第3章 面向对象程序设计

C#是一种面向对象的程序设计语言。面向对象编程的主要思想是将数据以及处理这些数据的相应方法封装到类中，使用类创建的实例称为对象。使用类的好处在于它有利于程序的模块化设计和开发，可以隐藏内部的实现细节，并能增强程序代码的重用性。本章介绍C#面向对象的有关知识。

3.1 面向对象编程

面向对象是一种新的程序设计方法，或者是一种新的程序设计规范，其基本思想是使用对象、类、继承、封装和消息等基本概念来进行程序设计。从现实世界中客观存在的事物（即对象）出发来构造软件系统，并且在系统构造中尽可能运用人类的自然思维方式。

3.1.1 对象和类的概念

类和对象是面向对象程序设计的核心和本质。类是对象的模板，对象是类的实例。

对象（Object）代表现实世界中可以明确标识的任何事物。例如，一个人，一张桌子，一个矩形，甚至一笔抵押贷款都可以看作是对象，对象有自己的状态和行为。在C#语言中，对象的状态用数据来描述，对象的行为用方法来描述。以矩形为例，数据有长度和宽度，方法有求面积和求周长。从更抽象的角度来说，对象是问题域或实现域中某些事物的一个抽象，它反映该事物在系统中需要保存的信息和发挥的作用；它是一组数据和有权对这些数据进行操作的一组方法的封装体。客观世界是由对象和对象之间的联系组成的。

把众多的事物归纳、划分成一些类，是人类在认识客观世界时经常采用的思维方法。分类的原则是抽象。类（Class）是具有相同属性和方法的一组对象的集合，它为属于该类的所有对象提供了统一的抽象描述，其内部包括数据和方法两个主要部分。在面向对象的编程语言中，类是一个独立的程序单位，它应该有一个类名并包括数据说明和方法说明两个主要部分。类与对象的关系就如模具和铸件的关系，类的实例化结果就是对象，而对一类对象的抽象就是类。

3.1.2 面向对象程序设计

面向对象（Object Oriented）是一种解决问题的方法或者观点，它认为自然界是由一组彼此相关并能相互通信的实体——对象组成。面向对象程序设计（Object Oriented

Programming,OOP)是使用面向对象的观点来描述现实问题,然后用计算机语言来模仿并处理问题的一种程序设计方法。

程序是以类的实例——对象组成的,对象是由数据结构和对数据结构的操作封装而成的一个整体。封装使得算法和数据结构的关系由算法对数据结构单方面的依赖变成了相互依赖的关系。一个个不同类型的对象相互作用,自底向上构建整个应用程序,它以"对象=数据结构+算法,程序=(对象+对象+…+对象)+消息"取代了"程序=算法+数据结构"的传统程序设计模式,因而引起了一场程序设计概念的革命。

在用面向对象的软件方法解决现实世界的问题时,先将物理存在的实体抽象成概念世界的抽象数据类型,这个抽象数据类型里面包括实体中与需要解决的问题相关的数据和操作;然后再用面向对象的工具,如C#语言将这个抽象数据类型用计算机逻辑表达出来,即构造计算机能够理解和处理的类;最后将类实例化就得到现实世界实体的映射——对象。在程序中对对象进行操作,就可以模拟现实世界实体上的问题并且解决之,如图3-1所示。

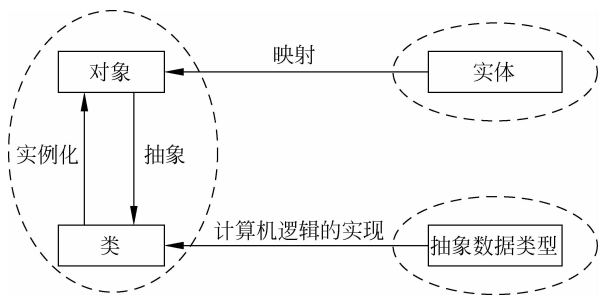

图3-1　实体、对象与类的相互关系

面向对象编程技术的关键性观念是:

(1) 将数据及对数据的操作封装在一起形成了类,类是描述相同类型的对象集合。面向对象编程就是定义类。

(2) 类作为抽象的数据类型用于创建类的对象。

(3) 程序的执行表现为一组对象之间的交互通信。对象之间通过公共接口进行通信,从而完成系统功能。对象的公共接口是该对象的应用程序编程接口,把对象的内部详细信息隐藏起来,使得对象变得抽象,将这种技术称为数据的抽象化。

3.1.3　OOP的4个基本特征

面向对象程序设计方法采用数据抽象与隐藏、层次结构体系、动态绑定等机制,提供一种模拟人类认知方式的软件建模方法,带来了系统的安全性、可扩充性、代码重用、易维护等人们期待的特性。

1. 抽象

为了能够处理客观事物,必须对对象进行抽象。抽象就是忽略一个主题中与当前目标无关的那些方面,以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题,而只是选择其中的一部分,忽略暂时不用的部分细节。在OOP中,抽象就是找出对象的本

质,抽出这一类对象的共有性质(数据和方法)并加以描述的过程。

例如,对圆这一类对象的抽象:

数据抽象——半径 radius,方法抽象——求面积 GetArea()。

对一个问题可能有不同的抽象结果,这取决于程序员看问题的角度和解决问题的需要。

2. 封装

封装就是把对象的数据和方法结合成一个独立的单位,并尽可能隐蔽对象的内部细节。

(1) 把对象的全部数据和方法结合在一起,形成一个不可分割的独立单位(即对象)。

(2) 信息隐蔽,尽可能隐蔽对象的内部细节,对外形成一个边界(或者说形成一道屏障),只保留有限的对外接口使之与外部发生联系。

在C#语言中,类是支持封装的工具。定义类时通过花括号{}封装了类的成员。使用 private 和 public 等关键字来控制对成员的访问,其中 private 修饰的成员是隐蔽的,而 public 修饰的成员是对外的公共接口。

封装的作用是:

(1) 彻底消除了传统结构方法中数据与操作分离所带来的种种问题,提高了程序的可复用性和可维护性,降低了程序员保持数据与操作相容的负担。

(2) 把对象的私有数据和公共数据分离开来,保护了私有数据,减少了可能的模块间干扰,达到降低程序复杂性、提高可控性的目的。

(3) 增强使用的安全性,使用者不必了解实现细节,而只需要通过设计者提供的外部接口来操作它。

(4) 容易实现高度模块化,从而产生软件构件,利用构件快速地组装程序。

3. 继承与派生

对象之间不仅在横向上具有关联特性,在纵向上也存在继承与派生的特性(遗传与变异)。在已有类基础上派生出新的类,这个过程称为继承。新类继承了原始类的特性,新类称为原始类的派生类或子类,而原始类称为新类的基类、父类或超类。派生类可以从它的基类那里继承方法和变量,并且可以修改或增加新的方法使之更适合特殊的需要。继承是一种联结类的层次模型,它提供了一种明确表述共性的方法,允许和鼓励类的重用。利用继承能够将共性的东西抽出来放在父类中,从而简化程序的设计,也可以对现有系统或程序加以重用,扩充或完善现有系统。

4. 多态性

多态性是指允许不同类的对象对同一消息作出不同的响应。比如同样是绘图,圆和矩形将画出不同的结果。在 OOP 中,多态性是通过方法重写实现的,即在基类中定义的方法在子类中加以重写并给出自己的实现,但要求保持方法说明形式的一致。方法被子类重写后,可以表现出不同的行为,这使得同一行为在父类及其各子类中具有不同的操作结果。其好处是达到行为标识统一,减少程序中标识符的个数,方便使用,也为扩展或调整功能提供了机制。

3.2　类的定义与创建对象

类是组成C#程序的基本要素，它是描述一类对象的共同属性（数据成员）和行为方法（方法成员）的封装体，是对象的原型。定义一个新的类，就是创建一个新的数据类型。用类创建对象就是类的实例化。

3.2.1　类的定义

类使用class关键字定义，它可以包含数据成员、方法成员，以及嵌套的类型成员。定义的形式为：

[类修饰符] class 类名称[:基类以及实现的接口列表] {类体}[;]

- 修饰符表示类的可访问性（public、internal等）和一些其他特性（abstract、new和sealed等）。对于一般的类有两种访问权限：public（公有）和internal（缺省）。内部类还可以有private（私有）和protected（保护）。
- 冒号表示类的继承关系或实现关系。当后面的列表中出现类名是表示继承关系，出现接口是表示实现关系。
- 花括号括起的部分为类的内容，即类体。当所定义类包含在另一个类内部，定义类的语句后面需要加分号。

如下示例定义了一个圆（Circle）类。处在大括号内部的是类体。类体中包含1个数据成员radius（半径）和1个用于求面积的方法成员GetArea()。

```
using System;
namespace App3_2_1{
    class Circle{
        public double radius;
            public double GetArea(){
                return Math.PI * radius * radius;
            }
        }
    }
```

3.2.2　创建和使用对象

1. 声明和创建对象

声明对象就是用类来定义对象变量。声明对象的格式为：

类名 对象名;

下面的语句声明一个Circle类型的变量myCircle：

```
Circle myCircle;
```

声明一个对象变量后，该变量的初值是null，对象尚未创建。创建对象需要使用new。

对象也称为类的实例,因此创建一个对象也称为类的实例化。一旦一个对象被创建,它的引用(地址)就被赋给对应的变量。例如,下面的语句通过为对象分配存储空间,并将它的内存引用赋给变量 myCirlce,创建了一个 Cirlce 对象。

```
myCircle = new Circle();
```

注意:对象变量的值是一个引用,是对象的地址,因此一个对象类型的变量有时被称为引用变量。使用一个对象之前必须先创建它。操作一个尚未创建的对象会造成空引用异常(NullReferenceException)。

