Глава 6

Объекты и интерфейсы

В этой главе…

- Основы объектно-ориентированного программирования
- Классы
- Объекты
- Интерфейсы
- Использование стандартных интерфейсов
- Библиотека System.Collections
- Библиотека System.Collections.Specialized
- Резюме
- Контрольные вопросы

Наследуя все лучшие стороны языка C++ и дополняя их современными технологиями программирования, основанными на идеологии объектно-ориентированного программирования, язык программирования C# позволяет разработчикам использовать все новейшие достижения в области программирования, и в частности классы и объекты, на основе которых создано множество компонентов, применение которых при разработке программ позволяет значительно повысить скорость создания новых программ при минимальном количестве ошибок. Программирование становится похожим на игру с кубиками-компонентами, из которых производится сборка программы. При этом программист должен хорошо знать все компоненты, находящиеся в его распоряжении, и чем лучше он это знает, тем быстрее и качественнее он разработает программу. Но помимо знания всех возможностей стандартных библиотек, часто приходится и самому разрабатывать отдельные компоненты, реализующие уникальную логику работы только вашей программы, которая не отражена в стандартных библиотеках. Понятно, что в сравнительно небольших стандартных библиотеках невозможно сохранить все разнообразие программ и алгоритмов, которые могут потребоваться для решения вашей задачи, поэтому многое придется делать самому. Но зато при этом вы создадите уникальную программу, которая хорошо будет решать поставленную вами задачу. Создав новый компонент, вы можете включить его в библиотеку программ и использовать в дальнейшем.

Именно на разработку новых компонентов и классов будет обращено внимание в данной главе, при этом будут приведены примеры программ с описанием их особенностей.

Основы объектно-ориентированного программирования

В настоящее время объектно-ориентированное программирование (ООП) принадлежит к числу ведущих компьютерных технологий и используется в большинстве языков программирования. Основная цель ООП, как и многих других подходов к программированию, — повышение эффективности разработки программ.

Появлению объектно-ориентированного программирования способствовало то обстоятельство, что программисты явно или неявно старались создавать программы, описывающие действия над различными структурами, выделяя их в отдельные программные фрагменты, что помогало лучшему восприятию логики программы. С программой становилось значительно легче работать и обнаруживать ошибки. Все это и привело к созданию концепции ООП. В традиционных способах программирования изменение данных или правил и методов их обработки часто приводило к значительным модификациям в программе. Всякое существенное изменение программы для программиста оборачивается дополнительными ошибками и дополнительным временем, необходимым для ее отладки. Использование ООП позволяет выйти из такой ситуации с минимальными потерями, сведя модификацию программы только к ее расширению и дополнению.

Вы уже знаете, как применять в своих программах процедуры и функции для программирования действий по обработке данных, которые приходится выполнять многократно. Использование процедур в свое время было важным шагом на пути к увеличению эффективности программирования. Процедура может иметь формальные параметры, заменяемые при обращении к ней аргументами. Но и в этом случае есть опасность вызова процедур с неправильными данными, что может привести к нарушению работы программы. Поэтому естественным обобщением традиционного подхода к программированию является объединение данных, процедур и функций, предназначенных для их обработки, в одном месте, или, используя терминологию языка C#, в одном типе. Для обозначения такого типа применяется общепринятый термин `class` (класс).

Классы

Базовым в объектно-ориентированном программировании является понятие класса. Класс имеет заранее заданные свойства. Состояние класса задается значениями его полей. Класс решает определенные задачи, т.е. располагает методами решения. Но сам класс непосредственно в программах не используется. Образно говоря, класс — это шаблон, на основе которого создаются экземпляры класса, или объекты. Программа, написанная с использованием ООП, состоит из объектов, которые могут взаимодействовать между собой.

Например, как только вы сформируете новое приложение типа Console Application в среде разработки .NET, в редакторе кодов автоматически появляется объявление класса `Class1`, которое выглядит так:

```
using System;
namespace ConsoleApplication2
{
  /// <summary>
  /// Описание для класса Class1.
  /// </summary>
  class Class1
  {
    /// <summary>
    /// Точка входа для приложения.
    /// </summary>
```

```
    [STAThread]
    static void Main(string[] args)
    {
      //
      // СДЕЛАТЬ: Добавить код приложения сюда.
      //
    }
  }
}
```

Это можно рассматривать как заготовку для разработки класса. Несколько сложнее это будет выглядеть при создании приложения для Windows (Windows Application).

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace WindowsApplication3
{
  /// <summary>
  /// Описание для класса Form1.
  /// </summary>
  public class Form1: System.Windows.Forms.Form
  {
    /// <summary>
    /// Необходимо для переменных проектирования.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1()
    {
      //
      // Необходимо для поддержки графического окна проектирования.
      //
      InitializeComponent();

      //
      // Сделать: Добавить код конструктора после инициализации
            компонентов.
      //
    }

    /// <summary>
    /// Освободить использованные ресурсы.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if (components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }
```

```
    #region Windows Form Designer generated code
    /// <summary>
    /// Необходимый метод для окна проектирования — не изменяйте
    /// содержимое через редактор кодов.
    /// </summary>
    private void InitializeComponent()
    {
      this.components = new System.ComponentModel.Container();
      this.Size = new System.Drawing.Size(300,300);
      this.Text = "Form1";
    }
    #endregion

    /// <summary>
    /// Точка входа для приложения.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.Run(new Form1());
    }
  }
}
```

В любом случае создается класс (`Class1` или `Form1`) и описывается метод `Main()`, который всегда вызывается первым при запуске приложения. Большая сложность второго типа приложения связана с тем, что формируется необходимая среда как для окна проектирования формы, так и для самого оригинального оконного приложения. В первом случае нет необходимости создавать все это, так как применяется стандартная консоль. Для изучения классов будет использоваться первый вариант приложения, а именно приложения типа Console Application, в котором меньше дополнительных деталей и лучше виден сам класс. Ниже перечислены основные понятия, связанные с классами.

- **Инкапсуляция**. Данные, методы их обработки и детали реализации скрыты от внешнего пользователя объекта. Преимущества инкапсуляции заключаются в модульности и изоляции кода объекта от другого кода программы.
- **Наследование**. Возможность создания новых объектов, которые наследуют свойства и поведение родительских классов. Такая концепция позволяет создавать иерархии объектов (например, библиотеки классов), включающие наборы объектов, порожденных от одного общего предка и обладающих все большей специализацией и функциональностью по сравнению со своими предшественниками, но, тем не менее, использующие все возможности родительских классов.
- **Полиморфизм**. Аналогом слова "полиморфизм" будет словосочетание "много форм". В данном случае под этим подразумевается, что вызов метода созданного объекта приведет к выполнению кода, взятого из конкретного экземпляра класса, соответствующего данному объекту. Хотя методы с аналогичными именами могут быть и в других родительских классах.

Преимущества наследования заключаются, в первую очередь, в совместном использовании многими объектами общего кода. От каких классов унаследован класс `Form1`, о котором говорилось выше, можно посмотреть в справочнике, если сделать поиск для строки "Form class". В описании класса будет приведена иерархия наследования для класса `Form` (рис. 6.1).

Рис. 6.1. *Иерархия наследования для класса* `Form`

Видно, что вверху иерархии находится класс `Object`. Обо всем этом подробнее будет говориться далее, а начнем с элементов класса.

Поля

Поле можно также определить как переменную экземпляра, представляющую собой данные конкретного объекта. К полям объекта по умолчанию не разрешен прямой доступ. Это способствует их защите от случайного или предумышленного искажения. Доступ к полям происходит через свойства, которые могут выступать как фильтры и не пропускать недопустимых значений. Объявления полей происходят как объявления обычных переменных. Например, можно в классе `Class1` объявить несколько переменных, как показано ниже. Это будут закрытые переменные, так как по умолчанию с ними будет использоваться ключевое слово `private`, о чем речь пойдет ниже.

```
class Class1
{
  int число_1;       // Объявление переменной типа int.
  string строка_1;   // Объявление переменной типа string.

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

Области видимости

Язык C# предоставляет дополнительный контроль уровня доступа к членам классов (полям и методам) с помощью ключевых слов `protected`, `private`, `public`, `internal` и `protected internal`, которые в данном случае называют модификаторами.

- `Private` (закрытый). Объявленные в данном разделе переменные и методы доступны только для кода, находящегося в блоке реализации самого объекта. Директива `private` скрывает особенности реализации объекта от пользователей и защищает члены этого объекта от непосредственного доступа и изменения извне.
- `Protected` (защищенный). Члены класса, объявленные в этом разделе, доступны объектам, производным от данного класса. Это позволяет скрыть внутреннее устройство

объекта от пользователя и в то же время обеспечить необходимую гибкость, а также эффективность доступа к полям и методам объекта для его потомков.

- `Public` (открытый). Объявленные в этом разделе члены объекта доступны в любом месте программы. Конструкторы и деструкторы всегда должны быть объявлены как `public`.
- `Internal` (внутренний). Область доступа ограничена текущей сборкой.
- `Protected internal` (защищенный внутренний). Область доступа ограничена текущей сборкой и объектами, производным от данного класса.

Модификаторы, присваиваемые по умолчанию, перечислены в таблице 6.1.

Таблица 6.1. Модификаторы, присваиваемые по умолчанию

Член типа	*Доступ по умолчанию*	*Разрешенные модификаторы*
`enum`	`public`	Нет

`class`	`private`	`public`
		`protected`
		`internal`
		`private`
		`protected internal`

`interface`	`public`	Нет

`struct`	`private`	`public`
		`internal`
		`private`

Методы

Методы представляют собой процедуры и функции, принадлежащие классу. Можно сказать, что методы определяют поведение класса. В классе всегда должен присутствовать важнейший метод — конструктор. Наличие деструктора не является обязательным, так как происходит автоматическая сборка мусора и после того, как метод будет выполнен, и все связанные с ним ресурсы будут освобождены автоматически. Но вы можете конкретизировать все вопросы, связанные с освобождением ресурсов, с помощью деструктора `Dispose()`, о чем будет говориться в дальнейшем. При проектировании класса можно создать произвольное количество любых других методов, необходимых для решения конкретных задач.

Для создания метода можно или вручную написать весь его код, или воспользоваться средствами среды разработки .NET и несколько автоматизировать процесс. Для этого откройте окно Class View, выберите нужный класс и щелкните на нем правой кнопкой мыши, в появившемся контекстном меню выберите пункт Add и затем команду Add Metod, после чего появится диалоговое окно, как показано на рис. 6.2 и 6.3.

