第3章 面向对象编程

学完基础语法，就具备了一定的“面向过程”编程能力。简单项目用面向过程的方式就可以了。但复杂项目，问题就突显出来了——代码很难调试、维护和阅读。

为此，针对复杂项目的要求，人们逐步总结出一套分析和解决编程问题的方法——面向对象编程(Object-Oriented Programming，OOP)模式。

面向对象编程不是高深的理论，同时也不是放弃面向过程。简单理解一下，面向对象编程是一种通过对象的方式把现实世界映射到计算机模型的一种编程方法。

如何学习面向对象编程？

(1) 对面向对象编程的基础概念有一定的理解。

(2) 学习编写基本的面向对象结构：类、对象(又称为实例)、方法。

(3) 进一步理解面向对象的三大特征：封装、继承、多态。

(4) 在实践中，逐步掌握面向对象的代码编写规则，进而学会复杂项目的面向对象开发技能。

3.1 面向对象编程基础

对象就是现实中的实体；类就是现实中的分类。

例如，现在要实现一个通讯录。班里有章珊、李思等同学，同学就是类，而章珊和李思就是对象。

从通讯录需求出发，需要章珊和李思等实体提供具体属性信息：姓名、性别、住址、手机号。此外，章珊、李思的住址和手机号会发生变化，这相当于实体的行为，该行为能引起住址和手机号属性值的变化。

章珊实体可标记为：

```
同学  章珊{ 姓名："章珊"; 性别："女"; 住址："北京市海淀区双清路 30 号"; 手机号："13901820560"; 改地址(新地址); 改手机(新手机号); }
```

李思实体可标记为：

```
同学  李思{ 姓名："李思"; 性别："男"; 住址:"上海市杨浦区邯郸路 220 号"; 手机号："13724929023"; 改地址(新地址); 改手机(新手机号); }
```

这里的姓名、性别、住址、手机号就可看成对象的属性，其具体信息就是对象的属性值。

同学章珊和同学李思,显然同属类别“同学”,这样一个同学类别就形成了,即

```
类别  同学{ 姓名; 性别; 住址; 手机号; 改地址(新地址); 改手机(新手机号); }
```


3.1.1 第一个类的定义和对象

通过对通讯录的分析,现实世界的实体章珊和李思,抽象为同学这种类别。那么,如何映射到C#面向对象编程世界来表达同学分类?可定义同学类结构 class Mate。

【例 3-1】 定义同学类结构。

```
class Mate                                  //分类: 同学
{
  public String name;                       //成员: 属性姓名(实为成员变量,C#属性后续讲解)
  public char sex;                          //成员: 属性性别
  public String addr;                       //成员: 属性住址
  public String mobile;                     //成员: 属性手机号
  public void ChangeAddr(String newAddr)    //成员: 行为方法——改地址
  {
    addr = newAddr;
  }
  public void ChangeMobile(String newMobile)  //成员: 行为方法——改手机号
  {
    mobile = newMobile;
  }
}
```

对象是类的实例,如何得到C#的章珊和李思?创建同学类的对象就可。

【例 3-2】 创建同学类的对象。

```
Mate zhangSan = new Mate(); //zhangSan 是定义 Mate 类型的变量
Mate liSi = new Mate();
```

如何将属性值赋予章珊和李思?用成员操作符“.”获取属性后,进行赋值即可。

【例 3-3】 对章珊和李思对象的属性进行赋值。

```
zhangSan.name = "章珊";
zhangSan.sex = '女';
zhangSan.addr = "北京市海淀区双清路 30 号";
zhangSan.mobile = "13901820560";
liSi.name = "李思";
liSi.sex = '男';
liSi.addr = "上海市杨浦区邯郸路 220 号";
liSi.mobile = "13724929023";
```

如何调用章珊和李思对象的行为方法?同样使用成员操作符“.”调用即可。

【例 3-4】 调用章珊和李思对象的行为方法。

```
zhangSan.ChangeAddr("北京市海淀区双清路 30 号");
zhangSan.ChangeMobile("13901820560");
liSi.ChangeAddr("上海市杨浦区邯郸路 220 号");
liSi.ChangeMobile("13724929023");
```

