规范

避免使用常规catch块，而应该使用捕捉System.Exception的catch块来代替。
避免捕捉无法获知其正确行动的异常。对这种异常不进行处理比处理地不正确要好。
避免在重新引发前捕捉和记录异常。要允许异常逃脱，直至它被正确处理。

4.9.2 使用throw语句报告错误

C#允许开发人员从代码中引发异常，代码清单4-23和输出4-15对此进行了演示。

代码清单4-23 引发一个异常

```
using System;
class ThrowingExceptions
{
  static void Main()
  {
      try
      {

          Console.WriteLine("Begin executing");
          Console.WriteLine("Throw exception...");
          throw new Exception("Arbitrary exception");
          Console.WriteLine("End executing");
      }
      catch (FormatException exception)
      {
          Console.WriteLine(
              "A FormateException was thrown");
      }
      catch(Exception exception)
      {
          Console.WriteLine(
              "Unexpected error:  {0}", exception.Message);
      }
      catch
      {
          Console.WriteLine("Unexpected error!");
      }

      Console.WriteLine(
          "Shutting down...");
  }
}
```

输出4-15

```
Begin executing
Throw exception...
Unexpected error:  Arbitrary exception
Shutting down...
```

如代码清单4-23中的箭头所示，如果引发一个异常，会使执行从异常的引发点跳转到与引发的异常类型兼容的第一个catch块[①]。在本例中，是第二个catch块处理引发的异常，它在屏幕上输出一条错误消息。这个程序没有finally块，所以随后会执行try/catch块之后的System.Console.WriteLine()语句。

为了引发异常，需要有Exception的实例。代码清单4-23使用关键字new，后跟异常的数据类型，从而创建了这样的实例。大多数异常类型都允许在引发该类型的异常时传递消息，以便在发生异常时获取消息。

有时catch块能捕捉到异常，但不能正确或者完整地处理它。在这种情况下，可以让这个catch块重新引发异常，具体的办法是使用一个单独的throw语句，不要在它后面指定任何异常，如代码清单4-24所示。

代码清单4-24 重新引发一个异常

```
...
        catch(Exception exception)
        {
            Console.WriteLine(
                "Rethrowing unexpected error:  {0}",
                exception.Message);
            throw;
        }
...
```

注意，在代码清单4.24中，throw语句是“空”的，没有指定exception变量所引用的异常。区别在于，throw;保持了异常中的“调用栈”信息，而throw exception;将那些信息替换成当前调用栈信息。而调试时一般需要知道原始调用栈。

规范

要在捕捉并重新引发异常时使用空的throw语句，以便保持调用栈。

要通过引发异常而不是返回错误码来报告执行失败。

不要让公共成员将异常作为返回值或者out参数。**要**通过异常来指明错误；**不要**通过它们作为返回值来指明错误。

避免使用异常处理来处理预料之中的情况

开发人员应当尽量避免为预料之中的情况或者正常的控制流引发异常。例如，开发者事先应该料到用户可能在输入年龄的时候输入无效的文本[②]。所以，不要用异常来验证用户输入的数据。相反，应该在尝试转换前对数据进行检查（甚至可以考虑从一开始就防止用户输入无效的数据）。异常是专门为了跟踪例外的、事先没有预料到的、而且可能造成严重后果的情况而设计的。为预料之中的情况使用异常，会造成代码难以阅读、理解和维护。

除此之外，和大多数语言一样，C#在抛出异常时会产生些许性能损失——相较于大多数操作都是纳秒级的速度，它可能造成毫秒级的延迟。人们平常注意不到这个延迟——除

① 从技术上说，也可能被一个兼容的catch筛选器捕捉。

② 通常，开发者必须假定用户会采取非预期的行为，所以应当防卫性地编写代码，提前为所有能想到的“愚蠢的用户行为”拟定对策。

非异常没有得到处理。例如，执行代码清单4-19的程序，并输入一个无效的年龄，由于异常没有得到处理，所以当“运行时”在环境中搜索有一个可以加载的调试器时，你会感觉到明显的延迟。幸好，程序都在关闭时即使性能较差，一般也不会引起人们太多的关注。

规范

不要用异常来处理正常的、预期的情况；用异常处理异常的、非预期的情况。

高级主题：使用TryParse()来执行数值转换

Parse()方法的问题在于，要知道转换能否成功，唯一的办法就是尝试执行类型转换，并在失败的时候引发并捕捉异常。因为异常处理代价相当高，所以更好的办法是尝试执行转换，同时不进行异常处理。在C#的第一个版本中，唯一支持这样做的是一个double类型的方法，名为double.TryParse()。然而，在CLI 2.0版本中，CLI为所有基本数值类型都实现了这个方法。它要求使用out关键字，因为从TryParse()返回的是bool值，而不是转换好的值。以下代码演示了如何使用int.TryParse()来尝试转换：

```
...
if (int.TryParse(ageText, out age))
{
    System.Console.WriteLine(
        "Hi {0}!  You are {1} months old.", firstName,
        age * 12);
}
else
{
    System.Console.WriteLine(
        "The age entered ,{0}, is not valid.", ageText);
}
...
```

从.NET Framework 4开始，枚举类型也添加了TryParse()方法。

有了TryParse()方法，就不必仅仅为了处理从字符串向数值的转换就动用try/catch块了。

4.10 小结

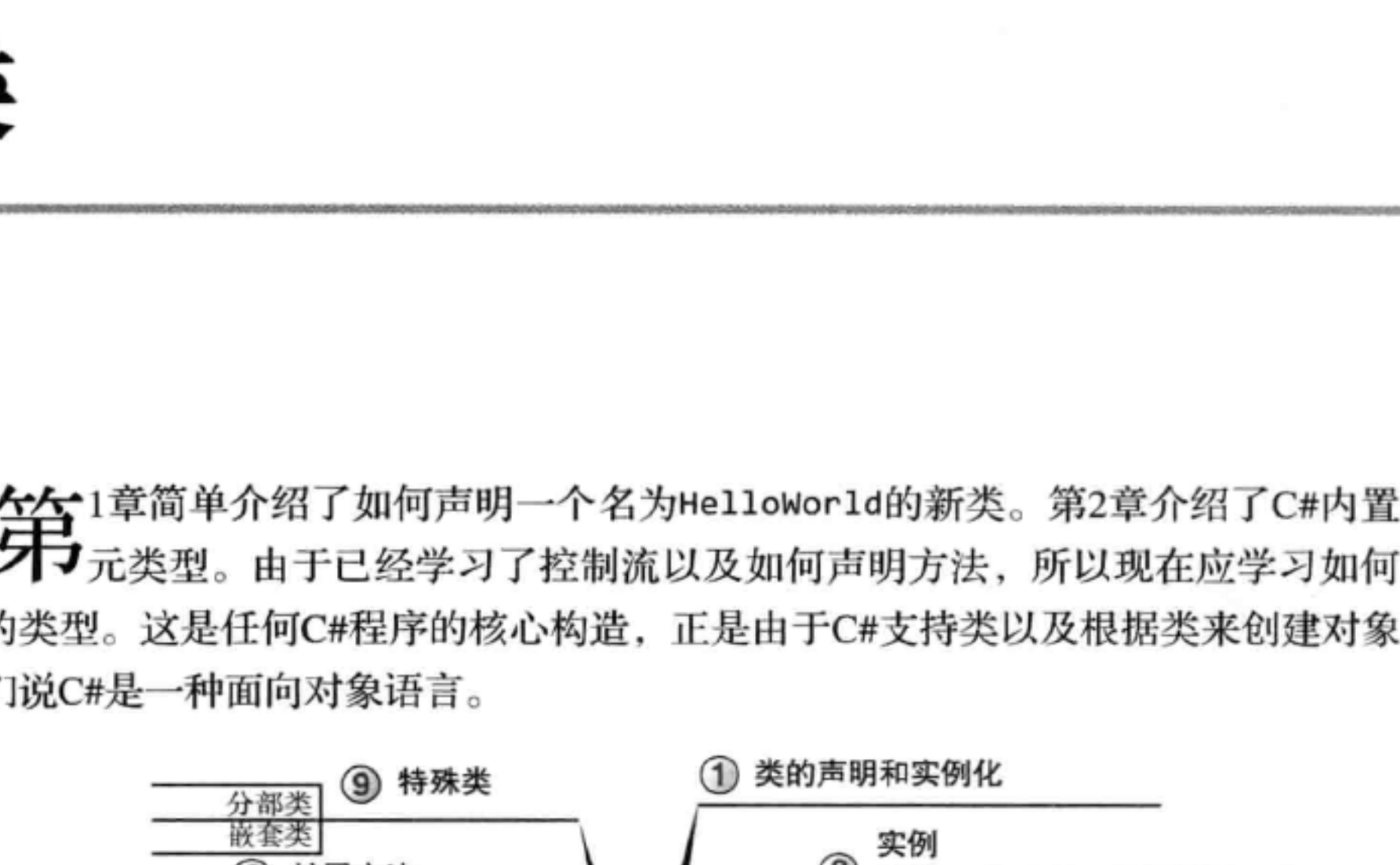
本章讨论了方法的声明和调用细节，包括如何使用关键字out和ref传递变量而非其值。除了方法声明，本章还介绍了基本的异常处理机制。

为了写出容易理解的代码，要利用“方法”这种基本的编程单元。但是，不要在一个方法中包含大量语句，而应当利用方法为编写的代码“分段”，一个方法通常不要超过10行代码。将一个较大的任务分解成多个较小的子任务是代码重构的一种手段，目的是使代码更容易理解和维护。

下一章开始讨论类，并解释它如何将方法（行为）和字段（数据）封装为一个整体。

第5章 类

第1章简单介绍了如何声明一个名为HelloWorld的新类。第2章介绍了C#内置的基元类型。由于已经学习了控制流以及如何声明方法，所以现在应学习如何定义自己的类型。这是任何C#程序的核心构造，正是由于C#支持类以及根据类来创建对象，所以我们说C#是一种面向对象语言。



本章利用C#介绍面向对象编程的基础知识。重点在于如何定义类，可以将类理解成对象的模板。

在面向对象编程中，之前学过的所有结构化编程构造仍然适用。然而，将那些构造封装在类中，可以创建更大、更有条理以及更容易维护的程序。从结构化的、基于控制流的程序转向面向对象的程序，是思维模式发生的一个根本性变化，因为面向对象编程提供了

一个额外的组织层次。结果是较小的程序在某种程度上得到了简化。但更重要的是，现在更容易创建较大的程序，因为程序中的代码得到了更好的组织。

面向对象编程的关键优势之一是不需要完全从头创建新的程序。而是可以将现有的一系列对象组装到一起，并用新的功能扩展类，或者添加更多的类。

还不熟悉面向对象编程的读者应该阅读“初学者主题”，以便对它有一个初步的理解。“初学者主题”以外的内容将着重讨论如何使用C#进行面向对象编程，并假定读者已经熟悉了面向对象的概念。

为了支持封装，C#必须支持类、属性、访问修饰符以及方法。本章着重讨论前3种构造，因为方法已在第4章进行了讨论。掌握这些基础知识之后，第6章将讨论如何通过面向对象编程实现继承和多态性。

初学者主题：面向对象编程

如今，成功编程的关键在于提供恰当的组织和结构，以满足大型应用程序的复杂需求。面向对象编程能够很好地解决问题。开发人员一旦熟悉了面向对象编程，除非写一些极为简单程序，否则很难回到结构化编程。

面向对象编程最基本的构造是类。一组类构成了编程抽象、模型或者模板，通常对应于一个现实世界的概念。例如，OpticalStorageMedia（光学存储媒体）类可能有一个Eject()方法，该方法会让光盘从播放器中弹出。OpticalStorageMedia类是现实世界的CD/DVD播放器对象的编程抽象。

类是面向对象编程的3个主要特征（封装、继承和多态性）的基础。

封装

封装的目的是隐藏细节。在必要的时候，细节仍然可以访问，但通过巧妙地封装细节，大的程序变得更容易理解，数据不会因为不慎而被修改，代码也变得更容易维护（因为对一处代码进行修改所造成的影响被限制在封装的范围之内）。方法就是封装的一个例子。虽然可以将代码从方法中拿出，把它们直接嵌入调用者的代码中，但将特定的代码重构成一个方法，能享受到封装所带来的好处。

继承

考虑这个例子：DVD是光学存储媒体的一个类型。它具有特定的存储容量，能容纳一部数字电影。CD也是光学存储媒体的一个类型，但它具有不同的特征。CD上的版权保护有别于DVD的版权保护，两者的存储容量也不同。无论CD还是DVD，它们都有别于硬盘、U盘和软盘。虽然所有这些都是“存储媒体”，但分别具有不同的特征——即使一些基本功能也是不同的，比如所支持的文件系统，以及媒体的实例是只读还是可读可写的。

面向对象编程中的继承允许在这些相似但又不同的物件之间建立“属于”（is a）关系。可以合理地认为DVD和CD都“属于”存储媒体。因而，它们都具有存储能力。类似地，CD和DVD都“属于”光学存储媒体，后者又“属于”存储媒体。

为上面提到的每种存储媒体类型都定义一个类，就得到一个**类层次结构**，它由一系列“属于”关系构成。例如，可将基类型（所有存储媒体都从它派生）定义成StorageMedia（存储媒体）。CD、DVD、硬盘、U盘和软盘都属于StorageMedia。但是，CD和DVD不必直接从StorageMedia派生。相反，可以从中间类型OpticalStorageMedia（光学存储媒体）派生。可以用一幅UML（Unified Modeling Language，统一建模语言）风格的类关系图

来描述类层次结构，如图5-1所示。

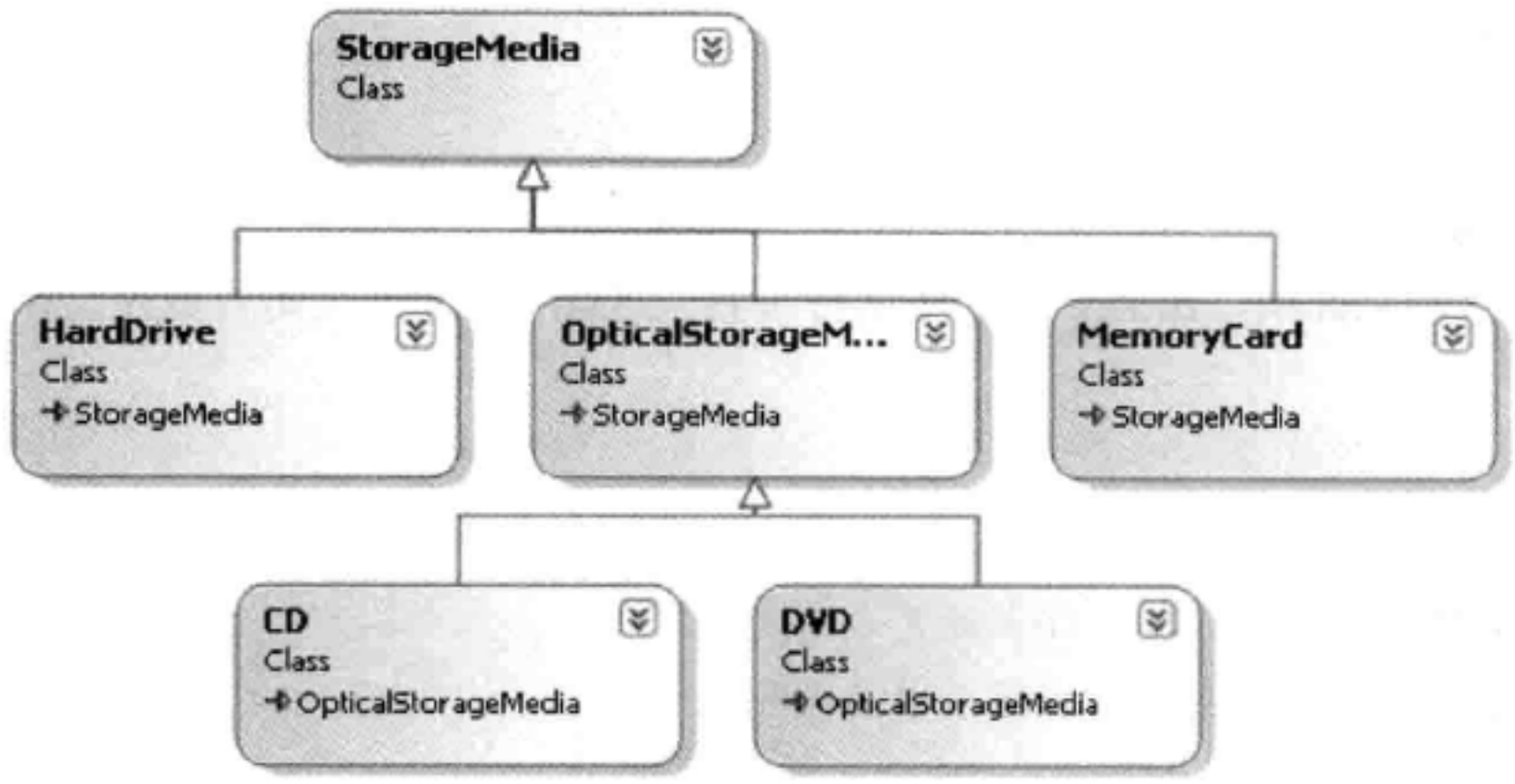


图5-1 类层次结构

继承关系至少涉及两个类，其中一个是另一个的更具体的版本。在图5-1中，HardDrive是更具体的StorageMedia。虽然更特化的类型HardDrive属于StorageMedia，但反之不成立，因为StorageMedia的一个实例并非肯定是HardDrive。如图5-1所示，继承涉及的类可能不止两个。

更具体的类型称为**派生类型**或者**子类型**。更常规的类型称为**基类型**或者**超类型**。也经常将基类型称为“父”类型，将派生类型称为它的“子”类型。虽然这种说法很常见，但可能会带来混淆。“子”毕竟不是一种“父”！本书将采用“派生类型”和“基类型”的说法。

为了从一个类型**派生**或**继承**，需要对那个类型进行**特化**，这意味着要对基类型进行自定义，为了满足特定的需要而调整它。基类型可能包含所有派生类型都适用的实现细节。

继承最关键的一点是所有派生类型都继承了基类型的成员。派生类型中可以修改基类型的成员，但无论如何，派生类型除了自己显式添加的成员，还包含了基类型的成员。

派生类型确保了用一致性的层次结构组织类。在这个层次结构中，派生类型比它们的基类型更特别。

多态性

多态性这个词由一个表示“多”的词和一个表示“态”的词构成。讲到对象时，多态性意味着一个方法或类型可以具有多种形式的实现。假定有一个媒体播放机，它既能播放音乐CD，也能播放包含MP3歌曲的DVD。然而，Play()方法的具体实现会随着媒体类型的变化而变化。在一个音乐CD对象上调用Play()方法，或者在一张音乐DVD上调用Play()方法，都能播放出音乐，因为每种类型都理解自己具体如何“播放”。媒体播放器唯一知道的就是公共基类型OpticalStorageMedia以及它定义了Play()方法的事实。在多态性的帮助下，不同的类型能自己照料一个方法的实现细节，因为共享同一个公共基类型（或接口）的多个派生类型也包含了相同的方法签名。