После появления диалогового окна (мастер создания методов) нужно только ввести необходимые имена, модификаторы и параметры и щелкнуть на кнопке Finish. В редакторе кодов будет автоматически создана заготовка для метода, которую останется только заполнить кодом, выполняющим логику работы метода.

Ниже, в листинге 6.1, продемонстрирована программа с двумя методами: первый — это метод `Main()`, а второй — метод для вывода строки на консоль с именем `Write()`.

Листинг 6.1 Демонстрация использования методов

```
using System;
namespace ConsoleApplication2
{
  class Class1
  {
```

```
    [STAThread]
    static void Main(string[] args)
    {
      for (int i=0;i<args.Length;i++)
        Write(args[i]+"\r\n");
      Console.WriteLine("Нажми <Enter> для выхода из программы");
      Console.ReadLine();
    }

    static void Write(string str)
    {
      Console.Write(str);
    }
  }
}
```

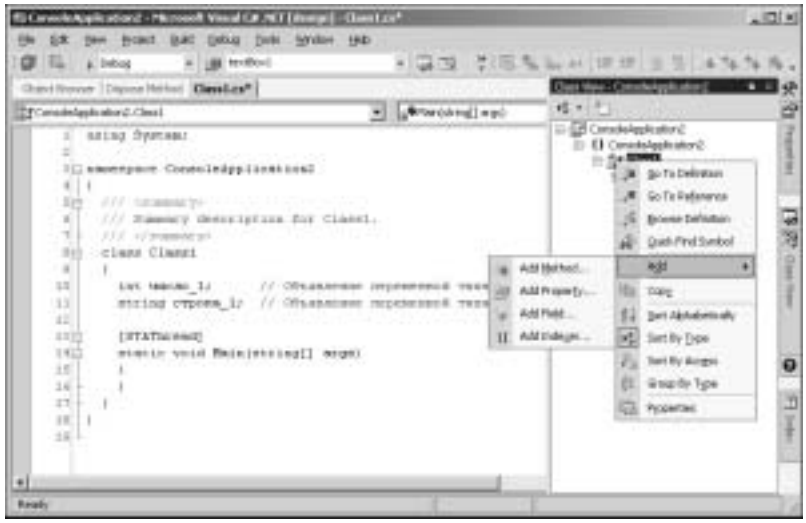

Рис. 6.2. *Создание метода с помощью среды разработки*

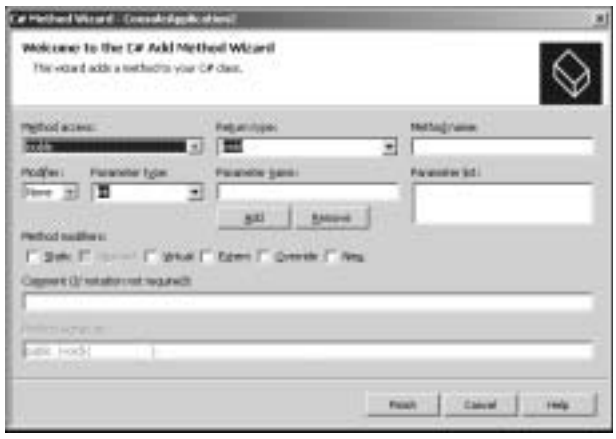

Рис. 6.3. *Диалоговое окно для создания метода*

В данной программе производится вывод на консоль всех аргументов командной строки, которые могут использоваться при вызове программы. Например, вы можете производить ее вызов не простым щелчком мыши на файле в каталоге, а с помощью командной строки, где можно добавить к имени программы дополнительные аргументы. Такие программы рассчитаны на то, чтобы их можно было запускать из других программ, передавая в них необходимые аргументы. Их можно запускать и с помощью командного файла, где также можно вводить дополнительные аргументы. Или просто поместить с помощью мыши на пиктограмму программы любой другой файл, как показано на рис. 6.4. В этом случае аргументом командной строки при запуске программы будет путь этого файла, который и отобразится программой (рис. 6.5).

Рис. 6.4. *Использование аргументов при запуске программы*

Рис. 6.5. *Вывод, выполненный программой*

Обратите внимание, что метод `Write()` объявлен с ключевым словом `static`, что необходимо для того, чтобы можно было применять этот метод без создания объекта. Поэтому, если вы объявите его с ключевым словом `public`, то придется дополнительно создавать объект типа `Class1`.

Типы методов

Методы могут быть описаны как невиртуальные, виртуальные или статические, для чего к ним добавляются соответствующие директивы.

Когда метод объявляется с ключевым словом (модификатором) `virtual`, то говорят, что это виртуальный метод. Когда нет этого модификатора, то говорят, что это невиртуальный метод.

Реализация невиртуального метода всегда только одна. Метод может вызываться как для экземпляра класса, так и для объекта, который создан на основе производного класса. В противоположность этому, описание виртуального метода может быть заменено в производном классе. Процесс замены описания унаследованного от базового класса метода называется переопределением.

Вызов виртуальных методов из-за возможности их переопределения немного сложнее, чем вызов невиртуальных методов, так как во время компиляции адрес конкретного вызываемого метода неизвестен. Для решения этой задачи компилятор строит таблицу виртуальных методов, обеспечивающую определение адреса метода в процессе выполнения программы. Таблица виртуальных методов содержит все виртуальные методы предков и виртуальные методы самого объекта.

Переопределение (замена) методов в языке C# реализует концепцию полиморфизма, позволяя изменять поведение метода от наследника к наследнику. Переопределение метода возможно только в том случае, если первоначально он был объявлен с модификатором `virtual`. Для переопределения метода при его объявлении вместо ключевого слова `virtual` следует указать ключевое слово `override`.

В примере ниже (листинг 6.2) показано различие между виртуальными и невиртуальными методами.

Листинг 6.2. Виртуальные и невиртуальные методы

```
using System;
class A
{
  public void F() { Console.WriteLine("A.F"); }
  public virtual void G() { Console.WriteLine("A.G"); }
}
class B: A
{
  new public void F() { Console.WriteLine("B.F"); }
  public override void G() { Console.WriteLine("B.G"); }
}
class Test
{
  static void Main()
  {
    B b = new B();
    A a = b;
    a.F();
    b.F();
    a.G();
    b.G();
  }
}
```

В классе `A` объявлены невиртуальный метод `F` и виртуальный метод `G`. В классе `B`, являющемся производным от класса `F`, представлен новый невиртуальный метод `F`, который скрывает унаследованный метод `F`, и переопределенный виртуальный метод `G`. В результате получается следующий вывод на консоль:

```
A.F
B.F
B.G
B.G
```

Обратите внимание, что в утверждении `a.G();` вызывается метод класса `B`, а не аналогичный метод класса `A`. Это происходит потому, что во время выполнения данный экземпляр принимает тип `B` и соответственно вызывает метод, связанный с этим типом, хотя во время компиляции он был объявлен как тип `A`. Для невиртуальных методов вызов производится в соответствии с объявленным типом.

Поскольку методы могут скрывать унаследованные методы, то в классе может содержаться несколько виртуальных методов с одинаковыми сигнатурами. Но это не представляет неразрешимых проблем с неоднозначностью, поскольку большинство методов можно скрыть, используя ключевое слово `new`. Например:

```
using System;
class A
{
   public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
   public override void F() { Console.WriteLine("B.F"); }
}
class C: B
{
```

```
   new public virtual void F() { Console.WriteLine("C.F"); }
}
class D: C
{
   public override void F() { Console.WriteLine("D.F"); }
}
class Test
{
   static void Main() {
      D d = new D();
      A a = d;
      B b = d;
      C c = d;
      a.F();
      b.F();
      c.F();
      d.F();
   }
}
```

В этой программе классы `C` и `D` содержат методы с одинаковыми сигнатурами: в первом случае аналогичный метод представлен в классе `A`, а во втором — в классе `B`. Метод из класса `C` скрывает унаследованный метод класса `A`. Таким образом, модификатор `override` в классе `D` переопределяет метод класса `C`, и для метода из класса `D` нет возможности переопределить метод класса `A`. В этом случае будет сделан следующий вывод на консоль:

```
B.F
B.F
D.F
D.F
```

Невиртуальные методы работают подобно обычным процедурам или функциям. Этот тип методов устанавливается по умолчанию. Адрес такого метода известен уже на стадии компиляции, и компилятор в коде программы оформляет все вызовы данного метода как невиртуальные. Такие методы работают быстрее других, однако не могут быть перегружены с целью полиморфизма объектов.

Статические методы

Для использования обычного метода необходимо сначала создать объект (экземпляр класса) и только потом делать вызов метода в программе, для чего применяются имена объекта и метода, написанные через точку. Например, в ранее приведенном объекте `a` вызов метода `F()` происходит как `a.F()`, но если перед объявлением метода поставить ключевое слово `static`, то этот метод может быть вызван как обычная процедура или функция без создания экземпляра класса, членом которого он является. Необходимо иметь в виду, что при создании подобных методов нельзя использовать никакой информации экземпляра класса, поскольку это приведет к ошибке компиляции, так как объект не существует. С ключевым словом `static` можно объявлять любые члены класса, как видно из листинга 6.3.

Листинг 6.3. Использование статических членов

```
using System;
class A
{
   public static int X;
   static A() {
```

```
      X = B.Y + 1;
   }
}
class B
{
   public static int Y = A.X + 1;
   static B() {}
   static void Main() {
      Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
   }
}
```

В результате выполнения этой программы будет сделан следующий вывод:

```
X = 1, Y = 2
```

Перегрузка методов

Подобно обычным процедурам и функциям, методы могут быть перегружены таким образом, чтобы класс содержал несколько методов с одним именем, но с различными списками параметров (различные сигнатуры). Перегруженные методы выявляются автоматически и нет необходимости добавлять ключевое слово `overload`, как это делается в других языках программирования. Вот пример использования класса с перегруженными методами (листинг 6.4).

