第5章 C#面向对象核心语法

学习目的与要求

C#的类模型是建立在.NET虚拟对象系统上的，对象模型不再是编程语言的一部分，而是基础架构的一部分。在本章中，我们将学习面向对象编程的基础知识，通过本章的学习，读者将熟悉面向对象编程的相关概念和核心语法。

本章主要内容

- 对象和类。
- 构造方法和析构方法。
- 封装、继承和多态。

5.1 面向对象的概念

5.1.1 对象和类概述

在用面向对象编程(Object Oriented Programming，OOP)技术开发的系统中，对象是基本的组成部分。对象可以是具体的物理实体，也可以是人为定义的概念。

在程序设计阶段，应清楚所要解决的问题中有多少实体，每一个实体具有哪些属性，以及各个实体之间的联系等，然后把具有相同属性和行为的实体划分为一个“类”，并明确每个类的基本属性和行为方法。

从编程的角度来说，类是一种数据结构，也可以说是一种数据类型。而此时每个对象就是某一个类的一个实例，即对象是从类中创建出来的。

5.1.2 类、方法和变量

1. 类的定义

在 C# 中使用关键字 class 定义类，类的定义格式如下：

```
[访问修饰符]class<类名>
{
    类的主体部分
}
```

说明：

① 访问修饰符种类将在下面详细介绍，在以这种方式声明的类中只能使用访问修饰符中的 public 和 internal，默认情况下类声明为 internal。

② 这些声明类的修饰符更常被用于声明类成员，下面将详细介绍。

③ 类的主体部分包括成员变量和成员方法。

2. 类的成员变量的定义

类的成员变量的定义格式如下：

```
[访问修饰符]  数据类型  <成员变量名>
```

以下代码表示在 Person 类的定义中声明成员变量：

```
1.    public class Person
2.    {
3.       private string name;
4.       private int age;
5.    }
```

示例说明：

Person 类中包含两个成员变量 name 和 age。

3. 类的成员方法的定义

类的成员方法的定义格式如下：

```
[访问修饰符]  返回值类型  <方法名>([参数列表])
{
    成员方法主体
}
```

【示例 5-1】 以下代码在 Person 类的定义中声明成员方法。

```
1.    public class Person
2.    {
3.        private string name;
4.        private int age;
5.        public void PrintName()
6.        {
```

```
7.            Console.WriteLine ("姓名:"+name);
8.        }
9.        public void PrintAge()
10.       {
11.           Console.WriteLine ("年龄:"+age);
12.       }
13.   }
```

示例说明:

Person类中包含两个成员变量name和age,包含两个成员方法PrintName和PrintAge。

4. 对象的创建

为了从类中产生对象,必须建立类对象的实例,C#中可以使用new操作符建立一个类的新实例。

以下代码创建了一个Person类的对象onePerson:

```
Person onePerson=new Person();
```

示例说明:

此行代码有效的前提是Person类已被先行声明。

5. 访问修饰符

访问修饰符用于控制类中的成员变量和成员方法的访问权限,可以保证其中的数据不被随意访问和修改,C#提供了以下几种访问修饰符:

- private:类中的成员变量和方法只能在这个类中访问,外部无法访问。
- protected:类中的成员变量和方法除了可被本身访问外,还可被该类的派生类访问。
- public:类中的成员变量和方法可被任何外部类访问。
- internal:类中的成员变量和方法在整个项目中都可以被访问,项目外的其他代码不能访问。

当上述修饰符用来声明类时,其含义如下:

- private/protected:私有或保护访问限制的类不能在类外被访问,很少使用。
- public:类可以在任何地方访问。
- internal:类只能在当前项目中访问。

注意:在类声明时如果不指定访问修饰符,则默认的访问修饰符为internal,但类中成员变量和方法的默认访问修饰符为private。

【示例5-2】 以下对比说明private和public对类成员访问权限的控制作用。

```
1.    class Person
2.    {
3.        private string name;
4.        public int age;
5.     }
6.    class Test
```

```
7.    {
8.         static void Main(string[] args)
9.         {
10.             Person onePerson=new Person();
11.             onePerson.name="张明";
12.             onePerson.age=20;
13.         }
14.   }
```

示例说明：

第 11 行代码会出错，因为作为 Person 类的成员变量 name，其声明时为私有访问权限 private，是不能在 Person 类外被访问的；相反，第 12 行则不会出错，因成员变量 age 的访问权限是 public，可以在其所属类 Person 的外部被访问。

5.1.3 构造方法和析构方法

1. 构造方法

构造方法通常又称为构造函数，是类的一个特殊的成员方法。

构造函数用来完成类成员变量的自动初始化，C# 中每次创建类的实例时都会调用类中的构造函数。

如果一个类中不包含任何构造函数的声明，系统就会自动提供一个默认的构造函数，这个默认的构造函数没有任何参数；反之，如果为类编写了构造函数（有参数的），那么 C# 将不会为该类提供默认的构造函数，所以，此时如果想不使用任何参数来创建类实例，那么必须在类中声明默认的构造函数。