5.1 类的定义和实例化

定义类首先要指定关键字class，后跟一个标识符，如代码清单5-1所示。

代码清单5-1 定义类

```
class Employee
{
}
```

该类的所有代码放到类声明之后的大括号中。虽然并非必须，但一般应该将每个类都放到它自己的文件中，用类名对文件进行命名。这样可以更容易地寻找定义了一个特定类的代码。

规范

不要在一个源代码文件中放置多个类。

要用所含公共类型的名称来命名源代码文件。

定义好新类后，就可以像使用.NET Framework内置的类那样使用它了。换言之，可以声明那个类型的变量，或者定义方法来接收新类型的参数。代码清单5-2对此进行了演示。

代码清单5-2 声明类类型的变量

```
class Program
{
  static void Main()
  {
      Employee employee1, employee2;
      // ...
  }

  static void IncreaseSalary(Employee employee)
  {
      // ...
  }
}
```

初学者主题：对象和类

在非正式场合，**类**和**对象**这两个词经常互换着使用。然而，对象和类具有截然不同的含义。**类**是模板，定义了对象在实例化的时候看起来像什么样子。所以，**对象**是类的实例。类就像模具，定义了零件的样子。对象就是用这个模具创建的零件。从类创建对象的过程称为**实例化**（instantiation），因为对象是类的实例（instance）。

现在已定义了一个新的类类型，接着可以实例化该类型的对象。效仿它的前任语言，C#使用new关键字实例化对象（参见代码清单5-3）。

代码清单5-3 实例化一个类

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      Employee employee2;
      employee2 = new Employee();
```

```
            IncreaseSalary(employee1);
        }
    }
```

不用吃惊，声明和赋值既可以在同一行上完成，也可以分行完成。

和之前使用的基元数据类型（如int）不同，不能用字面值指定一个Employee。相反，要用new操作符指示“运行时”为Employee对象分配内存、实例化对象，并返回对实例的引用。

虽然现在有专门的new操作符分配内存，但没有对应的操作符回收内存。相反，“运行时”会在对象变得不可访问之后的某个时间自动回收内存。具体是由**垃圾回收器**回收。它判断哪些对象不再由其他活动对象引用，然后安排一个时间回收对象占用的内存。这样一来，就不需要在编译时判断在程序的什么位置回收并归还内存。

在这个简单的例子中，没有数据或方法与Employee关联，所以相当于实例化了一个完全没用的对象。所以，下一节把重点放在如何为对象添加数据上。

初学者主题：封装（第一部分）：对象将数据和方法组合到一起

假定接收到一叠写有员工名字的索引卡片、一叠写有员工姓氏的索引卡片以及一叠写有他们工资的索引卡片，那么除非知道每一叠卡片都按相同顺序排列，否则这些索引卡片没有什么作用。即使符合这个条件，也很难使用上面的数据，因为要判断一个人的全名，需要搜索两叠卡片。更糟的是，如果丢掉其中的一叠卡片，就没有办法再将名字与姓氏和工资关联起来。在这个时候，需要的是一叠员工卡片，其中所有数据都组合到一张卡片中。换言之，要将名字、姓氏和工资封装到一起。

日常生活中的**封装**是将一系列物品装入封套。类似地，面向对象编程将方法和数据装入对象。这提供了所有类**成员**（类的数据和方法）的一个分组，使它们不再需要单独处理。不需要将名字、姓氏和工资作为3个单独的参数传给方法。相反，可以在调用时传递对一个员工对象的引用。一旦被调用的方法接收到对象引用，就可以在对象上发送一条消息（例如可以调用像AdjustSalary()这样的方法），从而执行一个特定的操作。

语言对比：C++——delete操作符

程序员应将new的作用理解成实例化对象而不是分配内存。在堆和栈上分配对象都支持new操作符，这进一步强调了new不是关于内存分配的，也不是关于是否有必要进行回收的。

所以，C#不需要C++中的delete操作符。内存分配和回收是“运行时”的细节。这使开发人员可以将注意力更多地放在业务逻辑上。然而，虽然“运行时”会管理内存，但它不会管理其他资源，比如数据库连接、网络端口等。和C++不同，C#不支持**隐式确定性资源清理**（在编译时确定的位置进行隐式对象析构）。幸好，C#通过using语句支持**显式确定性资源清理**，通过终结器支持**隐式非确定性资源清理**。

5.2 实例字段

面向对象设计的一个核心部分是对数据进行分组，以提供一个特定的结构。本节将讨论如何在Employee类中添加数据。在面向对象术语中，在类中存储数据的变量称为**成员变量**。这个术语在C#中很好理解，但更标准的、更符合规范的是**字段**，它是与包容类型关联的具名存储单元。**实例字段**是在类的级别上声明的变量，用于存储与对象关联的数据。因此，**关联**（association）是字段类型和包容类型之间的关联。

5.2.1 实例字段的声明

代码清单5-4对Employee进行了修改，在其中包含了3个字段：FirstName、LastName和Salary。

代码清单5-4 声明字段

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;
}
```

添加好这些字段之后，就可以随同每个Employee实例存储一些基本的数据。本例添加访问修饰符public作为字段前缀。为字段添加public前缀，意味着可以从Employee之外的其他类访问该字段中的数据（参见本章稍后的5.5节“访问修饰符”）。

和局部变量声明一样，字段声明包含字段所引用的数据类型。此外，还可以在声明的同时赋给字段一个初始值，如代码清单5-5的Salary字段所示。

代码清单5-5 在声明时设置字段的初始值

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary = "Not enough";
}
```

字段命名和编码规范会在稍后在介绍了C#属性之后再给出。现在只需知道代码清单5-5不符合规范。

5.2.2 实例字段的访问

可以设置和获取字段中的数据。注意，字段不包含static修饰符，这意味着它是实例字段。只能从其包容类的实例（对象）中访问实例字段，无法直接从类中访问（换言之，不创建实例就不能访问）。

代码清单5-6展示了Program类更新后的样子，并展示了它利用Employee类的情况。输出5-1展示了结果。

代码清单5-6 访问字段

```
class Program
{
```

```
    static void Main()
    {
        Employee employee1 = new Employee();
        Employee employee2;
        employee2 = new Employee();

        employee1.FirstName = "Inigo";
        employee1.LastName = "Montoya";
        employee1.Salary = "Too Little";
        IncreaseSalary(employee1);
        Console.WriteLine(
            "{0} {1}: {2}",
            employee1.FirstName,
            employee1.LastName,
            employee1.Salary);
        // ...
    }

    static void IncreaseSalary(Employee employee)
    {
        employee.Salary = "Enough to survive on";
    }
}
```

输出5-1

```
Inigo Montoya: Enough to survive on
```

代码清单5-6实例化两个Employee对象，这和之前的例子一样。接着设置每个字段，调用IncreaseSalary()来更改工资，然后显示与employee1引用的对象关联的每个字段。

注意，首先必须指定要操作哪个Employee实例。所以，在对字段进行赋值和访问（取值）的时候，都要添加employee1变量作为字段名的前缀。

5.3 实例方法

除了在WriteLine()方法中对姓名进行格式化，再在Main()中调用WriteLine()方法，另一种可选择的办法是在Employee类中提供方法专门进行格式化。将功能修改成由Employee提供，而不是作为Program的成员，这符合类的封装原则。为什么不把与员工姓名相关的方法放到包含姓名数据的类中呢？

代码清单5-7演示了如何创建这样的一个方法。

代码清单5-7 从包容类内部访问字段

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;

  public string GetName()
  {
      return FirstName + " " + LastName;
  }
}
```

与第4章的方法相比，这里的GetName()方法没有太多特别之处，只是方法现在访问对象上的字段，而非访问局部变量。此外，方法声明没有用static来标记。本章稍后会讲到，静态方法不能直接访问类的实例字段，必须获取类的实例才能调用实例成员——无论该实例成员是方法还是字段。

在添加了GetName()方法之后，就可以更新Program.Main()来使用这个新方法，如代码清单5-8和输出5-2所示。

代码清单5-8 从包容类的外部访问字段

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      Employee employee2;
      employee2 = new Employee();

      employee1.FirstName = "Inigo";
      employee1.LastName = "Montoya";
      employee1.Salary = "Too Little";
      IncreaseSalary(employee1);
      Console.WriteLine(
          "{0}: {1}",
          employee1.GetName(),
          employee1.Salary);
      // ...
  }
  // ...
}
```

输出5-2

```
Inigo Montoya: Enough to survive on
```

5.4 使用this关键字

在类的实例成员内部，可以获取对这个类的引用。在C#中，为了显式指出当前访问的字段或方法是包容类的实例成员，可以使用关键字this。调用任何实例成员时this都是隐式的，它返回对象本身的实例。

例如，我们来看一下代码清单5-9中的SetName()方法。

代码清单5-9 使用this来显式地标识字段的所有者

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;

  public string GetName()
  {
      return FirstName + " " + LastName;
```

```
    }

    public void SetName(
        string newFirstName, string newLastName)
    {
        this.FirstName = newFirstName;
        this.LastName = newLastName;
    }
}
```

这个例子使用关键字this指出字段FirstName和LastName是类的实例成员。

虽然可为所有本地类成员引用添加this前缀，但规范的原则是，如果不会带来更多的价值就不要在代码中“添乱”。所以，只在必要时才使用this关键字。代码清单5-12是必须使用this的例子。代码清单5-9和代码清单5-10则不是。在代码清单5-9中，舍弃this不会改变代码的含义。而代码清单5-10可修改字段命名规范来避免局部变量和字段之间的歧义。

初学者主题：依靠编码风格来避免歧义

在SetName()方法中没必要使用this关键字，因为FirstName显然有别于newFirstName。然而，现在假定参数不叫做newFirstName，而叫做FirstName（使用Pascal大小写风格），如代码清单5-10所示。

代码清单5-10 使用this来避免歧义

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;

  public string GetName()
  {
      return FirstName + " " + LastName;
  }

  // Caution:  Parameter names use Pascal case
  public void SetName(string FirstName, string LastName)
  {
      this.FirstName = FirstName;
      this.LastName = LastName;
  }
}
```

在这个例子中，要引用FirstName字段就必须显式地指明对象所在的Employee对象。this就像在Program.Main()方法中使用的employee1变量前缀（参见代码清单5-8），标识了在其上调用SetName()方法的引用。

代码清单5-10不符合C#命名规范，即参数要像局部变量那样使用camel大小写风格来命名。这可能造成很难发现的bug，因为将FirstName（本来想引用字段）赋值给FirstName（参数），代码仍然能编译并运行。为了避免这个问题，最好是为参数和局部变量采用与字段不同的命名规范。本章稍后会演示这样的一个规范。

语言对比：Visual Basic——使用Me访问类的实例

C#关键字this完全等价于Visual Basic关键字Me。

在代码清单5-9和代码清单5-10中，GetName()方法没有使用this关键字，它是可选的。然而，假如存在与字段同名的局部变量或参数（参见代码清单5-10的SetName()方法），省略this将访问局部变量或参数，而不是字段。所以，在这种情况下，this是必须的。

还可使用this关键字显式访问类的方法。例如，可在SetName()方法内使用this.GetName()输出新赋值的姓名（参见代码清单5-11和输出5-3）。

代码清单5-11 this作为方法名前缀

```
class Employee
{
  // ...

  public string GetName()
  {
      return FirstName + " " + LastName;
  }

  public void SetName(string newFirstName, string newLastName)
  {
      this.FirstName = newFirstName;
      this.LastName = newLastName;
      Console.WriteLine("Name changed to '{0}'",
          this.GetName());
  }
}
```

```
class Program
{
  static void Main()
  {
      Employee employee = new Employee();

      employee.SetName("Inigo", "Montoya");
      // ...
  }
  // ...
}
```

输出5-3

```
Name changed to 'Inigo Montoya'
```

有时需要使用this传递对当前正在执行的对象的引用。如代码清单5-12中的Save()方法所示。

代码清单5-12 在方法调用中传递this

```
class Employee
{
```

```
    public string FirstName;
    public string LastName;
    public string Salary;

    public void Save()
    {
        DataStorage.Store(this);
    }
}

class DataStorage
{
    // Save an employee object to a file
    // named with the Employee name.
    public static void Store(Employee employee)
    {
        // ...
    }
}
```

Save()方法调用DataStorage类的Store()方法。然而，需要向Store()方法传递准备进行持久化存储的Employee对象。这是使用关键字this来完成的，它传递了正在其上调用Save()方法的那个Employee对象实例。

高级主题：存储和载入文件

在DataStorage内部，Store()方法的实现要涉及System.IO命名空间中的类，如代码清单5-13所示。在Store()内部，首先要实例化一个FileStream对象，将它与一个对应于员工全名的文件关联起来。FileMode.Create参数指明，假如<firstname> <lastname>.dat文件不存在，就新建一个；假如文件存在，就覆盖它。接着要创建一个StreamWriter类。StreamWriter类负责将文本写入FileStream。数据是用WriteLine()方法写入的，就像向控制台写入一样。

代码清单5-13 将数据持久化存储到文件

```
using System;
// IO namespace
using System.IO;

class DataStorage
{
    // Save an employee object to a file
    // named with the Employee name.
    // Error handling not shown.
    public static void Store(Employee employee)
    {
        // Instantiate a FileStream using FirstNameLastName.dat
        // for the filename. FileMode.Create will force
        // a new file to be created or override an
        // existing file.
        FileStream stream = new FileStream(
            employee.FirstName + employee.LastName + ".dat",
            FileMode.Create);①

        // Create a StreamWriter object for writing text
```

① 可用using语句改进这行代码，但暂时没有使用这个构造，因为到现在为止还没有讲到。

```
        // into the FileStream
        StreamWriter writer = new StreamWriter(stream);

        // Write all the data associated with the employee.
        writer.WriteLine(employee.FirstName);
        writer.WriteLine(employee.LastName);
        writer.WriteLine(employee.Salary);

        // Close the StreamWriter and its Stream.
        writer.Close();  // Automatically closes the stream
    }
    // ...
}
```

写入操作完成后，FileStream和StreamWriter都需要关闭，避免它们在等待垃圾回收器运行期间，处于“不确定性打开”的状态。上述代码不包含任何错误处理机制，所以假如引发一个异常，两个Close()方法都不会得到调用。

载入过程与存储过程相似，如代码清单5-14所示。

代码清单5-14 从文件中获取数据

```
class Employee
{
  // ...
}
```

```
// IO namespace
using System;
using System.IO;

class DataStorage
{
  // ...

  public static Employee Load(string firstName, string lastName)
  {
      Employee employee = new Employee();

      // Instantiate a FileStream using FirstNameLastName.dat
      // for the filename. FileMode.Open will open
      // an existing file or else report an error.
      FileStream stream = new FileStream(
          firstName + lastName + ".dat", FileMode.Open);①

      // Create a SteamReader for reading text from the file.
      StreamReader reader = new StreamReader(stream);

      // Read each line from the file and place it into
      // the associated property.
      employee.FirstName = reader.ReadLine();
      employee.LastName = reader.ReadLine();
      employee.Salary = reader.ReadLine();

      // Close the StreamReader and its Stream.
      reader.Close();  // Automatically closes the stream
```

① 可用using语句改进这行代码，但暂时没有使用这个构造，因为到现在为止还没有讲到。

```
        return employee;
    }
}
```

```
class Program
{
    static void Main()
    {
        Employee employee1;

        Employee employee2 = new Employee();
        employee2.SetName("Inigo", "Montoya");
        employee2.Save();

        // Modify employee2 after saving.
        IncreaseSalary(employee2);

        // Load employee1 from the saved version of employee2
        employee1 = DataStorage.Load("Inigo", "Montoya");

        Console.WriteLine(
            "{0}: {1}",
            employee1.GetName(),
            employee1.Salary);

        // ...
    }
    // ...
}
```

输出5-4展示了结果。

输出5-4

```
Name changed to 'Inigo Montoya'
Inigo Montoya:
```

代码清单5-14展示的是和存储相反的过程，它使用StreamReader而不是StreamWriter。同样地，一旦数据读取完毕，就要在FileStream和StreamReader上调用Close()方法。

输出5-4没有在Inigo Montoya:之后显示任何工资信息，因为只有在调用Save()之后，才会通过调用IncreaseSalary()将Salary设为Enough to survive on。

注意，在Main()中，可以从员工实例调用Save()，但为了载入新员工，调用的是DataStorage.Load()。需要载入员工时，一般还没有可在其中载入（数据）的员工实例，所以Employee类的实例方法不可取。除了在DataStorage类本身上调用Load，另一个办法是为Employee类添加静态Load()方法（详情参见5.8节），这样就可以调用Employee.Load()——注意，是直接在Employee类上调用，而不是在它的实例上调用。

注意，代码清单顶部包含using System.IO指令。这样可以直接访问每个IO类，无需为其附加完整的命名空间前缀。

5.5 访问修饰符

本章早些时候声明字段时，曾为字段声明添加了关键字public作为前缀。public是访

问修饰符（access modifier），它标识了所修饰成员的封装级别。可以选择5个访问修饰符，即public、private、protected、internal和protected internal。本节介绍前两个。

初学者主题：封装（第二部分）：信息隐藏

除了组合数据和方法，封装的另一个重要作用是隐藏对象的数据和行为的内部细节。方法在某种程度上也能做到这一点：在方法外部，调用者看到的只有方法声明，内部实现不可见。但面向对象编程更进一步，它能控制类成员在类外部的可视程度。在类的外部不可见的成员称为**私有成员**。

在面向对象编程中，封装的作用不仅仅是组合数据和行为，它还能隐藏类中的数据和行为的实现细节，使类的内部工作机制不被暴露。这减少了调用者对数据进行不恰当修改的几率。

访问修饰符的作用是提供封装。public显式指明可从Employee类的外部访问被它修饰的字段。例如，可以从Program类中访问那些字段。

然而，假定Employee类要包含一个Password字段，那么应该如何设计？在这种情况下，应该允许调用一个Employee对象，并用Logon()方法验证密码。但是，不应该允许从类的外部访问Employee对象的Password字段。

为了隐藏Password字段，禁止从它的包容类的外部访问，应该使用private访问修饰符代替public，如代码清单5-15所示。这样就无法从Program类中访问Password字段了。

代码清单5-15 使用private访问修饰符

```
class Employee
{
  public string FirstName;
  public string LastName;
  public string Salary;
  private string Password;
  private bool IsAuthenticated;

  public bool Logon(string password)
  {
      if(Password == password)
      {
          IsAuthenticated = true;
      }
      return IsAuthenticated;
  }

  public bool GetIsAuthenticated()
  {
      return IsAuthenticated;
  }
  // ...
}
```

```
class Program
{
  static void Main()
  {
      Employee employee = new Employee();
```

```
        employee.FirstName = "Inigo";
        employee.LastName = "Montoya";

        // ...

