authors (ил. 22.3) состоит из трех столбцов с уникальным идентификатором автора, именем и фамилией. На ил. 22.4 представлены данные из таблицы Authors.

Столбец	Описание
AuthorID	Идентификационный номер автора. В базе данных Books этот целочисленный столбец определяется как идентифицирующий столбец, также называемый столбцом-счетчиком — для каждой записи, вставляемой в таблицу, значение AuthorID автоматически увеличивается на 1, чтобы у каждой строки это значение было уникальным. Столбец является первичным ключом
FirstName	Имя автора (строка)
LastName	Фамилия автора (строка)

**Ил. 22.3.** Таблица Authors базы данных Books

AuthorID	FirstName	LastName
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano

**Ил. 22.4.** Данные из таблицы Authors базы данных Books

### Таблица Titles базы данных Books

Таблица Titles (ил. 22.5) состоит из четырех столбцов с информацией о каждой книге в базе данных: в них хранится номер ISBN, название, номер издания и год регистрации авторских прав. На ил. 22.6 представлены данные из таблицы Titles.

Столбец	Описание
ISBN	Номер ISBN (строка); первичный ключ таблицы. ISBN — схема нумерации, при которой каждой книге присваивается уникальный идентификационный номер
Title	Название книги (строка)
EditionNumber	Номер издания (целое число)
Copyright	Год регистрации авторских прав (строка)

**Ил. 22.5.** Таблица Titles базы данных Books

ISBN	Title	EditionNumber	Copyright
0132151006	Internet & World Wide Web How to Program	5	2012
0132575663	Java How to Program	9	2012
013299044X	C How to Program	7	2013
0132990601	Simply Visual Basic 2010	4	2013
0133406954	Visual Basic 2012 How to Program	6	2014
0133379337	Visual C# 2012 How to Program	5	2014
0136151574	Visual C++ 2008 How to Program	2	2008
0133378713	C++ How to Program	9	2014
0132121360	Android for Programmers: An App-Driven Approach	1	2012

**Ил. 22.6.** Данные из таблицы Titles базы данных Books**Таблица AuthorISBN базы данных Books**

Таблица AuthorISBN (ил. 22.7) состоит из двух столбцов, в которых хранится номер ISBN и идентификатор автора. Эта таблица связывает информацию об авторах с книгами. Столбец AuthorID является *внешним ключом*, то есть значения столбца этой таблицы совпадают со столбцом первичного ключа в другой таблице (AuthorID в таблице Authors). Столбец ISBN тоже является внешним ключом — он соответствует столбцу первичного ключа (ISBN) из таблицы Titles. База данных может состоять из нескольких таблиц. Проектировщик базы данных стремится сократить количество данных, дублируемых между таблицами. Внешние ключи, задаваемые в момент создания таблицы в базе данных, связывают данные нескольких таблиц. Комбинация столбцов AuthorID и ISBN этой таблицы образует составной первичный ключ. Таким образом, каждая строка таблицы однозначно связывает одного автора с номером ISBN одной книги.

Столбец	Описание
AuthorID	Идентификационный номер автора, внешний ключ для таблицы Authors
ISBN	Номер ISBN книги, внешний ключ для таблицы Titles

**Ил. 22.7.** Таблица AuthorISBN базы данных Books

На ил. 22.8 приведены данные из таблицы AuthorISBN базы данных Books.

AuthorID	ISBN
1	0132151006
1	0132575663
1	013299044X

**Ил. 22.8.** Данные из таблицы AuthorISBN базы данных Books (продолжение ↗)

AuthorID	ISBN
1	0132990601
1	0133406954
1	0133379337
1	0136151574
1	0133378713
1	0132121360
2	0132151006
2	0132575663
2	013299044X
2	0132990601
2	0133406954
2	0133379337
2	0136151574
2	0133378713
2	0132121360
3	0132151006
3	0132990601
3	0132121360
3	0133406954
4	0136151574
5	0132121360

**Ил. 22.8.** Данные из таблицы AuthorISBN базы данных Books (окончание)

Каждое значение внешнего ключа должно присутствовать как значение первичного ключа другой таблицы, чтобы СУБД могла проверить действительность его значения. Например, чтобы убедиться в том, что значение AuthorID в конкретной строке таблицы AuthorISBN действительно (см. ил. 22.8), СУБД проверяет, что в таблице Authors имеется строка, у которой это значение AuthorID является первичным ключом. Внешние ключи также позволяют выполнять выборку взаимосвязанных данных из нескольких таблиц — это называется *объединением* данных. Между первичным ключом и соответствующим внешним ключом существуют отношения «один ко многим» (например, один автор может написать несколько книг и одна книга может быть написана несколькими авторами). Это означает, что внешний ключ может многократно встречаться в своей таблице, но только один раз (как первичный ключ) в другой таблице. Например, номер ISBN 0132151006 может встречаться в нескольких строках таблицы AuthorISBN (потому что у этой книги несколько авторов), но только один раз в таблице Titles, где ISBN является первичным ключом.

### Диаграмма сущностей и отношений для базы данных Books

На ил. 22.9 изображена диаграмма сущностей и отношений для базы данных Books. На этой диаграмме изображены таблицы базы данных и отношения между ними. В первом отделении каждого блока находится имя таблицы. Курсивом записаны

первичные ключи — `AuthorID` в таблице `Authors`, `AuthorID` и `ISBN` в таблице `AuthorISBN`, `ISBN` в таблице `Titles`. В каждой строке столбец (или группа столбцов) первичного ключа должен содержать значение, причем это значение должно быть уникальным в таблице; в противном случае СУБД сообщает об ошибке. В таблице `AuthorISBN` оба имени `AuthorID` и `ISBN` записаны курсивом — их сочетание образует *составной первичный ключ* таблицы `AuthorISBN`.



**Ил. 22.9.** Диаграмма сущностей и отношений базы данных Books

Линии, соединяющие таблицы на ил. 22.9, представляют отношения между таблицами. Рассмотрим линию между таблицами `Authors` и `AuthorISBN`. На стороне `Authors` линия помечена знаком 1, а на стороне `AuthorISBN` — знаком бесконечности ( $\infty$ ). Линия обозначает связь «один ко многим» — для каждого автора из таблицы `Authors` в таблице `AuthorISBN` может храниться произвольное количество номеров ISBN книг, написанных этим автором (то есть автор может написать любое количество книг). Обратите внимание: линия отношения соединяет столбец `AuthorID` таблицы `Authors` (где `AuthorID` является первичным ключом) со столбцом `AuthorID` таблицы `AuthorISBN` (где `AuthorID` является внешним ключом) — линия между таблицами связывает первичный ключ с соответствующим внешним ключом.

Линия между таблицами `Titles` и `AuthorISBN` демонстрирует связь типа «один ко многим» — одна книга может быть написана несколькими авторами. Обратите внимание: линия между таблицами соединяет первичный ключ `ISBN` в таблице `Titles` с соответствующим внешним ключом таблицы `AuthorISBN`. Отношения на ил. 22.9 показывают, что единственной целью таблицы `AuthorISBN` является создание отношения «многие ко многим» между таблицами `Authors` и `Titles` — автор может написать много книг, и у книги может быть много авторов.

## 22.4. LINQ to Entities и ADO.NET Entity Framework

При использовании ADO.NET Entity Framework вы взаимодействуете с базами данных через классы, которые IDE генерирует для схемы базы данных. Процесс запускается включением в проект новой модели данных сущностей ADO.NET (см. подраздел 22.5.1).

### Классы, генерируемые в модели данных сущностей

Для таблиц `Authors` и `Titles` в базе данных Books IDE создает в модели данных два класса:

- ❑ Первый класс представляет строку таблицы и содержит свойства для всех столбцов таблицы. Объекты этого класса — называемые *объектами строк* — содержат данные из отдельных строк таблицы. IDE использует в качестве имени класса строк версию имени таблицы в единственном числе (для таблицы `Authors` базы данных `Books` класс строки будет называться `Author`).
- ❑ Второй класс представляет саму таблицу. В объекте этого класса хранится коллекция объектов строк, соответствующих всем строкам в таблице. Классы таблицы базы данных `Books` называются `Authors` и `Titles`.

Сгенерированные классы модели данных сущностей пользуются полной поддержкой IntelliSense в IDE. В разделе 22.7 представлены запросы, которые используют отношения между таблицами базы данных `Books` для объединения данных.

### Отношения между таблицами в модели данных сущностей

Обратите внимание: говоря о сгенерированных классах, мы не упоминаем таблицу `AuthorISBN` базы данных `Books`. Вспомните, что эта таблица связывает:

- ❑ каждого автора в таблице `Authors` с книгами этого автора в таблице `Titles`
- ❑ и каждую книгу в таблице `Titles` с авторами книги в таблице `Authors`.

Отношения между таблицами учитываются в сгенерированных классах модели данных сущностей. Например, класс строки `Author` содержит *навигационное свойство* с именем `Titles`, при помощи которого можно получить объекты `Title` для всех книг, написанных автором. IDE автоматически добавляет «s» к имени «Title», показывая, что свойство представляет коллекцию объектов `Title`. Аналогичным образом класс строки `Title` содержит навигационное свойство `Authors`, при помощи которого можно получить объекты `Author`, представляющие авторов заданной книги.

### Класс `DbContext`

Класс `DbContext` (пространство имен `System.Data.Entity`) управляет передачей данных между программой и базой данных. Когда среда разработки генерирует классы строк и таблиц модели данных сущностей, она также создает класс, производный от `DbContext`, для конкретной базы данных, с которой выполняются операции. Для базы данных `Books` в этот производный класс будут включены свойства для таблиц `Authors` и `Titles`. Как вы вскоре увидите, они могут использоваться как источники данных для работы с информацией запросов LINQ и в графическом интерфейсе. Любые изменения в данных, находящихся под управлением `DbContext`, сохраняются в базе данных вызовом метода `SaveChanges` класса `DbContext`.

### Интерфейс `IQueryable<T>`

Технология LINQ to Entities основана на интерфейсе `IQueryable<T>`, производном от интерфейса `IEnumerable<T>`, представленного в главе 9. Когда запрос LINQ to Entities для объекта `IQueryable<T>` применяется к базе данных, результаты загружаются в объекты соответствующих классов модели данных сущностей для удобства работы с ними в коде.



### Использование методов расширения для работы с объектами `IQueryable<T>`

Вспомните, что методы расширения добавляют функциональность в существующий класс без изменения исходного кода класса. В главе 9 были представлены различные методы расширения LINQ, включая `First`, `Any`, `Count`, `Distinct`, `ToArray` и `ToList`. Эти методы, определенные как статические методы класса `Enumerable` (пространство имен `System.Linq`), могут применяться к любым объектам, реализующим интерфейс `IEnumerable<T>`, включая массивы, коллекции и результаты запросов LINQ to Objects.

В этой главе мы используем комбинацию синтаксиса запросов LINQ и методов расширения LINQ для работы с содержимым базы данных. Используемые методы расширения определяются как статические методы класса `Queryable` (пространство имен `System.Linq`) и могут применяться к любому объекту, реализующему интерфейс `IQueryable<T>`, — в том числе к различным объектам модели данных сущностей и результатам запросов LINQ to Entities.

## 22.5. Запрос к базе данных с использованием LINQ

Из этого раздела вы узнаете, как подключиться к базе данных, обратиться к ней с запросом и вывести результаты запроса. Программного кода потребуется совсем немного; IDE предоставляет необходимые средства визуального программирования и мастеров, упрощающих работу с данными в приложениях. Эти средства обеспечивают подключение к базе данных и создают объекты, необходимые для просмотра и обработки данных через элементы управления Windows Forms, — этот механизм называется *привязкой данных*.

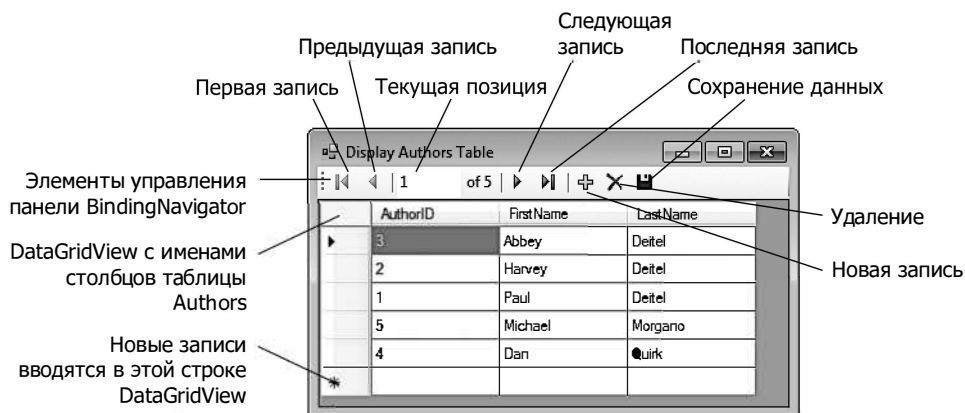
В разделах 22.5–22.8 мы создадим одно решение, состоящее из нескольких проектов. Одним из таких проектов станет библиотека классов, содержащая модель данных сущностей ADO.NET для взаимодействия с базой данных Books. В остальных проектах будут создаваться приложения Windows Forms, использующие модель данных сущностей ADO.NET для работы с базой данных.

Первый пример выполняет простой запрос к базе данных Books из раздела 22.3. Мы загружаем все записи таблицы `Authors`, упорядоченные по фамилии, а затем по имени автора. Затем привязка данных используется для вывода данных в `DataGridView` — элементе управления из пространства имен `System.Windows.Forms`, способном выводить информацию из источника данных в табличном формате. Основные выполняемые действия:

- ☐ Создание классов модели данных сущностей ADO.NET для работы с базой данных.
- ☐ Добавление объекта модели данных сущностей, представляющего таблицу `Authors`, как источника данных.

- ❑ Перетаскивание источника данных таблицы **Authors** в режиме конструктора для создания графического интерфейса, в котором выводятся данные таблицы.
- ❑ Добавление в файл программной логики **Form** кода, обеспечивающего взаимодействие приложения с базой данных.

Графический интерфейс программы изображен на ил. 22.10. Все элементы управления в этом интерфейсе генерируются автоматически при перетаскивании источника данных, представляющего таблицу **Authors**, на форму в режиме конструктора. Панель инструментов **BindingNavigator** в верхней части окна также позволяет добавлять и удалять записи, изменять существующие записи и сохранять изменения в базе данных. Чтобы добавить новую запись, нажмите кнопку **Add new** (+), затем введите имя и фамилию автора. Чтобы удалить существующую запись, выберите автора (либо в элементе управления **DataGridView**, либо при помощи элементов управления **BindingNavigator**) и нажмите кнопку **Delete** (X). Также можно отредактировать существующую запись; для этого щелкните в поле имени или фамилии и введите новое значение. Чтобы сохранить изменения в базе данных, просто щелкните на кнопке **Save Data** (дискета). Пустые значения в таблице **Authors** базы данных **Books** запрещены, поэтому при попытке сохранения записи, не содержащей значений в обоих полях (имени и фамилии), происходит исключение.



**Ил. 22.10.** Графический интерфейс приложения **Display Authors Table**

### 22.5.1. Создание библиотеки классов модели данных сущностей ADO.NET

В этом разделе описаны действия, необходимые для создания модели данных сущностей из существующей базы данных. Модель описывает данные, с которыми вы будете работать, — в нашем случае это данные, представленные таблицами базы данных **Books**.

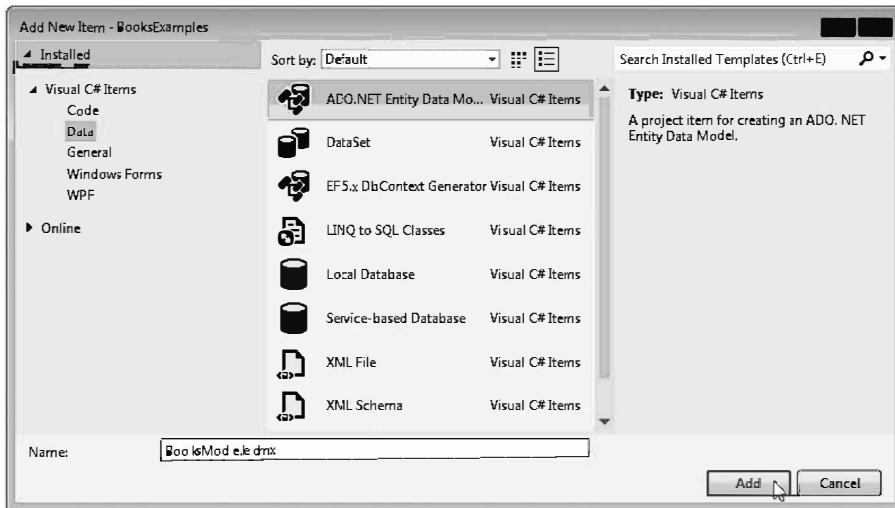
### Шаг 1. Создание проекта библиотеки классов для модели данных сущностей ADO.NET

Выберите команду **FILE ► New Project...**, чтобы вызвать диалоговое окно **New Project**. Затем выберите в списке шаблонов **Visual C#** пункт **Class Library** и введите имя проекта **BooksExamples**. Щелкните на кнопке **OK**, чтобы создать проект. Удалите файл **Class1.cs** из окна **Solution Explorer**.

### Шаг 2. Добавление модели данных сущностей ADO.NET в библиотеку классов

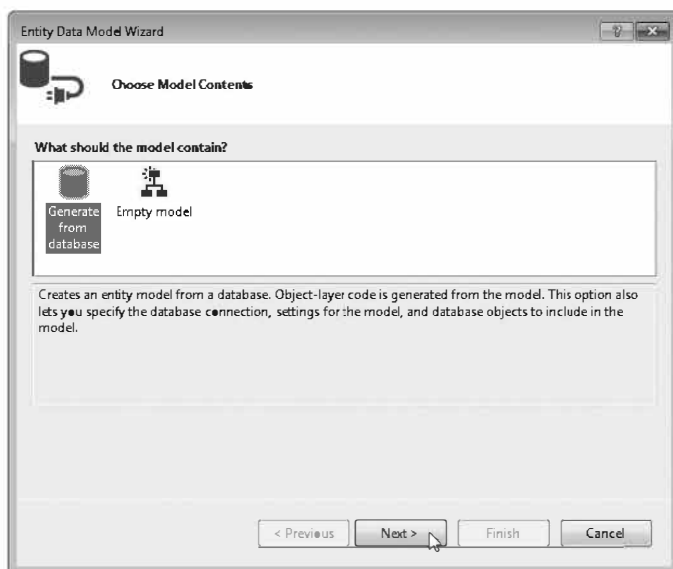
Для работы с базой данных необходимо включить модель данных сущностей ADO.NET в проект библиотеки классов. При этом также производится настройка подключения к базе данных.

1. **Добавление модели данных сущностей ADO.NET.** Щелкните правой кнопкой мыши на проекте **BooksExamples** в окне **Solution Explorer** и выберите команду **Add ► New Item...**, чтобы открыть диалоговое окно **Add New Item** (ил. 22.11). В категории **Data** выберите пункт **ADO.NET Entity Data Model** и введите имя модели **BooksModel.edmx** (в этом файле будет храниться информация о создаваемой модели данных). Щелкните на кнопке **Add**, чтобы добавить модель данных в библиотеку классов и вызвать диалоговое окно **Entity Data Model Wizard**.



**Ил. 22.11.** Выбор модели данных сущностей ADO.NET в диалоговом окне **Add New Item**

2. **Выбор содержимого модели.** На шаге **Choose Model Contents** мастера **Entity Data Model Wizard** (ил. 22.12) выбирается содержимое модели данных сущностей. В наших примерах модель состоит из данных из базы данных **Books**; выберите вариант **Generate from Database** и щелкните на кнопке **Next>**, чтобы перейти к шагу **Choose Your Data Connection**.



**Ил. 22.12.** Выбор содержимого модели в окне мастера Entity Data Model Wizard



**Ил. 22.13.** Диалоговое окно свойств подключения

3. **Выбор подключения.** На шаге Choose Your Data Connection щелкните на кнопке New Connection..., чтобы вызвать диалоговое окно Connection Properties (ил. 22.13). (Если на экране появляется диалоговое окно Choose Data Source, выберите Microsoft SQL Server и щелкните на кнопке OK.) Текстовое поле Data source: должно содержать текст Microsoft SQL Server Database File (SqlClient). (Если это не так, щелкните на кнопке

Change..., чтобы вызвать диалоговое окно для изменения источника данных.) Щелкните на кнопке Browse..., найдите и выберите файл Books.mdf из каталога Databases в примерах этой главы. Кнопка Test Connection позволяет проверить подключение к базе данных из IDE через SQL Server Express. Щелкните на кнопке OK, чтобы создать подключение. На ил. 22.14 показана строка подключения для базы данных Books.mdf. Строка содержит информацию, необходимую ADO.NET Entity Framework для подключения к базе данных во время выполнения. Щелкните на кнопке Next>. На экране появляется диалоговое окно с предложением добавить файл базы данных в проект. Щелкните на кнопке Yes, чтобы перейти к следующему шагу.



Ил. 22.14. Окно Choose Your Data Connection после выбора файла Books.mdf

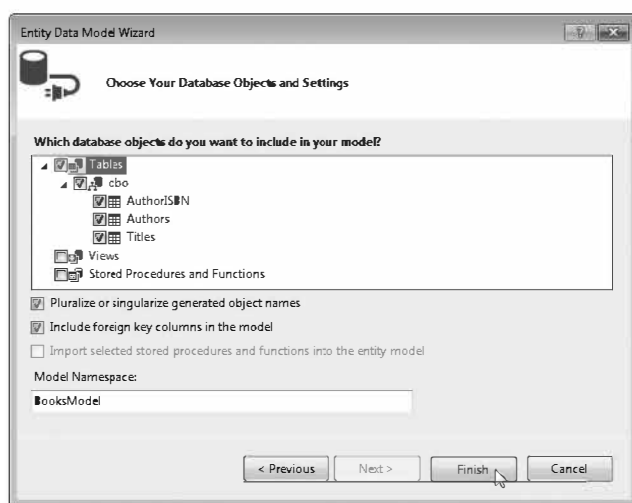


#### КАК ИЗБЕЖАТЬ ОШИБОК 22.1

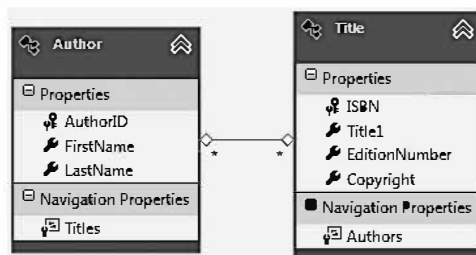
SQL Server Express LocalDB позволяет только одной программе подключаться к файлу базы данных в любой момент времени. Проследите за тем, чтобы перед подключением к базе данных файл не использовался другой программой.

4. **Выбор объектов базы данных для включения в модель.** На этом шаге указываются части базы данных, которые должны использоваться в модели данных сущностей ADO.NET. Выберите узел Tables, как показано на ил. 22.15, и щелкните на кнопке Finish.
5. **Просмотр диаграммы модели данных сущностей в конструкторе модели.** На этой стадии IDE создает модель данных сущностей и отображает диаграмму (ил. 22.16) в конструкторе модели. На диаграмме изображены сущности Author

и Title — они представляют авторов и названия книг в базе данных, а также их свойства. Обратите внимание: столбец Title таблицы Titles переименовывается в Title1, чтобы избежать конфликта имен с классом Title, представляющим строку таблицы. Линия между сущностями обозначает отношения между авторами и названиями книг; это отношение реализовано в базе данных Books таблицей AuthorISBN. Звездочки (\*) на обоих концах линий обозначают отношение «многие ко многим» — автор может написать много книг, а у книги может быть много авторов. Раздел Navigation Properties сущности Author содержит свойство Titles, которое связывает автора со всеми названиями книг, написанных этим автором. Аналогичным образом раздел Navigation Properties сущности Title содержит свойство Authors, связывающее название книги со всеми ее авторами.



**Ил. 22.15.** Выбор таблиц базы данных для включения в модель данных сущностей ADO.NET



**Ил. 22.16.** Диаграмма модели данных для сущностей Author и Title

6. **Построение библиотеки классов.** Выберите команду BUILD ► Build Solution, чтобы построить библиотеку классов для нескольких следующих примеров. Команда компилирует всю модель данных сущностей, сгенерированных средой

разработки. При построении библиотеки классов средой разработки. При построении библиотеки классов IDE генерирует классы, используемые для выполнения операций с базой данных. В этот набор входят класс для каждой таблицы, из которой осуществляется выборка, и производный от DbContext класс с именем BooksEntities, предназначенный для взаимодействия с базой данных на программном уровне. [Примечание: при построении проекта в IDE выполняется сценарий, который создает и компилирует все классы модели данных сущностей. На экране появляется диалоговое окно с предупреждением о том, что сценарий может причинить вред вашему компьютеру. Щелкните на кнопке ОК, чтобы разрешить выполнение сценария. Предупреждение актуально в основном для шаблонов Visual Studio, загруженных из Интернета)].

## 22.5.2. Создание проекта Windows Forms и его настройка для использования модели данных сущностей

Еще раз напомним, что несколько следующих примеров будут частью одного решения, состоящего из нескольких проектов — библиотеки классов с моделью для повторного использования и отдельных приложений Windows Forms для каждого примера. В этом разделе мы создадим новое приложение Windows Forms и настроим его для использования модели данных сущностей, созданной в предыдущем разделе.

### Шаг 1. Создание проекта

Чтобы добавить новый проект Windows Forms в существующее решение, выполните следующие действия:

1. Щелкните правой кнопкой мыши на имени решения в окне Solution Explorer и выберите команду Add ► New Project.... На экране появляется диалоговое окно Add New Project.
2. Выберите пункт Windows Forms Application, введите имя проекта DisplayTable и щелкните на кнопке ОК.
3. Измените имя исходного файла Form1.cs на DisplayAuthorsTable.cs. IDE обновляет имя класса формы в соответствии с именем исходного файла. Задайте свойству Text объекта Form значение DisplayAuthorsTable.
4. Настройте решение таким образом, чтобы новый проект выполнялся при выборе команды DEBUG ► Start Debugging (или нажатии клавиши F5). Для этого щелкните правой кнопкой мыши на имени проекта DisplayTable в окне Solution Explorer и выберите команду Set as Startup Project.

### Шаг 2. Добавление ссылки на библиотеку классов BooksExamples

Чтобы использовать классы модели данных сущностей для привязки данных, необходимо сначала добавить ссылку на библиотеку классов, созданную

в разделе 22.5.1, — это позволит новому проекту использовать эту библиотеку классов. В каждый созданный вами проект обычно по умолчанию включаются ссылки на несколько библиотек классов .NET (называемых *сборками*) — например, проект Windows Forms содержит ссылку на библиотеку `System.Windows.Forms`. В результате компиляции библиотеки классов IDE создает файл с расширением `.dll`, содержащий компоненты библиотеки. Чтобы добавить ссылку на библиотеку классов с классами модели данных сущностей, выполните следующие действия:

1. Щелкните правой кнопкой мыши на узле `References` проекта `DisplayTable` в окне `Solution Explorer` и выберите команду `Add Reference...`
2. В левом столбце открывшегося диалогового окна `Reference Manager` выберите пункт `Solution`, чтобы открыть другие проекты решения. В центре диалогового окна выберите проект `BooksExamples` и щелкните на кнопке `OK`. Библиотека `BooksExamples` появляется в узле `References` проектов.

### Шаг 3. Добавление ссылок на `System.Data.Entity` и `EntityFramework`

Для использования ADO.NET Entity Framework нам также понадобятся ссылки на библиотеки `System.Data.Entity` и `EntityFramework`. Чтобы добавить ссылку на `System.Data.Entity`, повторите шаг 2 для добавления ссылки на библиотеку `BooksExamples`, но в левом столбце диалогового окна `Reference Manager` выберите категорию `Assemblies`, найдите библиотеку `System.Data.Entity`, убедитесь в том, что флажок рядом с ней установлен, и щелкните на кнопке `OK`. Библиотека `System.Data.Entity` должна появиться в узле `References` проектов.

Библиотека `EntityFramework` была добавлена в проект библиотеки классов `BooksExamples` при создании модели данных сущностей, но библиотека `EntityFramework` также необходима в каждом приложении, использующем модель данных сущностей. Чтобы добавить ссылку на библиотеку `EntityFramework`, выполните следующие действия:

1. Щелкните правой кнопкой мыши на имени решения в окне `Solution Explorer` и выберите команду `Manage NuGet Packages for Solution...`. На экране появляется диалоговое окно `Manage NuGet Packages`.
2. В открывшемся диалоговом окне щелкните на кнопке `Manage`, чтобы вызвать диалоговое окно `Select Projects`, выберите проект `DisplayTable` и щелкните на кнопке `OK`.
3. Щелкните на кнопке `Close`, чтобы закрыть диалоговое окно `Manage NuGet Packages`. Библиотека `System.Data.Entity` должна появиться в узле `References` проектов.