Листинг 6.4. Перегрузка методов

```
using System;
namespace ConsoleApplication3
{
  class Class1
  {
    void Write(int x)
    {
      Console.WriteLine("целое = {0}", x);
    }
    void Write(float x)
    {
      Console.WriteLine("вещественное = {0}", x);
    }
    [STAThread]
    static void Main(string[] args)
    {
      Class1 c = new Class1();
      c.Write(10);
      c.Write(1.1f);
      Console.WriteLine("Нажми <Enter> для выхода из программы");
      Console.ReadLine();
    }
  }
}
```

В данном случае компилятор будет производить отбор не только по имени, но и учитывать параметры, которые должны быть различными для методов с одинаковыми именами. Поэтому вывод, сделанный программой, будет следующим:

```
целое = 10
вещественное = 1,1
```

Дублирование имен методов

Иногда может понадобиться к одному из классов добавить метод, перегружающий метод с тем же именем, но принадлежащий предку этого класса. В данном случае требуется не переопределять исходный метод, а полностью его заменить. Это тоже можно сделать без специального указания компилятору, он разберется во всем автоматически, как показано в примере ниже (листинг 6.5).

Листинг 6.5. Перегрузка методов в разных классах

```
using System;
namespace ConsoleApplication3
{
  class Class1
  {
    public void Write(int x)
    {
      Console.WriteLine("целое = {0}", x);
    }

    [STAThread]
    static void Main(string[] args)
    {
      Class1 c = new Class1();
      Class2 d = new Class2();
      c.Write(10);
      d.Write(1.1f);
      Console.WriteLine("Нажми <Enter> для выхода из программы");
      Console.ReadLine();
    }
  }

  class Class2: Class1
  {
    public void Write(float x)
    {
      Console.WriteLine("вещественное = {0}", x);
    }
  }
}
```

Вывод на консоль, сделанный этой программой, будет аналогичен выводу предыдущей программы.

Указатель this

Во всех методах класса доступна неявная переменная `this`, представляющая собой указатель на тот экземпляр класса, который был использован при вызове этого метода. Переменная `this` передается компилятором методу в качестве скрытого параметра. Статические методы не имеют указателя `this`, так как их нельзя связать ни с каким объектом, а только с самим классом. Указатель `this` может использоваться для доступа к членам класса в конструкторах или нестатических методах. Ниже представлено несколько примеров применения указателя `this`.

- Уточнение члена, скрытого аналогичным именем:

```
public Employee(string name, string alias)
{
   this.name = name;
   this.alias = alias;
}
```

- Передача объекта как параметра в метод:

```
CalcTax(this);
```

- Объявление индексаторов:

```
public int this [int param]
{
   get
   {
      return array[param];
   }
   set
   {
      array[param] = value;
   }
}
```

Использование указателя `this` в статических методах, статических свойствах или при инициализации полей приводит к ошибке времени компиляции.

Свойства

В основе свойств лежат промежуточные методы, обеспечивающие доступ к данным и коду, содержащемуся в классе, таким образом избавляя конечного пользователя от деталей реализации класса. По отношению к компонентам свойства являются теми элементами, сведения о которых отображаются в инспекторе объектов. Добавим свойство `Счетчик` для закрытой переменной `счетчик` в класс `Class1`, для чего необходимо выполнить следующие действия.

1. Создать приложение типа Console Application.
2. В классе `Class1` создать закрытую (`private`) переменную `счетчик`.
3. Открыть окно Class View и щелкнуть правой кнопкой мыши на имени класса. В появившемся контекстном меню выбрать команду Add⇨Add Property.
4. В появившемся диалоговом окне Property Wizard (Мастер создания свойств) выбрать необходимые параметры и нажать кнопку Finish, как показано на рис. 6.6. Среда разработки .NET автоматически закончит написание кодов. В результате будет получен представленный ниже листинг 6.6.

Листинг 6.6. Автоматическое завершение написания свойств

```
using System;
namespace ConsoleApplication4
{
  class Class1
  {
    int счетчик;
```

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

    public int Счетчик
    {
      get
      {
        return 0;
      }
      set
      {
      }
    }
  }
}
```

***Рис. 6.6.** Мастер создания свойств*

В этом листинге показан результат создания свойства `Счетчик`, что выражается во введении дополнительных процедур `get` и `set`. Для того чтобы связать свойство `Счетчик` с закрытой переменной `счетчик`, необходимо в процедуры `get` и `set` добавить соответствующие коды. Например:

```
public int Счетчик
{
  get
  {
    return счетчик;
  }
  set
  {
    счетчик = value;
  }
}
```

Со свойством можно обращаться как с обычной переменной. При попытке присвоить свойству `Счетчик` некоторое значение вызывается процедура `set`, предназначенная для изменения значения поля `счетчик`. Ключевое слово `value` в данном случае определяет вводимое значение. Для считывания также можно создать функцию.

Такая технология имеет два основных преимущества. Во-первых, она создает для конечного пользователя некий интерфейс, полностью скрывающий реализацию объекта и предоставляющий контроль за доступом к нему. Во-вторых, данная технология позволяет замещать методы в производных классах, что обеспечивает полиморфизм поведения объектов.

При объектно-ориентированном подходе прямой доступ к полям объекта считается плохим стилем программирования, поскольку детали реализации объекта могут со временем измениться. Предпочтительнее работать со свойствами, представляющими собой стандартный интерфейс объекта, скрывающий его конкретную реализацию. Например:

Листинг 6.7. Использование свойств

```
using System;
public class Employee
{
   public static int numberOfEmployees;
   private static int counter;
   private string name;

   // Свойство для чтения и записи в переменную name.
   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }

   // Свойство только для чтения из переменной counter.
   public static int Counter
   {
      get
      {
         return counter;
      }
   }

   // Constructor:
   public Employee()
   {
      // Рассчитать номер сотрудника.
      counter = ++counter + numberOfEmployees;
   }
}

public class MainClass
{
   public static void Main()
   {
```

```
      Employee.numberOfEmployees = 100;
      Employee e1 = new Employee();
      e1.Name = "Иван Иванов";
      Console.WriteLine("Номер сотрудника: {0}", Employee.Counter);
      Console.WriteLine("Имя сотрудника: {0}", e1.Name);
   }
}
```

Вывод, сделанный программой, будет следующим:

```
Номер сотрудника: 101
Имя сотрудника: Иван Иванов
```

Пример создания производного класса

Для того чтобы лучше понять принципы объектно-ориентированного программирования, описанные в разделе довольно кратко, следует привести реальный пример создания производного класса, расширяющий возможности стандартного класса `FileStream`. Стандартный класс обеспечивает взаимодействие программы с рабочими файлами, позволяя записывать и считывать данные из файла по одному или по несколько байтов. Производный от `FileStream` класс, наследуя все методы базового класса, дает возможность записывать и считывать не только по байтам, но и по битам, что необходимо в программах шифрования и сжатия данных. Полностью рабочие классы приведены в листинге 6.8.

Листинг 6.8. Классы для записи и чтения произвольного количества битов, производные от класса FileStream

```
using System;
using System.IO;
namespace Press
{
  /// <summary>
  /// Класс WriteBits используется для записи заданного количества
  /// бит в выходной файл. Если выходной файл существует, он будет
  /// перезаписан. Максимальное число для записи не более 2^24.
  /// Для целей сжатия информации это вполне достаточно, поэтому нет
  /// проверки превышения этого значения.
  /// </summary>
  public class WriteBits: FileStream  // Наследуем класс FileStream
  {
    uint    bitBuf = 0;  // Буфер для временного хранения чисел.
    sbyte   index  = 0;  // Конец последнего записанного числа.

    public WriteBits (string filepath): base(filepath,
             FileMode.Create) {}

    // Метод для записи числа с заданным количеством бит.
    public void WriteQBit (sbyte qBit, uint dig)
    {
      while (index>7)
      {
        // Записываем левый байт буфера bitBuf.
        WriteByte ((byte)(bitBuf>>24)); // Сдвигаем буфер
            и записываем байт.
        bitBuf <<= 8;                   // Модифицируем буфер.
        index -= 8;                     // Модифицируем индекс.
      }
```

```
    bitBuf += dig<<(32-index-qBit);   // Записываем в буфер
          число dig,
    index += qBit;                    // занимающее qBit бит.
  }

  // Очистка буфера.
  public void FlushBitBuf ()
  {
    while (index>0)
    {
      WriteByte ((byte)(bitBuf>>24));
      bitBuf <<= 8;
      index -= 8;
    }
    Flush();
    Close();
  }
}

/// <summary>
/// Класс ReadBits используется для считывания заданного количества
/// бит из входного файла и преобразования их в целое число.
/// </summary>

public class ReadBits: FileStream
{
  int    bitBuf=0, tmpInt;
  sbyte    index  = 0;
  bool    endFile = false;

  // Метод для считывания числа с заданным количеством бит.
  public ReadBits (string filePath): base (filePath,
        FileMode.Open) {}

  public int ReadQBit (sbyte qBit)
  {
    if (!endFile)
    {
      while (index<25)
      {
        if ((tmpInt=ReadByte())==-1)  // Считывание байта из файла и
        {                             // контроль конца файла.
          endFile = true;
          Close();
          break;
        }
        else
        {
          tmpInt <<= 24-index;    // Сдвиг байта на нужное место.
          bitBuf |= tmpInt;       // Запись байта в буфер.
          index += 8;             // Увеличиваем индекс
                на размер байта.
        }
      }
    }
    if (index<qBit)
      return -1;
    else
```

```
      {
        tmpInt = bitBuf>>(32-qBit);      // Сдвигаем буфер.
        bitBuf <<= qBit;                 // Модифицируем буфер.
        index -= qBit;                   // Модифицируем индекс.
        return (tmpInt&=((1<<qBit)-1)); // Возвращаем очередное число,
      }                                  // занимающее qBit бит.
    }
  }
}
```

Как видно из листинга, вновь созданные классы имеют методы для считывания и записи битов в файл. В качестве параметров для этих методов указывается число и количество битов, необходимых для его записи или считывания. Это позволяет сократить размер файла за счет того, что для записи чисел можно выделять размер, не кратный одному байту. Например, для записи числа 257 можно использовать всего 9 битов, а не два байта (16 бит), как это придется делать при непосредственной записи в поток `FileStream`.

Сейчас не будем полностью анализировать работу этой программы, хотя принцип ее функционирования можно понять из сделанных комментариев. Идея очень простая: так как нельзя непосредственно записать в файл заданное число битов, используется промежуточный буфер, в который записываются биты, а в файл считываются или записываются байты. Обратите внимание только на оформление производного класса. В конструкторы производных классов передаются те параметры, которые необходимы в конструкторах базового класса. Конструкторы базового класса вызываются при создании классов, для чего используется ключевое слово `base`. Методы базового класса (`WriteByte`, `ReadByte`) вызываются безо всякой ссылки на базовый класс, т.е. можно считать, что они принадлежат классу `WriteBits` или `ReadBits` соответственно. Методы `Flush()` и `Close()` также относятся к базовым классам.

В дальнейшем вновь созданные классы `WriteBits` и `ReadBits` сами могут использоваться в качестве базовых, например:

```
public class Coder: WriteBits
{
  ...
}
```

Объекты

Как уже говорилось ранее, объекты (экземпляры класса) представляют собой сущности, которые могут содержать данные и код. Объекты в языке C# предоставляют программисту все основные возможности объектно-ориентированного программирования, такие как наследование, инкапсуляция и полиморфизм.

Объявление и создание объекта

Перед тем как создать объект, следует объявить класс, на основе которого этот объект будет создан. Разумеется, существует большое число стандартных классов, которые объявлять не нужно, так как они уже объявлены. В языке программирования C# класс объявляется с помощью ключевого слова `class`.

Только после объявления класса можно делать объявление переменных этого типа. Например:

```
WriteBits wb = new WriteBits();
```

Полностью объект создается с помощью вызова одного из конструкторов соответствующего класса, которые имеют имя, совпадающее с именем класса. Конструктор отвечает за создание объекта, а также за выделение памяти и необходимую инициализацию полей. Конструктор не только создает объект, но и приводит этот объект в состояние, необходимое для его дальнейшего применения. Каждый класс содержит по крайней мере один конструктор, который может иметь различное количество параметров разного типа — в зависимости от типа объекта. Ниже рассматривается только простейший конструктор без параметров.

Необходимо запомнить, что, в отличие от других языков программирования, например C++, конструкторы в языке C# не вызываются автоматически. Создание каждого объекта с помощью вызова его конструктора входит в обязанности программиста. Синтаксис вызова конструктора следующий:

```
[Тип] [Имя объекта] = new [Конструктор используемого типа]
```

Здесь используется ключевое слово `new`, которое является указанием компилятору на выделение памяти объекту заданного типа для размещения в ней переменных объекта. Вызов конструктора для создания экземпляра класса часто называют *созданием объекта*. При создании объекта с помощью конструктора компилятор гарантирует, что все поля объекта будут инициализированы, все числовые поля будут обнулены, указатели примут значение `null`, а строки будут пусты.

При создании объекта производится инициализация всех полей объекта. Если в конструкторе не задаются значения отдельных полей, то числовые поля будут обнулены, указатели примут значение null, а строки будут пусты.

Уничтожение объекта

По окончании использования объекта следует освободить выделенную для него память, иначе вполне возможно переполнение памяти и, соответственно, крах программы. Но в языке C# нет необходимости делать это явно, так как уничтожением уже ненужных объектов занимается сборщик мусора. Сборка мусора производится по довольно сложному алгоритму, который здесь рассматриваться не будет, только стоит отметить, что автоматическая сборка мусора не всегда может устраивать программиста. Например, перед тем как объект выйдет из области видимости работающей программы, т.е. не останется активных ссылок на него, он попадает в поле зрения сборщика мусора, который может уничтожить его без предупреждения в любой момент. Но при этом до уничтожения объекта иногда возникает необходимость произвести некоторые предварительные действия, например закрыть созданное объектом соединение. Этого сборщик мусора сделать не сможет и поэтому программисту необходимо самому предусмотреть необходимые действия до уничтожения объекта. Это можно осуществить с помощью специального метода — деструктора, который в языке C# называется так же, как и имя класса, но с добавленной впереди имени тильдой (~), указывающей на то, что данный метод является деструктором. Именно в деструктор разработчик и помещает необходимый код, который нужно выполнить до того, как объект будет уничтожен. То есть, если сборщик мусора видит, что в объекте находится деструктор, он вызывает его на выполнение до того, как разрушит объект и передаст занимаемую им память в распоряжение операционной системы.

Интерфейсы

Возможно, наиболее важным средством, делающим язык C# удобным для решения большого круга задач, являются интерфейсы. Интерфейс определяет набор функций и процедур, которые могут быть использованы для взаимодействия программы с объектом, так как в объектно-

ориентированном программировании обычно сначала требуется определить, что именно класс должен делать, а не то, как он это будет делать. В классе может быть реализовано несколько интерфейсов. В результате объект становится "многоликим", являя клиенту каждого интерфейса свое особое лицо.

Интерфейсы определяют, каким образом могут общаться между собой объект и его клиент. Класс, поддерживающий некоторый интерфейс, обязан обеспечить реализацию всех его методов.

Формальное определение интерфейса звучит так: интерфейс — это набор семантически связанных абстрактных членов. Количество членов, определенных в конкретном интерфейсе, зависит от того, какое поведение моделируется посредством этого интерфейса. С точки зрения синтаксиса интерфейсы в C# определяются следующим образом:

```
// Интерфейс IPointy определяет возможности работы с углами
// геометрической фигуры
public interface IPointy
{
  byte GetNumberOfPoints(); // Автоматически этот
                            // метод становится абстрактным.
}
```

Интерфейсы могут поддерживать любое количество свойств (и событий). Например, интерфейс `IPointy` может содержать свойство для чтения и записи:

```
public interface IPointy
{
  // Чтобы сделать это свойство "только для чтения" или "только
  // для записи", достаточно просто удалить соответствующий
  // блок set или get.
  byte Points(get; set;)
}
```

В любом случае интерфейс — это не более чем именованный набор абстрактных членов, а это значит, что в любом классе, реализующем его, необходимо полностью описывать каждый из членов этого интерфейса. Это еще один способ реализации полиморфизма в приложении, поскольку в разных классах члены одних и тех же интерфейсов будут реализованы по-разному, в результате чего эти классы будут реагировать на одни и те же вызовы по-своему, т.е. в соответствии с тем кодом, который сделан при их описании.

Вышеприведенный интерфейс `IPointy` — это элементарный интерфейс, определяющий поведение, связанное с углами геометрических фигур. В качестве примера будет использована иерархия геометрических фигур, производных от базового класса `Shape`. Главная идея проста: у некоторых геометрических фигур, например у шестиугольника (класс `Hexagon`) и треугольника (класс `Triangle`), есть углы, поэтому в них можно использовать интерфейс `IPointy`, который и говорит о том, что данные классы применяют углы и к ним можно обращаться для получения какой-либо информации об углах. У других классов, например окружность (`Circle`), углов нет, и такие классы этот интерфейс поддерживать не должен. Поэтому, если известно, что какой-то класс поддерживает определенный интерфейс, то можно ожидать от него соответствующей реакции на методы этого интерфейса. Таким образом, если в классах `Hexagon` и `Triangle` реализован интерфейс `IPointy`, то эти классы должны реагировать на метод `GetNumberOfPoints()` и возвращать количество углов фигуры, так как об этом говорит осмысленное имя метода — получить количество углов.

Интерфейс — это чистая синтаксическая конструкция, которая предназначена только для создания более удобной для понимания и анализа программы. Интерфейсы никогда не являются типами данных и в них не бывает реализаций методов по умолчанию. Каждый член

интерфейса (будь то свойство или метод) автоматически становится абстрактным, т.е. не имеющим кодов, описывающих логику работы. Кроме того, необходимо учесть, что в языке C#, да и в среде .NET, наследование более чем от одного базового класса (множественное наследование) запрещено. В то же время реализация в классе сразу нескольких интерфейсов позволяет обойти это ограничение и сымитировать множественное наследование, например:

```
interface IMyInterface: IBase1, IBase2
{
   void MethodA();
   void MethodB();
}
```

Реализация интерфейса

Когда в языке C# какой-либо класс должен реализовать необходимые интерфейсы, названия этих интерфейсов помещаются через запятую после двоеточия, следующего за именем этого класса. Рассмотрим класс, в котором реализованы два интерфейса: `IEnglishDimensions` (английская система мер) и `IMetricDimensions` (метрическая система мер). В обоих интерфейсах имеются одинаковые члены: `Length` и `Width`.

```
// Объявляем интерфейс с английской системой.
interface IEnglishDimensions
{
   float Length();
   float Width();
}
// Объявляем интерфейс с метрической системой.
interface IMetricDimensions
{
   float Length();
   float Width();
}
// Объявляем класс Box с реализацией этих интерфейсов.
class Box: IEnglishDimensions, IMetricDimensions
{
   float lengthInches;
   float widthInches;
   public Box(float length, float width)
   {
      lengthInches = length;
      widthInches = width;
   }
// Реализация члена интерфейса IEnglishDimensions.
   float IEnglishDimensions.Length()
   {
      return lengthInches;
   }
   float IEnglishDimensions.Width()
   {
      return widthInches;
   }
// Реализация члена интерфейса IMetricDimensions:
   float IMetricDimensions.Length()
   {
      return lengthInches * 2.54f;
   }
   float IMetricDimensions.Width()
```

```
    {
        return widthInches * 2.54f;
    }

    public static void Main()
    {
        // Объявляем объект myBox.
        Box myBox = new Box(30.0f, 20.0f);
        // Объявляем экземпляр интерфейса IEnglishDimensions.
        IEnglishDimensions eDimensions = (IEnglishDimensions) myBox;
        // Объявляем экземпляр интерфейса IMetricDimensions.
        IMetricDimensions mDimensions = (IMetricDimensions) myBox;
        // Выводим результаты в английской системе мер.
        System.Console.WriteLine("Length(in): {0}",
                  eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}",
                  eDimensions.Width());
        // Выводим результаты в метрической системе мер.
        System.Console.WriteLine("Length(cm): {0}",
                  mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}",
                  mDimensions.Width());
    }
}
```

Получаем следующий вывод:

```
Length(in): 30
Width (in): 20
Length(cm): 76.2
Width (cm): 50.8
```

Но если вы хотите сделать по умолчанию так, чтобы все измерения выводились в английской системе мер, то можете реализовать методы `Length` и `Width` обычным образом, и отдельно реализовать методы `Length` и `Width` для метрической системы.