        // Password is private, so it cannot be
        // accessed from outside the class.
        // Console.WriteLine(
        //     ("Password = {0}", employee.Password);
    }
    // ...
}
```

虽然代码清单5-15没有演示使用private来修饰方法，但实际上可以这样做。

注意，如果不为类成员添加访问修饰符，那么默认使用的是private。也就是说，成员默认为私有成员。公共成员必须显式指定。

5.6 属性

5.5节演示了如何使用private关键字封装密码，从而禁止从类的外部访问。然而，这种形式的封装通常过于彻底。例如，可能希望字段在外部只读，但在内部可以更改。又例如，可能希望允许对类中的一些数据执行写操作，但需要验证对数据的更改。再例如，可能希望动态地构造数据。以前为了满足这些要求，采取的办法是将字段标记为私有，然后提供取值和赋值方法来访问和修改数据。代码清单5-16将FirstName和LastName更改为私有字段。每个字段的公共取值和赋值方法用于访问和更改它们的值。

代码清单5-16 声明取值和赋值方法

```
class Employee
{

  private string FirstName;
  // FirstName getter
  public string GetFirstName()
  {
      return FirstName;
  }
  // FirstName setter
  public void SetFirstName(string newFirstName)
  {
      if(newFirstName != null && newFirstName != "")
      {
          FirstName = newFirstName;
      }
  }

  private string LastName;
  // LastName getter
  public string GetLastName()
  {
      return LastName;
  }
  // LastName setter
  public void SetLastName(string newLastName)
  {
```

```
            if(newLastName != null && newLastName != "")
            {
                LastName = newLastName;
            }
        }
        // ...
    }
```

遗憾的是，这一更改会影响Employee类的可编程性。无法再用赋值操作符来设置类中的数据。另外，只能调用方法来访问数据。

5.6.1 属性的声明

考虑到经常会用到这种编程模式，C#的设计者决定为它提供显式的语法支持。这种新语法称为**属性**（property），如代码清单5-17和输出5-5所示。

代码清单5-17 定义属性

```
class Program
{
  static void Main()
  {
      Employee employee = new Employee();

      // Call the FirstName property's setter.
      employee.FirstName = "Inigo";

      // Call the FirstName property's getter.
      System.Console.WriteLine(employee.FirstName);
  }
}
```

```
class Employee
{
  // FirstName property
  public string FirstName
  {
      get
      {
          return _FirstName;
      }
      set
      {
          _FirstName = value;
      }
  }
  private string _FirstName;

  // LastName property
  public string LastName
  {
      get
      {
          return _LastName;
      }
      set
      {
          _LastName = value;
      }
  }
  private string _LastName;
```

```
    // ...
}
```

输出5-5

```
Inigo
```

在代码清单5-17中，首先注意到的不是属性代码本身，而是Program类中的代码。现在其实已经没有FirstName和LastName字段了。但这一点从Program类本身看不出来。访问员工名字和姓氏所用的代码根本没有改变。仍然可以使用简单的赋值操作符对姓或名进行赋值，例如employee.FirstName = "Inigo"。

属性的关键特点在于，它提供了从编程角度看类似于字段的API。但是，事实上并不存在这样的字段。属性声明看起来和字段声明一样，但跟随在属性名之后的是一对大括号，要在其中添加属性的实现。属性的实现由两个可选的部分构成。其中，get标志属性的取值方法，它直接对应代码清单5-16定义的GetFirstName()和GetLastName()方法。它实现了字段的取值语法，例如为了访问FirstName属性，要调用employee.FirstName。类似地，set标志属性的赋值方法，它实现了字段的赋值语法：

```
employee.FirstName = "Inigo";
```

属性的定义使用了3个上下文关键字。其中，get和set关键字分别标识属性的取值和赋值部分（称为属性的取值方法和赋值方法）。此外，赋值方法可以使用value关键字引用赋值操作的右侧部分。所以，当Program.Main()调用employee.FirstName = "Inigo"时，赋值方法中的value被设为"Inigo"，该值可以赋给_FirstName字段。代码清单5-17的属性实现是最常见的。调用取值方法时，比如Console.WriteLine(employee2.FirstName)，会获取字段（_FirstName）的值，并将其写入控制台。

5.6.2 自动实现的属性

从C# 3.0开始，属性语法有了简化版本。属性加支持字段（比如上例的_FirstName），并用取值方法和赋值方法来获取和设置这个字段，由于这是十分常见的设计，而且代码比较琐碎（参考FirstName和LastName的实现就知道了），所以C#3.0编译器（及更高版本编译器）现在允许在声明属性时，不添加取值或赋值方法，也不声明任何支持字段。一切都将自动实现。代码清单5-18展示了这个语法，输出5-6展示了结果。

代码清单5-18 自动实现的属性

```
class Program
{
  static void Main()
  {
        Employee employee1 =
            new Employee();
        Employee employee2 =
            new Employee();

        // Call the FirstName property's setter.
        employee1.FirstName = "Inigo";

        // Call the FirstName property's getter.
        System.Console.WriteLine(employee1.FirstName);
```

```
            // Assign an auto-implemented property
            employee2.Title = "Computer Nerd";
            employee1.Manager = employee2;

            // Print employee1's manager's title.
            System.Console.WriteLine(employee1.Manager.Title);
        }
    }

    class Employee
    {
        // FirstName property
        public string FirstName
        {
            get
            {
                return _FirstName;
            }
            set
            {
                _FirstName = value;
            }
        }
        private string _FirstName;

        // LastName property
        public string LastName
        {
            get
            {
                return _LastName;
            }
            set
            {
                _LastName = value;
            }
        }
        private string _LastName;
        // ...

        // Title property
        public string Title { get; set; }

        // Manager property
        public Employee Manager { get; set; }

    }
```

输出5-6

```
Inigo
Computer Nerd
```

自动实现的属性使我们能更容易地进行属性编码，也使代码更易读。除此之外，假如未来需要添加一些额外的代码，比如要在赋值方法中进行验证，那么虽然要修改现在的属性声明，但调用它们的代码不必进行任何更改。

本书剩余部分会频繁使用C#3.0及更高版本的新的语法，并且不会专门指出它们是C#3.0引入的特性。

5.6.3 属性和字段的设计编码规范

由于可以写显式的赋值和取值方法而不是属性，所以有时会产生是应该使用属性还是使用方法的疑问。一般的编码规范是，方法应该用于代表行动，而属性用于代表数据。属性旨在简化对简单数据的访问。在编码中期望做到的是，调用属性的代价不应比访问字段高出太多。

至于命名，注意在代码清单5-18中属性名是FirstName，字段名变成了_FirstName。其实就是添加了下划线前缀的Pascal大小写风格。对于为属性提供支持的私有字段，其他常见的命名规范还有_firstName和m_FirstName（延续自C++的命名规范，其中m代表member variable，即成员变量）。另外，还可以采用camel大小写风格，就像局部变量那样[①]。不过，应尽量避免camel大小写风格，因为局部变量和参数也经常采用这种大小写风格，造成名称重复的的可能性很高。另外，为了符合封装原则，字段不应被声明为public或protected。

规范

要使用属性简化对对简单数据（进行少量计算）的访问。

避免从属性的取值方法中引发异常。

要在属性引发异常时保留原始属性值。

不管为私有字段使用哪一种命名方案，都要依据Pascal大小写风格来命名属性。因此，属性应使用LastName和FirstName这类表示名词、名词短语或形容词形式的名称。事实上，属性名和类型名相同的情况也不罕见。例如，Person对象中的Address类型的Address属性。

规范

考虑为支持字段和属性使用相同的大小写风格，为支持字段附加“_”前缀。但不要使用双下划线，因为以双下划线开头的标识符是为C#编译器保留的。

要使用名词、名词短语或形容词来命名属性。

考虑让属性和它的类型同名。

避免用camel大小写风格命名字段。

如果有用的话，**要**为布尔属性附加“Is”、“Can”或“Has”前缀。

不要声明public或protected的实例字段（而是通过属性来公开字段）。

要用Pascal大小写风格命名属性。

要优先使用自动实现的属性而不是字段。

如果没有额外的实现逻辑，**要**优先使用自动实现的属性，而不是自己编写完整版本。

5.6.4 提供属性验证

在代码清单5-19中，注意Employee的Initialize()方法使用属性而不是字段进行赋值。虽然并非必须如此，但这样做的结果是，无论在类的内部还是外部，属性的赋值方法

① 我个人更喜欢_FirstName，下划线就足够了，名称前的m太多余。另外，使用与属性名称相同的大小写风格，Visual Studio 代码模块扩展工具中就可以只设置一个字符串，而不必为属性名和字段名各设一个。

中的任何验证都会得到调用。例如，假定更改LastName属性，在把value赋给_LastName之前检查它是否为null或空字符串，那么会发生什么？

代码清单5-19 提供属性验证

```
class Employee
{
  // ...
  public void Initialize(
      string newFirstName, string newLastName)
  {
      // Use property inside the Employee
      // class as well.
      FirstName = newFirstName;
      LastName = newLastName;
  }


  // LastName property
  public string LastName
  {
      get
      {
          return _LastName;
      }
      set
      {
          // Validate LastName assignment
          if(value == null)
          {
              // Report error
              throw new ArgumentNullException("value");
          }
          else
          {
              // Remove any whitespace around
              // the new last name.
              value = value.Trim();
              if(value == "")
              {
                  throw new ArgumentException(
                      "LastName cannot be blank.", "value");①
              }
              else
                  _LastName = value;
          }
      }
      private string _LastName;
      // ...
  }
```

在新实现中，如果为LastName赋了无效的值（要么从同一个类的另一个成员赋值，要么在Program.Main()内直接向LastName赋值），代码就会引发异常。拦截赋值，并通过字段风格的API对参数进行验证，这是属性的优点之一。

① 向Teller、Cher、Sting、Mandonna、Bono、Prince和Liberace说声抱歉。（这些都是《公主新娘》中的人物，都没有姓。——译者注）

作为良好的编程习惯，应该只在属性实现的内部访问属性的支持字段。换言之，要一直使用属性，不要直接调用字段。许多时候，即使在属性所在的类中，也不应该从属性实现的外部访问它的支持字段。这样一来，只要为属性添加了代码（如验证代码），整个类就可以马上利用这些代码。本章后面会讲到，在字段被标记为只读的时候会产生例外。因为在这种情况下，一旦类完成了实例化，即便在属性的赋值方法中也不能设置值。

虽然很少见，但确实可以在赋值方法中赋值，就像代码清单5-19那样。在这个例子中，调用value.Trim()会移除新姓氏值左右的任何空白字符。

规范

避免从属性外部（即使是在包容属性的类中）访问属性的支持字段。

调用ArgumentException()或ArgumentNullException()构造器时，要为paramName参数传递“value”（“value”是属性赋值方法隐含的参数名）。

5.6.5 只读和只写属性

通过移除某个属性的取值方法或者赋值方法部分，可以改变属性的可访问性。只有赋值方法的属性是只写属性，这是十分罕见的情形。类似地，只提供取值方法会得到只读属性；任何赋值企图都会造成编译错误。例如，为了使Id为只读，可以像代码清单5-20那样编码。

代码清单5-20 定义只读属性

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      employee1.Initialize(42);

      // ERROR:  Property or indexer 'Employee.Id'
      // cannot be assigned to -- it is read-only
      employee1.Id = "490";
  }
}

class Employee
{
  public void Initialize(int id)
  {
      // Use field because Id property has no setter,
      // it is read-only.
      _Id = id.ToString();
  }

  // ...
  // Id property declaration
  public string Id
  {
      get
      {
```

```
            return _Id;
        }
        // No setter provided.
    }
    private string _Id;

}
```

代码清单5-20从Employee构造函数（而不是属性）中对字段进行赋值（_Id = id）。通过属性来赋值会造成编译错误，就像Program.Main()中的employee1.Id = "490";那一行代码那样。

规范

如果不想调用者更改属性的值，**要**创建只读属性。

5.6.6 为取值方法和赋值方法指定访问修饰符

如前所述，字段一般不要从它们的属性外部访问。否则，为属性添加的验证逻辑或其他逻辑可能失去意义。遗憾的是，C# 1.0要求属性取值和赋值方法的封装级别必须一致。换言之，不能在属性中同时创建公共取值方法和私有赋值方法，使外部类只能对属性进行只读访问，而允许类内的代码向属性写入。

在C# 2.0中，情况发生了变化。在属性的实现中，可为get或set部分指定访问修饰符（但不能为两者都指定），从而覆盖为属性声明指定的访问修饰符。代码清单5-21展示了一个例子。

代码清单5-21 为赋值方法指定访问修饰符

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      employee1.Initialize(42);
      // ERROR: The property or indexer 'Employee.Id'
      // cannot be used in this context because the set
      // accessor is inaccessible
      employee1.Id = "490";
  }
}
```

```
class Employee
{
  public void Initialize(int id)
  {
      // Set Id property
      Id = id.ToString();
  }

  // ...
  // Id property declaration
  public string Id
  {
```

```
        get
        {
            return _Id;
        }
        // Providing an access modifier is in C# 2.0
        // and higher only
        private set
        {
            _Id = value;
        }
    }
    private string _Id;
}
```

为赋值方法指定private修饰符后，属性对于除Employee的其他类来说就是只读的。在Employee类内部，属性是可读/可写的，所以可在构造器中对属性进行赋值。为取值方法或赋值方法指定访问修饰符时，注意该访问修饰符的“限制性”必须比应用于整个属性的访问修饰符更“严格”。例如，将属性声明为较严格的private，但将它的赋值方法声明为较宽松的public，就会发生编译错误。

规范

要为所有属性的取值方法和赋值方法的实现应用适当的可访问性修饰符。

不要提供只写属性，也**不要**让属性的赋值方法的可访问性比取值方法更宽松。

5.6.7 属性作为虚字段使用

可以看出，属性的行为与虚字段相似。有时甚至根本不需要支持字段；相反，可以让属性的取值方法返回计算好的值，而让赋值方法解析值，并把它持久存储到其他成员字段中。在代码清单5-22中，请注意Name属性的实现。输出5-7展示了结果。

代码清单5-22 定义属性

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();

      employee1.Name = "Inigo Montoya";
      System.Console.WriteLine(employee1.Name);

      // ...
  }
}
```

```
class Employee
{
  // ...

  // FirstName property
  public string FirstName
  {
```

```
        get
        {
            return _FirstName;
        }
        set
        {
            _FirstName = value;
        }
    }
    private string _FirstName;

    // LastName property
    public string LastName
    {
        get
        {
            return _LastName;
        }
        set
        {
            _LastName = value;
        }
    }
    private string _LastName;
    // ...

    // Name property
    public string Name
    {
        get
        {
            return FirstName + " " + LastName;
        }
        set
        {
            // Split the assigned value into
            // first and last names.
            string[] names;
            names = value.Split(new char[]{' '});
            if(names.Length == 2)
            {
                FirstName = names[0];
                LastName = names[1];
            }
            else
            {
                // Throw an exception if the full
                // name was not assigned.
                throw new System. ArgumentException (
                    string.Format(
                    "Assigned value '{0}' is invalid", value));
            }
        }
    }

    // ...
}
```

输出5-7

```
Inigo Montoya
```

Name属性的取值方法将FirstName和LastName属性返回的值连接到一起。事实上，所赋的姓名并没有真正存储下来。为Name属性赋值时，右侧的值会解析成名字和姓氏部分。

5.6.8 属性和方法调用不允许作为ref或out参数值使用

C#允许属性像字段那样使用，只是不允许将它们作为ref或out参数值来传递。ref和out参数值在内部实现时，需要将内存地址传给目标方法。但是，由于属性可能是没有支持字段的虚字段，也有可能是只读或只写的，因此不可能传递存储地址。所以，不能将属性作为ref或out参数值传递。同样的道理也适用于方法调用。如果需要将属性或方法调用作为ref或out参数值传递，首先必须将值复制到一个变量中，然后传递该变量。方法调用结束之后，再将变量的值赋回属性。

高级主题：属性的内部工作机制

代码清单5-23证明，取值方法和赋值方法在CIL代码中是以get_FirstName()和set_FirstName()的形式出现的。

代码清单5-23 与属性对应的CIL代码

```
// ...

.field private string _FirstName
.method public hidebysig specialname instance string
        get_FirstName() cil managed
{
  // Code size       12 (0xc)
  .maxstack  1
  .locals init (string V_0)
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldfld      string Employee::_FirstName
  IL_0007:  stloc.0
  IL_0008:  br.s       IL_000a

  IL_000a:  ldloc.0
  IL_000b:  ret
} // end of method Employee::get_FirstName

.method public hidebysig specialname instance void
        set_FirstName(string 'value') cil managed
{
  // Code size       9 (0x9)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldarg.1
  IL_0003:  stfld      string Employee::_FirstName
  IL_0008:  ret
} // end of method Employee::set_FirstName

.property instance string FirstName()
{
  .get instance string Employee::get_FirstName()
  .set instance void Employee::set_FirstName(string)
} // end of property Employee::FirstName

// ...
```

除了外观与普通方法无异，还要注意，属性在CIL中也属于一种显式的构造[①]。如代码清单5-24所示，取值方法和赋值方法由CIL属性调用，而CIL属性是IL代码中的一种显式构造。因此，语言和编译器并不限于总是依据一个命名惯例对属性进行解释。相反，正是由于CIL属性的这个独特的设计，所以编译器和代码编辑器才能提供特殊的语法。

代码清单5-24 属性是CIL的显式构造

```
.property instance string FirstName()
{
  .get instance string Program::get_FirstName()
  .set instance void Program::set_FirstName(string)
} // end of property Program::FirstName
```

注意在代码清单5-23中，作为属性一部分的取值方法和赋值方法包含了specialname元数据。IDE（比如Visual Studio）根据这个修饰符向“智能感知”（IntelliSense）隐藏成员。

自动实现的属性在CIL中看起来和显式定义支持字段的属性几乎完全一样。C#编译器在IL中生成名为<PropertyName>k_BackingField的字段。该字段包含了名为System.Runtime.CompilerServices.CompilerGeneratedAttribute的特性（参见第17章）。无论取值方法还是赋值方法，都用同一个特性进行修饰——实现与代码清单5-23以及代码清单5-24相同。

5.7 构造器

现在已经在类中添加了用于存储数据的字段。接着，应该考虑数据的有效性。如代码清单5-6所示，可以使用new操作符实例化对象。但是，这样可能创建包含无效数据的员工对象。刚刚实例化employee1之后，得到的是姓名和工资尚未初始化的Employee对象。在那个特定的代码清单中，是在实例化员工之后，立即对尚未初始化的字段进行赋值。但假如忘记进行初始化，编译器也不会发出警告。最终结果是得到含有无效姓名的Employee对象。

5.7.1 构造器的声明

为了纠正这个问题，需要提供一种方式在创建对象时指定必需的数据。这是用构造器来实现的，如代码清单5-25所示。

代码清单5-25 定义构造器

```
class Employee
{
  // Employee constructor
  public Employee(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }

  public string FirstName{ get; set; }
  public string LastName{ get; set; }
```

① 所谓“显式的构造”，就是明确存在的一个构造，——译者注

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

    // ...
}
```

为了定义构造器，要创建一个没有返回类型的方法，而且方法名必须完全和类名相同。

构造器是“运行时”用来初始化对象实例的方法。在这个例子中，构造器获取员工的名字和姓氏作为参数，允许程序员在实例化Employee对象时指定这些参数的值。代码清单5-26演示了如何调用构造器。

代码清单5-26 调用构造器

```
class Program
{
  static void Main()
  {
      Employee employee;
      employee = new Employee("Inigo", "Montoya");
      employee.Salary = "Too Little";

      System.Console.WriteLine(
          "{0} {1}: {2}",
          employee.FirstName,
          employee.LastName,
          employee.Salary);
  }
  // ...
}
```

注意，new操作符返回一个引用，它指向实例化好的对象（虽然在构造器的声明或实现中没有显式指定返回类型或使用返回语句）。此外，已经移除了名字和姓氏的初始化代码，因为现在是在构造器内部进行初始化。在这个例子中，由于没有在构造器内部初始化salary，所以对工资进行赋值的代码仍然予以保留。

开发者应注意到既在声明中又在构造器内部进行赋值的情况。假如字段在声明时赋值（比如代码清单5-5中的string Salary = "Not enough"），那么只有在这个赋值发生之后，构造器内部的赋值才会发生。所以，最终生效的是构造器内部的赋值，它会覆盖声明时的赋值。粗心的人可能误以为在对象实例化之后，字段的值是在声明时赋予的。所以，有必要考虑一种编码风格，避免在同一个类中，既在声明时赋值，又在构造器中赋值。

高级主题：new操作符的实现细节

在内部，new操作符和构造器是像下面这样交互的。new操作符从内存管理器获取“空白”内存，调用指定的构造器，将对“空白”内存的引用作为隐式的this参数传给构造器。构造器链剩余的部分开始执行，在构造器之间传递引用。这些构造器都没有返回类型（行为都像是返回void）。构造器链上的执行结束后，new操作符返回内存引用。现在，该引用指向的内存处于初始化好的形式。

5.7.2 默认构造器

必须注意，一旦显式添加了构造器，在Main()中实例化Employee就必须指定名字和姓氏。代码清单5-27的代码无法编译。

代码清单5-27 没有默认构造器了