声明对象和创建对象可以一步完成:

```
类名 对象名 = new 类名();
```

例如,下面的语句声明并实例化 myCircle:

```
Circle myCirlce = new Circle();
```

如果不在同一命名空间中的类要建立 Circle 类的实例,需要指明命名空间。例如:

```
App3_2_1.Circle c = new App3_2_1.Circle();
```

为了简单起见,可以用 using 指令引用给定的命名空间。

2. 使用对象

创建对象后,它的数据成员和方法成员可以通过运算符“.”来访问。例如,给圆的半径赋值 10.0,调用 GetArea()方法求面积:

```
myCircle.radius = 10.0;                    //圆的半径为 10.0
double s = myCircle.GetArea();             //获得圆的面积
```

3.2.3 案例 3-1 测试圆类

用圆类创建一个对象,并且使用对象的数据和方法。运行的结果如图 3-2 所示。

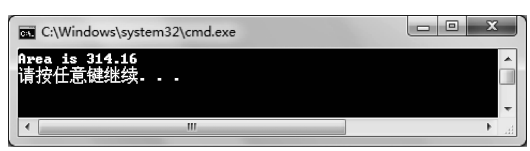

图 3-2 圆类案例运行结果

【技术要点】

- 定义两个类:Circle 类和 Program 类。Program 类包含一个 Main()方法。
- 在 Main()方法中创建一个 Circle 类的对象,并输出它的半径和面积。

【设计步骤】

(1) 在 Visual Studio 下建立一个控制台应用程序 App3_2_3。

(2) 打开【解决方案资源管理器】窗口,右击项目,在弹出的快捷菜单中选择【添加】→

【新建项】命令，在弹出的【添加新项】对话框中选择【类】模板，文件命名为 Circle.cs，单击【确定】按钮，建立一个新的类，程序代码如下所示：

```
using System;
namespace App3_2_3{
    class Circle{
        public double radius;
            public double GetArea(){
                return Math.PI * radius * radius;
        }
    }
}
```

(3) 打开的 Program.cs 文件，编写代码如下所示：

```
using System;
namespace App3_2_3{
    class Program{
        static void Main(string[] args){
            Circle c = new Circle();
            c.radius = 10.0;
            Console.WriteLine("Area is {0:F2}",c.GetArea());
        }
    }
}
```

注意：两个类不在一个命名空间，因此要使用 using 引入命名空间。

(4) 选择【调试】→【开始执行(不调试)(H)】命令执行程序。

3.2.4　构造函数

有的时候希望在创建对象时直接给对象的数据赋值。比如，在创建圆的时候就给半径赋值。在 C# 中，可以在类中定义一个与类同名的函数，称为构造函数，利用它能够初始化对象的数据。

例如，将下述构造方法添加到 Circle 类中：

```
public Circle(double radius){
     this.radius = radius;
}
```

这样就可以使用下面的语句创建圆，并直接给圆的半径赋值：

```
myCircle = new Circle(5.0);                    //将 myCircle.radius 赋值为 5.0
```

这时是否还能使用如下语句创建对象吗？

```
myCircle = new Circle();
```

不能，因为它使用了无参的构造函数。一个类没有定义构造函数，编译系统自动为其添加一个默认的构造函数，默认的构造函数是一个无参的构造函数。但是，当类中定义了构造函

数,系统就不再为类自动添加无参的构造函数。如果自定义的构造函数是有参的,这时如果还利用无参的构造函数建立对象就会出错。为此,可再增加一个构造函数:

```
public Circle(){
}
```

构造函数是可以有多个的,这称为构造函数的重载。构造函数是在创建对象时自动调用的。在创建对象时,根据参数的不同将调用不同的构造函数。构造函数主要用来为对象分配存储空间,完成初始化操作(如给类的成员赋值等)。在 C# 中,类的构造函数遵循以下规定:

(1) 构造函数的函数名与类的名称相同。

(2) 一个类可以有多个构造函数,参数不同。

(3) 构造函数无返回值,也无返回值类型。

(4) 构造函数只能由 new 操作符调用,即建立对象时自动调用。

(5) 构造函数通过关键字 this 调用同一个类的另一个构造函数,例如:

```
public Circle(): this(0,20){
}
```

3.2.5 访问控制

1. 程序集与类的访问控制

当一个应用程序规模较大时,整个应用程序可能由多个项目构成,通过【解决方法资源管理器】窗口来进行管理。不同的项目属于不同的程序集,默认程序集的名字和项目名称相同。如果希望修改程序集的名称,可以双击 Properties 打开项目属性设置界面,设置程序集的名称,在此界面也可以设置默认的域名空间名称。

程序集名称就是编译后的目标文件名称,例如,若一个项目为类库,程序集名称为 MMMM,那么编译后的目标文件名为 MMMM.dll。在新建一个新的类时,所建的类将保存在该程序集中,并使用默认的命名空间。如果要指定特定的命名空间,可以在代码中直接修改。

在定义类时,如果没有指定可访问性,默认的可访问性是 internal,即仅限于同一程序集中的类访问。如果希望跨程序集访问,需要在类的前面加上 public 修饰符,即将类定义成公共类。

在使用不同程序集中的类时,需要先将程序集引入进来。引入程序集可打开【解决方案资源管理器】窗口,展开项目列表,在【引用】上单击鼠标右键,在弹出的快捷菜单中选择【添加引用】命令,打开【添加引用】对话框,如图 3-3 所示。如果要引入没有编译的程序集,可切换到【项目】选项卡,选择要添加的项目名称;如果要引入已经编译的程序集,可切换到【浏览】选项卡,找到需要的目标文件,单击【确定】按钮,即可引入程序集。

对于非嵌套类(不是在一个类中的子类),只有两种可访问性,即 internal 和 public,因此不管哪种情况,同一程序集均可访问。但如果命名空间不同,使用时需要引用命名空间。对于嵌套类,访问属性还可以是 protected 和 private,这和类的成员访问属性类似。

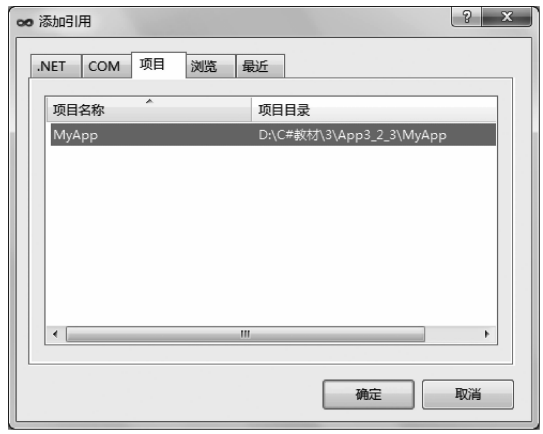

图 3-3　【添加引用】对话框

2. 成员的访问控制

在 C# 中，可以使用访问修饰符 public、protected、internal、internal protected 或 private 声明类成员的可访问性。

- public：访问不受限制。
- protected：访问仅限于当前类和其子类。
- internal：访问仅限于当前程序集。
- internal protected：访问仅限于当前程序集或不属于同一程序集中的子类。
- private：访问仅限于当前类。

如果在成员声明中未指定访问修饰符，则使用默认的可访问性，默认的可访问性为 private。

3.2.6　属性和索引器

1. 属性

属性是一种间接访问数据成员的机制，它不允许直接操作数据内容，而是通过访问器（也称为属性方法）访问数据成员。给属性赋值时使用 set 访问器（Accessor），set 访问器始终使用 value 设置属性值；获取属性值时使用 get 访问器，get 访问器通过 return 返回属性值。定义属性的格式如下：

```
访问修饰符 属性类型 属性名 {
    get { return 属性名; }
    set { 属性名 = value; }
}
```

在访问声明中，如果只有 get 访问器，表示只读属性；如果只有 set 访问器，表示只写属性；如果既有 get 访问器，又有 set 访问器，表示读写属性。

下面的示例将圆的半径定义成属性。

```
using System;
```

```
namespace App3_2_6{
    class Circle{
        private double radius;
        public Circle(){
        }
        public Circle(double radius){
            this.radius = radius;
        }
        public double Radius{
            get { return radius; }
            set { radius = value; }
        }
            public double GetArea(){
                return Math.PI * radius * radius;
            }
    }
}
```

这种机制可以把读取和写入对象的某些特性与一些操作关联起来。甚至,它们还可以对此特性进行计算。尽管访问修饰符可以用来控制对类成员的访问,但是通过使用属性,可以更有效地管理对类成员的访问。

较新版本的C#还支持自动建立默认的get和set访问器,使代码编写更加简练。这种方式不用先声明私有变量,可直接用如下方式定义属性:

```
public String Name {
    get; set;
}
```

可以说,属性是一种特殊的方法,但属性和方法也有不同之处,主要有:

(1) 属性不必使用圆括号,但方法一定使用圆括号。

(2) 属性不能指定参数,方法可以指定参数。

(3) 属性不能使用void类型,方法则可以使用void类型。

2. 索引器

索引器允许类或结构的实例按照与数组相同的方式进行索引。索引器类似于属性,不同的是索引器包含参数。例如,假定有一个表示一周的星期几的名为DayCollection的类,希望直接通过该类的实例week用week[1]来获取英文的"Man"这个字符串,就可以通过索引器来实现。

定义索引需要使用this关键字,并有一个索引参数,一般格式如下:

```
public 类型 this[int index]{
    //get 和 set 访问器
}
```