### Шаг 4. Добавление строки подключения в приложение Windows Forms

Каждому приложению, использующему модель данных сущностей, также необходима строка подключения, которая сообщает среде Entity Framework, как следует подключаться к базе данных. Строка подключения хранится в файле `App.Config` библиотеки классов `BooksExamples`. В окне `Solution Explorer` откройте файл `App.Config` библиотеки классов `BooksExamples` и скопируйте строки 7–9, которые имеют следующий формат:

```
<connectionStrings>  
Здесь находится информация строк подключения  
</connectionStrings>
```



Откройте файл App.Config в проекте DisplayTable и вставьте скопированный фрагмент между строками `</entityFramework>` и `</configuration>`. Сохраните файл App.Config.

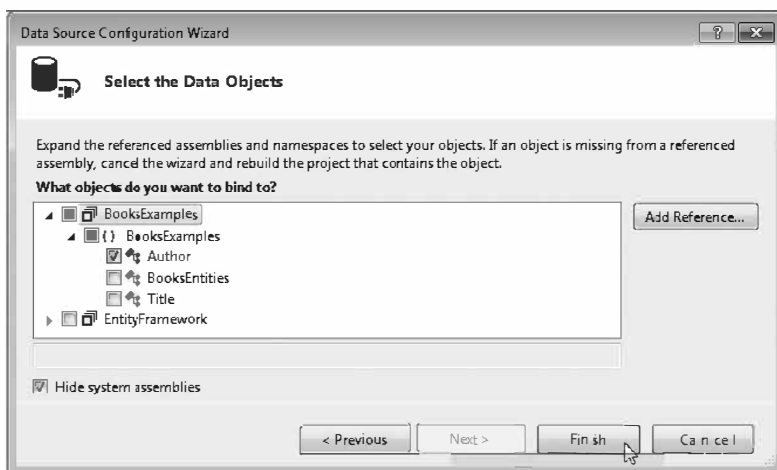
### 22.5.3. Привязка данных между элементами управления и моделью данных сущностей

Теперь мы воспользуемся средствами IDE для построения графического интерфейса, взаимодействующего с базой данных Books. Нам придется написать небольшой фрагмент кода, чтобы автоматически сгенерированный интерфейс мог взаимодействовать с моделью данных сущностей. Чтобы содержимое таблицы Authors выводилось в графическом интерфейсе, необходимо выполнить следующие действия.

#### Шаг 1. Добавление источника данных для таблицы Authors

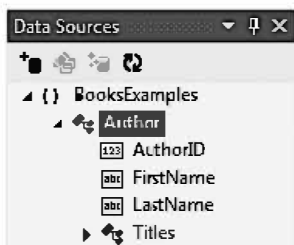
Чтобы использовать классы модели данных сущностей для привязки данных, следует добавить их в качестве *источника данных*. Это делается так:

1. Выполните команду **VIEW ► Other Windows ► Data Sources**, чтобы вывести окно Data Sources в левой части IDE. В этом окне щелкните на ссылке **Add New Data Source...**, чтобы вызвать мастера настройки источников данных.
2. Классы модели данных используются для создания объектов, представляющих таблицы в базе данных, поэтому мы используем источник данных Object. В диалоговом окне выберите вариант Object и щелкните на кнопке **Next>**. Раскройте дерево, как показано на ил. 22.17, и убедитесь в том, что флажок рядом со строкой **Author** установлен. Объект этого класса будет использоваться в качестве источника данных.
3. Щелкните на кнопке **Finish**.



Ил. 22.17. Выбор класса Author в качестве источника данных

Таблица `Authors` в базе данных теперь является источником данных, из которого элементы управления с привязкой к данным могут получать информацию. В окне `Data Sources` (ил. 22.18) отображается класс `Author`, добавленный на предыдущем шаге. Ниже должны располагаться свойства, представляющие столбцы таблицы `Authors` базы данных, и навигационное свойство `Titles`, представляющее отношения между таблицами `Authors` и `Titles` базы данных.



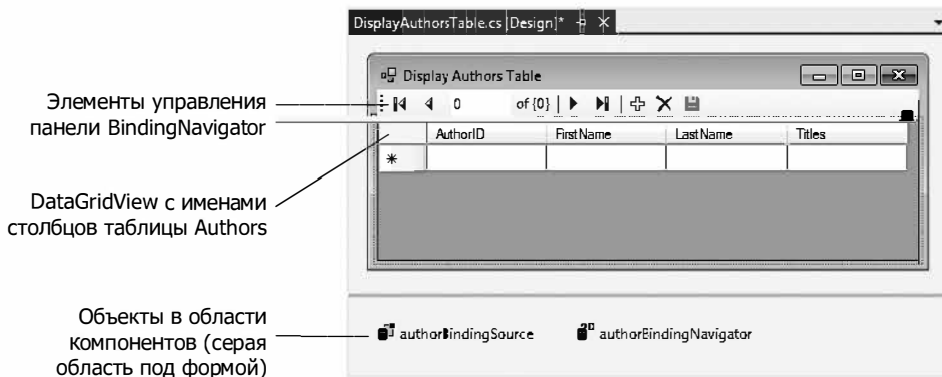
Ил. 22.18. Окно `Data Sources` с классом `Author` как источником данных

## Шаг 2. Создание элементов графического интерфейса

Затем мы воспользуемся режимом конструктора для создания элемента управления `DataGridView`, в котором выводятся данные таблицы `Authors`. Для этого выполните следующие действия:

1. Переключитесь в режим конструктора для класса `DisplayAuthorsTable`.
2. Щелкните на узле `Author` в окне `Data Sources`; он должен заменить раскрывающимся списком. Откройте список при помощи кнопки со стрелкой и убедитесь в том, что в нем выбран вариант `DataGridView` (по умолчанию), — этот элемент управления будет использоваться для вывода данных и взаимодействия с ними.
3. Перетащите узел `Author` из окна `Data Sources` на форму в режиме конструктора. Для размещения элемента управления `DataGridView` размеры формы придется изменить.

IDE создает элемент управления `DataGridView` (ил. 22.19) с именами столбцов, представляющими все свойства `Author`, включая навигационное свойство `Titles`. IDE также создает панель `BindingNavigator` с кнопками для перемещения между записями, добавления и удаления записей, а также сохранения изменений в базе данных. IDE также генерирует объект `BindingSource` (`authorBindingSource`), который обеспечивает передачу данных между источником данных и элементами управления с привязкой к данным на форме. Невизуальные компоненты — такие, как `BindingSource` и невидимые аспекты `BindingNavigator`, — отображаются в области компонентов — серой области под формой в режиме конструктора. IDE присваивает имена `BindingNavigator` и `BindingSource` (`authorBindingNavigator` и `authorBindingSource` соответственно) на основании имени источника данных (`Author`). В этой главе мы оставляем автоматически сгенерированным компонентам имена по умолчанию, чтобы показать, что именно делает IDE.



**Ил. 22.19.** В области компонентов в режиме конструктора размещаются невидимые компоненты

Чтобы элемент управления DataGridView занимал все окно под панелью BindingNavigator, выделите DataGridView и в окне свойств задайте свойству Dock значение Fill.

Вы можете растянуть окно по горизонтали, чтобы увидеть все столбцы DataGridView. Так как в нашем примере столбец Titles не используется, выделите DataGridView и выберите в окне свойств команду Edit Columns.... На экране появляется диалоговое окно Edit Columns. Выберите в списке Selected Columns столбец Titles и щелкните на кнопке Remove, чтобы исключить его из вывода.

### Шаг 3. Связывание источника данных с authorBindingSource

На последнем шаге источник данных связывается с authorBindingSource, чтобы приложение могло взаимодействовать с базой данных. На ил. 22.20 приведен код, необходимый для получения данных из базы данных и сохранения внесенных изменений в базе.

```

1 // Ил. 22.20: DisplayAuthorsTable.cs
2 // Отображение данных из таблицы базы данных в DataGridView.
3 using System;
4 using System.Data.Entity;
5 using System.Data.Entity.Validation;
6 using System.Linq;
7 using System.Windows.Forms;
8
9 namespace DisplayTable
10 {
11     public partial class DisplayAuthorsTable : Form
12     {
13         // Конструктор
14         public DisplayAuthorsTable()
15         {
16             InitializeComponent();
17         } // Конец конструктора

```

**Ил. 22.20.** Отображение данных из таблицы базы данных в DataGridView (продолжение ↗)

```

18
19 // Entity Framework DbContext
20 private BooksExamples.BooksEntities dbcontext =
21     new BooksExamples.BooksEntities();
22
23 // Загрузка информации из базы данных в DataGridView
24 private void DisplayAuthorsTable_Load( object sender, EventArgs e )
25 {
26     // Таблица Authors с упорядочением по LastName и FirstName
27     dbcontext.Authors
28         .OrderBy( author => author.LastName )
29         .ThenBy( author => author.FirstName )
30         .Load();
31
32     // Назначение источника данных для authorBindingSource
33     authorBindingSource.DataSource = dbcontext.Authors.Local;
34 } // Конец метода DisplayAuthorsTable_Load
35
36 // Обработчик события Click для кнопки Save панели BindingNavigator
37 // сохраняет изменения в данных
38 private void authorBindingNavigatorSaveItem_Click(
39     object sender, EventArgs e )
40 {
41     Validate(); // Проверка полей ввода
42     authorBindingSource.EndEdit(); // Завершение текущей правки
43
44     // Попытка сохранения изменений
45     try
46     {
47         dbcontext.SaveChanges(); // Запись изменений в базу данных
48     } // Конец try
49     catch( DbEntityValidationException )
50     {
51         MessageBox.Show( "FirstName and LastName must contain values",
52             "Entity Validation Exception" );
53     } // Конец catch
54 } // Конец метода authorBindingNavigatorSaveItem_Click
55 } // Конец класса DisplayAuthorsTable
56 } // Конец пространства имен DisplayTable

```

	AuthorID	FirstName	LastName
▶	3	Abbey	Deitel
	2	Harvey	Deitel
	1	Paul	Deitel
	5	Michael	Morgano
	4	Dan	Quirk
*			

**Ил. 22.20.** Отображение данных из таблицы базы данных в DataGridView (окончание)

### Создание объекта DbContext

Как упоминалось в разделе 22.4, объект `DbContext` взаимодействует с базой данных от имени приложения. Класс `BooksEntities` (производный от `DbContext`) автоматически генерируется IDE при создании классов модели данных сущностей для обращения к базе данных `Books` (см. подраздел 22.5.1). В строках 20–21 создается объект класса `dbcontext`.

### Обработчик события `DisplayAuthorsTable_Load`

Обработчик события `Load` формы (строки 24–34) создается двойным щелчком на заголовке формы в режиме конструктора. В этом приложении для передачи данных между объектом `DbContext` и базой данных используются методы расширения `LINQ to Entities`, извлекающие данные из свойства `Authors` объекта `BooksEntities` (строки 27–30), соответствующего таблице `Authors` базы данных. Выражение

```
dbcontext.Authors
```

указывает, что мы получаем данные из таблицы `Authors`.

Вызов метода расширения `OrderBy`

```
.OrderBy( author => author.LastName )
```

указывает, что строки таблицы должны выбираться в порядке возрастания фамилий авторов. Аргументом `OrderBy` является лямбда-выражение, определяющее простой анонимный метод. Лямбда-выражение начинается со списка параметров — в данном случае `author`, объект класса модели данных сущностей `Author`. Лямбда-выражение вычисляет тип лямбда-параметра по типу `dbcontext.Authors`, содержащему объекты `Author`. За списком параметров следует оператор `=>` и выражение, представляющее тело функции. Значение, полученное в результате обработки выражения, — фамилия автора — неявно возвращается лямбда-выражением. Возвращаемый тип для лямбда-выражения не указывается; он определяется по возвращаемому значению. Синтаксис лямбда-выражений более подробно рассматривается по мере их упоминания в этой главе. Дополнительную информацию о лямбда-выражениях можно найти по адресу

[msdn.microsoft.com/en-us/library/bb397687.aspx](http://msdn.microsoft.com/en-us/library/bb397687.aspx)

Если в таблице содержатся данные нескольких авторов с одинаковой фамилией, они должны быть отсортированы по возрастанию в алфавитном порядке имен. Вызов метода расширения `ThenBy`

```
.ThenBy( author => author.FirstName )
```

позволяет упорядочить результаты по дополнительному столбцу. Сортировка применяется к объектам `Author`, уже упорядоченным по фамилии.

Наконец, в строке 30 вызывается метод расширения `Load`, определяемый в классе `DBExtensions` из пространства имен `System.Data.Entity`. Этот метод выполняет запрос `LINQ to Entities` и загружает результаты в память. Объект `BooksEntities` отслеживает эти данные в локальной памяти, так что любые изменения

в них в конечном итоге будут сохранены в базе данных. Строки 27–30 эквиваленты следующей команде:

```
(from author in dbcontext.Authors
 orderby author.LastName, author.FirstName
 select author).Load();
```

Строка 33 задает свойству `DataSource` объекта `authorBindingSource` значение свойства `Local` объекта `dbcontext.Authors`. В данном случае `Local` содержит объект `ObservableCollection<Author>`, представляющий результаты запроса, загруженные в память в строках 27–30. Когда свойству `DataSource` объекта `BindingSource` присваивается `ObservableCollection<T>` (пространство имен `System.Collections.ObjectModel`), графический интерфейс, связанный с `BindingSource`, оповещается об изменении в данных для соответствующего обновления графического интерфейса. Кроме того, изменения, внесенные пользователем в данные, будут отслеживаться, чтобы объект `DbContext` смог в дальнейшем сохранить эти изменения в базе данных.

### **Обработчик события `authorBindingNavigatorSaveItem_Click`: сохранение модификаций в базе данных**

Изменения, вносимые пользователем в `DataGridView`, должны сохраняться в базе данных. По умолчанию кнопка **Save Data** (📁) панели `BindingNavigator` заблокирована. Чтобы сделать ее доступной, щелкните правой кнопкой мыши на значке кнопки на панели `BindingNavigator` и выберите команду `Enabled`. Затем сделайте двойной щелчок на значке, чтобы создать обработчик события `Click` (строки 38–54).

Сохранение данных, введенных в `DataGridView`, выполняется в три этапа. Сначала проверяются все элементы управления на форме (строка 41) вызовом унаследованного метода `Validate` объекта `DisplayTableForm` — если любой элемент управления содержит обработчик события `Validating`, то этот обработчик будет выполнен. Обычно это событие используется для проверки действительности содержимого элемента управления. Затем в строке 42 вызывается метод `EndEdit` объекта `authorBindingSource`, который заставляет его закрепить все несохраненные изменения в модели `BooksEntities` в памяти. Наконец, в строке 47 вызывается метод `SaveChanges` для объекта `BooksEntities (dbcontext)` для сохранения изменений в базе данных. Этот вызов заключен в команду `try`, потому что таблица `Authors` не допускает пустые значения в полях имени и фамилии — эти правила были заданы при исходном создании базы данных. При вызове `SaveChanges` все изменения в таблице `Authors` должны удовлетворять правилам таблицы. Если какие-либо изменения нарушают эти правила, выдается исключение `DbEntityValidationException`.

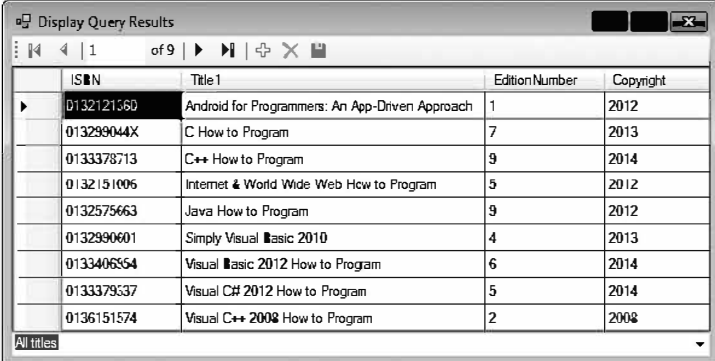
## **22.6. Динамическая привязка результатов запроса**

Теперь, когда вы знаете, как вывести все содержимое таблицы базы данных в `DataGridView`, мы покажем, как выполнить несколько разных запросов и вывести

результаты в `DataGridView`. Это приложение ограничивается чтением данных из модели данных сущностей, поэтому мы заблокировали кнопки `BindingNavigator` для добавления и удаления записей. Позже мы объясним, почему в этом примере изменение базы данных не поддерживается.

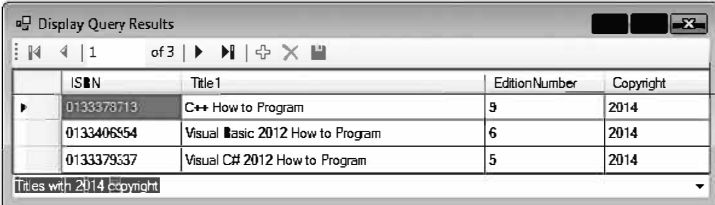
Приложение `DisplayQueryResults` (ил. 22.21) позволяет выбрать запрос из поля со списком (`ComboBox`) в нижней части окна, после чего выводит результаты запроса.

а) В результатах запроса «Все названия» выводится содержимое таблицы `Titles`, упорядоченное по названиям книг



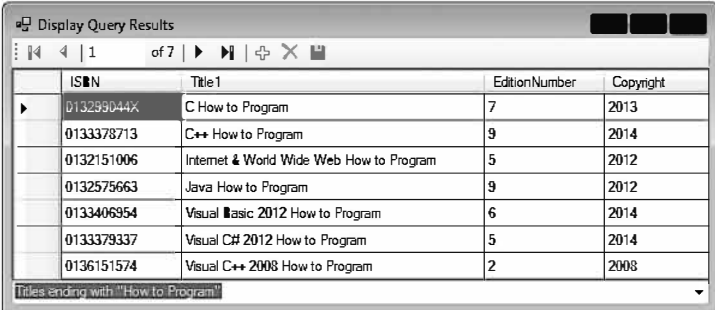
ISBN	Title1	EditionNumber	Copyright
0132121360	Android for Programmers: An App-Driven Approach	1	2012
013299044X	C How to Program	7	2013
0133378713	C++ How to Program	9	2014
0132151006	Internet & World Wide Web How to Program	5	2012
0132575663	Java How to Program	9	2012
0132990601	Simply Visual Basic 2010	4	2013
0133406554	Visual Basic 2012 How to Program	6	2014
0133379337	Visual C# 2012 How to Program	5	2014
0136151574	Visual C++ 2008 How to Program	2	2008

б) Результаты запроса «Названия с 2014 годом регистрации авторских прав»



ISBN	Title1	EditionNumber	Copyright
0133378713	C++ How to Program	9	2014
0133406554	Visual Basic 2012 How to Program	6	2014
0133379337	Visual C# 2012 How to Program	5	2014

в) Результаты запроса «Названия, завершающиеся текстом „How to Program“»



ISBN	Title1	EditionNumber	Copyright
013299044X	C How to Program	7	2013
0133378713	C++ How to Program	9	2014
0132151006	Internet & World Wide Web How to Program	5	2012
0132575663	Java How to Program	9	2012
0133406554	Visual Basic 2012 How to Program	6	2014
0133379337	Visual C# 2012 How to Program	5	2014
0136151574	Visual C++ 2008 How to Program	2	2008

**Ил. 22.21.** Выполнение приложения `DisplayQueryresults`

## 22.6.1. Создание графического интерфейса

Чтобы построить графический интерфейс приложения `DisplayQueryResults`, выполните следующие действия.


### Шаг 1. Создание проекта

Выполните действия из раздела 22.5.2, чтобы создать новый проект приложения Windows Forms с именем `DisplayQueryResult` в одном решении с приложением `DisplayTable`. Переименуйте файл `Form1.cs` в `TitleQueries.cs`. Задайте свойству `Text` объекта формы значение `Display Query Results`. Не забудьте назначить `DisplayQueryResult` стартовым проектом.

### Шаг 2. Создание элемента управления `DataGridView` для вывода таблицы `Titles`

Выполните шаги 1 и 2 из раздела 22.5.3 для создания источника данных и `DataGridView`. На этот раз выберите в качестве источника данных класс `Title` (вместо `Author`) и перетащите узел `Title` из окна `Data Sources` на форму. Удалите столбец `Authors` из `DataGridView`, так как он не будет использоваться в этом примере.

### Шаг 3. Добавление элемента управления `ComboBox` на форму

В режиме конструктора добавьте на форму элемент управления `ComboBox` с именем `queriesComboBox` под элементом управления `DataGridView`. В этом элементе управления пользователь выбирает запрос для выполнения. Задайте свойству `Dock` элемента управления `ComboBox` значение `Bottom`, а свойству `Dock` элемента управления `DataGridView` — значение `Fill`. Затем добавьте имена запросов в список: откройте окно `String Collection Editor` для элемента управления `ComboBox` (щелкните правой кнопкой мыши на `ComboBox` и выберите команду `Edit Items...`). Также можно вызвать `String Collection Editor` из меню смарт-тега `ComboBox`. Меню смарт-тега открывает быстрый доступ к свойствам, часто задаваемым для элементов управления (как, например, свойство `Multiline` элемента управления `TextBox`), поэтому вы можете задать эти свойства непосредственно в режиме конструктора вместо окна свойств. Меню смарт-тега элемента управления открывается щелчком на маленькой стрелке  в правом верхнем углу элемента управления в режиме конструктора при выделении элемента. В окне `String Collection Editor` добавьте в `queriesComboBox` три варианта — по одному для каждого из создаваемых нами запросов:

```
All titles
Titles with 2014 copyright
Titles ending with "How to Program"
```

## 22.6.2. Код приложения `DisplayQueryResults`

Переходим к созданию кода приложения (ил. 22.22).

### Настройка обработчика события `Load` формы

Создайте обработчик события `TitleQueries_Load` (строки 22–29) двойным щелчком на заголовке в режиме конструктора. При загрузке на форме должен загружаться полный список книг из таблицы `Titles`, отсортированный по названию. Строка 24 вызывает метод расширения `Load` для свойства `Titles` объекта `DbContext` модели `BookEntities` для загрузки содержимого таблицы `Titles` в память. Вместо определения



такого же запроса LINQ, как в строках 40–41, мы инициируем выполнение обработчика события `queriesComboBox_SelectedIndexChanged` на программном уровне, устанавливая свойство `SelectedIndex` объекта `queriesComboBox` равным 0 (строка 28).

```

1 // Ил. 22.22: TitleQueries.cs
2 // вывод результата запроса, выбранного пользователем, в DataGridView.
3 using System;
4 using System.Data.Entity;
5 using System.Linq;
6 using System.Windows.Forms;
7
8 namespace DisplayQueryResult
9 {
10     public partial class TitleQueries : Form
11     {
12         public TitleQueries()
13         {
14             InitializeComponent();
15         } // Конец конструктора
16
17         // Entity Framework DbContext
18         private BooksExamples.BooksEntities dbcontext =
19             new BooksExamples.BooksEntities();
20
21         // Загрузка информации из базы данных в DataGridView
22         private void TitleQueries_Load( object sender, EventArgs e )
23         {
24             dbcontext.Titles.Load(); // Загрузка таблицы Titles в памяти
25
26             // Настройка ComboBox для вывода запроса по умолчанию,
27             // выводящего все книги из таблицы Titles
28             queriesComboBox.SelectedIndex = 0;
29         } // Конец метода TitleQueries_Load
30
31         // Загрузка данных в titleBindingSource по выбранному запросу
32         private void queriesComboBox_SelectedIndexChanged(
33             object sender, EventArgs e )
34         {
35             // Назначение вывода в соответствии с выбранным вариантом
36             switch ( queriesComboBox.SelectedIndex )
37             {
38                 case 0: // Все названия
39                     // Использование LINQ для упорядочения книг по названию
40                     titleBindingSource.DataSource =
41                         dbcontext.Titles.Local.OrderBy( book => book.Title1 );
42                     break;
43                 case 1: // Названия с 2014 годом
44                     // Использование LINQ для получения книг с 2014 годом
45                     // и сортировки их по названию
46                     titleBindingSource.DataSource =
47                         dbcontext.Titles.Local
48                             .Where( book => book.Copyright == "2014" )
49                             .OrderBy( book => book.Title1 );
50                     break;

```

**Ил. 22.22.** Вывод результатов запроса, выбранного пользователем, в DataGridView (продолжение ↗)

```

51         case 2: // Названия, заканчивающиеся текстом "How to Program"
52             // Использование LINQ для получения книг, заканчивающихся
53             // текстом "How to Program", и сортировки их по названию
54             titleBindingSource.DataSource =
55                 dbcontext.Titles.Local
56                 .Where( book =>
57                     book.Title1.EndsWith( "How to Program" ) )
58                 .OrderBy( book => book.Title1 );
59         break;
60     } // Конец switch
61
62     titleBindingSource.MoveFirst(); // Переход к первой записи
63 } // Конец метода queriesComboBox_SelectedIndexChanged
64 } // Конец класса TitleQueries
65 } // Конец пространства имен DisplayQueryResult

```

**Ил. 22.22.** Вывод результатов запроса, выбранного пользователем, в DataGridView (окончание)

### Обработчик события queriesComboBox\_SelectedIndexChanged

Затем необходимо написать код выполнения соответствующего запроса каждый раз, когда пользователь выберет новый вариант из поля queriesComboBox. Сделайте двойной щелчок на queriesComboBox в режиме конструктора, чтобы сгенерировать обработчик события queriesComboBox\_SelectedIndexChanged (строки 32–63) в файле TitleQueries.cs. Добавьте в обработчик события команду switch (строки 36–60). Каждая секция case в switch заменяет свойство DataSource объекта titleBindingSource результатами запроса, возвращающего нужный набор данных. Привязки данных, создаваемые IDE, автоматически обновляют titleDataGridView при каждом изменении DataSource. Метод MoveFirst объекта BindingSource (строка 62) осуществляет перемещение к первой строке результата при каждом выполнении запроса. Результаты запросов в строках 40–41, 46–49 и 54–58 представлены на ил. 22.21, а, б и в соответственно. Так как в нашем приложении данные не изменяются, каждый запрос применяется к представлению таблицы Titles в памяти, доступному через dbcontext.Titles.Local.

### Упорядочение книг по названию

В строках 40–41 вызов метода расширения OrderBy для свойства dbcontext.Titles.Local упорядочивает объекты Title по значению свойства Title1. Как упоминалось ранее, IDE переименовывает столбец Title таблицы Titles базы данных, присваивая ему имя Title1 в сгенерированном классе модели данных сущностей Title для предотвращения конфликта имен. Вспомните, что Local возвращает объект ObservableCollection<T> с объектами строк заданной таблицы — в нашем случае Local возвращает ObservableCollection<Title>. При вызове OrderBy для ObservableCollection<T> метод возвращает IEnumerable<T>. Этот объект задается свойству DataSource объекта titleBindingSource. При изменении свойства DataSource элемент управления DataGridView перебирает содержимое IEnumerable<T> и выводит данные.

### Выбор книг с 2014 годом регистрации авторских прав

В строках 46–49 выводимые названия фильтруются методом расширения `Where` с передачей лямбда-выражения

```
book => book.Copyright == "2014"
```

в качестве аргумента. Лямбда-выражение получает параметр — один объект `Title` (с именем `book`) — и использует его для сравнения свойства `Copyright` объекта `Title` (строки базы данных) с 2014. Лямбда-выражение, используемое с методом расширения `Where`, должно возвращать значение `bool`. Выбираются только те объекты `Title`, для которых лямбда-выражение возвращает `true`. Вызов `OrderBy` упорядочивает результаты по свойству `Title1`, чтобы книги выводились в порядке возрастания названий. Тип параметра `book` лямбда-выражения определяется по свойству `dbcontext.Titles.Local`, содержащему объекты `Title`. Как только свойство `DataSource` объекта `titleBindingSource` изменяется, элемент управления `DataGridView` обновляется результатами запроса.

### Выбор книг с названиями, заканчивающимися текстом «How to Program»

В строках 54–58 выводимые названия фильтруются методом расширения `Where` с передачей лямбда-выражения

```
book => book.Title1.EndsWith( "How to Program" )
```

в качестве аргумента. Лямбда-выражение получает параметр — один объект `Title` (с именем `book`) — и использует его для проверки, завершается ли свойство `Title1` объекта `Title` строкой "How to Program". Выражение `books.Title1` возвращает строку, хранящуюся в свойстве, после чего метод `EndsWith` класса `string` проверяет условие. Результаты упорядочиваются по свойству `Title1`, чтобы книги выводились в порядке возрастания названий.