```
class Program
{
  static void Main()
  {
      Employee employee;
      // ERROR:      No overload for method 'Employee'
      // takes '0' arguments.
      employee = new Employee();

      // ...
  }
}
```

假如类没有显式定义的构造器，C#编译器会在编译时自动添加一个。该构造器不获取参数，称为**默认构造器**。一旦为类显式添加了构造器，C#编译器就不再自动提供默认构造器。因此，在定义了Employee(string firstName, string lastName)之后，编译器不再添加默认构造器Employee()。虽然可以手动添加，但会再度允许构造没有指定员工姓名的Employee对象。

没有必要依赖由编译器定义的默认构造器。程序员可以显式定义默认构造器，比如用它将某些字段初始化成特定值。声明不获取任何参数的构造器就能定义默认构造器。

5.7.3 对象初始化器

C# 3.0新增了**对象初始化器**[①]，用于初始化对象中所有可以访问的字段和属性。具体地说，调用构造器创建对象时，可以在后面的一对大括号中添加成员初始化列表。每个成员的初始化操作都是一个赋值操作，等号左边是可以访问的字段或属性，右边是要赋的值，如代码清单5-28所示。

代码清单5-28 调用对象初始化器

```
class Program
{
  static void Main()
  {
        Employee employee1 = new Employee("Inigo", "Montoya")
            { Title = "Computer Nerd", Salary = "Not enough"};
      // ...
  }
}
```

注意，使用对象初始化器时要遵守相同的构造器规则。事实上，这只是一种语法糖，最终生成CIL代码和用单独的语句对字段及属性进行赋值无异。C#中的成员初始化顺序，决定了在CIL构造器调用之后的语句中，对属性和字段进行赋值的顺序。

总之，构造器退出时，所有属性都应该初始化成合理的默认值。利用属性的赋值方法的校验逻辑，可以制止将无效的数据赋给属性。但有时一个或多个属性值可能会导致同一个对象的其他属性包含无效值。发生这种情况时，应推迟为无效状态引发异常，直到无效的、不相关的属性值变得相关。

---

① object initializer 有时也称为“对象初始化列表”。——译者注

规范

要为所有属性提供有意义的默认值，确保默认值不会造成安全漏洞或造成代码效率大幅下降。对于自动实现的属性，要通过构造器设置默认值。

要允许以任意顺序设置属性，即使这会造成对象临时处于无效状态。

高级主题：集合初始化器

采用和对象初始化器相似的语法，C# 3.0还新增了**集合初始化器**。集合初始化器支持与对象初始化器相似的功能集，只是这些功能是作用于集合的。具体地说，使用集合初始化器，可以在集合实例化期间为集合内部的项赋值。它借用数组语法来初始化集合中的每一项。例如，为了初始化Employee列表，可在构造器调用之后的一对大括号中指定每一项，如代码清单5-29所示。

代码清单5-29 调用一个集合初始化器

```
class Program
{
  static void Main()
  {
      List<Employee> employees = new List<Employee>()
         {
            new Employee("Inigo", "Montoya"),
            new Employee("Chuck", "McAtee")
         };
      // ...
  }
}
```

如果像这样为新的集合实例赋值，编译器生成的代码会按顺序实例化每个对象，并通过Add()方法把它们添加到集合中。

高级主题：终结器

构造器定义了在类的实例化过程中发生的事情。为了定义在对象销毁过程中发生的事情，C#提供了**终结器**。与C++的析构器不同，终结器不是在对一个对象的所有引用都消失之后就马上运行。相反，终结器是在对象被判定“不可达”之后的不确定时间内执行。具体地说，垃圾回收器会在一次垃圾回收过程中识别出带有终结器的对象。但不是立即回收这些对象，而是将它们添加到一个终结队列中。一个独立的线程遍历终结队列中的每一个对象，调用其终结器，然后将其从队列中删除，使其再次可供垃圾回收器处理。第9章深入讨论了这个过程以及资源清理的主题。

5.7.4 构造器的重载

构造器可以重载——可同时存在多个构造器，只要参数的数量和类型有区别。如代码清单5-30所示，可以提供一个构造器，除了获取员工姓名，还获取员工ID；再提供一个构造器只获取员工ID。

代码清单5-30 重载构造器

```
class Employee
{
  public Employee(string firstName, string lastName)
  {
      FirstName = firstName;
      LastName = lastName;
  }

  public Employee(
      int id, string firstName, string lastName )
  {
      Id = id;
      FirstName = firstName;
      LastName = lastName;
  }

  public Employee(int id)
  {
      Id = id;

      // Look up employee name...
      // ...
  }

  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Salary { get; set; }

  // ...
}
```

这样，当Program.Main()根据姓名来实例化员工对象时，既可只传递员工ID，也可同时传递姓名和ID。例如，在系统中创建新员工时，可以使用同时获取姓名和ID的构造器。而从文件或数据库加载现有员工时，则使用只获取ID的构造器。

和方法重载 一样，多个构造器可用来支持多种情形，可以使用少量参数的构造器支持简单情形，使用带有附加参数的构造器支持复杂情形。应优先使用可选参数而不是重载，以便在API中清楚地看出“默认”属性的默认值。例如，Person的一个构造器签名Person(string firstName, string lastName, int? age = null)就清楚地指明如果Person的Age未指定，就将它默认为null。

规范

如果使用构造器参数来设置属性，构造器参数（camel大小写风格）要使用和属性（Pascal大小写风格）相同的名称，区别仅仅是大小写风格。

要为构造器提供可选参数，并且/或者提供便利的重载构造器，用好的默认值初始化属性。

5.7.5 构造器链：使用this调用另一个构造器

在代码清单5-30中，注意现在对Employee对象进行初始化的代码在好几个地方都重复了，所以必须在多个地方进行维护。虽然这个例子的代码量比较小，但完全可以从一个构

造器中调用另一个构造器，以避免输入重复的代码。这称为**构造器链**，它是用**构造器初始化器**来实现的。构造器初始化器会在执行当前构造器的实现之前，判断要调用另外哪一个构造器，如代码清单5-31所示。

代码清单5-31 从一个构造器中调用另一个构造器

```
class Employee
{
  public Employee(string firstName, string lastName)
  {
      FirstName = firstName;
      LastName = lastName;
  }

  public Employee(
      int id, string firstName, string lastName )
      : this(firstName, lastName)
  {
      Id = id;
  }

  public Employee(int id)
  {
      Id = id;

      // Look up employee name...
      // ...

      // NOTE:  Member constructors cannot be
      // called explicitly inline
      // this(id, firstName, lastName);
  }

  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Salary { get; set; }

  // ...
}
```

要从一个构造器中调用同一个类的另一个构造器（针对相同对象实例），C#采取的语法格式是在一个冒号后面添加this关键字，再添加被调用构造器的参数列表。在本例中，获取3个参数的构造器调用获取两个参数的构造器。但是，通常采取的是相反的调用模式——参数最少的构造器调用具有参数最多的构造器，为未知的参数传递默认值。

初学者主题：集中初始化

注意在代码清单5-31中，不能在Employee(int id)构造器的实现中调用this(id, firstName,lastName)，因为当前上下文不存在firstName和 lastName这两个名称。要将所有初始化代码都集中到一个方法中，必须创建单独的方法，如代码清单5-32所示。

代码清单5-32 提供初始化方法

```
class Employee
{
  public Employee(string firstName, string lastName)
```

```
        {
            int id;
            // Generate an employee ID...
            // ...
            Initialize(id, firstName, lastName);
        }

        public Employee(int id, string firstName, string lastName )
        {
            Initialize(id, firstName, lastName);
        }

        public Employee(int id)
        {
            string firstName;
            string lastName;
            Id = id;

            // Look up employee data
            // ...

            Initialize(id, firstName, lastName);
        }

        private void Initialize(
            int id, string firstName, string lastName)
        {
            Id = id;
            FirstName = firstName;
            LastName = lastName;
        }
        // ...
    }
```

本例是将方法命名为Initialize()，它同时获取员工的名字、姓氏和ID。注意，仍然可以从一个构造器中调用另一个构造器，就像代码清单5-31展示的那样。

高级主题：匿名类型

C# 3.0引入了对匿名类型的支持。它们是由编译器动态生成的数据类型，而不是通过显式的类定义来声明的。代码清单5-33展示了这样的一个声明。

代码清单5-33 将匿名类型的实例赋给隐式类型的局部变量

```
using System;

class Program
{
  static void Main()
  {
      var patent1 =
          new
          {
              Title = "Bifocals",
              YearOfPublication = "1784"
          };
      var patent2 =
          new
          {
              Title = "Phonograph",
```

```
            YearOfPublication = "1877"
        };
    var patent3 =
        new
        {
            patent1.Title,
            Year = patent1.YearOfPublication
        };

    System.Console.WriteLine("{0} ({1})",
        patent1.Title, patent1.YearOfPublication);
    System.Console.WriteLine("{0} ({1})",
        patent2.Title, patent1.YearOfPublication);

    Console.WriteLine();
    Console.WriteLine(patent1);
    Console.WriteLine(patent2);

    Console.WriteLine();
    Console.WriteLine(patent3);
  }
}
```

输出5-8展示了结果。

输出5-8

```
Bifocals (1784)
Phonograph (1877)

{ Title = Bifocals, YearOfPublication = 1784 }
{ Title = Phonograph, YearOfPublication = 1877 }

{ Title = Bifocals, Year = 1784 }
```

代码清单5-33演示了如何将匿名类型的值赋给隐式类型的（var）局部变量。

编译器遇到匿名类型的语法时，会自动生成一个CIL类，该类具有与匿名类型声明中的已经命名的值和数据类型对应的属性。虽然C#没有为自动生成的类型提供名称，但该类型仍然是静态类型（而非动态类型）。例如，类型的属性具有完全的可访问性。代码清单5-33中在Console. WriteLine()语句中调用patent1.Title和patent2.YearOfPublication。试图调用不存在的成员会造成编译错误。在Visual Studio中，“智能感知”功能可以正确地支持匿名类型。

代码清单5-33向一个名称赋值，从而显式地标识了匿名类型的成员名称，具体请参见patent1和patent2赋值时使用的Title和YearOfPublication这两个名称。然而，如果所赋的值是属性或字段，那么在没有明确指定名称的前提下，会默认使用字段或属性的名称。例如，patent3用属性名称“Title”定义，而不是向隐式的名称赋值。如输出5-8所示，编译器根据从中获取值的那个属性来决定最终的属性名称。

虽然编译器支持像代码清单5-33那样的匿名类型声明，但除非使用Lambda表达式和查询表达式关联来自不同类型的数据，或者对数据进行水平投射（以减少特定类型的总体数据量），否则一般情况下还是应该尽量避免使用匿名类型，甚至避免使用var来指定隐式类型的变量。除非需要频繁查询集合中的数据而使显式的类型声明成为负担，否则最好还是像本章描述的那样显式地声明类型。

5.8 静态成员

关键字static在第1章的HelloWorld例子中简单地接触过。本节将完整地定义关键字static。

先来考虑一个例子。假定每个员工Id值都必须是不重复的。为此，一个办法是存储一个计数器来跟踪每个员工ID。但如果用实例字段存储值，每次实例化对象时，都要创建一个新的NextId字段，造成Employee对象的每个实例都因为那个字段消耗内存。最大的问题在于，每次实例化Employee对象，以前实例化的所有Employee对象的NextId值都要用下一个ID值来进行更新。所以需要一个单独的字段，它能由所有Employee对象实例共享。

语言对比：C++/Visual Basic——全局变量和函数

和以前的许多语言不同，C#没有全局变量或全局函数。C#的所有字段和方法都在类的上下文中。在C#中，与全局字段或函数等价的是静态字段或方法。“全局变量/函数”和“C#静态字段/方法”没有功能上的差异，只是静态字段/方法可以包含访问修饰符，比如private，从而对访问进行限制，并提供更好的封装。

5.8.1 静态字段

为了定义能由多个实例共享的数据，需要使用static关键字，如代码清单5-34所示。

代码清单5-34 声明一个静态字段

```
class Employee
{
  public Employee(string firstName, string lastName)
  {
      FirstName = firstName;
      LastName = lastName;
      Id = NextId;
      NextId++;
  }

  // ...

  public static int NextId;
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Salary { get; set; }

  // ...
}
```

在这个例子中，NextId字段声明包含static修饰符，所以称为**静态字段**。和Id不同，Employee的所有实例都共享一个NextId存储位置。在Employee构造器内，先将NextId的值赋给新Employee对象的Id，然后立即对NextId进行递增。创建另一个Employee对象时，由于NextId的值已经递增，所以新Employee对象的Id字段将获得不同的值。

实例字段，也就是非静态字段，可以在声明的同时进行初始化。静态字段也可以，如代码清单5-35所示。

代码清单5-35 声明的同时为静态字段赋值

```
class Employee
{
  // ...
  public static int NextId = 42;
  // ...
}
```

和实例字段不同，未初始化的静态字段将获得默认值（0、null、false等，即default(T)的结果，其中T是类型名）。所以，即使没有显式赋值的静态字段也能被访问。

非静态字段（实例字段）针对它们从属于的每个对象，都有一个新的值。静态字段不从属于实例，而是从属于类。因此，从类的外部访问静态字段时要使用类名。下面研究一下代码清单5-36展示的新的Program类（它使用了来自代码清单5-34的Employee类）。

代码清单5-36 访问静态字段

```
using System;

class Program
{
  static void Main()
  {
      Employee.NextId = 1000000;

      Employee employee1 = new Employee(
          "Inigo", "Montoya");
      Employee employee2 = new Employee(
          "Princess", "Buttercup");

      Console.WriteLine(
          "{0} {1} ({2})",
          employee1.FirstName,
          employee1.LastName,
          employee1.Id);
      Console.WriteLine(
          "{0} {1} ({2})",
          employee2.FirstName,
          employee2.LastName,
          employee2.Id);

      Console.WriteLine("NextId = {0}", Employee.NextId);
  }

  // ...
}
```

输出5-9显示了代码清单5-36的结果。

输出5-9

```
Inigo Montoya (1000000)
Princess Buttercup (1000001)
NextId = 1000002
```

设置和获取静态字段NextId的初始值要使用类名Employee，而不是使用对类型的实例的引用。只有在类（或派生类）内部的代码中，才可以省略类名。换言之，Employee(...)

构造器不需要使用Employee.NextId。这些代码已经在Employee类的上下文中出现过了，所以从作用域就能得知上下文。变量的作用域是可以不加限定来引用的程序代码区域，而静态字段的作用域是类（和任何派生类）。

虽然引用静态字段的方式与引用实例字段的方式稍有区别，但不能在同一个类中定义同名的静态字段和实例字段。引用错误字段的几率会很高，C#的设计者决定禁止这样的代码。所以，重复的名称在声明空间中会造成冲突。

初学者主题：数据能够与类和对象关联

类和对象都能关联数据。将类想象成模具，将对象想象成根据这个模具浇铸出来的零件，可以更好地理解这一点。

例如，一个模具拥有的数据可能包括：到目前为止已经用它浇铸出来的零件数、下一个零件的序列号、当前注入模具的液态塑料的颜色以及它每小时生产的零件数量。类似地，零件也拥有它自己的数据：它的序列号、它的颜色以及生产日期和时间。虽然零件的颜色就是生产零件时在模具中注入的塑料的颜色，但它显然不包含模具中当前注入的塑料颜色数据，也不包含要生产的下一个零件的序列号数据。

设计对象时，程序员要考虑字段和方法应该声明成静态的还是基于实例的。通常，应将不需要访问任何实例数据的方法声明为静态方法，将需要访问实例数据的方法（实例不作为参数传给方法）声明为实例方法。静态字段主要存储对应于类的数据，如新实例的默认值，或是已创建实例的个数，等等。而实例字段主要存储与对象关联的数据。

5.8.2 静态方法

和静态字段一样，要直接在类名之后访问静态方法（如Console.ReadLine()）。访问这种方法不需要有实例。

代码清单5-37又展示了一个声明和调用静态方法的例子。

代码清单5-37 为DirectoryInfo类定义静态方法

```
public static class DirectoryInfoExtension
{
  public static void CopyTo(
      DirectoryInfo sourceDirectory, string target,
      SearchOption option, string searchPattern)
  {
      if (target[target.Length - 1] !=
            Path.DirectorySeparatorChar)
      {
          target += Path.DirectorySeparatorChar;
      }
      if (!Directory.Exists(target))
      {
          Directory.CreateDirectory(target);
      }

      for (int i = 0; i < searchPattern.Length; i++)
      {
          foreach (string file in
              Directory.GetFiles(
                  sourceDirectory.FullName, searchPattern))
          {
```

```
            File.Copy(file,
                target + Path.GetFileName(file), true);
        }
    }

    //Copy SubDirectories (recursively)
    if (option == SearchOption.AllDirectories)
    {
        foreach(string element in
            Directory.GetDirectories(
                sourceDirectory.FullName))
        {
            Copy(element,
                target + Path.GetFileName(element),
                searchPattern);
        }
    }
  }
}
```

```
        // ...
        DirectoryInfo directory = new DirectoryInfo(".\\Source");
        directory.MoveTo(".\\Root");
        DirectoryInfoExtension.CopyTo(
            directory, ".\\Target",
            SearchOption.AllDirectories, "*");
        // ...
```

DirectoryInfoExtension.CopyTo()方法获取一个DirectoryInfo对象，并将基础目录结构复制到新位置。

由于静态方法不通过实例引用，所以this关键字在静态方法中无效。除此之外，要在静态方法内部直接访问实例字段或实例方法，必须先获得对字段或方法所从属于的那个特定的实例的引用。（Main()就是静态方法。）

该方法本应由System.IO.Directory类提供，或作为System.IO.DirectoryInfo类的实例方法提供。但由于两个类都没有提供，所以代码清单5-37在一个全新的类中定义了该方法。在本章后面讲述扩展方法的小节中，将解释如何使它“表现”为DirectoryInfo类的实例方法。

5.8.3 静态构造器

除了静态字段和方法，C#还支持**静态构造器**，用于对类（而不是类的实例）进行初始化。静态构造器不显式调用；相反，“运行时”在首次访问类时自动调用静态构造器。“首次访问类”发生在调用普通构造器的时候，也可能发生在访问类的静态方法或字段时。由于静态构造器不能显式调用，所以不允许任何参数。

使用静态构造器将类中的静态数据初始化成特定的值，尤其是无法通过声明时的一次简单赋值来获得初始值的时候。代码清单5-38展示了一个例子。

代码清单5-38 声明静态构造器

```
class Employee
{
  static Employee()
  {
```

```
        Random randomGenerator = new Random();
        NextId = randomGenerator.Next(101, 999);
    }