下面的示例说明了索引器的声明和用法。

```
using System;
namespace App3_2_6_1{
```

```
    class DayCollection{
        string[] days = {"Sun","Mon","Tues","Wed","Thurs","Fri","Sat" };
        public string this[int index]{
            get { return days[index]; }
        }
    }
    class Program{
        static void Main(string[] args){
            DayCollection week = new DayCollection();
            Console.WriteLine(week[1]);
        }
    }
}
```

3.2.7　案例 3-2　Person 类

定义一个人的类(Person),允许在建立对象时直接赋值,也允许在建立对象时不赋值。对于数据成员要定义属性,运行界面如图 3-4 所示。

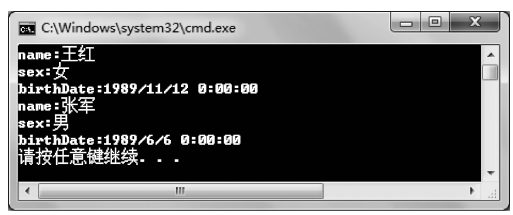

图 3-4　Person 类案例运行界面

【技术要点】

- 定义一个 Person 类,为类设计一个无参的构造函数和一个有参的构造函数,每个数据成员定义属性方法。
- 在 Program 类中创建两个对象进行测试。

【设计步骤】

(1) 在 Visual Studio 下建立一个控制台应用程序 App3_2_7。

(2) 在项目中添加一个新的类 Person,编写代码如下所示:

```
using System;
namespace App3_2_7{
    class Person{
        private string name;                    //姓名
        private char sex;                       //性别
        private DateTime birthDate;             //出生日期
        public Person(){
        }
        public Person(string name, char sex, DateTime birthDate){
            this.name = name;
            this.sex = sex;
```

```
            this.birthDate = birthDate;
        }
        public string Name{
            get{ return name;}
            set{ name = value;}
        }
        public char Sex{
                get{ return sex;}
                set{ sex = value;}
        }
        public DateTime BirthDate {
                get{ return birthDate;}
                set{ birthDate = value;}
        }
        public void Show(){
            Console.WriteLine("name:{0}", name);
            Console.WriteLine("sex:{0}", sex);
            Console.WriteLine("birthDate:{0}", birthDate);
        }
    }
}
```

(3) 打开的 Program.cs 文件,程序代码如下所示:

```
using System;
namespace App3_2_7{
    class Program{
        static void Main(string[]args) {
            Person p1 = new Person();          //建立对象
            Person p2 = new Person("张军", '男', DateTime.Parse("1989-6-6"));
            p1.Name = "王红";                  //调用属性方法赋值
            p1.Sex = '女';
            p1.BirthDate = DateTime.Parse("1989-11-12");
            p1.Show();                         //显示
            p2.Show();                         //显示
        }
    }
}
```

(4) 选择【调试】→【开始执行(不调试)(H)】命令执行程序。

3.3 类的方法与重载

3.3.1 方法的定义

在C#中,数据和操作均封装在类中,数据是以成员变量的形式出现,而操作主要体现在方法的使用上。

在类中,方法定义的一般格式为:

[方法修饰符] 返回值类型 方法名([参数列表]){方法体}

C#中方法除了3.2.5节提到的访问控制修饰符外，还有new、virtual、override、sealed和abstract这5种修饰符，如表3-1所示。

表3-1　方法的修饰符

修饰符	说　明
static	该方法是类的一部分，而不是类实例的一部分。这意味着可以指定classname.method(parameters)来访问方法，而无须创建类实例
virtual	指示该方法可以在子类中覆盖，它不能与static或private访问该修饰符一同使用
override	指示该方法覆盖了基类中的同名方法，这样它就能定义子类特有的行为。基类中被覆盖的方法必须是virtual(虚方法)
new	允许继承类中的一个方法隐藏基类中同名的非虚方法。它会取代原方法，而不是覆盖
sealed	禁止派生此方法，必须与override修饰符一起使用
abstract	该方法不包含具体实现细节，而且必须由子类实现。只能用做abstract类的成员
extern	指示该方法是在外部实现的。它常与DllImport属性一起使用。DLLImport属性指示要由一个DLL提供实现

3.3.2　方法的参数类型

C#方法的参数有五种类型：值参数、引用参数、对象类型参数、输出参数和参数数组。

未用任何修饰符声明的值类型的参数为值参数。值参数在调用该参数所属的方法时创建，并用调用中给定的实参值初始化。当从该方法返回时值参数被销毁。对值参数的修改不会影响到原自变量。值参数通过复制原自变量的值来初始化。

用ref修饰符声明的参数为引用参数。引用参数就是调用者提供的自变量的别名。引用参数并不定义变量，而是直接引用原自变量，因此对引用参数的修改就将直接影响相应自变量的值。在方法调用中，引用参数必须被赋初值。

对象类型的参数传递的是地址，因此对参数的成员的修改将直接影响相应实参。在方法调用时，对象参数必须被赋初值。

用out修饰符定义的参数称为输出参数。如果希望函数返回多个值，可使用输出参数。输出参数与引用参数类似，它并不定义自己的变量，而是直接引用原变量，这样当在函数内为输出参数赋值时，就相当于给原自变量赋值。与引用参数的差别在于：输出参数在调用方法前无需对变量进行初始化。

用params修饰符声明的变量称为参数数组，它允许向函数传递个数变化的参数。在方法的参数类表中只允许出现一个参数数组，而且在方法同时具有固定参数和参数数组的情况下，参数数组必须放在整个参数列表的最后，同时参数数组只允许是一维数组。不能将params修饰符与ref和out修饰符组合起来使用。

3.3.3　案例3-3　演示方法的参数类型

本案例演示方法的5种参数，运行界面如图3-5所示。

图 3-5 演示方法的参数类型程序运行界面

【技术要点】

定义了 5 个方法：Swap()、Swap1()、Swap2()、OutMultiValue()和 MutiParams()，分别使用了值参数、引用参数、对象参数、输出参数和参数数组。

【设计步骤】

(1) 在 Visual Studio 下建立一个控制台应用程序 App3_4_3。

(2) 打开 Program.cs 文件，程序代码如下所示：

```
using System;
namespace App3_4_3{
    class AB{
        public int a;
        public int b;
    }
    class Program{
        //a,b 为值参数
        static void Swap(int a, int b){
            int t;
            t = a; a = b; b = t;
        }
        //a,b 为引用参数
        static void Swap1(ref int a, ref int b){
            int t;
            t = a; a = b; b = t;
        }
        //x 为对象参数
        public static void Swap2(AB x){
            int t = 0;
            if (x.a > x.b){
                t = x.a; x.a = x.b; x.b = t;
            }
        }
        //b 为输出参数
        static void OutMultiValue(int a, out int b){
            b = a * a;
        }
        //a 为参数数组
```

```
        static void MutiParams(params int[] a){
             for (int i = 0; i < a.Length; i++){
                 Console.WriteLine("a[{0}] = {1}", i, a[i]);
             }
        }
        static void Main(string[] args){
            int x = 10, y = 20;
            int[] b = { 10, 20, 30 };
            int v;
            Swap(x, y);
            Console.WriteLine("x = {0},y = {1}", x, y);
            Swap1(ref x, ref y);
            Console.WriteLine("x = {0},y = {1}", x, y);
            AB t = new AB();
            t.a = 5;
            t.b = 2;
            Swap2(t);
            Console.WriteLine("t.a = {0},t.b = {1}", t.a, t.b);
            OutMultiValue(10, out v);
            Console.WriteLine("v = {0}", v);
            MutiParams(b);
        }
    }
}
```

(3) 选择【调试】→【开始执行(不调试)(H)】命令执行程序。

3.3.4　方法重载

方法重载(overload)是指一个类有多个方法,名字相同,但方法的参数列表不一样,这里的不一样可能是个数或类型不一样。重载和方法的返回值无关,返回值可以相同,也可以不同。在同一个类中可以定义多个同名方法。

例如,下面的 OverTest 类包含三个 Area()方法,分别用于求圆、矩形、立方体的面积:

```
using System;
namespace App3_4_4{
    class OverTest{
        public double Area(double radius){
            return (Math.PI * radius * radius);
        }
        public double Area(double length, double breadth){
            return (length * breadth);
        }
        public double Area(double length, double breadth, double height){
            return (length * breadth * height);
        }
    }
    class Program{
        static void Main(string[] args){
            OverTest shape = new OverTest();
```

```
            Console.WriteLine("半径为{0},面积为{1}", 4.0, shape.Area(4.0));
            Console.WriteLine("长为{0},宽为{1},面积为{2}", 4.0, 5.0, shape.Area(4.0, 5.0));
            Console.WriteLine("长为{0},宽为{1},高为{2},面积为{3}", 4.0, 5.0, 6.0, shape.
Area(4.0, 5.0, 6.0));
        }
    }
}
```

3.4 实例成员和类成员

实例成员属于类的实例,只能通过对象访问。类成员属于类,通过类名直接访问。加static修饰的成员为静态成员,也称为类成员,否则为实例成员。类成员包含类变量和类方法。

3.4.1 实例变量和类变量

前面知识中所提的Circle类中的变量radius是一个实例变量,它属于类中的每一个对象实例,不能被同一个类的不同对象共享。例如,对于下面两条语句创建的两个对象:

```
Circle c1 = new Circle();
Circle c2 = new Circle(5.0);
```

c1中的radius独立于c2中的radius,存储在不同的内存空间。c1中radius的变化不会影响c2中的radius,反之亦然。

如果想让一个类的所有实例共享数据,可使用类变量。这种变量是使用static修饰符的数据成员,也称为静态变量,而没有使用static修饰符的数据成员是实例变量。例如,下面的语句定义了一个类变量:

```
static int numOfCircle;
```