构造函数具有如下特点：

- 构造函数与类同名。
- 构造函数不返回任何值。

定义构造函数的格式如下：

```
[访问修饰符]  类名()
{
    构造函数主体部分
}
```

说明：

构造函数主体部分主要包括对成员变量的初始化语句。

【示例 5-3】 以下代码在定义的 Person 类中声明了默认构造函数，也编写了一个带参数的构造函数。

```
1.    public class Person
2.    {
3.        private string name;
4.        private int age;
5.        public Person()
```

```
6.        {
7.        }
8.        public Person(string x1,int x2)
9.        {
10.           name=x1;
11.           age=x2;
12.       }
13.   }
```

2. 析构方法

析构方法通常又称为析构函数,是类成员方法中的另一个特殊方法,它由带有“~”前缀的类名来声明,其作用与构造函数相反,用于进行清除操作以释放资源。

析构函数具有如下特点:

- 一个类只能有一个析构函数。
- 析构函数没有访问修饰符,不带任何参数。
- 析构函数不能被显式调用,它由系统自动调用。

定义析构函数的格式如下:

```
~类名()
{
    析构函数主体部分
}
```

示例说明:

通常析构函数的主体部分包含关闭与数据库、文件、网络等资源连接的语句。

以下代码在 Person 类中声明了析构函数:

```
1.    public class Person
2.    {
3.        ~Person()
4.         {
5.            //Destructor body
6.         }
7.    }
```

5.1.4 方法重载

方法重载是指在同一个类中,存在方法名相同但参数类型或参数个数不全相同的多个方法。即实现方法重载有两种形式:参数个数不同的方法重载和参数类型不同的方法重载。

在示例 5-3 的代码中第 5 行和第 8 行,分别声明了同名的方法 Person(),这两个同名不同参数的方法就是被重载了的构造方法。

【示例 5-4】 以下代码中编写了 Main 函数,用来说明被重载的构造方法的使用。

```
1.    public class Person
```

```
2.    {
3.         private string name;
4.         private int age;
5.         public Person(){name="zhangsan"; age=0;}
6.         public Person(string x1,int x2){ name=x1; age=x2;}
7.         public void PrintName(){Console.WriteLine("姓名:"+name);}
8.         public void PrintAge(){Console.WriteLine("年龄:"+age);}
9.    }
10.   class Program
11.   {
12.        static void Main(string[] args)
13.        {
14.             Person onePerson1=new Person();
15.             Person onePerson2=new Person("张明",21);
16.             onePerson1.PrintName();
17.             onePerson1.PrintAge();
18.             onePerson2.PrintName();
19.             onePerson2.PrintAge();
20.             Console.Read();
21.        }
22.   }
```

执行结果：

```
姓名：zhangsan
年龄：0
姓名：张明
年龄：21
```

【示例 5-5】 参数个数不同的方法重载问题。

```
1.    public class OverloadTest1
2.    {
3.         public double CalArea(double x,double y)
4.         {
5.              return x * y;
6.         }
7.         public double CalArea(double x,double y,double z)
8.         {
9.               return 2 * (x * y+y * z+z * x);
10.         }
11.   }
```

【示例 5-6】 参数类型不同的方法重载问题。

```
1.    public class OverloadTest2
2.    {
3.         public int add(int x,int y)
```

```
4.        {
5.             return x+y;
6.        }
7.        public string add(string str1,string str2)
8.        {
9.             return str1+str2;
10.       }
11.   }
```

5.1.5 this的使用

关键字this有两种基本的用法：一是用在类成员的内部,用来引用对象实例;二是在声明构造函数时指定需要先执行的构造函数。

在这里说明this的第一种用法,第二种用法将在5.3.2节中对比base关键字进行具体说明。

【示例5-7】 在示例5-4中,类定义中访问类的成员变量时,变量名前都省略了this,以下是添加了可省略的this后的代码。

```
1.    class Person
2.    {
3.        private string name;
4.        private int age;
5.        public Person()
6.        {
7.            this.name="zhangsan";
8.            this.age=0;
9.        }
10.       public Person(string x1,int x2)
11.       {
12.           this.name=x1;
13.           this.age=x2;
14.       }
15.       public void PrintName()
16.       {
17.           Console.WriteLine("姓名:"+this.name);
18.       }
19.       public void PrintAge()
20.       {
21.           Console.WriteLine("年龄:"+this.age);
22.       }
23.   }
24.   class Program
25.   {
26.       static void Main(string[] args)
27.       {
```

```
28.             Person onePerson1=new Person();
29.             Person onePerson2=new Person("张明",21);
30.             onePerson1.PrintName();
31.             onePerson1.PrintAge();
32.             onePerson2.PrintName();
33.             onePerson2.PrintAge();
34.             Console.Read();
35.         }
36.     }
```

