专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 13 章

深入学习 C#

根据第Ⅱ部分创建的 UML 模型，我们几乎能马上使用 C#开发学生选课系统(SRS)。在开始 SRS 的具体编码前，本书将会介绍一些 C#语言的特征，在构建 SRS 时会用到这些特征。

短短一章篇幅不可能介绍 C#语言的所有特性。C#是一种极其丰富的语言，大多数优秀的 C#参考书至少得有几百页的篇幅。本书的目的不是重复 C#参考书的工作，而是向您展示如何在对象模型和 C#代码间的鸿沟上架起一座桥梁，补充参考书的不足之处，这正是其他参考书较少涉及的领域。

请记住这一点，本章将有选择性地介绍 C#语言的几个方面：即那些最能帮助理解第 14~16 章中 SRS 编码示例的知识。而且在学习完本章后，您会对 C#语言有更深入的了解。

提示

即使您已经用 C#编写过程序，并因此觉得已经掌握了这门语言的语法，但本书还是建议您在开始学习第 14 章的内容前先浏览本章的内容，因为本章的很多内容和 SRS 的实现有关。

本章主要介绍以下主题：

- C#名称空间(namespace)的概念——如何定义名称空间以及使用名称空间的原因
- string(字符串)的对象特性以及用来操作字符串的方法和属性
- 预定义的 Object 类，它是 C#所有类型的基类
- 使用关键词 this，从而在对象的方法中自我引用
- 数组的对象特性以及用来操作数组的方法和属性
- List 和 Dictionary 类，它们是.NET Framework 通用集合类
- C#中对象标识符的特性，如何找出对象属于哪个类，如何检测两个 C#对象是否等价
- Main 方法和 Base 关键字的重要变化
- 如何删除动态创建的对象，从而回收它所占用的内存，以及公共语言运行库(Common Language Runtime，CLR)的垃圾回收器(garbage collector)在回收过程中

专业图书网上第一专营店 www.china-pub.com 独家提供样章

扮演的角色
- .NET Framework 语言结构(属性)

本章还将回顾前几章介绍的内容，使您对其有更深入的了解。

13.1 名称空间

在本书第 I 部分和第 II 部分的示例和接下来的大部分示例中，在程序的顶部都会放置一条 using 语句(更正式的说法是 using 指令)，以通过简单名称(simple names)访问特定名称空间(namespace)的元素。第 1 章曾简要提到过，名称空间是相关编程元素的逻辑组。.NET Framework 的库太大，通过名称空间可将其划分为一些更方便管理的子库。

简单名称就是类声明中显示的类名，例如：

```
// 这个类的简单名称是 Student。
public class Student {
  // 细节从略。
}
```

当在名称空间中放入一个类时，类的名称会发生改变，因为类需要加上名称空间，以形成完全限定名称。例如，在本书前面提到过，因为 String 类包含在 System 的名称空间中，所以其完全限定名称是 System.String，而该类的简单名称是 String。

可以想象，属于不同名称空间 A 和 B 的两个类，能够拥有相同的简单名称 X。打个比方，只要位于不同的 Windows 文件夹(如 C:\MyDocs 和 D:\Stuff)，就能用相同的名称(如 xyz.doc)创建两个不同的 Microsoft Word 文档。这两个类的完全限定名称是 A.X 和 B.X，从而保证了唯一性，就像 Word 文档一样，两个具有相同名称的文档会有不同的完全限定名称，如 C:\MyDocs\xyz.doc 和 D:\Stuff\xyz.doc。

为了确保在任何情况下编译器都能明白所要使用的类，可以在程序中始终使用类的完全限定名称。

```
// 注意，本程序没有使用 using 指令。

public class SimpleProgram3
{
   static void Main() {
     System.String name = "Jackson";
     System.Console.WriteLine("The name is " + name);
   }
}
```

在程序中必须输入每个名称空间成员的完全限定名称会很麻烦，而且会使代码的可读性降低。幸运的是，C#语言提供了 using 指令，从而能够使用简单名称来方便地访问名称空间。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

前面已经看到过，如果在文件中需要访问某个名称空间的成员，则需要在源文件的最顶端加入该名称空间的 using 指令，这样就能在源文件中随时用简单名称来引用这个类。

```
// 计划使用 System 名称空间中的 Console 类。
using System;

// 计划在 BarStuff 名称空间中使用 Foo 类。
using BarStuff;

public class SimpleProgram3
{
  static void Main() {
    // 通过简单名称引用 Foo 和 Console 类。
    Foo x = new Foo();
    Console.WriteLine("A Foo is born every minute!");
  }
}
```

编译器将依次搜索每个指定的名称空间，从而保证它能找到 Console 和 Foo 的声明。

如果在代码中引用的类名称不止在一个名称空间出现，则会产生一个小问题。例如，假设想要在相同的程序中使用两个不同的类(但名称都是 Course)，一个在 SRS 的名称空间中定义，另一个在 ObjectStart 名称空间中定义。即使提供了这两个名称空间的 using 指令，也仍然需要提供 Course 类的完全限定名称来避免混淆：

```
// Example.cs

using ObjectStart;
using SRS;

public class Example
{
  static void Main() {
    // 在这里使用 SRS 的 Course 类……
    SRS.Course math = new SRS.Course();

    // ……在这里使用 ObjectStart 的 Course 类。
    ObjectStart.Course english = new ObjectStart.Course();
   // 等等。
  }
}
```

因此，在这个特殊的示例中，没有为 SRS 或 ObjectStart 名称空间提供 using 指令。

当然，如果想要使用 SRS 或 ObjectStart 名称空间中唯一命名的类，则不需要完全限定名称，使用 using 指令即可。例如，这里不仅需要使用上例中两个不同版本的 Course，

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

还要使用仅在 SRS 名称空间中存在的 Student 对象和仅在 ObjectStart 名称空间中存在的 Professor 对象，如图 13-1 所示。

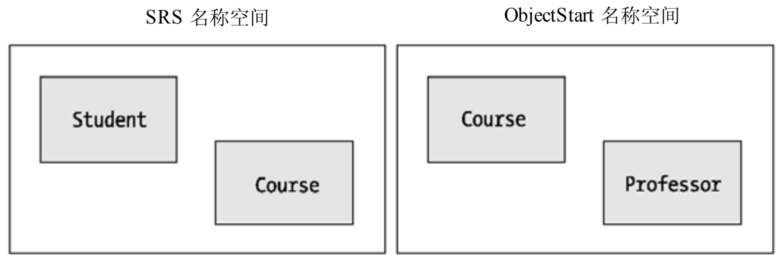

图 13-1 Course 存在于两个名称空间中，但 Professor 和 Student 不是

这里需要再次使用 Course 的完全限定名称，但如果在程序中包含了 using 指令，则不需要完全限定 Student 或 Professor 的名称，如下列代码所示：

```
using SRS;
using ObjectStart;
using System;

public class Example
{
  static void Main() {
    // 仍然需要在程序中使用 Course 的完全限定名称……
    SRS.Course math = new SRS.Course();
    Obj ectStart.Course english = new ObjectStart.Course();

    // ……因为在代码顶部有 using 指令，所以可以使用 Professor 的简单名称字符串。
    string name = "Dinesh Prabhu";
    math.Professor = name;

    // Student 类亦是如此。
    Student s = new Student();

    // 等等。
  }
}
```

在第 14~16 章开发 SRS 应用程序的过程中，本书将使用 5 个.NET Framework 名称空间中的预定义类。

- System 名称空间，包括 String、Console 和 Array 类
- System.Collections.Generic 名称空间，包括 List 和 Dictionary 集合类
- System.IO 名称空间，包括 FileStream、StreamReader 和 StreamWriter 类，在第 15 章中将用它们向文件或从文件中保存和恢复 SRS 使用的数据

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

- System.Windows.Forms 和 System.Drawing 名称空间，包括将在第 16 章用来为 SRS 创建 GUI 前台的 GUI 类和支持类。

13.1.1 程序员自定义的名称空间

C#语言能够让我们创建自己的名称空间。这样做的原因和 Microsoft 创建/设计.NET Framework 的原因相同：

- 为了在逻辑上划分类以方便重用。例如，一名火箭科学家可能会将所有与行星相关的类放入 Planets 名称空间中，以方便天文学家重用；将所有与火箭设计相关的类放入 Rockets 名称空间中，以方便火箭制造商重用。
- 为用户自定义的类确保唯一的完全限定名称。例如，如果想要设计一个类，它的名称和.NET 的某个名称空间中的类名相同(如 Console 类)，则可以在同一个程序中使用 System.Console 和 ObjectStart.Console 类。

为了将某个特定的编程元素(如类或接口)分配到名称空间中，需要在程序顶部输入 namespace 关键字，后面跟上自定义名称空间的名称(遵循 Pascal 大小写约定)，再加上括号{...}，其中包含 1 个或多个类、接口或其他需要包含在特定名称空间中的元素的声明。

例如，如果正在设计管理宠物店的应用程序，则可能需要创建名为 PetStore 的名称空间，以包含表示不同宠物类型的类。为了包含表示宠物鼠的类，这里会创建名为 PetRat.cs 的源代码文件，如下所示：

```
// PetRat.cs

// 其他名称空间所需的 using 指令放在这里。本例使用了两个类：Console 和 Seed，它们分
// 别位于 System 和 AnimalFood 名称空间中。
using System;       // 标准的.NET Framework 名称空间
using AnimalFood;  // 用户自定义的名称空间，在其他位置定义

// 只要在名称空间的声明中使用 PetStore，就能创建 PetStore 名称空间!
namespace PetStore
{
  // 所有在该名称空间中声明的内容都将成为 PetStore 名称空间的一部分。

  // PetRat 类现已成为 PetStore 名称空间的一部分，其完全限定名称是
  // PetStore.PetRat。
  public class PetRat
  {
    // 字段。
    private string name;
    private string coatColor;

    // Seed 是 AnimalFood 名称空间中的类，但是因为在上面包含了 using AnimalFood
    // 指令，所以可以通过其简单名称来引用这个 Seed 类。
    Seed favoriteSeedType;
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        public void DisplayRatInfo() {
          // 上面的 using System 指令能够通过简单名称来引用 Console 类。
          Console.WriteLine("Rat's Name: " + name);
          Console.WriteLine("Coat Color: " + coatColor);
          Console.WriteLine("Favorite Seed Type: " +
              FavoriteSeedType.Name);
          // 等等。
        }
      }

      // 如果需要，可以在此插入其他 PetStore 的类/接口，或放入单独的(预定义的)源文件中。
    } // 名称空间声明结束
```

PetRat 类将被指派给 PetStore 名称空间。

如果还有一个类(如 Tarantula)想要包含在 PetStore 的名称空间中，则可以在相同的文件中完成该任务。然而，名称空间在多个文件中定义也是很常见的。例如，如果想要向 PetStore 名称空间添加表示 Tarantula 的类，则可以在名为 Tarantula.cs 的单独文件中实现这个类，如下所示：

```
// Tarantula.cs

// Tarantula 类需要引用的其他名称空间的 using 指令放在此处，细节从略。

// 和 PetRat 类一样，这里将 Tarantula 类添加到了同一个名称空间中。
namespace PetStore
{
   // Tarantula 类现已成为 PetStore 名称空间的一部分，它的完全限定名称是
   // PetStore.Tarantula。
   public class Tarantula
   {
   // 细节从略。
   }
}
```

然后，如果想要在客户代码中通过简单名称来访问 PetRat 或 Tarantula 类，则可以在代码的顶端插入 using PetStore;指令：

```
// NamespaceDemo.cs

using PetStore;
// Example 程序所需的任何其他 using 指令都放在这里……

public class NamespaceDemo
{
  static void Main() {
    // 这里能够使用简单名称 PetRat……
    PetRat r = new PetRat();
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        r.Name = "Baby Grode";

        // ……这里能使用简单名称 Tarantula。
        Tarantula t = new Tarantula();
        t.Name = "Fuzzy";

        // 等等。
    }
}
```

包含名称空间定义的源代码的编译器命令不会发生变化。如果 NamespaceDemo.cs、PetRat.cs 和 Tarantula.cs 文件位于同一个目录，则可使用以下命令编译应用程序：

```
csc NamespaceDemo.cs PetRat.cs Tarantula.cs
```

当编译需要大量源文件的应用程序或者重用来自现有名称空间的文件时，通常可以将源文件放入不同的文件夹或目录中。如果 PetRat.cs 和 Tarantula.cs 文件位于包含 Namespace.cs 文件目录中的子文件 PetStore 中，则可以使用以下编译命令：

```
csc NamespaceDemo.cs PetStore\PetRat.cs PetStore\Tarantula.cs
```

13.1.2 全局名称空间

最后需要指出的是，如果没有在源文件中包含显式的名称空间指令，则在源文件中定义的类或接口将会分配给无名称的全局名称空间。如果类 A 和 B 位于全局名称空间：

- A 能够使用 B 的简单名称引用 B
- B 能够使用 A 的简单名称引用 A
- 两个类都能正确编译

因为这两个类都在全局名称空间中，所以不需要使用 using 指令来找到对方。当在第 14 章中开始编写 SRS 相关类时，为了简化内容，本书将不使用用户自定义的类来保存 SRS 的类，而是将它们放入(默认的)全局名称空间。因此，所有 SRS 类都互相认识，从而能够在不包含 using 指令的情况下限定简单名称 Student 和 Course，如下代码所示：

```
public class SRS
{
  static void Main() {
    Student s = new Student(); // 使用简单名称。
    Course c = new Course();   // 同上。
    // 等等
  }
}
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.2 作为对象的字符串

在第 1 章中，我们介绍了包括 string 类型在内的一些预定义的 C#类型。当时我们只是暗示，而没有明确说明字符串就是对象。第 1 章介绍了创建和使用字符串的基本知识，本节将回顾这些知识，并将深入介绍字符串的对象特性。

13.2.1 string 别名

关键字 string 实际上是 System 名称空间中定义的 String 类的别名。当声明一个 string 变量并赋值时：