类变量的值存储于类的共用内存。因为是共用内存,所以如果某个对象修改了类变量的值,同一类的所有对象都会受到影响。对于整个类来说,类变量的值只存一份。

对于public类型的类变量,外界可通过类名访问,也可通过静态方法或实例方法间接访问。对于public类型的实例变量,外界可通过对象访问,或通过实例方法间接访问。

3.4.2 this关键字

类中的所有实例方法都可以访问实例变量,所以它们称为全局变量。而在方法内说明的变量只能在该方法内部使用,称为局部变量。

局部变量与全局变量同名时,局部变量优先,同名的全局变量被隐藏。用关键字this可以访问隐藏的实例变量。this是实例(对象)的引用。例如,圆类构造函数中就使用了this:

```
public Circle(double radius) {
    this.radius = radius;                    //等号左边是实例变量,右边是局部变量
}
```

3.4.3　实例方法和类方法

没有使用 static 修饰符的方法为实例方法，这种方法必须通过对象来调用。在案例 3-2 中，Person 类中的方法都是实例方法。通过不同的对象调用实例方法，实例方法操作的数据是不同的。

例如，对于下面两条语句创建的两个对象：

```
Person p1 = new Person("张军", '男', DateTime.Parse("1989-6-6"));
Person p2 = new Person("王红", '女', DateTime.Parse("1989-11-12"));
```

p1.Show()操作的是 p1 的数据，而 p2.Show()操作的是 p2 的数据。

与类变量类似，C＃也支持类方法，也称为静态方法。一个方法在定义的时候加上 static 修饰符就是类方法。与实例方法不同的是，类方法只能操作类变量，但实例方法既可以操作实例变量也可以操作类变量。

例如，如果在 Person 类中增加一个类变量 numOfPerson（人数），那么可以定义一个类方法 getNumOfPerson()：

```
private static int numOfPerson
⋮
public static int GetNumOfPerson() {
    return numCircle;
}
```

在 C＃中类方法只能通过类名调用，例如：

```
Console.WriteLIne(Person.GetNumOfPerson());
```

由于直接通过类名就可以使用类方法，因此有的时候把方法定义成类方法会给使用带来方便。例如，Math 类中的方法都是类方法，并且 Math 类的构造函数是私有的，所以也不能建对象，只能通过类名来调用这些方法。

3.4.4　案例 3-4　银行账户

模拟一个银行账户类，假设用户账户由系统自动产生，第一个顾客的账户为 000001，第二个顾客的账户为 000002，第三个顾客的账户为 000003……运行界面如图 3-6 所示。

图 3-6　银行账户案例运行界面

【技术要点】

- 使用一个类变量 lastAccountNumber 存储最后一个账户号，初始值为 0，逐渐累加。

- 定义一个私有的类方法,用于在内部产生新的账户号。
- 账户号属性只要 get 访问器,没有 set 访问器,表明外界只能读取,不能设置,账户号由系统自动产生。

【设计步骤】

(1) 在 Visual Studio 下建立一个控制台应用程序 App3_5_4。

(2) 在项目中添加一个新类 Account,程序代码如下所示:

```
using System;
namespace App3_5_4{
    class Account{
        private string ownerName;            //所有者
        private string acountNumber;         //账号
        private float balance;               //账额
        private static int lastAccountNumber = 0;          //账户总数
        public Account(): this("", 0){
        }
        public Account(string ownerName, float balance){
            this.ownerName = ownerName;
            this.acountNumber = NewAccountNumber();
            this.balance = balance;
        }
        public string OwnerName{
            get { return ownerName; }
            set { ownerName = value; }
        }
        public string AcountNumber{
            get { return acountNumber; }
        }
        public float Balance{
            get { return balance; }
            set { balance = value; }
        }
        //存款
        public float Desposit(float amount){
            balance += amount;
            return balance;
        }
        //取款
        public float Withdraw(float amount){
            balance -= amount;
            return balance;
        }
        //生成新的账号
        private static string NewAccountNumber(){
            return String.Format("{0:D6}", ++lastAccountNumber);
        }
        //取账户总数
        public static int GetAcountNumber(){
```

```
            return lastAccountNumber;
        }
    }
}
```

(3) 打开的 Program.cs 文件,编写代码如下所示:

```
using System;
namespace App3_5_4{
    class Program{
        static void Main(string[] args){
            Account account1 = new Account("John", 30f);  //建立第一个账户
            Account account2 = new Account();             //建立第二个账户
            account2.OwnerName = "Mary";                  //设置账户人名
            account2.Balance = 50f;                       //设置初始存款
            account1.Desposit(60f);                       //向第一个账户存款
            account1.Withdraw(10f);                       //从第一个账户取款
            Console.WriteLine(Account.GetAcountNumber());
            Console.WriteLine("{0},{1},{2}", account1.AcountNumber, account1.OwnerName,
        account1.Balance);
            Console.WriteLine("{0},{1},{2}", account2.AcountNumber,account2.OwnerName,
        account2.Balance);
        }
    }
}
```

(4) 选择【调试】→【开始执行(不调试)(H)】命令执行程序。

3.5　继承与多态

继承是面向对象程序设计的一个重要特征,它允许在现有类的基础上创建新类,新类从现有类中继承类的成员,而且可以重新定义或加进新的成员,从而形成类的层次或等级。一般称被继承的类为基类、父类或超类,而继承后产生的新类为派生类或子类。

(1) 如果类 A 是类 B 的派生类,则类 A 继承了类 B 的变量和方法。在派生类 B 中包括了两部分内容:从父类 A 中继承下来的变量和方法,自己新增加的变量和方法。

(2) 在 C# 中类只支持单一继承,不支持多重继承,接口可弥补这方面的一些缺陷。

(3) 继承是可传递的。如果 C 从 B 派生,而 B 从 A 派生,那么 C 就会既继承在 B 中声明的成员,又继承在 A 中声明的成员。

(4) 派生类可扩展它的直接基类,添加新的成员,但不能移除父类中定义的成员。

(5) 除构造函数外,其他非私有成员都可以被继承。私有数据成员虽然不能被继承,但在派生类中可以通过公有方法间接访问。

(6) 派生类可以通过声明具有相同说明的新成员来隐藏那个被继承的成员,但隐藏继承成员并不移除该成员,它只是使被隐藏的成员在派生类中不可直接访问。

3.5.1 派生类的声明

1. 派生类的声明格式

派生类的声明格式为：

```
类修饰符 class 派生类类名: 基类类名 {
    类体
}
```

在类的声明中，通过在类名的后面加上冒号和基类名表示继承。
例如在下面的示例中，Cylinder 类继承 Circle 类：

```
using System;
namespace App3_6_1{
    class Circle {
        private double radius;
        public Circle() {
            radius = 1.0;
        }
        public Circle(double radius) {
            this.radius = radius;
        }
        public double Radius{
            get { return radius; }
            set { radius = value; }
        }
        public double GetArea(){
            return Math.PI * radius * radius;
        }
        public double GetPerimeter() {
            return 2 * Math.PI * radius;
        }
    }
    class Cylinder: Circle {
        private float height;
        public Cylinder(){
        }
        public Cylinder(float radius, float height):base(radius){
            this.height = height;
        }
        public float Height{
            get{ return height;}
            set{ height = value;}
        }
        public double GetVolumn() {
            return GetArea() * height;
        }
    }
```

```
    class Program{
        static void Main(string[] args){
            Cylinder c = new Cylinder(6,20);      //建立一个底半径为 6,高为 20 的圆
            Console.WriteLine(c.GetArea());       //显示圆柱的底面积
            Console.WriteLine(c.GetVolume());     //显示圆柱的体积
        }
    }
}
```

圆柱类是圆类的派生类,它继承了圆类的成员,因此不需再定义底半径,只需定义一个height(高)。圆柱类可以利用 GetArea()(这是圆类的方法)求底面积。此外,圆柱类扩展了圆类,增加了求体积的方法。

2. 派生类对父类成员的继承

父类中用 public 修饰的公有成员,派生类内部可以直接使用。父类中用 private 修饰的私有成员,派生类内部不能直接使用,但可以通过公有方法或属性间接使用。父类中用 protected 修饰的保护成员,派生类内部也可以直接使用。与 public 不同的是,用 protected 修饰的成员不能被外界访问。

3. base 关键字

关键字 this 指类的实例自己,而关键字 base 指父类。关键字 base 的作用有两个:

1) 调用父类的构造函数

在 C# 中,当创建派生类对象时,系统会首先执行父类的构造函数,然后再执行派生类的构造函数。在类层次结构中,父类总是首先被实例化的。如果想要调用父类的非缺省构造函数,那么必须使用 base 关键字。

例如,圆柱类的无参构造函数中调用了父类无参构造函数,这时 base 语句可以省略;而在有参的构造函数中调用了父类的有参构造函数,就使用了 base:

```
public Cylinder() {
}
public Cylinder(float radius,float height):base(radius){
    this.height = height;
}
```

注意:假设父类中定义几种构造函数,但没有无参的构造函数,那么派生类就不能使用父类的无参构造函数,这种情况必须显示使用 base 调用父类有参构造函数。

2) 调用父类的方法

可通过 base 调用父类的方法,例如,在 Cylinder 类增加一个 GetArea()方法,用于求圆柱的表面积,Cylinder 类将使用自己的 GetArea()方法。在这种情况下,如果 Cylinder 类中还想使用 Circle 类的 GetArea()方法,就需要通过 base。

```
//求圆柱的表面积
public double GetArea() {
    return 2 * base.GetArea() + GetPerimeter() * height;
}
```

```
//求圆柱的体积
public double GetVolume() {
    return base.GetArea() * height;     //这里使用了 super,表明使用的父类的方法
}
```