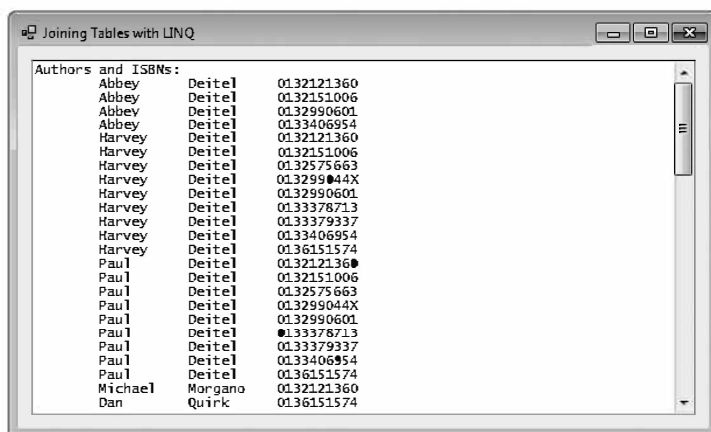
## 22.7. Выборка данных из нескольких таблиц с использованием LINQ

В этом разделе мы займемся выполнением запросов LINQ to Entities с использованием синтаксиса запросов LINQ, представленного в главе 9. В частности, вы узнаете, как получить результаты запросов, объединяющие данные из нескольких таблиц (ил. 22.23).

Приложение `JoinQueries` использует технологию LINQ to Entities для объединения и упорядочения данных нескольких таблиц, после чего выводит результаты следующих запросов:

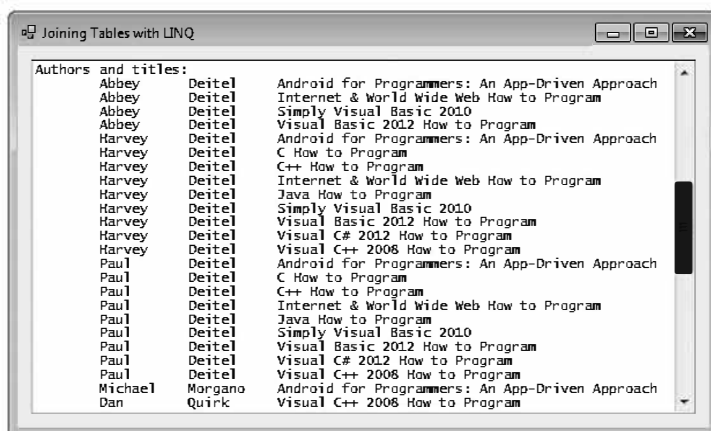
- ❑ Получение списка всех авторов и кодов ISBN написанных ими книг, отсортированного сначала по фамилиям, затем по именам (см. ил. 22.23, а).

а) Список авторов и кодов ISBN написанных ими книг; авторы сортируются сначала по фамилиям, затем по именам



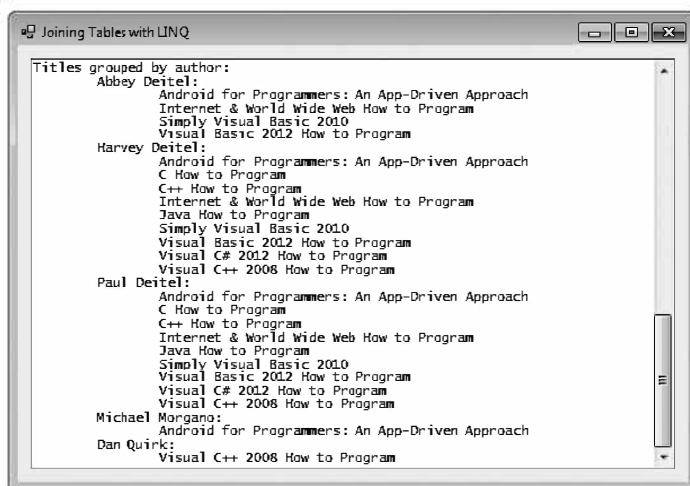
Authors and ISBNs:		
Abbey	Deitel	0132121360
Abbey	Deitel	0132151006
Abbey	Deitel	0132990601
Abbey	Deitel	0133406954
Harvey	Deitel	0132121360
Harvey	Deitel	0132151006
Harvey	Deitel	0132575663
Harvey	Deitel	013299044X
Harvey	Deitel	0132990601
Harvey	Deitel	0133378713
Harvey	Deitel	0133379337
Harvey	Deitel	0133406954
Harvey	Deitel	0136151574
Paul	Deitel	0132121360
Paul	Deitel	0132151006
Paul	Deitel	0132575663
Paul	Deitel	013299044X
Paul	Deitel	0132990601
Paul	Deitel	0133378713
Paul	Deitel	0133379337
Paul	Deitel	0133406954
Paul	Deitel	0136151574
Michael	Morgano	0132121360
Dan	Quirk	0136151574

б) Список авторов и названий написанных ими книг; авторы сортируются сначала по фамилиям, затем по именам; книги каждого автора сортируются по алфавиту



Authors and titles:		
Abbey	Deitel	Android for Programmers: An App-Driven Approach
Abbey	Deitel	Internet & World Wide Web How to Program
Abbey	Deitel	Simply Visual Basic 2010
Abbey	Deitel	Visual Basic 2012 How to Program
Harvey	Deitel	Android for Programmers: An App-Driven Approach
Harvey	Deitel	C How to Program
Harvey	Deitel	C++ How to Program
Harvey	Deitel	Internet & World Wide Web How to Program
Harvey	Deitel	Java How to Program
Harvey	Deitel	Simply Visual Basic 2010
Harvey	Deitel	Visual Basic 2012 How to Program
Harvey	Deitel	Visual C# 2012 How to Program
Harvey	Deitel	Visual C++ 2008 How to Program
Paul	Deitel	Android for Programmers: An App-Driven Approach
Paul	Deitel	C How to Program
Paul	Deitel	C++ How to Program
Paul	Deitel	Internet & World Wide Web How to Program
Paul	Deitel	Java How to Program
Paul	Deitel	Simply Visual Basic 2010
Paul	Deitel	Visual Basic 2012 How to Program
Paul	Deitel	Visual C# 2012 How to Program
Paul	Deitel	Visual C++ 2008 How to Program
Michael	Morgano	Android for Programmers: An App-Driven Approach
Dan	Quirk	Visual C++ 2008 How to Program

в) Список названий книг, сгруппированный по авторам; авторы сортируются сначала по фамилиям, затем по именам; книги каждого автора сортируются по алфавиту



Titles grouped by author:	
Abbey Deitel:	Android for Programmers: An App-Driven Approach
	Internet & World Wide Web How to Program
	Simply Visual Basic 2010
	Visual Basic 2012 How to Program
Harvey Deitel:	Android for Programmers: An App-Driven Approach
	C How to Program
	C++ How to Program
	Internet & World Wide Web How to Program
	Java How to Program
	Simply Visual Basic 2010
	Visual Basic 2012 How to Program
	Visual C# 2012 How to Program
	Visual C++ 2008 How to Program
Paul Deitel:	Android for Programmers: An App-Driven Approach
	C How to Program
	C++ How to Program
	Internet & World Wide Web How to Program
	Java How to Program
	Simply Visual Basic 2010
	Visual Basic 2012 How to Program
	Visual C# 2012 How to Program
	Visual C++ 2008 How to Program
Michael Morgano:	Android for Programmers: An App-Driven Approach
Dan Quirk:	Visual C++ 2008 How to Program

Ил. 22.23. Результат выполнения приложения JoinQueries

- ❑ Получение списка всех авторов и названий написанных ими книг, отсортированного сначала по фамилиям, затем по именам; книги каждого автора сортируются по алфавиту (ил. 22.23, б).
- ❑ Получение списка названий всех книг, сгруппированных по автору, отсортированного сначала по фамилиям, затем по именам; книги каждого автора сортируются по алфавиту (ил. 22.23, в).

### Графический интерфейс приложения

Для нового приложения (ил. 22.24–22.27) выполните действия из раздела 22.5.2, чтобы создать новый проект приложения Windows Forms с именем `JoinQueries` в одном решении с предыдущими примерами. Переименуйте файл `Form1.cs` в `JoiningTableData.cs`. Задайте свойству `Text` объекта формы значение `Joining Tables with LINQ`. Не забудьте назначить `JoinQueries` стартовым проектом. Задайте следующие свойства объекта `outputTextBox`:

- ❑ Свойство `Font`: выберите значение `Lucida Console` для вывода результатов моноширинным шрифтом.
- ❑ Свойство `Anchor`: выберите значение `Top, Bottom, Left, Right`, чтобы при изменении размеров окна размеры элемента управления `outputTextBox` изменялись соответствующим образом.
- ❑ Свойство `Scrollbars`: выберите значение `Vertical` для прокрутки вывода.

### Создание объекта `DbContext`

В коде приложения классы модели данных сущностей используются для объединения данных из таблиц в базе данных `Books` и вывода отношений между авторами и книг в трех разных вариантах. Код класса `JoiningTableData` разделен на несколько листингов (см. ил. 22.24–22.27) для удобства вывода. Как и в предыдущих примерах, объект `DbContext` (см. ил. 22.24, строки 19–20) обеспечивает взаимодействие программы с базой данных.

```
1 // Ил. 22.24: JoiningTableData.cs
2 // Использование LINQ для объединения данных из нескольких таблиц.
3 using System;
4 using System.Linq;
5 using System.Windows.Forms;
6
7 namespace JoinQueries
8 {
9     public partial class JoiningTableData : Form
10     {
11         public JoiningTableData()
12         {
13             InitializeComponent();
14         } // Конец конструктора
```

**Ил. 22.24.** Создание объекта `BooksDataContext` для запросов к базе данных `Books` (продолжение ↗)

```

15
16     private void JoiningTableData_Load(object sender, EventArgs e)
17     {
18         // Entity Framework DbContext
19         BooksExamples.BooksEntities dbcontext =
20             new BooksExamples.BooksEntities();
21

```

**Ил. 22.24.** Создание объекта BooksDataContext для запросов к базе данных Books (окончание)

### Объединение имен авторов с номерами ISBN написанных ими книг

Первый запрос (см. ил. 22.25, строки 24–27) объединяет данные из двух таблиц и возвращает список имен авторов и номеров ISBN написанных ими книг, отсортированный по полям LastName и FirstName. Запрос использует свойства классов модели данных сущностей, созданные на основании отношений внешнего ключа между таблицами базы данных. Эти свойства позволяют легко объединять данные взаимосвязанных строк из нескольких таблиц.

```

22         // get authors and ISBNs of each book they co-authored
23         var authorsAndISBNs =
24             from author in dbcontext.Authors
25             from book in author.Titles
26             orderby author.LastName, author.FirstName
27             select new { author.FirstName, author.LastName, book.ISBN };
28
29         outputTextBox.AppendText( "Authors and ISBNs:" );
30
31         // display authors and ISBNs in tabular format
32         foreach ( var element in authorsAndISBNs )
33         {
34             outputTextBox.AppendText(
35                 String.Format( "\r\n\t{0,-10} {1,-10} {2,-10}",
36                     element.FirstName, element.LastName, element.ISBN ) );
37         } // end foreach
38

```

**Ил. 22.25.** Получение списка авторов и номеров ISBN написанных ими книг

Первая секция from (строка 24) выбирает всех авторов из таблицы Authors. Вторая секция from (строка 25) использует сгенерированное свойство Titles класса Author для получения номера ISBN текущего автора. Модель данных сущностей использует информацию внешнего ключа, хранящуюся в таблице AuthorISBN, для получения соответствующих номеров ISBN. Объединенный результат двух секций from представляет собой коллекцию всех авторов и номеров ISBN написанных ими книг. Две секции from вводят две диапазонные переменные в области действия запроса — другие условия могут обращаться к обеим диапазонным переменным для объединения данных из нескольких таблиц. Строка 26 упорядочивает результаты сначала по фамилии автора (LastName), затем по имени (FirstName). В строке 27 создается новый анонимный тип, содержащий имя и фамилию автора из таблицы Authors с номером ISBN книги из таблицы Titles, написанной этим автором.

### Анонимные типы

Как вы знаете, анонимные типы позволяют создавать простые классы, предназначенные для хранения данных без написания определения класса. Объявление анонимного типа (строка 27), формально называемое *выражением создания анонимного объекта*, напоминает инициализатор объекта (раздел 10.13). Объявление анонимного типа начинается с ключевого слова `new`, за которым следует список инициализаторов в фигурных скобках (`{}`). После ключевого слова `new` имя класса не указывается. На основании выражения создания анонимного объекта компилятор генерирует определение класса. Этот класс содержит свойства, указанные в списке инициализаторов, — `FirstName`, `LastName` и `ISBN`. Все свойства анонимного типа являются открытыми. Свойства анонимного типа доступны только для чтения — после того, как объект будет создан, изменить значение свойства невозможно. Тип каждого свойства определяется по присваиваемым ему значениям. Определение класса генерируется автоматически, поэтому имя типа класса в приложении неизвестно (отсюда и термин «анонимный тип»). Таким образом, для хранения ссылок на объекты анонимных типов необходимо использовать локальные переменные с неявной типизацией (строка 32). Хотя здесь эта возможность не используется, компилятор при создании определения класса анонимного типа определяет метод `ToString`. Метод возвращает строку в фигурных скобках, содержащую список пар «ИмяСвойства = значение», разделенных запятыми. Компилятор также предоставляет метод `Equals`, который сравнивает свойства анонимного объекта, для которого вызывается метод, и анонимный объект, полученный в аргументе.

### Объединение имени автора с названиями книг

Второй запрос (см. ил. 22.26, строки 41–45) выдает похожие результаты, но использует отношения внешнего ключа для получения названий всех книг, написанных автором.

```

39      // Получение авторов и названий книг
40      var authorsAndTitles =
41          from book in dbcontext.Titles
42          from author in book.Authors
43          orderby author.LastName, author.FirstName, book.Title1
44          select new { author.FirstName, author.LastName,
45                      book.Title1 };
46
47      outputTextBox.AppendText( "\r\n\r\nAuthors and titles:" );
48
49      // Вывод авторов и книг в табличном формате
50      foreach ( var element in authorsAndTitles )
51      {
52          outputTextBox.AppendText(
53              String.Format( "\r\n\t{0,-10} {1,-10} {2}",
54                          element.FirstName, element.LastName, element.Title1 ) );
55      } // Конец foreach
56

```

**Ил. 22.26.** Получение списка авторов и названий написанных ими книг

Первая секция `from` (строка 41) получает все книги из таблицы `Titles`. Вторая секция `from` (строка 42) использует сгенерированное свойство `Authors` класса

`Title` для получения авторов только текущей книги. Модель данных сущностей использует информацию внешнего ключа, хранящуюся в таблице `AuthorISBN`, для получения соответствующих авторов. Объекты `author` предоставляют доступ к именам авторов текущей книги. Условие `select` (строки 44–45) используют диапазонные переменные `author` и `book`, упоминавшиеся ранее в этой главе, для получения `FirstName` и `LastName` каждого автора из таблицы `Authors` и названия каждой книги из таблицы `Titles`.

### Группировка названий книг по автору

Многие запросы возвращают результаты со стандартной для реляционных таблиц структурой из строк и столбцов. Последний запрос (см. ил. 22.27, строки 60–66) возвращает иерархические результаты. Каждый элемент результата содержит имя автора и список названий книг, написанных этим автором. Для этого используется *вложенный запрос* в секции `select`: внешний запрос перебирает авторов в базе данных, а внутренний берет конкретного автора и выбирает все названия книг, написанных указанным автором. Секция `select` (строки 62–66) создает анонимный тип с двумя свойствами:

- ❑ Свойство `Name` (строка 62) строит полное имя автора, разделяя имя и фамилию пробелом.
- ❑ Свойство `Titles` (строка 63) получает результат вложенного запроса, который возвращает названия всех книг автора.

В данном случае мы задаем имена свойств анонимного типа. При создании анонимного типа имена свойств указываются в формате *имя = значение*.

```
57         // Получение авторов и названий книг
58         // с группировкой по авторам
59         var titlesByAuthor =
60             from author in dbcontext.Authors
61             orderby author.LastName, author.FirstName
62             select new { Name = author.FirstName + " " + author.LastName,
63                 Titles =
64                     from book in author.Titles
65                     orderby book.Title1
66                     select book.Title1 };
67
68         outputTextBox.AppendText( "\r\n\r\nTitles grouped by author:" );
69
70         // Вывод книг, написанных каждым автором, с группировкой по авторам
71         foreach ( var author in titlesByAuthor )
72         {
73             // Вывод имени автора
74             outputTextBox.AppendText( "\r\n\t" + author.Name + ":" );
75
76             // Вывод книг, написанных автором
77             foreach ( var title in author.Titles )
```

**Ил. 22.27.** Получение списка книг, сгруппированных по авторам (продолжение ↗)



```

78         {
79             outputTextBox.AppendText( "\r\n\t\t" + title );
80         } // Конец внутреннего цикла foreach
81     } // Конец внешнего цикла foreach
82 } // Конец метода JoiningTableData_Load
83 } // Конец класса JoiningTableData
84 } // Конец пространства имен JoinQueries

```

**Ил. 22.27.** Получение списка книг, сгруппированных по авторам (окончание)

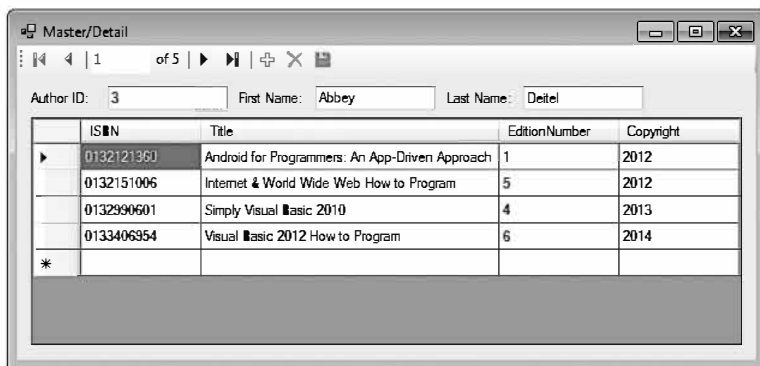
Диапазонная переменная `range` во вложенном запросе перебирает книги текущего автора с использованием свойства `Titles`. Свойство `Title1` конкретной книги возвращает столбец `Title` строки таблицы `Titles` базы данных.

Вложенные команды `foreach` (строки 71–81) используют свойства анонимного типа, созданного запросом для вывода иерархических результатов. Внешний цикл выводит имя автора, а внутренний цикл — названия всех книг, написанных этим автором.

## 22.8. Создание приложения типа «главное/детализированное представление»

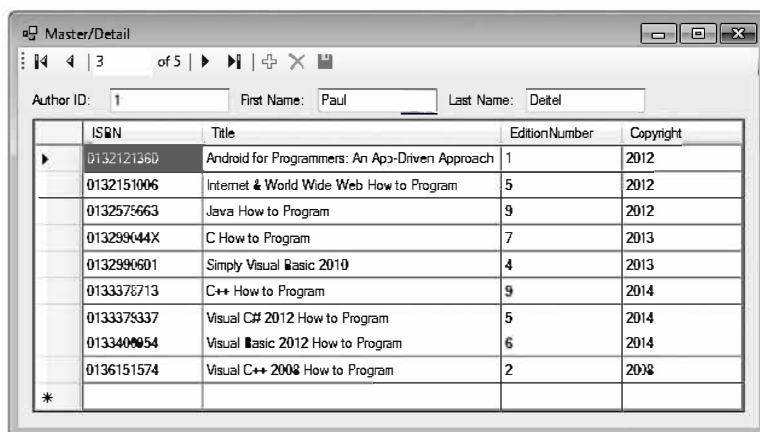
Приложение на ил. 22.28 относится к типу «главное/детализированное представление» — одна часть графического интерфейса (главное представление) позволяет выбрать запись, а в другой части (детализированном представлении) выводится подробная информация об этой записи. При загрузке приложения выводится имя первого автора в источнике данных и книги этого автора (см. ил. 22.28, а). Когда вы используете кнопки панели `BindingNavigator` для изменения автора, приложение выводит подробную информацию о книгах, написанных этим автором (см. ил. 22.28, б). Приложение ограничивается чтением данных из модели данных сущностей, поэтому мы заблокировали кнопки добавления и удаления записей на панели `BindingNavigator`.

а) Приложение  
выводит список  
книг первого  
автора из  
источника данных



**Ил. 22.28.** Приложение MasterDetail (продолжение ↗)

6) Приложение выводит список книг третьего автора из источника данных



Ил. 22.28. Приложение MasterDetail (окончание)

Запустите приложение и поэкспериментируйте с элементами управления Binding-Navigator. Кнопки управления похожи на кнопки DVD-плеера и позволяют изменить текущую строку.

## 22.8.1. Создание графического интерфейса приложения MasterDetail

Вы уже видели, что IDE может автоматически генерировать компоненты Binding-Source, BindingNavigator и элементы графического интерфейса при перетаскивании источника данных на форму. Сейчас мы используем два объекта BindingSource — один для главного списка авторов, другой для названий, связанных с выбранным автором. Оба объекта будут генерироваться средой разработки.

### Шаг 1. Создание проекта

Выполните действия из подраздела 22.5.2, чтобы создать новый проект приложения Windows Forms с именем MasterDetail. Присвойте файлу исходного кода имя Details.cs и задайте свойству Text объекта формы значение Master/Detail.

### Шаг 2. Добавление источника данных для таблицы Authors

Выполните шаги из подраздела 22.5.3 для создания источника данных для таблицы Authors. Хотя мы будем выводить записи из таблицы Titles для каждого автора, добавлять источник данных для этой таблицы не нужно. Информация о названиях книг будет браться из навигационного свойства Titles в классе модели данных сущностей Author.

### Шаг 3. Создание элементов графического интерфейса

Затем используйте режим конструктора для создания компонентов графического интерфейса — перетащите элементы из окна Data Sources на форму. Ранее вы

перетаскивали объект из окна **Data Sources** на форму для создания **DataGridView**. IDE позволяет задать тип элемента(-ов) управления, который будет создан при перетаскивании объекта из окна **Data Sources** на форму. Для этого выполните следующие действия:

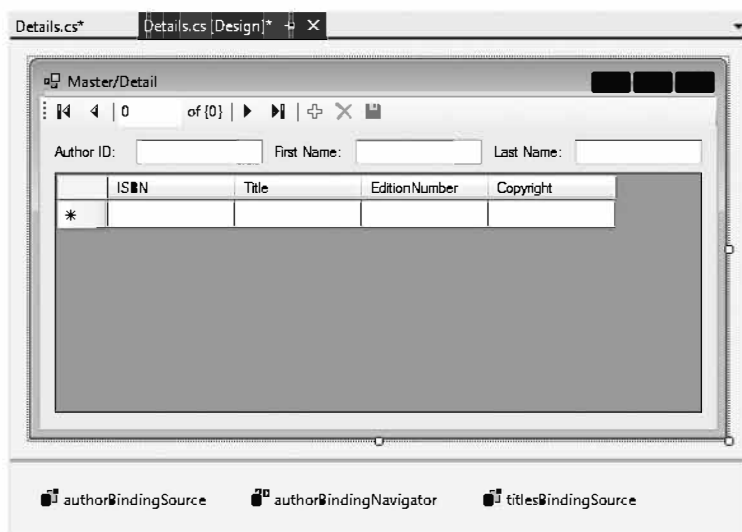
1. Переключитесь в режим конструктора для класса **Details**.
2. Щелкните на узле **Author** в окне **Data Sources** — он должен превратиться в раскрывающийся список. Откройте список, щелкнув на кнопке со стрелкой, и выберите вариант **Details** — это означает, что мы хотим генерировать пары **Label–TextBox**, представляющие каждый столбец таблицы **Authors**.
3. Перетащите узел **Author** из окна **Data Sources** на форму в режиме конструктора. При этом создаются объекты **authorBindingSource**, **authorBindingNavigator** и пары **Label–TextBox**, представляющие каждый столбец в таблице. Изначально элементы выглядят так, как показано на ил. 22.29. Мы изменили расположение элементов управления так, как показано на ил. 22.28.



**Ил. 22.29.** Детализированное представление автора

4. По умолчанию навигационное свойство **Titles** реализуется в классах модели данных сущности в формате **HashSet<Title>**. Чтобы привязка к элементам графического интерфейса была выполнена правильно, необходимо заменить его типом **ObservableCollection<Title>**. Для этого раскройте узел **BooksModel.edmx** проекта библиотеки классов в окне **Solution Explorer**, затем раскройте узел **BooksModel.tt** и откройте файл **Author.cs** в редакторе. Добавьте команду **using** для пространства имен **System.Collections.ObjectModel**. Затем в теле конструктора **Author** замените **HashSet** на **ObservableCollection**. Щелкните правой кнопкой мыши на проекте библиотеки классов в окне **Solution Explorer** и выберите команду **Build**, чтобы перекомпилировать класс.
5. Щелкните на узле **Titles**, вложенном в узел **Author** в окне **Data Sources**, — он должен превратиться в раскрывающийся список. Откройте раскрывающийся список, щелкнув на кнопке со стрелкой, и убедитесь в том, что в списке выбран вариант **DataGridView** — этот элемент управления будет использоваться для отображения данных из таблицы **Titles**, соответствующих заданному автору.
6. Перетащите узел **Titles** на форму в режиме конструктора. При этом создается объект **titlesBindingSource** и элемент управления **DataGridView**. Поскольку элемент управления предназначен только для просмотра данных, задайте его свойству **ReadOnly** значение **True** в окне свойств. Так как мы перетащили узел **Titles** с узла **Author** в окне свойств, **DataGridView** автоматически выводит книги текущего автора после привязки данных автора к **authorBindingSource**.

Мы использовали свойство `Anchor` элемента управления `DataGridView` для закрепления его за всеми четырьмя сторонами формы. Мы также задали свойствам `Size` и `MinimumSize` значения 550, 300, чтобы задать исходный и минимальный размеры формы соответственно. Полный графический интерфейс изображен на ил. 22.30.



Ил. 22.30. Интерфейс приложения MasterDetail

## 22.8.2. Код приложения MasterDetail

Код вывода автора и соответствующих книг (ил. 22.31) достаточно тривиален. В строках 17–18 создается объект `DbContext`. Обработчик события `Load` формы (строки 22–32) упорядочивает объекты `Author` по полям `LastName` (строка 26) и `FirstName` (строка 27), после чего загружает их в память (строка 28). Затем в строке 31 свойству `DataSource` объекта `authorBindingSource` присваивается значение `dbContext.Authors.Local`. На этой стадии:

- ❑ на панели `BindingNavigator` выводится количество объектов `Author` и информация о том, что выбран первый результат из набора;
- ❑ в текстовых полях выводятся значения свойств `AuthorID`, `FirstName` и `LastName` текущего выбранного объекта `Author`;
- ❑ набор книг выбранного автора автоматически присваивается свойству `DataSource` объекта `titlesBindingSource`, в результате чего эти книги выводятся в элементе управления `DataGridView`.

Теперь при выборе другого автора на панели `BindingNavigator` в элементе управления `DataGridView` выводятся соответствующие названия книг.

```

1 // Ил. 22.31: Details.cs
2 // Использование DataGridView для вывода детализации выбранной записи.
3 using System;
4 using System.Data.Entity;
5 using System.Linq;
6 using System.Windows.Forms;
7
8 namespace MasterDetail
9 {
10     public partial class Details : Form
11     {
12         public Details()
13         {
14             InitializeComponent();
15         } // Конец конструктора
16
17         // Entity Framework DbContext
18         BooksExamples.BooksEntities dbcontext =
19             new BooksExamples.BooksEntities();
20
21         // Инициализация источников данных при загрузке формы
22         private void Details_Load( object sender, EventArgs e )
23         {
24             // Загрузка данных Authors, упорядоченных по LastName и FirstName
25             dbcontext.Authors
26                 .OrderBy( author => author.LastName )
27                 .ThenBy( author => author.FirstName )
28                 .Load();
29
30             // Назначение DataSource для authorBindingSource
31             authorBindingSource.DataSource = dbcontext.Authors.Local;
32         } // Конец метода Details_Load
33     } // Конец класса Details
34 } // Конец пространства имен MasterDetail

```

**Ил. 22.31.** Использование DataGridView для вывода детализированной информации для выбранного автора

## 22.9. Адресная книга

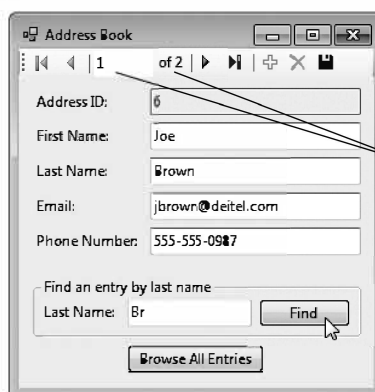
В нашем последнем примере (ил. 22.32) реализуется простое приложение адресной книги для выполнения различных операций с базой данных AddressBook.mdf (включенной в каталог примеров настоящей главы):

- ☐ Вставка новых контактов.
- ☐ Поиск контактов, фамилии которых начинаются с заданных букв.
- ☐ Обновление существующих контактов.
- ☐ Удаление контактов.