通过上面通讯录的例子可以看出：类是对现实中存在对象的描述，同属相同类的对象都具有共同的属性和行为。但是，根据不同的系统需求，同样的一种对象会被描述成具有不同属性和行为的类。例如客户类，对于银行系统，客户类应该具有账号、账户余额的属性和存钱、取钱的行为；而对于电信系统，客户类应该具有手机号、卡内余额的属性和充费、扣费的行为。因此，编写代码时要注意确认类的定义和它所封装的行为是否能够正确地反映出实际系统的需求。

3.1.2 类的成员变量

成员变量又称为字段或实例变量，它定义在类中，用于描述对象的特征。描述类的静态特征的成员变量称为静态变量。实例变量必须先创建对象方能使用；静态变量无须创建对象，通过类名直接调用。

实例变量定义格式：

```
[修饰符] 类型 成员变量名 [ = 默认值];
```

静态变量定义格式：

```
[修饰符] static 类型 成员变量名 [ = 默认值];
```

【例 3-5】 实例变量示例。

```
class Student                          //定义 Student 类,内部为不同类型实例变量
{
    public string name;                //实例成员：姓名
    public int number;                 //实例成员：学号
    public char sex;                   //实例成员：性别
    public double height;              //实例成员：身高
    public bool onCampus;              //实例成员：在校否
}
```

运行代码：

```
Student zs = new Student();            //创建对象
//实例成员,创建对象后方能调用
Console.WriteLine(zs.name);            //默认值为 null
Console.WriteLine(zs.number);          //默认值为 0
Console.WriteLine(zs.sex);             //默认值不见,实际为\u0000
Console.WriteLine(zs.height);          //默认值为 0.0
Console.WriteLine(zs.onCampus);        //默认值为 false
```

结果：

```
            //null, 控制台不见
0
            //\u0000, 控制台不见
0.0
false
```

可见，在定义成员变量时，若不进行初始化，C#使用默认的值对其初始化。初始化值可参考表2-1。若是引用类型，则值为null。

【例3-6】 静态变量示例。

```
class Mathematics                              //定义 Mathematics 类,内部为静态变量
{
    public static double PI = 3.14;            //静态成员
}
```

静态成员通过“类名.成员”方式就可直接调用：

```
Console.WriteLine(Mathematics.PI);
```

结果：

```
3.14
```

局部变量和成员变量的区别：不同于成员变量，局部变量就是放在方法内部的变量。当然，方法参数出现在方法内部，所以也是局部变量。局部变量在方法调用时创建；方法调用结束时，局部变量生命周期结束，就不可再调用了。

3.1.3 类的属性

前面学习成员变量时，可能发现，成员变量(字段)赋值时不能被有效控制。如学生的身高height被赋值－1.8，显然逻辑不当，但语法能通过，最后会造成后期计算平均身高不精确等问题。对此，C#中使用属性(property)来处理。

属性可认为是成员变量的扩展，使用访问器(accessors)让私有成员变量的值可被外部代码进行读(get)、写(set)操作。

属性语法格式：

```
public 属性类型 属性名
{
     get { 返回属性值 }
     set { 设置隐式参数 value 给属性值 }
}
```

若无额外逻辑，可简写为自动属性，即去除“{ }”，直接在set和get后加分号“;”结束，语法格式如下：

```
public 属性类型 属性名 { get; set; }
```

【例 3-7】 属性定义及使用示例。

```
class Emp
{
    public string Name { get; set; }                    //自动属性
    private int _Age;                                   //成员变量
    public int Age                                      //读写属性
    {
        get                                             //读属性
        {
            return _Age;
        }
        set                                             //写属性,含验证逻辑
        {
          if (value <= 0 || value >= 120)
          {
            throw new ApplicationException("年龄值不符合范围");
          }
          else
            _Age = value;
        }
    }
}
```