4. 隐藏类的成员

在派生类中,通过声明与父类同名的新成员可以隐藏父类的成员。

(1) 隐藏一个父类成员不算错误,但会导致编译器发出警告,若要取消警告,在派生类成员的声明中可以使用 new 关键字,表示派生类有意隐藏了基类成员。

(2) 隐藏方法,需要方法名称相同,且参数列表一样(个数及对应的类型相同),但返回值类型可以不同。

例如,前面提到的 Cirlce 中有 GetArea()方法,但在 Cylinder 重新定义了 GetArea()方法,这就是方法隐藏。如果想防止编译器发出警告,可以使用 new 关键字。

```
public new double GetArea() {                          //求圆柱表面积
    return 2 * base.GetArea() + GetPerimeter() * height;
}
```

3.5.2 方法覆盖与多态性

1. 方法覆盖

方法覆盖(Override)是指在派生类中重写基类的方法。派生类中重写的方法与父类的方法具有同样的签名(方法名称相同,参数列表、返回值类型及权限修饰一样)。只能重写基类中使用 abstract、virtual 或 override 修饰符修饰的方法,而且在派生类中的方法要使用 override 修饰符,即指明为覆盖方法。

例如,在前面的例子如果 Circle 中 GetArea()方法使用了 virtual 修饰符,而 Cylinder 中 GetArea()方法使用了 override 修饰符(不能使用 new 修饰符),那么 Cylinder 类中的 GetArea()方法就是覆盖了 Circle 中的 GetArea()方法。

Circle 中求圆的面积定义:

```
public virtual double GetArea() {
    return Math.PI * Radius * Radius;
}
```

Cylinder 中求圆柱的表面积定义:

```
public override double GetArea() {
    return 2 * base.GetArea() + GetPerimeter() * height;
}
```

关于方法覆盖要注意以下几点:

(1) 不能用派生类的静态方法覆盖父类中的实例方法。

(2) 带关键字 sealed 的方法不能被覆盖。

(3) 抽象方法必须在派生类中被覆盖,否则派生类也必须是抽象的。

2. 方法重载、方法隐藏、方法覆盖之间的区别

一个类有多个方法(包括从基类继承下来的方法)名字相同,但参数列表不一致,这是方法重载。对于方法重载,系统在编译时,根据传递的参数个数、类型信息决定调用哪个重载的方法。

一个派生类的方法与基类的方法同名,且参数列表一致,返回值类型可以不同,这是方法隐藏。对于方法隐藏,系统在编译时,根据变量的类型决定调用谁的成员方法。例如,如果变量是基类类型,即使存储的对象是派生类对象(C#允许这样做),调用的也是基类的方法;如果变量是派生类类型,调用的是派生类的方法。

一个派生类的方法与基类的方法签名一样,实现可以不同,并且基类的方法是使用abstract、virtual或override修饰符修饰的方法,派生类的方法使用了override修饰符,这是方法覆盖。对于方法覆盖,系统在运行时,根据实际存储的对象决定调用谁的成员方法。

3. 多态性

多态性是指不同的对象收到相同的消息时会产生不同动作。多态性允许以相似的方式来对待所有的派生类,尽管这些派生类是各不相同的。

在C#语言中,运行一类变量和接口变量,存放其下属类(子类或实现接口的类)的实例。如果下属类的某个方法是覆盖方法,那么当通过父类变量或接口变量(已经存放了下属类对象)调用该方法时,就可以在运行时根据实际存储的下属类对象决定调用谁的成员方法。多态性就是通过这种机制实现的。例如,对于前面定义的圆和圆柱类,可以用圆的变量来存圆的实例或圆柱的实例,当调用GetArea()方法时,具体根据存储的是圆的实例还是圆柱的实例决定调用谁的方法。

```
Circle c = new Cylinder(6, 20);
```

尽管c是圆类型的变量,但调用GetArea()时得到的是圆柱的表面积,因为GetArea()方法是覆盖方法,而c存储的又是圆柱对象。

在程序设计中,提倡使用更抽象的变量来存储实例,这样不仅便于程序的维护和扩展,也给程序设计带来方便。而这正依赖于多态性。用接口变量或父类变量存储下属类的实例,也能通过这样的变量动态调用实例的方法。

例如下面的示例中,Animal(动物)类有一个Sleep()方法,Cat(猫)类是Animal的派生类,它覆盖了Sleep()方法。Test(测试)类有一个方法InvokeSleep(),在该方法中调用Sleep()方法时,系统将根据animal变量存储的实际对象决定调用Animal的Sleep()方法,还是调用Cat的Sleep()方法。

```
using System;
namespace App3_6_2_1{
    class Animal{
        public virtual void Sleep(){
        Console.WriteLine("Sleeping");
    }
}
```

```
    class Cat:Animal{
        public override void Sleep(){
            Console.WriteLine("Cat Sleeping");
        }
    }
    class Test{
        public void InvokeSleep(Animal animal){
            animal.Sleep();
        }
    }
    class Program{
        static void Main(string[] args){
            Test test = new Test();
            Cat c = new Cat();
            test.InvokeSleep(c);
        }
    }
}
```

以上代码运行输出结果为：

```
Cat Sleeping
```

使用类的多态性的好处是可以采用一种通用的方式来处理派生类，因为可以认为派生类对象的类型是基类类型。

3.5.3 案例 3-5 用继承的方式定义 Student 类和 Teacher 类

定义一个学生类 Student 和一个教师类 Teacher。学生类的数据成员有姓名、性别、出生日期、专业和入学成绩；教师类的数据成员有姓名、性别、出生日期、部门和工资。设计一个测试类对上述类进行测试，运行界面如图 3-7 所示。

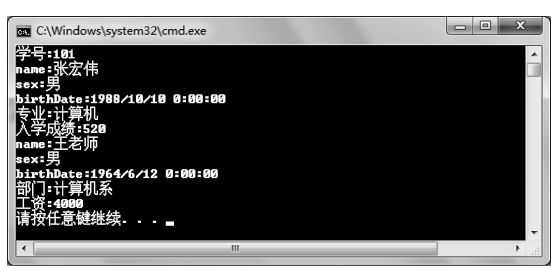

图 3-7 Student 和 Teacher 类演示界面

【技术要点】

在案例 3-2 基础上进行设计，Student 类和 Teacher 类都继承 Person 类，都重载 Show() 方法。

【设计步骤】

(1) 在 Visual Studio 下建立一个控制台应用程序项目 App3_6_3。

(2) 将案例 3-2 设计的 Person 类复制到项目中,域名空间改为 App3_6_3。

(3) 打开 Person.cs 文件,将 Show()方法改成虚方法,具体代码如下所示:

```
public virtual void Show(){
    Console.WriteLine("姓名:{0}", name);
    Console.WriteLine("性别:{0}", sex);
    Console.WriteLine("出生日期:{0}", birthDate);
}
```

(4) 在项目中添加一个新类 Teacher,程序代码如下所示:

```
using System;
namespace App3_6_3{
    class Teacher : Person{
        private string departmenet;                    //部门
        private float salary;                          //工资
        public Teacher(){
        }
        public Teacher(string name, char sex, DateTime birthDate, string departmenet, float
salary): base(name, sex, birthDate){
            this.departmenet = departmenet;
            this.salary = salary;
        }
        public string Departmenet{
            get { return departmenet; }
            set { departmenet = value; }
        }
        public float Salary{
            get { return salary; }
            set { salary = value; }
        }
        //覆盖父类的方法
        public override void Show(){
            base.Show();                               //调用父类的方法
            Console.WriteLine("部门:{0}", departmenet);
            Console.WriteLine("工资:{0}", salary);
        }
    }
}
```

(5) 在项目中添加一个新类 Student,编写代码如下所示:

```
using System;
namespace App3_6_3{
    class Student : Person{
        private string no;                             //学号
        private string major;                          //专业
        private float enterScore;                      //入学成绩
        public Student(){
        }
        public Student(string no, string name, char sex, DateTime birthDate, string major,
float enterScore): base(name, sex, birthDate){
```

```
            this.no = no;
            this.major = major;
            this.enterScore = enterScore;
        }
        public string No{
            get { return no; }
            set { no = value; }
        }
        public string Major{
            get { return major; }
            set { major = value; }
        }
        public float EnterScore{
            get { return enterScore; }
            set { enterScore = value; }
        }
        //覆盖父类的方法
        public override void Show(){
            Console.WriteLine("学号:{0}", no);
            base.Show();                                    //调用父类的方法
            Console.WriteLine("专业:{0}", major);
            Console.WriteLine("入学成绩:{0}", enterScore);
        }
    }
}
```

(6) 打开 Program.cs 文件,程序代码如下所示:

```
using System;
namespace App3_6_3{
    class Program{
        static void Main(string[] args){
            Person s = new Student("101","张宏伟", '男', DateTime.Parse("1988-10-10"),
"计算机", 520f);
            Person t = new Teacher("王老师", '男', DateTime.Parse("1964-06-12"),"计算机
系", 4000f);
            s.Show();
            t.Show();
        }
    }
}
```

(7) 选择【调试】→【开始执行(不调试)(H)】命令执行程序。

3.5.4 sealed 修饰符

在C#语言中,sealed 修饰符表示密封,用于类时,表示该类不能再被继承;用于方法和属性时,表示该方法或属性不能再被重写。

(1) 不能和 abstract 同时使用,因为这两个修饰符在含义上互相排斥。

(2) 使用 sealed 修饰符的方法或属性必须是基类中相应的虚成员,因此它和 override

关键字一起使用。

sealed 通常用于实现第三方类库时不想被客户端继承，或用于没有必要再继承的类以防止滥用继承造成层次结构体系混乱。恰当的利用 sealed 修饰符也可以提高一定的运行效率，因为不用考虑继承类会重写该成员。