    // ...
    public static int NextId = 42;
    // ...
}
```

代码清单5-38将NextId的初始值设为100～1 000的随机整数。由于初始值涉及方法调用，所以NextId的初始化代码被放到一个静态构造器中，而没有作为声明的一部分。

假如对NextId的赋值既在静态构造器中进行，又在声明时进行，那么当初始化结束的时候，最终获得的是什么值呢？观察C#编译器生成的CIL代码，发现声明时的赋值被移动了位置，成为静态构造器中的第一个语句。因此，NextId最终包含的是由randomGenerator.Next(101, 999)生成的随机数，而不是声明NextId时赋的值。结论是，在静态构造器中进行的赋值，将优先于声明时的赋值，这和实例字段的情况一样。注意，没有“静态终结器”的说法。

注意，不要在静态构造器中抛出异常，这会造成类型在应用程序剩余的生存期[①]内无法使用。

高级主题：最好是在声明的同时完成静态初始化

静态构造器在对类的任何成员进行首次访问之前执行，无论该成员是静态字段，是其他静态成员，还是实例构造器。为了支持这一行为，编译器会添加代码来检查类型的所有静态成员和构造器，确保静态构造器先运行。

没有静态构造器，编译器会将所有静态成员初始化为它们的默认值，而且不会添加对静态构造器的检查。结果是静态字段会在访问前得到初始化，但不一定在调用静态方法或任何实例构造器之前。有时对静态成员进行初始化的代价比较高，而且在访问前确实没必要初始化，所以这个设计能带来性能上的一定改进。总之，应考虑以内联（inline）的方式初始化静态字段，而不要使用静态构造器，也不一定非要在声明时初始化。

规范

考虑以内联方式初始化静态字段，不要使用静态构造器或者在声明时赋值。

5.8.4 静态属性

还可以将属性声明为static。例如，代码清单5-39将NextId数据包装到属性中。

代码清单5-39 声明静态属性

```
class Employee
{
    // ...
    public static int NextId
    {
        get
```

① 更准确的说法是“应用程序域生存期”，即“操作系统进程”在CLR中的虚拟等价物。

```
        {
            return _NextId;
        }
        private set
        {
            _NextId = value;
        }
    }
    public static int _NextId = 42;
    // ...
}
```

使用静态属性几乎肯定要比使用公共静态字段好，因为公共静态字段在任何地方都能调用，而静态属性则至少提供了一定程度的封装。

5.8.5 静态类

有的类不包含任何实例字段。例如，假定SimpleMath类包含与数学运算Max()和Min()对应的函数，如代码清单5-40所示。

代码清单5-40 声明静态类

```
// Static class introduced in C# 2.0
public static class SimpleMath
{
  // params allows the number of parameters to vary.
  static int Max(params int[] numbers)
  {
    // Check that there is a least one item in numbers.
    if(numbers.Length == 0)
    {
        throw new ArgumentException(
            "numbers cannot be empty");
    }

    int result;
    result = numbers[0];
    foreach (int number in numbers)
    {
        if(number > result)
        {
            result = number;
        }
    }
    return result;
  }

  // params allows the number of parameters to vary.
  static int Min(params int[] numbers)
  {
      // Check that there is a least one item in numbers.
      if(numbers.Length == 0)
      {
          throw new ArgumentException(
              "numbers cannot be empty");
      }

      int result;
```

```
            result = numbers[0];
            foreach (int number in numbers)
            {
                if(number < result)
                {
                    result = number;
                }
            }
            return result;
        }
    }
```

该类不包含任何实例字段（或方法），所以创建能实例化的类没有意义。所以用static关键字修饰该类。在声明类时使用static关键字，具有两个方面的意义。首先，它防止程序员写代码来实例化SimpleMath类。其次，它防止在类的内部声明任何实例字段或方法。既然类无法实例化，实例成员当然也就没有了意义。以前的代码清单中的Program类也应该设计成静态类，因为它只包含静态成员。

静态类的另一个特点是C#编译器自动在CIL代码中把它标记为abstract和sealed。这会将类指定为**不可扩展**；换言之，不能从它派生出其他类。

5.9 扩展方法

来考虑用于操纵文件系统目录的System.IO.DirectoryInfo类。类支持的功能包括列出文件和子目录（DirectoryInfo.GetFiles()）以及移动目录（DirectoryInfo.Move()）。但它不直接支持复制功能。需要这样的方法就得自己去实现，如本章前面的代码清单5-37所示。

当初声明的DirectoryInfoExtension.CopyTo()是标准的静态方法。但是， CopyTo()方法在调用方式上有别于DirectoryInfo.Move()。这是令人遗憾的一个设计。理想情况下，我们想为DirectoryInfo添加一个方法，用以在获得一个实例的情况下，将CopyTo()作为实例方法来调用，例如directory.CopyTo()。

C# 3.0引入了**扩展方法**的概念，能模拟为不同的类创建实例方法。只需更改静态方法的签名，使第一个参数成为要扩展的类型，并在类型名称前面附加this关键字，如代码清单5-41所示。

代码清单5-41 DirectoryInfo的静态CopyTo()方法

```
public static class DirectoryInfoExtension
{
    public static void CopyTo(
        this DirectoryInfo sourceDirectory, string target,
        SearchOption option, string searchPattern)
    {
        // ...
    }
}
```

```
// ...
    DirectoryInfo directory = new DirectoryInfo(".\\Source");
    directory.CopyTo(".\\Target",
        SearchOption.AllDirectories, "*");
// ...
```

这个改进使我们能为任何类添加实例方法，即使是那些不在同一个程序集中的类。但查看CIL代码，会发现扩展方法是作为普通静态方法调用的。

扩展方法的要求如下。

- 第一个参数是要扩展或者要操作的类型，这称为“被扩展的类型”。
- 为了指定扩展方法，要在被扩展的类型名称前面附加this修饰符。
- 要将方法作为一个扩展方法来访问，要用using指令导入扩展类型[①]的命名空间，或者使扩展类型和调用代码在同一个命名空间中。

如果扩展方法的签名已经和被扩展类型中的签名匹配（换言之，假如DirectoryInfo已经有一个CopyTo()方法了），扩展方法永远不会得到调用，除非是作为一个普通的静态方法。

注意，通过继承来特化类型（将于第6章讲述）要优于使用扩展方法。扩展方法无益于建立清楚的版本控制机制，因为一旦在被扩展的类型中添加了匹配的签名，就会覆盖现有的扩展方法，而且不会发出任何警告。如果对被扩展的类的源代码没有控制权，这个问题还会变得更加突出。另一个问题是，虽然Visual Studio的“智能感知”支持扩展方法，但假如只是查看调用代码（也就是调用了扩展方法的代码），是不易看出一个方法是不是扩展方法的。

总之，扩展方法要慎用。例如，不要为object类型定义扩展方法。第6章将讨论扩展方法如何与接口配合使用。没有这种配合，就很少定义扩展方法。

规范

避免轻率地定义扩展方法，尤其是要避免为自己没有所有权的类型定义扩展方法。

5.10 封装数据

除了本章前面讨论的属性和访问修饰符，还有其他几种特殊的方式可以将数据封装到类中。例如，还有另外两个字段修饰符。第一个是const修饰符，声明局部变量时已经遇到过它。第二个是readonly。

5.10.1 const

和const值一样，const字段（称为常量字段）包含在编译时确定的值，它不可以在运行时改变。像pi这样的值就很适合声明为常量字段。代码清单5-42展示了如何声明常量字段。

代码清单5-42 声明常量字段

```
class ConvertUnits
{
   public const float CentimetersPerInch = 2.54F;
   public const int CupsPerGallon = 16;
   // ...
}
```

常量字段自动成为静态字段，因为不需要为每个对象实例都生成新的字段实例。但是，

① 即对“被扩展的类型”进行扩展的那个类型，或者说明扩展办法的那个类型。——译者注

将常量字段显式声明为static会造成编译错误。另外，常量字段通常只使用包含字面值的类型（例如string、int和double等）。Program或System.Guid等类型不能用于常量字段。

在public常量表达式中，必须使用随着时间的推移不会发生变化的值。圆周率、阿伏加德罗常数以及赤道长度这样的值都是很好的例子。以后可能发生变化的值就不适合。例如，版本号、人口数量和汇率都不适合作为常量。

规范

要为永远不变的值使用常量字段。

不要为将来会发生变化的值使用常量字段。

高级主题：public常量应该是恒定值

public常量应该恒定不变，因为如果对它进行了修改，在使用它的程序集中不一定能反映出最新改变。如果一个程序集引用了另一个程序集中的常量，常量值将直接编译到引用程序集中。所以，如果被引用程序集中的值发生改变，而且引用程序集没有重新编译，那么引用程序集将继续使用原始值，而不是新值。将来可能改变的值应该指定为readonly，不要指定为常量。

5.10.2 readonly

和const不同，readonly修饰符只能用于字段（不能用于局部变量），它指出字段值只能从构造器中更改，或者直接在声明时指定。代码清单5-43演示了如何声明readonly字段。

代码清单5-43 声明readonly字段

```
class Employee
{
  public Employee(int id)
  {
      Id = id;
  }

  // ...
  public readonly int Id;
  public void SetId(int newId)
  {
      // ERROR:  read-only fields cannot be set
      //         outside the constructor.
      // Id = newId;
  }

  // ...
}
```

和const字段不一样，每个实例的readonly字段都可以不同。事实上，readonly字段的值可以在构造器中更改。除此之外，readonly字段既可以是实例字段，也可以是静态字段。另一个关键区别在于，可以在执行时为readonly字段赋值，而非只能在编译时赋值。由于readonly字段必须从构造器中设置，所以编译器要求这种字段能从其属性外部访问。但除此之外，不要从除了包容属性之外的任何地方访问属性的支持字段。

和const字段相比，readonly字段的另一个重要特点是不限于包含字面值的类型。例如，可以声明readonly System.Guid实例字段，但不可以声明类似的常量字段，如代码5-44所示。

代码清单5-44 声明不包含字面值的类型的readonly字段

```
class CommonGuid
{
  public static readonly Guid ComIUnknownGuid =
      new Guid("00000000-0000-0000-C000-000000000046");
  public static readonly Guid ComIClassFactoryGuid =
      new Guid("00000001-0000-0000-C000-000000000046");
  public static readonly Guid ComIDispatchGuid =
      new Guid("00020400-0000-0000-C000-000000000046");
  public static readonly Guid ComITypeInfoGuid =
      new Guid("00020401-0000-0000-C000-000000000046");
  // ...
}
```

将readonly应用于数组，不会冻结数组的内容。只会冻结数组中的元素数量，因为无法将只读字段重新赋给新的实例。不过，数组中的元素仍然是可写的。

规范

要为预定义对象实例使用public static readonly字段。

5.11 嵌套类

在类中除了定义方法和字段，还可以定义另一个类。这称为**嵌套类**。假如一个类在它的包容类外部没有多大意义，就适合把它设计成嵌套类。

假定有一个类用于处理程序的命令行选项。通常，像这样的类在每个程序中的处理方式都是不同的，没有理由使CommandLine类能够从包含Main()的那个类的外部访问。代码清单5-45演示了这样的一个嵌套类。

代码清单5-45 定义嵌套类

```
// CommandLine is nested within Program
class Program
{
// Define a nested class for processing the command line.
private class CommandLine
{
    public CommandLine(string[] arguments)
    {
        for(int argumentCounter=0;
            argumentCounter<arguments.Length;
            argumentCounter++)
        {
            switch (argumentCounter)
            {
                case 0:
                    Action = arguments[0].ToLower();
                    break;
```

```
                case 1:
                    Id = arguments[1];
                    break;
                case 2:
                    FirstName = arguments[2];
                    break;
                case 3:
                    LastName = arguments[3];
                    break;
            }
        }
    }
    public string Action;
    public string Id;
    public string FirstName;
    public string LastName;
}

static void Main(string[] args)
{
    CommandLine commandLine = new CommandLine(args);

    switch (commandLine.Action)
    {
        case "new":
            // Create a new employee
            // ...
            break;
        case "update":
            // Update an existing employee's data
            // ...
            break;
        case "delete":
            // Remove an existing employee's file.
            // ...
            break;
        default:
            Console.WriteLine(
                "Employee.exe " +
 "new|update|delete <id> [firstname] [lastname]");
            break;
    }
  }
}
```

这个例子中的嵌套类是Program.CommandLine。和所有类成员一样，在包容类的内部，没有必要使用包容类的名称作为前缀。所以，可以直接把它引用为CommandLine。

嵌套类的独特之处是可以为类自身指定private访问修饰符。由于类的作用是解析命令行，并将每个参数放到一个单独的字段中，因此它在这个应用程序中只和Program类有关系。使用private访问修饰符，可以限定类的作用域，防止从类的外部访问。只有在类是嵌套类的时候，才可以这样做。

嵌套类中的this成员代表嵌套类而不是包容类的实例。嵌套类要想访问包容类的实例，一个办法是显式传递包容类的实例，比如通过构造器或者方法参数。

嵌套类的另一个有趣的特点是它能访问包容类的任何成员，其中包括私有成员。反之则不然，包容类不能访问嵌套类的私有成员。

嵌套类用得很少。要从包容类型外部引用，就不能定义成嵌套类。另外要警惕public嵌套类；它们意味着不良的编码风格，可能造成混淆和难以阅读。

规范

避免声明公共嵌套类型。唯一的例外是在这种类型的声明没有多大意义的时候，或者这种类型的声明是与一种高级的自定义场景有关。

语言对比：Java——内部类

Java不仅支持嵌套类，还支持内部类（inner class）。内部类对应的是同包容类实例关联的对象，而不是仅仅和包容类有一个语法上的包容关系。在C#中，可以在外层类中包含嵌套类型的一个实例字段，从而得到相同的结构。一个工厂方法或构造器可确保在内部类的实例中，也会设置对外部类的相应实例的引用。

5.12 分部类

C# 2.0新增的另一个语言特性是**分部类**。分部类是一个类的多个部分，这些部分可以合并成一个完整的类。可在同一个文件中定义两个或者更多的分部类，但分部类主要用于将一个类的定义划分到多个文件中。分部类对代码生成或修改工具来说意义重大。使用分部类，这些工具操作的文件可以独立于开发者正在人工编码的文件。

5.12.1 定义分部类

C# 2.0（和更高版本）使用class前的上下文关键字partial来声明分部类，如代码清单5-46所示。

代码清单5-46 定义分部类

```
// File: Program1.cs
partial class Program
{
}
```

```
// File: Program2.cs
partial class Program
{
}
```

在这个例子中，Program的每个部分都放到单独的文件中（参见注释）。除了用于代码生成器，分部类另一个常见的应用是将每个嵌套类都放到它们自己的文件中。这是为了与编程规范“将每个类定义都放到它自己的文件中”保持一致。例如，代码清单5-47将Program.CommandLine类放到和核心Program成员分开的文件中。

代码清单5-47 在一个分部类中定义嵌套类

```
// File: Program.cs
partial class Program
```

```
{
  static void Main(string[] args)
  {
    CommandLine commandLine = new CommandLine(args);

    switch (commandLine.Action)
    {
       // ...
    }
  }
}
```

```
// File: Program+CommandLine.cs
partial class Program
{
  // Define a nested class for processing the command line.
  private class CommandLine
  {
     // ...
  }
}
```

分部类不允许对编译好的类（其他程序集中的类）进行扩展。只能利用分部类在同一个程序集中将一个类的实现拆分到多个文件中。

5.12.2 分部方法

自C# 3.0起，语言的设计者引入了**分部方法**的概念，它对C# 2.0的分部类进行了扩展。分部方法只能存在于分部类中。另外，和分部类相似，其主要作用是为代码生成提供方便。

假定代码生成工具能根据数据库中的Person表为Person类生成对应的Person.Designer.cs文件。这个工具检查表的内容，为表中的每一列都创建属性。问题在于，工具经常都不能生成必要的验证逻辑，因为这些逻辑依赖于未在表定义中嵌入的业务规则。所以，Person类的开发者需要自己添加验证逻辑。Personal.Designer.cs是不好直接修改的，因为假如文件被重新生成（例如，为了适应数据库中新增的一个列），所做的更改就会丢失。相反，Person类的代码结构应独立出来，使生成的代码在一个文件中，自定义代码（带有业务规则）在另一个文件中，后者不受任何重新生成动作的影响。如上一节所示，分部类很适合将一个类打散到多个文件中。但有时这样还不够。经常还需要**分部方法**。

分部方法允许声明方法而不需要实现。然而，如果包含了可选的实现，这个实现就可以放到某个姊妹分部类定义中——可能在单独的文件中。代码清单5-48展示了如何为Person类声明和实现分部方法。

代码清单5-48 为Person类声明和实现分部方法

```
// File: Person.Designer.cs
public partial class Person
{
    #region Extensibility Method Definitions
    partial void OnLastNameChanging(string value);
    partial void OnFirstNameChanging(string value);
    #endregion

    // ...
    public System.Guid PersonId
```

```
{
    // ...
}
private System.Guid _PersonId;

// ...
public string LastName
{
    get
    {
        return _LastName;
    }
    set
    {
        if ((_LastName != value))
        {
            OnLastNameChanging(value);
            _LastName = value;
        }
    }
}
private string _LastName;

// ...
public string FirstName
{
    get
    {
        return _FirstName;
    }
    set
    {
        if ((_FirstName != value))
        {
            OnFirstNameChanging(value);
            _FirstName = value;
        }
    }
}
private string _FirstName;