```
// Обычная реализация.
public float Length()
{
    return lengthInches;
}
public float Width()
{
    return widthInches;
}
// Реализация интерфейса с метрической системой.
float IMetricDimensions.Length()
{
    return lengthInches * 2.54f;
}
float IMetricDimensions.Width()
{
    return widthInches * 2.54f;
}
```

В этом случае вы получаете доступ к английской системе мер через экземпляр класса, а к метрической — через интерфейс:

```
System.Console.WriteLine("Length(in): {0}", myBox.Length());
System.Console.WriteLine("Width (in): {0}", myBox.Width());
System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
```

Обратите внимание, что нельзя выбирать, какие методы в интерфейсе реализовывать, а какие — нет. В классе должны быть реализованы все методы данного интерфейса, иначе теряется суть использования интерфейсов — создать четко определенное поведение класса, при котором учитываются все, а не отдельные особенности данного поведения.

В классе обязательно должны быть реализованы все методы данного интерфейса.

Получение ссылки на интерфейс

Язык C# позволяет обращаться к членам интерфейсов несколькими способами. Предположим, что нужно создать объект одного из классов и необходимо вызвать для него метод `GetNumberOfPoints()`, входящий в интерфейс `IPointy`. Первый способ — воспользоваться явным приведением типов:

```
// Получаем ссылку на интерфейс IPointy,
// используя явное приведение типов
Hexagon hex = new Hexagon();
IPointy itfPt = (IPointy)hex;
Console.WriteLine(itfPt.GetNumberOfPoints());
```

Здесь создается ссылка на интерфейс `IPointy`, явно приводя объект класса `Hexagon` к типу `IPointy`. Если класс `Hexagon` поддерживает интерфейс `IPointy`, получим ссылку на интерфейс (она названа `itfPt`), и все будет хорошо. Однако что произойдет, если попробовать применить то же приведение типов к экземпляру класса, не поддерживающего интерфейс `IPointy` (например, объект типа `Circle`)? Понятно, что этого делать нельзя, и в результате будет получено сообщение об ошибке времени выполнения. При этом система генерирует исключение `InvalidCastException`.

Чтобы избежать проблем с генерацией исключения и остановкой программы, исключение можно перехватить.

```
// Используя .программные средства, поэтапно перехватываем исключение
Circle c = new Circle();
IPointy itfPt;
try  // Обрабатываем все исключения.
{
  itfPt = (IPointy)c;
  Console.WriteLine(itfPt.GetNumberOfPoints());
}
catch(InvalidCastException e) // Перехватываем заданное исключение.
{
  Console.WriteLine("СТОП! Нет углов!"); // Решаем что делать.
}
```

Второй способ получить ссылку на интерфейс — использовать ключевое слою `as`:

```
// Еще один способ получить ссылку на интерфейс
Hexagon hex2 = new Hexagon();
IPointy itfPt2:
itfPt2 = hex2 as IPointy;
```

```
if(itfPt2 != null)
  Console. WriteLine(itfPt2.GetNumberOfPoints());
else
  Console.WriteLine("СТОП! Нет углов!");
```

Если при использовании ключевого слова `as` попробовать создать ссылку на интерфейс через объект, который этот интерфейс не поддерживает, ссылка будет просто установлена в `null`, и при этом никаких исключений генерироваться не будет.

Третий способ получения ссылки на интерфейс — воспользоваться оператором `is`. Если объект не поддерживает интерфейс, условие станет равно `false`:

```
Triangle t = new Triangle();
if(t is IPointy)        // У t есть углы?
  Console.WriteLine(t.GetNumberOfPoints());
else
  Console.WriteLine("СТОП! Нет углов!");
```

Если имеется массив разных объектов и при этом необходимо выяснить в процессе выполнения, какие именно объекты из этого массива поддерживают определенный интерфейс, а какие нет, это можно сделать любым из приведенных выше способов.

Интерфейсы как параметры

Интерфейсы можно рассматривать как специальные переменные. Это сходство подтверждается тем, что язык C# позволяет использовать интерфейсы как параметры, принимаемые и возвращаемые методами. Для наглядности представим, что задан еще один интерфейс с именем `IDraw3D`:

```
// Интерфейс для отображения фигур в трех измерениях.
public interface IDraw3D
{
  void Draw3D();
}
```

Предположим, что этот интерфейс реализован только в двух из трех классов для геометрических фигур — в классах `Circle` и `Hexagon`:

```
// Circle поддерживает интерфейс IDraw3D.
public class Circle: Shape, IDraw3D
{
  public void DrawSDO
  {
    Console.WriteLine("Отобразить круг в трех измерениях!");
  }
}
// Если тип поддерживает несколько интерфейсов, они
// перечисляются через запятую после базового класса.
public class Hexagon: Shape, IPointy, IDraw3D
{
  public void Draw3D()
  {
    Console.Writeline("Отобразить шестигранник в трех измерениях!");
  }
}
```

Если определить какой-нибудь метод, принимающий интерфейс IDraw3D в качестве параметра, можно будет просто передавать этому методу любой объект, поддерживающий IDraw3D, например метод DrawThisShapeIn3D:

```
// Отображаем фигуры в трех измерениях, если возможно!
public class ShapesApp
{
  // Будут отображены все объекты с интерфейсом IDraw3D
  public static void DrawThisShapeIn3D(IDraw3D itf3d)
  {
    itf3d.Draw3D();
  }

  public static int Main(string[] args)
  {
    Shape[] s = { new Hexagon(), new Circle(), new Triangle() };
    for(int i=0; i<s.Length; i++)
    {
      if(s[i] is IDraw3D) // Этот объект использует IDraw3D?
        DrawThisShapeIn3D((IDraw3D)s[i]); // Да. Отобразить.
    }
    return 0;
  }
}
```

Явная реализация интерфейса

В предыдущем примере единственный метод интерфейса IDraw3D назван как Draw3D() во избежание конфликта имен с методом Draw(), определенным в базовом классе Shape.

Такое задание интерфейса и метода вполне приемлемо, однако более удобным и логичным именем метода будет все-таки имя Draw():

```
// Интерфейс для отображения фигур в трех измерениях.
public interface IDraw3D
{
  void Draw();
}
```

Однако не возникнет ли проблем, если попытаться создать класс, производный от класса Shape() и одновременно реализующий интерфейс IDraw3D? Ведь теперь метод с именем Draw() будет унаследован от базового класса и получен от интерфейса IDraw3D. Компилятор не будет фиксировать ошибку, однако что произойдет, если обратиться к объекту типа Line таким образом:

```
// Вызываем Line.Draw()
Line myLine = new Line();
myLine.Draw();
// Вызываем Line.Draw()другим способом.
Idraw3D itfDraw3d = (IDraw3D) myLine;
itfDraw3d.Draw();
```

Исходя из знаний о базовом классе Shape и интерфейсе IDraw3D, очень похоже, что наследуются сразу два абстрактных метода Draw(). Для обоих абстрактных методов класс Line предлагает единую конкретную реализацию Draw(), и это вполне допустимо. Как потом не вызывался бы этот метод — через ссылку на объект или на интерфейс, — все равно будет вызван тот же вариант метода.

Однако проблемы еще не кончились. А если, исходя из логики рассуждений, нужно будет иметь два метода с одинаковыми именами — `IDraw3D.Draw()` для отображения объекта в трех измерениях и переопределенный метод `Shape.Draw()` для обычного отображения фигуры на плоскости? То есть возникает необходимость обратиться к методам интерфейса только через ссылку на интерфейс, но не через ссылку на объект. В рассмотренной выше ситуации к методу `Draw()` можно обратиться обоими способами.

Это можно сделать, если воспользоваться явной реализацией интерфейса. Этим решаются сразу обе проблемы — устраняется потенциальный конфликт имен и запрещается обращаться к методам интерфейса иначе как через ссылку на интерфейс. В этой ситуации явная реализация интерфейса может выглядеть следующим образом:

```
// При явной реализации методов интерфейса можно
// иметь разные варианты метода Draw()
public class Line: Shape, IDraw3D
{
  // Этот метод будет вызываться через ссылку на интерфейс IDraw3D.
  void IDraw3D.Draw()
  {
    Console.WriteLine("Отображение в трехмерном пространстве.");
  }
  // Этот метод будет вызываться через ссылку на объект класса Line.
  public override void Draw()
  {
    Console.WriteLine("Отображение на плоскости");
  }
}
```

При использовании явной реализации интерфейса необходимо знать о некоторых ее тонкостях. Во-первых, уже нельзя задать модификатор области видимости для методов интерфейса:

```
// Так не получится.
public class Line: Shape, IDraw3D
{
  public void IDraw3D.Draw()
  {
    Console.WriteLine("Отображение в трехмерном пространстве.");
  }
}
```

Нетрудно понять, почему принят такой запрет. Единственное, для чего используется явная реализация интерфейса —это привязка методов интерфейса только на его уровне (чтобы к этим методам можно было обратиться только через ссылку на интерфейс). Если же используется модификатор области видимости `public`, то это будет значить, что этот метод переведен в группу открытых методов класса, и по логике реализации таких методов к ним можно обращаться по имени объекта.

Во-вторых, использование явного объявления интерфейса будет единственным способом уберечься от потенциальных конфликтов между именами методов разных интерфейсов. Например, предположим, у вас имеется класс, реализующий все три указанных ниже интерфейса:

```
// Интерфейсы используют методы с одинаковыми именами.
public interface IDraw
{
  void Draw();
```

```
}
public interface IDraw3D
{
  void Draw();
}
public interface IDrawToPrinter
{
  void Draw();
}
```

Если вы хотите создать геометрическую фигуру, поддерживающую все три метода `Draw()`, то необходимо воспользоваться только явной реализацией интерфейса.