示例说明：

① 第 7、8、12、13 行都是在类的构造函数中使用了 this，在第 28 行代码创建 onePerson1 时，第 7、8 行代码被执行，所以这其中的 this 引用了当前类 Person 的当前对象实例 onePerson1；同样在第 29 行代码创建 onePerson2 时，第 12、13 行代码被执行，所以这其中的 this 引用了当前类 Person 的当前对象实例 onePerson2。

② 第 17 行和第 21 行的 this 分别出现在类的成员方法 PrintName 和 PrintAge 的函数体内，在第 30～33 行的主函数代码中，这两个成员方法被分别调用了两次，其中第 30、31 行代码执行时的 this 引用了当前对象实例 onePerson1，第 32、33 行代码执行时的 this 引用了当前对象实例 onePerson2。

执行结果：

```
姓名：zhangsan
年龄：0
姓名：张明
年龄：21
```

以上代码中的 this 是不必要加的，在此添加是为了说明 this 的作用。因此，若是在代码段中写出了此类 this，其目的多是为了提高程序的可读性。

【示例 5-8】 以下代码段用来说明不可省略 this 的使用。

```
1.    class Person
2.    {
3.         private string name;
4.         private int age;
5.         private string sex;
6.         private double weight;
7.         public void TakeExerciseWith(Person someone)
8.        {
9.              Console.WriteLine("姓名:"+someone.name);
10.             Console.WriteLine("年龄:"+someone.age);
11.             Console.WriteLine("性别:"+someone.sex);
12.             Console.WriteLine("体重:"+someone.weight);
13.        }
14.        public void takeExercise()
15.        {
```

```
16.            Person onePerson=new Person();
17.            onePerson.TakeExerciseWith(this);
18.        }
19.    }
```

示例说明：

① 第17行中作为参数出现的this，把当前对象实例onePerson的引用传递给方法TakeExerciseWith()。

② 测试本程序，最好添加写出带参构造方法。

另外，在声明索引器时，this关键字代表索引器名出现，可以直观理解为对象的引用，具体参见索引器相关介绍。

5.1.6 命名空间

命名空间(Namespace)是面向对象编程为了避免类名冲突而使用的一种数据类型组织方式。

随着应用程序的模块化，每个人各自编写自己的模块，各自使用自己对类型的命名习惯。为了区别不同编程者定义中可能出现的相同类名，可以将它们放在不同的命名空间中，使用时在类名前加上空间的名字就可以了。

1. 新建命名空间

创建命名空间的基本格式如下：

```
namespace <空间名 1>
{
    类定义等声明模块
}
```

以下代码新建了一个命名空间MyNameSpace：

```
1.    namespace MyNameSpace
2.    {
3.        class A {}
4.        class B {}
5.    }
```

以下代码新建了一个含嵌套空间的命名空间YourNameSpace，其内嵌套的命名空间为SubNameSpace。

```
1.    namespace YourNameSpace
2.    {
3.        class A{}
4.        namespace SubNameSpace
5.        {
6.            class B{}
7.        }
8.    }
```

以下代码新建了两个命名空间 YourNameSpace 和 SubNameSpace，其功能与上例实现的功能完全相同。

```
1.    namespace YourNameSpace
2.    {
3.         class A{}
4.    }
5.    namespace YourNameSpace.SubNameSpace
6.    {
7.          class B{}
8.    }
```

2. 使用命名空间

使用命名空间的数据类型时通常有两种形式：

① 使用包含命名空间名称的完整类型名，例如：

```
YourNameSpace.SubNameSpace.B  ObjectB=new YourNameSpace.SubNameSpace.B();
```

② 使用 using 关键字把命名空间添加到当前命名空间中，例如：

```
using YourNameSpace.SubNameSpace;
B ObjectB=new B();
```

或者

```
using YourNameSpace;
using SubNameSpace;
B ObjectB=new B();
```

显然使用第二种形式可以使代码更易读，但有时会出现二义性，例如：

```
using MyNameSpace;
using YourNameSpace;
using SubNameSpace;
A ObjectA=new A();
B ObjectB=new B();
```

此时使用的类名 A 和类名 B，均出现了二义性，即出现了 MyNameSpace. A 和 YourNameSpace. A 之间的不明确引用，以及 MyNameSpace. B 和 YourNameSpace. SubNameSpace. B 之间的不明确引用。

修改如下：

```
using MyNameSpace;
using YourNameSpace;
MyNameSpace.A ObjectA1=new MyNameSpace.A();
YourNameSpace.A ObjectA2=new YourNameSpace.A();
MyNameSpace.B ObjectB1=new MyNameSpace.B();
SubNameSpace.B ObjectB2=new SubNameSpace.B();
```