```
string name = "Jackson";
```

实际上正在实例化一个 System.String 类的对象/实例并使用 Jackson 初始化该对象。

- 对 C#编译器来说，string 和 System.String 表达式在语法上等价。string 对象能够访问在 System.String 类中声明的方法、属性和构造函数。
- 虽然不会看到使用 new 操作符显式调用 System.String 类的构造函数，但还是可以通过简写的方式来完成调用。

要在程序中引用 string 类型，有以下几种选择：

- 如果在程序顶部包含 using System;指令，则可以引用类的简单名称 String(大写 S)：

```
using System;

public class Foo
{
  static void Main() {
   String s = "Whee!";
  }
}
```

- 同样可以使用完全限定的名称 System.String，这样无需在程序的顶部使用 using System;指令。

```
// 不需要 using 指令!

public class Foo
{
  static void Main() {
    System.String s = "Whee!";
  }
}
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章

> **注意**
>
> 如果想要通过简单名称访问其他 System 名称空间元素(如 Console 类)，则必需使用 using System;指令。

- C#推荐的方法是使用 string(小写 s)。它的语法比 System.String 简短，而且还允许在程序顶部省去 using System;指令：

```
// 不需要 using 指令!

public class Foo
{
  static void Main() {
    string s = "Whee!";
  }
}
```

考虑到 string 别名的方便性，因此不必在 C#应用程序中使用 String 的简单或完全限定的大小写形式。

> **注意**
>
> 第 1 章讨论的预定义简单形式——bool、float、int、double、long、char 等——实际上都是 System 名称空间中定义的类型的别名。

13.2.2 创建 String 实例

如第 1 章所述，通过声明类型为 string 的变量，就能创建一个字符串的实例，并能使用任何有效的字符串表达式作为其初始值：

```
string name = "Chen";
```

或

```
string name = student.Name;
```

或

```
string name = department.FindChairperson().Name;
```

字符串还可以通过使用重载的 String 类构造函数和 new 操作符来创建，一些常用的 String 构造函数的头如下：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public String(char[] characters)
public String(char c, int count)
public String(char[] characters, int start, int length)
```

以下是使用 String 类构造函数的示例：

```
char[] chars = { 'C', 'h', 'e', 'n' };
string name = new String(chars);
```

13.2.3 @字符

第 1 章曾介绍过，一些转义字符(escape character)可用来表示特殊字符，如跳格(\t)、换行(\n)或反斜杠(\\)。例如，如果想要定义文件路径：

```
C:\MyFolder\MySubFolder\NewFile.txt
```

就可以通过使用\\(反斜杠)转义字符，如下所示：

```
string filePath = C:\\MyFolder\\MySubFolder\\NewFile.txt";
```

C#还提供了另外一种不使用转义字符来声明类似字符串文本的方法。如果在字符串开始的双引号前加上@，则字符串就能从源代码文件中读入，允许字符串跨多行并能保持所有换行、跳格、反斜杠等字符不变。

```
string filePath = @"C:\MyFolder\MySubFolder\NewFile.txt";
```

13.2.4 特殊的 string 操作符

第 1 章介绍过，加号操作符(+)可以连接字符串值：

```
string x = "foo";
string y = "bar";
string z = x + y + "!"; // z now has the value "foobar!"
```

String 类还提供了= =和! =操作符，以比较两个字符串值是否相等。以下代码说明如何使用这些操作符：

```
string name = "Mary Jones";
if (name == "Cynthia Coleman") {
  Console.WriteLine("This is Cynthia Coleman");
}
else {
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
  Console.WriteLine("Hey! What happened to Cynthia?");
}
```

上面代码的执行结果如下：

```
Hey! What happened to Cynthia?
```

提示

您将在后面的章节中发现使用== 和!=来比较对象时，结果会有些区别。

13.2.5 String 属性

String 类还定义了两个有用的属性。

Length 属性将返回 String 中的字符数，包括空格：

```
string sentence = "How long am I?";
Console.WriteLine("Length = " + sentence.Length);
```

上面代码的输出如下：

```
Length = 14
```

String 类的另一个有用特性是称为 Chars 的特殊属性，它能作为索引器，以根据字符的位置或索引(index)访问字符串中的单个字符。使用 Chars 属性的方法是在括号内输入字符的位置，这和访问数组元素的语法类似：

```
String str = "Tom Servo";
Console.WriteLine("The first character is " + str[0]);
```

以上代码的结果如下：

```
The first character is Ts
```

13.2.6 String 方法

除了上面讨论过的运算符和属性之外，每个 string 对象还能访问由 String 类声明的方法。本节将讨论其中比较常用的方法。如果想要完全了解 String 类定义的方法，请参阅 MSDN 网站上的.NET Framework Class Library Reference。

- public bool StartsWith(string str)：如果字符串是以括号中的字符串表达式作为起

专业图书网上第一专营店 www.china-pub.com 独家提供样章

始字符的话，则返回 true；否则返回 false。

```
string string1 = "foobar";
string string2 = "foo";

// 结果为 true。
if (string1.StartsWith(string2))...
```

- public bool EndsWith(string str)：如果字符串是以括号中的字符串表达式作为结尾字符的话，则返回 true；否则返回 false。

```
string string1 = "foobar";

//结果为 true。
if (string1.EndsWith("bar"))...
```

- public int IndexOf(string str)：在字符串中查找作为参数的字符串表达式，如果找到则返回一个正整数值，说明参数在源字符串中的位置(从 0 开始计数)；如果未找到则返回 – 1：

```
string string1 = "foobar";
int i = string1.IndexOf("bar"); // i 将等于 3

string string2 = "cat";
int j = string1.IndexOf(string2); // j 将等于-1，因为在 foobar 中没有找到
                                  // 参数值 cat。
```

- public string Replace(char old, char new)：创建新的 string 对象，用新字符替换其中所有的旧字符，原来的字符串不受影响：

```
string string1 = "o1o2o3o4";
// 注意在字符两边使用单引号。
string p = string1.Replace('o', 'x'); // p 现在等于"x1x2x3x4"，而 string1
                                      // 仍然是"o1o2o3o4"
```

- public string Replace(string old, string new)：这是 Replace 方法的重载版本，用新字符串替换旧字符串，注意这两个字符串长度不必相同：

```
string string1 = "foobar";
string p = string1.Replace("foo", "candy"); // p 变成"candybar"
```

- public string Substring(int startIndex)：根据已有字符串的子字符串来创建新字符串，其值从 int 表达式说明的位置(从 0 开始计数)开始到字符串结束：

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
string string1 = "foobar";
int i = 3;
string p = string1.Substring(i); // p 等于"bar"
```

- public string Substring(int startIndex, int length)：这是 Substring 的重载方法，根据现有字符串的子串来创建新字符串，第一个 int 表达式说明起始位置，第二个 length 参数说明字符串的长度，第一个字符的位置以 0 开始计数：

```
string string1 = "foobar";
string p = string1.Substring(1, 4); // p 等于"ooba"
```

- public string ToLower()：返回当前字符串的副本，将所有字符转换为小写：

```
string string1 = "Jose Cruz";
string p = string1.ToLower(); // p 等于"jose cruz"
```

- public string ToUpper()：返回当前字符串的副本，将所有字符转换为大写：

```
string string1 = "Jose Cruz";
string p = string1.ToUpper(); // p 等于"JOSE CRUZ"
```

13.3 Object 类

除了 String 类，C#中还有另外几个值得特别重视的类——Object 类就是其中一个。Object 类是.NET 类层次结构中的根类。所有的.NET 类型，从简单的预定义值类型到字符串，到数组，到预定义的引用类型，到用户自定义类型，最终都继承于 Object 类。

从 Object 继承是隐式的，所以没有必要在类定义中包含如下语法：

```
: Object
```

所以，类定义语法

```
public class Student {...}
```

等价于

```
public class Student : Object {...}
```

Object 类包含在 System 名称空间中，但和 String 类一样，Object 类也有一个别名，即关键字 object(都是小写)，从而能够在不使用 using System;指令的情况下声明 Objects 的引用。以下两行代码等价：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
System.Object x = y;
```

和

```
object x = y;
```

Object 类声明了 7 个公有方法，所以可被任何其他类型的对象访问。下面将会讨论其中两个最常用的 Object 方法：Equals 和 ToString。

13.3.1 Equals 方法

Equals 方法用来确定两个对象引用是否“相等”。如果方法用来比较引用类型的对象(如类的实例或数组)，则 Equals 方法将确定两个引用是否引用内存中的同一个对象。如果方法用于值类型的对象(如 int 或 float 这样的基本类型)，则 Equals 将进行对象值的位比较：

- public virtual bool Equals(Object obj)：任何对象都能调用该方法，第二个对象将作为参数传入：if (x.Equals(y)){…}。
- public static bool Equals(Object objA, Object objB)：这个静态方法可在 Object 类上调用，要进行比较的两个引用将作为参数传入该方法：if (Object.Equals(x, y)) {…}。

例如，这里将创建 Student 对象并维护其两个引用：

```
Student s1 = new Student("Fred");
Student s2 = s1;
```

引用变量 s1 和 s2 都在内存中引用同一个对象。现在使用相同的数据值来创建第二个 Student 对象：

```
Student s3 = new Student("Fred");
```

这里将对象比作氦气球，如图 13-2 所示。

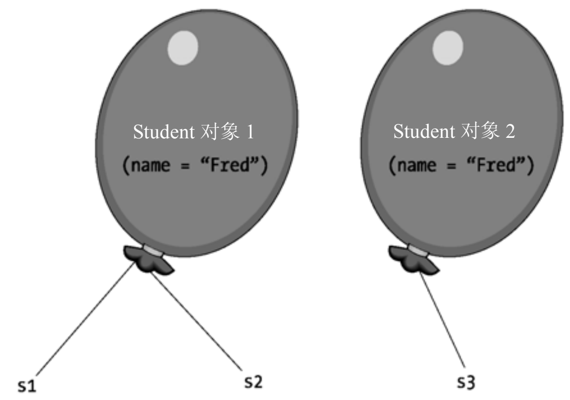

图 13-2 对象 2 和对象 1 的数据值相同，但在内存中不是同一个对象

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

如果使用 Equals 方法测试 s1 和 s2 是否相等：

```
if (s1.Equals(s2)) { // 使用 Object.Equals(s1, s2);
  Console.WriteLine("s1 equals s2");
}
else {
  Console.WriteLine("s1 does not equal s2");
}
```

则输出如下：

```
s1 equals s2
```

因为 s1 和 s2 确实引用了同一个对象，但是如果测试 s1 和 s3 是否相等：

```
if (s1.Equals(s3)) { // 或使用 Object.Equals(s1, s3);
  Console.WriteLine("s1 equals s3");
}
else {
  Console.WriteLine("s1 does not equal s3");
}
```

结果如下：

```
s1 does not equal s3
```

虽然 s1 和 s3 有相同的数据值(名字都是 Fred)，但它们引用的是两个不同的对象。

重写 Equals 方法

Object 类的非静态 Equals 方法被声明为 virtual，从而允许在派生类中重写该方法。我们经常会重写 Equals 方法以定义两个对象相等的条件。例如，在 Student 类中，也许会把两个在物理上有区别但拥有相同学号的 Student 对象看成是相等的对象。为了完成该功能，Student 类将重写 Equals 方法：

```
using System;

public class Student
{
  // 字段。
  private string id;

  // 属性。
  public string Id {
    // 存取器细节从略。
  }
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    // 重写 Equals 方法。
    public override bool Equals(object obj) {
      // 初始化一个标志。
      bool isEqual = false;

      // 使用 as 操作符将对象参数强制转换为 Student 对象，如果强制转换失败，则操作符返
      // 回 null
      Student s = obj as Student;

      // 如果强制转换的 Student 对象引用不是 null 并且当前 Student 对象的 Id 属性值相
      // 同，则将 isEqual 标志设置为 true
      if ( s != null && s.Id == this.Id ) {
          isEqual = true;
      }
      return isEqual;
    }
}
```

首先，重写的 Equals 方法将使用 as 操作符将对象参数强制转换为 Student。如果转换不成功，操作符将返回 null。如果转换的 Student 对象不是 null，而且 Id 属性值和当前 Student 对象的属性值相同，则将 isEqual 标志设置为 true。

这里将新重写的方法用于客户代码中：

```
public class Example
{
  static void Main() {
    Professor p = new Professor();

    Student s1 = new Student(); // 第一个对象……
    s1.Id = "123-45-6789";

    Student s2 = new Student(); // 第二个对象……
    s2.Id = "123-45-6789"; // 和 S1 的 ID 相同

    Student s3 = new Student(); // 第三个对象!
    s3.Id = "987-65-4321"; // 和 s1 与 s2 的 ID 不同

    Console.WriteLine("Is s1 equal to s2? " + s1.Equals(s2));
    Console.WriteLine("Is s1 equal to s3? " + s1.Equals(s3));
    Console.WriteLine("Is s1 equal to p? " + s1.Equals(p));
  }
}
```

上面的代码输出如下：

```
Is s1 equal to s2? True
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
Is s1 equal to s3? False
Is s1 equal to p? False
```

注意，这里使用= =操作符来测试两个引用是否相等

```
if (x == y) {...}
```

等价测试依赖于类。如果 x 和 y 都是引用类型的对象引用，则= =将测试两个引用是否在内存中引用同一个物理对象。如果 x 和 y 都是值类型，则= =将测试两个变量是否有相同的值。对于 string 对象而言，= =操作符将比较字符串的值，即使字符串本身就是对象。

对于重写 Equals 方法的用户自定义类型(如 Student)而言，还有另一种方法来重写= =操作符，但这已经超出本书的讨论范围，这里不再赘述。

13.3.2 ToString 方法

Object 类中最常用的方法(也是最常被重写的方法)是 ToString。它常用来返回被调用对象的字符串表示，而且有以下方法头：