а) Элементы управления BindingNavigator используются для перебора контактов в базе данных



б) Введите искомый текст в поле Last Name: и нажмите кнопку Find, чтобы найти контакты, начинающиеся с указанной подстроки. С префикса «Br» начинаются только две фамилии, поэтому BindingNavigator отображает две подходящие записи



Выводится первый из двух контактов, соответствующих текущему условию поиска

в) Щелкните на кнопке Browse All Entries, чтобы очистить условие поиска и вернуться к выводу всех контактов в базе данных



Теперь можно просматривать все шесть контактов

**Ил. 22.32.** Работа с адресной книгой

В базу данных заносятся шесть фиктивных контактов.

Вместо того чтобы выводить таблицу базы данных в `DataGridView`, это приложение выводит подробную информацию об одном контакте в нескольких текстовых полях. Панель `BindingNavigator` в верхней части окна используется для выбора строки

таблицы, данные которой выводятся в каждый конкретный момент. При помощи панели **BindingNavigator** также можно добавлять и удалять контакты — но только во время просмотра *полного* списка контактов. Если список контактов отфильтрован по фамилии, приложение блокирует кнопки **Add new** (+) и **Delete** (X) (вскоре мы объясним, почему). Щелчок на кнопке **Browse All Entries** снова открывает доступ к этим кнопкам. При добавлении новой записи текстовые поля очищаются, а в текстовом поле справа от надписи **Address ID** выводится нуль; это означает, что текстовые поля представляют новую запись. При сохранении новой записи нуль в поле **Address ID** автоматически заменяется уникальным идентификатором в базе данных. Изменения в базу данных вносятся только при щелчке на кнопке **Save Data** (💾).

## 22.9.1. Создание графического интерфейса

Код приложения будет рассмотрен чуть позднее, а сначала мы займемся созданием модели данных сущностей и приложения **Windows Forms**.

### Шаг 1. Создание проекта библиотеки классов для модели данных сущностей

Выполните действия, описанные в разделе 22.5.1, для создания проекта библиотеки классов с именем **AddressExample**. Библиотека содержит модель данных сущностей для базы данных **AddressBook.mdf**, которая состоит из таблицы **Addresses** со столбцами **AddressID**, **FirstName**, **LastName**, **Email** и **PhoneNumber**. Присвойте модели данных сущностей имя **AddressModel.edmx**. База данных **AddressBook.mdf** находится в папке **Databases** из папки примеров этой главы.

### Шаг 2. Создание проекта приложения **Windows Forms** для приложения **AddressBook**

Выполните действия, описанные в разделе 22.5.2, для создания проекта приложения **Windows Forms** с именем **AddressBook** в решении **AddressExample**. Выберите для файла с формой имя **Contacts.cs**, затем задайте свойству **Text** объекта формы значение **Address Book**. Назначьте **AddressBook** стартовым проектом решения.

### Шаг 3. Добавление объекта **Address** как источника данных

Добавьте объект **Address** модели данных сущностей как источник данных (как было сделано с объектом **Author** на шаге 1 в разделе 22.5.3).

### Шаг 4. Вывод подробной информации о каждой строке

В режиме конструктора выберите узел **Address** в окне **Data Sources**. Откройте список, щелкнув на кнопке со стрелкой, и выберите вариант **Details** — это означает, что среда разработки должна сгенерировать пары **Label**–**TextBox** для вывода детализированной информации по одной записи.

### Шаг 5. Перетаскивание узла **Address** на форму

Перетащите узел **Address** из окна **Data Sources** на форму. При этом автоматически создается панель **BindingNavigator** и пары **Label**–**TextBox**, соответствующие столбцам

таблицы. Поля размещаются в алфавитном порядке. Разместите компоненты в режиме конструктора в порядке, показанном на ил. 22.32. Также следует изменить порядок обхода элементов управления; для этого выберите команду **VIEW ► Tab Order** и щелкайте на текстовых полях в порядке их следования на ил. 22.32.

### Шаг 6. Ограничение доступа к полю Address ID

Столбец `AddressID` таблицы `Addresses` содержит идентификатор, значение которого генерируется автоматически, поэтому редактирование этого столбца пользователем следует запретить. Выделите текстовое поле `Address ID` и задайте его свойству `ReadOnly` значение `True` в окне свойств.

### Шаг 7. Добавление элементов управления для поиска по фамилии

Хотя панель `BindingNavigator` позволяет просматривать данные адресной книги, было бы удобнее иметь возможность найти нужную запись по фамилии. Чтобы добавить эту функциональность в приложение, необходимо создать элементы управления для ввода фамилии и предоставить обработчики событий для выполнения поиска.

Добавьте на форму надпись с именем `findLabel1`, текстовое поле с именем `findTextBox` и кнопку с именем `findButton`. Разместите эти элементы управления в контейнере `GroupBox` с именем `findGroupBox`, после чего задайте его свойству `Text` значение `Find an entry by last name`. Задайте свойству `Text` надписи значение `Last Name:`, а свойству `Text` кнопки — значение `Find`.

### Шаг 8. Возврат к просмотру всех строк базы данных

Чтобы пользователь после поиска контактов с заданной фамилией мог вернуться к просмотру всех контактов, добавьте на форму кнопку с именем `browseAllButton` под контейнером `findGroupBox`. Задайте свойству `Text` кнопки `browseAllButton` значение `Browse All Entries`.

## 22.9.2. Код приложения адресной книги

Файл программной логики `Contacts.cs` разделен на несколько листингов (ил. 22.33–22.37) для удобства вывода.

### Метод `RefreshContacts`

Как было показано в предыдущих примерах, объект `addressBindingSource`, управляющий графическим интерфейсом, необходимо связать с объектом `DbContext`, взаимодействующим с базой данных. В нашем примере объект `DbContext` для `AddressEntities` объявляется в строке 20 ил. 22.33, но его создание и привязка данных выполняются в методе `RefreshContacts` (строки 23–43), который вызывается из других методов приложения. При вызове этого метода, если ссылка `dbcontext` отлична от `null`, мы вызываем метод `Dispose` для этого объекта, после чего создаем новый объект `DbContext` в строке 30. Это делается для пересортировки данных в модели данных сущностей. Если бы один объект `dbcontext.Addresses` поддерживался в памяти на протяжении всего времени работы программы, а пользователь изменил бы



имя или фамилию контакта, записи сохранили бы свой исходный порядок в объекте `dbContext.Addresses`, даже если бы этот порядок был неправильным. В строках 34–37 объекты `Address` упорядочиваются сначала по `LastName`, затем по `FirstName`, после чего загружаются в память. Далее строка 40 задает свойству `DataSource` объекта `addressBindingSource` значение `dbContext.Addresses.Local`, чтобы связать данные в памяти с графическим интерфейсом.

```

1  // Ил. 22.33: Contact.cs
2  // Работа с адресной книгой.
3  using System;
4  using System.Data;
5  using System.Data.Entity;
6  using System.Data.Entity.Validation;
7  using System.Linq;
8  using System.Windows.Forms;
9
10 namespace AddressBook
11 {
12     public partial class Contacts : Form
13     {
14         public Contacts()
15         {
16             InitializeComponent();
17         } // Конец конструктора
18
19         // Entity Framework DbContext
20         private AddressExample.AddressBookEntities dbContext = null;
21
22         // Заполнение addressBindingSource строками, упорядоченными по имени
23         private void RefreshContacts()
24         {
25             // Уничтожение старого объекта DbContext, если он существует
26             if ( dbContext != null )
27                 dbContext.Dispose();
28
29             // Создание нового объекта DbContext для переупорядочения записей при правке
30             dbContext = new AddressExample.AddressBookEntities();
31
32             // Использование LINQ для упорядочения содержимого таблицы
33             // Addresses по фамилии, затем по имени
34             dbContext.Addresses
35                 .OrderBy( entry => entry.LastName )
36                 .ThenBy( entry => entry.FirstName )
37                 .Load();
38
39             // Назначение DataSource для addressBindingSource
40             addressBindingSource.DataSource = dbContext.Addresses.Local;
41             addressBindingSource.MoveFirst(); // Переход к первому результату
42             findTextBox.Clear(); // Очистка текстового поля Find
43         } // Конец метода RefreshContacts
44

```

**Ил. 22.33.** Создание объекта `BooksDataContext` и определение метода `RefreshContacts` для использования в других методах

### Метод `Contacts_Load`

Метод `Contacts_Load` (см. ил. 22.34) вызывает `RefreshContacts` (строка 48), чтобы при запуске приложения выводилась первая запись. Как и прежде, обработчик события `Load` создается двойным щелчком на заголовке формы.

```
45      // При загрузке форма заполняется данными из базы
46      private void Contacts_Load( object sender, EventArgs e )
47      {
48          RefreshContacts(); // Заполнение данными из базы
49      } // Конец метода Contacts_Load
50
```

**Ил. 22.34.** Вызов `RefreshContacts` для заполнения текстовых полей при загрузке приложения

### Метод `addressBindingNavigatorSaveItem_Click`

Метод `addressBindingNavigatorSaveItem_Click` (см. ил. 22.35) сохраняет изменения в базе данных при щелчке на кнопке `Save Data` на панели `BindingNavigator`. (Не забудьте разблокировать эту кнопку.) В базе данных `AddressBook` обязательными являются значения имени, фамилии, телефона и электронной почты. Если при попытке сохранения поле оказывается пустым, возникает исключение `DbEntityValidationException`. После сохранения вызывается метод `RefreshContacts` для повторной сортировки данных, после чего происходит возврат к первому элементу.

```
51      // Обработчик события Click для кнопки Save в BindingNavigator
52      // сохраняет изменения, внесенные в базу данных
53      private void addressBindingNavigatorSaveItem_Click(
54          object sender, EventArgs e )
55      {
56          Validate(); // Проверка полей ввода
57          addressBindingSource.EndEdit(); // Завершение текущей правки
58
59          // Попытка сохранения изменений
60          try
61          {
62              dbcontext.SaveChanges(); // Запись изменений в файл базы данных
63          } // Конец try
64          catch ( DbEntityValidationException )
65          {
66              MessageBox.Show( "Columns cannot be empty",
67                              "Entity Validation Exception" );
68          } // Конец catch
69
70          RefreshContacts(); // Возврат к исходным нефильТРованным данным
71      } // Конец метода addressBindingNavigatorSaveItem_Click
72
```

**Ил. 22.35.** Сохранение изменений в базе данных при щелчке на кнопке `Save Data`

### Метод `findButton_Click`

Метод `findButton_Click` (см. ил. 22.36) использует синтаксис запросов LINQ (строки 79–83) для выбора людей, фамилии которых начинаются с текста, введенного в `findTextBox`. Запрос сортирует результаты по фамилии, затем по имени.

Технология LINQ to Entities не позволяет напрямую привязать результаты запроса LINQ к свойству DataSource объекта BindingSource. По этой причине в строке 86 вызывается метод ToList объекта запроса для получения представления отфильтрованных данных в виде объекта List, который присваивается свойству DataSource объекта BindingSource. При преобразовании результата запроса в List в DbContext отслеживаются только изменения в существующих записях DbContext — все записи, добавленные или удаленные во время просмотра отфильтрованных данных, будут потеряны. По этой причине мы блокируем кнопки добавления и удаления во время вывода отфильтрованных данных. Когда вы вводите фамилию и щелкаете на кнопке Find, панель BindingNavigator позволяет пользователю просмотреть только записи, содержащие подходящие фамилии. Это связано с тем, что источник данных, привязанный к элементам управления формы (результат запроса LINQ), изменился и теперь содержит ограниченное подмножество строк.

```

73      // Использование LINQ для создания источника данных, содержащего
74      // только контакты с фамилиями, начинающимися с заданного префикса
75      private void findButton_Click( object sender, EventArgs e )
76      {
77          // Использование LINQ для фильтрации контактов с фамилиями,
78          // начинающимися с содержимого findTextBox
79          var lastNameQuery =
80              from address in dbContext.Addresses
81              where address.LastName.StartsWith( findTextBox.Text )
82              orderby address.LastName, address.FirstName
83              select address;
84
85          // Вывод подходящих контактов
86          addressBindingSource.DataSource = lastNameQuery.ToList();
87          addressBindingSource.MoveFirst(); // Переход к первому результату
88
89          // Запретить добавление/удаление в режиме фильтрации
90          bindingNavigatorAddNewItem.Enabled = false;
91          bindingNavigatorDeleteItem.Enabled = false;
92      } // Конец метода findButton_Click
93

```

**Ил. 22.36.** Поиск контактов, фамилии которых начинаются с заданной строки

### Метод browseAllButton\_Click

Метод browseAllButton\_Click (ил. 22.37) возвращает пользователя к просмотру всех строк после поиска конкретного префикса. Сделайте двойной щелчок на кнопке browseAllButton, чтобы создать обработчик события click. Обработчик снимает блокировку с кнопок Add new и Delete, после чего вызывает метод RefreshContacts для восстановления в источнике данных полного списка контактов (в порядке сортировки) и очистки findTextBox.

```

94      // Повторная загрузка всех строк в addressBindingSource
95      private void browseAllButton_Click( object sender, EventArgs e )
96      {
97          // Без фильтрации кнопки добавления/удаления доступны

```

**Ил. 22.37.** Возвращение к просмотру всех контактов (продолжение ↗)

```
98         bindingNavigatorAddNewItem.Enabled = true;
99         bindingNavigatorDeleteItem.Enabled = true;
100         RefreshContacts(); // Возврат к исходным нефильтрованным данным
101     } // Конец метода browseButton_Click
102 } // Конец класса Contacts
103 } // Конец пространства имен AddressBook
```

**Ил. 22.37.** Возвращение к просмотру всех контактов (окончание)

## 22.10. Инструменты и сетевые ресурсы

Раздел LINQ Resource Center по адресу [www.deitel.com/LINQ](http://www.deitel.com/LINQ) содержит много ссылок на дополнительную информацию, включая блоги участников группы LINQ из Microsoft, учебные материалы, видеоролики, загружаемые файлы, списки FAQ, веб-касты и другие ресурсы.

Еще один полезный инструмент для изучения LINQ — LINQPad ([www.linqpad.net](http://www.linqpad.net)) — позволяет выполнить и просмотреть результаты любого выражения C# или Visual Basic, включая запросы LINQ. Программа также поддерживает ADO.NET Entity Framework и LINQ to Entities.

Эта глава лишь в общих чертах знакомит читателя с базами данных, ADO.NET Entity Framework и LINQ to Entities. На сайте Entity Framework компании Microsoft представлено много дополнительной информации о работе с ADO.NET Entity Framework и LINQ to Entities: учебные материалы, видеоролики и т. д.

[msdn.microsoft.com/en-us/data/aa937723](http://msdn.microsoft.com/en-us/data/aa937723)

## 22.11. Итоги

Эта глава знакомит читателя с моделью реляционных баз данных, ADO.NET Entity Framework, LINQ to Entities и средствами визуального программирования Visual Studio 2012 по работе с базами данных. Мы рассмотрели содержимое простой базы данных Books и отношения между таблицами базы данных. Технология LINQ to Entities и классы модели данных сущностей, сгенерированной IDE, использовались для чтения, добавления, удаления и обновления данных в базах данных SQL Server Express.

Мы рассмотрели классы модели данных сущностей, автоматически генерируемые IDE, — например, класс `DbContext`, управляющий взаимодействиями приложения с базой данных. Вы научились использовать инструменты IDE для подключения к базам данных и генерирования классов модели данных сущностей по схеме существующей базы данных. Затем мы использовали перетаскивание, чтобы автоматически построить графический интерфейс для вывода и работы с данными.

# 23 Разработка веб-приложений с использованием ASP.NET

## 23.1. Введение

Эта глава посвящена разработке веб-приложений с использованием технологии Microsoft ASP.NET. Веб-приложения создают веб-контент для клиентов, работающих в браузере.

Мы рассмотрим несколько примеров, демонстрирующих разработку веб-приложений с использованием веб-форм, веб-элементов управления (также называемых серверными элементами управления ASP.NET) и программирования Visual C#. Файлы веб-форм имеют расширение `.aspx` и содержат графический интерфейс веб-страницы. Функциональность веб-форм определяется добавлением веб-элементов управления: надписей, текстовых полей, изображений, кнопок и других компонентов графического интерфейса. Файл веб-формы представляет веб-страницу, отправляемую клиентскому браузеру. Файлы веб-форм часто называются файлами ASPX.

У файла ASPX, созданного в Visual Studio, имеется соответствующий класс, написанный на языке .NET, — в этой книге используется Visual C#. Этот класс содержит обработчики событий, код инициализации, вспомогательные методы и прочий служебный код. Файл, содержащий этот класс, называется *файлом программной логики* (code-behind file) и предоставляет программные реализации файла ASPX.

### **Программные продукты, используемые в этой главе**

Для построения кода и графического интерфейса в этой главе используется Microsoft Visual Studio Express 2012 for Web — бесплатная среда разработки веб-приложений ASP.NET. Полная версия Visual Studio 2012 включает функциональность Visual

Studio Express 2012 for Web, так что приведенные в этой главе инструкции также относятся к Visual Studio 2012.

## 23.2. Основы веб-программирования

В этом разделе мы поговорим о том, что происходит при запросе пользователем веб-страницы в браузере. В простейшей форме веб-страница представляет собой документ *HTML* (HyperText Markup Language) с расширением *.html* или *.htm*, который сообщает информацию о содержимом документа и его формате.

Документы HTML обычно содержат гиперссылки на другие страницы или другие части той же страницы. Когда пользователь щелкает на гиперссылке, веб-сервер находит запрошенную страницу и отправляет ее браузеру пользователя. Также пользователь может ввести адрес запрашиваемой страницы в адресной строке браузера и нажать Enter.

В этой главе мы займемся разработкой веб-приложений с использованием визуальных средств, сходных с теми, которые использовались для приложений Windows Forms в главах 14–15. Для полноценной разработки веб-приложений также необходимо разбираться в HTML5, CSS3 и JavaScript — эти темы рассматриваются в нашем учебнике «Internet & World Wide Web How to Program», 5-е изд. (см. *deitel.com/books/iw3htp5*).

### URI и URL

*URI-адреса* (Uniform Resource Identifiers) используются для идентификации ресурсов в Интернете. URI-адреса, начинающиеся с префикса `http://`, называются *URL-адресами* (Uniform Resource Locators). Типичные URL-адреса обозначают файлы, каталоги или код на стороне сервера, выполняющий такие задачи, как поиск по базе данных, поиск в Интернете и реализация бизнес-логики. Если вам известен URL-адрес общедоступного ресурса, вы можете ввести его в адресной строке своего браузера, чтобы браузер обратился к этому ресурсу.

### Строение URL-адреса

URL-адрес содержит информацию, по которой браузер определяет маршрут к запрошенному ресурсу. Для предоставления доступа к таким ресурсам со стороны веб-клиентов используются *веб-серверы*, наиболее популярными из которых являются Microsoft IIS (Internet Information Services), Apache HTTP Server and Nginx<sup>1</sup>.

Рассмотрим компоненты, из которых состоит URL-адрес.

`http://www.deitel.com/books/downloads.html`

Префикс `http://` указывает, что для получения ресурса должен использоваться протокол *HTTP* (HyperText Transfer Protocol). Веб-протокол HTTP обеспечивает

---

<sup>1</sup> w3techs.com/

взаимодействие между клиентами и серверами. Далее в URL-адресе следует полное имя хоста (*www.deitel.com*) — имя компьютера с веб-сервером, на котором находится ресурс. Этот компьютер называется *хостом*. Имя хоста *www.deitel.com* преобразуется в *IP-адрес* (от Internet Protocol) — числовое значение, однозначно идентифицирующее сервер в Интернете. Сервер *DNS* (Domain Name System) ведет базу данных имен хостов и соответствующих им IP-адресов и выполняет преобразования автоматически.

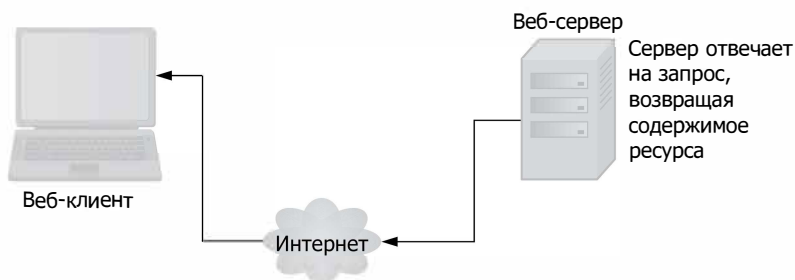
Оставшаяся часть URL-адреса (*/books/downloads.html*) задает местонахождение ресурса (*/books*) и его имя (*downloads.html*) на веб-сервере. Местонахождение может представлять каталог в файловой системе веб-сервера. Однако по соображениям безопасности местонахождение обычно задается в виде виртуального каталога. Веб-сервер преобразует виртуальный каталог в реальный каталог на сервере, скрывая таким образом истинное местонахождение ресурса.

### Отправка запроса и получение ответа

Получив URL-адрес, браузер использует протокол HTTP для загрузки веб-страницы, находящейся по этому адресу. На ил. 23.1 изображен браузер, отправляющий запрос веб-серверу. На ил. 23.2 показано, как веб-сервер реагирует на такой запрос.



**Ил. 23.1.** Клиент запрашивает ресурс с веб-сервера



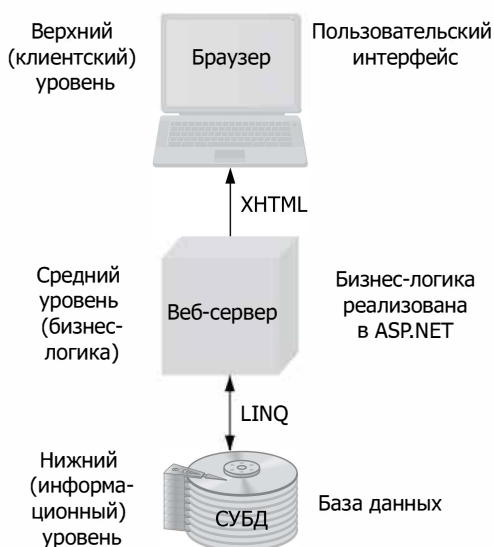
**Ил. 23.2.** Клиент получает ответ от веб-сервера

## 23.3. Многоуровневая архитектура приложений

Веб-приложения имеют *многоуровневую* архитектуру. В многоуровневых приложениях функциональность делится на уровни (то есть логические группы функциональных аспектов). Хотя уровни могут находиться на одном компьютере, в веб-приложениях они чаще размещаются на *разных* компьютерах по соображениям безопасности и масштабируемости. На ил. 23.3 изображена базовая архитектура трехуровневого веб-приложения.

### Информационный уровень

Нижний уровень (также называемый *информационным*) обеспечивает хранение данных приложения. Обычно на этом уровне информация хранится в реляционной базе данных. Например, в магазине может использоваться база данных с информацией о товарах: описания, цены, складские запасы. В той же базе данных могут храниться сведения о клиентах: имена, адреса и номера кредитных карт. На этом уровне могут располагаться несколько баз данных, совокупность которых составляет данные, необходимые для работы приложения.



Ил. 23.3. Трехуровневая архитектура

### Бизнес-логика

Средний уровень реализует бизнес-логику, логику контроллера и представления для управления взаимодействиями между клиентами приложения и его данными. Средний уровень выполняет функции посредника между данными информационного уровня и клиентами. Логика контроллера обрабатывает запросы клиентов (например, запросы на просмотр каталога товаров) и получает информацию из базы данных. Логика представления среднего уровня обрабатывает данные от



информационного уровня и представляет контент клиенту. Веб-приложения представляют данные клиентам в форме веб-страниц.

Бизнес-логика на среднем уровне контролирует соблюдение *бизнес-правил* и проверяет их надежность перед тем, как серверное приложение обновит базу данных или представит данные пользователям. Бизнес-правила управляют обращениями клиентов к данным (или блокировкой этих обращений) и обработкой данных. Например, бизнес-правило среднего уровня веб-приложения магазина может следить за тем, чтобы количество единиц товара было положительным. Запрос клиента, приводящий к заданию отрицательного количества единиц товара в базе данных с информацией о продуктах на нижнем уровне, будет отвергнут бизнес-логикой среднего уровня.

### Клиентский уровень

На верхнем (клиентском) уровне находится пользовательский интерфейс приложения, который занимается получением входных данных и выводом результатов. Пользователи напрямую взаимодействуют с приложением через пользовательский интерфейс (обычно просматриваемый в браузере), используя клавиатуру и мышь. В ответ на действия пользователя (например, щелчок на гиперссылке) клиентский уровень взаимодействует со средним уровнем, выдавая запросы и получая данные от информационного уровня. Затем клиентский уровень выводит для пользователя данные, полученные от среднего уровня. Клиентский уровень никогда не взаимодействует с информационным уровнем напрямую.

## 23.4. Первое веб-приложение

В нашем первом примере в окне браузера выводится время суток на веб-сервере (ил. 23.4). При выполнении этого приложения — то есть при запросе веб-страницы приложения браузером — веб-сервер выполняет код приложения, который запрашивает текущее время и выводит его в элементе управления `Label`. Затем веб-сервер возвращает результат браузеру, выдавшему запрос, а браузер отображает веб-страницу с временем. Мы выполнили это приложение в браузерах Internet Explorer и Firefox, чтобы показать, что веб-страница одинаково выглядит в этих браузерах — как, впрочем, должна выглядеть и в большинстве остальных браузеров.



Ил. 23.4. Веб-приложение WebTime в Internet Explorer и Firefox (продолжение ➤)

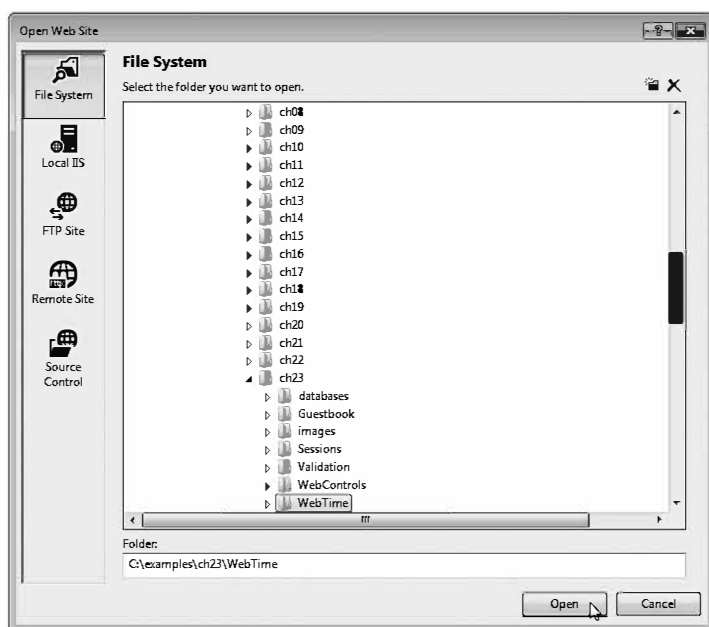


**Ил. 23.4.** Веб-приложение WebTime в Internet Explorer и Firefox (окончание)

### Тестирование приложения в браузере по умолчанию

Чтобы протестировать приложение в браузере по умолчанию, выполните следующие действия:

1. Откройте Visual Studio Express For Web.
2. Выполните команду Open Web Site... из меню FILE.
3. В диалоговом окне Open Web Site (ил. 23.5) убедитесь в том, что выбрана вкладка File System, перейдите к примерам этой главы, выберите папку WebTime и щелкните на кнопке Open.
4. Выделите файл WebTime.aspx в окне Solution Explorer и нажмите Ctrl+F5, чтобы выполнить веб-приложение.

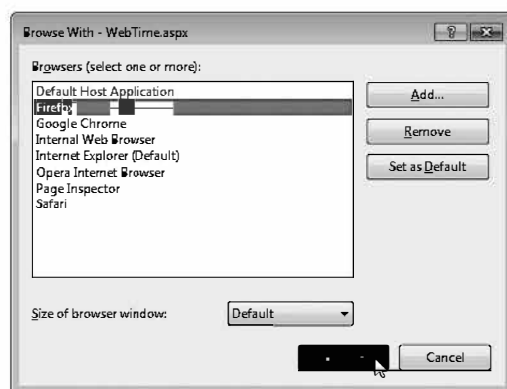


**Ил. 23.5.** Диалоговое окно Open Web Site

### Тестирование приложения в выбранном браузере

Если вы хотите выполнить приложение в другом браузере, скопируйте адрес страницы из адресной строки браузера по умолчанию и вставьте его в адресную строку другого браузера или выполните следующие действия:

1. В окне **Solution Explorer** щелкните правой кнопкой мыши на файле **WebTime.aspx** и выберите команду **Browse With...**, чтобы вызвать диалоговое окно **Browse With** (ил. 23.6).



**Ил. 23.6.** Выбор браузера для выполнения веб-приложения

2. В списке **Browsers** выберите браузер, в котором вам хотелось бы протестировать веб-приложение, и щелкните на кнопке **Browse**.

Если нужный браузер отсутствует в списке, используйте окно **Browse With** для добавления и удаления элементов в список браузеров.