}
```

```
// File: Person.cs
partial class Person
{
  partial void OnLastNameChanging(string value)
  {
      if (value == null)
      {
          throw new ArgumentNullException("LastName");
      }
      if(value.Trim().Length == 0)
      {
          throw new ArgumentException(
              "LastName cannot be empty.");
      }
  }
}
```

在Person.Designer.cs的代码清单中，包含了OnLastNameChanging()和OnFirstNameChanging()方法的声明。除此之外，LastName和FirstName属性调用了它们对应的Changing方法。虽然这两个方法只有声明而没有实现，但却能成功通过编译。关键在于方法声明附加了上下文关键字partial，其所在的类也是一个partial类。

代码清单5-48只实现了OnLastNameChanging()方法。在这个实现中，会检查建议的新的LastName值，如果发现无效，就引发异常。注意两个地方的OnLastNameChanging()签名是匹配的。

注意，分部方法必须返回void。如果不返回null，同时没有提供实现，那么调用未实现的方法，返回什么才算合理呢？为了避免对返回值进行任何无端的猜测，C#的设计者决定只允许方法返回void。类似地，out参数在分部方法中不允许。需要返回值可以使用ref参数。

总之，分部方法使生成的代码能调用并非一定要实现的方法。除此之外，如果没有为分部方法提供实现，CIL中不会出现分部方法的任何踪迹。这样在保持代码尽量小的同时，还保证了高的灵活性。

5.13 小结

本章讲解了C#类以及面向对象程序设计。讨论了字段，并讨论了如何在类的实例上访问它们。

是在每个实例上都存储一份数据，还是为一个类型的所有实例统一存储一份数据？本章仔细讲解了应该如何取舍。静态数据同类关联，而实例数据在每个对象上存储。

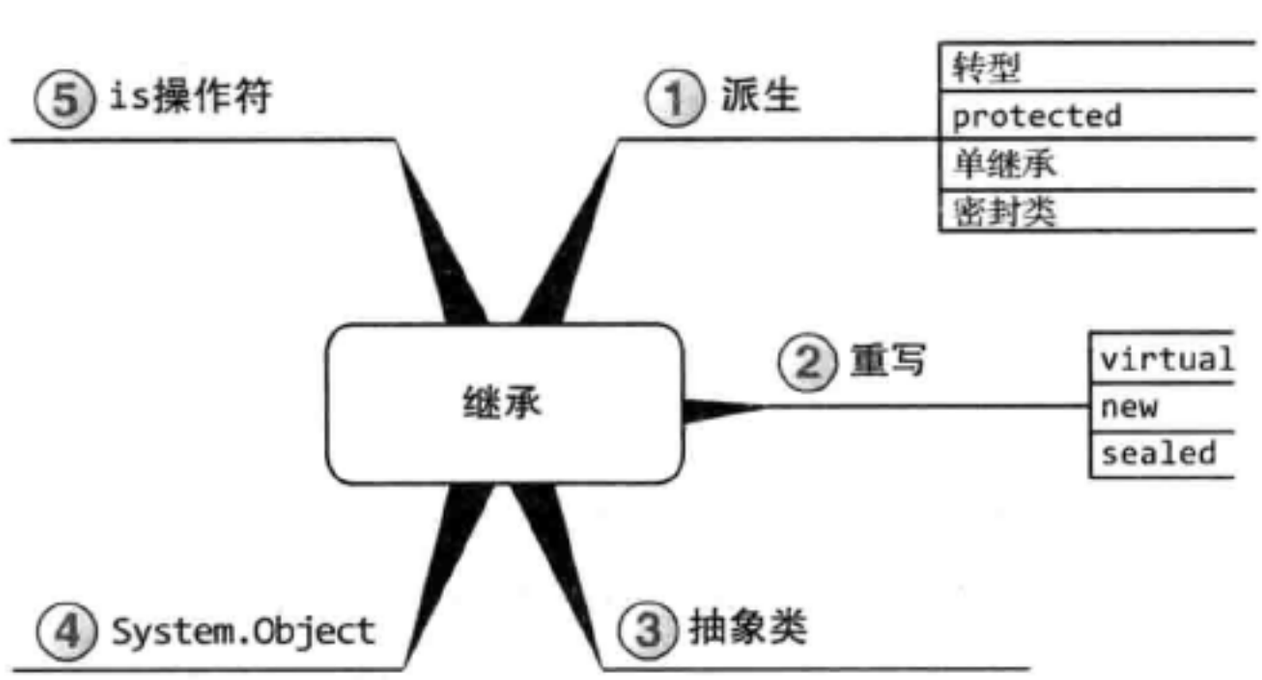
本章以方法和数据的访问修饰符为背景探讨了封装问题。介绍了C#属性，并解释了如何用它封装私有字段。

下一章将学习如何通过“继承”将不同的类关联起来，并学习继承这一面向对象的构造给编程思维带来的益处。

第6章

继承

第5章讨论了一个类如何通过字段和属性来引用其他类。本章讨论了如何利用类的继承关系来构建类层次结构。



初学者主题：继承的定义

第5章已经简单介绍了继承。下面是对已定义的术语的简单回顾。

- 派生/继承：对基类进行特化，包含附加的成员或者对基类成员进行自定义。
- 派生类型/子类型：继承了较常规类型的成员的特化类型。
- 基/超/父类型：其成员由派生类型继承的常规类型。

继承建立了“属于”关系。派生类型总是隐式地属于基类型。如同硬盘属于存储设备一样，从存储设备类型派生的其他任何类型都属于存储设备。反之则不成立。存储设备不一定是硬盘。

注意

代码中的继承用于定义“属于”关系，派生类是对基类的特化。

6.1 派生

经常需要扩展现有类型来添加更多功能（行为和数据）。继承正是为了这个目的而设计的。例如，假定已经有Person类，可以创建Employee类，在其中添加EmployeeId和Department等员工特有的属性。也可以采取相反的操作。例如，假定已经有Contact类，它在PDA（个人数字助理）中使用，而你希望为PDA添加日历支持。为此，可以创建Appointment类。然而，不用重新定义这两个类都适用的方法和属性，可以对Contact类进行**重构**。具体地说，可以将两者都适用的方法和属性从Contact移动到名为PdaItem的基类中，并让Contact和Appointment都从这个基类派生，如图6-1所示。

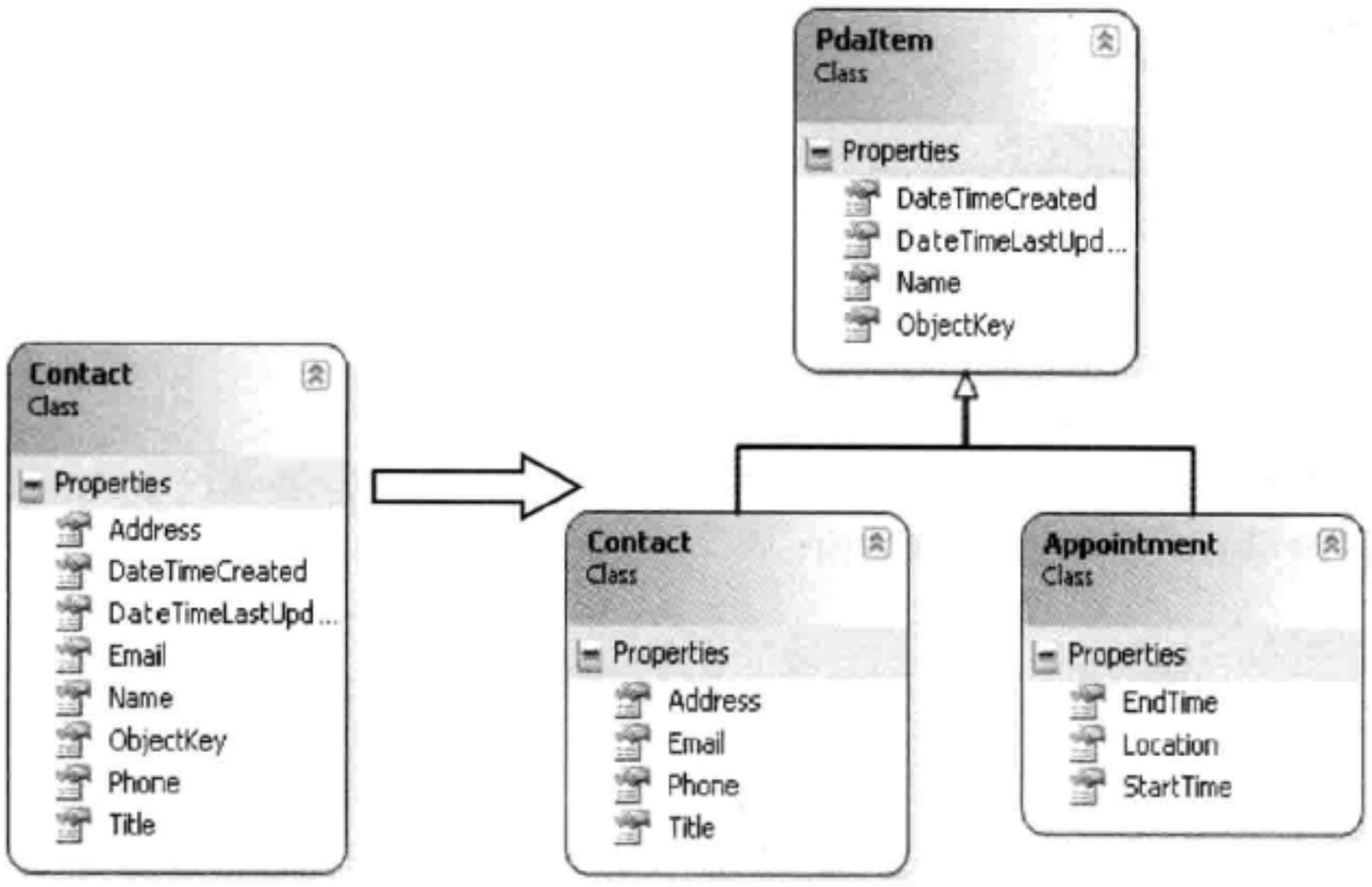


图6-1 重构形成一个基类

在这个例子中，两者共用的项是Created、LastUpdated、Name和ObjectKey等。通过派生，在基类PdaItem中定义的方法可以从PdaItem的所有子类中访问。

定义派生类要在类标识符后面添加一个冒号，接着添加基类名称，如代码清单6-1所示。

代码清单6-1 从一个类派生出另一个类

```
public class PdaItem
{
  public string Name { get; set; }

  public DateTime LastUpdated { get; set; }
}
```

```
// Define the Contact class as inheriting the PdaItem class
public class Contact : PdaItem
{
  public string Address { get; set; }
  public string Phone { get; set; }
}
```

代码清单6-2展示了如果访问Contact中定义的属性。

代码清单6-2 使用继承的属性

```
public class Program
{
  public static void Main()
  {
      Contact contact = new Contact();
      contact.Name = "Inigo Montoya";

      // ...
  }
}
```

虽然Contact并非直接定义了名为Name的属性，但Contact的所有实例仍然可以访问来自PdaItem的Name属性，并把它作为Contact的一部分使用。除此之外，从Contact派生的其他任何类也会继承PdaItem类（或者PdaItem的父类）的成员。这个继承链是没有限制的，每个派生类都拥有由其所有基类公开的全部成员，如代码清单6-3所示。

> **注意**
>
> 通过继承，基类的每个成员都会出现在派生类的链条中。

代码清单6-3 逐个派生的类形成继承链

```
public class PdaItem : object
{
  // ...
}
```

```
public class Appointment : PdaItem
{
  // ...
}
```

```
public class Contact : PdaItem
{
  // ...
}
```

```
public class Customer : Contact
{
  // ...
}
```

换言之，虽然Customer不直接从PdaItem派生，但它仍然继承了PdaItem的成员。

在代码清单6-3中，PdaItem显式地从object派生。虽然C#允许这样写，但却没有必要。所有类都隐式地派生于object，不管是否这样指定。

> **注意**
>
> 除非明确指定了基类，否则所有类都默认从object派生。

6.1.1 基类型和派生类型之间的转型

如代码清单6-4所示，由于派生建立了“属于”关系，所以总是可以将派生类型的值直接赋给基类型的变量。

代码清单6-4 隐式基类型转换

```
public class Program
{
  public static void Main()
  {
    // Derived types can be implicitly converted to
    // base types
    Contact contact = new Contact();
    PdaItem item = contact;
    // ...

    // Base types must be cast explicitly to derived types
    contact = (Contact)item;
    // ...
  }
}
```

派生类型Contact“属于”PdaItem类型，可以直接赋给PdaItem类型的变量。这称为**隐式转型**，不需要添加转型操作符。而且根据规则，转换总会成功，不会引发异常。

反之则不成立。PdaItem并非一定“属于”Contact。它可能是一个Appointment或其他派生类型。所以，从基类型转换为派生类型，要求执行**显式转型**，而显式转型在运行时可能会失败。为了执行显式转型，要在原始引用名称之前，将要转换成的目标类型放到圆括号中，如代码清单6-4所示。

执行显式转型时，程序员相当于请求编译器信任他，或者说程序员告诉编译器他知道这样做的后果。只要圆括号中的目标类型确实是从基类型派生的，C#编译器就允许这个转换。虽然在编译的时候，C#编译器允许在可能兼容的类型之间执行显式转型，但CLR仍然会在运行时验证这次显式转型。对象实例不属于目标类型将引发异常。

即使类型的层次结构允许进行隐式转型，C#编译器也允许添加转型操作符（虽然纯属多余）。例如，将contact赋给item时，可以像下面这样添加转型操作符：

```
item = (PdaItem)contact;
```

甚至在无需转型的时候，也可以添加一个转型操作符：

```
contact = (Contact)contact;
```

> **注意**
>
> 派生类型可隐式转型为它的基类型。相反，基类向派生类的转换要求显式的转型操作符，因为转换可能会失败。虽然编译器允许可能有效的显式转型，但“运行时”会坚持进行检查，一个非法的转型会引发异常。

初学者主题：在继承链中进行类型转换

隐式转型为基类不会实例化新实例。相反，同一个实例被引用为基类型，而它现在

提供的功能（也就是可访问的成员）是基类型的。这类似于将CD-ROM驱动器说成是一种存储设备。由于并非所有存储设备都支持弹出操作，因此CD-ROM在转型为存储设备之后，就不能再支持弹出。调用storageDevice.Eject()，即使被转型的对象原本是支持Eject()方法的CDROM对象，也无法通过编译。

类似地，将基类向下转型为派生类，会引用更具体的类型，类型可用的操作也会进行扩展。但这种转换是有限制的，被实例化的类型必须是要转换成的目标类型（或者它的某个派生类型）的实例。

高级主题：定义自定义转换

类型间的转换并不限于单一继承链中的类型。完全不相关的类型相互之间也能进行转换。关键在于在两个类型之间提供转型操作符。C#允许类型包含显式或隐式转型操作符。在转型有可能失败时，比如从long转型为int，开发者应定义显式转型操作符。这样可以提醒别人只有在确信转型会成功的时候才执行这样的转型；否则，就准备好在失败的时候捕捉异常。执行有损转换时，开发者也应优先进行显式转型而不是隐式转型。例如，将float转型为int，小数部分会被丢弃。即使接着执行一次反向转换（int转型回float），丢失的部分也找不回来。

代码清单6-5演示了在GPSCoordinates和UTMCoordinates这两种类型之间使用的隐式和显式转型操作符。

代码清单6-5 定义转型操作符

```
class GPSCoordinates
{
  // ...

  public static implicit operator UTMCoordinates(
      GPSCoordinates coordinates)
  {
      // ...
  }
}
```

本例写的是从GPSCoordinates向UTMCoordinates的隐式转换。可以写类似的转换来反转上述过程。将implicit替换成explicit就是显式转换。

6.1.2 private访问修饰符

派生类继承了除构造器和析构器之外的所有基类成员。但是，继承并不意味着一定能访问。例如，在代码清单6-6中，private字段_Name不可以在Contact类中使用，因为私有成员只能在声明它们的那个类型中才能访问

代码清单6-6 私有成员继承但不能访问

```
public class PdaItem
{
  private string _Name;
  // ...
}
```

```
public class Contact : PdaItem
{
  // ...
}
```

```
public class Program
{
  public static void Main()
  {
      Contact contact = new Contact();
      // ERROR:  'PdaItem._Name' is inaccessible
      // due to its protection level
      //contact._Name = "Inigo Montoya";
  }
}
```

根据封装原则，派生类不能访问基类的private成员[①]。这样一来，基类开发者就必须明确地决定一个成员是否能由派生类访问。在本例中，基类定义了一个API，其中_Name只能通过Name属性更改。假如以后在Name属性中添加了验证机制，那么所有派生类不需要进行任何修改，就可以马上享受到验证带来的好处，因为它们从一开始就不能直接访问_Name属性。

> **注意**
>
> 非嵌套派生类不能访问基类的私有成员。

6.1.3 protected访问修饰符

除了pubic或private，还可以进行更细致的封装。可在基类中定义只有派生类才能访问的成员。例如，看一下代码清单6-7展示的ObjectKey属性。

代码清单6-7 protected成员只能从派生类访问

```
public class Program
{
  public static void Main()
  {
      Contact contact = new Contact();
      contact.Name = "Inigo Montoya";

      // ERROR:  'PdaItem.ObjectKey' is inaccessible
      // due to its protection level
      // contact.ObjectKey = Guid.NewGuid();
  }
}
```

```
public class PdaItem
{
  protected Guid ObjectKey
  {
      get { return _ObjectKey; }
      set { _ObjectKey = value; }
  }
  private Guid _ObjectKey;

  // ...
```

① 一个极少见的例外情况是假如派生类同时是基类的一个嵌套类。

```
}

public class Contact : PdaItem
{
  void Save()
  {
      // Instantiate a FileStream using <ObjectKey>.dat
      // for the filename.
      FileStream stream = System.IO.File.OpenWrite(
          ObjectKey + ".dat");
  }

  void Load(PdaItem pdaItem)
  {
      // ERROR:  'pdaItem.ObjectKey' is inaccessible
      // due to its protection level
      // pdaItem.ObjectKey = ...;

      Contact contact = pdaItem as Contact;
      if(contact != null)
      {
          contact.ObjectKey = ...;
      }