```
public virtual string ToString()
```

Object 类对 ToString 方法的实现，只是简单地返回被调用对象类型的完全限定名称。例如，如果 Student 类属于 SRS 名称空间，而且会在 Student 对象(继承于 Object 类)上调用 ToString 方法：

```
Student s = new Student();
s.Name = "Dianne Bolden";
s.Id = "999999";
Console.WriteLine(s.ToString());
```

则输出结果为

```
SRS.Student
```

然而，仅仅输出类的名称意义并不大。幸运的是，和 Equals 方法的情况一样，Object 类的 ToString 方法也声明为 virtual，从而能在继承类中重写该方法。

1. 重写 ToString 方法

在上面的示例中，我们希望 Student 的 ToString 方法能够返回更多的信息，例如，标签“Student：”后面跟上学生的姓名和学号，如下所示：

```
Student: Dianne Bolden [999999]
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章

要达到这个目的，需要在 Student 类中重写 ToString 方法：

```
public class Student {
  // 字段。
  private string name;
  private string id;
  // 其他细节从略……

   public override string ToString() {
     return "Student: " + Name + " [" + Id + "]";
   }
```

前面的代码片段

```
Student s = new Student();
s.Name = "Dianne Bolden";
s.Id = "999999";
Console.WriteLine(s.ToString());
```

调用重写后的 ToString 方法得到的结果是：

```
Student: Dianne Bolden [999999]
```

2. ToString 的使用内幕

ToString 经常会在幕后被调用，如通过 Console.WriteLine 方法。重写的 Console.WriteLine 方法通常会接受字符串参数，它能将任意的对象引用作为参数传入方法。因此可以这样编写代码：

```
Student s = new Student();
s.Name = "Cheryl Richter";
s.Id = "123456";
Console.WriteLine(s);
```

当任意对象引用传入 WriteLine 方法时，WriteLine 方法将自动调用对象的 ToString 方法以获得其特定的字符串表示，然后将字符串输出到控制台上。对于前面在 Student 类中重写的 ToString 方法，运行以上代码的输出结果为：

```
Student: Cheryl Richter [123456]
```

许多在.NET Framework 类库中的预定义类都重写了 ToString 方法。更重要的是，有必要为所有用户自定义的类重写 ToString 方法，从而保证无论何时在类实例上调用 ToString，都能得到有意义的结果。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.4 使用 this 进行对象的自身引用

在客户代码(如程序的 Main 方法)中，我们会声明引用变量，以保存对象的引用：

```
Student s = new Student(); // 声明 Student 类型的引用变量
```

然后通过操作引用变量，就能方便地访问这些引用变量所引用的对象：

```
s.Name = "Fred";
```

当执行某个对象自己方法中的代码时，有时需要让对象引用自身，即自身引用(self-reference)，如下面的代码所示：

```
public class Student
{
   Professor facultyAdvisor;
   // 其他细节从略
   public void SelectAdvisor(Professor p) {
      // 这里是 SelectAdvisor()方法的内部，为特定 Student 对象执行该方法。

      // 将对导师的引用保存到某个字段中。
      facultyAdvisor = p;

      // ……现在要让该 Professor 对象将它自己放入(Student)学生之列。
      // Professor 类有这样一个方法签名：
      //
      // public void AddAdvisee(Student s);
      //
      // 这里需要在导师的对象上调用这个方法并将自身作为传入的参数，但如何引用自己呢？
      p.AddAdvisee(???);
   }
}
```

在方法体中，当需要引用正在其上执行方法的对象时，可使用保留字 this 来进行“自身引用”。在上面这个示例中，可以这样写：

```
p.AddAdvisee(this);
```

这样会将“这个”Student 的引用——即此时正在执行方法的 Student 对象——作为 Advisee 方法的参数传递给 Professor p。

第 4 章提到过，在某个类的方法中可以调用同一个类的另一个方法，无需使用点符号：

```
public class Student
{
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    // 细节从略

    public void MethodA() {
      // 可以在不使用点符号的情况下在 MethodA 的内部调用 MethodB。
      MethodB();
      // 其他细节从略……
    }

    public void MethodB() {
      // 细节从略……
    }
  }
```

以上代码是下面点符号的简写版本：

```
public void MethodA() {
  // 从 MethodA 中调用 MethodB。
  this.MethodB();
  // 其他细节从略……
}
```

因为默认会包含 this.前缀，所以不需要包含该关键字，但您可以明确地写出它。本章后面将介绍 this 关键字的另一种用法，这种用法和构造函数有关。

13.5　C#集合类

第 6 章曾讨论过，应该有一种方法能在创建对象时收集对象的引用，这样就能按照需要遍历这些对象并检索某个特定的对象，等等。在面向对象编程语言(OOPL)中，可以用一种称为集合(collection)的特殊对象类型来达到这个目的。C#中最简单的集合类型是定长数组。在第 6 章介绍数组的时候，已经提到过数组是 C#中的对象；而 System.Array 类是这些数组的基类。本节将回顾数组的内容，并介绍一些有关 System.Array 类的有趣特征。

第 6 章曾讨论过，当程序运行时很难预料需要创建多少个对象，因此用定长数组来存储不定数量对象的效率会很低。.NET Framework Class Library(FCL)的 System.Collections 和 System.Collections.Generic 名称空间定义了多种可选的集合类，以用来存储对象集合。本章将讨论两个集合类，在构建 SRS 时将会用到它们，即 List 类和 Dictionary 类。

注意

System.Array 类不在 System.Collections 名称空间中。然而，和 System.Collections 类

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

相似，System.Array 类实现了 ICollection 接口(该接口在 System.Collections 名称空间中定义)，因此 Array 类将共享其他集合类的常用行为集合。

13.5.1 再论数组

System.Array 类定义了很多有用的方法和属性，以用于搜索、排序和修改数组以及确定数组的长度。

1. 数组的 Length 属性

Array 中最常用的属性是 Length 属性，其类型为 int，表示数组中所有元素的数量。以下代码说明了如何使用 Length 属性来获得一维数组的长度：

```
int[] x = new int[20];

// 忽略数组内容初始化的细节……

// 遍历数组。
// 在 i 等于 x.length 前结束
for (int i = 0; i < x.Length; i++) {
  Console.WriteLine(x[i]);
}
```

因为数组从 0 开始计数，所以在 for 循环中使用其作为上限时，必须使其小于数组的长度，如上例所示。

注意，数组的长度不能反映数组中被显式赋值的元素数量，因为从技术上说，即使没有在数组中显式存储任何内容，数组元素仍会使用该数组类型中和零等价的值来进行自动填充(第 6 章讨论过)。数组的长度只表示数组的总容量，其实在声明数组时已经确定了其容量，以后不能再更改。

注意

Array 类中有一个方法能够用来重新设置数组的大小，后面将会介绍该方法。

Length 属性可用于多维数组中。在下面这个示例中，Length 属性确定了二维交错数组的行数和每行的列数：

```
// 创建二维数组，该数组有 3 行，每行有不同数量的列。
double[][] values = new double[3][];
    values[0] = new double[4];
    values[1] = new double[2];
    values[2] = new double[3];

    Console.WriteLine("Number of rows = "+values.Length);
    for(int i=0; i<values.Length; ++i) {
      Console.WriteLine("Number of columns in row "+i+
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
                        " is "+values[i].Length);
  }
```

输出如下：

```
Number of rows = 3
Number of columns in row 0 is 4
Number of columns in row 1 is 2
Number of columns in row 2 is 3
```

2. Array 方法

Array 类声明了很多有用的静态和实例方法，可用来检查或操作数组，下面将会讨论几个常用的方法。在 MSDN 网站的 FCL Reference 中能够找到有关 Array 类方法的完整说明。

这里讨论的方法是静态方法，这意味着这些方法必须在作为整体的 Array 类上进行调用，数组实例将作为参数传入：

- public static void Clear(Array array, int StartIndex, int length)：将数组中所有或部分内容重新设置为与零等价的值。参数包括要清空的数组，起始索引(从 0 开始计数)以及需要清空的元素数量：

```
// 创建名为 x 的数组并用值填充数组

// 清空整个数组。
Array.Clear(x, 0, x.Length);
```

- public static void Reverse(Array array)：在一维数组中反置元素的顺序：

```
int[] x = {1, 2, 3};
Array.Reverse(x);
// 现在 x 包含值：{3, 2, 1}
```

该方法还有一个重载版本，其中可以指定重置元素的起始索引和数量：

```
public static void Reverse(Array array, int startIndex, int length)
```

- public static void Sort(Array array)：对一维数组中的元素进行排序。字符串默认排序规则是按字母表次序，数字则按从小到大排序。

```
string[] names = {"Vijay", "Tiger", "Phil"};
Array.Sort(names);
// 现在元素的顺序为："Phil", "Tiger", "Vijay"
```

该方法也有重载版本，其中可以指定排序的起始索引和需要排序的元素数量：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public static void Sort(Array array, int startIndex, int length)
```

注意

其他能够进行用户自定义排序的 Sort 方法可以指定任意的对象类型，但这超出了本书的讨论范围。

- public static void Resize<T>(ref T[] array, int newSize)：该方法用于重新设置一维数组的大小。严格来说，实际发生了如下操作：创建指定大小的新数组，将旧数组的元素复制到新数组中，然后用新数组取代旧数组。
  该方法的参数列表有一些新的特性。ref 关键字允许方法更改对数组变量的引用。换句话说，当更改数组大小时，会创建新数组，变量引用将指向新数组。T[]语法说明 Resize 是通用方法，对于包含任意值或引用类型的数组都可用。

```
double[] data = new double[10]; // 初始大小是 10 个元素

int numData = 13;
for(int i=0; i<numData; ++i) { // 即将超过数组的大小

  //按需要重新设置数组大小。
  if ( i == data.Length ) {
      Array.Resize(ref data, data.Length+5);
  }
  data[i] = 1.0;
}

Console.WriteLine("Array size = "+data.Length);

// 使用 Resize 方法来减少数组大小以匹配数据量
Array.Resize(ref data, numData);

Console.WriteLine("New array size = "+data.Length);
```

结果如下：

```
Array size = 15
New array size = 13
```

还有很多 Array 类的方法可用来操作数组，请参考 MSDN 网站 FCL Reference 上对这些方法的说明。

13.5.2 List 类

本书在第 6 章中首次介绍了 List 类。它是一个通用的、能动态改变长度的一维有序列表，能存储不同数量的元素，而不用担心容器的大小。List 在逻辑上等同于一维数组

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

(Array)，但大小能够随需要改变。

List 类是等价于 ArrayList 类的通用类，这里先回顾一下 ArrayList 集合：

- 能存储任何类型的元素。
- 将元素存储为 Object 类型。
- 能够向现有 ArrayList 添加任何类型的元素。没有类型检测，而且可能在 ArrayList 中混合并匹配类型。
- 程序员必须跟踪向 ArrayList 中添加的元素类型。
- 从 ArrayList 提取的元素始终返回 Object 类型，必须强制转换成它们原来的类型。

相反，通用的 List 集合有以下特征：

- 当声明 List 时会保存特定类型的元素。
- 只有声明过的类型元素才能添加到 List 中。
- 从 List 中提取的元素类型是 List 声明的保存类型。不需要从类型 Object 强制转换到提取的元素类型。

由于 List 继承的类型安全性并且提供了更好的性能，List 集合要优于 ArrayList 集合，因此在第 14 章编写代码时我们将使用 List 类。

List 类在 System.Collections.Generic 名称空间中定义。为了通过简单名称引用 List 类，可在代码顶部包含 using 指令：

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

或通过完全限定名称来引用 List：

```
System.Collections.Generic.List
```

1. 创建 List

可以用 List 类提供的 3 个构造函数来实例化 List 对象：

- 最简单的形式是无参数的构造函数。例如，要创建保存 Course 对象集合的 List：

  ```
  List<Course> coursesTaken = new List<Course>();
  ```

  这个构造函数将创建一个空 List，其初始的默认容量为 0。当第一个项被添加到 List 中时，容量将设置为 4。当 List 中的元素数量达到 List 的当前容量时，其容量将翻倍。
- List 构造函数的第二种形式带一个整数参数，该参数表示 List 的初始容量(即保留空间)，例如：

  ```
  List<Student> students = new List<Student>(400);
  ```

  当我们知道会向 List 中添加大量的对象引用时，可使用这种形式的构造函数。使

专业图书网上第一专营店 www.china-pub.com 独家提供样章

用较大的容量能够提高性能，因为这减少了重新调整 List 大小的次数。

- 第三种构造函数将使用集合元素来初始化 List，即实现 IEnumerable 接口的集合，而 IEnumerable 接口是在 System.Collections.Generic 名称空间中定义的。例如，List 可以使用 Course 对象的一维数组来进行初始化：

```
// 创建 Course 对象的数组……
Course[] courses = new Course[3];
courses[0] = new Course("Math 101");
courses[1] = new Course("Management 283");
courses[2] = new Course("Physics 250");