```
public class Image: IDraw, IDrawToPrinter, IDraw3D
{
  void IDraw.Draw()
  {
    // Вывод изображения на плоскости.
  }
  void IDrawToPrinter.Draw()
  {
    // Вывод на принтер.
  }
  void IDraw3D.Draw()
  {
    // Вывод трехмерного изображения.
  }
)
```

Создание иерархий интерфейсов

Как уже не раз говорилось, два класса могут вступать между собой в отношения наследования, при которых один класс становится базовым, а другой — производным, наследующим все методы своих предшественников. Интерфейсы, как и классы, тоже могут наследовать все методы своих предшественников. Однако в отличие от классов, интерфейсы не реализуют методы, т.е. не содержат код, определяющий функционирование метода. Когда интерфейс наследует некоторые методы, то он только гарантирует, что эти методы будут переданы в класс для их реализации.

Часто бывает, что базовый интерфейс (интерфейс более высокого уровня в иерархии) определяет общее поведение, в то время как производные интерфейсы вводят специфические особенности. Простая иерархия интерфейсов может выглядеть следующим образом:

```
// Базовый интерфейс
interface IDraw1
(
  void Draw();
}
interface IDraw2: IDraw1
{
  void DrawToPrinter();
}
interface IDraw3: IDraw2
{
  void DrawToMetaFile();
}
```

Если класс должен поддерживать поведение, определенное во всех трех интерфейсах, он должен использовать интерфейс самого нижнего уровня (`IDraw3`). Все методы, определенные в базовых интерфейсах, будут автоматически включены в производные интерфейсы. Например:

```
// Класс поддерживает интерфейсы IDraw1, IDraw2 и IDraw3
public class Image: IDraw3
{
  void IDraw1.Draw()
  {
    // Вывод на экран.
  }
  void IDraw2.DrawToPrinter()
  {
    // Вывод на принтер.
  }
    void IDraw3.DrawToMetafile()
  {
    // Вывод в метафайл.
  }
}
```

Воспользоваться интерфейсами можно следующим образом:

```
public class TheApp
{
  public static void Main()
  {
    Image im = new Image();
    // Получаем ссылку на интерфейс IDraw1
    IDraw1 itfDraw = (IDraw1)im;
    itfDraw.Draw1();
    // А теперь получаем ссылку на интерфейс IDraw3
    if(itfDraw is IDraw3)
    {
      IDraw3 itfDraw3 = (IDraw3)itfDraw;
      itfDraw3.DrawToMetaFile();
      itfDraw3.DrawToPrinter();
    }
  }
}
```

Наследование от нескольких базовых интерфейсов

В отличие от классов, в языке программирования C# можно использовать наследование сразу от нескольких базовых интерфейсов и таким образом сымитировать множественное наследование. Предположим, имеется набор интерфейсов, задающих поведение текстового редактора:

```
interface IPress
{
  void Press();
}
interface ICode
{
  void Code();
}
```

```
// Интерфейс, производный от двух предыдущих.
interface ISave: IPress, ICode
{
  void WriteToFile();
}
```

Если создается класс, который реализует интерфейс `ISave`, то в нем необходимо реализовать все методы каждого из интерфейсов. В данной ситуации таких методов три: `Press()` — сжать, `Code()` — зашифровать и `WriteToFile()` — сохранить.

```
public class Editor; WriteToFile
{
  public Editor(){}
  // Унаследованные члены
  void IPress.Press()
  {
    // Сжатие текста.
  }
  void ICode.Code()
  {
    // Шифрование текста.
  }
  void ISave.WriteToFile()
  {
    // Запись текста в файл.
  }
}
```

Теперь можно воспользоваться любым режимом при сохранении текста.

```
Editor ed = new Editor();
if(ed is ISave)
{
  ((ISave)ed).Code();
  ((ISave)ed).Press();
  ((ISave)ed).WriteToFile();
}
```

Интерфейсные индексаторы

С помощью интерфейсов можно задать и индексатор. Объявление индексатора в интерфейсе имеет следующий формат записи:

```
тип_элемента this[int индекс]{ get; set; }
```

Индексаторы, предназначенные только для чтения или для записи, содержат соответственно методы `get` или `set`.

Ниже показан интерфейса `ISeries`, в который добавлены свойство и индексатор, предназначенный только для чтения элемента ряда.

```
using System;
namespace ИнтерфИндексаторы
{
// Добавление в интерфейс индексатора.
  public interface ISeries
  {
```

```
  // Свойство.
  int next
  {
    get;   // Возвращает следующее число ряда.
    set;   // Устанавливает следующее число ряда.
  }
  // Индексатор.
  int this[int index]
  {
    get;   // Возвращает заданный член ряда.
  }
}

// Реализация интерфейса ISeries.
class ByTwos: ISeries
{
  int val;
  public ByTwos()
  {
    val = 0;
  }
  // Получаем или устанавливаем значение с помощью свойства.
  public int next
  {
    get
    {
      val += 2;
      return val;
    }
    set
    {
      val = value;
    }
  }
  // Получаем значение с помощью индексатора.
  public int this[int index]
  {
    get
    {
      val = 0;
      for(int i=0; i<index; i++)
        val += 2;
      return val;
    }
  }
}
// Демонстрация использования индексатора.
class SeriesDemo
{
  public static void Main()
  {
    ByTwos ob = new ByTwos();
    // Получаем доступ к ряду посредством свойства.
    for (int i=0; i < 5; i++)
      Console.WriteLine("Следующее значение равно " + ob.next);
    Console.WriteLine("\nНачинаем с числа 21");
    ob.next = 21;
    for(int i=0; i < 5; i++)
```

```
        Console.WriteLine("Следующее значение равно " + ob.next);
      Console.WriteLine("\nПереход в исходное состояние.");
      ob.next = 0;
      // Получаем доступ к ряду посредством индексатора.
      for(int i=0; i < 5; i++)
        Console.WriteLine("Следующее значение равно " +
          ob[i]);
      Console.WriteLine("\n\rНажми <Enter> для выхода из программы");
      Console.ReadLine();
    }
  }
}
```

В результате выполнения программы будет сделан следующий вывод:

```
Следующее значение равно 2
Следующее значение равно 4
Следующее значение равно 6
Следующее значение равно 8
Следующее значение равно 10

Начинаем с числа 21
Следующее значение равно 23
Следующее значение равно 25
Следующее значение равно 27
Следующее значение равно 29
Следующее значение равно 31

Переход в исходное состояние.
Следующее значение равно 0
Следующее значение равно 2
Следующее значение равно 4
Следующее значение равно 6
Следующее значение равно 8
```

Использование стандартных интерфейсов

К этому моменту вы уже узнали о назначении интерфейсов, и теперь настало время познакомиться с теми интерфейсами, которые уже встроены в библиотеку базовых классов .NET, и научиться работать с ними. Если присмотреться к библиотеке базовых классов .NET, то можно обнаружить, что множество классов реализует одни и те же стандартные интерфейсы. Разумеется, можно создавать и свои пользовательские типы, поддерживающие те же интерфейсы. То, как это делается, рассмотрим на примере интерфейсов `IEnumerable` и `IEnumerator`.

Создание перечислений (IEnumerable и IEnumerator)

Интерфейсы `IEnumerable` и `IEnumerator` используются во многих стандартных классах. Чтобы в этом убедиться, достаточно посмотреть в справочнике раздел “Enumerable interface”, где будут перечислены все классы, применяющие этот интерфейс, поверьте, их будет не меньше сотни. Широкое использование этих интерфейсов в стандартных классах говорит о том, что они необходимы во многих ситуациях и поэтому вы также должны их применять в своих разработках.

Для того чтобы научиться работать с данными интерфейсами, рассмотрим несложный пример. Положим, создается класс, в котором находится массив некоторых объектов. Пусть это будут сотрудники некоего отдела. Тогда класс `Отдел` можно написать так:

```
struct Сотрудник
{
  string имя;
  string фамилия;
  ushort возраст;
  public Сотрудник(string Имя, string Фамилия, ushort Возраст)
  {
    имя = Имя;
    фамилия = Фамилия;
    возраст = Возраст;
  }
}
class Отдел
{
  Сотрудник[] отдел;
  public Отдел()
  {
    отдел = new Сотрудник[4];
    отдел[0] = new Сотрудник("Иван", "Иванов", 25);
    отдел[1] = new Сотрудник("Петр", "Петров", 31);
    отдел[2] = new Сотрудник("Семен", "Семенов", 28);
    отдел[3] = new Сотрудник("Анна", "Романова", 22);
  }
}
```

С точки зрения использования класса `Отдел` в приложениях было бы очень удобно обращаться ко всем сотрудникам этого класса с помощью конструкции `foreach`:

```
public class Отдел
{
  public static void Main()
  {
    Отдел управления = new Отдел();
    // Пробуем foreach для получения справки по
    // каждому сотруднику в отделе.
    foreach(Сотрудник с in управления)
    {
      Console.WriteLine("Имя: {0}, Фамилия: {1}, Возраст: {2}",
                        с.имя, с.фамилия, с.возраст);
    }
  }
}
```

Однако если запустить этот код на выполнение, компилятор зафиксирует следующую ошибку: "G:\Work\Books\C#\Programs\Интерфейсы\Class1.cs(35): foreach statement cannot operate on variables of type 'Интерфейсы.Отдел' because 'Интерфейсы.Отдел' does not contain a definition for 'GetEnumerator', or it is inaccessible", что можно понимать так: в классе `Отдел` не реализован метод `GetEnumerator()`. Этот метод задается в интерфейсе `IEnumerable`, который находится в пространстве имен `System.Collections`. Чтобы справиться с возникшей проблемой, необходимо переопределить класс `Отдел` таким образом, чтобы он поддерживал и этот интерфейс, и этот метод:

```
// При использовании конструкции foreach необходимо,
// чтобы класс реализовывал интерфейс IEnumerable.
public class Отдел: IEnumerable
  {
// В IEnumerable задан один метод.
   public IEnumerator GetEnumerator()
  {
    ...
  }
}
```

Метод `GetEnumerator()` возвращает еще один интерфейс — `IEnumerator`. Именно этот интерфейс и используется для обращения к членам внутреннего набора объектов. Он также находится в пространстве имен `System.Collections` и включает следующие три метода:

```
// Метод GetEnumerator() возвращает интерфейс IEnumerator
public interface IEnumerator
{
  bool MoveNext();        // Передвинуть внутренний указатель
                          // на одну позицию
  object Current {get;}   // Получить текущий элемент набора
  void Reset();           // Установить указатель на первый элемент.
}
```

Учитывая, что метод `IEnumerable.GetEnumerator()` возвращает интерфейс `IEnumerator`, в класс `Отдел` необходимо добавить следующий код:

```
public class Отдел: IEnumerable, IEnumerator
{
  // Реализация IEnumerable
  public IEnumerator GetEnumerator()
  {
    return (IEnumerator)this; } // Перечислитель для класса Отдел
  }
}
```

Последнее, что необходимо сделать, — наполнить реальным содержанием методы `MoveNext()`, `Current` и `Reset()`. Таким образом, окончательный вариант класса `Отдел`, поддерживающего интерфейсы `Enumerable` и `IEnumerator`, и вся программа, могут выглядеть так, как показано в листинге 6.9.

Листинг 6.9. Использование интерфейсов Enumerable и IEnumerator

