第3章

C#面向对象程序设计

第2章介绍了C#的语法和基础知识，据此已经可以写出一些控制台应用程序了，但是，要了解C#语言的强大功能，还需要使用面向对象编程(Object-Oriented Programming，OOP)技术。实际上，前面的例子已经在使用这些技术，但没有重点讲述。

本章先探讨OOP的原理，包括OOP的基础知识和与OOP相关的术语。接着学习如何在C#中定义类，包括基本的类定义语法、用于确定类可访问性的关键字以及接口的定义。然后讨论如何定义类成员，包括如何定义字段、属性和方法等成员。最后说明一些高级技术，包括集合、运算符重载、高级转换、深度复制和定制异常。

3.1 面向对象编程简介

3.1.1 什么是面向对象编程

面向对象编程代表了一种全新的程序设计思路，与传统的面向过程开发方法不同，面向对象的程序设计和问题求解更符合人们的思维习惯。

前面介绍的编程方法都是面向过程的程序设计方法，这种方法常常会导致所谓的单一应用程序，即所有的功能都包含在几个代码模块中(常常是一个代码模块)，适合解决比较小的简单问题。而OOP技术则按照现实世界的特点来管理复杂的事物，把它们抽象为对象，具有自己的状态和行为，通过对消息的反应来完成一定的任务。这种编程方法提供了非常强大的多样性，大大增加了代码的重用机会，增加了程序开发的速度；同时降低了维护负担，将具备独立性特制的程序代码包装起来，修改部分程序代码时不至于会影响到程序的其他部分。

1. 对象

什么是对象？实际上，现实世界就是由各种对象组成的，如人、汽车、动物、植物等。复杂的对象可以由简单的对象组成。对象都具有各自的属性，如形状、颜色、重量等；对外界都呈现出各自的行为，如人可以走路、说话、唱歌；汽车可以启动、加速、减速、刹车、停止等。

在OOP中，对象就是变量和相关的方法的集合。其中变量表明对象的属性，方法表明对象所具有的行为。一个对象的变量构成了这个对象的核心，包围在它外面的方法使这个

对象和其他对象分离开来。例如：我们可以把汽车抽象为一个对象，用变量来表示它当前的状态，如速度、油量、型号、所处的位置等，它的行为则为上面提到的加速、刹车、换档等。操作汽车时，不用去考虑汽车内部各个零件如何运作的细节，而只需根据汽车可能的行为使用相应的方法即可。实际上，面向对象的程序设计实现了对象的封装，使我们不必关心对象的行为是如何实现的这样一些细节。通过对对象的封装，实现了模块化和信息隐藏。有利于程序的可移植性和安全性，同时也利于对复杂对象的管理。

简单地说，对象非常类似于本书前面讨论的结构类型。略为复杂的对象可能不包含任何数据，而是只包含函数，表示一个过程。

2. 类

在研究对象时主要考虑对象的属性和行为，有些不同的对象会呈现相同或相似的属性和行为，如轿车、卡车、面包车。通常将属性及行为相同或相似的对象归为一类。类可以看成是对象的抽象，代表了此类对象所具有的共同属性和行为。典型的类是“人类”，表明人的共同性质。比如我们可以定义一个汽车类来描述所有汽车的共性。通过类定义人们可以实现代码的复用。不用去描述每一个对象（如某辆汽车），而是通过创建类（如汽车类）的一个实例来创建该类的一个对象，这样大大简化了软件的设计。

类是对一组具有相同特征的对象的抽象描述，所有这些对象都是这个类的实例。在 C# 中，类是一种数据类型，而对象是该类型的变量，变量名即是某个具体对象的标识名。

3. 属性和字段

通过属性和字段可以访问对象中包含的数据。对象数据可以区分不同的对象，因为同一个类的不同对象可能在属性和字段中存储了不同的值。包含在对象中的不同数据统称为对象的状态。

假定一个对象类表示一杯咖啡，叫做 CupOfCoffee。在实例化这个类（即创建这个类的对象）时，必须提供对于类有意义的状态。此时可以使用属性和字段，让代码能通过该对象来设置要使用的咖啡品牌，咖啡中是否加牛奶或方糖，咖啡是否即溶等。给定的咖啡对象就有一个指定的状态，例如 Columbian filter coffee with milk and two sugars。

可以把信息存储在字段和属性中，作为 string 变量、int 变量等。但是，属性与字段是不同的，属性不能直接访问数据。一般情况下，在访问状态时最好提供属性，而不是字段，因为这样可以更好地控制整个过程，而使用它们的语法是相同的。