      // ...
  }
}
```

ObjectKey使用protected访问修饰符定义。结果是在PdaItem的外部，它只能从PdaItem的派生类中访问。由于Contact从PdaItem派生，所以Contact的所有成员都能访问ObjectKey。由于Program不是从PdaItem派生，所以在Program内使用ObjectKey属性会造成编译错误。

> **注意**
>
> 基类中的受保护成员只能从基类及其派生类中访问。

Contact.Load()方法有一个容易被忽视的细节。开发者经常都会惊讶地发现，从Contact类内部的代码中，无法访问一个显式的PdaItem的受保护ObjectKey，即使Contact是从PdaItem派生的。这是由于万一PdaItem是一个Address呢？Contact不应该能够访问Address的受保护成员。所以，封装成功阻止了Contact修改Address的ObjectKey。成功转型为Contact可以绕过这一限制。基本规则是，要从派生类中访问受保护成员，必须在编译时确定是从派生类（或者它的某个子类）的实例中访问受保护成员。

6.1.4 扩展方法

扩展方法从技术上说不是类型的成员，所以不可继承。但由于每个派生类都可作为它的任何基类的实例使用，所以对一个类型进行扩展的方法也可扩展它的任何派生类型。如果扩展基类（如PdaItem），所有扩展方法在派生类中也可以使用。但和所有扩展方法一样，实例方法有更高的优先级。如果在继承链中出现了一个兼容的签名，它将优先于扩展方法。

很少为基类写扩展方法。扩展方法的一个基本原则是，假如手上有基类的代码，直接修改基类会更好。即使基类代码不可用，程序员也应考虑在基类或个别派生类实现的接口上添加扩展方法。下一章将具体讨论接口，并讨论它们如何与扩展方法配合使用。

6.1.5 单继承

继承链中的类理论上是无限的。例如，Customer派生自Contact，Contact派生自PdaItem，PdaItem派生自object。但C#是**单继承**语言，C#编译成的CIL语言也是一样。这意味着一个类不能直接从两个类派生。例如，Contact不能既直接派生自PdaItem，又直接派生自Person。

> **语言对比：C++——多继承**
>
> C#的单继承是其与C++的关键区别之一。对于像**活动模板库**（Active Template Library，ATL）这样的编程库来说，由于它的整个机制都是依赖于多继承的，所以在迁移至C#时，改动会相当大。

在极少数需要多继承类结构的时候，一般的解决方案是使用**聚合**（aggregation）；换言之，不是一个类从另一个类继承，而是一个类包含另一个类的实例。图6-2展示了这种类结构的一个例子。在关联关系中，如果定义了包容对象的一个核心组件，就会发生聚合。对于多继承，这涉及挑选一个类作为主要基类（PdaItem），然后从中派生出一个新类（Contact）。第二个希望的基类（Person）作为派生类（Contact）中的一个字段添加。接着，字段（Person）上的所有非私有成员都在派生类（Contact）上重新定义。然后，派生类（Contact）将调用委托给字段（Person）。由于方法要重新声明，所以一些代码会重复。但是，重复的代码不会很多，因为实际的方法主体只在聚合类（Person）中实现。

在图6-2中，Contact包含名为InternalPerson的私有属性，它被描绘成到Person类的一个关联。Contact也包含FirstName和LastName属性，但是没有对应的字段。相反，FirstName和LastName属性将调用分别委托给InternalPerson.FirstName和InternalPerson.LastName。代码清单6-8展示了最终的代码。

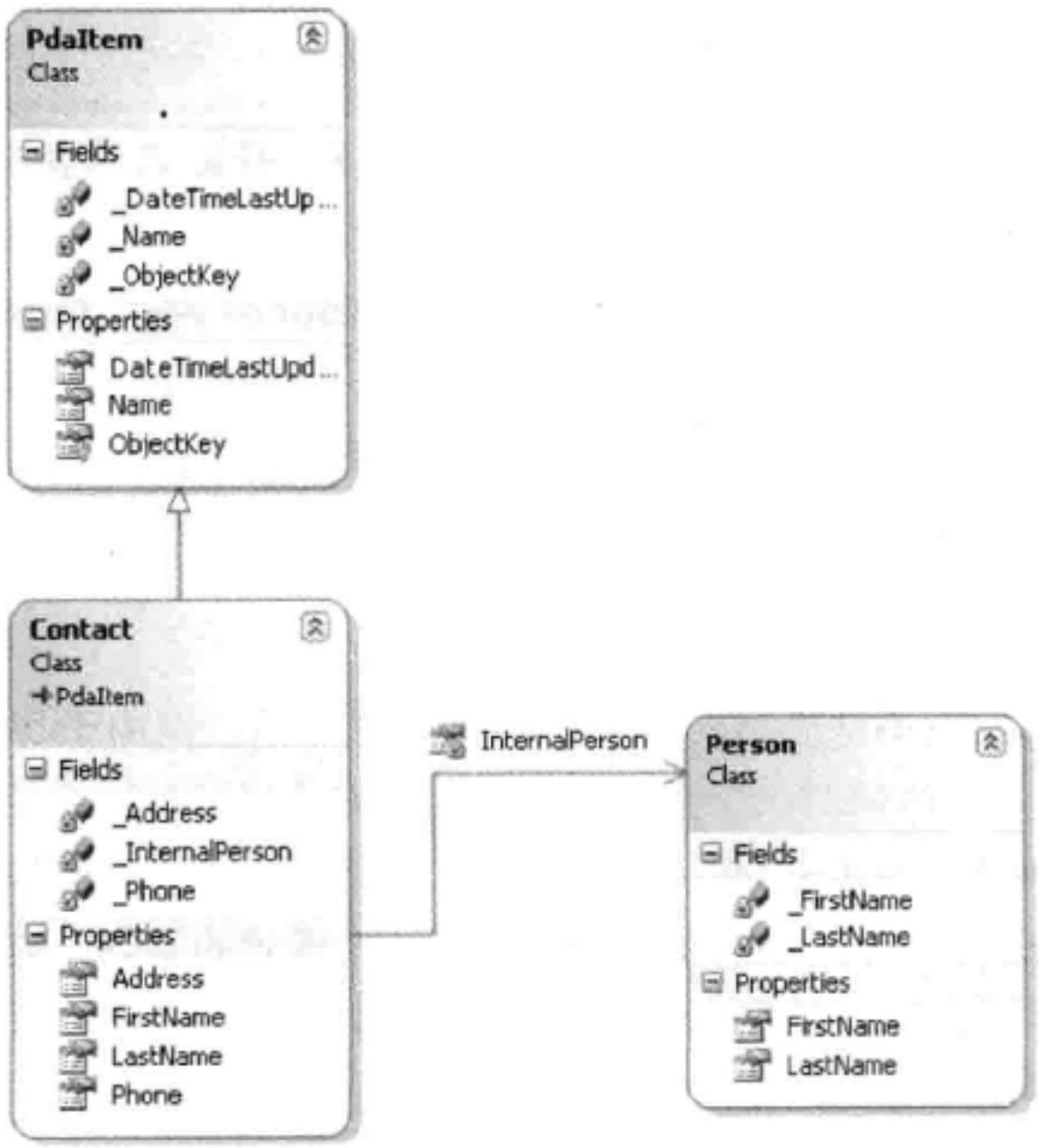


图6-2 用聚合模拟多继承

代码清单6-8 用聚合解决单继承问题

```
public class PdaItem
{
  // ...
}

public class Person
{
  // ...
}

public class Contact : PdaItem
{
  private Person InternalPerson { get; set; }

  public string FirstName
  {
      get { return InternalPerson.FirstName; }
      set { InternalPerson.FirstName = value; }
  }

  public string LastName
  {
      get { return InternalPerson.LastName; }
      set { InternalPerson.LastName = value; }
  }

  // ...
}
```

除了因为委托而增加的复杂性，另一个缺点在于，在字段类（Person）上新增的任何方法都需要人工添加到派生类（Contact）中；否则，Contact无法公开新增的功能。

6.1.6 密封类

为了正确设计类，让其他人通过继承来扩展它的功能，需要对它进行全面测试，验证派生能成功地进行。为了避免非预期的派生，并避免出现因此而产生的问题，可以把类标记为sealed，如代码清单6-9所示。

代码清单6-9 使用密封类防止派生

```
public sealed class CommandLineParser
{
  // ...
}
```

```
// ERROR:  Sealed classes cannot be derived from
public sealed class DerivedCommandLineParser :
  CommandLineParser
{
  // ...
}
```

密封类要求使用sealed修饰符，这样做的结果是不能从它们派生出其他类。string类型就用sealed修饰符禁止了派生。

6.2 基类的重写

基类除构造器和析构器之外的所有成员都会在派生类中继承。但某些情况下，一个成员可能在基类中没有得到最佳的实现。下面以PdaItem的Name属性为例。在由Appointment类继承的时候，它的实现或许是可以接受的。然而，对于Contact类，Name属性应该返回FirstName和LastName属性合并起来的结果。类似地，对Name进行赋值时，应分解成FirstName和LastName。换言之，对于派生类，基类属性的声明是合适的，但实现并非总是合适的。因此，需要一种机制在派生类中使用自定义的实现来重写（override，通常也称为覆盖）基类中的实现。

6.2.1 virtual修饰符

C#支持重写实例方法和属性，但不支持重写字段或者任何静态成员。为了进行重写，要求在基类和派生类中都显式地执行一个操作。在基类中，必须将允许重写的每个成员标记为virtual。假如一个public或protected成员没有包含virtual修饰符，就不允许子类重写该成员。

语言对比：Java——默认虚方法

默认情况下，Java中的方法都是虚方法。假如希望方法具有非虚的行为，就必须显式密封它。相反，C#的方法默认为非虚方法。

代码清单6-10展示了属性重写的一个例子。

代码清单6-10 重写属性

```
public class PdaItem
{
    public virtual string Name { get; set; }
    // ...
}

public class Contact : PdaItem
{
    public override string Name
    {
        get
        {
            return FirstName + " " + LastName;
        }

        set
        {
            string[] names = value.Split(' ');
            // Error handling not shown.
            FirstName = names[0];
            LastName = names[1];
        }
}

public string FirstName { get; set; }
public string LastName { get; set; }
```

```
        // ...
    }
```

PdaItem的Name属性包含了virtual修饰符，Contact的Name则用关键字override进行了修饰。在上例中，拿掉virtual会产生错误，拿掉override则会产生警告，稍后会详细讨论。C#要求显式使用override关键字来重写方法。

换句话说，virtual标志着方法或属性可在派生类中被替换（重写）。

语言对比：Java和C++——隐式重写

与Java和C++不同，C#应用于派生类的override关键字是必需的。C#不允许隐式重写。为了重写方法，基类和派生类成员必须匹配，而且要有对应的virtual和override关键字。此外，override关键字意味着派生类的实现会替换基类的实现。

对成员进行重载，会造成“运行时”调用最深的或者说派生得最远的实现，如代码清单6-11所示。

代码清单6-11 “运行时”调用虚方法派生得最远的实现

```
public class Program
{
  public static void Main()
  {
      Contact contact;
      PdaItem item;

      contact = new Contact();
      item = contact;

      // Set the name via PdaItem variable
      item.Name = "Inigo Montoya";

      // Display that FirstName & LastName
      // properties were set.
      Console.WriteLine("{0} {1}",
          contact.FirstName, contact.LastName);
  }
```

输出6-1展示了代码清单6-11的结果。

输出6-1

```
Inigo Montoya
```

代码清单6-11调用了item.Name，而item被声明为一个PdaItem。不过，设置的仍然是contact的FirstName和LastName。这里的规则是：“运行时”遇到虚方法时，它会调用虚成员派生得最远的重写。在本例中，代码实例化一个Contact并调用Contact.Name，因为Contact包含了Name派生得最远的实现。

创建类时必须谨慎选择是否允许重写方法，因为控制不了派生的实现。虚方法不应包含关键代码，因为如果派生类重写了它，那些代码就永远得不到调用。除此之外，将方

法从虚更改为非虚，可能破坏重写了该方法的派生类。由于会破坏代码，所以要避免这样做，尤其是那些准备由第三方使用的程序集。

代码清单6-12包含虚方法Run()。Controller的程序员在调用Run()时，假如粗心地以为关键的Start()和Stop()方法无论如何都会被调用，就会出问题。

代码清单6-12 粗心地依赖于虚方法的实现

```
public class Controller
{
  public void Start()
  {
      // Critical code
  }
  public virtual void Run()
  {
      Start();
      Stop();
  }
  public void Stop()
  {
      // Critical code
  }
}
```

别的开发者在重写Run()时，完全可能选择不调用关键性的Start()和Stop()方法。为了强制调用Start()和Stop()，Controller的程序员应该像代码清单6-13那样定义类。

代码清单6-13 强制Run()语义

```
public class Controller
{
    public void Start()
    {
        // Critical code
    }

    private void Run()
    {
        Start();
        InternalRun();
        Stop();
    }

    protected virtual void InternalRun()
    {
        // Default implementation
    }

    public void Stop()
    {
        // Critical code
    }
}
```

在新的代码清单中，Controller的程序员阻止用户错误地调用InternalRun()，因为它是protected的。另一方面，将Run()声明为private，可以保证Start()和Stop()被正确调用。而通过在派生类中重写InternalRun()，仍然可以修改默认的Controller执行方式。

虚方法只提供默认实现，这种实现可由派生类完全重写。然而，由于继承设计的复杂性，所以请事先想好是否需要虚方法。

语言对比：C++——构造期间对方法调用的调度

在C++中，构造期间不调度虚方法。相反，在构造期间，类型与基类型关联，而不是与派生类型关联，虚方法调用的是基类的实现。C#则相反，会将虚方法调用调度给派生得最远的类型。这是为了与以下设计原则保持一致："总是调用派生得最远的虚成员，即使派生的构造器尚未完全执行完毕"。但无论如何，在C#中应尽量避免出现这种情况[①]。

最后要说的是，只有实例成员才可以是virtual的。CLR根据具体化的类型（在实例化期间指定）来判断将虚方法调用调度到哪里。所以static virtual方法毫无意义，编译器也不允许。

6.2.2 new修饰符

如果重写方法没有使用override关键字，编译器会报告警告消息，如输出6-2或者输出6-3所示。

输出6-2

```
warning CS0114: '<derived method name>' hides inherited member
'<base method name>'. To make the current member override that
implementation, add the override keyword. Otherwise add the new
keyword.
```

输出6-3

```
warning CS0108: The keyword new is required on '<derived property
name>' because it hides inherited member '<base property name>'
```

一个明显的解决方案是添加override修饰符（假定基类成员是virtual的）。然而，正如警告文本指出的那样，还可以使用new修饰符。请思考表6-1总结的情形——这种情形称为**脆弱的基类**（brittle base class或者fragile base class）。

表6-1 为什么使用new修饰符

活 动	代 码
程序员A定义了Person类，其中包含属性FirstName和LastName	public class Person { public string FirstName { get; set; } public string LastName { get; set; } }

① 说了这么多，意思就是不要在构造器中调用会影响所构造对象的任何虚方法。原因是假如这个虚方法在当前要实例化的类型的派生类型中进行了重写，就会调用重写的实现。但在继承层次结构中，字段尚未完全初始化。所以，调用虚方法将导致无法预测的行为。——译者注

续表

活 动	代 码
程序员B从Person派生出Contact，并添加了额外的属性Name。除此之外，还定义了Program类，其Main()方法会实例化一个Contact，对Name赋值，然后输出姓名	public class Contact : Person { public string Name { get { return FirstName + " " + LastName; } set { string[] names = value.Split(' '); // Error handling not shown. FirstName = names[0]; LastName = names[1]; } } }
不久，程序员A自己也添加了Name属性，但不是将取值方法实现成FirstName + " " + LastName，而是实现成LastName + ", " + FirstName。除此之外，没有将属性定义为virtual，而且在DisplayName()方法中使用了该属性	// ... public class Person { public string Name { get { return LastName + ", " + FirstName; } set { string[] names = value.Split(", "); // Error handling not shown. LastName = names[0]; FirstName = names[1]; } } public static void Display(Person person) { // Display <LastName>, <FirstName> Console.WriteLine(person.Name); } }

Person.Name非虚表明程序员A希望Display()方法总是使用Person的实现，即便传给它的数据类型是Person的派生类型Contact。然而，程序员B希望在变量数据类型为Contact的任何情况下，都使用Contact.Name（程序员B其实并不知道Person.Name，因为最开始并不存在Person.Name属性）。为了允许添加Person.Name，同时不破坏两个程序员预期的行为，你不能假定该属性是virtual的。除此之外，由于C#要求重写成员显式使用override修饰符，所以必须采用其他某种形式的语法，确保在基类中添加的成员不会造成派生类编译失败。

这种语义要用new修饰符实现，它在基类面前隐藏了派生类重新声明的成员。这时不是调用派生得最远的成员。相反，是搜索继承链，找到使用new修饰符的那个成员之前的成员，然后调用该成员。如果继承链中仅包含两个类，就会使用基类的成员，感觉就像是派生类没有重写那个成员（如果派生实现重新声明了基类成员）。虽然编译器会显示如输出6-2或输出6-3所示的警告，但假如既没有指定override，也没有指定new，它就默认为new，从而维持了版本的安全性。

来看看代码清单6-14的例子，输出6-4展示了结果。

代码清单6-14 override与new修饰符的对比

```
public class Program
{
  public class BaseClass
  {
      public void DisplayName()
      {
          Console.WriteLine("BaseClass");
      }
  }

    public class DerivedClass : BaseClass
    {
        // Compiler WARNING: DisplayName() hides inherited
        // member. Use the new keyword if hiding was intended.
        public virtual void DisplayName()
        {
            Console.WriteLine("DerivedClass");
        }
    }

public class SubDerivedClass : DerivedClass
{
    public override void DisplayName()
    {
        Console.WriteLine("SubDerivedClass");
    }
}

public class SuperSubDerivedClass : SubDerivedClass
{
    public new void DisplayName()
    {
        Console.WriteLine("SuperSubDerivedClass");
    }
}

public static void Main()
{
    SuperSubDerivedClass superSubDerivedClass
        = new SuperSubDerivedClass();

    SubDerivedClass subDerivedClass = superSubDerivedClass;
    DerivedClass derivedClass = superSubDerivedClass;
    BaseClass baseClass = superSubDerivedClass;

    superSubDerivedClass.DisplayName();
    subDerivedClass.DisplayName();
```

```
        derivedClass.DisplayName();
        baseClass.DisplayName();
    }
}
```

输出6-4

```
SuperSubDerivedClass
SubDerivedClass
SubDerivedClass
BaseClass
```

之所以会得到输出6-4的结果，是由于以下几个原因。

- SuperSubDerivedClass.DisplayName()显示SuperSubDerivedClass，因为没有派生类，所以没有重载。
- SubDerivedClass.DisplayName()是重写了基类虚成员的派生得最远的成员。使用了new修饰符的SuperSubDerivedClass.DisplayName()被隐藏。
- DerivedClass.DisplayName()是虚方法，而SubDerivedClass.DisplayName()是重写了它的派生得最远的成员。和前面一样，使用了new修饰符的SuperSubDerivedClass.DisplayName()被隐藏。
- BaseClass.DisplayName()没有重新声明任何基类成员，而且非虚。所以，它会被直接调用。

就CIL来说，new修饰符对编译器生成的代码没有任何影响。然而，一个“新”方法会生成方法的newslot元数据特性。从C#的角度看，它唯一的作用就是移除编译器警告。

6.2.3 sealed修饰符

为类使用sealed修饰符是禁止从该类继承。类似地，虚成员也可密封，如代码清单6-15所示。这会禁止子类重写继承链中高层基类的虚成员。例如，假定子类B重写了基类A的一个成员，并希望禁止子类B的派生类继续重写这个成员，就可考虑使用sealed修饰符。

代码清单6-15 密封成员

```
class A
{
  public virtual void Method()
  {
  }
}
class B : A
{
  public override sealed void Method()
  {
  }
}

class C : B
{
  // ERROR:  Cannot override sealed members
  // public override void Method()
  // {
  // }
}
```

在这个例子中，为B类的Method()声明使用sealed修饰符，会禁止C重写Method()。

一般很少将整个类标记为密封。除非迫切需要这种限制。事实上，人们越来越倾向于将类型设置成非密封的，单元测试在这方面显得更突出，因为单元测试需要支持创建模拟制对象（mock object，也称为测试替身或者写作test double）来代替真正的实现。一种可能的密封整个类的情形是，对单独虚成员进行密封的开销高于将类设置为非密封的而带来的收益。但是，一般都倾向于对单独的成员进行更有目的性的密封，这可能是因为可以依赖基类的实现来获得正确的行为。

6.2.4 base成员

重写成员时，开发者经常需要调用该成员的基类版本，如代码清单6-16所示。

代码清单6-16 访问基类成员

```
public class Address
{
    public string StreetAddress;
    public string City;
    public string State;
    public string Zip;

    public override string ToString()
    {
        return string.Format("{0}" + Environment.NewLine +
            "{1}, {2}  {3}",
            StreetAddress, City, State, Zip);
    }
}

public class InternationalAddress : Address
{
    public string Country;

    public override string ToString()
    {
        return base.ToString() + Environment.NewLine +
            Country;
    }
}
```

在代码清单6-16中，InternationalAddress从Address继承，并实现了ToString()。为了调用基类的实现，要使用base关键字。它的语法几乎和this一样，包括支持将base作为构造器的一部分使用（稍后还会详述）。

附带说明的是，即使是在Address.ToString()实现中，也需要使用override修饰符，因为ToString()是object的成员。用override修饰的任何成员都自动成为虚成员，其他子类能进一步“特化”它的实现。

> **注意**
>
> 用override修饰的任何方法都自动成为虚方法。只能对基类的虚方法进行重写，所以重写获得的方法也是虚方法。

6.2.5 构造器

实例化一个派生类时，“运行时”首先调用基类的构造器，以避免绕过对基类的初始化。然而，假如基类没有可访问的（非私有的）默认构造器，就不清楚如何构造基类，C#编译器会报告一个错误。

为了避免因为缺少可访问的默认构造器而造成错误，程序员需要在派生类构造器的头部显式指定要运行哪一个基类构造器，如代码清单6-17所示。

代码清单6-17 指定要调用的基类构造器

```
public class PdaItem
{
  public PdaItem(string name)
  {
      Name = name;
  }

  // ...
}
```

```
public class Contact : PdaItem
{
  public Contact(string name) :
      base(name)
  {
      Name = name;
  }

  public string Name { get; set; }
  // ...
}
```

通过在代码中明确指定基类构造器，“运行时”就知道在调用派生类构造器之前要调用哪一个基类构造器。

6.3 抽象类

在前面的许多继承例子中，都定义了一个名为PdaItem的类，它定义了在Contact和Appointment等派生类中通用的方法和属性。然而，PdaItem本身不适合实例化。PdaItem的实例是没有意义的。只有作为基类，在从其派生的一系列数据类型之间共享默认的方法实现，才是PdaItem类真正的意义。这意味着PdaItem应该被设计成**抽象类**。抽象类是仅供派生的类。无法实例化抽象类，只能实例化从它派生的类。不抽象、可直接实例化的类称为**具体类**。

初学者主题：抽象类

抽象类代表抽象的实体。其**抽象成员**定义了从抽象实体派生的对象应包含什么，但这种成员不包含实现。通常，抽象类中的大多数功能都没有实现。一个类要从抽象类成功地派生，必须为抽象基类中的抽象方法提供具体的实现。

要定义抽象类，C#要求为类定义添加abstract修饰符，如代码清单6-18所示。

代码清单6-18 定义抽象类