```
using System;
using System.Collections;
namespace Интерфейсы
{
  struct Сотрудник
  {
    public string имя;
    public string фамилия;
    public ushort возраст;
    public Сотрудник(string Имя, string Фамилия, ushort Возраст)
    {
      имя = Имя;
```

```
    фамилия = Фамилия;
    возраст = Возраст;
  }
}
class Отдел: IEnumerable, IEnumerator
{
  Сотрудник[] отдел;
  int pos = -1;

  public Отдел()
  {
    отдел = new Сотрудник[4];
    отдел[0] = new Сотрудник("Иван", "Иванов", 25);
    отдел[1] = new Сотрудник("Петр", "Петров", 31);
    отдел[2] = new Сотрудник("Семен", "Семенов", 28);
    отдел[3] = new Сотрудник("Анна", "Романова", 22);
  }

  // Реализация методов интерфейса IEnumerator.
  public bool MoveNext()
  {
    if ( pos < отдел.Length)
    {
      pos++;
      return true;
    }
    else
      return false;
  }
  public void Reset()
  {
    pos = -1;
  }
  public object Current
  {
    get
    {
        return отдел[pos];
    }
  }
  // Реализация метода интерфейса IEnumerable
  public IEnumerator GetEnumerator()
  {
    return (IEnumerator)this;
  }

  [STAThread]
  static void Main(string[] args)
  {
    Отдел управления = new Отдел();
    // Пробуем foreach для получения справки по
    // каждому сотруднику в отделе.
    foreach(Сотрудник с in управления)
    {
      Console.WriteLine("Имя: {0}, Фамилия: {1}, Возраст: {2}",
                        с.имя, с.фамилия, с.возраст);
    }
    Console.WriteLine("Нажми <Enter> для выхода из программы");
```

```
      Console.ReadLine();
    }
  }
}
```

Теперь осталось разобраться с тем, что же получилось в результате предоставленной классу `Отдел` поддержки интерфейсов `IEnumerator` и `IEnumerable`.

Во-первых, теперь с данным классом стало возможно работать посредством синтаксиса `foreach`:

```
foreach(Сотрудник с in управления)
{
  Console.WriteLine("Имя: {0}, Фамилия: {1}, Возраст: {2}",
                    с.имя, с.фамилия, с.возраст);
}
```

Во-вторых, теперь в вашем распоряжении есть новые способы обращения к объектам типа `Сотрудник`, находящимся внутри класса `Отдел`:

```
// Обращаемся к объектам Сотрудник через IEnumerator.
IEnumerator itfEnum;
itfEnum = (IEnumerator)Отдел;
// Устанавливаем указатель на начало списка сотрудников.
ItfEnum.Reset();
// Перемещаем указатель на один шаг вперед.
itfEnum.MoveNext();
```

Создание клонов объектов (ICloneable)

Ранее уже говорилось о том, что в класс `System.Object` включен метод `MemberwiseClone()`. Этот метод используется для *поверхностного копирования* объекта. При этом копирования как такового не происходит, а просто создается еще одна ссылка на область оперативной памяти, занимаемую данным объектом. Разработчики непосредственно не используют метод `MemberwiseClone()`, он вызывается автоматически, когда к объектам ссылочного типа применяется оператор присваивания (`=`), то есть одна ссылка начинает указывать на ту же область оперативной памяти, что и другая.

Предположим, имеется класс `Точка`:

```
public class Point
{
  public int x. y:
  public Точка(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  // Замещаем Object.ToString()
  public override string ToString()
  (
    return "X:  " + x + "   Y:  " + y;
  )
}
```

Класс `Точка` относится к ссылочным типам. Если применить к объектам типа `Точка` оператор присваивания (`=`), то при этом будет вызван метод `MemberwiseClone()`, который создаст еще одну ссылку на область, занимаемую объектом в оперативной памяти. Но очень часто

бывает нужно создавать не дополнительные ссылки, а реальные копии объекта, т.е. произвести *глубокое копирование*. Для того чтобы можно было применять глубокое копирование к объектам класса посредством стандартных методов, класс должен реализовывать интерфейс `ICloneable`.

В интерфейсе `ICloneable` предусмотрен единственный метод — `Clone()`. Реализация данного метода, конечно же, зависит от того, какие члены определены в исходном классе. Однако смысл работы этого метода для всех классов будет одним и тем же — необходимо создать новый объект, у которого всем членам будут присвоены значения соответствующих переменных исходного объекта. Давайте создадим клон объекта `Точка`.

```
// Реализуем поддержку глубокого копирования
// с помощью интерфейса ICloneable.
public class Точка: ICloneable
{
  public int x, y;      // Состояние объекта.
  public Точка(int x, int y) // Конструктор.
  {
    this.x = x;
    this.y = y;
  }
  // Реализуем метод интерфейса ICloneable.
  public object Clone()
  {
    return new Точка(this.x, this.y)
  }
  public override string ToString()
  {
    return "X:  " + x + " Y:  " + y;
  )
}
```

Теперь можно создавать полностью независимые от исходного объекта копии объектов типа `Точка`. Выглядеть это может так:

```
// Обратите внимание, что Clone() тип Object. Чтобы преобразовать его
// в нужный тип делается явное преобразование типов.
Точка точка1 = new Точка (20, 20);
Точка точка2 = (Точка)точка1.Clone();
```

Но будьте внимательны, если ваш класс является производным от одного из многочисленных стандартных встроенных классов из библиотеки базовых классов языка C#, то вполне возможно, что этот интерфейс в нем уже реализован. Это легко выяснить, обратившись к справочной документации Visual Studio .NET.

Если созданный вами класс является наследником одного из встроенных классов, то в нем уже могут быть реализованы некоторые интерфейсы.

Создание сравниваемых объектов (IComparable)

Еще один распространенный интерфейс `IComparable`, также заданный в пространстве имен `System`, позволяет производить сортировку объектов, основываясь на специально определенном внутреннем ключе. Формальное определение этого интерфейса выглядит следующим образом:

```
// Интерфейс позволяет определять место объекта
// среди аналогичных объектов
interface Icomparable
{
  int CompareTo ( object o);
}
```

Давайте используем уже рассмотренный ранее класс `Отдел` и попробуем применить статический метод `Sort()`, который находится в классе `System.Array`, чтобы упорядочить по возрастанию массив сотрудников, например, по возрасту. Понятно, что так просто это сделать невозможно, и утверждение

```
Array.Sort(управления.отдел);
```

ни к чему не приведет.

Если попробовать запустить этот код на выполнение, будет сгенерировано исключение `System.InvalidOperationException` со следующим комментарием: "Specified IComparer threw an exception." (Используемый `IComparable` сгенерировал исключение). Таким образом, чтобы можно было стандартным способом производить сортировку пользовательских объектов, они должны реализовывать интерфейс `IComparable`. Поскольку этот интерфейс состоит из единственного метода `CompareTo()`, то следует грамотно реализовать этот метод. При этом необходимо принять решение, по значению какой внутренней переменной будет производиться сортировка. Для типа `Сотрудник` будет использоваться сортировка по возрасту.

```
// Такая реализация метода СотрагеТо() позволит сортировать
// сотрудников по возрасту.
class Сотрудник: IComparable
  {
    public string имя;
    public string фамилия;
    public ushort возраст;
    public Сотрудник(string Имя, string Фамилия, ushort Возраст)
    {
      имя = Имя;
      фамилия = Фамилия;
      возраст = Возраст;
    }

    public int CompareTo(object obj)
    {
      Сотрудник сотр = (Сотрудник)obj;
      if ( this.возраст > сотр.возраст )
        return 1;
      if ( this.возраст < сотр.возраст )
        return -1;
      else
        return 0;
    }

  }
```

Обратите внимание, что когда в текстовом редакторе будет введено имя `IComparable` в заголовке класса `Сотрудник`, появится напоминание о том, что автоматически будет создан шаблон метода `CompareTo()`, если нажать клавишу <Tab>. Можете воспользоваться этой возможностью и не вводить весь код метода вручную.

Как видно из этого кода, метод `CompareTo()` сравнивает значение поля `возраст` для текущего объекта (того, для которого вызван этот метод) со значением поля `возраст` для принимаемого объекта (того, который передан этому методу в качестве входящего параметра). В зависимости от результатов сравнения выдается одно из трех возможных значений. Что может означать каждое из этих значений, показано в таблице 6.2.

Таблица 6.2. Значения, возвращаемые методом CompareTo()

Значение	*Описание*
Любое число меньше нуля	Значение у текущего объекта меньше, чем у принимаемого в качестве параметра
Нуль	Значения у текущего и принимаемого объекта равны
Любое число больше нуля	Значение у текущего объекта больше, чем у принимаемого

Теперь, когда интерфейс `IComparable` и метод `CompareTo()` делают то, что от них требуется, можно производить сортировку объектов типа `Сотрудник` посредством стандартного метода `Sort()`, как показано ниже.