对属性的读写访问也可以由对象来明确定义。某些属性是只读的，只能查看它们的值，而不能改变它们（至少不能直接改变）。还可以有只写的属性，其操作方式类似。

除了对属性的读写访问外，还可以为字段和属性指定另一种访问许可，这种可访问性确定了什么代码可以访问这些成员，它们是可用于所有的代码（公共），还是只能用于类中的代码（私有），或者更复杂的模式。常见的情况是把字段设置为私有，通过公共属性访问它们。

例如，CupOfCoffee 类可以定义 5 个成员：Type、isInstant、Milk、Sugar、Description 等。

4. 方法

对象的所有行为都可以用方法来描述，在 C# 中，方法就是对象中的函数。

方法用于访问对象的功能，与字段和属性一样：方法可以是公共的或私有的，按照需要限制外部代码的访问。它们常常使用对象状态——访问私有成员。例如，CupOfCoffee 类定义了一个方法 AddSugar()来增加方糖数属性。

实际上，C＃中的所有东西都是对象。控制台应用程序中的 Main()函数就是类的一个方法。前面介绍的每个变量类型都是一个类。前面使用的每个命令都是一个属性或方法。句点字符“.”把对象实例名和属性或方法名分隔开来。

5. 对象的生命周期

每个对象都有一个明确定义的生命周期，即从使用类定义开始一直到删除它为止。在对象的生命周期中，除了“正在使用”的正常状态之外，还有两个重要的阶段。

- 构造阶段。对象最初进行实例化的时期。这个初始化过程称为构造阶段，由构造函数完成。
- 析构阶段。在删除一个对象时，常常需要执行一些清理工作，例如释放内存，由析构函数完成。

(1) 构造函数

所有的对象都有一个默认的构造函数，该函数没有参数，与类本身有相同的名称。一个类定义可以包含几个构造函数，它们有不同的签名，代码可以使用这些签名实例化对象。带有参数的构造函数通常用于给存储在对象中的数据提供初始值。

在 C＃中，构造函数用 new 关键字来调用。例如，可以用下面的方式实例化一个 CupOfCoffee 对象：

```
CupOfCoffee myCup = new CupOfCoffee();
```

对象还可以用非默认的构造函数来创建。与默认的构造函数一样，非默认的构造函数与类同名，但它们还带有参数，例如：

```
CupOfCoffee myCup = new CupOfCoffee("Blue Mountain");
```

构造函数与字段、属性和方法一样，可以是公共或私有的。在类外部的代码不能使用私有构造函数实例化对象，而必须使用公共构造函数。一些类没有公共的构造函数，外部的代码就不可能实例化它们。

(2) 析构函数

析构函数用于清理对象。一般情况下，不需要提供解构方法的代码，而是由默认的析构函数执行操作。但是，如果在删除对象实例前，需要完成一些重要的操作，就应提供特定的析构函数。

属性、方法和字段等成员是对象实例所特有的，即改变一个对象实例的这些成员不影响其他的实例中的这些成员。除此之外，还有一种静态成员(也称为共享成员)，例如静态方法、静态属性或静态字段。静态成员可以在类的实例之间共享，所以它们可以看作是类的全局对象。静态属性和静态字段可以访问独立于任何对象实例的数据，静态方法可以执行与对象类型相关、但不是特定实例的命令，在使用静态成员时，甚至不需要实例化类型的对象。例如，前面使用的 Console. WriteLine()方法就是静态的。

3.1.2 OOP 技术

前面介绍了一些基础知识，下面讨论 OOP 中的一些技术，包括抽象与接口、继承、多态性、重载等。

1. 抽象与接口

抽象化是为了要降低程序版本更新后，在维护方面的负担，使得功能的提供者和功能的用户分开，各自独立，彼此不受影响。

为了达到抽象化的目的，需要在功能提供者与功能使用者之间提供一个共同的规范，功能提供者与功能使用者都要按照这个规范来提供、使用这些功能。这个共用的规范就是接口，接口定义了功能数量、函数名称、函数参数、参数顺序等。它是一个能声明属性、字段和方法的编程构造。它不为这些成员实现，只提供定义。接口定义了功能提供者与功能使用者之间的准则，因此只要接口不变，功能提供者就可以任意更改实现的程序代码，而不影响到使用者。

一旦定义了接口，就可以在类中实现它。这样，类就可以支持接口所指定的所有属性和成员。注意，不能实例化接口，执行过程必须在实现接口的类中实现。

在前面的咖啡范例中，可以把较一般用途的属性和方法例如 AddSugar()，Milk，Sugar 和 Instant 组合到一个接口中，称为 IhotDrink(接口的名称一般用大写字母 I 开头)。然后就可以在其他对象上使用该接口，例如 CupOfTea 类。