```
// Define an abstract class
public abstract class PdaItem
{
  public PdaItem(string name)
  {
      Name = name;
  }

  public virtual string Name { get; set; }
}
```

```
public class Program
{
  public static void Main()
  {
      PdaItem item;
      // ERROR:  Cannot create an instance of the abstract class
      // item = new PdaItem("Inigo Montoya");
  }
}
```

不可实例化只是抽象类的一个较次要的特征。其主要特征是它包含**抽象成员**。抽象成员是没有实现的方法或属性，其作用是强制所有派生类提供实现。

来看看代码清单6-19的例子。

代码清单6-19 定义抽象成员

```
// Define an abstract class
public abstract class PdaItem
{
  public PdaItem(string name)
  {
      Name = name;
  }

  public virtual string Name { get; set; }
  public abstract string GetSummary();
}
```

```
public class Contact : PdaItem
{
    public override string Name
    {
        get
        {
            return FirstName + " " + LastName;
        }

        set
        {
            string[] names = value.Split(' ');
            // Error handling not shown.
            FirstName = names[0];
            LastName = names[1];
        }
```

```
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }

    public override string GetSummary()
    {
        return string.Format(
            "FirstName: {0}\n"
            + "LastName: {1}\n"
            + "Address: {2}", FirstName, LastName, Address);
    }

    // ...
}

public class Appointment : PdaItem
{
    public Appointment(string name) :
        base(name)
    {
        Name = name;
    }

    public DateTime StartDateTime  { get; set; }
    public DateTime EndDateTime  { get; set; }
    public string Location  { get; set; }

    // ...

    public override string GetSummary()
    {
        return string.Format(
            "Subject: {0}" + Environment.NewLine
            + "Start: {1}"  + Environment.NewLine
            + "End: {2}" + Environment.NewLine
            + "Location: {3}",
            Name, StartDateTime, EndDateTime, Location);
    }
}
```

代码清单6-19将GetSummary()定义为抽象成员，所以它不包含任何实现。随即，代码在Contact中重写它，并提供具体的实现。由于抽象成员应当被重写，所以自动成为虚成员（但不能用virtual关键字显式地这样声明）。除此之外，抽象成员不能声明为私有，否则派生类看不见它们。

开发具有良好设计的对象层次结构殊为不易。所以在编程抽象类型时，一定要自己实现至少一个（最好多个）从抽象类型派生的具体类型，以检验自己的设计。

> **注意**
>
> 抽象成员必须被重写，因此会自动成为虚成员，但不能用virtual关键字显式声明。

语言对比：C++——纯虚函数

C++使用神秘的“=0”表示法来定义抽象函数。这些函数在C++中称为纯虚函数。与C#相反，C++不要求类本身有任何特殊的声明。和C#的抽象类修饰符abstract不同，当C++类包含纯虚函数时，不需要对该类的声明进行任何特殊处理。

假如在Contact中没有提供GetSummary()的实现，编译器会报错。

注意

通过声明抽象成员，抽象类的编程者清楚地指出：为了建立具体类与抽象基类（本例是PdaItem）之间的“属于”关系，派生类必须实现抽象成员——抽象类为这种成员提供不了合适的默认实现。

初学者主题：多态性

相同签名的成员在不同的类中有不同的实现，这称为**多态性**（polymorphism）。英语“poly”代表“多”，“morph”代表“形态”，多态性是指同一个签名可以有多个实现。同一个签名不能在一个类中多次使用，所以该签名的每个实现都必然包含在不同的类中。

多态性的基本设计思想是：只有对象自己才知道如何最好地执行特定操作，通过规定调用这些操作的通用方式，多态性还促进了代码重用，因为通用的东西不必重复编码。例如，假定有多种类型的文档，每种文档都知道具体如何执行自己这种文档的Print()操作，那么不是定义单个Print()方法，在其中包含switch语句来处理每种文档类型的特殊打印逻辑，而是利用多态性调用与想要打印的文档类型对应的Print()方法。例如，为字处理文档类调用Print()，会根据字处理文档的特点进行打印。相反，为图形文档类调用同一个方法，会根据图形文档的特点进行打印。无论如何，对于任意文档类型，打印它唯一要做的就是调用Print()，其他不用考虑。

将具体打印逻辑从switch语句中移除有利于维护。首先，具体的实现位于每种文档类型的类的上下文中，而不是位于另外一个较远的地方，这符合封装原则。其次，以后添加新的文档类型不需要更改switch语句。相反，唯一要做的就是在新的文档类型中实现Print()签名。

抽象成员是实现多态性的一个手段。基类指定方法的签名，而派生类提供具体的实现，如代码清单6-20所示。

代码清单6-20 利用多态性来列出PdaItem

```
public class Program
{
  public static void Main()
  {
      PdaItem[] pda = new PdaItem[3];
```

```
        Contact contact = new Contact("Sherlock Holmes");
        contact.Address = "221B Baker Street, London, England";
        pda[0] = contact;

        Appointment appointment =
            new Appointment("Soccer tournament");
        appointment.StartDateTime = new DateTime(2008, 7, 18);
        appointment.EndDateTime = new DateTime(2008, 7, 19);
        appointment.Location = "Estádio da Machava";
        pda[1] = appointment;

        contact = new Contact("Hercule Poirot");
        contact.Address =
            "Apt 56B, Whitehaven Mansions, Sandhurst Sq, London";
        pda[2] = contact;

        List(pda);
    }

    public static void List(PdaItem[] items)
    {
        // Implemented using polymorphism. The derived
        // type knows the specifics of implementing
        // GetSummary().
        foreach (PdaItem item in items)
        {
            Console.WriteLine("________");
            Console.WriteLine(item.GetSummary());
        }
    }
}
```

代码清单6-20的结果如输出6-5所示。

输出6-5

```
________
FirstName: Sherlock
LastName: Holmes
Address: 221B Baker Street, London, England

________
Subject: Soccer tournament
Start: 7/18/2008 12:00:00 AM
End: 7/19/2008 12:00:00 AM
Location: Estádio da Machava

________
FirstName: Hercule
LastName: Poirot
Address: Apt 56B, Whitehaven Mansions, Sandhurst Sq, London
```

这样就可以调用基类的方法，但方法具体由派生类来实现。

6.4 所有类都从System.Object派生

任何对象，不管是自定义的类，还是系统内建的类，都定义好了如表6-2所示的方法。

表6-2 System.Object的成员

方 法 名	说 明
public virtual bool Equals(object o)	如果作为参数提供的对象和当前对象实例包含相同的值，就返回true
public virtual int GetHashCode()	返回对象值的散列码。它对于HashTable这样的集合非常有用
public Type GetType()	返回与对象实例的类型对应的System.Type类型的一个对象
public static bool ReferenceEquals(object a, object b)	如果两个参数引用同一个对象就返回true
public virtual string ToString()	返回对象实例的字符串表示
public virtual void Finalize()	析构器的一个别名，通知对象准备终结。C#禁止直接调用这个方法
protected object MemberwiseClone()	创建当前对象的浅表副本。引用会被复制，但被引用类型中的数据不会被复制

所有这些方法都通过继承为所有对象提供，所有对象最终都从object派生（不管是直接派生还是通过继承链派生）。即使字面值也支持这些方法。所以，下面这种看起来颇为奇怪的代码实际是合法的：

```
Console.WriteLine( 42.ToString() );
```

即使类定义没有显式地指明自己从object派生，也肯定是从object派生的。所以，在代码清单6-21中，PdaItem的两个声明会产生完全一致的CIL。

代码清单6-21 如果不显式指定自己从哪里派生，就隐式地派生自System.Object

```
public class PdaItem
{
  // ...
}
```

```
public class PdaItem : object
{
  // ...
}
```

假如object的默认实现不够用，程序员可以重写表6-2中总结的3个virtual方法。第9章将详细介绍如何做。

6.5 使用is操作符验证基础类型

由于C#允许在继承链中向下转型，因此有时需要在转换前判断基础类型是什么。除此之外，在没有实现多态性的情况下，一些要依赖特定类型的行为也可能要事先确定类型。C#提供了is操作符来判断基础类型，如代码清单6-22所示。

代码清单6-22 用is操作符判断基础类型

```
public static void Save(object data)
{
    if (data is string)
    {
        data = Encrypt((string) data);
    }

    // ...
}
```

在代码清单6-22中，只有在基础类型是string的前提下才对数据进行加密。这比直接加密好，因为许多基础类型并非string的类型也支持转型为string。

虽然is操作符比较有用，但仍然应该优先考虑多态性。多态性允许将一个行为扩展到其他数据类型，同时不必修改对行为进行定义的实现。例如，从一个通用的基类型派生，然后将那个类型作为Save()方法的参数使用，就可避免显式检查string，并允许其他数据类型从同一个基类派生，以提供数据保存期间的加密支持。

6.6 使用as操作符进行转换

is操作符的优点在于，它允许验证一个数据项是否属于特定类型。as操作符则更进一步，它会像一次转型所做的那样，尝试将对象转换为特定数据类型。但和转型不同的是，如果对象不能转换，as操作符会返回null。这一点相当重要，因为它避免了可能因为转型而造成的异常。代码清单6-23演示了如何使用as操作符。

代码清单6-23 使用as操作符进行数据类型转换

```
object Print(IDocument document)
{
  if(thing != null)
  {
       // Print document...
  }
  else
  {
  }
}

static void Main()
{
  object data;

  // ...

  Print(data as Document);
}
```

使用as操作符可避免用额外的try-catch代码处理转换无效的情况，因为as操作符提供了尝试执行转型但转型失败后不引发异常的一个办法。

is操作符相较于as操作符的一个优点是后者不能成功判断基础类型。后者能在继承

链中向上或向下转型，也支持提供了转型操作符的类型。所以，as不能判断基础类型而is能[①]。

6.7 小结

本章讨论了如何从一个类派生，并添加附加的方法和属性，从而对那个类进行“特化”。讨论了如何使用private和protected访问修饰符控制封装级别。

本章还详细讨论了如何重写基类实现，以及如何使用new修饰符来隐藏基类的实现。为了控制重写，C#提供了virtual修饰符，它告诉派生类的程序员需要重写哪些成员。要完全禁止派生，需要为类使用sealed修饰符。类似地，为成员使用sealed修饰符，会禁止子类继续重写该成员。

本章最后简单讨论了所有类型都是从object派生。第9章将进一步讨论这方面的问题。届时，会讲解object的3个virtual方法，会重点描述影响重载过程的规则和准则。然而，在学习那些知识之前，首先要掌握在面向对象编程的基础上发展起来的另一种编程模式——接口。这将是第7章的主题。

---

① 关于is操作符，原书讲得并不深入，也许存在一些不好理解的地方。经过与原书作者Mark沟通，他为中文版的读者提供了以下更加清晰的说明：

我在这里的意思是，is操作符能做的，并非仅仅是检查数据能成功转型为一个string。换言之，一个类可能实现了向string的强制转换，所以能成功转型为一个string。然而，is操作符不仅检查转型是否成功，还会检查底层对象本身是否真的是一个string。虽然实现了向string转型的一个对象能成功转型为一个string，但是，为那个对象使用is操作符会返回false。如下例所示：

```
class Thing
{
  public static explicit operator string(Thing thing)
  {
    return thing.ToString();
  }
  public override string ToString()
  {
    return "Thing";
  }
}
public class One
{
  static void Main(string[] args)
  {

    object o;
    string s = "text";
    Thing t = new Thing();
    o = t;
    Trace.Assert(!(t is string));
    Trace.Assert( ((string)t) == "Thing");
    o = s;
    Trace.Assert( ((string)o) == "text" );
    Trace.Assert( o is string );
  }
}
```

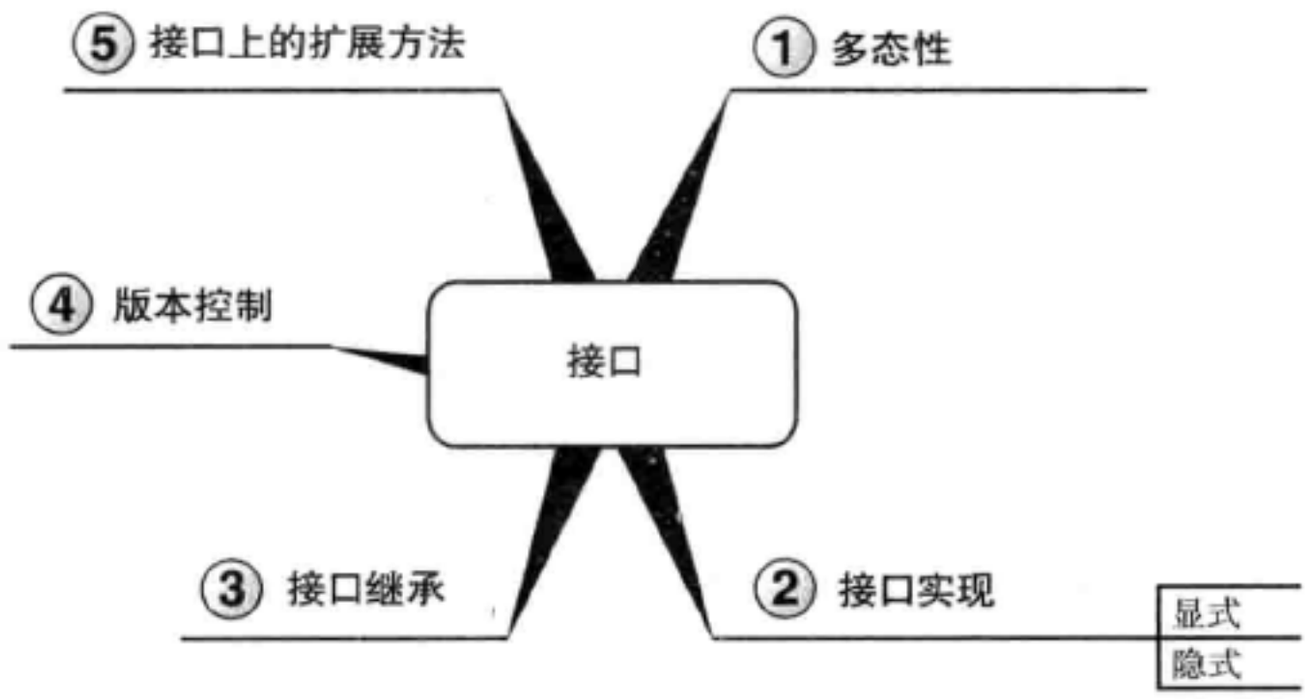
——译者注

第7章

接　口

并非只能通过继承使用多态性（像第6章讨论的那样），还能通过接口使用。和抽象类不同，接口不包含任何实现。但和抽象类相似，接口也定义了一组成员，调用者可认为这些成员已经实现。

类型通过实现接口来定义它的功能。接口实现关系是一种“能做”（can do）关系：类型“能做”接口所规定的事情。在“实现接口的类型”和“使用接口的代码”之间，接口订立了“契约[①]”。实现接口的类型必须使用接口要求的签名来定义方法。本章讨论了接口的实现和使用。



7.1　接口概述

初学者主题：为什么需要接口

接口是非常有用的，因为和抽象类不同，接口能将实现细节和提供的服务完全隔

① MSDN文档称为“协定”。——译者注

离开。举一个现实生活中的例子，接口就像是电源插座。电力如何被输送到插座是实现细节：可能是煤电、核电或太阳能发电；发电机可能在隔壁，也可能在很远的地方，等等。插座订立了“契约”。它以特定频率提供特定的电压，并要求使用该接口的电器提供兼容的插头。电器不必关心电力是如何输送到插座的，它需要关注的只是提供与之兼容的插头。

来看看下面这个例子：目前有许多文件压缩格式，包括.zip、.7-zip、.cab、.lha、.tar、.tar.gz、.tar. bz2、.bh、.rar、.arj、.arc、.ace、.zoo、.gz、.bzip2、.xxe、.mime、.uue以及.yenc等。假如为每种压缩格式都单独创建一个类，那么每个压缩实现都会有不同的方法签名，无法提供标准调用规范。虽然方法可在基类中声明为抽象成员，但假如都从一个共有的基类派生，会用掉唯一的基类机会（C#只允许单继承）。不同的压缩实现没有什么通用的代码可以放到基类中，从而使基类实现变得毫无意义。这里的重点是，基类除了允许共享成员签名，还允许共享实现。但是，接口只允许共享成员签名，不允许共享实现。

所以，在这种情况下，不是共享一个共有的基类，而是每个压缩类都实现一个共有的接口。接口订立了契约，类必须履行这个契约，才能同实现该接口的其他类进行交互。虽然存在着多种压缩算法，但假如它们都实现了IFileCompression接口以及该接口的Compress()和Uncompress()方法，那么在需要压缩和解压缩的时候，只需执行到IFileCompression接口的一次转型，然后调用它的成员方法即可，根本不用关心具体是什么类在实现哪些方法。这正是多态性的一个反映，因为每个压缩类都有相同的方法签名，但那个签名的具体实现是不同的。

代码清单7-1展示了示例接口IFileCompression。

代码清单7-1 定义接口

```
interface IFileCompression
{
  void Compress(string targetFileName, string[] fileList);
  void Uncompress(
      string compressedFileName, string expandDirectoryName);
}
```

IFileCompression定义了一个类为了同其他压缩类协作而必须实现的方法。定义好接口之后，调用者就可以在不同的实现之间自由切换，而不需要修改调用代码——只要每个压缩类都实现了IFileCompression接口。

接口的关键特点是既不包含实现，也不包含数据。注意其中的方法声明，它用一个分号取代了大括号。字段（数据）不能在接口声明中出现。如果接口要求派生类包含特定数据，会声明属性而不是字段。由于没有属性的任何实现可以作为接口声明的一部分，所以属性不引用支持字段。

接口声明的成员描述了在实现该接口的类型中必须能够访问的成员。而所有非公共成员的目的是阻止其他代码访问成员。所以，C#不允许为接口成员使用访问修饰符。所有成员都自动定义为公共成员。

规范

接口名称要使用Pascal大小写风格，并以“I”作为前缀。

7.2 通过接口实现多态性

来看看代码清单7-2的例子。IListable接口定义了一个类为了让ConsoleListControl类显示它而必须支持的成员。任何类只要实现了IListable接口，就可以使用ConsoleListControl显示它自身。IListable接口要求只读属性ColumnValues。

代码清单7-2 实现和使用接口

```
interface IListable
{
  // Return the value of each column in the row.
  string[] ColumnValues
  {
      get;
  }
}
```

```
public abstract class PdaItem
{
  public PdaItem(string name)
  {
      Name = name;
  }

  public virtual string Name{get;set;}
}
```

```
class Contact : PdaItem, IListable
{
    public Contact(string firstName, string lastName,
        string address, string phone) : base(null)
    {
        FirstName = firstName;
        LastName = lastName;
        Address = address;
        Phone = phone;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public string Phone { get; set; }

    public string[] ColumnValues
    {
        get
        {
            return new string[]
            {
                FirstName,
```