// ……然后在程序中使用该数组来初始化列表。
List<Course> coursesTaken = new List<Course>(courses);
```

在上个代码片段中，List 的长度是 3，而且其包含数组中所有 Course 对象的引用。注意，这两个集合现在都引用相同的对象，如图 13-3 所示。

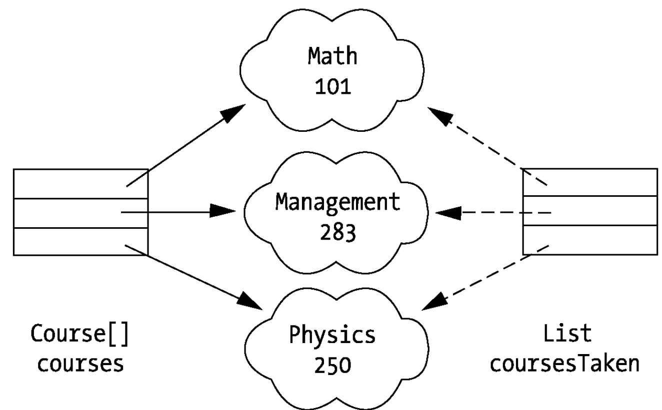

图 13-3 使用另一个集合的内容初始化 List

2. List 属性

List 类声明了许多属性以返回有关 List 的信息：

- Capacity 属性用来获得或设置 List 的容量(如果试图通过编程方式将 List 的容量重新调整为小于其现有容量，则会抛出 ArgumentOutOfRangeException 异常)。
- Count 属性返回 List 中当前包含元素的数量。
- 要访问 List 中存储的元素，List 类定义了一个特殊的索引器 Item 属性，从而允许我们获得或设置 List 的元素，就像访问标准数组的元素一样——括号中的整数表达式表示所需的索引。

下例将说明这些属性：

```
// 这里没有显示：使用 Course 对象初始化 List 集合。

// 访问 Capacity 和 Count 属性。
```

专业图书网上第一专营店
www.china-pub.com
独家提供样章


```
Console.WriteLine("capacity = " + coursesTaken.Capacity);
Console.WriteLine("no. of elements = " + coursesTaken.Count);

// 使用索引器取出 List 中的元素。
for (int i = 0; i < coursesTaken.Count; i++) {

    Course c = coursesTaken[i];

    // 现在可以在 Course c 上调用属性和方法。
    Console.WriteLine(c.Name);
}
```

3. List 方法

常用的 List 类方法如下：

- public void Add(T item)：在 List 末尾添加一个元素，如果需要包含引用，则会自动扩展 List。通用集合的一个关键优点在于编译器将检查添加的项是否和 List 声明的类型兼容。例如，如果向声明类型为 Course 的 List 中添加 string 对象，则会产生以下编译器错误：

```
Error: The best overloaded method match for
'List<Course>.Add(Course)' has some invalid arguments.
Cannot convert from 'string' to 'Course'.
```

- public void Insert(int index, T item)：在 List 的特定位置中插入特定项。现有元素将向后移动以容纳新项。
- public void AddRange(IEnumerable<T> collection)：在 List 末尾添加特定集合的内容。
- public int IndexOf(T item)：在 List 中搜索某个特定对象，如果找到，则返回一个说明其(第一个)索引位置(从 0 开始计数)的整数。如果没有找到特定对象，则返回－1。
- public bool Contains(T item)：在 List 中搜索某个对象，如果找到则返回值为 true，否则为 false。
- public void RemoveAt(int index)：删除特定索引处的元素，并填补其空缺。Count 属性将会减 1，但 Capacity 保持不变。如果特定索引超出 Count 或小于 0，则抛出 ArgumentOutOfRangeException 异常。
- public bool Remove(T item)：搜索特定对象，如果找到，则从 List 中删除第一个这样的对象，并填补其空缺。Count 会减 1，但 Capacity 保持不变。如果成功删除这个项，则返回 true；如果没有在 List 中找到这个项或删除该项不成功，则返回 false。如果没有找到特定对象，则 List 保持不变。要删除项的所有实例，可以在 while 循环中结合使用 Contains 和 Remove 方法。

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
while (list.Contains(x)) {
  list.Remove(x)
}
```

- public void sort()：对 List 中的元素排序。默认按字母顺序对 string 元素排序，按从小到大的次序对数值型元素排序。

> **注意**
>
> 该方法的其他版本还能为任意的对象类型定制排序算法，但本书不会讨论用户自定义的排序算法。

- public void Clear()：清空 List，Count 属性被设为 0，Capacity 属性不变。
- public T[] ToArray()：创建一个数组实例，并将 List 中的元素复制到该数组中。数组中元素的类型和 List 中包含的元素类型相同。

List 方法还有很多！请参阅 MSDN 网站上的 FCL Reference 以了解所有关于 List 方法的说明。

13.5.3 Dictionary 类

Dictionary(字典)类提供了另一种在 C#中管理元素集合的方法。Dictionary 比 List 略微高级一些，因为它能根据唯一的键值来访问给定的对象，这是字典集合类型的实现，已在第 6 章中定义过。和 List 类一样，Dictionary 类表示通用集合。当初始化 Dictionary 时会指定键(key)和值(value)，通常键是字符串。

和 List 类一样，Dictionary 类能在 System.Collections.Generic 名称空间中找到。要通过简单名称引用 Dictionary，可在源代码文件的顶端放入 using System.Collections.Generic; 指令，或者可通过完全限定名称 System.Collections.Generic.Dictionary 来引用 Dictionary。

1. 创建 Dictionary

您可以使用任意一种 Dictionary 类的构造函数来创建 Dictionary 对象。最简单的方法是使用不带参数的构造函数来初始化 Dictionary，这样将创建一个空的 Dictionary。然后可使用 Add 方法插入对象，其方法头如下：

```
public void Add(TKey key, TValue value)
```

注意，必须为在 Dictionary 中添加的每个值定义一个键，这样就能使用键来检索值。本例将使用简单的 string 对象作为键，即学生的学号：

```
// 创建 Dictionary 实例。
Dictionary<string, Student> students = new Dictionary<string, Student>();

// 创建几个 Student 对象
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
Student s1 = new Student("123-45-6789", "David Chen");
Student s2 = new Student("987-65-4321", "Mary Jones");
Student s3 = new Student("654-32-1987", "Gerson Lopez");

// 在Dictionary中存储Student对象，使用Id属性(其类型为string)作为每个对象的键。
students.Add(s1.Id, s1);
students.Add(s2.Id, s2);
students.Add(s3.Id, s3);
```

Dictionary 类定义了索引器，可使用它来获得或设置 Dictionary 中的元素。和 List 类一样，索引器通过一对括号来表示；然而对于 Dictionary 对象来说，会在元素的键值两边使用方括号：

```
// 根据键来检索Student对象。
Student s = students["123-45-6789"]; // 检索表示Student David Chen的对象
                                     // 引用
Console.WriteLine("name is " + s.Name);
```

2. Dictionary 属性

除了 Count 和索引器属性外(Dictionary 中这两个属性的工作原理和 List 中的相同)，Dictionary 类声明了许多其他的属性以返回有关 Dictionary 的信息。以下是两个属性的明细：

- Keys 属性返回 Dictionary(TKey, TValue).KeyCollection 对象，其元素是包含在 Dictionary 中的键。
- Values 属性返回 Dictionary(TKey, TValue).ValueCollection 对象，其元素是包含在 Dictionary 中的值。

有关 Dictionary 属性的完整列表，请参阅 MSDN 网站的 FCL Reference。

3. Dictionary 方法

Dictionary 类的一些常用方法如下：

- public void Add(TKey key, TValue value)：前面已经讨论过，该方法会在 Dictionary 中插入一个值，这样就能使用特定键来检索值。如果在这个键的位置已经存储过对象，则会抛出 ArgumentException 异常，因此有必要先验证这个键不在使用表中，这可以通过后面将会讨论的 ContainsKey 方法来实现。编译器将完成类型检查，如果键和值的类型与 Dictionary 声明的不匹配，则会发生错误。
- public bool ContainsKey(TKey key)：如果在 Dictionary 中找到指定的键，则返回 true；否则返回 false。

下面这个示例说明如何结合使用 ContainsKey 方法和 Add 方法，从而确保没有在 Dictionary 中重写已有的记录项：

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
Student s = new Student("111-11-1111", "Arnold Brown");
if (students.ContainsKey(s.Id)) {
  // 噢！这里发生了重复，我们需要决定如何操作！
  // 细节从略……
}
else {
  students.Add(s.Id, s); // 没有问题，因为没有检测到重复。
}
```

- public bool ContainsValue(TValue value)：在不使用关联键的情况下搜索指定的值，如果找到该值则返回 true，否则返回 false。
- public bool Remove(TKey key)：尝试删除对应于 Dictionary 中给定键的值。如果成功删除，则方法返回 true；如果 Dictionary 中不存在这个键或删除失败，则返回 false。
- public void Clear()：清空 Dictionary，就像重新初始化一样。Count 属性将设置为 0。

还有更多 Dictionary 类的方法，有关 Dictionary 所有方法的细节内容，请参考 MSDN 网站上的 FCL Reference。

13.5.4 使用 foreach 循环迭代访问集合

foreach 循环是我们在第 1 章中讨论过的 C#控制流结构之一。foreach 循环提供了迭代遍历 C#集合中的元素的方法。

foreach 循环的通用语法如下：

```
foreach (type variable_name in collection_name) {
   // 将要执行的代码
}
```

在 foreach 关键字后面的圆括号中，我们发现：

- 在集合中将被处理的项的类型——在集合中正被处理的项将自动转换为这种类型
- 能够逐个引用集合中每一项的局部引用变量
- 要进行搜索的集合名称，前面有关键字 in

以下示例将使用 foreach 循环迭代访问存储 Student 对象的通用 List 集合：

```
List<Student> students = new List<Student>();

// 省略填入 List 集合的细节

// 使用 foreach 循环迭代访问 List。
foreach ( Student student in students ) {
  Console.WriteLine(student.Name);
}
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章

每次 foreach 循环检索完指定类型的元素时，都会执行 foreach 语句中提供的代码块。

注意，一旦进入一个 foreach 代码块，就不能通过给变量赋新值的方法来更改 foreach 引用变量所指向的对象。以下代码无法编译：

```
foreach (Student s in studentBody) {
  // 无法编译！引用变量 s 是只读的。
  s = studentBody[0];
}

error: cannot assign to 's' because it is a 'foreach' iteration variable
```

当然，引用当前集合中的元素的迭代变量 s 是可访问且可修改的，这与其他对象引用相同：

```
foreach (Student s in studentBody) {
  // 完全可以如此操作。
  s.Name = "?";
}
```

当使用 foreach 循环遍历 Dictionary 集合时会有更多的选项。Dictionary 存储 1 个或多个键-值对。Dictionary 类的 Keys 属性可用来获得到 Dictionary.KeyCollection 对象的引用，该对象包含 Dictionary 的键。foreach 循环可用来迭代这些键：

```
Dictionary<string, Student> students =
                   new Dictionary<string, Student>();

// 这里省略了向 Dictionary 中添加键-值对的细节。

foreach( string key in students.Keys ) {
    Console.WriteLine("key name is "+key);
}
```

同样，Dictionary 类的 Values 属性可用来获得 Dictionary.ValueCollection 对象的引用，该对象包含由 Dictionary 存储的值。foreach 循环可用来迭代这些值：

```
Dictionary<string, Student> students =
                 new Dictionary<string, Student>();

// 这里省略了向 Dictionary 中添加键-值对的细节。

foreach( Student s in students.Values ) {
      Console.WriteLine("Student name is "+s.Name);
}
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章

foreach 循环还可以直接应用于 Dictionary 对象，以同时获得由 Dictionary 存储的键和值。在本例中，元素将被检索为 KeyValuePair 对象。KeyValuePair 类的 Key 和 Value 属性可用来获得 Dictionary 中键和值的引用。

```
Dictionary<string, Student> students =
                  new Dictionary<string, Student>();

// 这里省略了向 Dictionary 中添加键-值对的细节。

foreach( KeyValuePair<string, Student> kv in students ) {
    Student s = kv.Value;
    Console.WriteLine("key is "+kv.Key+
                         " Student name is "+s.Name);
}
```

13.6 再论字段

既然我们已经学过 FCL 中通用集合类的细节内容，那么下面将进一步讨论字段。本书前面的章节已经讨论过字段，它是用来存储特定于给定对象的数据的值或对象。在前面的章节中，我们忽略了其中一些细节，本节将介绍一些有关字段的细节内容。

13.6.1 再论变量初始化

第 1 章曾提过，如果没有显式初始化局部变量就试图访问它的话，会发生编译错误。例如，以下代码：

```
public class Example
{
  static void Main() {
      // 在 Main()方法中声明一些局部变量。
      int i; // 没有自动初始化
      int j; // 同上
      j = i; // 编译错误!
  }
}
```

将会产生以下编译错误：

```
error: Use of unassigned local variable 'i'
```

第 3 章曾经提过，如果没有显式地给变量赋值，则会给变量赋一个与 0 等价的值。然而，这些变量初始化语句显得过于简单，现在我们将会纠正这个问题。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

要正确理解 C#中初始化的概念，必须区分局部变量(local variables)(即在方法中声明的变量，其范围局限于该方法)和类的字段(fields of a class)，无论是实例还是静态变量，都在类的范围层面上被声明。事实证明：

- 任何类型的局部变量，在程序中被显式初始化之前，编译器都会将其看作未初始化。
- 任何类型的字段会自动被初始化为与 0 等价的值。如果是布尔型则初始化为 false，数值型则初始化为 0 或 0.0，引用类型则初始化为 null，等等。

以下实例将说明这几点：

```
public class Student
{
    // 自动初始化字段。
    private int age; // 初始化为 0
   private double gpa; // 初始化为 0.0
   private bool isHonorsStudent; // 初始化为 false
   private Professor myAdvisor; // 初始化为 null
   // This includes STATIC variables.
   private static int studentCount; // 初始化为 0
   // 等等。

   // 方法。
   public void UpdateGPA() {
   // 不会自动初始化局部变量。
   double val; // 未初始化——值未定义。
   Course c;    // 未初始化——值未定义。
   // etc.
  }
}
```

13.6.2 隐式输入的局部变量

在方法体中声明的局部变量，如果没有显式声明，则会指定隐式类型为 var。隐式输入的局部变量必须在其声明时被初始化，而且编译器会根据初始化的值推断变量类型。隐式输入的局部变量可用于 for 和 foreach 语句中。考虑以下示例：

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

// 这个简单程序说明了 var 的语法

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

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    // name 被编译为字符串
    var name = "Lisa";

    // names 被编译为 List<string>
    var names = new List<string>();

    // 在 foreach 语句中使用 var
    foreach(var item in names ) { // 假设 var 是字符串
      // 使用 item 进行操作
    }
  }
}
```

var 关键字只能用于局部变量中，而不能用于在类级别上声明的字段。数组的初始化不能使用隐式输入的局部变量。下面的 var 示例不能编译：

```
public class VarDemo
{
  // 不能编译，这是一个类变量
  private var id = "111-11-111.dat";

  static void Main() {

    // 数组的初始化不能用 var
    var names = { "Math 101", "Ballroom 262", "Physics 245" };
  }
}
```