一个类可以支持多个接口，多个类也可以支持相同的接口。

2. 继承

继承是 OOP 最重要的特性之一。任何类都可以从另一个类继承，这就是说，这个类拥有它继承的类的所有成员。在 OOP 中，被继承(也称为派生)的类称为父类(也称为基类)。注意，C＃中的对象仅能派生于一个基类。

公共汽车、出租车、货车等都是汽车，但它们是不同的汽车，除了具有汽车的共性外，它们还具有自己的特点，如不同的操作方法，不同的用途等。这时可以把它们作为汽车的子类来实现，它们继承父类(汽车)的所有状态和行为，同时增加自己的状态和行为。通过父类和子类实现了类的层次，可以从最一般的类开始，逐步特殊化，定义一系列的子类。同时，通过继承也实现了代码的复用，使程序的复杂性线性地增长，而不是呈几何级数增长。

在继承一个基类时，成员的可访问性就成为一个重要的问题。派生类不能访问基类的私有成员，但可以访问其公共成员。不过，派生类和外部的代码都可以访问公共成员。这就是说，只使用这两个可访问性，不仅可以让一个成员被基类和派生类访问，而且也能够被外部的代码访问。为了解决这个问题，C＃提供了第三种可访问性：protected，只有派生类才能访问 protected 成员。

除了成员的保护级别外，还可以为成员定义其继承行为。基类的成员可以是虚拟的，也就是说，成员可以由继承它的类重写。派生类可以提供成员的其他执行代码。这种执行代

码不会删除原来的代码，仍可以在类中访问原来的代码，但外部代码不能访问它们。如果没有提供其他执行方式，外部代码就访问基类中成员的执行代码。虚拟成员不能是私有成员。

基类还可以定义为抽象类。抽象类不能直接实例化。要使用抽象类，必须继承这个类，抽象类可以有抽象成员，这些成员在基类中没有代码实现，所以这些执行代码必须在派生类中提供。

最后，类可以是密封的。密封的类不能用作基类，所以也没有派生类。

在 C# 中，所有的对象都有一个共同的基类 object，参见第 2 章中的相关内容。

3. 多态性

多态是面向对象程序设计的又一个特性。在面向过程的程序设计中，主要工作是编写一个个的过程或函数，这些过程和函数不能重名。例如在一个应用中，需要对数值型数据进行排序，还需要对字符型数据进行排序，虽然使用的排序方法相同，但要定义两个不同的过程（过程的名称也不同）来实现。

在面向对象程序设计中，可以利用“重名”来提高程序的抽象度和简洁性。首先来理解实际的现象，例如，“启动”是所有交通工具都具有的操作，但是不同的具体交通工具，其“启动”操作的具体实现是不同的，如汽车的启动是“发动机点火——启动引擎”、“启动”轮船时要“起锚”、气球飞艇的“启动”是“充气——解缆”。如果不允许这些功能使用相同的名字，就必须分别定义“汽车启动”、“轮船启动”、“气球飞艇启动”多个方法。这样一来，用户在使用时需要记忆很多名字，继承的优势就荡然无存了。为了解决这个问题，在面向对象的程序设计中引入了多态的机制。

多态是指一个程序中同名的不同方法共存的情况。主要通过子类对父类方法的覆盖来实现多态。这样一来，不同类的对象可以响应同名的方法来完成特定的功能，但其具体的实现方法却可以不同。例如同样的加法，把两个时间加在一起和把两个整数加在一起肯定完全不同。

通过方法覆盖，子类可以重新实现父类的某些方法，使其具有自己的特征。例如对于车类的加速方法，其子类（如赛车）中可能增加了一些新的部件来改善提高加速性能，这时可以在赛车类中覆盖父类的加速方法。覆盖隐藏了父类的方法，使子类拥有自己的具体实现，更进一步表明了与父类相比，子类所具有的特殊性。

多态性使语言具有灵活、抽象、行为共享的优势，很好地解决了应用程序函数同名问题。

注意并不是只有共享同一个父类的类才能利用多态性。只要子类和孙子类在继承层次结构中有一个相同的类，它们就可以用相同的方式利用多态性。

4. 重载

方法重载是实现多态的另一个方法。通过方法重载，一个类中可以有多个具有相同名字的方法，由传递给它们的不同个数的参数来决定使用哪种方法。例如，对于一个作图的类，它有一个 draw()方法用来画图或输出文字，可以传递给它一个字符串、一个矩形、一个圆形，甚至还可以再制定作图的初始位置、图形的颜色等。对于每一种实现，只需实现一个新的 draw()方法即可，而不需要新起一个名字，这样大大简化了方法的实现和调用，程序员