下面的示例演示了 sealed 的使用。

```
using System;
namespace App3_6_4{
    public class A{
        public virtual void M(){
            Console.WriteLine("A.M()");
        }
        public virtual void M1(){
            Console.WriteLine("A.M1()");
        }
    }
    public class B : A{
        public sealed override void M(){
            Console.WriteLine("B.M()");
        }
        public override void M1(){
            Console.WriteLine("B.M1()");
        }
    }

    public sealed class C : B{
        /*此处若不注释，为出现编译错误
        protected override void M(){
             Console.WriteLine("C.M()");
        }*/
        public override void M1(){
             Console.WriteLine("C.M1()");
        }
    }
    /*此处若不注释，为出现编译错误
    public class D : C{
        public override void M1() {
             Console.WriteLine("D.M1()");
        }
    }
    */
    class Program{
        static void Main(string[] args){
            A a = new A();
            a.M();
            a.M1();
            A a1 = new B();
            a1.M();
            a1.M1();
            B b = new C();
```

```
            b.M1();
        }
    }
}
```

程序的运行结果为：

```
A.M()
A.M1()
B.M()
B.M1()
C.M1()
```

3.6 抽象类与接口

3.6.1 抽象类

C#语言中，用 abstract 关键字来修饰一个类时，这个类称为抽象类。用 abstract 关键字来修饰一个方法时，这个方法称为抽象方法。抽象类的定义格式如下：

```
[修饰符] abstract class 类名 {                    //抽象类
…                                                  //类体
}
```

抽象方法的定义格式如下：

```
[修饰符] abstract 返回值类型 方法名([参数列表]);  //抽象方法
```

(1) 抽象方法只有声明，没有实现。

(2) 抽象类可以包含抽象方法，也可以不包含抽象方法。但是包含抽象方法的类必须定义成抽象类。

(3) 抽象类不能被实例化，抽象类可以被继承，不能被定义成 sealed 类。

(4) 继承抽象类的类必须实现抽象类的抽象方法，否则也必须定义成抽象类。

(5) 一个类实现某个接口，但没有实现该接口的所有方法，这个类必须定义成抽象类。

(6) 抽象方法不能使用 virtual 修饰。

使用抽象类的目的是它可以把派生类共有部分抽出来，并且实现所能实现的部分，从而为派生类提供继承，但不必实现所有的方法，对于那些只知道行为是什么，不用知道具体怎么做的方法，可以只给出说明，即定义成抽象的，而把具体的实现交给派生类去做。把那些共有的、但不能具体实现的行为抽出来定义成抽象的方法，作用有两点：一是为派生类规定了统一的规范，二是为了实现多态性。

3.6.2 案例 3-6 一组图形类

要开发一个绘图软件，需要定义正方形、矩形和圆类。这里定义的圆与前面介绍的圆不同，它不是用于求面积，而是用于绘图。每个类都有数据成员：x、y(坐标)，c(颜色)，g(图形

对象)和绘图方法 draw()。在测试类中使用上述三个类建立对象并绘图,运行界面如图 3-8 所示。

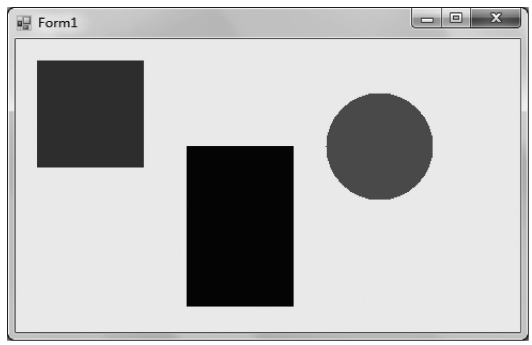

图 3-8　一组图形类运行界面

【技术要点】

- 利用 C# 类的继承特性来构造这组图形类。先定义 Shape 类,包含了所有图形所共有的成员变量 x,y,c 和一个抽象的方法 draw()。
- Square(正方形)由 Shape 派生而来; Rectangle(矩形)由 Square 派生而来; Circle(圆)由 Shape 派生而来。

【设计步骤】

(1) 在 Visual Studio 中建立【Windows 窗体应用程序】项目 App3_7_2。

(2) 在项目中添加一个新类 Shape,编写代码如下所示:

```
using System;
using System.Drawing;
namespace App3_7_2{
    abstract class Shape{
        protected int x, y;                    //坐标
        protected Color c;                     //色彩
        //无参构造方法
        public Shape(){
        }
        //有参的构造方法
        public Shape(int x, int y, Color c){
            this.x = x;
            this.y = y;
            this.c = c;
        }
        public int X{
            get{ return this.x; }
            set{ this.x = value;}
        }
        public int Y{
            get{ return this.y; }
            set{ this.y = value;}
```

```
        }
        public int Color{
            get{ return this.color; }
            set{ this.color = value;}
        }
        //抽象的绘图方法
        public abstract void Draw(Graphics g);
    }
}
```

(3) 在项目中添加一个新类 Square,编写代码如下所示:

```
using System;
using System.Drawing;
namespace App3_7_2{
    //正方形类,继承了形状类
    class Square : Shape{
        protected int w;
        public Square(){
        }
        public Square(int x, int y, int w, Color c)
            : base(x, y, c){
            this.w = w;
        }
        public int W{
            get{ return this.w; }
            set {this.w = value;}
        }
        //实现求画图的方法
        public override void Draw(Graphics g){
            SolidBrush sb = new SolidBrush(c);
            g.FillRectangle(sb, x, y, w, w);              //画填充矩形
        }
    }
}
```

(4) 在项目中添加一个新类 Rectangle,编写代码如下所示:

```
using System;
using System.Drawing;
namespace App3_7_2{
//矩形类,继承了正方形类
    class Rectangle : Square {
        private int h;
        public Rectangle() { }
        public Rectangle(int x, int y, int w, int h, Color c)
            : base(x, y, w, c){
            this.h = h;
        }
        public int H{
            get{ return this.h; }
            set{ this.h = value;}
```

```
        }
        public override void Draw(Graphics g){
            SolidBrush sb = new SolidBrush(c);
            g.FillRectangle(sb, x, y, w, h);          //画填充矩形
        }
    }
}
```

(5) 在项目中添加一个新类 Circle,程序代码如下所示:

```
using System;
using System.Drawing;
namespace App3_7_2{
    //定义圆类,继承了形状类
    class Circle : Shape{
        private int r;
        public Circle() { }
        public Circle(int x, int y, int r, Color c)
            : base(x, y, c){
            this.r = r;
        }
        public int R{
            get{ return this.r; }
            set{ this.r = value;}
        }
        public override void Draw(Graphics g){
            SolidBrush sb = new SolidBrush(c);
            g.FillEllipse(sb, x - r, y - r, 2 * r, 2 * r);     //画填充圆
        }
    }
}
```

(6) 为窗体 Form1 添加 Paint 事件,要编写的事件处理程序如下所示:

```
private void Form1_Paint(object sender, PaintEventArgs e){
    Graphics g = e.Graphics;
    Shape s1 = new Square(20, 20, 100, Color.Blue);
    s1.Draw(g);                              //调用绘图方法
    Square s2 = new Rectangle(160, 100, 100, 150, Color.Black);
    s2.Draw(g);
    Shape s3 = new Circle(340, 100, 50, Color.Green);
    s3.Draw(g);
}
```

(7) 运行程序。

3.6.3　接口

一个接口定义一个协定。实现接口的类必须遵守其协定。在某种程度上,接口像一个抽象类。和任何类一样,接口可以定义方法、属性、事件和索引,但是接口不提供成员的实现。实现接口的任何类都必须提供接口中所声明的抽象成员的实现。

1. 接口声明

接口声明的格式为：

```
[访问修饰符] interface 接口名 [: 基接口] {
    接口体
}
```

接口与类的定义相似。但接口的成员只能是方法、属性、事件和索引，且不允许声明成员上的修饰符，即使是 public 都不行，因为接口成员总是公有的，也不能声明为虚的和静态的。如果需要修饰符，最好让实现类来声明接口成员必须全都具有公共可访问性。

例如，下面声明的接口包含 4 种成员。

```
public delegate void EventHandler(object sender, EventArgs e);
interface IExample{
    string this[int index]{ get; set; }    //索引
    event EventHandler E;                  //事件
    void F(int value);                     //方法
    string P{ get; set; }                  //属性
}
```

接口可以使用多重继承，一个接口继承一个或多个基接口。例如：

```
interface IControl{
    void Paint();
}
interface ITextBox: IControl{              //从 IControl 继承
    void SetText(string text);
}
interface IListBox: IControl{              //从 IControl 继承
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox{   //从 ITextBox 和 IListBox 继承
}
interface IDataBind{
    void Bind();
}
```

接口 IComboBox 同时从 ITextBox 和 IListBox 继承。

2. 接口实现

接口可以由类实现。实现接口的标识符出现在类的基列表中。

(1) 实现接口的任何类都必须提供接口中所声明的抽象成员的定义，否则该类必须定义成抽象类。

(2) 当一个类实现多个接口时，接口之间用逗号分隔。

(3) 类的基列表同时包含基类和接口时，列表中首先出现的是基类。

如下示例中，类 EditBox 从类 Control 派生，并且同时实现 ITextBox 接口。

```
class Control: IControl {
    public void Paint(){
        Console.WriteLine("运行了 Paint");        //这里只是简单模拟
    }
}
class EditBox: Control, ITextBox {
    public void SetText(string text){
        Console.WriteLine("运行了 SetText");      //这里只是简单模拟
    }
    public void Bind(){
        Console.WriteLine("运行了 Bind");
    }
}
class Program{
    static void Main(string[] args){
        EditBox e = new EditBox();
        e.Paint();
        e.SetText("abc");
        e.Bind();
    }
}
```