注意，过度使用 var 会使代码的可读性变差。应该始终显式声明变量的类型。var 主要用于语言集成查询(Language-Integrated Query，LINQ)特性中，但它不在本书的讨论范围之内。

13.7 再论 Main 方法

在第 1 章中，我们介绍了 Main 方法，它是 C#程序执行的起始点。本节将进一步介绍 Main 方法的一些特征。

13.7.1 Main 方法的几种形式

Main 方法有以下 4 种不同的形式：

- 第一种版本在第 1 章中已经介绍过，即无参数的方法，返回类型是 void：

```
static void Main()
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章

- 第二种版本的返回类型为 int：

```
static int Main()
```

返回 int 值可以告诉执行环境(操作系统)程序是如何终止的：

```
public class Foo
{
  static int Main() {
    if (something goes awry) {
      return -1;
    }
    else {
        // 一切正常!
    return 0;
    }
  }
}
```

返回值为 0 通常用来说明程序正常执行。让 Main 方法返回一个值有时会用于在批处理中运行程序。在批处理文件中可以测试或检查该返回值。

- 另外两种 Main 方法接收 string 数组作为参数，该 string 数组表示程序调用时传递给 Main 方法的命令行参数。注意，数组名称通常是 args，但实际上可以是任何有效的数组名称：

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

在 14 章建立 SRS 的命令行驱动版本的程序时会介绍如何在 C#程序中传递命令行参数。

注意，这 4 种 Main 方法都没有包含访问修饰符(如 public)。即使包含访问修饰符，也会在运行时忽略它们。C#的惯例是在 Main 方法中忽略访问修饰符。

13.7.2 静态 Main 方法

为何要将应用程序的 Main 方法声明为 static？当.NET 运行库通过调用 Main 方法启动一个应用程序时，此时还没有任何对象，因为 Main 方法才是初始化应用程序对象的起始点。所以我们将面临“鸡与蛋”的矛盾：如果没有初始对象调用 Main 方法，如何才能调用 Main 方法来创建对象？

第 7 章曾介绍过，静态方法是一种能作为整体而在类上调用的方法类型，即使还没有这个类的实例。所以，对于 C#设计人员，在创建任何对象前，最好的选择是将 Main 方法强制设计为静态方法，这样就能从其“包装器”调用该方法，

专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.8 再论输出到屏幕

第 1 章曾介绍过，调用 Console.WriteLine( )或 Console.Write( )这两个方法就能将文本消息输出到命令行窗口中，如下所示：

```
Console.WriteLine(expression to be printed);
Console.Write(expression to be printed);
```

当时没有详细指出上述语法的特别之处，现在我们已经比较了解对象的概念，本节将深入介绍这些语法。

Console 类由 System 名称空间中的 CLR Base Class Library(FCL 的一部分)提供，该类定义了许多重载的静态 Write 和 WriteLine 方法。不同的方法版本接收不同的参数类型：字符串、各种预定义的值类型或任意的对象引用：

```
public static void WriteLine(string s)
public static void WriteLine(int i)
public static void WriteLine(double d)
public static void WriteLine(bool b)
public static void WriteLine(object obj)
```

等等(Write 方法亦是如此)。

Write 和 WriteLine 方法能够接受任意复杂的表达式，并尽力将其呈现为单个 string 值，然后将其显示在标准输出窗口(即调用程序的命令行窗口)中。对于非字符串参数，通过在幕后调用 ToString 方法可将其呈现为字符串的表示，本章前面已经讨论过这一点。

下面的代码说明了几个 Console.Write 的复杂调用形式。第一次调用时，作为参数传入的表达式的值为字符串值，第二次调用传入的表达式值为 double 值，本例调用了两个不同(重载的)版本的 Write 方法：

```
Professor p = new Professor();
// 细节从略。
Console.Write("Professor " + p.Name + " has an advisee named " +
                  p.Advisee.Name + " with a GPA of ");
Console.Write(p.Advisee.ComputeGPA());
Console.WriteLine(".");
```

输出如下：

```
Professor Jacquie Barker has an advisee named Sandy Tucker with a GPA of 4.0.
```

格式化输出

我们已经使用过+运算符在 Write 和 WriteLine 方法的参数中连接不同的字符串。.NET

专业图书网上第一专营店 www.china-pub.com 独家提供样章

Framework 还提供了定义格式化字符串的功能，它能灵活地指定输出到屏幕或写入文件的方式。声明格式化字符串的最简单方法是在字符串中的大括号内放入整数标记，这个整数对应于在字符串后包含参数的位置。例如，将上例中的字符串重写为以下格式化字符串：

```
Console.WriteLine(
  "Professor {0} has an advisee name {1} with a GPA of {2}.",
  p.Name, p.Advisee.Name, p.Advisee.ComputeGPA());
```

在大括号中使用参数索引是使用格式化输出的最简单方法。您还可以指定输出值的宽度和对齐方式，以及显示值的格式(如十进制、指数、百分数等)。有关格式化输出的更多细节，请参考 MSDN 文档页面。

13.9 再论构造函数

在我们深入学习 C#的一些特性时，我们还需要回顾一下构造函数。第 4 章曾学过，当使用 new 操作符初始化全新的对象时，此时创建的是一个“骨架”对象，其所有字段值都为空(前面已经讨论过，每个字段将被初始化为 0、null 或对应于给定字段类型为 0 的值)。如果要更智能地创建对象——即当创建对象时要完成更精确的操作——则需要声明一个构造函数。

回顾一下前面所学的内容，构造函数

- 具有与类名相同的名称
- 没有显式的返回类型，因为它有一个默认返回类型，与类为其定义的类型相同——构造函数返回该类型的新对象/实例
- 可接受任意数量的参数

下面是 Student 类的构造函数示例：

```
public class Student
{
   // 字段。
   private string name;
   // 其他细节从略

   // 构造函数(注意：没有返回类型！)
   // 该构造函数接受一个字符串值，表示在第一次初始化实例时要赋给 Student 对象的
name
   // 字段值。
   public Student(string name) {
      this.name = name;
   }

   // 等等。
}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

在上个示例中，this 关键字出现在构造函数中，因为这里发生了命名冲突，在构造函数参数列表中的参数有着和类中声明字段相同的名称(即 name)。这个 this.name 语法告诉编译器我们正在引用的是字段而非参数。

下面将介绍构造函数的其他内容。

13.9.1 重载构造函数

我们可以为同一个类，根据不同的参数组合，创建许多不同的构造函数——这称为重载(overloading)，第 5 章曾介绍过这个概念。只要每个构造函数有不同的参数签名，那么它们就是不同的构造函数：

```
// 一个 string 参数。
public Student(string name) {
    // 细节从略……
}

// 两个 string 参数，没有问题！
public Student(string name, string id) {
    // 细节从略……
}

// 一个 int 参数和一个 string 参数，这也没有问题！
public Student(string name, int id) {
    // 细节从略……
}
```

如果要将下面第 4 个构造函数添加到 Student 类中，则编译器会报错，因为已经存在有两个 string 参数的构造函数——参数名不同是无效的。

```
public Student(string firstName, string lastName) {
    // 细节从略……
}
```

13.9.2 替换默认的无参数构造函数

第 4 章曾提到过，如果没有为类声明任何构造函数，则系统会提供一个默认的无参数构造函数，它会将所有字段初始化为与 0 等价的值。这里有一个关于 C#默认构造函数的重要警告：如果我们自己创建一个带任何参数的构造函数，则系统将不再自动提供默认的无参数构造函数。这么做是故意的，因为如果在构造函数的编程过程中碰到过麻烦，则我们必定会有有关该类的特殊初始化需求，C#默认的构造函数无法预测该需求。

如果除了带参数的构造函数外，还需要无参数的构造函数，则必须显式地编写无参

专业图书网上第一专营店 www.china-pub.com 独家提供样章

数的构造函数。

下面是Student类的另一个版本的示例，该示例提供了多个构造函数。注意，本例替换了“消失的”无参构造函数(请阅读代码中的注释)：

```
// Student.cs

using System;

public class Student
{
  private string name;
  private string id;
  private Professor facultyAdvisor;
  // 构造函数。

   // 该构造函数有3个参数。
  public Student(string n, string id, Professor p) {
    name = n;
    this.id = id;
    facultyAdvisor = p;
  }

   // 该构造函数有2个参数。
  public Student(string n, string id) {
    name = n;
    this.id = id;

    // 因为在该构造函数中还没有传递来的Professor对象，所以此时将facultyAdvisor
    // 字段设为null(严格来说，不需要如此操作，因为它会被自动初始化为null，但这样
    // 做会使代码更易读)
    facultyAdvisor = null;
  }

   // 这里提供无参构造函数。
  public Student() {
    // 注意这里为name和id字段提供了一些“占位符”值，以防止没有传递任何值的情况。

    name = "???";
    id = "???-??-????";
    facultyAdvisor = null;
  }

  public string Name {
    // 存取器细节从略。
  }

  // 其他属性，等等。
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    public string GetFacultyAdvisorName() {
      // 注意：因为有些构造函数使用 Professor 对象初始化 facultyAdvisor，而有些没
有，
      // 所以我们无法假设在调用该方法时能够用 Professor 的引用来初始化这个字段。因此在
      // 处理前，请确保 facultyAdvisor 字段不为 null。
      if (facultyAdvisor != null) {
        return facultyAdvisor.Name;
      }
      else {
        return "TBD";
      }
    }
}
```

下面给出了用于测试的 Professor 类的简化版本：

```
// Professor.cs

public class Professor {
  private string name;

  public string Name {
    get {
      return name;
    }
    set {
      name = value;
    }
  }
}
```

下面是测试不同形式构造函数的主程序：

```
public class MyProgram
{
  static void Main() {
    Student[] students = new Student[3];
    Professor p;

    p = new Professor();
    p.Name = "Dr. Oompah";

    // 这里将测试不同形式的 Student 构造函数。
    students[0] = new Student("Joe", "123-45-6789", p);
    students[1] = new Student("Bob", "987-65-4321");
    students[2] = new Student();

    Console.WriteLine("Advisor Information\n");
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        foreach( Student s in students ) {
          Console.WriteLine("Name: {0}\tAdvisor: {1} ",
                s.Name, s.GetFacultyAdvisorName());
        }
      }
    }
```

上例将产生以下输出：

```
Advisor Information

Name: Joe   Advisor: Dr. Oompah
Name: Bob   Advisor: TBD
Name: ???   Advisor: TBD
```

当考虑派生类和继承的构造函数时，还需要了解一些复杂之处——本章后面“更多关于继承和 C#的知识”一节中将会介绍这些内容。

13.9.3　在类中重用构造函数代码

本章前面已经使用过 this 关键字来实现对象自身引用，this 关键字的另一种用法和重用构造函数代码有关。

如果类声明了不止一个构造函数，而且想要在一个构造函数中重用另一个构造函数的代码，则可以在构造函数的声明中使用 this 关键字，以作为在另一个构造函数中运行该构造函数的速记方式：

```
: this(optional arguments)
```

下面是说明用法的最佳示例：

```
// Student.cs

using System;

public class Student
{
  private string name;
  private string id;
  private Transcript transcript;

   // 构造函数。

   // 这个版本带有一个参数。
   public Student(string n) {
     name = n;
     transcript = new Transcript();
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        // 进行其他复杂的操作……
    }

    // 这个版本带有两个参数。我们希望重用上一个构造函数中的逻辑，而不需要重复其中的代
    // 码。通过以下方法使用 this 关键字，可以从带有两个参数的构造函数中调用带有一个参
    // 数的构造函数：
    public Student(string n, string id) : this(n) {
        // 现在可以进行本构造函数所关心的其他操作。
        this.id = id;
    }

    // 等等。
}
```

通过使用语法: this(n)，等同于以下构造函数中的代码，不需要重复上面构造函数中的代码：

```
public Student(string n, string id) {
    // 从第一个构造函数中复制代码
    name = n;
    transcript = new Transcript();
    // 进行一些其他复杂的操作……

    // 现在可进行构造函数所关心的其他操作。
    this.id = id;
}
```

因此，通过使用:this(…)语法，可以在一个构造函数中重用另一个构造函数的代码，如果第一个构造函数的逻辑发生变化，则第二个构造函数也会受益。

因为类的每个重载版本的构造函数都确保有唯一的参数列表，所以传递到 this(…)表达式的参数将会明确地选择将要调用的构造函数。

注意，如果构造函数的第一个版本将调用构造函数的第二个版本，则调用第一个版本的构造函数时，执行的第一个操作是调用第二个版本的构造函数，如下所示：

```
public Student(string n, string id) : this(n) {
    // 在到达下一行代码时已经运行了带有一个参数的构造函数……
    this.id = id;
}
```

13.10 更多关于继承和 C#的知识

在第 5 章讨论继承时，曾经学过

- 当某个新类是已有类的特例时，继承可用来从现有类中派生出新类。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

- 派生类自动继承基类的数据结构和行为。
- C#中表示一个类从另一个类派生的语法是：在类声明后使用一个冒号加基类名称：

```
public class Person
{
  private string name;

  public string Name {
    get {
      return name;
    }
    set {
      name = value;
    }
  }
}

// 从 Person 类派生 Student 类。
public class Student : Person
{
    // 如果没有在类的内部定义任何内容，类仍会有一个字段(name)和一个属性(Name)，
    // 因为它们从 Person 中继承而来(事实上还有其他成员，因为 Person 类从 Object
    // 类中继承而来！)
}
```