## 23.4.1. Построение приложения WebTime

Давайте посмотрим, как создать такое приложение в Visual Studio Express For Web.

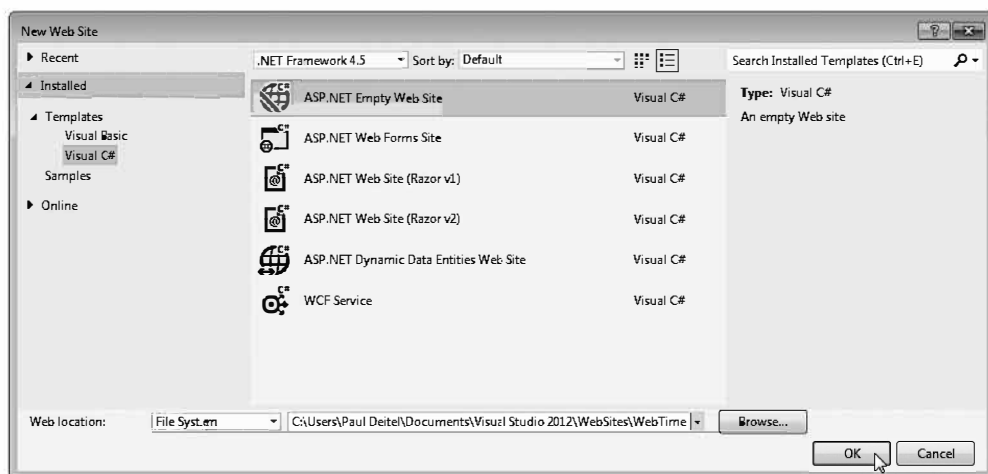
### Шаг 1. Создание проекта веб-сайта

Выполните команду **FILE ► New Web Site...**; на экране появится диалоговое окно **New Web Site** (ил. 23.7). Убедитесь в том, что в левом столбце окна выбрана категория **Visual C#**, после чего выберите шаблон **ASP.NET Empty Web Site** в среднем столбце. В нижней части диалогового окна можно выбрать местонахождение и имя веб-приложения.

Поле **Web location**: содержит следующие варианты:

- ☐ **File System**: создание нового сайта для тестирования на локальном компьютере. Такие сайты выполняются на локальной машине в IIS Express, а обращаться к ним можно только из браузеров, выполняемых на том же компьютере.

IIS Express — версия веб-сервера Microsoft IIS, предназначенная для локального тестирования веб-приложений. Позже вы можете опубликовать свой сайт на полном веб-сервере IIS для обращений через локальную сеть или Интернет. Выберите этот вариант, так как он будет использоваться во всех примерах этой главы.



**Ил. 23.7.** Создание проекта ASP.NET Web Site в Visual Studio Express For Web

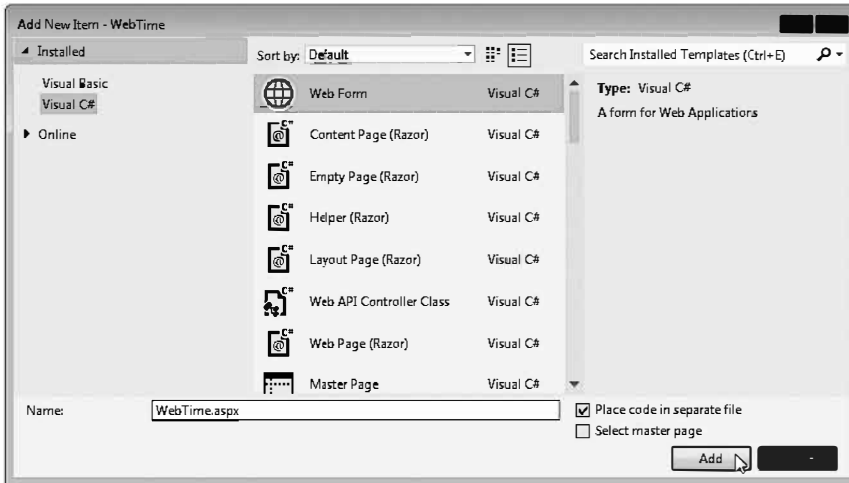
- ❑ **HTTP:** создание нового сайта на веб-сервере IIS и использование HTTP для размещения файлов сайта на сервере. IIS — программный продукт компании Microsoft, используемый для обеспечения работы сайтов в условиях реальной эксплуатации. Если вы являетесь владельцем сайта и хотите использовать собственный веб-сервер, возможно, вам стоит выбрать этот вариант для построения нового сайта прямо на компьютере сервера. Чтобы использовать этот вариант, необходимо обладать правами администратора на компьютере, на котором выполняется IIS.
- ❑ **FTP:** использование протокола *FTP* (File Transfer Protocol) для размещения файлов сайта на сервере. Администратор сервера сначала должен создать для вас сайт на сервере. Вариант с FTP обычно используется поставщиками услуг *хостинга*, чтобы владельцы сайтов могли совместно использовать серверный компьютер, обслуживающий много сайтов.

Измените имя веб-приложения `WebSite1` на `WebTime` и щелкните на кнопке **OK**, чтобы создать сайт.

## Шаг 2. Добавление веб-формы и окно Solution Explorer

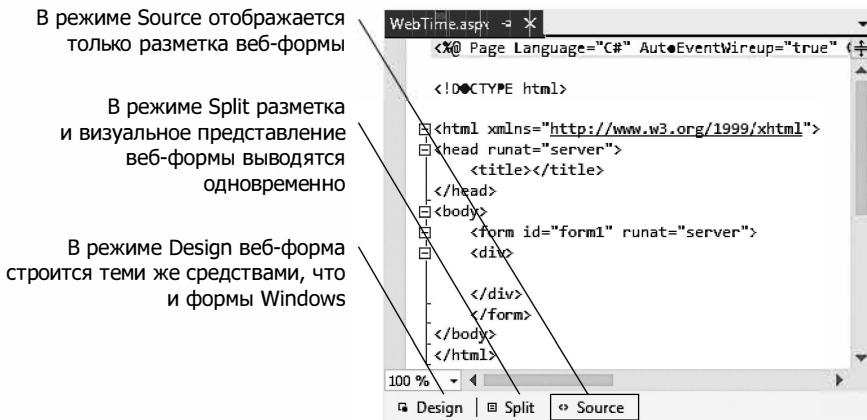
Веб-форма представляет одну страницу веб-приложения — мы будем часто использовать термины «страница» и «веб-форма» как синонимы. Веб-форма содержит графический интерфейс веб-приложения. Чтобы создать веб-форму `WebTime.aspx`, выполните следующие действия:

1. Щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer и выберите команду **Add ▸ Add New Item....** На экране появляется диалоговое окно **Add New Item** (ил. 23.8).



**Ил. 23.8.** Добавление новой веб-формы в диалоговом окне Add New Item

2. Проследите за тем, чтобы в левом столбце была выбрана категория **Visual C#**, и выберите в среднем столбце шаблон **Web Form**.
3. В текстовом поле **Name:** введите имя файла **WebTime.aspx** и щелкните на кнопке **Add Button**.



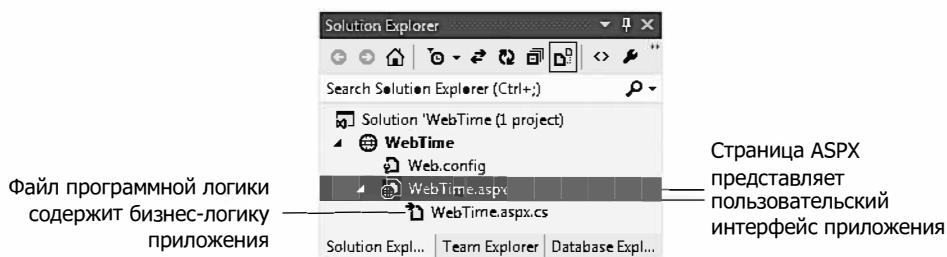
**Ил. 23.9.** Веб-форма в режиме просмотра исходного кода

После добавления веб-формы IDE по умолчанию открывает ее в режиме просмотра исходного кода (ил. 23.9), в котором отображается разметка формы. Когда вы

лучше освоите ASP.NET и построение веб-сайтов вообще, вы сможете использовать режим исходного кода для выполнения точной настройки разметки (HTML и/или CSS) или программирования на языке JavaScript для браузеров. В этой главе для простоты мы будем работать исключительно в режиме конструктора. Чтобы переключиться в режим конструктора, щелкните на кнопке **Design** в нижней части окна редактора кода.

### Окно Solution Explorer

В окне **Solution Explorer** (ил. 23.10) показано содержимое сайта. Мы развернули узел **WebTime.aspx**, чтобы показать файл программной логики **WebTime.aspx.cs**. В **Visual Studio Express For Web** окно **Solution Explorer** содержит кнопку **Nest Related Files**, которая упорядочивает все веб-формы и их файлы программной логики.



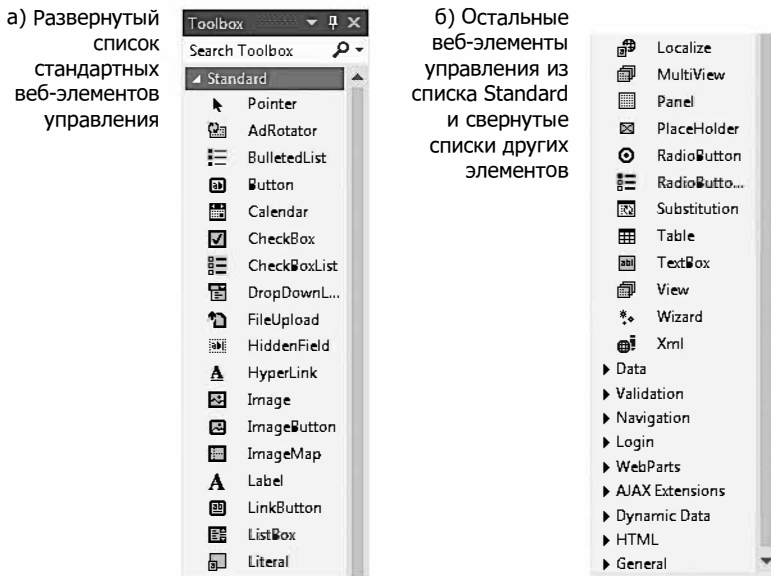
**Ил. 23.10.** Окно **Solution Explorer** для проекта **Empty Web Site** после добавления веб-формы **WebTime.aspx**

Если файл **ASPX** не открыт в IDE, откройте его в режиме конструктора, сделав двойной щелчок на нем в окне **Solution Explorer** и выбрав вкладку **Design**, или же щелкните на нем правой кнопкой мыши и выберите команду **View Designer**. Чтобы открыть файл программной логики в редакторе кода, сделайте на нем двойной щелчок в окне **Solution Explorer** или:

- ☐ выберите файл **ASPX** в окне **Solution Explorer** и щелкните на кнопке **View Code (< >)**;
- ☐ щелкните правой кнопкой мыши на файле **ASPX** в окне **Solution Explorer** и выберите команду **View Code**;
- ☐ щелкните правой кнопкой мыши на файле программной логики в окне **Solution Explorer** и выберите команду **Open**.

### Панель элементов

На ил. 23.11 изображена панель элементов, отображаемая в IDE при загрузке проекта. В части *а* изображено начало стандартного списка веб-элементов управления, а в части *б* — остальные веб-элементы и список других групп элементов управления. Имена многих элементов управления идентичны или похожи на имена элементов управления **Windows Forms**, представленных ранее в книге.



**Ил. 23.11.** Панель элементов в Visual Studio Express For Web

### Конструктор веб-форм

На ил. 23.12 изображена исходная веб-форма в режиме конструктора. Вы можете перетаскивать на нее элементы управления с панели элементов, а также вводить *статический текст* в текущей позиции курсора. В ответ на такие действия IDE генерирует соответствующую разметку в файле ASPX.



**Ил. 23.12.** Режим Design конструктора веб-форм

### Шаг 3. Изменение заголовка страницы

Прежде чем приступить к содержимому веб-формы, мы заменим ее заголовок текстом Simple Web Form Example. Обычно этот текст выводится в строке заголовка или на вкладке браузера, в которой выводится страница (см. ил. 23.4). Также он используется такими поисковыми системами, как Google и Bing, в процессе индексирования

веб-сайтов для поиска. Каждая страница должна иметь заголовок. Чтобы изменить текст заголовка, выполните следующие действия:

1. Убедитесь в том, что файл ASPX открыт в режиме Design.
2. В раскрывающемся списке окна свойств просмотрите свойства веб-формы; для этого выберите пункт **DOCUMENT**, представляющий веб-форму (веб-страница часто называется *документом*).
3. Измените свойство Title страницы, задав ему значение Simple Web Form Example.

### Построение страницы

Веб-страницы проектируются примерно так же, как и формы Windows. Чтобы добавить элемент управления на страницу, перетащите его с панели элементов на веб-форму в режиме конструктора. Сама веб-форма и элемент управления, который вы на нее добавляете, представлены объектами, содержащими свойства, методы и события. Свойства можно задавать визуальнo в окне свойств, на программном уровне в файле программной логики или посредством прямого редактирования разметки в файле .aspx. Также текст можно вводить напрямую в позиции курсора.

Элементы управления и другие элементы размещаются на веб-форме последовательно в порядке их перетаскивания на форму. Курсор обозначает позицию вставки на странице. Если вы захотите разместить элемент управления между существующим текстом или другими элементами управления, «сбросьте» его в нужную позицию между существующими элементами страницы.

Также в режиме конструктора можно изменять порядок следования элементов управления, перетаскивая их мышью. Позиции элементов управления и других элементов задаются относительно левого верхнего угла веб-формы. Такой тип позиционирования называется *относительным*; перемещение элементов и изменение их размеров осуществляется с учетом размера окна браузера. Относительное позиционирование используется по умолчанию, и мы будем использовать его в этой главе.

Для точного определения позиции и размера элементов используется *абсолютное* позиционирование, при котором элементы размещаются в той позиции, в которой они были сброшены на форму. Если вы хотите использовать абсолютное позиционирование:

1. Выполните команду **TOOLS ▸ Options**. На экране появляется диалоговое окно Options.
2. Раскройте узел **HTML Designer**, выберите узел **CSS Styling** и установите флажок **Change positioning to absolute for controls added using Toolbox, paste or drag and drop**.

### Шаг 4. Добавление текста и надписи

Теперь мы добавим на веб-форму текст и надпись. Выполните следующие действия:

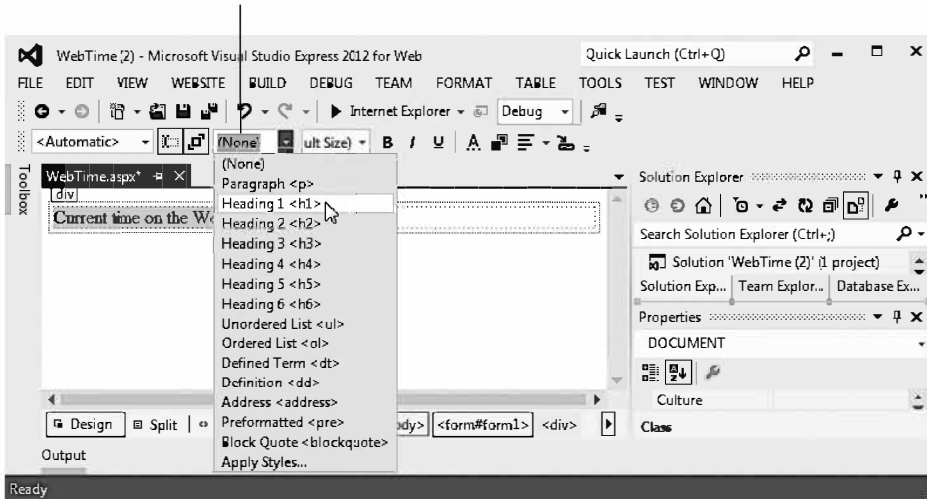
1. Убедитесь в том, что веб-форма открыта в режиме конструктора.
2. Введите следующий текст в текущей позиции курсора:

Current time on the Web server:



3. Выделите только что введенный текст, затем выберите строку **Heading 1** в поле **Block Format** на панели инструментов форматирования (ил. 23.13). Текст форматируется как заголовок первого уровня, что обычно означает применение более крупного жирного шрифта. На более сложных страницах заголовки помогают определить относительную важность частей контента (по аналогии с главами книги и разделами глав).

Поле Block Format панели инструментов форматирования

**Ил. 23.13.** Преобразование текста в заголовок первого уровня

4. Щелкните справа от только что введенного текста и нажмите **Enter**, чтобы начать новый абзац на странице. Примерный вид веб-формы показан на ил. 23.14.

**Ил. 23.14.** WebTime.aspx после вставки текста и нового абзаца

5. Перетащите надпись (**Label**) с панели элементов на новый абзац или сделайте двойной щелчок на соответствующем значке панели элементов, чтобы вставить надпись в текущей позиции курсора.
6. В окне свойств задайте свойству (**ID**) элемента управления **Label** значение **timeLabel**. Оно определяет имя переменной, которая будет использоваться для программного изменения текста надписи.

7. Так как свойство `Text` будет задаваться на программном уровне, удалите текущее значение свойства `Text` надписи. Когда надпись не содержит текста, в режиме конструктора отображается ее имя, заключенное в квадратные скобки (ил. 23.15). Во время выполнения этот текст не выводится.



Ил. 23.15. WebTime.aspx после добавления надписи

### Шаг 5. Форматирование надписи

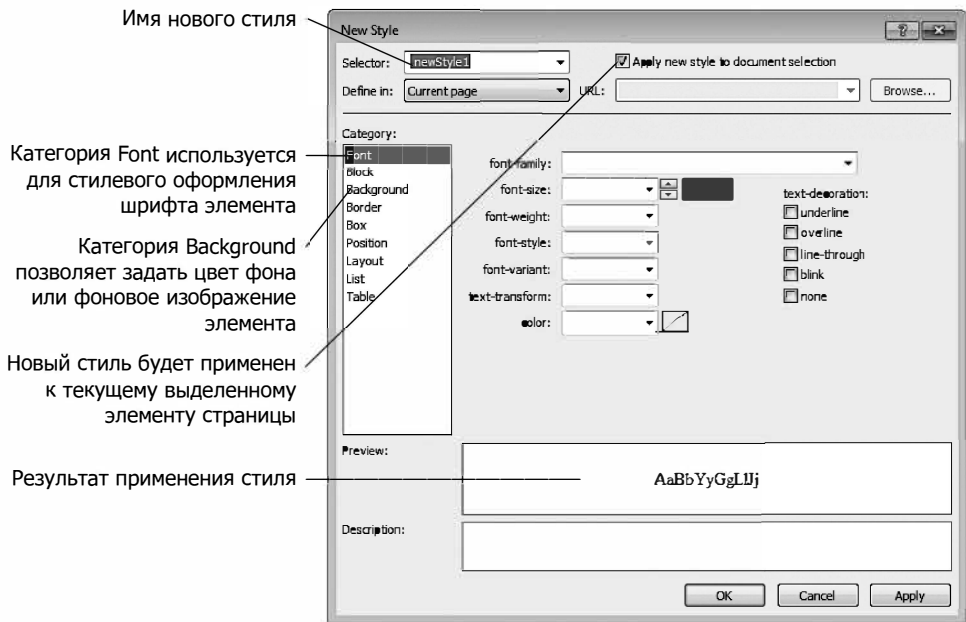
Форматирование веб-страниц осуществляется средствами *CSS* (Cascading Style Sheets). Подробное описание *CSS* выходит за рамки книги. Тем не менее инструмент Visual Studio Express For Web позволяет легко применять *CSS* для форматирования текста и элементов веб-форм. В нашем примере требуется назначить надписи черный цвет фона и желтый цвет текста, увеличив размер шрифта. Форматирование надписи выполняется следующим образом:

1. Выделите надпись, щелкнув на ней в режиме конструктора.
2. Выполните команду **VIEW** ► **CSS Properties**; в левой части IDE открывается окно свойств *CSS* (ил. 23.16).

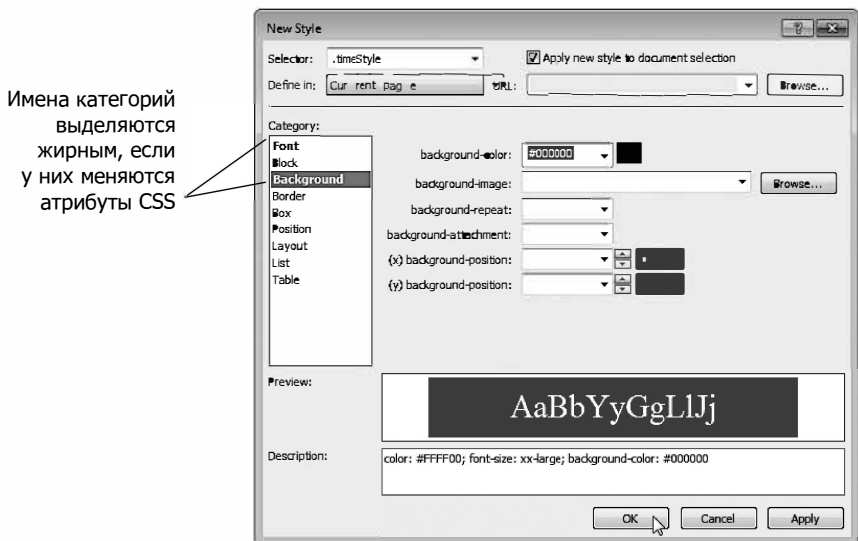


Ил. 23.16. Окно свойств *CSS*

3. Щелкните правой кнопкой мыши под строкой **Applied Rules** и выберите команду **New Style...**. На экране появляется диалоговое окно **New Style** (ил. 23.17).
4. Введите имя нового стиля в поле **Selector**: — мы выбрали имя `.timeStyle`, так как этот стиль будет использоваться для форматирования времени, выводимого на странице. Имена стилей, применяемых к конкретным элементам, должны начинаться с точки (`.`). Такие стили называются *классами CSS*.

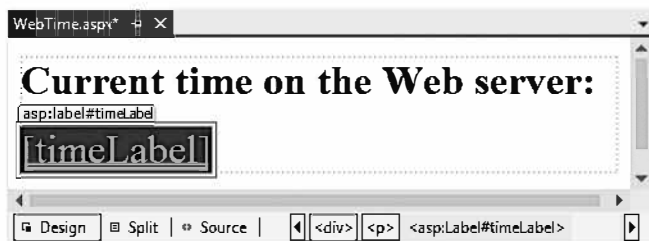
**Ил. 23.17.** Диалоговое окно New Style

5. Настройки, вносимые в диалоговом окне New Style, называются *атрибутами CSS*. Чтобы изменить цвет текста timeLabel, выберите категорию Font из списка category, а затем выберите желтый образец цвета для атрибута color.

**Ил. 23.18.** Диалоговое окно New Style после изменения стиля надписи

6. Задайте атрибуту `font-size` значение `xx-large`.
7. Чтобы изменить цвет фона `timeLabel`, выберите категорию `Background`, затем выберите черный образец цвета для атрибута `background-color`.

Диалоговое окно `New Style` должно выглядеть так, как показано на ил. 23.18. Щелкните на кнопке `ОК`, чтобы применить стиль к элементу `timeLabel`; надпись принимает вид, изображенный на ил. 23.19. Также обратите внимание на то, что свойству `CssClass` элемента управления `Label` в окне свойств задается значение `timeStyle`.



**Ил. 23.19.** Режим конструктора после изменения стиля надписи

### Шаг 6. Добавление логики страницы

Итак, графический интерфейс построен, и мы можем перейти к написанию кода в файле программной логики для получения времени на сервере и выводе его в элементе управления `Label`. Откройте файл `WebTime.aspx.cs` двойным щелчком в окне `Solution Explorer`. В этом примере мы добавим в файл программной логики обработчик события `Init` веб-формы, происходящего при запросе страницы браузером. Обработчик этого события `Page_Init` инициализирует страницу. В нашем примере инициализация сводится к заданию свойства `Text` элемента управления `timeLabel` текущего времени на компьютере с веб-сервером. Файл программной логики изначально содержит обработчик события `Page_Load`. Чтобы создать обработчик события `Page_Init`, просто отредактируйте имя `Page_Load`, замените его именем `Page_Init` и вставьте в тело следующий код:

```
// Вывод текущего времени на сервере в timeLabel
timeLabel.Text = DateTime.Now.ToString("hh:mm:ss");
```

### Шаг 7. Назначение стартовой страницы и запуск программы

Чтобы страница `WebTime.aspx` загружалась при запуске приложения, щелкните на ней правой кнопкой мыши в окне `Solution Explorer` и выберите команду `Set As Start Page`. Теперь программу можно запустить одним из нескольких способов.

В начале раздела 23.4 вы узнали, что для просмотра веб-формы в браузере следует нажать клавиши `Ctrl+F5`. Также можно щелкнуть на файле `ASPX` в окне `Solution Explorer` и выбрать команду `View in Browser`. Оба способа запускают `IIS Express`, открывают браузер по умолчанию и загружают страницу в браузере, запуская таким образом веб-приложение. `IIS Express` автоматически останавливается при выходе из `Visual Studio Express For Web`.

Если во время запуска приложения возникают проблемы, вы можете запустить приложение в отладочном режиме командой **DEBUG ▶ Start Debugging**, кнопкой **Start Debugging Button ( ▶ )** или клавишей **F5**. Отладка веб-приложений возможна только в том случае, если она явно включена в файле **Web.config** приложения — файле, генерируемом при создании веб-приложения ASP.NET. В этом файле хранятся параметры конфигурации приложения. Необходимость в прямом редактировании **Web.config** встречается довольно редко. Когда вы впервые выполняете команду **DEBUG ▶ Start Debugging** в проекте, на экране появляется диалоговое окно с вопросом, должна ли среда разработки изменить файл **Web.config** для включения отладки. Если нажать кнопку **OK**, IDE выполняет приложение. Отладка останавливается командой **DEBUG ▶ Stop Debugging**.

Независимо от способа запуска веб-приложения IDE компилирует проект перед выполнением. Более того, ASP.NET компилирует веб-страницу при любом изменении между запросами HTTP. Допустим, вы просматриваете страницу, а затем изменяете файл **ASPX** или добавляете код в файл программной логики. При перезагрузке страницы ASP.NET перекомпилирует страницу на сервере, прежде чем возвращать ответ браузеру. Такое поведение гарантирует, что клиент всегда будет видеть последнюю версию страницы. Вы можете вручную откомпилировать весь сайт командой **Build Web Site** из меню **DEBUG Visual Studio Express For Web**.

### 23.4.2. Файл программной логики **WebTime.aspx**

На ил. 23.20 представлен файл программной логики **WebTime.aspx.cs**. В строке 5 начинается объявление класса **WebTime**. Объявление класса может распространяться на несколько файлов исходного кода — отдельные части объявления класса в разных файлах называются *частичными классами*. Модификатор **partial** означает, что файл программной логики является частью большего класса. Как и в приложениях **Windows Forms**, остальной код класса генерируется на основе визуальных взаимодействий по созданию графического интерфейса приложения в режиме конструктора. Компилятор собирает все частичные классы с одним именем в объявление одного класса.

```

1 // Ил. 23.20: WebTime.aspx.cs
2 // Файл программной логики страницы с временем на веб-сервере
3 using System;
4
5 public partial class WebTime : System.Web.UI.Page
6 {
7     // Инициализация содержимого страницы
8     protected void Page_Init( object sender, EventArgs e )
9     {
10         // Вывод текущего времени на сервере в timeLabel
11         timeLabel.Text = DateTime.Now.ToString( "hh:mm:ss" );
12     } // Конец метода Page_Init
13 } // Конец класса WebTime

```

**Ил. 23.20.** Файл программной логики для страницы, отображающей время на веб-сервере

В строке 5 указано, что класс `WebTime` является производным от класса `Page` из пространства имен `System.Web.UI`. Это пространство имен содержит классы и элементы управления для построения веб-приложений. Класс `Page` представляет стандартные возможности каждой страницы в веб-приложении — все страницы наследуют (прямо или косвенно) от этого класса.

Строки 8–12 определяют обработчик события `Page_Init`, инициализирующий страницу по событию `Init` страницы. Для нашей страницы вся инициализация сводится к заданию в свойстве `Text` элемента управления `timeLabel` текущего времени на компьютере веб-сервера. Команда в строке 11 получает текущее время (`DateTime.Now`) и преобразует его в формат *чч:мм:сс* (например, 09:00:00 или 02:30:00). Как будет показано ниже, переменная `timeLabel` представляет элемент управления ASP.NET `Label`. Элементы ASP.NET определяются в пространстве имен `System.Web.UI.WebControls`.

## 23.5. Стандартные веб-элементы управления

В этом разделе представлены некоторые веб-элементы управления из стандартной части панели элементов (см. ил. 23.11). На ил. 23.21 приведена сводка элементов управления, использованных в следующем примере.