在前面的示例中，ITextBox 接口中的 SetText()方法和 IDataBound 接口中的 Bind()方法是使用 EditBox 类的公共成员实现的。C# 提供了另一种方式来实现这些方法，使得实现类避免将这些成员设置成公共的。这就是接口成员可以用限定名来实现。例如，在 EditBox 类中将 SetText()方法命名为 ITextBox.SetText()，将 Bind()方法命名为 IDataBound.Bind 方法。

```
class EditBox : Control, ITextBox, IDataBind{
    void ITextBox.SetText(string text){
        Console.WriteLine("运行了 SetText,设置的文本为{0}",text);
    }
    void IDataBind.Bind(){
        Console.WriteLine("运行了 Bind");
    }
}
```

用这种方式实现的接口成员称为显式接口成员，显式接口成员只能通过接口来调用。例如，在 EditBox 中实现的 SetText()方法只能通过 ITextBox 来调用。

```
class Program{
    static void Main(string[] args){
        EditBox e = new EditBox();
        IDataBind idb = e;
        ITextBox itb = e;
        e.Paint();
        idb.Bind();
        itb.SetText("abc");
    }
}
```

3. 接口与抽象类的区别

接口与抽象类很相似,但也有不同:

(1) 抽象类可提供某些方法的实现,而接口的方法都是抽象的。

(2) 抽象类中可以有成员变量(包含静态成员变量)、属性、常量和静态方法,并且它们可以是非公共的;而接口中不能有成员变量、常量、静态方法,只能有方法、属性、事件和索引,而且不允许声明成员上的修饰符。

(3) 抽象类中增加一个具体的方法,则子类都具有此具体方法;而接口中新增加方法,则子类必须实现此方法。

(4) 子类最多能继承一个抽象类,而接口可以继承多个接口,一个类可以实现多个接口。

(5) 抽象类和它的子类之间应该是一般和特殊的关系,而接口仅仅是它的子类应该实现的一组规则,无关的类也可以实现同一接口。

4. 接口的作用

接口的作用主要体现在以下几个方面:

(1) 接口可以规范类的方法,使实现接口的类具有相同的方法签名。任何实现了接口的类都必须实现接口所规定的方法,否则必须定义为抽象类。

(2) 接口提供了一种抽象的机制,通过接口可以把功能设计和实现分离。接口只告诉用户方法的特征是什么,它并不关注是如何实现的,接口指出如何使用一个对象,而不说明它如何实现。

(3) 接口能更好地体现多态性,通过接口实现不相关类的相同行为,而无需考虑这些类之间的关系。任何实现接口的类的实例都可以通过接口来调用。通过抽象的接口来操纵具体的对象,可以极大地减少子系统实现之间的相互依赖关系,使对象之间彼此独立,并可以在运行时替换有相同接口的对象,动态改变它们相互的关系,实现多态。

5. 面向接口编程

接口变量可以存放实现这一个接口的类的对象,并且可以调用实现接口的方法。利用这一特征可以实现设计和代码分离。

下面是一个分层结构程序设计的例子。假设 A 层使用 B 层,可以先根据 A 层的需要设计好 B 层的方法,即定义 B 层的接口,如 IB。这样的好处是不用先考虑具体实现,而仅着眼于功能的设计。

```
public interface IB {
    void methodOfB();                    //B层所要具有的方法
}
```

在 A 层中不直接使用 B 类而是使用接口变量,然后根据需要传入实现接口的对象。这样的好处是通过接口既可以统一 B 层的规范,又允许 B 层有不同的实现。

```
public class A {
```

```
    private IB b;                          //定义接口变量,用于存放对象
    public IB B {                          //用于传入对象
        set{ b = value;}
    }
    public void methodOfA(){
        b.methodOfB();                     //通过接口变量使用 B 层的方法
    }
}
```

根据需要,给出 IB 的不同实现,并且只要接口不变,B 层的不同实现不会对 A 层造成影响,便于程序修改和维护。下面是两个不同的实现类:

```
public class B1:IB{
    public void methodOfB() {
        Console.WriteLine("第一种实现");
    }
}
public class B2:IB{
    public void methodOfB() {
        Console.WriteLine("第二种实现");
    }
}
```

根据需要传入不同的对象:

```
IB b1 = new B1();
A a = new A();
a.B = b1;                                  //传入第一种实现类的对象
a.methodOfA();                             //输出的结果是"第一种实现"
IB b2 = new B2();
a.B = b2;                                  //传入第二种实现类的对象
a.methodOfA();                             //输出的结果是"第二种实现"
```

在程序设计中,尽量使用接口变量,这种编程思想称为面向接口的编程。按照这种思想,在系统分析和架构中分清层次和依赖关系,每个层次不是直接向其上层提供服务(即不是直接实例化在上层中),而是通过定义一组接口,仅向上层暴露其接口功能,上层对于下层仅仅是接口依赖,而不依赖具体类。利用接口使设计与实现相分离,使利用接口的用户程序不受不同接口实现的影响,不受接口实现改变的影响。

3.7　委托与事件

3.7.1　委托

委托是一种包装方法调用的类型。就像类型一样,可以在方法之间传递委托实例,并且可以像方法一样调用委托实例。委托属于引用类型,它引用方法,引用方法的过程称为绑定。委托主要用于.NET Framework 中的事件处理程序和回调函数。

所有委托都是从 System.Delegate 继承而来的,并且有一个调用列表,这是在调用委托

时所执行的方法的一个链接列表。产生的委托可以用匹配的签名引用任何方法。没有为具有返回类型并在调用列表中包含多个方法的委托定义返回值。

下面的示例中定义一个委托 GreetingDelegate,该委托可以绑定有一个 string 参数,返回值类型为 void 的方法。GreetPeople()方法接受一个 GreetingDelegate 类型的参数,从而实现可将不同的方法作为参数传递给它,以实现不同欢迎语言的显示。

```
using System;
namespace App3_8_1{
    //定义委托,它定义了可以代表的方法的类型
    public delegate void GreetingDelegate(string name);
    class Program {
        private static void EnglishGreeting(string name) {
            Console.WriteLine("Morning, " + name);
        }
        private static void ChineseGreeting(string name) {
            Console.WriteLine("早上好, " + name);
        }
        //注意此方法,它接受一个 GreetingDelegate 类型的方法作为参数
        private static void GreetPeople(string name, GreetingDelegate MakeGreeting) {
            MakeGreeting(name);
        }
        static void Main(string[] args) {
            GreetPeople("Jimmy Zhang", EnglishGreeting);
            GreetPeople("张子阳", ChineseGreeting);
        }
    }
}
```

输出如下:

```
Morning, Jimmy Zhang
早上好, 张子阳
```

可以将多个方法赋给同一个委托,或者叫将多个方法绑定到同一个委托,当调用这个委托的时候,将依次调用其所绑定的方法。例如:

```
static void Main(string[] args) {
    GreetingDelegate delegate1;
    delegate1 = EnglishGreeting;          //先给委托类型的变量赋值
    delegate1 += ChineseGreeting;         //给此委托变量再绑定一个方法
    //将先后调用 EnglishGreeting 与 ChineseGreeting 方法
    delegate1("Jimmy Zhang");             //绕过 GreetPeople 方法,通过委托来直接调用
    GreetPeople("张子阳", delegate1);     //通过 GreetPeople 方法调用
}
```

输出为:

```
Morning, Jimmy Zhang
早上好, Jimmy Zhang
Morning, Jimmy Zhang
早上好, 张子阳
```

也可以用匹配的方法初始化代理对象，例如：

```
GreetingDelegate delegate1 = new GreetingDelegate(EnglishGreeting);
delegate1 += ChineseGreeting;              //给此委托变量再绑定一个方法
```

既然可以给委托绑定一个方法，那么也应该有办法取消对方法的绑定，很容易想到“－＝”。例如：

```
static void Main(string[] args) {
    GreetingDelegate delegate1 = new GreetingDelegate(EnglishGreeting);
    delegate1 += ChineseGreeting;          //给此委托变量再绑定一个方法
    //将先后调用 EnglishGreeting 与 ChineseGreeting 方法
    GreetPeople("Jimmy Zhang", delegate1);
    delegate1 -= EnglishGreeting;          //取消对 EnglishGreeting 方法的绑定
    //将仅调用 ChineseGreeting
    GreetPeople("张子阳", delegate1);
}
```

输出为：

```
Morning, Jimmy Zhang
早上好, Jimmy Zhang
早上好, 张子阳
```

在 2.0 以前声明委托的唯一方法是使用命名方法，到了 2.0 引入了匿名方法，也就是说可以不需要事先创建单独的方法，而是在委托调用中执行未命名内联语句块，因此减少了实例化委托所需的编码系统开销，例如：

```
delegate void TestDelegate(string s);
class TestClass{
    static void Main(){
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };
        testDelB("Hello 2 ");
    }
}
```

3.7.2　事件

事件是对象发送的消息，以发信号通知操作的发生。操作可能是由用户交互（例如鼠标单击）引起的，也可能是由某些其他的程序逻辑触发的。引发（触发）事件的对象叫做事件产生对象。捕获事件并对其作出响应的对象叫做事件接收对象。两者之间存在一个媒介，这个媒介就是委托。可以通过使用委托把事件与处理这些事件的方法进行绑定。委托允许其他类向指定类注册处理事件的方法（事件处理程序）。当事件发生时，委托将调用绑定的方法。