虽然在第5章中介绍了继承的基础知识，但仍有许多C#继承技术的要点还没有提及，下面将会介绍这些。

13.10.1 继承组件的可访问性

通过继承，在基类中定义的任何内容都会在派生类中体现——继承是“全有或全无”的命题。然而，继承类不能直接访问某些继承的成员(字段、方法和属性等)，这取决于它们在基类中定义的访问权限。

考虑以下基类：

```
public class Person
{
  // 字段。
  Accessibility_modifier int age;

  // 该类的其他细节从略。
}
```

第4章中介绍过，成员的可访问性有：

专业图书网上第一专营店 www.china-pub.com 独家提供样章

- private(私有)
- public(公有)
- protected(受保护)
- internal(内部)
- protected internal(内部受保护)
- 未指定(字段、方法和属性默认为 private)

第 4 章中已经介绍过 public 和 private 的可访问性。对于给定 protected 可访问性的成员，其局限于类的内部，对声明该成员的类和派生类来说是公有的，对于其他类来说是私有的。另外两种可访问性 internal 和 protected 用得比较少，已经超出了本书讨论的范围。

假设从 Person 类派生 Student 类：

```
public class Student : Person
{
   // age 字段继承于基类……
   // 添加一个方法，直接通过名称操作该字段。
  public bool IsOver65( ) {
     if (age > 65) {
       return true;
    }
    else {
      return false;
    }
  }
}
```

当试图编译 Student 类时会发生什么情况？答案取决于 Person 类中对 age 字段可访问性的定义：

- 如果在 Person 中将 age 声明为 protected 或 public，则 Student 类可通过简单名称继承并直接访问 age 字段，上面的 Student 类将通过编译而不会产生错误。
- 另一方面，如果在 Person 中将 age 声明为 private，则在编译以下 Student 代码时将导致编译错误：

  ```
  if (age > 65) {
  ```

  错误消息为：

  ```
  error: 'Person.age' is inaccessible due to its protection level
  ```

  这是因为虽然继承了 age 字段——它是组成 Student 对象的数据结构之一——但它对 Student 类而言不可见！这和我们的身体器官类似，例如，心脏是身体的一部分，但我们却无法直接看到或者触摸到它。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

如果倾向于将所有字段声明为 private，又如何在子类中操作继承而来的私有字段？答案很简单：通过从父类继承的 public(或 protected)的存取器或属性。这里将修改上面的示例程序以说明这种技术。

首先，在 Person 类中声明一个公有的 Age 属性：

```
public class Person
{
  private int age;
  // 细节从略。

   // 为子类提供了能被继承的属性(如果只希望子类能访问它，则可以将属性声明为
protected)。
   public int Age {
     get {
       return age;
     }
     set {
       age = value;
     }
   }
}
```

然后，在 Student 类的 IsOver65 方法中使用继承的 Age 属性，Student 类的编译正常。

```
public class Student : Person
{
  //其他细节从略。

  public bool IsOver65( ) {
    // 虽然无法直接访问 age 字段，但从 Person 继承的 Age 属性能够访问 age 字段的值。
    if (Age > 65) return true;
    else return false;
  }
}
```

因为 Age 属性声明为 public，所以其他所有类都能访问它。如果只想让派生类(包含 Person 类)访问，Age 属性应该声明为 protected。第 4 章曾提到过，最好使用属性来访问字段值(即使在类自己的方法中亦是如此)，这样能利用属性存取器为该字段提供的特殊处理的好处。

13.10.2　重用基类行为：base 关键字

第 5 章已经学过，如果在派生类中提供与基类方法签名相同的方法(相同的方法名和参数列表)，则我们就重写(override)了基类的方法(基类方法签名应该包括 virtual 关键字)。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

何时需要这么做？在派生类需要对方法调用采取不同于基类方法的操作时，需要这样做，如下例所示：

```
public class Person
{
  private string name;
  private string id;

  public string Name {
    // 存取器细节从略。
  }

  // 等等。

   // 让 Person 对象描述自己。例如，“John Doe(123-45-6789)”
   public virtual string GetDescription() {
       return Name + " (" + Id + ")";
   }
}

public class Student : Person
{
  private string major;

  public string Major {
    // 存取器细节从略。
  }

   // 我们想要让 Student 对象返回与父类方法不同的自身描述。因此，这个子类有着和父类
相
  // 同的方法签名；这个方法重写了继承的方法。
   // 这个方法添加了有关学生专业的信息。
   public override string GetDescription() {
     return Name + " (" + Id + ") [" + Major + "]";
   }
}
```

第 5 章已经介绍过，派生类方法可以使用 base 关键字调用基类的方法。这个功能可降低代码的冗余，因为这样不仅能重用基类所做的工作，而且可以添加派生类方法所需的其他代码。

下面将修改 Student 类的 GetDescription 方法，这样它会首先调用 Person 类的版本：

```
public class Student : Person
{
  private string major;

  // 等等。
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    // 和 Person 中定义的方法名相同，所以这个方法重写了继承的版本。
    // 然而请注意，现在调用父类的这个方法，从而重用其中的代码。
    public override string GetDescription() {
        return base.GetDescription() + " [" + Major + "]";
    }
}
```

就像 this 关键字通常用来在方法中引用其对象一样，base 通常用于在方法中引用其成员在父类中的版本。

base 关键字的另一种用法与构造函数和继承有关，下面将会讨论。

13.10.3 继承和构造函数

构造函数的继承与方法和属性的继承不同，这里将通过下面的示例说明其中的复杂之处。

首先为 Person 类声明一个带有两个参数的构造函数：

```
public class Person
{
  private string name;
  private string id;

  // 声明了公有的 Name 和 Id 属性，细节从略。

  // 只显式声明了一个构造函数。
  public Person(string n, string id) {
    Name = n;
    Id = id;
  }

  // 等等。
}
```

前面已讨论过，Person 类现在只能识别一个构造函数——带有两个参数的构造函数——因为 Person 类默认的无参数构造函数将被删除。

现在，假设从 Person 类派生 Student 类，而且想为 Student 类定义两个构造函数——带有 2 个参数和带有 3 个参数的构造函数。因为构造函数不能被继承，所以我们不能从 Person 类的带有两个参数的构造函数中获得好处，必须为 Student 显式地重新编码构造函数，如下所示：

```
public class Student : Person
{
  private string major;
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
  // 提供了主要属性，细节从略。

  // 带有两个参数的构造函数。
  public Student(string n, string id) {
    // 注意这个构造函数及其父类构造函数间的逻辑冗余——稍后会修正这个问题。
    Name = n; // 冗余
    Id = id;  // 冗余
    Major = null;
  }

  // 带有三个参数的构造函数。
  public Student(string n, string id, string m) {
    // 更多的冗余!
    Name = n; // 冗余
    Id = id; // 冗余
    Major = m;
  }

  // 没有显式声明其他构造函数。
}
```

除了有很多的冗余，上面的代码还无法通过编译。派生的构造函数总是首先调用基类的构造函数——它默认将调用不带参数的基类构造函数。如果基类没有不带参数的构造函数(Person 类没有)，则会发生编译错误。

幸运的是，有一种方法可以重用父类中构造函数的代码而无需在派生类的构造函数中重复这些代码的逻辑。通过前面讨论方法重用时提到的 base 关键字就能实现这一点。为了显式地重用父类中特定的构造函数，可以使用: base(optional arguments)并传入其所需的任何参数，如下面改写的 Student 类所示：

```
public class Student : Person
{
  private string major;
  // 提供主要属性，细节从略。

  // 带有两个参数的构造函数。
  // 将显式调用带有两个参数的 Person 构造函数，传入 n 和 id 的值。
  public Student(string n, string id) : base(n, id) {
    // 现在只需关心 Student 类需要进行的特定操作。
    Major = null;
  }

  // 带有 3 个参数的构造函数。
  // 见上面的注释。
  public Student(string n, string id, string m) : base(n, id) {
    Major = m;
  }
}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

与使用:this(...)语法在同一个类中重用构造函数代码相似，如果派生类的构造函数调用基类的构造函数，则当调用派生类的构造函数时会先调用基类的构造函数。

13.10.4　base( )的隐式调用

无论是否在派生类的构造函数中使用: base(...)语法来调用基类的构造函数，C#始终会试图执行所有祖先类的构造函数。在开始启动给定类的构造函数代码前，会按照类的层次结构，从一般到特殊，逐层调用它们的构造函数。

例如，当创建 Student 对象时，实际上同时创建了 Student 对象的 3 个层次。首先会使用 Object 的构造函数初始化 Object 部分，然后使用 Person 构造函数初始化 Person 类的成员，最后使用 Student 构造函数初始化 Student 的成员。Student 可看作是一个 Object、Person 和 Student。当调用 Student 构造函数时，首先会执行 Object 构造函数，接着是 Person 构造函数，最后才是 Student 构造函数。因此，如果不利用 base(...)语法来编写 Student 的构造函数，如下所示：

```
public class Student : Person
{
  private string major;

  // 带有两个参数的构造函数。
   // 此处没有调用任何特定基类的构造函数。
   public Student(string n, string id) {
    // 现在只需关心 Student 类需要进行的特定操作。
    major = null;
}
```

这就像编写以下代码一样(注意粗体字)，从而显式调用了无参数的 Person 构造函数：

```
public class Student : Person
{
  private string major;

  // 带有两个参数的构造函数。
  public Student(string n, string id) : base() {
    // 现在只需关心 Student 类需要进行的特定操作。
    major = null;
}
```

这是第一反应，因为继承表示的是“是一个”的关系——Student 是 Person，而 Person 是 Object——所以创建 Object 对象和 Person 对象必须完成的操作，在创建 Student 时也必须完成这些操作。如果定义了多个基类，则会调用哪个基类的构造函数？

要回答这个问题，需要讨论几种不同的情况。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

1. 情况 1：派生类没有声明自己的构造函数

如果派生一个类(如 Student)，但没有为该派生类定义任何构造函数，那么 C#会为该派生类提供默认的无参数的构造函数。当创建派生类的新对象时，派生类中的默认构造函数会隐式地调用基类中的无参数的构造函数。

下面的示例将无法通过编译(稍后将解释原因)：

```
public class Person
{
  private string name;

   // 带有一个参数的构造函数——只要创建这个构造函数，就不会有 Person 默认的(无参数
  // 的)构造函数。
   public Person(string n) {
     Name = n;
   }

   // 本例中没有无参数的构造函数。
}

public class Student : Person
{
  private string major;

   // 没有为 Student 显式地定义任何构造函数！所以将使用默认的无参数的构造函数。
}
```

当编译 Student 时，会得到以下错误消息：

```
error: 'Person' does not contain a constructor that takes '0' arguments
```

这是因为 C#编译器试图为 Student 类创建一个默认的无参数的构造函数。但若要这么做，编译器需要在 Student 默认构造函数中调用 Person 中无参数的构造函数——但是 Person 没有这样的构造函数！从本质上来说，编译器正试图为 Student 创建以下默认的构造函数：

```
public Student() : base() {
   // 初始化仅有“骨架”的 Student——细节从略。
}
```

对于这个难题有以下解决方法：

- 为 Person 类显式地编写无参数的构造函数，以取代那个丢失的默认 Person 构造函数。这样当创建默认的 Student 类的构造函数时，编译器就能利用它(这是首选方法)。
- 始终使用 Student 类的构造函数，通过 base 关键字显式地调用特定的 Person 构

专业图书网上第一专营店 www.china-pub.com 独家提供样章

造函数。

后面将介绍第二种方法。

2. 情况 2：派生类显式声明一个或多个构造函数

此类情况实际上可以分为以下两种子情况。

子情况 2A：派生类的构造函数没有显式地调用基类的构造函数

如果派生类的构造函数没有通过 base:(…)结构显式地调用基类的构造函数，则仍会调用基类中无参数的构造函数，如情况 1 中所述。和前面的原因相同，以下代码不能编译通过：

```
public class Person
{
  // 细节从略。

   // 带一个参数的构造函数——只要创建这个构造函数，就不会有 Person 默认的无参数的构
  // 造函数。
  public Person(string n) {
    Name = n;
  }
}

public class Student : Person {
  // 细节从略。
   // 声明 Student 构造函数，但是没有显式地调用特定的 Person 构造函数。
   public Student(string n, string m) {
    Name = n;
    Major = m;
  }
}
```

在编译 Student 时，会得到以下编译器的错误消息：

```
No constructor matching Person() found in class Person
```

子情况 2B：派生类的构造函数显式地调用基类的构造函数

通过 base(…)结构让派生类的构造函数显式地调用特定父类的构造函数，可以解决子情况 2A 中的问题：

```
public class Person
{
  private string name;

   // 带一个参数的构造函数——只要创建这个构造函数，就不会有 Person 默认的无参数的构
  // 造函数。
   public Person(string n) {
    name = n;
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    }

    // 没有为 Person 提供其他构造函数。
}

public class Student : Person
{
  private string major;

  // 构造函数。
  // 将使用一个参数显式地调用 Person 的构造函数，传入 n 的值。
  public Student(string n, string m) : base(n) {
    major = m;
  }
}
```

上面的代码编译正确！

13.10.5 对象初始化器

到目前为止，我们已经使用了构造函数为对象的字段提供初始值。对象初始化器(object initializer)是另一种给字段和对象属性赋值的方法，无需调用带参数的构造函数(它仍会调用无参数的构造函数)。字段值和属性值由逗号分隔并放入括号内部。例如，考虑 Person 类定义了两个字段：name 和 age。

```
public class Person
{
    private string name;
    private int age;

    // 构造函数
    public Person() {
        Name = "??";
        Age = 0;
    }

    public Person(string n, int age) {
        Name = n;
        Age = age;
    }

    // 属性
    public string Name {
        get {
            return name;
        }
        set {
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
                name = value;
            }
        }

        public int Age {
            get {
                return age;
            }
            set {
                age = value;
            }
        }