和用户不需要记住很多的方法名，只需要传入相应的参数即可。

因为类可以包含运算符如何运算的指令，所以可以把运算符用于从类实例化而来的对象。我们为重载运算符编写代码，把它们用作类定义的一部分，而该运算符作用于这个类。也可以重载运算符，以相同的方式处理不同的类，其中一个（或两个）类定义包含达到这一目的的代码。

注意：只能用这种方式重载现有的C#运算符，不能创建新的运算符。

5. 消息和事件

对象之间必须要进行交互来实现复杂的行为。例如，要汽车加速，必须发给它一个消息，告诉它进行何种动作（这里是加速）以及实现这种动作所需要的参数（这里是需要达到的速度等）。一个消息包含三个方面的内容：消息的接收者、接收对象应采用的方法、方法所需要的参数。同时，接收消息的对象在执行相应的方法后，可能会给发送消息的对象返回一些信息。如上例中，汽车的仪表上会出现已达到的速度等。

在C#中，消息处理称为事件。对象可以激活事件，作为处理的一部分。为此，需要给代码添加事件处理程序，这是一种特殊类型的函数，在事件发生时调用。还需要配置这个处理程序，以监听我们感兴趣的事件。

使用事件可以创建事件驱动的应用程序，这类应用程序很多。例如，许多基于Windows的应用程序完全依赖于事件。每个按钮单击或滚动条拖动操作都是通过事件处理实现的，其中事件是通过鼠标或键盘触发的。本章的后面将介绍事件是如何工作的。

3.2 定 义 类

本节将重点讨论如何定义类本身。首先介绍基本的类定义语法、用于确定类可访问性的关键字、指定继承的方式以及接口的定义。

3.2.1 C# 中的类定义

1. 类的定义

C#使用class关键字来定义类。其基本结构如下：

```
Class MyClass
{
  // class members
}
```

这段代码定义了一个类MyClass。定义了一个类后，就可以对该类进行实例化。在默认情况下，类声明为内部的，即只有当前代码才能访问，可以用intemal访问修饰符关键字显式指定，如下所示（但这是不必要的）：

```
internal class MyClass
```

```
{
  // class members
}
```

另外，还可以制定类是公共的，可以由其他任意代码访问。为此，需要使用关键字 public：

```
public class MyClass
{
  // class members
}
```

除了这两个访问修饰符关键字外，还可以指定类是抽象的(不能实例化，只能继承，可以有抽象成员)或密封的(sesled，不能继承)。为此，可以使用两个互斥的关键字 abstract 或 sealed。所以，抽象类必须用下述方式声明：

```
public abstract class MyClass
{
  // class members, may be abstract
}
```

密封类的声明如下所示：

```
public sealed class MyClass
{
  //class members
}
```

还可以在类定义中指定继承。C＃支持类的单一继承，即只能有一个基类，语法如下：

```
class MyClass: MyBaseClass
{
  // class members
}
```

在 C＃的类定义中，如果继承了一个抽象类，就必须执行所继承的所有抽象成员(除非派生类也是抽象的)。

编译器不允许派生类的可访问性比其基类更高。也就是说，内部类可以继承于一个公共类，但公共类不能继承于一个内部类。因此，下述代码就是不合法的：

```
internal class MyBaseClass
{
  // class members
}

public class MyClass: MyBaseClass
{
  // class members
}
```

在 C＃中，类必须派生于另一个类。如果没有指定基类，则被定义的类就继承于基类 System. Object。

除了以这种方式指定基类外，还可以指定支持的接口。如果指定了基类，它必须紧跟在

冒号的后面，之后才是指定的接口。必须使用逗号分隔基类名(如果有基类)和接口名。

例如，给 MyClass 添加一接口，如下所示：

```
class MyClass: IMyInterface
{
  // class memebrs
}
```

所有的接口成员都必须在支持该接口的类中实现，但如果不想使用给定的接口成员，可以提供一个“空”的执行方式(没有函数代码)。

下面的声明是无效的，因为基类 MyBaseClass 不是继承列表中的第一项：

```
class MyClass: IMyInterface, MyBaseClass
{
  // class members
}
```

指定基类和接口的正确方式如下：

```
class: MyBaseClass, ImyInterface
{
  // class members
}
```

可以指定多个接口，所以下面的代码是有效的：

```
public class MyClass: MyBaseClass, ImyInterface, ImySecondInterface
{
  // class members
}
```