1. 定义事件数据类

事件发生时，事件产生对象向事件接收对象传递的数据称为事件数据。EventArgs 是事件数据的类的基类，在事件引发时不向接收方传递事件数据，可以直接使用此类。若需要

传递数据,可从 EventArgs 派生一个新的类。例如,下面的示例定义一个图书发行事件的事件数据类,该类封装了图书信息。

```
using System;
namespace App3_8_2
    class IssueEventArgs: EventArgs{
        private string bookName;
        private string bookAuthor;
        public IssueEventArgs(string bookName, string bookAuthor){
            this.bookName = bookName;
            this.bookAuthor = bookAuthor;
        }
        public string BookName {
            get{
                return bookName;
            }
        }
        public string BookAuthor {
            get{
                return bookAuthor;
            }
        }
    }
}
```

2. 在事件发生类中声明并触发事件

在事件发生类中,先创建一个委托,再利用委托声明事件,然后在相应的方法中触发事件。例如,下面示例定义了一个出版社类,该类声明一个图书发行事件。

```
using System;
namespace App3_8_2{
    class Publisher {
        //声明事件所需的代理
        public delegate void IssueHandler(object sender, IssueEventArgs e);
        public event IssueHandler OnIssue;    //声明事件
        private string name;
        public Publisher(string name){
            this.name = name;
        }
        public string Name{
            get{
                return name;
            }
        }
        //触发事件的方法
        public void Issue(string bookName, string bookAuthor){
            if(OnIssue != null){
                Console.WriteLine("发行" + bookName);
```

```
                    OnIssue(this,new IssueEventArgs(bookName, bookAuthor));
                }
            }
        }
    }
```

3. 在事件接收类中定义事件处理程序

下面定义一个订购者类，该类的 Receive()方法用于事件处理。

```
using System;
namespace App3_8_2{
    //订购者
    class Subscriber {
        //定义事件处理程序
        public void Receive(object sender, IssueEventArgs e){
            Console.WriteLine("订购的图书出版,马上发送给你!");
            Console.WriteLine("书名:{0}", e.BookName);
            Console.WriteLine("作者:{0}", e.BookAuthor);
            Console.WriteLine("出版社:{0}", ((Publisher) sender).Name);
        }
    }
}
```

4. 注册事件

可以通过使用委托把事件与处理这些事件的方法进行绑定。例如：

```
using System;
namespace App3_8_2{
    class Program{
        static void Main(string[] args){
            Publisher pub = new Publisher("清华大学出版社");
            Subscriber zs = new Subscriber();
            pub.OnIssue += new Publisher.IssueHandler(zs.Receive);
            pub.Issue("C# 程序设计与案例教程", "杨树林");     //触发事件
        }
    }
}
```

程序运行的结果为：

```
发行 C# 程序设计与案例教程
订购的图书出版,马上发送给你!
    书名:C# 程序设计与案例教程
    作者:杨树林
    出版社:清华大学出版社
```

3.7.3　案例 3-7　档位模拟

本案例演示如何通过委托来定义事件。运行界面如图 3-9 所示。

图 3-9　档位模拟程序运行界面

【技术要点】

- 新建一个 Windows 应用程序。界面上放置 1 个标签,1 个框架,框架上放置 5 个按钮。各按钮的 Tag 属性按照从左到右的次序设置为 0、1、2、3、4。
- 控件的主要属性设置如表 3-2 所示。

表 3-2　控件的主要属性设置

控　　件	属　　性	值
Form1	Text	事件示例
label1	BorderStyle	Fixed3D
	TextAlign	MiddleLeft
	Text	
groupBox1	Text	
groupBox2	Text	档位模拟
button1	Text	启动
button2	Text	1 档
button3	Text	2 档
button4	Text	3 档
button5	Text	倒档

- 建立一个 ShiftArgs 类作为事件参数类型,用来传递档位数据。
- 建立一个委托类型 OnShiftHandler,用于声明事件。
- 建立一个 Stalls(档位)类,在该类中定义一个事件 OnShift 和一个方法 Shift(),在 Shift()方法中触发事件。
- 在窗体的 Load 事件中创建 Stalls 实例,并绑定事件处理程序为 ChangeGear()。
- 5 个按钮的单击事件处理程序均为 button_Click(),在 button_Click 中调用 Stalls 对象的 Shift()方法,以便引发 OnShift 事件。

【设计步骤】

(1) 在 Visual Studio 下建立一个 Windows 窗体应用程序 App3_8_3。

(2) 打开窗体 Form1,在窗体上添加 2 个 GroupBox,在 groupBox1 上放置 1 个 Label,在 groupBox2 上放置 5 个 Button。

(3) 调整控件位置和大小,并按表 3-1 设置相关属性。

(4) 在项目中添加一个类 BreakArgs,编写代码如下所示:

```
using System;
namespace App3_8_3{
    //建立能够传递档位数值的参数类型
    public class ShiftArgs : EventArgs{
        private int gear;
        public int Gear{
            get { return gear; }
            set { gear = value; }
        }
    }
}
```

(5) 在项目中添加一个类 Stalls,编写代码如下所示:

```
using System;
namespace App3_8_3{
//声明一个委托
public delegate void OnShiftHandler(object sender, ShiftArgs e);
//建立一个类,该类声明了一个事件
  public class Stalls{
    public event OnShiftHandler OnShift;
    public void Shift(object sender, ShiftArgs e){
        if (OnShift != null){
            OnShift(sender, e);
        }
    }
  }
}
```

(6) 打开窗体文件,按 F7 键切换到【代码】视图,添加如下代码:

```
private Stalls stalls;                          //档位
private void button_Click(object sender, System.EventArgs e){
    ShiftArgs shiftArgs = new ShiftArgs();    //建 BreakArgs 实例并传入档位数值
    shiftArgs.Gear = Convert.ToInt32(((Button)sender).Tag);
    stalls.Shift(sender, shiftArgs);          //引发事件
}
private void ChangeGear(object sender, ShiftArgs e){
    switch (e.Gear){
        case 0:
            label1.Text = "您启动了汽车,时速 10";
            break;
        case 1:
            label1.Text = "当前档位: 1 档,时速 20";
            break;
        case 2:
            label1.Text = "当前档位: 2 档,时速 40";
            break;
        case 3:
            label1.Text = "当前档位: 3 档,时速 80";
            break;
        case 4:
            label1.Text = "当前档位: 倒档,时速 - 5";
```

```
            break;
        }
    }
```

(7) 打开窗体,为窗体添加 Load 事件,并编写代码如下所示:

```
private void Form1_Load(object sender, EventArgs e){
    stalls = new Stalls();                  //建立档位实例
    stalls.OnShift += new OnShiftHandler(ChangeGear);    //绑定事件处理程序
}
```

(8) 为 5 个按钮添加 Click 事件,事件处理代码均设置为 button_Click。

(9) 运行程序。

本章小结

面向对象的程序设计的基本特征是抽象、封装、继承和多态。类和对象是它的核心和本质。对象代表现实世界中可以明确标识的任何事物。类是具有相同属性和方法的一组对象的集合,它为属于该类的所有对象提供了统一的抽象描述。类是对象的模板,对象是类的实例。

类是组成 C# 程序的基本要素,它是描述一类对象的共同属性(数据成员)和行为方法(方法成员)的封装体,是对象的原型。创建一个新的类就是创建一个新的数据类型。类使用 class 关键字定义。

对于非嵌套类(不是在一个类中的子类),只有两种可访问性,即 internal 和 public。对于嵌套类,访问属性还可以是 protected 和 private,这和类的成员访问属性类似。类成员的可访问性有 public、protected、internal、internal protected 和 private。

实例成员属于类的实例,只能通过对象来访问。类成员属于类,通过类名直接来访问。加 static 修饰的成员为静态成员,也称为类成员,否则为实例成员。类成员包含类变量和类方法。

继承是面向对象程序设计的一个重要特征,它允许在现有类的基础上创建新类,新类从现有类中继承类的成员,而且可以重新定义或加进新的成员,从而形成类的层次或等级。一般称被继承的类为基类、父类或超类,而继承后产生的新类为派生类或子类。

多态性是指不同的对象收到相同的消息时会产生不同动作。多态性是通过方法覆盖实现的。如果下属类的某个方法是覆盖方法,那么当通过父类变量或接口变量(已经存放了下属类对象)调用该方法时,就可以在运行时根据实际存储的下属类对象决定调用谁的成员方法。

委托是封装了一系列方法引用的类。委托简化了多态性的实现,它允许程序员指定日后定义的方法调用。

习题 3

问答题

3-1 什么是类? 类的基本成员有哪几种?

3-2 C# 的访问控制符有哪些? 它们对类成员分别有哪些访问控制限制作用?

3-3　什么是构造函数？构造函数有什么作用？在C#中构造函数的调用顺序如何？

3-4　类的静态成员和非静态成员有什么区别？

3-5　什么是继承？

3-6　什么是抽象类？什么是接口？接口与抽象类有什么不同？

3-7　什么是多态性？Java是如何实现多态的？多态性的作用是什么？

3-8　什么是委托？委托和事件有什么关系？

3-9　定义一个点类Point，要求重载构造函数，并能够求两点间距离。

3-10　定义一个雇员类Employee。雇员类中包含三个数据成员：id、name和employeeCount。employeeCount用于存储雇员数，建立对象时自动增值。要求重载构造函数，定义相应的属性、实例方法和类方法。

3-11　定义一个车辆(Vehicle)类，具有Speed(速度)、MaxSpeed(最大速度)、Weight(重量)等属性，以及Run()、Stop()等方法。然后以该类为基类，派生出Bicycle(自行车)、Car(轿车)等类，并编程对派生类的功能进行测试。