        // 其他Person类的成员从略。
    }
```

对象初始化器语法可用来创建Person对象，同时可使用以下语法给Name和Age属性赋值：

```
Person p = new Person{ Name="Emile Sanchez", Age=21 };
```

注意，对象初始化器语法和调用构造函数的语法不同，括号用来包含属性赋值语句。换句话说，需要提供属性的名称和其初始值。只有可访问的字段和属性才能用对象初始化器进行赋值。例如，对于拥有私有访问权限的name和age字段，在下面的对象初始化器中为其赋值：

```
Person p = new Person{ name="Emile Sanchez", age=21 };
```

会产生编译错误：

```
Error: 'Person.name' is inaccessible due to its protection level
```

对象初始化器的一个有用功能是可以初始化包含自动实现属性的对象——本章后面将会介绍这一功能。

13.11 再论方法

我们已经在本章和本书的第I部分讨论了很多关于方法的细节。现在将进一步讨论有关方法的更重要的内容。

13.11.1 消息链

在C#这样的面向对象的编程语言中，通常会通过点符号将一个方法调用和另一个方

专业图书网上第一专营店 www.china-pub.com 独家提供样章

法调用连接起来以形成复杂的表达式，这种机制称为消息链(message chaining)。下面是一个虚构的示例：

```
Student s = new Student();
s.Name = "Fred";

Professor p = new Professor();
p.Name = "John";

Course c = new Course();
c.Name = "Math";

s.setFacultyAdvisor(p);
p.setCourseTaught(c);

Course c2 = new Course();

// 消息链。
c2.Name = "Beginning " + (s.GetFacultyAdvisor().GetCourseTaught().Name);
```

如第 1 章所述，表达式的计算是按照从最里面的括号到最外面的括号，从左到右的顺序。现在计算最后一行中的表达式：

(1) 查找嵌套括号中最深的一层，发现表达式最深有两层括号，所以先计算最左边的子表达式——即在 Students 的引用 s 上调用 GetFacultyAdviser 方法，该方法返回 Professor p 的引用：

```
s.GetFacultyAdvisor()
```

(2) 接着，在 Professor 引用 p 上调用 GetCourseTaught( )方法，其返回对 Course c 的引用：

```
p.GetCourseTaught()
```

(3) 然后访问 Course 对象的 Name 属性，其返回字符串值“Math”：

```
c.Name
```

(4) 现在已经完成最深层括号中表达式的计算，得到以下等价的表达式：

```
c2.Name = "Beginning " + "Math";
```

因此，从最后的分析中可以看出，这个复杂表达式的结果是将名称“Beginning Math”赋值给 Course 对象 c2 的 Name 属性。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.11.2 方法隐藏

在第 5 章中曾介绍过，派生类可以重写在基类中声明为 virtual 的方法：派生类可用相同的签名并在方法中包含 override 关键字的方法来声明新版本的基类方法。

即使没有在基类中使用 virtual 声明方法，还有一种方法也可用来替换基类方法的逻辑，即通过方法隐藏(method hiding)的技术。

要隐藏基类方法，派生类必须在方法声明中使用 new 关键字替代 override 关键字，从而使用相同的方法签名来定义方法。下面这个示例中的派生类 Student 隐藏了首先在 Person 基类中声明的 PrintDescription 方法：

```
public class Person
{
    // 细节从略

    // 注意：没有 virtual 关键字——Person 类的设计者没有提供可被重写的方法。
    public void PrintDescription() {
      // 细节从略
    }
}

public class Student : Person
{
    // Student 类通过使用 new 关键字在 Person 类中隐藏 PrintDescription( )方法。
    public new void PrintDescription() {
      // 细节从略
    }
}
```

任何基类方法——无论使用 virtual 关键字与否——都能被隐藏。因此，如果想要修改派生类从基类继承而来的方法，但又发现基类的方法签名中没有包含 virtual 关键字，则可以使用这种方式来重写方法。注意，隐藏非虚基类的方法和重写虚基类方法有显著的差别，这与第 7 章讨论的多态概念有关。下面将讨论这个问题。

1. 方法隐藏和多态

第 7 章曾介绍过，多态(polymorphism)是指属于不同类的不同对象以类特定的方式响应同一个方法调用。例如，假设 ChooseMajor 是 Student 类的虚方法，而且该方法被 GraduateStudent 和 UndergraduateStudent 派生类重写，根据 Student s 的类型，以下代码会在 s 上选择调用何种版本的 ChooseMajor：

```
// 遍历 Students 的 List。
List<Student> students = new List<Student>;
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// List 的其他细节从略。
foreach( Student s in students ) {

    // 下一行代码是多态的：如果 s 在运行时引用 GraduateStudent，则会执行
GraduateStudent
    // 版本的 ChooseMajor 方法；如果 s 在运行时引用 UndergraduateStudent，则会执行
    // UndergraduateStudent 版本的 ChooseMajor 方法。
    s.ChooseMajor();
}
```

相对而言，给定对象运行的隐藏方法会在编译时“锁定”。现在假设 ChooseMajor 是 Student 类中的非虚方法，而且被 GraduateStudent 和 UndergraduateStudent 派生类隐藏(不是重写)。下面对 ChooseMajor 的两次调用会分别使用派生类的方法，因为我们分别是在显式声明为 GraduateStudent 和 UndergraduateStudent 的引用变量上调用该方法：

```
GraduateStudent g = new GraduateStudent();
g.ChooseMajor(); // 将执行 GraduateStudent 版本的方法
UndergraduateStudent s = new UndergraduateStudent();
s.ChooseMajor(); // 将执行 UndergraduateStudent 版本的方法
```

多态在下面的示例中没有起作用——因为在编译时 s 被声明为 Student 类型，所以会为 GraduateStudent 和 UndergraduateStudent 都执行 Student 基类版本的 ChooseMajor 方法：

```
// 遍历 Students 的 List。
foreach( Student s in students ) {

    // 下一行代码不是多态的：
    // 无论 s 在运行时引用的是 GraduateStudent 还是 UndergraduateStudent，都会
执
   // 行 Student 版本的 ChooseMajor 方法。
    s.ChooseMajor();
}
```

前面已提过，无论方法使用 virtual 关键字与否，都可以隐藏任何方法。为什么明知不能实现多态，还要对虚方法使用隐藏技术而不是重写技术呢？因为隐藏方法通常比虚方法性能更好(运行更快)。

2. 关于方法隐藏的最后注意事项

最后要注意以下几点：

- 和重写一样，在新的派生/隐藏的方法中可使用 base 关键字调用原始基类版本的隐藏方法。
- 派生类的“隐藏”方法可以有不同的返回类型，这些返回类型不同于基类的返回类型。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

- 抽象(abstract)方法不能被隐藏。

13.12　再论属性

第 4 章介绍过，属性是获得或设置私有访问字段值的一种方法。使用属性就像使用公有数据成员一样，但在幕后它们将使用称为存取器(accessor)的特殊方法类型。本节将介绍更有用且更强大的属性功能。

13.12.1　非对称的可访问性

属性的 get 和 set 存取器默认拥有与属性相同的访问权限。例如，以下定义的属性将用于访问 Student 类的 id 字段：

```
public string Id {
  get {
    return id;
  }
  set {
        id = value;
  }
}
```

在上面这个示例中，Id 属性被声明为 public，所以 get 和 set 存取器默认也被声明为 public。这种情况可能并非我们所需。例如，您可能想要将 set 存取器('它能修改 id 字段值)的访问权限只限于 Student 类和其派生类。

使用非对称访问性(asymmetric accessibility)能够为属性的两个存取器提供不同的访问权限。通常，get 存取器是 public 访问权限，而 set 存取器会有不同的可访问性。要修改上面这个示例使得只有 Student 类及其派生类才能修改 id 字段值，则可将 set 存取器设为 protected 访问权限。

```
public string Id {
    get {
        return id;
    }
    protected set {
        id = value;
    }
}
```

非对称可访问性只适用于为属性同时定义 get 和 set 这两个存取器并且只有一个存取

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

器的访问权限要和属性的访问权限不同的情况。

13.12.2 自动实现的属性

属性的 get 和 set 存取器都是方法，如果需要，可在存取器中放入复杂的编程逻辑。然而，属性通常只用来访问或修改字段值。例如，Person 类可能会声明表示人名的字段和用来访问或修改 name 字段值的属性：

```
public class Person
{
    private string name;

    // 构造函数
    public Person (string name) {
        Name = name;
    }

    // 访问 name 字段的属性
    public string Name {
        get {
             return name;
        }
    set {
        name = value;
    }
}
// 其他 Person 类的成员从略。
}
```

假设执行以下代码：

```
Person p = new Person("Jackson");
Console.WriteLine("Name is "+p.Name);
```

则会产生以下输出：

```
Name is Jackson
```

如果存取器中没有其他逻辑而只是访问字段值，则可通过自动实现属性(auto-implemented property)来简化属性语法。这样可以省略存取器的结构体，而只使用 get 和 set 关键字。

```
public class Person
{
// 删除了 name 字段!
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
// 构造函数
public Person (string name) {
Name = name;
}

// 自动实现的属性
public string Name { get; set; }

// 其他 Person 类的成员从略。
}
```

假设运行相同的客户代码：

```
Person p = new Person("Jackson");
Console.WriteLine("Name is "+p.Name);
```

其输出与前面示例的输出相同，即：

```
Name is Jackson
```

查看这两个版本的 Person 类，会发现最大的区别在于自动实现的属性可用来存储和访问人名，而无需声明 name 字段。发生这种差异是因为自动实现的属性是同一类型的私有、匿名(即隐藏)字段。私有隐藏字段的值可以通过属性的 get 和 set 存取器来访问。

事实证明，可以完全删除 Person 的构造函数，并且可以使用对象初始化器的语法来创建 Person 对象：

```
Person p = new Person { Name="Jackson" };
```

如果删除构造函数，Person 类的代码清单将变得相当简单：

```
public class Person
{
    // 自动实现的属性
    public string Name { get; set; }
}
```

最后要注意的是，自动实现的属性必须定义 get 和 set 存取器。要让属性只读，可以将 set 存取器设为私有访问权限：

```
// Name 属性是只读的
public string Name { get; private set; }
```

在第 14 章中编写 SRS 类的代码时，会使用自动实现的属性。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.12.3 回顾重写和抽象类

第 7 章曾介绍过，通过为抽象基类声明的所有抽象方法提供非抽象实现，可以从抽象类中派生出具体的类。结果证明，派生类还能使用抽象方法重写在基类中声明的非抽象方法。下面将介绍使用预定义 C#类的示例。

所有 C#类都隐式地从 System 名称空间中的 Object 类派生而来。Object 类中声明的 ToString 方法，是一个非抽象的虚方法，本章前面已经讨论过该方法。我们可以定义一个隐式派生自 Object 的 Person 类(所有类都从 Object 派生)，但它会使用一个抽象版本来重写继承自 Object 类的非抽象 ToString 方法：

```
// 隐式派生自 Object 类。
public abstract class Person
{
   // 使用一个抽象版本重写 Object 类中非抽象的 ToString 方法。
   public abstract override string ToString();

  // 其他 Person 类的细节从略。
}
```

有关 Person 类还要注意以下几点：

- 本章前面已经讨论过，从 Object 类的派生是隐式的，所以不需要在 Person 类的声明中使用: Object 语法。
- 第二，因为 ToString 方法在 Person 类中定义为抽象方法，所以 Person 类本身被声明为抽象类。
- 最后，因为 Person 的 ToString 方法重写了 Object 类的 ToString 方法，所以要在方法声明中使用 override 关键字。

为何要用抽象方法来重写非抽象方法？答案是强迫 Person 类的派生类实现该类特定的 ToString 方法。也就是说，我们不希望 Person 类的派生类太“懒”，只是继承 Object 类中 ToString 方法的默认行为——我们希望能有效地“抹去”其具体实现。当在第 14 章中创建 SRS 应用程序的 Person 类时会这样做。Person 类将声明一个抽象的 ToString 方法，它的派生类会重写该方法。

13.13 对象标识

在前面的示例中，GraduateStudent 和 UndergraduateStudent 对象都存储在 List 集合中，List 集合被声明为保存 Student 对象。从 List 中提取的元素都将返回为一个 Student 对象。提取的对象没有忘记它属于哪个类。对象始终会记住其类标识，这将方便我们用不同类型的引用变量来引用对象，而且会帮助编译器判断对象应该响应哪些消息。我们将使用以下示例帮助理解这一点。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.13.1 派生类的对象也是基类的对象

如前所述，如果实例化 Professor 对象，则它也是 Person 类型以及 Object 类型，图 13-4 说明了 Professor 对象在内存中的分布情况。

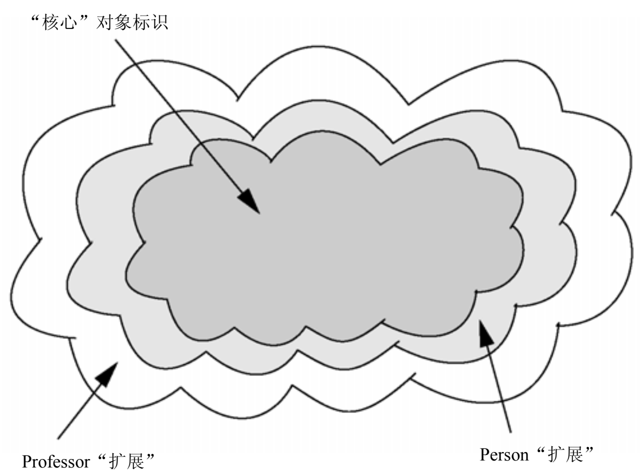

图 13-4 Professor 对象同时也是 Person 对象和 Object 对象，三位一体！

我们可以创建不同类型(该对象派生链中的任何类型)的引用变量来存储 Object/Person/Professor 的引用，每个引用变量都将引用对象的不同“呈现(rendition)”：