表 3.1 是类定义时可以使用的访问修饰符组合。

表 3.1　访问修饰符

修　饰　符	含　义
none 或 internal	类只能在当前程序中被访问
public	类可以在任何地方访问
abstract 或 internal abstract	类只能在当前程序中被访问，不能实例化，只能继承
public abstract	类可以在任何地方访问，不能实例化，只能继承
sealed 或 internal sealed	类只能在当前程序中被访问，不能派生，只能实例化
public sealed	类可以在任何地方访问，不能派生，只能实例化

2. 接口的定义

接口声明的方式与声明类的方式相似，但使用的是关键字 interface，例如：

```
interface ImyInterface
{
  // interface members
}
```

访问修饰符关键字 public 和 internal 的使用方式是相同的，所以要使接口的访问是公共的，就必须使用 public 关键字：

```
public interface ImyInterface
{
  // interface members
}
```

关键字 abstract 和 sealed 不能在接口中使用，因为这两个修饰符在接口定义中是没有意义的（接口不包含执行代码，所以不能直接实例化，且必须是可以继承的）。

接口的继承也可以用与类继承的类似方式来指定。主要的区别是可以使用多个基接口，例如：

```
public interface IMyInterface: IMyBaseInterface, ImyBaseInterface2
{
  // interface members
}
```

下面看一个类定义的范例。

【例 3.1】

```
using System;
public abstract class MyBaseClass
{
}
class MyClass: MyBaseClass
{
}
public interface IMyBaseInterface
{
}
interface IMyBaseInterface2
{
}
interface ImyInterface: IMyBaseInterface, IMyBaseInterface2
{
}
sealed class MyComplexClass: MyClass,IMyInterface
{
}
class Class1
{
  static void Main(string[] args)
  {
      MyComplexClass myObj = new MyComplexClass();
      Console.WriteLine(myObj.ToString());
  }
}
```

这里的 Clsss1 不是主要类层次结构中的一部分，而是处理 Main()方法的应用程序的入口点。MyBaseClass 和 IMyBaseInterface 被定义为公共的，其他类和接口都是内部的。其中

MyComplexClass 继承 MyClass 和 IMyInterface，MyClass 继承 MyBassClass，IMyInterface 继承 IMyBaseInterface 和 IMyInterface2，而 MyBaseClass 和 IMyBaseInterface、IMyBaseInterface2 的共同的基类为 Object。Main()中的代码调用 MyComplexClass 的一个实例 myObj 的 ToString()方法。这是继承 System.Object 的一种方法，功能是把对象的类名作为一个字符串返回，该类名用所有相关的命名空间来限定。

3.2.2 Object 类

前面提到所有的.NET 类都派生于 System.Object。实际上，如果在定义类时没有指定基类，编译器就会自动假定这个类派生于 Object。其重要性在于，自己定义的所有类除了自己定义的方法和属性外，还可以访问为 Object 定义的许多公共或受保护的成员方法。在 Object 中定义的方法如表 3.2 所示。

表 3.2 Object 中的方法

方 法	访问修饰符	作 用
string ToString()	public virtual	返回对象的字符串表示。在默认情况下，这是一个类类型的限定名，但它可以被重写，以便给类类型提供合适的实现方式
int GetHashTable()	public virtual	在实现散列表时使用
bool Equals(object obj)	public virtual	把调用该方法的对象与另一个对象相比较，如果它们相等，就返回 true。以默认的执行方式进行检查，以查看对象的参数是否引用了同一对象。如果想以不同的方式来比较对象，可以重写该方法
bool Equals(object objA,object objB)	public static	这个方法比较传递给它的两个对象是否相等。如果两个对象都是空引用，这个方法会返回 true
bool ReferenceEquals (object objA,object objB)	public static	比较两个引用是否指向同一个对象
Type GetType()	public	返回对象类型的详细信息
object MemberwiseClone()	protected	通过创建一个新对象实例并复制成员来复制该对象。成员复制不会得到这些成员的新实例。新对象的任何引用类型成员都将引用与源类相同的对象，这个方法是受保护的，所以只能在类或派生的类中使用

这些方法是.NET Framework 中对象类型必须支持的基本方法，但可以从不使用它们。下面将介绍几种方法的作用。

GetType()方法：这个方法返回从 System.Type 派生的类的一个实例。在利用多态性时，GetType()是一个有用的方法，它允许根据对象的类型来执行不同的操作。联合使用 GetType()和 typeof()，就可以进行比较，如下所示：

```
if (myObj.GetType() == typeof(MyComplexClass))
{
  // myObj is an instance of the class MyComplexClass
}
```