Веб-элемент управления	Описание
<code>TextBox</code>	Используется для ввода и вывода текста
<code>Button</code>	Иницирует событие при щелчке
<code>HyperLink</code>	Отображает гиперссылку
<code>DropDownList</code>	Выводит раскрывающийся список, из которого пользователь может выбрать нужный вариант
<code>RadioButtonList</code>	Группа переключателей
<code>Image</code>	Выводит графику (например, PNG, GIF или JPG)

**Ил. 23.21.** Часто используемые веб-элементы управления

### Форма для ввода данных

На ил. 23.22 изображена форма для ввода данных пользователем. Никаких операций эта программа не выполняет — то есть при щелчке на кнопке `Register` никакие действия не выполняются. Здесь мы сосредоточимся на добавлении этих элементов управления на веб-форму и задании их свойств. В дальнейших примерах будет показано, как организуется обработка событий многих из этих элементов управления. Чтобы запустить приложение, выполните следующие действия:

1. Выполните команду `Open Web Site...` из меню `FILE`.
2. В диалоговом окне `Open Web Site` убедитесь в том, что выбрана вкладка `File System`, перейдите к примерам этой главы, выберите папку `WebControls` и щелкните на кнопке `Open`.

3. Выделите файл `WebControls.aspx` в окне Solution Explorer и нажмите `Ctrl+F5`, чтобы выполнить веб-приложение в браузере по умолчанию.

### Шаг 1. Создание сайта

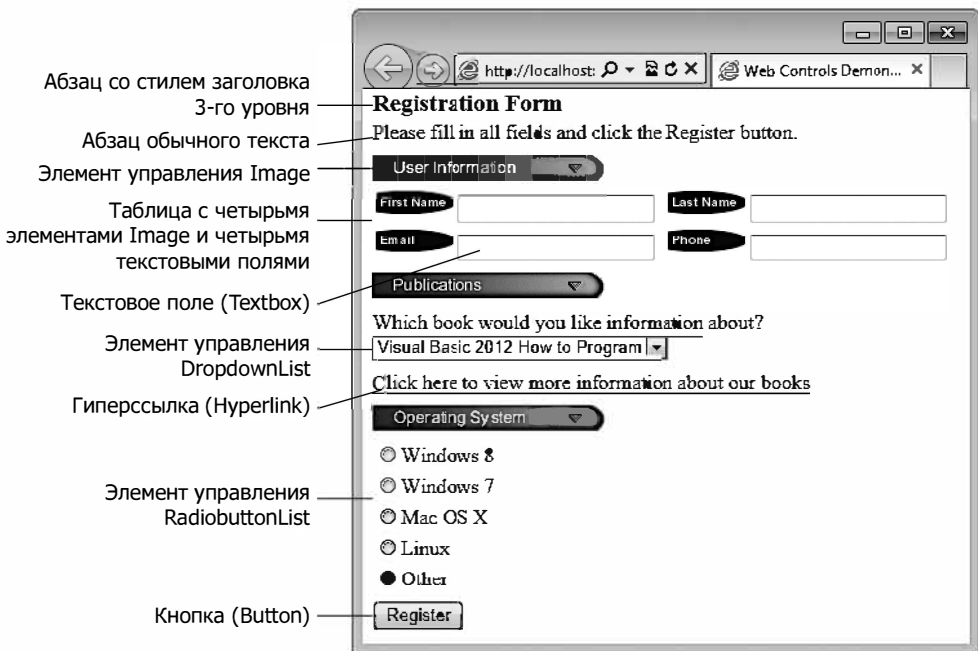
Выполните действия из раздела 23.4.1 для создания пустого сайта (Empty Web Site) с именем `WebControls`, затем добавьте в проект веб-форму с именем `WebControls.aspx`. Задайте свойству `Title` документа значение `"Web Controls Demonstration"`. Щелкните правой кнопкой мыши на файле `WebControls.aspx` в окне Solution Explorer и выберите команду `Set As Start Page`, чтобы страница загружалась при запуске приложения.

### Шаг 2. Добавление графики в проект

Графика, использованная в этом примере, находится в папке `images` примеров этой главы. Чтобы графика приложения могла отображаться на веб-форме, ее сначала необходимо добавить в проект. Чтобы добавить папку `images` в проект, выполните следующие действия:

1. Откройте Проводник Windows.
2. Найдите и откройте папку примеров этой главы (`ch23`).
3. Перетащите папку `images` из Проводника Windows в окно Solution Explorer в Visual Studio Express For Web и «сбросьте» ее на имя вашего проекта.

IDE автоматически копирует папку и ее содержимое в проект.



**Ил. 23.22.** Веб-форма, демонстрирующая использование веб-элементов управления

### Шаг 3. Добавление текста и графики на форму

На следующем шаге начинается создание страницы. Выполните следующие действия:

1. Создайте заголовок страницы. В текущей позиции курсора введите текст "RegistrationForm", затем используйте поле со списком Block Format на панели инструментов IDE и преобразуйте текст к формату Heading 3.
2. Нажмите клавишу Enter, чтобы начать новый абзац. Введите текст "Please fill in all fields and click the Register button.".
3. Нажмите клавишу Enter, чтобы начать новый абзац. Сделайте двойной щелчок на элементе управления Image на панели элементов. На веб-страницу вставляется изображение в текущей позиции курсора. Задайте свойству (ID) элемента управления Image значение userInfoImage. Свойство ImageUrl указывает, где находится выводимое изображение. В окне свойств щелкните на кнопке с многоточием свойства ImageUrl; на экране появляется диалоговое окно Select Image. Откройте папку images в категории Project folders: и выберите изображение user.png.
4. Щелкните на кнопке ОК, чтобы вывести изображение в режиме конструктора. Щелкните справа от изображения и нажмите Enter, чтобы начать новый абзац.

### Шаг 4. Добавление таблицы на форму

Элементы форм иногда размещаются в ячейках таблиц для создания упорядоченных макетов — как, например, элементы «имя», «фамилия», «электронная почта» и «телефон» на ил. 23.22. Создайте таблицу с двумя строками и двумя столбцами в режиме конструктора.

1. Выполните команду Table ► Insert Table; на экране появляется диалоговое окно Insert Table (ил. 23.23). В этом окне задаются основные параметры таблицы.
2. Убедитесь в том, что в группе Size значения в полях Rows и Columns равны 2 — эти значения используются по умолчанию.
3. Щелкните на кнопке ОК, чтобы закрыть диалоговое окно Insert Table и создать таблицу.

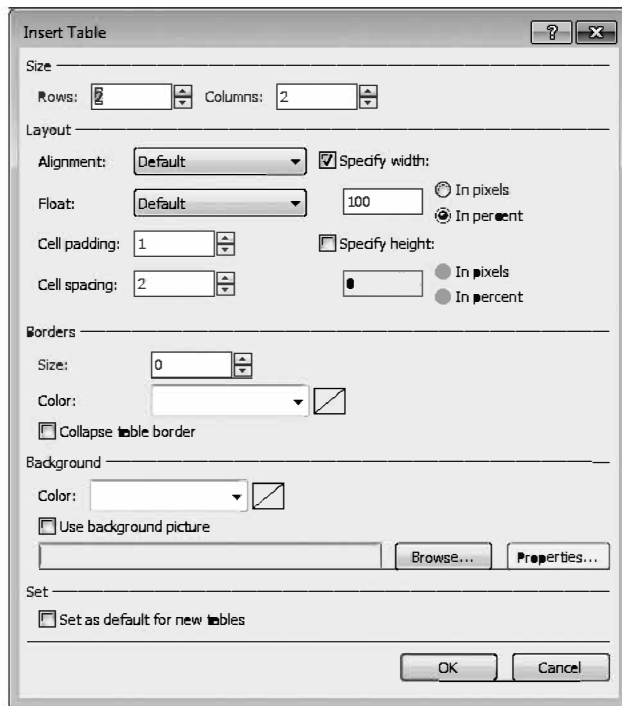
По умолчанию содержимое ячеек таблиц выравнивается по вертикали в середине ячейки.

После создания таблицы в ее ячейках можно размещать элементы управления и текст для создания аккуратных макетов. Добавьте в каждую ячейку таблицы элементы управления Image и TextBox:

1. Щелкните на ячейке таблицы в первой строке и первом столбце, затем сделайте двойной щелчок на элементе управления Image на панели элементов. Задайте его свойству (ID) значение firstNameImage, а свойству ImageUrl — значение fname.png.
2. Сделайте двойной щелчок на элементе управления TextBox на панели элементов. Задайте его свойству (ID) значение firstNameTextBox. Как и в приложениях



Windows Forms, элемент управления `TextBox` предназначен для ввода и вывода текста.



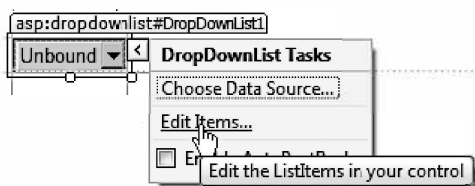
**Ил. 23.23.** Диалоговое окно Insert Table

3. Повторите процесс для первой строки и второго столбца, но задайте свойству (ID) элемента управления `Image` значение `lastNameImage`, а свойству `ImageUrl` — значение `lname.png`. Задайте свойству (ID) элемента управления `TextBox` значение `lastNameTextBox`.
4. Повторите пп. 1 и 2 для второй строки и первого столбца. Задайте свойству (ID) элемента управления `Image` значение `emailImage`, а свойству `ImageUrl` — значение `email.png`. Задайте свойству (ID) элемента управления `TextBox` значение `emailTextBox`.
5. Повторите пп. 1 и 2 для второй строки и второго столбца. Задайте свойству (ID) элемента управления `Image` значение `phoneImage`, а свойству `ImageUrl` — значение `phone.png`. Задайте свойству (ID) элемента управления `TextBox` значение `phoneTextBox`.

### Шаг 5. Создание раздела публикаций на странице

В этом разделе содержится изображение `Image`, блок текста, элемент управления `DropDownList` и элемент управления `HyperLink`. Выполните следующие действия:

1. Щелкните под таблицей и используйте инструменты, уже описанные в этом разделе, для добавления элемента управления Image с именем publicationsImage, в котором выводится изображение publications.png.
2. Щелкните справа от элемента Image, нажмите Enter и введите в следующем абзаце текст "Which book would you like information about?".
3. Удерживая Shift, нажмите клавишу Enter, чтобы создать новую строку в текущем абзаце. Сделайте двойной щелчок на элементе управления DropDownList на панели элементов. Задайте его свойству (ID) значение booksDropDownList. Этот элемент управления похож на элемент Windows Forms ComboBox, но не позволяет пользователю вводить текст. Когда пользователь щелкает на раскрывающемся списке, тот раскрывается и предлагает пользователю перечень вариантов.
4. Для добавления вариантов в DropDownList можно воспользоваться окном ListItem Collection Editor; щелкните на кнопке с многоточием рядом со свойством Items элемента управления DropDownList в окне свойств или же воспользуйтесь меню смарт-тега DropDownList Tasks. Чтобы открыть это меню, щелкните на кнопке со стрелкой в правом верхнем углу элемента управления в режиме конструктора (ил. 23.24). Visual Studio Express 2012 for Web выводит меню смарт-тегов для многих элементов управления ASP.NET, чтобы упростить выполнение основных операций. Команда Edit Items... в меню DropDownList Tasks открывает окно ListItem Collection Editor, в котором можно добавить элементы ListItem в DropDownList. Добавьте варианты "Visual Basic 2012 How to Program", "Visual C# 2012 How to Program", "Java How to Program" и "C++ How to Program", четырежды щелкнув на кнопке Add. Выделите каждый вариант и задайте его свойству Text одно из четырех названий книг.



**Ил. 23.24.** Меню смарт-тега DropDownList Tasks

5. Щелкните справа от элемента управления DropDownList и нажмите Enter, чтобы начать новый абзац. Сделайте двойной щелчок на элементе управления HyperLink на панели элементов, чтобы добавить гиперссылку на веб-страницу. Задайте ее свойству (ID) значение booksHyperLink, а свойству Text — значение "Click here to view more information about ourbooks". Задайте свойству NavigateUrl значение <http://www.deitel.com>. Это свойство задает ресурс или веб-страницу, которые будут запрошены при щелчке на гиперссылке. Задайте свойству Target значение \_blank; это означает, что запрашиваемая веб-страница должна открыться в новой вкладке или окне браузера. По умолчанию элементы управления HyperLink открывают страницы в том же окне браузера.

### Шаг 6. Завершение страницы

Теперь мы создадим секцию страницы `Operating System` и кнопку `Register`. В этой секции содержится элемент управления `RadioButtonList` с группой переключателей, из которых пользователь может установить только один. Меню смарт-тега `RadioButtonList Tasks` содержит команду `Edit Items...`, которая открывает окно `ListItem Collection Editor` для создания вариантов в списке. Выполните следующие действия:

1. Щелкните справа от элемента управления `HyperLink` и нажмите клавишу `Enter`, чтобы создать новый абзац. Добавьте элемент управления `Image` с именем `osImage`, в котором будет выводиться файл `os.png`.
2. Щелкните справа от `Image` и нажмите `Enter`, чтобы создать новый абзац. Добавьте элемент управления `RadioButtonList`, задайте его свойству (`ID`) значение `osRadioButtonList`. Используйте окно `ListItem Collection Editor` для добавления вариантов, представленных на ил. 23.22.
3. Наконец, щелкните справа от `RadioButtonList` и нажмите клавишу `Enter`, чтобы создать новый абзац; добавьте `Button` — веб-элемент управления, представляющий кнопку, при щелчке на которой инициируется действие. Задайте свойству (`ID`) кнопки значение `registerButton`, а свойству `Text` — значение `Register`. Как упоминалось ранее, при нажатии кнопки `Register` в этом примере ничего не происходит.

Чтобы просмотреть веб-форму в браузере, запустите приложение (`Ctrl+F5`).

## 23.6. Проверка данных

В этом разделе представлена новая разновидность веб-элементов управления — *элементы проверки данных*; они проверяют правильность формата данных в другом веб-элементе управления. Например, элемент проверки данных может проверить, ввел ли пользователь информацию в обязательном поле или состоит ли почтовый индекс из шести цифр. Элементы проверки данных предоставляют механизм проверки ввода на стороне клиента и сервера. При отправке страницы клиенту элемент проверки данных преобразуется в код `JavaScript`, который выполняет проверку в браузере клиента. (`JavaScript` — язык сценариев, который расширяет функциональность веб-страниц и обычно выполняется на стороне клиента.) К сожалению, некоторые браузеры могут не поддерживать сценарии или же эта функциональность может быть отключена пользователем, поэтому всегда следует выполнять проверку данных на сервере.

Элементы проверки данных `ASP.NET` могут функционировать на стороне клиента, на стороне сервера или на обеих сторонах.

### Проверка данных на веб-формах

Веб-форма на ил. 23.25 предлагает пользователю ввести имя, адрес электронной почты и номер телефона. Сайт может использовать подобные формы для сбора информации о посетителях.

После того как пользователь ввел данные — но до того, как эти данные будут отправлены веб-серверу, — элементы проверки данных убеждаются в том, что в каждом поле введено значение, а адрес электронной почты и номер телефона заданы в допустимом формате. В нашем примере (555) 123-4567, 555-123-4567 и 123-4567 считаются допустимыми телефонными номерами. После отправки данных веб-сервер отвечает выводом сообщения, повторяющего отправленную информацию. Вероятно, реальное приложение сохранит данные в базе или в файле на сервере. Мы же просто возвращаем данные клиенту, чтобы показать, что они были получены сервером. Чтобы выполнить это приложение:

1. Выполните команду **Open Web Site...** из меню **FILE**.
2. В диалоговом окне **Open Web Site** убедитесь в том, что выбрана вкладка **File System**, перейдите к примерам этой главы, выберите папку **Validation** и щелкните на кнопке **Open**.
3. Выделите файл **Validation.aspx** в окне **Solution Explorer** и нажмите **Ctrl+F5**, чтобы выполнить веб-приложение в браузере по умолчанию.

В приведенном примере:

- ❑ На ил. 23.25, *а* изображена исходная веб-форма.
- ❑ На ил. 23.25, *б* показан результат отправки данных перед вводом данных в текстовых полях.
- ❑ На ил. 23.25, *в* показаны результаты после заполнения всех текстовых полей, но с недействительным адресом электронной почты и номером телефона.
- ❑ На ил. 23.25, *г* показаны результаты после ввода действительных значений во всех трех текстовых полях и отправки формы.

а) Исходная веб-форма



The screenshot shows a web browser window with the address bar displaying 'http://localhost:...' and the page title 'Demonstrating Valid...'. The main content area contains the text 'Please fill in all the fields in the following form:'. Below this text are three input fields: 'Name:', 'E-mail:', and 'Phone:'. The 'E-mail:' field contains the text 'email@domain.com' and the 'Phone:' field contains the text '(555) 555-1234'. At the bottom of the form is a 'Submit' button.

**Ил. 23.25.** Элементы проверки данных на веб-форме для ввода контактной информации (продолжение ↗)

б) Веб-форма после нажатия пользователем кнопки Submit с пустыми текстовыми полями; за каждым текстовым полем выводится сообщение об ошибке, предоставленное элементом проверки данных

Элементы управления  
RequiredFieldValidator

A screenshot of a web browser window showing a form titled "Please fill in all the fields in the following form:". The form has three input fields: "Name:", "E-mail:", and "Phone:". Each field has a corresponding error message below it: "Please enter your name", "Please enter your e-mail address", and "Please enter your phone number". A "Submit" button is at the bottom.

в) Веб-форма после ввода имени, недействительного адреса электронной почты и недействительного телефона в текстовых полях и последующего нажатия кнопки Submit; элементы проверки данных выводят сообщения об ошибках в адресе электронной почты и телефоне

Элементы управления  
RegularExpressionValidator

A screenshot of a web browser window showing the same form. The "Name:" field now contains "Bob White". The "E-mail:" field contains "bwhite@domain.com" and has an error message: "Please enter an e-mail address in a valid format". The "Phone:" field contains "55 1234" and has an error message: "Please enter a phone number in a valid format". A "Submit" button is at the bottom.

г) Веб-форма после ввода действительных значений во всех трех текстовых полях и нажатия кнопки Submit

Надпись отображается после того, как пользователь введет действительные значения в поля и отправит данные формы

A screenshot of a web browser window showing the form after successful submission. The "Name:" field contains "Bob White", "E-mail:" contains "bwhite@fakeemail.com", and "Phone:" contains "(555) 555-5555". Below the fields is a "Submit" button. At the bottom of the form, a thank-you message is displayed: "Thank you for your submission. We received the following information. Name: Bob White. E-mail: bwhite@fakeemail.com. Phone: (555) 555-5555".

**Ил. 23.25.** Элементы проверки данных на веб-форме для ввода контактной информации (окончание)

### Шаг 1. Создание веб-сайта

Создайте веб-приложение `Validation` по шаблону `Empty Web Site` (см. подраздел 23.4.1). Добавьте в проект веб-форму с именем `Validation.aspx`. Задайте свойству `Title` документа значение `"Demonstrating Validation Controls"`. Чтобы страница `Validation.aspx` первой загружалась при запуске приложения, щелкните на ней правой кнопкой мыши в окне `Solution Explorer` и выберите команду `Set As Start Page`.

### Шаг 2. Создание графического интерфейса

Чтобы создать страницу, выполните следующие действия:

1. Введите текст `"Please fill in all the fields in the following form:"`, затем используйте поле со списком `Block Format` на панели инструментов IDE и преобразуйте текст к формату `Heading 3`.
2. Вставьте таблицу из трех строк и двух столбцов. Элементы будут добавлены в таблицу чуть позже.
3. Щелкните под таблицей и добавьте кнопку. Задайте ее свойству (`ID`) значение `submitButton`, а свойству `Text` — значение `Submit`. По умолчанию элемент управления `Button` на веб-форме отправляет содержимое формы серверу для обработки. Выделите кнопку и воспользуйтесь полем `Block Format` на панели инструментов IDE, чтобы заключить кнопку в формат `Paragraph`, — тем самым вы добавите дополнительное свободное пространство выше и ниже кнопки.
4. Щелкните справа от кнопки и нажмите клавишу `Enter`, чтобы начать новый абзац. Добавьте надпись (`Label`), задайте ее свойству (`ID`) значение `outputLabel` и удалите текущее содержимое свойства `Text` — оно будет задаваться на программном уровне при щелчке на кнопке `submitButton`. Задайте свойству `Visible` элемента управления `outputLabel`, чтобы надпись не появлялась в клиентском браузере при исходной загрузке страницы (она будет включаться в программном коде при отправке пользователем действительных данных).

На следующем шаге мы добавим текст и элементы управления в таблицу, созданную на шаге 2. Выполните следующие действия:

1. В левом столбце введите текст `"Name:"` в первой строке, `"E-mail:"` во второй строке и `"Phone:"` в третьей строке.
2. В правом столбце первой строки добавьте текстовое поле (`TextBox`) и задайте его свойству (`ID`) значение `nameTextBox`.
3. В правом столбце второй строки добавьте текстовое поле и задайте его свойству (`ID`) значение `emailTextBox`. Задайте его свойству `TextMode` значение `Email` — при отображении такого поля в клиентском браузере отображается поле HTML5, предназначенное для ввода адресов электронной почты. Щелкните справа от текстового поля и введите текст `"email@domain.com"` (пример текста, вводимого в текстовом поле).

4. В правом столбце третьей строки добавьте текстовое поле и задайте его свойству (ID) значение `phoneTextBox`. Задайте его свойству `TextMode` значение `Phone` — при отображении такого поля в клиентском браузере отображается поле HTML5, предназначенное для ввода телефонов. Щелкните справа от текстового поля и введите текст "(555) 555-1234" (пример текста, вводимого в текстовом поле).

### Шаг 3. Использование элементов управления `RequiredFieldValidator`

Мы используем три элемента управления `RequiredFieldValidator` (из секции `Validation` панели элементов) для проверки того, что текстовые поля имени, адреса электронной почты и телефона не были пусты на момент отправки формы. Элемент `RequiredFieldValidator` делает поле ввода обязательным для заполнения. Если такое поле остается пустым, проверка данных завершается неудачей. Добавьте элемент управления `RequiredFieldValidator` следующим образом:

1. Щелкните справа от поля `nameTextBox` в таблице и нажмите клавишу `Enter`, чтобы перейти к следующей строке.
2. Добавьте элемент управления `RequiredFieldValidator`, задайте его свойству (ID) значение `nameRequiredFieldValidator`, а свойству `ForeColor` — значение `Red`.
3. Задайте свойству `ControlToValidate` значение `nameTextBox`; это означает, что элемент проверки данных будет проверять содержимое `nameTextBox`.
4. Задайте свойству `ErrorMessage` значение "Please enter your name". Это сообщение будет выводиться на веб-форме только при неудачной проверке.
5. Задайте свойству `Display` значение `Dynamic`, чтобы элемент проверки данных занимал место на веб-форме только в случае неудачи при проверке. Когда это происходит, место на форме выделяется динамически, а все элементы, расположенные ниже, сдвигаются вниз для вывода сообщения об ошибке (см. ил. 23.25, а–в).

Повторите эти действия для добавления еще двух элементов `RequiredFieldValidator` во второй и третьей строках таблицы. Задайте их свойствам (ID) значение `emailRequiredFieldValidator` и `phoneRequiredFieldValidator` соответственно, а свойствам `ErrorMessage` — значения "Please enter your email address" и "Please enter your phone number" соответственно.

### Шаг 4. Использование элементов управления `RegularExpressionValidator`

В нашем примере также используются два элемента управления `RegularExpressionValidator`, которые проверяют формат введенного адреса электронной почты и телефона. Visual Studio Express 2012 for Web предоставляет несколько заранее определенных регулярных выражений, которые можно просто выбрать для использования этого мощного элемента проверки данных. Добавьте элементы `RegularExpressionValidator` следующим образом:

1. Щелкните справа от `emailRequiredFieldValidator` во второй строке таблицы и добавьте `RegularExpressionValidator`. Задайте его свойству (ID) значение `emailRegularExpressionValidator`, а свойству `ForeColor` — значение `Red`.
2. Задайте свойству `ControlToValidate` значение `emailTextBox`; это показывает, что элемент проверяет содержимое `emailTextBox`.
3. Задайте свойству `ErrorMessage` значение "Please enter an e-mail address in a valid format".
4. Задайте свойству `Display` значение `Dynamic`, чтобы элемент проверки данных занимал место на веб-форме только в случае неудачи при проверке.

Повторите описанные действия для добавления другого элемента `RegularExpressionValidator` в третьей строке таблицы. Задайте его свойству (ID) значение `phoneRegularExpressionValidator`, а свойству `ErrorMessage` — значение "Please enter a phone number in a valid format".

Свойство `ValidationExpression` элемента `RegularExpressionValidator` задает регулярное выражение, проверяющее содержимое `ControlToValidate`. Кнопка с многоточием рядом со свойством `ValidationExpression` в окне свойств вызывает диалоговое окно `Regular Expression Editor` со списком стандартных выражений для телефонных номеров, индексов и других отформатированных данных. Для элемента управления `emailRegularExpressionValidator` выбрано стандартное выражение `Internet e-mail address`. Если пользователь ввел в `emailTextBox` текст в неправильном формате, а пользователь щелкает в другом текстовом поле или пытается отправить данные формы, то сообщение `ErrorMessage` выводится красным шрифтом. Для `phoneRegularExpressionValidator` выбран шаблон `U.S. phone number` — он гарантирует, что телефонный номер содержит необязательный код города из трех цифр (либо в круглых скобках с необязательным последующим пробелом, либо без круглых скобок с обязательным последующим дефисом). После необязательного кода города следует номер телефона, состоящий из трех цифр, дефиса и еще четырех цифр. Например, каждая из комбинаций — (555) 123-4567, 555-123-4567 и 123-4567 — считается действительным номером телефона.

### Отправка содержимого веб-формы на сервер

Если все пять элементов проверки данных отработали успешно (то есть все текстовые поля заполнены, а адреса электронной почты и номер телефона действительны), щелчок на кнопке `Submit` отправляет данные формы на сервер. Как показано на ил. 23.25, з, сервер отвечает выводом отправленных данных в `outputLabel`.

### Файл программной логики веб-формы, получающей вводимые данные

На ил. 23.26 представлен файл программной логики приложения. Обратите внимание: этот файл не содержит кода реализации, относящегося к проверке данных (вскоре мы поговорим об этом более подробно). В данном примере мы реагируем на событие `Load` страницы для обработки данных, отправленных пользователем. Событие `Load`, как и `Init`, происходит каждый раз, когда страница загружается



в браузере, — различие заключается в том, что при обратной передаче (postback) вы не сможете обратиться к отправленным данным в элементах управления из обработчика Init. Обработчик этого события называется Page\_Load (строки 8–33) и создается автоматически при добавлении новой веб-формы. Добавьте в него код, приведенный на ил. 23.26.

```

1 // Ил. 23.26: Validation.aspx.cs
2 // Файл программной логики для формы с элементами проверки данных.
3 using System;
4
5 public partial class Validation : System.Web.UI.Page
6 {
7     // Обработчик события Page_Load выполняется при загрузке страницы
8     protected void Page_Load( object sender, EventArgs e )
9     {
10         // Блокировка ненавязчивой проверки
11         UnobtrusiveValidationMode =
12             System.Web.UI.UnobtrusiveValidationMode.None;
13
14         // Если страница загружается не в первый раз
15         // (то есть если пользователь уже отправил данные формы)
16         if ( IsPostBack )
17         {
18             Validate(); // Проверка формы
19
20             // Если форма действительна
21             if ( IsValid )
22             {
23                 // Получение данных, отправленных пользователем
24                 string name = nameTextBox.Text;
25                 string email = emailTextBox.Text;
26                 string phone = phoneTextBox.Text;
27
28                 // Вывод отправленных значений
29                 outputLabel.Text = "Thank you for your submission<br/>" +
30                     "We received the following information:<br/>";
31                 outputLabel.Text +=
32                     String.Format( "Name: {0}{1}E-mail:{2}{1}Phone:{3}",
33                         name, "<br/>", email, phone);
34                 outputLabel.Visible = true; // Вывод сообщения
35             } // Конец if
36         } // Конец if
37     } // Конец метода Page_Load
38 } // Конец класса Validation

```

**Ил. 23.26.** Файл программной логики для формы,  
демонстрирующей элементы проверки данных

## Ненавязчивая проверка ASP.NET 4.5

До выхода ASP.NET 4.5 при использовании элементов проверки данных, представленных в этом разделе, в код JavaScript веб-страницы внедрялся значительный объем кода JavaScript, обеспечивающего работу элементов проверки данных в браузере. В ASP.NET 4.5 используется *ненавязчивая* (unobtrusive) проверка

данных, существенно снижающая объем внедряемого в страницу кода JavaScript, а это означает повышение быстродействия за счет ускорения загрузки. При создании сайта на базе веб-форм ASP.NET все необходимое для ненавязчивой проверки обычно настраивается автоматически — если только вы не используете шаблон ASP.NET Empty Web Site, как мы делали в примерах этой главы. Чтобы приложение корректно работало в браузере, в строках 11–12 ненавязчивая проверка отключается.