```
// 创建 Professor 对象，并维护 3 种类型的引用。
Professor pr = new Professor();
Person p = pr;
object obj = pr;
```

如果通过对象引用 obj 来引用该 Professor 对象，就编译器而言，仅存在 Object 的“核心”，而对象的 Person 和 Professor“扩展”都存在问题，如图 13-5 所示。

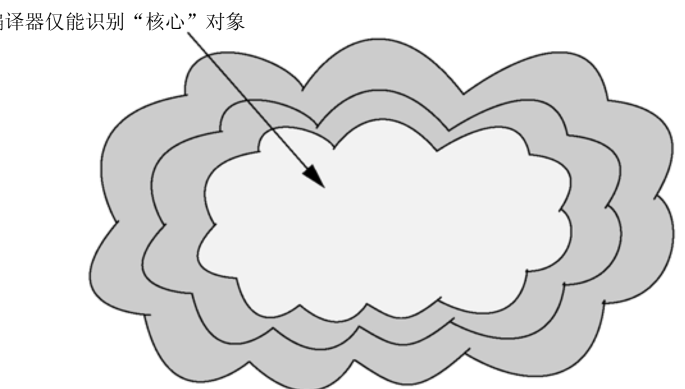

图 13-5 编译器只知道 Professor 对象中的 Object 内容

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

所以，编译器会拒绝访问 obj 中由 Professor 和 Person 定义的成员：

```
// 编译器将拒绝在 obj 上调用为 Professor 类声明的 AddAdvisee()方法，因为虽然能从代
// 码中知道它指向 Professor(其句柄存储为 Object)，但编译器不能确定这一点。
obj.AddAdvisee(); // 编译器错误

// 然而，可以调用 Professor 类从 Object 类继承的任何方法：
obj.ToString();
```

如果通过 Person 的引用 p 来引用 Professor 对象，对编译器而言，仅存在对象的 Object "核心"与 Person "扩展"，而对象的 Professor 扩展存在问题，如图 13-6 所示。

```
// 编译器将拒绝在这个对象上调用为 Professor 类声明的 AddAdvisee()方法，因为虽然能
// 从代码中知道它指向 Professor(其引用在运行时存储为 Person)，但编译器不能在运行时
// 确定这一点。
p.AddAdvisee(); // 编译器错误

// 然而，可以调用 Professor 类从 Object 类继承而来的任何方法……
p.ToString();

// ……或者调用从 Person 类继承而来的任何方法
p.Name;
```

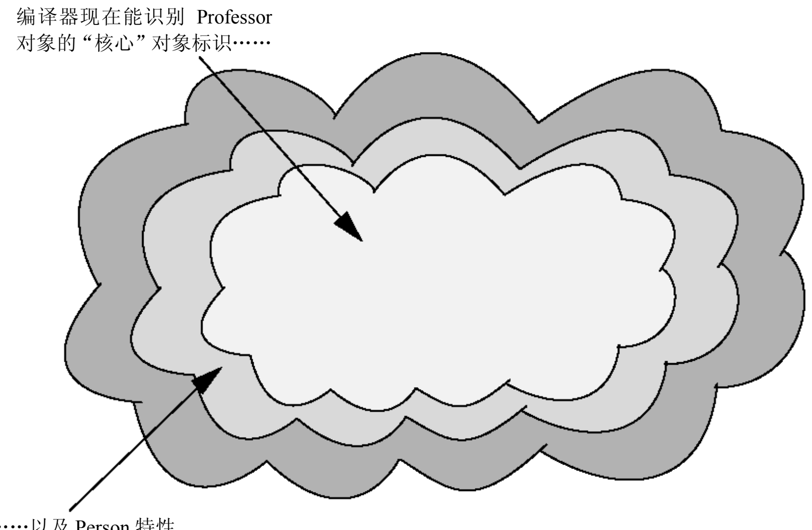

图 13-6 编译器现在能识别 Professor 对象的 Object "核心"和 Person "扩展"

只有通过 Professor 的引用变量 pr 指向对象，编译器才允许访问对象中 Professor 特有的成员。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.13.2 确定对象所属的类

有几种方法能在运行时确定对象所属的类，可以利用：

- Object 类定义的 GetType 方法
- typeof 操作符
- is 关键字

1. GetType 方法

前面介绍过，通过从 Object 类中派生，每个对象都能继承一个方法：

```
public Type GetType()
```

当在对象上调用该方法时，方法将返回类型为 Type 的对象，表示调用该 GetType 方法的对象类型。例如，如果 Person 类属于 SRS 名称空间，则下面对 GetType 的调用将得到表示 SRS.Person 类的 Type 对象：

```
Person p = new Person();
Type t = p.GetType();
```

Type 类定义了一个名为 FullName 的字符串属性，可用来访问类型的完全限定名称：

```
Person p = new Person();
Type t = p.GetType();
String s = t.FullName; // s 现在等于 SRS.Person。
```

结合使用该方法和属性，可以找出对象所属的类，如下所示：

```
reference.GetType().FullName;
```

例如：

```
using System;
using System.Collections;

public class CastingExample
{
  static void Main() {
    Student s = new Student();
    Professor p = new Professor();

    ArrayList list = new ArrayList();
    list.Add(s);
    list.Add(p);
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    for (int i = 0; i < list.Count; i++) {
      // 注意这里没有强制转换对象!
      // 这里得到的是泛型对象。
      object o = list[i];
      Console.WriteLine(o.GetType().FullName);
    }
  }
}
```

该程序将产生以下输出(假设 Student 和 Professor 都不属于某个已命名的名称空间):

```
Student
Professor
```

这说明了对象能够记住它们自己的“根”。

2. typeof 操作符

另一种判断给定对象引用所属类的方法是通过 typeof 操作符。该操作符接受类型名称(注意，类型名称不在引号中)作为参数并返回相应的 Type 对象。以下示例说明了如何使用 typeof 操作符来判断引用的类型(因为 C#语言中的每种类型，无论是用户自定义类型还是预定义类型，都仅有一个 System.Type 对象，所以==操作符能在本例中使用)。

```
Student s = new Student();
// 判断引用变量 s 的类型是否为 Student 类型。
if (s.GetType() == typeof(Student)) {
Console.WriteLine("s is a Student");
}
```

上面的代码将产生以下输出：

```
s is a Student
```

注意，如果想要引用属于某个显式命名的名称空间中的类名，而且没有在代码中包括合适的 using 语句，则必须在使用 typeof 时使用类的完全限定名称：

```
//假设 Bar 是 Foo 名称空间中的类。
if (x.GetType() == typeOf(Foo.Bar) {...}
```

注意

虽然使用括号包含类名，如 typeof(Student)，会使 typeof 看上去像是一个方法，但它实际上是一个操作符。

专业图书网上第一专营店 www.china-pub.com 独家提供样章

3. is 操作符

确定对象所属类型的第 3 种方法是使用 is 操作符，它产生的代码比使用 typeof 操作符产生的代码更清楚。例如，如果上面的示例使用 is 操作符：

```
Student s = new Student();

// 判断 s 引用变量的类型是否为 Student 类型。
if (s is Student) {
  Console.WriteLine("s is a Student");
}
```

13.14 对象删除和垃圾回收

C++语言中存在 delete 操作符，它能在不需要某个动态分配的对象时，允许程序员显式地控制对象，从而回收内存。这样做有利有弊！其优点是让 C++程序员绝对控制程序中的内存资源。但是如果 C++程序员忘记回收对象，那么程序会逐渐耗尽内存——这称为内存泄漏(memory leak)。

然而 C#没有 delete 操作符：任何动态创建的对象——即除了值类型以外的所有引用类型——当消除所有对它的引用时，那么它就会成为 C#垃圾回收的目标。就像在第 3 章的“对象氢气球”示例中的讨论一样，当我们松开气球上的所有系绳时，气球就飘走了。垃圾回收是.NET 公共语言运行库(CLR)的自动功能：当对象被垃圾回收时，会回收为该对象分配的内存，并把内存放回内存池中，这样可在运行时用于新对象的创建。

有多种方法可以释放对象上的句柄，从而使它们成为垃圾回收的目标，如下面的示例所示：

```
// 声明两个引用变量保存 Student 对象
Student s1;
Student s2;

// 创建 Student 对象，并在变量 s1 中存储到该对象的引用。
s1 = new Student();

// 将句柄复制到 s2 中，这样在同一对象上就有了两个句柄。
s2 = s1;

// 现在介绍两种不同的删除对象引用的方法。

// 可将保存句柄的变量设置为 null。
s1 = null; // (对象仍有一个引用)

// ……或者可将其他对象的句柄传给变量，使它放弃第一个句柄。通过创建第二个 Student
// 对象并将其传给 s2，s2 放弃了唯一指向第一个 Student 对象的句柄。
s2 = new Student( ); // 原来的 Student 对象上再也没有指向它的句柄了！
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

我们已经删除了原来 Student 实例上的引用。如果这个对象上没有任何引用，它就会成为垃圾回收的候选目标。之所以强调是候选目标，是因为垃圾回收器不会立即回收这个对象。只有在运行时决定需要回收资源时，垃圾回收器才会运行；也就是说，当应用程序可用于分配新对象的可用内存较少时，才会开始回收资源。因此，在一段时间内，Student 对象还在内存中，只是没有任何句柄指向它。

C#中的垃圾回收从根本上解决了内存泄漏的问题。注意，运行时仍有可能会用完内存，比如过多的对象上保存了过多的句柄，因此 C#程序员不能完全忽略内存管理——但和 C/C++相比，犯错误的机会将少很多。

13.15 属性

有些面向对象的书籍会使用属性(attribute)来表示类中的数据成员。本书一直使用术语字段(field)，因为属性在.NET 中有非常特殊的含义。虽然在建立 SRS 中不会使用.NET 的属性，但至少在此要做个简单介绍。

.NET 属性是派生自 System.Attribute 类的引用类型，用于给其他编程元素分配元数据标签(metadata tags)——即关于这些元素的描述信息。

通过在代码元素前的方括号中放入属性名，属性可应用于任何代码元素(类、构造函数、委托、方法、字段等)。例如，System.Array 类有名为 SerializableAttribute 的相关属性，如下所示：

```
[Serializable()]
public abstract class Array:ICloneable,IList,ICollection,IEnumerable {...}
```

向 Array 类应用这个属性说明 Array 对象是可序列化的(serializable)，表示可以通过二进制的形式存储到硬盘或从硬盘中恢复。

作为程序员，当然我们也可以通过查阅 C#语言文档知道 Array 是可序列化的。属性有何妙处？通过反射(reflection)机制，就能在运行时编程“发现”，“知识”通过属性的应用来进行传递(有关反射的工作原理超出了本书的讨论范围)。

可以将 FCL 提供的预定义属性应用于代码。例如，可以在想要标识为丢弃的方法头前面放入预定义的 ObsoleteAttribute 属性，以说明将来可能会停止使用该方法：

```
public class MyClass
{
   // 向 Foo()添加属性。
   [Obsolete()]
  public void Foo() {
    // 方法细节从略……
  }

  // 其他细节从略。
}
```

专业图书网上第一专营店 www.china-pub.com 独家提供样章

假设客户代码会在 MyClass 的实例上调用 Foo 方法：

```
public class AttrDemo {
  static void Main() {
    MyClass x = new MyClass();

     // 调用被标识为丢弃的方法。
     x.Foo();
  }
}
```

当编译 AttrDemo 类时，编译器会发现在某个对象上调用的方法被标记为丢弃，并将生成以下提醒：

```
warning CS0612: 'MyClass.Foo()' is obsolete
```

程序仍将执行，但会提醒程序员：不建议使用 Foo 方法。

注意

还可以根据需要(如调试)为任意特定的元数据创建用户自定义的属性，但本书不会介绍这些内容。

13.16 本章小结

本章快速介绍了 C#语言！虽然还有很多 C#的内容未介绍，但您已经具备所有基本知识，可用于理解和实践本章后面将要建立的 SRS 应用程序。

本章介绍了以下内容：

- C#名称空间的概念，以及如何用名称空间将类和接口分解成逻辑单元
- string 的对象特征以及操作字符串的方法
- C#中所有类型的父类：Object 类
- 深入理解 Array 的对象特征
- 使用 C#集合类(List 和 Dictionary)的方法，以及如何使用 foreach 循环遍历集合中的内容
- 变量初始化的一些细节
- 关于 Main 方法的进一步讨论
- 在创建对象时使用构造函数初始化对象的字段以及如何使用对象初始化器完成相同的操作
- 自动实现的属性如何简化代码清单
- C#语言中的继承，特别是成员的可见性如何影响其派生类使用成员的方式、如

专业图书网上第一专营店 www.china-pub.com 独家提供样章

何通过 base 关键字重用基类的行为以及关于构造函数和继承的复杂性

- C#中对象标识的特性，如何发现对象所属的类以及如何测试两个 C#对象是否相同
- 如何动态删除已创建的对象，从而在运行时回收其内存，C#垃圾回收器在回收内存中所起的作用

具备了 C#的知识后，下面就能开始建立 SRS 应用程序。

提示

如果还没有下载本书后面章节中的相关代码，则建议您从 Apress 网站上进行下载！具体操作参见附录 B。

13.17 练习

(1) 按照附录 A 中的操作，下载最新版本的 Microsoft .NET Framework 并在您的计算机上运行。

(2) 编写能按逆序输出整数 1~10 的 C#程序。

(3) 使用自动实现的属性重新编写“重写 Equals 方法”一节中的代码，修改测试代码以使用对象初始化器语法初始化 Student 对象。

(4) 编写一个简单程序，创建通用的 List 集合并填入随机的字符串。使用 List 类中的方法将字符串按照字母表进行排序。使用 foreach 循环进行遍历并输出排序后的字符串。