```
static void Main(string[] args)
{
  Отдел управления = new Отдел();
  // Пробуем foreach для получения справки по
  // каждому сотруднику в отделе.
  Console.WriteLine("В порядке записи");
  foreach(Сотрудник с in управления)
  {
    Console.WriteLine("Сотрудник: {0} {1}\t Возраст: {2}",
                      с.имя, с.фамилия, с.возраст);
  }
  управления.Reset();
  Array.Sort(управления.отдел);
  Console.WriteLine("Отсортированные по возрасту");
  foreach(Сотрудник с in управления)
  {
    Console.WriteLine("Сотрудник: {0} {1}\t Возраст: {2}",
                      с.имя, с.фамилия, с.возраст);
  }
  Console.WriteLine("\n\rНажми <Enter> для выхода из программы");
  Console.ReadLine();
}
```

Если у нескольких объектов в массиве будет одинаковое значение поля, по которому производится сортировка, то они будут расставлены в том же порядке относительно друг друга, в котором поступили на операцию сортировки.

Библиотека System.Collections

Наиболее простым вариантом работы с набором элементов в языке C# является использование массива `System.Array`. Он уже обладает всеми полезными встроенными функциями, дающими возможность производить операции сортировки, клонирования, перечисления и расстановки элементов в обратном порядке. Однако создатели библиотеки базовых классов языка C# подготовили большое количество встроенных типов, которые позволят сэкономить массу времени при решении часто встречающихся задач. В этом разделе вы познакомитесь со

встроенными типами, определенными в пространстве имен System.Collections. Как можно понять из названия System.Collections, все эти типы предназначены для работы с наборами элементов.

Необходимо сказать, что в пространстве имен System.Collections задан набор стандартных интерфейсов, большая часть которых описана в этой главе. Кроме того, эти же интерфейсы наследуются в большинстве классов пространства имен System.Collections. Перечень всех интерфейсов пространства имен System.Collections представлен в таблице 6.3.

Таблица 6.3. Интерфейсы пространства имен System.Collections

Интерфейс	*Описание*
ICollection	Определяет размер, перечисления и методы синхронизации для всех наборов
IComparer	Содержит методы для сравнения двух объектов
IDictionary	Представляет набор пар ключ-значение
IDictionaryEnumerator	Перечисление элементов словаря
IEnumerable	Возвращает интерфейс IEnumerator для заданного объекта
IEnumerator	Поддерживает простые перемещения в наборе, в основном для конструкции foreach
IHashCodeProvider	Расчет хэш-кода для объекта с помощью пользовательского алгоритма хэширования
IList	Представляет набор методов, посредством которых обеспечивается доступ по индексу для добавления и удаления объектов

Все классы, определенные в пространстве имен System.Collections, представлены в таблице 6.4.

Таблица 6.4. Классы библиотеки System.Collections

Класс	*Описание*
ArrayList	Реализует интерфейс IList для динамических массивов
BitArray	Управление компактным массивом битовых значений, которые представлены как булевы типы, где для true используется значение 1, а для false — значение 0
CaseInsensitiveComparer	Сравнение двух объектов, игнорируется регистр букв
CaseInsensitiveHashCodeProvider	Возврат хэш-кода для объекта с помощью алгоритма, в котором игнорируется регистр букв
CollectionBase	Обеспечение абстрактного класса для строго типизированных наборов
Comparer	Сравнение двух объектов
DictionaryBase	Обеспечение абстрактного класса для строго типизированных наборов и пар ключ-значение
Hashtable	Представление набора пар ключ-значение, организация которого базируется на хэш-коде ключа
Queue	Представление набора объектов типа first-in, first-out (первый вошел, первый вышел)

Окончание табл. 6.4

Класс	*Описание*
`ReadOnlyCollectionBase`	Обеспечение абстрактного класса для строго типизированных наборов только для чтения
`SortedList`	Представление набора пар ключ-значение, которые сортируются по ключу и к которым предоставляется доступ по ключу и по индексу
`Stack`	Представляет простой набор объектов last-in-first-out (последний вошел, первый вышел)

Библиотека System.Collections.Specialized

Если ни один из классов, представленных в пространстве имен `System.Collections`, вам не подходит, то их можно поискать в библиотеке `System.Collections.Specialized`. В этой библиотеке находятся типы для работы с наборами элементов. Как следует из названия библиотеки, эти типы предназначены для специальных случаев. В качестве примера можно назвать типы `StringDictionary` и `ListDictionary`, которые специальным образом реализуют интерфейс `IDictionary`. Поэтому прежде чем заняться собственной реализацией классов для работы с коллекциями элементов, загляните в электронную документацию Visual Studio .NET и познакомьтесь с уже разработанными классами. Вполне вероятно, что значительная часть вашей работы уже сделана.

Применение ArrayList

При ближайшем рассмотрении классов, находящихся в библиотеке `System.Collections`, становится понятным, что все они выполняют похожие действия и реализуют одни и те же интерфейсы. Поэтому, вместо того чтобы рассматривать реализацию каждого из классов, подробно проанализируем функционирование лишь одного из них — `System.Collections.ArrayList`. После этого разобраться в использовании остальных классов не составит особого труда.

В этой главе будет рассматриваться класс `Отдел`, который представляет собой набор объектов типа `Сотрудник`. Ранее класс `Отдел` был реализован на основе простого массива — то есть в качестве базового был использован класс `System.Array`. Поскольку у обычного массива возможностей не так много, то пришлось создавать значительное количество дополнительного кода для того, чтобы можно было обращаться к элементам класса `Отдел` из внешнего мира. Кроме того, в классе `Отдел` было серьезное ограничение — фиксированный размер использованного массива для сотрудников.

Гораздо удобнее и эффективнее было бы не делать лишней работы, а воспользоваться уже готовым к употреблению классом `System.Collections.ArrayList`, в котором многие нужные в большинстве случаев возможности реализованы изначально. Этот класс также предоставляет в ваше распоряжение готовые методы для вставки, удаления и нумерации внутренних элементов.

Для того чтобы воспользоваться возможностями `ArrayList`, применим не классическое наследование, а модель включения, когда класс `ArrayList` будет вложен внутрь класса `Отдел`. Фактически единственное, что необходимо сделать, — реализовать в классе `Отдел` открытые методы, которые будут передавать вызовы на выполнение различных действий внутреннему классу `списокСотрудников`, производному от `ArrayList`. Выглядеть это может так:

Листинг 6.10. Использование класса ArrayList с помощью включения

```
using System;
using System.Collections;
using System.Collections.Specialized;

namespace Class_ArrayList
{
  public class Сотрудник
  {
    public string имя;
    public string фамилия;
    public ushort возраст;
    public Сотрудник(string Имя, string Фамилия, ushort Возраст)
    {
      имя = Имя;
      фамилия = Фамилия;
      возраст = Возраст;
    }
  }

  class Отдел
  {
    // Больше не нужно реализовывать интерфейс IEnumerator
    // - все уже сделано в ArrayList.
    public ArrayList Сотрудники;
    public Отдел()
    {
      Сотрудники = new ArrayList();
    }

    // Реализуем методы и свойства для работы с сотрудниками.
    public void НовыйСотрудник(Сотрудник с)
    {
      Сотрудники.Add(с);
    }

    public void УдалитьСотрудника(int номер)
    {
      Сотрудники.RemoveAt(номер);
    }

    public int КоличествоСотрудников
    {
      get
      {
        return Сотрудники.Count;
      }
    }

    public bool ЕстьВШтате(Сотрудник с)
    {
      return Сотрудники.Contains(с);
    }

    // Важно! Все связанное с реализацией IEnumerator,
    // перенаправляем в класс Сотрудники (тип ArayList).
    public IEnumerator GetEnumerator()
```

```
    {
      return Сотрудники.GetEnumerator();
    }

    [STAThread]
    static void Main(string[] args)
    {
      Отдел управления = new Отдел();
      управления.НовыйСотрудник(new Сотрудник("Иван","Иванов",38));
      управления.НовыйСотрудник(new Сотрудник("Петр","Петров",25));
      управления.НовыйСотрудник(new Сотрудник("Семен","Семенов",31));
      управления.НовыйСотрудник(new Сотрудник("Анна","Романова",22));

      Console.WriteLine("В отделе управления {0} сотрудников",
                        управления.КоличествоСотрудников);
      foreach (Сотрудник с in управления)
      {
        Console.WriteLine("{0} {1}", с.имя, с.фамилия);
      }
      Console.WriteLine("\n\rНажми <Enter> для выхода из программы");
      Console.ReadLine();
    }
  }
}
```

Результат работы этой программы показан на рис. 6.7.

***Рис. 6.7.** Результат работы программы*

Теперь рассмотрим вопрос, почему не рекомендуется, чтобы класс `Отдел` был унаследован от класса `ArrayList`? Зачем потребовалось все усложнять и создавать внутри класса `Отдел` вспомогательный класс и дополнительные методы? Ответ очень прост — класс `ArrayList` сам по себе работает с любыми объектами. Это значит, что при использовании классического наследования класс `Объект` можно было бы заполнять объектами абсолютно любых типов языка C#, что может привести к ошибкам. Чтобы этого не произошло, нужно запретить классу `Отдел` содержать какие-либо объекты кроме объектов типа `Сотрудник`, почему и была применена модель включения. Средством контроля над поступлениями в класс `Отдел` стали методы этого класса, хотя ничто не запрещает сделать и так, как в листинге 6.11.

Листинг 6.11. Использование класса ArrayList с помощью наследования

```
using System;
using System.Collections;

namespace Class_ArrayList1
{
  public class Сотрудник
  {
    public string имя;
    public string фамилия;
```

```
    public ushort возраст;
    public Сотрудник(string Имя, string Фамилия, ushort Возраст)
    {
      имя = Имя;
      фамилия = Фамилия;
      возраст = Возраст;
    }
  }

  class Отдел: ArrayList
  {
    [STAThread]
    static void Main(string[] args)
    {
      Отдел управления = new Отдел();
      управления.Add(new Сотрудник("Иван","Иванов",38));
      управления.Add(new Сотрудник("Петр","Петров",25));
      управления.Add(new Сотрудник("Семен","Семенов",31));
      управления.Add(new Сотрудник("Анна","Романова",22));
      Console.WriteLine("В отделе управления {0} сотрудников",
управления.Count);
      foreach (Сотрудник с in управления)
      {
        Console.WriteLine("{0} {1}", с.имя, с.фамилия);
      }
      Console.WriteLine("\n\rНажми <Enter> для выхода из программы");
      Console.ReadLine();
    }
  }
}
```

Резюме

В данной главе кратко описаны основные понятия объектно-ориентированного программирования и способы их реализации с помощью классов и объектов. Рассмотрены структура класса и элементы класса: поля, методы и свойства. Также приведена информация об интерфейсах и их использовании для более гибкого построения логики программы. В библиотеках базовых классов .NET определено большое количество готовых стандартных интерфейсов. В этой главе были представлены наиболее важные интерфейсы, определенные в пространстве имен `System.Collections`. Вы можете применять их в своих пользовательских классах для того, чтобы реализовать в них поддержку необходимых вам возможностей, таких как клонирование объектов, сортировка и нумерация.

В дальнейшем будут раскрыты существенные детали реализации элементов класса и показаны конкретные примеры их применения.

Контрольные вопросы

1. Назовите три основных принципа объектно-ориентированного программирования. В чем заключаются преимущества их использования?
2. Дайте определение понятиям “класс” и “объект”?
3. Что такое поле? Как поле связано со свойством?
4. Что такое метод?
5. Для чего нужны перегрузка и переопределение методов?

6. Какие области видимости для полей и методов разрешены в классе?
7. Что такое конструктор и деструктор?
8. Какую область видимости должен иметь конструктор?
9. Что такое интерфейс? Как вы понимаете его основное назначение?
10. Что такое включение и чем оно отличается от наследования?
11. Как с помощью интерфейсов создать множественное наследование?