### Первый запрос страницы и обратная передача

Веб-программисты, использующие ASP.NET, часто проектируют свои веб-страницы так, чтобы текущая страница перезагружалась при отправке данных формы пользователем; это позволяет программе получить входные данные, обработать их так, как нужно, и вывести результаты на той же странице при ее повторной загрузке. Обычно такие страницы содержат форму, которая при отправке передает значения всех элементов управления серверу и инициирует повторный запрос текущей страницы. Это событие называется *обратной передачей* (postback). Строка 16 использует свойство `IsPostBack` класса `Page` для определения, была ли страница загружена в результате обратной передачи. При первом запросе веб-страницы свойство `IsPostBack` равно `false`, а страница выводит только форму для пользовательского ввода. При обратной передаче (когда пользователь щелкает на кнопке `Submit`) свойство `IsPostBack` содержит `true`.

### Проверка веб-форм на стороне сервера

Проверка веб-форм на стороне сервера должна быть реализована на программном уровне. В строке 18 вызывается метод `Validate` текущего объекта `Page` для проверки информации запроса. При этом проверка осуществляется по правилам, заданным элементами проверки данных в веб-форме. В строке 21 по значению свойства `IsValid` класса `Page` определяется, успешно ли прошла проверка. Если свойство равно `true` (то есть проверка прошла успешно, а данные веб-формы действительны), выводится информация с веб-формы. В противном случае страница загружается без изменений (не считая того, что все элементы проверки данных, не прошедшие проверку, выводят свое сообщение об ошибке).

### Обработка данных, введенных пользователем

В строках 24–26 читаются значения `nameTextBox`, `emailTextBox` и `phoneTextBox`. При отправке данных веб-серверу веб-приложение получает доступ к введенным пользователем значениям через свойства веб-элементов управления. Затем в строках 29–33 свойству `Text` поля `outputLabel` задается сообщение с именем, адресом электронной почты и телефоном, отправленными на сервер. Обратите внимание: в строках 29, 30 и 33 для начала новой строки в `outputLabel` вместо `\n` используется последовательность `<br/>` — разрыв строки в веб-страницах. В строке 34 свойству `Visible` надписи `outputLabel` задается значение `true`, чтобы при перезагрузке страницы в браузере пользователь видел благодарственное сообщение и отправленные данные.

## 23.7. Отслеживание сеанса

Когда-то интернет-магазины и сетевой бизнес критиковали за то, что те не предоставляют клиенту индивидуальный сервис, типичный для «физических» магазинов. Для решения этой проблемы были созданы механизмы, позволяющие пользователю настраивать процесс просмотра и адаптировать контент под конкретного пользователя. Для этого данные о перемещениях клиента в Интернете объединяются с информацией, предоставленной самим клиентом (платежная информация, личные предпочтения, интересы, увлечения и т. д.).

### Персонализация

*Персонализация* позволяет бизнесу эффективно взаимодействовать со своими клиентами и упрощает поиск нужных товаров и услуг для пользователя. Если компания предоставляет контент, интересующий пользователя, то у нее с пользователем формируются отношения, которые развиваются со временем. Кроме того, если компания ориентируется на клиента и обращается к нему с персональными предложениями, рекомендациями, рекламой и услугами, это способствует развитию потребительской лояльности. Сайты порой используют хитроумные технологии настройки домашних страниц под индивидуальные потребности клиента. Сетевые магазины также часто сохраняют информацию о пользователях, чтобы приспособить оповещения и специальные предложения к их интересам. При таком сервисе пользователь начинает чаще посещать сайт и более регулярно делать покупки.

### Конфиденциальность

Сайт должен выдержать разумный баланс между персонализацией сервиса и защитой конфиденциальности. Некоторые потребители горячо поддерживают адаптированный контент, но другие боятся возможных нежелательных последствий, если предоставленная ими информация будет передана другим сторонам или собрана технологиями отслеживания. Сторонники конфиденциальности спрашивают: а что, если организация, которой мы предоставляем свою личную информацию, продаст или передаст ее другим организациям? А если мы не хотим, чтобы наши действия в Интернете (как предполагается, анонимной среде) будут отслеживаться или записываться без нашего ведома? А если посторонние получают доступ к данным личного характера — номерам кредитных карт или истории болезни? Все эти аспекты приходится учитывать самым разным людям — программистам, потребителям, коммерческим руководителям и законодателям.

### Распознавание клиентов

Чтобы предоставить персонализированный сервис клиентам, организация должна распознавать их при запросе информации с сайта. Как упоминалось ранее, механизм «запрос-ответ», лежащий в основе веб-технологий, работает на базе протокола HTTP. К сожалению, HTTP относится к категории *протоколов без сохранения состояния* — он не предоставляет информации, которая бы позволила веб-серверу

сохранять информацию состояния для конкретных клиентов. А это означает, что веб-сервер не может определить, поступил ли запрос от конкретного клиента или кем была сгенерирована серия запросов — одним или разными клиентами.

Для решения этой проблемы сайт предоставляет механизм идентификации конкретных клиентов. *Сеанс* представляет взаимодействие конкретного клиента с сайтом. Если клиент покидает сайт и возвращается позднее, он все равно распознается как тот же пользователь. Сеанс обычно завершается при закрытии браузера. Чтобы сервер мог различать клиентов, каждый клиент должен каким-то образом идентифицировать себя на сервере. Механизм отслеживания клиентов обычно называется *отслеживанием сеанса*. Популярные способы отслеживания сеансов основаны на использовании cookie (см. подраздел 23.7.1) и объекта ASP.NET `httpSessionState` (см. подраздел 23.7.2). Другие механизмы отслеживания сеанса выходят за рамки книги.

### 23.7.1. Cookie

Для персонализации веб-страниц часто используются *cookie* — фрагменты данных, сохраняемые браузером в небольшом текстовом файле на компьютере пользователя. Информация о клиенте сохраняется в cookie между сеансами. При первом посещении сайта пользователем компьютер пользователя получает cookie с сервера; в дальнейшем загруженный файл cookie активизируется при каждом повторном посещении сайта посетителем. Предполагается, что собранная информация будет использоваться для персонализации последующих посещений сайта пользователем. Например, у интернет-магазина в cookie могут храниться уникальные идентификаторы посетителей. Когда пользователь добавляет товар в корзину или выполняет другую операцию, которая сопровождается выдачей запроса к серверу, сервер получает cookie с уникальным идентификатором пользователя. Далее этот уникальный идентификатор используется для поиска корзины и выполнения всех необходимых действий.

Кроме идентификации пользователей, cookie также могут использоваться для обозначения покупательских предпочтений. Получая запрос от клиента, веб-форма может проанализировать cookie, отправленные клиенту во время предыдущих взаимодействий, выявить предпочтения пользователя и немедленно вывести список продуктов, представляющих интерес для клиента.

Любое взаимодействие клиента с сервером на базе HTTP включает заголовок с информацией либо о запросе (от клиента к серверу) или об ответе (при обращении от сервера к клиенту). Когда веб-форма получает запрос, в заголовок включается такая информация, как тип запроса и cookie, отправленные ранее с сервера для сохранения на клиентском компьютере. Когда сервер формулирует свой ответ, его заголовок содержит все cookie, которые сервер хочет сохранить на компьютере клиента, и другую информацию (например, тип данных в ответе).

*Срок действия* cookie определяет продолжительность хранения cookie на компьютере клиента. Если срок действия cookie не задан, то браузер сохраняет cookie на время сеанса просмотра. В противном случае cookie хранится до истечения срока действия, после чего удаляется.



### МНОГОПЛАТФОРМЕННОСТЬ 23.1

Пользователь может отключить поддержку cookie в своем браузере по соображениям конфиденциальности. У таких пользователей возникнут проблемы с веб-приложениями, использующими cookie для хранения информации состояния.

## 23.7.2. Отслеживание сеанса с использованием HttpSessionState

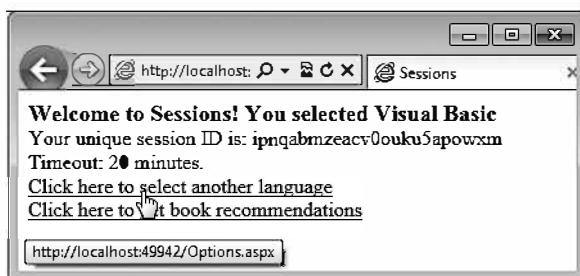
Следующее веб-приложение демонстрирует механизм отслеживания сеанса с использованием класса `HttpSessionState`. При выполнении этого приложения страница `Options.aspx` (ил. 23.27, а), являющаяся стартовой страницей приложения, позволяет выбрать язык программирования из группы переключателей. [*Примечание:* прежде чем запускать это приложение, щелкните правой кнопкой мыши на файле `Options.aspx` в окне `Solution Explorer` и выберите команду `Set As Start Page`.] Когда пользователь щелкает на кнопке `Submit`, выбор пользователя передается веб-серверу для обработки. Веб-сервер использует объект `HttpSessionState` для сохранения выбранного языка и номера ISBN одной из книг по этой теме. Каждому пользователю, посетившему сайт, назначается уникальный объект `HttpSessionState`, так что выбор одного пользователя хранится отдельно от выбора остальных пользователей. После сохранения полученных данных сервер возвращает страницу браузеру (ил. 23.27, б) и выводит выбор пользователя и информацию об уникальном сеансе пользователя (здесь она выводится исключительно для демонстрации). В страницу также включены ссылки, позволяющие выбрать другой язык программирования или просмотреть страницу `Recommendations.aspx` (ил. 23.27, д) со списком книг по выбранному ранее языку. Если пользователь щелкает на ссылке страницы с рекомендациями, информация, хранимая в уникальном объекте `HttpSessionState`, используется для построения списка рекомендаций. Чтобы протестировать это приложение, выполните следующие действия:

1. Выполните команду `Open Web Site...` из меню `FILE`.
2. В диалоговом окне `Open Web Site` убедитесь в том, что выбрана вкладка `File System`, перейдите к примерам этой главы, выберите папку `Sessions` и щелкните на кнопке `Open`.
3. Выделите файл `Options.aspx` в окне `Solution Explorer` и нажмите `Ctrl+F5`, чтобы выполнить веб-приложение в браузере по умолчанию.

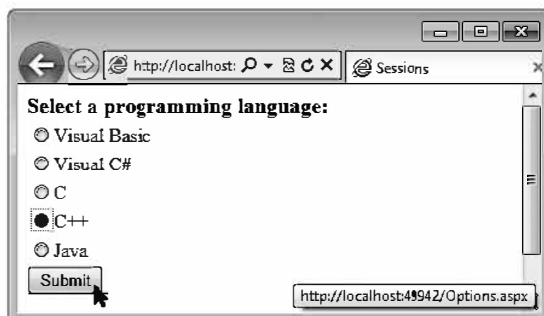
- а) Пользователь выбирает язык на странице Options.aspx и нажимает кнопку Submit для отправки выбора на сервер



- б) Страница Options.aspx обновляется: на ней скрываются переключатели выбора языка и выводится информация о выбранном языке. Пользователь щелкает на гиперссылке списка языков, чтобы выбрать другой вариант



- в) Пользователь выбирает другой язык на странице Options.aspx и нажимает кнопку Submit для отправки выбора на сервер

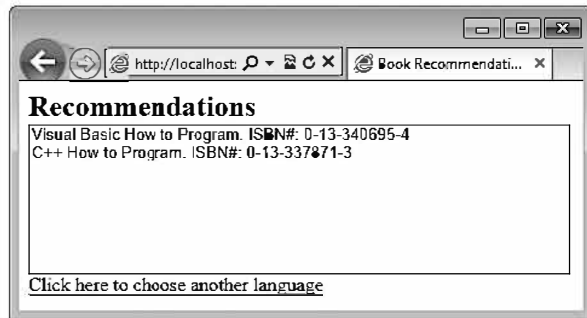


- г) Страница Options.aspx обновляется: на ней скрываются переключатели выбора языка и выводится информация о выбранном языке. Пользователь щелкает на гиперссылке для получения рекомендаций



**Ил. 23.27.** Файл ASPX выводит список языков программирования (продолжение ➞)

д) Страница  
Recommendations.aspx  
выводит список  
рекомендованных книг  
на основании выбора  
пользователя



**Ил. 23.27.** Файл ASPX выводит список языков программирования (окончание)

### Создание веб-сайта

Выполните действия из раздела 23.4.1 и создайте сайт по шаблону Empty Web Site с именем Sessions. Добавьте в проект две веб-формы с именами Options.aspx и Recommendations.aspx. Задайте свойству Title документа Options.aspx значение "Sessions", а свойству Title документа Recommendations.aspx — значение "Book Recommendations". Чтобы страница Options.aspx первой загружалась при запуске приложения, щелкните на ней правой кнопкой мыши в окне Solution Explorer и выберите команду Set As Start Page.

### 23.7.3. Options.aspx: выбор языка программирования

Страница Options.aspx (см. ил. 23.27, а) содержит следующие элементы управления, выровненные по вертикали:

1. Надпись (Label), у которой свойство (ID) содержит значение promptLabel, а свойство Text — значение "Select a programming language:". Мы создаем для этой надписи стиль CSS с именем .labelStyle (см. шаг 5 подраздела 23.4.1), задаем атрибуту font-size значение large, а атрибуту font-weight — значение bold.
2. Пользователь выбирает язык программирования при помощи переключателей в группе RadioButtonList. У каждого переключателя имеются свойства Text и Value. Содержимое свойства Text выводится рядом с переключателем, а свойство Value отправляется серверу, когда пользователь устанавливает переключатель и отправляет форму. В нашем примере свойство Value представляет номер ISBN рекомендуемой книги. Создайте элемент управления RadioButtonList со свойством (ID), равным languageList. Используйте окно ListItem Collection Editor для добавления пяти переключателей, у которых свойство Text содержит названия языков программирования (Visual Basic, Visual C#, C, C++ и Java), а свойство Value — номера ISBN (0-13-340695-4, 0-13-337933-7, 0-13-299044-X, 0-13-337871-3 и 0-13-294094-9).

3. Кнопка (Button), у которой свойство (ID) содержит значение `submitButton`, а свойство `Text` — значение `Submit`. В нашем примере будет обрабатываться событие `Click` этой кнопки. Обработчик события создается двойным щелчком на кнопке в режиме конструктора.
4. Надпись (Label), у которой свойство (ID) содержит значение `responseLabel`, а свойство `Text` — значение `"Welcome to Sessions!"`. Надпись должна размещаться непосредственно справа от кнопки, чтобы при сокрытии предыдущих элементов она отображалась в верхней части страницы. Используйте стиль CSS, созданный на пп. 1, задав свойству `CssClass` этой надписи значение `labelStyle`.
5. Еще две надписи (Label), у которых свойства (ID) равны `idLabel` и `timeoutLabel` соответственно. Сотрите содержимое свойства `Text` обоих элементов управления — это свойство будет заполняться на программном уровне информацией о сеансе текущего пользователя.
6. Гиперссылка (HyperLink), у которой свойство (ID) содержит значение `languageLink`, а свойство `Text` — значение `"Click here to choose another language"`. Задайте свойство `NavigateUrl`; для этого щелкните на кнопке с многоточием рядом со свойством в окне свойств и выберите файл `Options.aspx` в диалоговом окне `Select URL`.
7. Гиперссылка (HyperLink), у которой свойство (ID) содержит значение `recommendationsLink`, а свойство `Text` — значение `"Click here to get book recommendations"`. Задайте свойство `NavigateUrl`; для этого щелкните на кнопке с многоточием рядом со свойством в окне свойств и выберите файл `Recommendations.aspx` в диалоговом окне `Select URL`.
8. Изначально элементы управления из пп. 4–7 не отображаются, поэтому мы задаем свойству `Visible` каждого элемента управления значение `false`.

### Свойство Session страницы

Каждое веб-приложение ASP.NET включает объект `HttpSessionState`, доступный в свойстве `Session` объекта `Page`. В этом разделе мы будем использовать это свойство для операций с объектом `HttpSessionState` текущего пользователя. Когда пользователь впервые запрашивает страницу в веб-приложении, ASP.NET создает уникальный объект `HttpSessionState` и задает его свойству `Session` объекта `Page`. Этот объект доступен и для других страниц приложения.

### Файл программной логики страницы Options.aspx

На ил. 23.28 приведен файл программной логики страницы `Options.aspx`. При запросе страницы обработчик события `Page_Load` (строки 10–40) выполняется до отправки ответа клиенту. Так как первый запрос страницы не является обратной передачей, код в строках 16–38 *не выполняется* при первой загрузке страницы.



```

1 // Ил. 23.28: Options.aspx.cs
2 // Приложение обрабатывает выбор языка программирования,
3 // выводит ссылки и записывает информацию в объект HttpSessionState.
4 using System;
5
6 public partial class Options : System.Web.UI.Page
7 {
8     // При обратной передаче скрыть форму и вывести ссылки
9     // на страницы выбора языка программирования и вывода рекомендаций.
10    protected void Page_Load( object sender, EventArgs e )
11    {
12        if ( IsPostBack )
13        {
14            // Пользователь отправил данные; вывести сообщение
15            // и подходящие гиперссылки.
16            responseLabel.Visible = true;
17            idLabel.Visible = true;
18            timeoutLabel.Visible = true;
19            languageLink.Visible = true;
20            recommendationsLink.Visible = true;
21
22            // Скрыть другие элементы, используемые для выбора языка.
23            promptLabel.Visible = false;
24            languageList.Visible = false;
25            submitButton.Visible = false;
26
27            // Если пользователь выбрал язык, вывести его в responseLabel
28            if ( languageList.SelectedItem != null )
29                responseLabel.Text += " You selected " +
30                    languageList.SelectedItem.Text;
31            else
32                responseLabel.Text += " You did not select a language.";
33
34            // Вывод идентификатора сеанса
35            idLabel.Text = "Your unique session ID is: " + Session.SessionID;
36
37            // Вывод тайм-аута
38            timeoutLabel.Text = "Timeout: " + Session.Timeout + " minutes.";
39        } // Конец if
40    } // Конец метода Page_Load
41
42    // Сохранение выбора пользователя в Session
43    protected void submitButton_Click( object sender, EventArgs e )
44    {
45        // Если пользователь сделал выбор
46        if ( languageList.SelectedItem != null )
47            // Добавление пары "имя-значение" в Session
48            Session.Add( languageList.SelectedItem.Text,
49                languageList.SelectedItem.Value );
50    } // Конец метода submitButton_Click
51 } // Конец класса Options

```

**Ил. 23.28.** Обработка выбора языка программирования: приложение выводит список ссылок и записывает информацию в объект HttpSessionState

### Обработчик обратной передачи

При нажатии кнопки `Submit` происходит обратная передача. Форма отправляется на сервер, и выполняется обработчик `Page_Load`. В строках 16–20 выводятся элементы управления, показанные на ил. 23.27, б, а в строках 23–25 скрываются элементы, изображенные на ил. 23.27, а. Затем строки 28–32 проверяют, выбрал ли пользователь язык, и если выбрал — выводят в `responseLabel` сообщение с информацией о выборе. В противном случае выводится сообщение "You did not select a language".

Приложение ASP.NET содержит информацию об объекте `HttpSessionState` (свойство `Session` объекта `Page`) текущего клиента. Свойство `SessionID` объекта (выводимое в строке 35) содержит уникальный идентификатор сеанса — последовательность случайных букв и цифр. При первом подключении клиента к веб-серверу создается уникальный идентификатор сеанса, а сервер записывает на стороне клиента временный объект `cookie`, чтобы сервер мог идентифицировать клиента при последующих запросах. Когда клиент выдает дополнительные запросы, идентификатор сеанса из временного `cookie` сравнивается с идентификатором сеанса, хранимым в памяти веб-сервера для получения объекта `HttpSessionState` клиента. (Так как пользователи могут отключить `cookie` в браузере, также можно использовать сеансы без `cookie`; за дополнительной информацией обращайтесь по адресу [msdn.microsoft.com/en-us/library/aa479314.aspx](http://msdn.microsoft.com/en-us/library/aa479314.aspx).) Свойство `Timeout` объекта `HttpSessionState` (выводимое в строке 38) задает максимальный интервал времени, в течение которого объект `HttpSessionState` может оставаться неактивным до того, как он будет уничтожен. По умолчанию, если пользователь 20 минут не взаимодействует с веб-приложением, объект `HttpSessionState` уничтожается сервером, а при повторном взаимодействии пользователя с приложением будет создан новый объект. На ил. 23.29 перечислены некоторые часто используемые свойства `HttpSessionState`.

Свойства	Описание
<code>Count</code>	Задаёт количество пар «ключ-значение» в объекте <code>Session</code>
<code>IsNewSession</code>	Возвращает признак нового сеанса (то есть был ли сеанс создан во время загрузки данной страницы)
<code>Keys</code>	Возвращает коллекцию ключей объекта <code>Session</code>
<code>SessionID</code>	Возвращает уникальный идентификатор сеанса
<code>Timeout</code>	Задаёт максимальную продолжительность неактивности сеанса (то есть отсутствия запросов) в минутах, после которой срок действия сеанса истекает. По умолчанию промежуток неактивности составляет 20 минут

Ил. 23.29. Свойства `HttpSessionState`

### Метод `submitButton_Click`

Выбор пользователя сохраняется в объекте `HttpSessionState` при щелчке на кнопке `Submit`. Обработчик события `submitButton_Click` (строки 43–50) добавляет в объект `HttpSessionState` текущего пользователя пару «ключ-значение» с выбранным языком и номером ISBN книги по этому языку. Объект `HttpSessionState` представляет собой словарь — структуру данных для хранения пар «ключ-значение». Программа

использует ключ для сохранения и выборки связанного значения из словаря (см. главу 21).

Пары «ключ-значение» в объекте `HttpSessionState` часто называются «элементами сеанса». Они включаются в объект `HttpSessionState` вызовом метода `Add`. Если пользователь сделал выбор (строка 46), строки 48–49 получают выбранный язык и соответствующее значение из `languageList`, обращаясь к свойствам `Text` и `Value` объекта `SelectedItem` соответственно, после чего вызывают метод `Add` объекта `HttpSessionState` для добавления пары «имя-значение» в виде элемента сеанса в объект `HttpSessionState (Session)`.

Если приложение добавляет элемент сеанса с таким же именем, как у уже хранящегося в объекте `HttpSessionState`, элемент сеанса заменяется — имена элементов сеанса должны быть уникальными. Также для размещения элементов сеанса в объекте `HttpSessionState` часто применяется `Session[имя] = значение`. Например, строки 48–49 можно заменить фрагментом

```
Session[ languageList.SelectedItem.Text ] =  
    languageList.SelectedItem.Value
```



#### **АРХИТЕКТУРНОЕ РЕШЕНИЕ 23.1**

Веб-форма не должна использовать переменные экземпляров для хранения информации состояния клиента, потому что каждый запрос или обратная передача обрабатывается новым экземпляром страницы. Вместо этого информацию состояния клиента следует хранить в объектах `HttpSessionState`, потому что такие объекты создаются для каждого клиента.



#### **АРХИТЕКТУРНОЕ РЕШЕНИЕ 23.2**

Преимущество использования объекта `HttpSessionState` (вместо cookie) заключается в том, что в качестве значений атрибутов в них могут храниться объекты любого типа (а не только строки). Эта особенность расширяет возможности выбора типа информации состояния, хранимой для клиентов.

### **23.7.4. Recommendations.aspx: вывод рекомендаций на основании сеансовых значений**

После обратной передачи `Options.aspx` пользователь может запросить рекомендуемые книги по заданной теме. Гиперссылка отправляет пользователя на страницу `Recommendations.aspx` (см. ил. 23.27, д) с рекомендациями, подобранными на основании языка, выбранного пользователем.

Страница содержит следующие элементы управления, выровненные по вертикали:

1. Надпись (`Label`), у которой свойство (`ID`) содержит значение `recommendationsLabel`, а свойство `Text` — значение `"Recommendations"`. Мы создаем для этой надписи стиль CSS с именем `.labelStyle` (см. шаг 5 раздела 23.4.1), задаем атрибуту `font-size` значение `x-large`, а атрибуту `font-weight` — значение `bold`.

2. Список (`ListBox`), у которого свойство (`ID`) содержит значение `booksListBox`. Мы создали для этого элемента управления стиль CSS с именем `.listBoxStyle`. В категории `Position` атрибуту `width` задается значение `450px`, а атрибуту `height` — значение `125px` (суффикс «px» означает, что значение задается в пикселах).
3. Гиперссылка (`HyperLink`), у которой свойство (`ID`) содержит значение `languageLink`, а свойство `Text` — значение "Click here to choose another language". Задайте свойство `NavigateUrl`; для этого щелкните на кнопке с многоточием рядом со свойством в окне свойств и выберите файл `Options.aspx` в диалоговом окне `Select URL`. При щелчке на этой ссылке перезагружается страница `Options.aspx`. Такой запрос страницы не считается обратной передачей, поэтому будет выведена исходная форма на ил. 23.27, а.

### Файл программной логики страницы `Recommendations.aspx`

На ил. 23.30 представлен файл программной логики страницы `Recommendations.aspx`. Обработчик события `Page_Init` (строки 8–28) получает сеансовую информацию. Если пользователь не выбрал язык на странице `Options.aspx`, свойство `Count` объекта `HttpSessionState` равно 0 (строка 11). Это свойство содержит количество элементов сеанса, содержащихся в объекте `HttpSessionState`. Если значение `Count` равно 0, страница выводит текст `No Recommendations` (строка 22), скрывает элемент управления `ListBox` (строка 23) и обновляет свойство `Text` элемента управления `HyperLink` для возврата к `Options.aspx` (строка 26).

```

1 // Ил. 23.30: Recommendations.aspx.cs
2 // Создание списка рекомендаций на основании объекта Session.
3 using System;
4
5 public partial class Recommendations : System.Web.UI.Page
6 {
7     // Чтение элементов сеанса и заполнение списка рекомендаций
8     protected void Page_Init( object sender, EventArgs e )
9     {
10         // Проверка наличия информации в Session
11         if ( Session.Count != 0 )
12         {
13             // Вывод пар "имя-значение" в Session
14             foreach ( string keyName in Session.Keys )
15                 booksListBox.Items.Add( keyName +
16                     " How to Program. ISBN#: " + Session[ keyName ] );
17         } // Конец if
18         else
19         {
20             // Если элементы сеанса отсутствуют, язык не выбран, поэтому
21             // вывести соответствующее сообщение и скрыть booksListBox
22             recommendationsLabel.Text = "No Recommendations";
23             booksListBox.Visible = false;
24
25             // Изменить languageLink, потому что язык не выбран
26             languageLink.Text = "Click here to choose a language";
27         } // Конец else
28     } // Конец метода Page_Init
29 } // Конец класса Recommendations

```

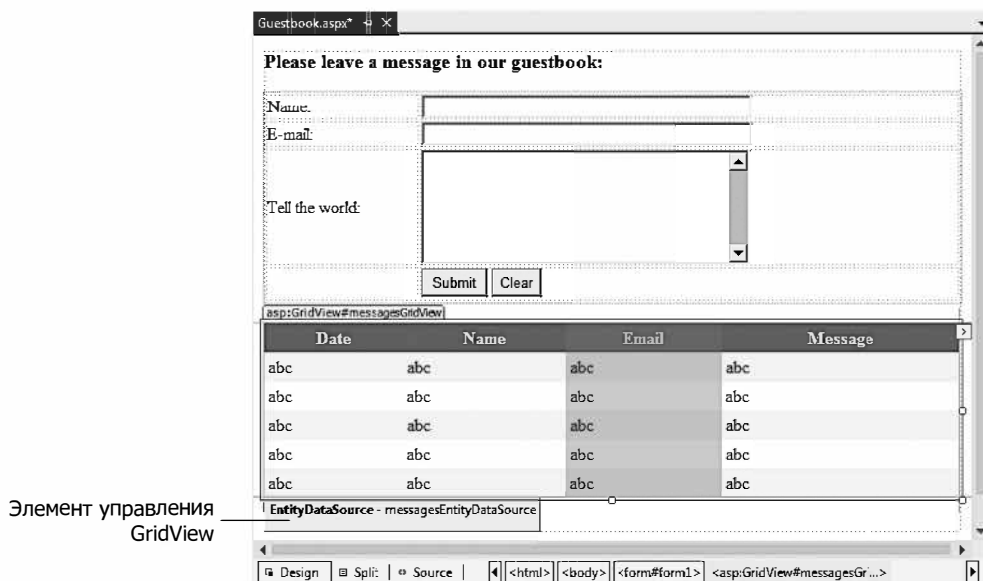
**Ил. 23.30.** Данные сеанса используются для формирования рекомендаций

Если пользователь выбрал по крайней мере один язык, цикл в строках 14–16 перебирает ключи объекта `HttpSessionState` (строка 14), обращаясь к свойству `Keys` объекта `HttpSessionState`, который возвращает коллекцию из всех ключей сеанса. В строках 15–16 объединяются `keyName`, строка "How to Program. ISBN#: " и значение, соответствующее ключу, полученное при помощи выражения `Session[keyName]`. В результате конкатенации образуется строка, добавляемая в список.

## 23.8. Пример: гостевая книга ASP.NET с использованием базы данных

Многие сайты позволяют своим посетителям оставить комментарии в гостевой книге. Как правило, для получения доступа к странице гостевой книги пользователь щелкает на ссылке, находящейся на домашней странице сайта. Обычно страница гостевой книги содержит форму с полями для ввода имени пользователя, адреса электронной почты, сообщения и т. д. Данные, введенные на форме гостевой книги, сохраняются в базе данных на сервере.

В этом разделе мы создадим приложение для веб-формы гостевой книги. Графический интерфейс (см. ил. 23.31) содержит элемент управления `GridView`, который отображает все записи книги в табличном формате. Этот элемент управления находится в разделе **Data** панели элементов. В режиме конструктора в таблице выводятся заполнители `abc`; они представляют значения, которые будут прочитаны из источника данных во время выполнения. Вскоре мы объясним, как выполняется создание и настройка `GridView` в программе.



Ил. 23.31. Интерфейс гостевой книги в режиме конструктора

### База данных Guestbook

Приложение хранит информацию гостевой книги в базе данных SQL Server с именем Guestbook.mdf на веб-сервере. (Она находится в папке **databases** примеров этой главы.) База данных состоит из одной таблицы с именем **Messages**.

а) Пользователь вводит данные (имя, адрес электронной почты, текст сообщения) и нажимает кнопку Submit, чтобы отправить данные серверу

Please leave a message in our guestbook:

Name:

E-mail:

Tell the world:

Date	Name	Email	Message
1/27/2013	Bob Green	bgreen@bug2bug.com	Great site!
1/28/2013	Sue Black	sblack@bug2bug.com	I love the site! Keep up the good work!
1/29/2013	Liz White	lwhite@bug2bug.com	Very useful information. Will visit again soon.

http://localhost:49530/Guestbook.aspx

б) Сервер сохраняет информацию в базе данных и обновляет GridView

Please leave a message in our guestbook:

Name:

E-mail:

Tell the world:

Date	Name	Email	Message
1/27/2013	Bob Green	bgreen@bug2bug.com	Great site!
1/28/2013	Sue Black	sblack@bug2bug.com	I love the site! Keep up the good work!
1/29/2013	Liz White	lwhite@bug2bug.com	Very useful information. Will visit again soon.
1/31/2013	Mike Brown	mbrown@bug2bug.com	Great use of ASP.NET!

Ил. 23.32. Пример выполнения приложения Guestbook

## Тестирование приложения

Чтобы протестировать это приложение, выполните следующие действия:

1. Выполните команду **Open Web Site...** из меню **FILE**.
2. В диалоговом окне **Open Web Site** убедитесь в том, что выбрана вкладка **File System**, перейдите к примерам этой главы, выберите папку **Guestbook** и щелкните на кнопке **Open**.
3. Выделите файл **Guestbook.aspx** в окне **Solution Explorer** и нажмите **Ctrl+F5**, чтобы выполнить веб-приложение.

На ил. 23.32, *а* изображен ввод новой записи пользователем, а на ил. 23.32, *б* новая запись выводится в последней строке **GridView**.

## 23.8.1. Построение веб-формы для вывода информации из базы данных

Сейчас мы займемся построением графического интерфейса и настройкой привязки данных между **GridView** и базой данных. Происходящее в целом напоминает то, что мы делали в главе 22 для организации работы с базой данных в приложении **Windows**. Файл фонового кода рассматривается в разделе 23.8.2.

### Шаг 1. Создание веб-сайта

Создайте веб-приложение **Guestbook** по шаблону **Empty Web Site** (см. раздел 23.4.1). Добавьте в проект веб-форму с именем **Guestbook.aspx**. Задайте свойству **Title** документа значение **"Guestbook"**. Чтобы страница **Guestbook.aspx** первой загружалась при запуске приложения, щелкните на ней правой кнопкой мыши в окне **Solution Explorer** и выберите команду **Set As Start Page**.

### Шаг 2. Создание формы для ввода данных

В режиме конструктора добавьте на страницу текст **Please leave a message in our guest book:**, затем используйте поле со списком **Block Format** на панели инструментов **IDE** и преобразуйте текст к формату **Heading 3**. Вставьте таблицу из четырех строк и двух столбцов. Поместите соответствующий текст (см. ил. 23.31) в верхние три ячейки левого столбца таблицы. Разместите текстовые поля с именами **nameTextBox**, **emailTextBox** и **messageTextBox** в верхних трех ячейках правого столбца. Настройте текстовые поля:

- ☐ Выберите команду **FORMAT ► New Style...**; на экране появляется диалоговое окно **New Style**. В поле **Selector** определите новый стиль с именем **.textBoxWidth**. Выберите категорию с именем **Position**, задайте в поле **width:** значение **300px** и щелкните на кнопке **OK**, чтобы создать стиль и закрыть диалоговое окно. Задайте свойству **CssClass** элементов управления **nameTextBox** и **emailTextBox** значение **textBoxWidth**. Тем самым вы назначаете обоим текстовым полям ширину 300 пикселей.

- ❑ Выберите команду **FORMAT ► New Style...**; на экране появляется диалоговое окно **New Style**. В поле **Selector** определите новый стиль с именем `.textBoxHeight`. Выберите категорию с именем **Position**, задайте в поле **height**: значение `100px` и щелкните на кнопке **OK**, чтобы создать стиль и закрыть диалоговое окно. Задайте свойству **CssClass** элемента управления `messageTextBox` значение

```
textBoxWidth textBoxHeight
```

В результате назначения стилей `.textBoxWidth` и `.textBoxHeight` поле `messageTextBox` имеет ширину 300 пикселей и высоту 100 пикселей. Также задайте свойству **TextMode** объекта `messageTextBox` значение `MultiLine`, чтобы пользователь мог ввести многострочное сообщение.

Наконец, разместите в правой нижней ячейке таблицы кнопки с именами `submitButton` и `clearButton`. Задайте свойствам **Text** кнопок значения `Submit` и `Clear` соответственно. Обработчики событий будут описаны при рассмотрении файла программной логики. Эти обработки событий создаются двойным щелчком на кнопке в режиме конструктора.

### Шаг 3. Добавление элемента управления GridView на веб-форму

Добавьте элемент управления `GridView` с именем `messagesGridView` для вывода записей гостевой книги. Этот элемент находится в разделе **Data** панели элементов. Для задания цвета `GridView` мы воспользуемся ссылкой **Auto Format...** меню смарт-тега, которое открывается при размещении `GridView` на странице. Щелчок на ссылке открывает диалоговое окно **AutoFormat** с несколькими вариантами. В нашем примере выбран вариант **Professional**. Вскоре мы покажем, как задать источник данных `GridView` (то есть источник, из которого элемент берет данные для вывода в строках и столбцах).

### Шаг 4. Создание модели данных сущностей

На следующем шаге в проект включается модель данных сущностей. Выполните следующие действия:

1. Щелкните правой кнопкой мыши на имени проекта в окне **Solution Explorer** и выберите команду **Add ► Add New Item....** На экране появляется диалоговое окно **Add New Item**.
2. Выберите шаблон **ADO.NET Entity Data Model**, измените имя на `GuestbookModel.edmx` и щелкните на кнопке **Add**. Открывается диалоговое окно с предложением поместить классы новой модели данных сущностей в папку `App_Code`; щелкните на кнопке **Yes**. IDE создает папку `App_Code` и помещает в нее классы модели данных сущностей. По соображениям безопасности к этой папке может обращаться только веб-приложение на сервере — клиент не сможет обратиться к этой папке по сети.
3. Открывается диалоговое окно **Entity Data Model Wizard**. Убедитесь в том, что выбран режим **Generate from database**, чтобы модель генерировалась по базе данных `Guestbook.mdf`, и щелкните на кнопке **Next>**.

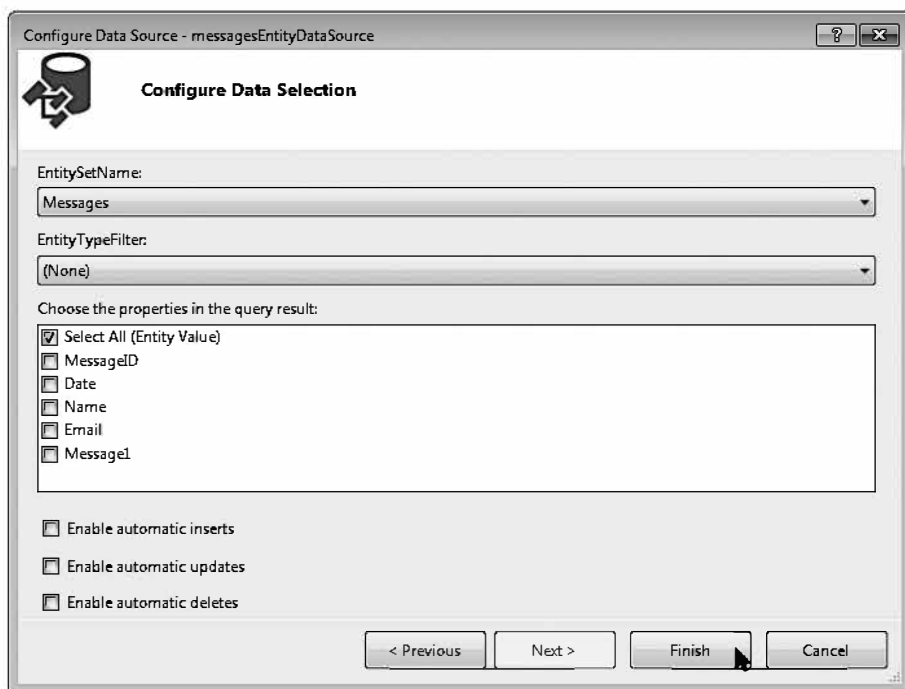


4. На шаге Choose Your Data Connection мастера Entity Data Model Wizard выберите команду New Connection... и выберите в диалоговом окне Connection Properties файл базы данных Guestbook.mdf (находится в папке databases примеров этой главы). Щелкните на кнопке OK, чтобы создать подключение, а затем завершите шаг Choose Your Data Connection кнопкой Next>.
5. Открывается диалоговое окно с предложением скопировать файл базы данных в проект. Щелкните на кнопке Yes. IDE создает папку App\_Data и помещает в нее файл Guestbook.mdf. Как и папка App\_Code, эта папка доступна только для веб-приложения на сервере.
6. На шаге Choose Your Database Objects and Settings мастера Entity Data Model Wizard выберите из базы данных таблицу Messages. По умолчанию IDE присваивает модели имя GuestbookModel1. Убедитесь в том, что флажок Pluralize or singularize generated object names установлен, оставьте другие настройки без изменений и щелкните на кнопке Finish. IDE отображает модель GuestbookModel1 в редакторе, где можно увидеть, что класс Message содержит свойства MessageID, Date, Name, Email и Message1. Свойство Message было переименовано в Message1 средой разработки для предотвращения конфликтов с классом Message модели данных сущностей.
7. Выполните команду BUILD ► Build Solution, чтобы откомпилировать классы модели данных сущностей.

### Шаг 5. Привязка элемента управления GridView к таблице Messages базы данных Guestbook

Теперь нужно настроить GridView для отображения информации из базы данных.

1. В меню смарт-тега GridView выберите вариант <New data source...> в поле Choose Data Source. На экране появляется окно мастера Data Source Configuration Wizard.
2. В этом примере мы используем элемент управления EntityDataSource для взаимодействия приложения с базой данных Guestbook.mdf. Выберите значок Entity, введите в поле Specify an ID for the data source строку messagesEntityDataSource и щелкните на кнопке OK, чтобы запустить мастер Configure Data Source.
3. На шаге ConfigureObjectContext выберите в поле Named Connection вариант GuestbookEntities и щелкните на кнопке Next>.
4. В окне Configure Data Selection (ил. 23.33) можно указать, какие данные объект EntityDataSource должен получить из контекста данных. В раскрывающемся списке EntitySetName перечислены свойства DbContext, представляющие таблицы базы данных. Для базы данных Guestbook выберите в списке вариант Messages. Проследите за тем, чтобы на панели Choose the properties in the query result: был установлен флажок Select All; он указывает, что из таблицы Messages выбираются все столбцы.




**Ил. 23.33.** Настройка EntityDataSource

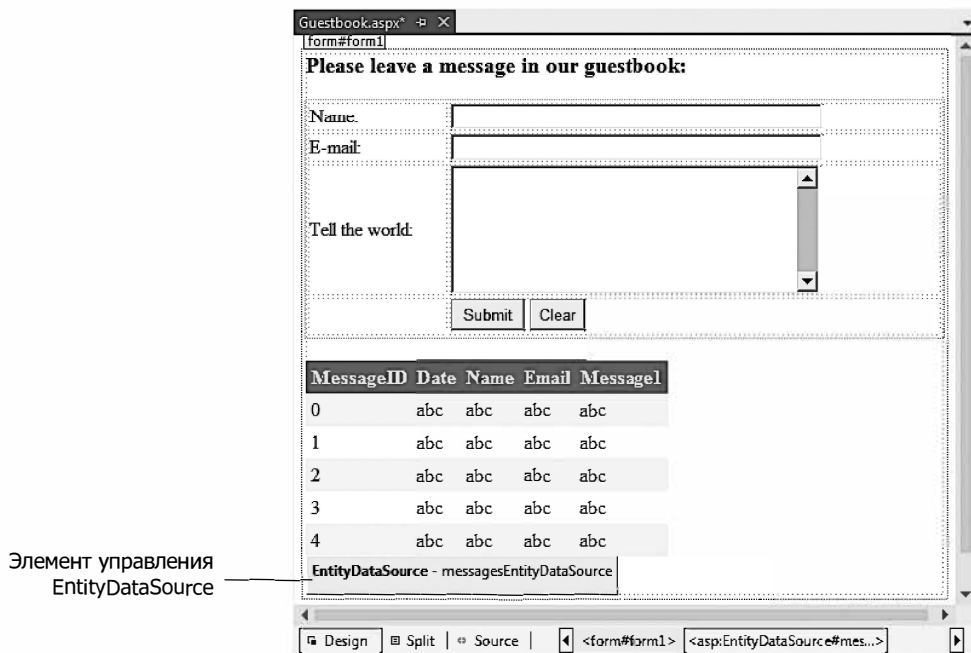
5. Щелкните на кнопке **Finish**, чтобы завершить работу мастера. На веб-форме прямо под **GridView** появляется элемент управления с именем **messagesEntityDataSource**. В режиме конструктора он представлен серым прямоугольником с указанием типа и значения. На веб-странице он не отображается — серый прямоугольник всего лишь позволяет визуальнo взаимодействовать с элементом управления в режиме конструктора (по аналогии с объектами в области компонентов в приложениях Windows Forms).
6. Щелкните на элементе управления **messagesEntityDataSource**. Выберите в меню смарт-тега команду **Refresh Schema**. Элемент управления **GridView** обновляется, и в нем отображаются заголовки столбцов, соответствующие столбцам таблицы **Messages** (ил. 23.34). Каждая строка содержит либо число (признак автоматически увеличиваемого столбца), либо **abc** (признак строковых данных). При просмотре файла **ASPX** в браузере в этих строках будут отображаться фактические данные из файла базы данных **Guestbook.mdf**.

### Шаг 6. Настройка столбцов источника данных, отображаемых в GridView

Выводить столбец **MessageID** при просмотре записей в гостевой книге не нужно — этот столбец всего лишь содержит уникальный первичный ключ, необходимый для таблицы **Messages**. Настроим **GridView** так, чтобы исключить этот столбец из

отображения на веб-форме. Мы также изменим заголовок столбца Message1, чтобы вместо него выводился текст Message.

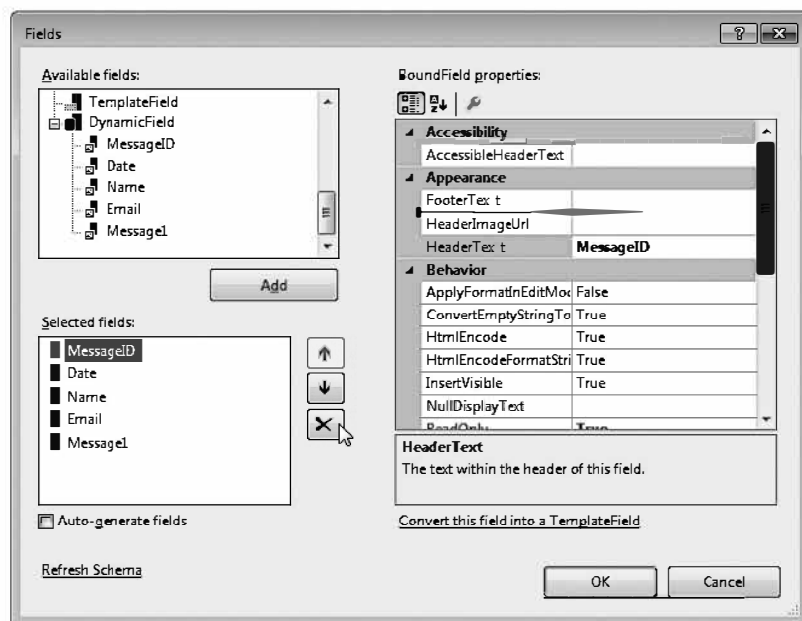
1. В меню смарт-тега GridView Tasks выберите команду Edit Columns, чтобы вызвать диалоговое окно Fields (ил. 23.35).
2. Выберите строку MessageID на панели Selected fields и щелкните на кнопке . Столбец MessageID удаляется из GridView.



**Ил. 23.34.** Элемент управления EntityDataSource для GridView в режиме конструктора

3. Выберите на панели Selected fields строку Message1 и измените ее свойство HeaderText на Message. Среда разработки переименовала это поле для предотвращения конфликта имен в классах модели данных сущностей. Щелкните на кнопке ОК, чтобы вернуться к главному окну IDE.
4. Создайте стиль для задания ширины GridView. Выполните команду **FORMAT ▸ New Style...**, чтобы вызвать диалоговое окно New Style. В поле Selector введите имя нового стиля .gridViewWidth. Выберите категорию с именем Position, задайте ширину (width:) равной 650px и щелкните на кнопке ОК, чтобы создать стиль и закрыть диалоговое окно. Затем задайте свойству CssClass элемента управления messagesGridView значение gridViewWidth.

Элемент управления GridView должен выглядеть так, как показано на ил. 23.31.



Ил. 23.35. Удаление столбца MessageID из GridView

## 23.8.2. Изменение файла программной логики приложения Guestbook

После построения веб-формы и настройки элементов данных, используемых в нашем примере, сделайте двойной щелчок на кнопках **Submit** и **Clear** в режиме конструктора, чтобы создать соответствующие обработчики событий **Click** в файле программной логики (ил. 23.36). IDE генерирует пустые обработчики событий, поэтому в них необходимо добавить код, обеспечивающий работу этих кнопок. Обработчик события **clearButton** (строки 39–44) очищает текстовое поле, задавая его свойству **Text** пустую строку. Форма инициализируется для отправки новой записи гостевой книги.

```

1 // Ил. 23.36: Guestbook.aspx.cs
2 // Файл программной логики с обработчиками событий гостевой книги.
3 using System;
4
5 public partial class Guestbook : System.Web.UI.Page
6 {
7     // Кнопка Submit отправляет новую запись гостевой книги в базу данных,
8     // очищает форму и выводит обновленный список записей
9     protected void submitButton_Click( object sender, EventArgs e )

```

Ил. 23.36. Файл программной логики приложения Guestbook (продолжение ↗)

```
10  {
11      // Использование DbContext для добавления сообщения
12      using ( GuestbookEntities dbContext = new GuestbookEntities() )
13      {
14          // Создание нового сообщения для добавления в базу данных;
15          // Message - класс модели данных, представляющий строку таблицы
16          Message message = new Message();
17
18          // Задание свойств нового объекта Message
19          message.Date = DateTime.Now.ToShortDateString();
20          message.Name = nameTextBox.Text;
21          message.Email = emailTextBox.Text;
22          message.Message1 = messageTextBox.Text;
23
24          // Добавление нового объекта Message в DbContext
25          dbContext.Messages.Add( message );
26          dbContext.SaveChanges(); // Сохранение изменений в базе данных
27      } // Конец команды using
28
29      // Очистка текстовых полей
30      nameTextBox.Text = String.Empty;
31      emailTextBox.Text = String.Empty;
32      messageTextBox.Text = String.Empty;
33
34      // Обновление GridView содержимым таблицы базы данных
35      messagesGridView.DataBind();
36  } // submitButton_Click
37
38  // Кнопка Clear очищает текстовые поля веб-формы
39  protected void clearButton_Click( object sender, EventArgs e )
40  {
41      nameTextBox.Text = String.Empty;
42      emailTextBox.Text = String.Empty;
43      messageTextBox.Text = String.Empty;
44  } // clearButton_Click
45 } // Конец класса Guestbook
```

**Ил. 23.36.** Файл программной логики приложения Guestbook (окончание)

Строки 9–36 содержат код обработки `submitButton`, который добавляет информацию пользователя в таблицу `Messages` базы данных `Guestbook`. Команда `using` в строках 12–27 начинается с создания объекта `GuestbookEntities` для взаимодействия с базой данных. Вспомните, что команда `using` вызовет `Dispose` для объекта `GuestbookEntities` после завершения. Так рекомендуется поступать при запросе веб-страниц ASP.NET, чтобы не удерживать подключение к базе данных после обработки запроса.

В строке 16 создается объект модели данных сущностей класса `Message`, представляющего строку таблицы базы данных `Messages`. Строки 19–22 задают свойствам объекта `Message` значения, которые должны быть сохранены в базе данных. Строка 25 вызывает метод `Add` свойства `Messages` объекта `GuestbookEntities`, представляющего таблицу `Messages` базы данных. Метод добавляет новую запись в представление таблицы в модели данных сущностей. Строка 26 сохраняет изменения в базе данных.

После того как данные будут включены в базу данных, строки 30–32 стирают содержимое текстовых полей, а строка 35 вызывает метод `DataBind` объекта `messagesGridView` для обновления данных, выводимых в `GridView`. Это заставляет `messagesEntityDataSource` (источник данных `GridView`) получить обновленные данные таблицы `Messages` из базы данных.

## 23.9. Итоги

Эта глава была посвящена разработке веб-приложений с использованием ASP.NET и Visual Studio Express 2012 for Web. Мы начали с рассмотрения простых операций HTTP, происходящих при запросе и получении веб-страницы в браузере. Затем были описаны три уровня (клиентский уровень, уровень бизнес-логики и информационный уровень), из которых состоит большинство веб-приложений.

Затем была рассмотрена роль файлов ASPX (то есть файлов веб-форм) и файлов программной логики, а также отношений между ними. Вы узнали, как ASP.NET компилирует и выполняет веб-приложения для отображения в браузере. Также было показано, как построить веб-приложение ASP.NET с использованием Visual Studio Express For Web.

В этой главе были продемонстрированы некоторые типичные веб-элементы управления ASP.NET, используемые для вывода текста и графики на веб-формах. Также были рассмотрены элементы проверки данных, которые позволяют проверить выполнение некоторых требований к вводимым данным на веб-страницах. Мы обсудили преимущества сохранения состояния пользователя между разными страницами сайта. Далее было показано, как реализовать такую функциональность в веб-приложении на базе объектов `HttpSessionState`.

Глава завершается приложением гостевой книги для ввода и сохранения комментариев о сайте. Вы узнали, как сохранить пользовательский ввод в базе данных и как вывести сохраненные комментарии на веб-странице.



# Приоритет операторов

В следующей таблице операторы перечислены в порядке убывания приоритета; группы операторов с одинаковым приоритетом разделяются пустыми серыми строками. Порядок применения операторов указан в правом столбце.

Операторы	Тип	Порядок применения
	Обращение к членам класса	слева направо
()	Вызов метода	
[]	Обращение к элементу	
++	Постфиксный инкремент	
--	Постфиксный декремент	
new	Создание объекта	
typeof	Получение объекта System.Type для типа	
sizeof	Получение размера в байтах для типа	
checked	Проверяемое вычисление	
unchecked	Непроверяемое вычисление	
+	Унарный плюс	справа налево
-	Унарный минус	
!	Логическое отрицание	
~	Поразрядное дополнение	
++	Префиксный инкремент	

продолжение ↗

Операторы	Тип	Порядок применения
--	Префиксный декремент	
(тип)	Преобразование типа	
*	Умножение	слева направо
/	Деление	
%	Вычисление остатка	
+	Сложение	слева направо
-	Вычитание	
>>	Сдвиг вправо	
<<	Сдвиг влево	
<	Меньше	слева направо
>	Больше	
<=	Меньше или равно	
>=	Больше или равно	
is	Сравнение типа	
as	Преобразование типа	
!=	Не равно	слева направо
==	Равно	
&	Логический оператор И	слева направо
^	Логический оператор ИСКЛЮЧАЮЩЕЕ ИЛИ	слева направо
	Логический оператор ИЛИ	слева направо
&&	Условный оператор И	слева направо
	Условный оператор ИЛИ	слева направо
??	Оператор проверки null	справа налево
?:	Условный оператор	справа налево



Операторы	Тип	Порядок применения
=	Присваивание	справа налево
*=	Умножение с присваиванием	
/=	Деление с присваиванием	
%=	Вычисление остатка с присваиванием	
+=	Сложение с присваиванием	
-=	Вычитание с присваиванием	
<<=	Сдвиг влево с присваиванием	
>>=	Сдвиг вправо с присваиванием	
&=	Логический оператор И с присваиванием	
^=	Логический оператор ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием	
=	Логический оператор ИЛИ	

# Б Простые типы

Тип	Размер в битах	Диапазон значений	Стандарт
bool	8	true или false	
byte	8	От 0 до 255 включительно	
sbyte	8	От -128 до 127 включительно	Юникод
char	16	От '\u0000' до '\uFFFF' (от 0 до 65535) включительно	
ushort	16	От 0 до 65535	
int	32	От -2 147 483 648 до 2 147 483 647 включительно	
uint	32	От 0 до 4 294 967 295 включительно	
float	32	Приближенный диапазон отрицательных значений: от -3.4028234663852886E+38 до -1.40129846432481707E-45 Приближенный диапазон положительных значений: от 1.40129846432481707E-45 до 3.4028234663852886E+38 Другие поддерживаемые значения: положительный и отрицательный ноль; положительная и отрицательная бесконечность; «не число» (NaN, Not A Number)	IEEE 754 IEC 60559
long	64	От -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 включительно	
ulong	64	От 0 до 18 446 744 073 709 551 615 включительно	
double	64	Приближенный диапазон отрицательных значений: от -1.7976931348623157E+308 до -4.94065645841246544E-324 Приближенный диапазон положительных значений: от 4.94065645841246544E-324 до 1.7976931348623157E+308 Другие поддерживаемые значения: положительный и отрицательный ноль; положительная и отрицательная бесконечность; «не число» (NaN, Not A Number)	IEEE 754 IEC 60559
decimal	128	Отрицательный диапазон: от -79 228 162 514 264 337 593 543 950 335 (-7.9E+28) до -1.0E-28 Положительный диапазон: от 1.0E-28 до 79 228 162 514 264 337 593 543 950 335 (7.9E+28)	

## Дополнительная информация о простых типах

Материалы приложения основаны на информации разделов 4.1.4–4.1.8 спецификации языка C# компании Microsoft и разделов 11.1.4–11.1.8 ECMA-334 (версии спецификации языка C# от ECMA). Эти документы доступны на следующих сайтах:

*[msdn.microsoft.com/en-us/vcsharp/aa336809.aspx](http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx)*

*[www.ecma-international.org/publications/standards/Ecma-334.htm](http://www.ecma-international.org/publications/standards/Ecma-334.htm)*

- ☐ Точность представления значений `float` составляет 7 знаков.
- ☐ Точность представления значений `double` составляет 15–16 знаков.
- ☐ Значения типа `decimal` представляются целыми значениями, нормированными по степеням 10. Значения в диапазоне от  $-1.0$  до  $1.0$  представляются ровно 28 цифрами.
- ☐ За дополнительной информацией о IEEE 754 обращайтесь по адресу *[grouper.ieee.org/groups/754/](http://grouper.ieee.org/groups/754/)*.

# В Набор символов ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

В левом столбце содержатся левые цифры десятичного эквивалента (0–127) кода символа, а в верхней строке — правые цифры. Например, код символа «F» равен 70, а код символа «&» — 38.

Набор символов ASCII используется для представления символов латиницы на многих компьютерах. Набор символов ASCII является подмножеством набора символов Юникод, используемого в C# для представления символов большинства мировых языков.