使用属性,如下:

```
Emp emp = new Emp();
emp.Name = "章珊";                          //调用属性 set
Console.WriteLine(emp.Name);                //调用属性 get
emp.Age = -1;                               //会抛出 ApplicationException 异常,然后程序中断
```

如上最后一行代码 emp.Age=-1 执行时会抛出“年龄值不符合范围”异常,系统阻止了赋值。有关异常,后续会系统学习。在实际项目中,若发生以上异常,会尝试让用户再次输入 Age 属性值,程序将继续运行。

3.1.4 类的成员方法

成员方法简称方法。

方法类似于面向过程中的函数。面向过程中,函数是复用代码的最基本单位。通过函数间调用,组成了程序。面向对象的基本单位是类,方法是定义在类中的。

与成员变量类似,成员方法也分为实例和静态两种。实例方法属于对象的,必须创建对象后使用;静态方法属于类的,通过类名加成员符直接调用。

实例方法定义格式:

```
[修饰符] 方法返回类型 方法名([形参列表])
{
```

```
    方法体语句
}
```


静态方法定义格式：

```
[修饰符] static 方法返回类型 方法名( [形参列表] )
{
    方法体语句
}
```

【例 3-8】 定义 Circle 类，调用其中静态方法和实例方法。

```
class Circle
{
    public double radius;                                    //实例方法
    public static void WhoAmI()                              //静态方法
    {
        Console.WriteLine("Circle Class");
    }
    public double GetArea()
    {
        return Math.PI * radius * radius;
    }
}
```

静态方法和实例方法调用：

```
Circle.WhoAmI();                          //静态方法,属于类的方法,通过类名.直接调用
Circle c = new Circle();
c.radius = 2;
double area = c.GetArea();                //实例方法,属于对象的方法,创建对象后调用
Console.WriteLine(area);
```

结果：

```
Circle Class
12.566370614359172
```

3.1.5 类的构造方法

用于创建对象的特殊方法称为构造方法，又称为构造器(constructor)，简称构造。C#通过 new 关键字调用构造方法，创建出相应类的对象。

构造方法的名称应与类名一致，且无返回。

此外，若没有定义构造方法，编译器会自动添加一个无参构造。当然，若定义了构造，则编译器将不再添加无参构造。

构造方法定义格式：

```
[修饰符] 类名([形参列表])
{
    构造方法体语句
}
```

【例 3-9】 没有定义构造方法，编译器会自动添加一个无参构造。

```
public class User
{
    String name;
    String pass;
}
```

创建 User 对象：

```
User u = new User();      //没有问题，编译器已自动生成一个无参构造
```

【例 3-10】 定义构造方法并调用。

```
public class Customer
{
    String name;
    double balance;
    public Customer(String name, double balance)
    {
        //当成员变量名和参数名冲突时，用 this.区分
        this.name = name;
        this.balance = balance;
    }
}
```

创建 Customer 对象：

```
Customer cindy = new Customer("Cindy", 0);      //调用构造
//下方代码编译报错，已定义构造，系统不再生成无参构造
Customer customer = new Customer();
```

有时需要在一个构造方法中调用另一个构造，可以使代码更简洁美观、更易于阅读与维护。此时可使用 this 关键字。

【例 3-11】 使用 this 关键字进行构造之间的调用。

```
class Customer
{
    String name;
    double balance;
    String level;
```

```
    public Customer(String name, double balance)
    {
        this.name = name;
        this.balance = balance;
    }
    //用 this()调用另一个构造
    public Customer(String name, double balance, String level): this(name, balance)
    {
        this.level = level;
    }
}
```

【总结】 构造是用来创建对象用的特殊方法。当构造带参数时,一般用于同时初始化属性值。构造名称与类名相同,没有返回值。不写构造时,系统会生成一个构造。构造可以带参数,为此可写多个构造,构造间用 this 调用。

3.1.6 方法的重载

构造是特殊方法,构造可以写多个。若构造的名称相同,参数不同,此时构造间就是重载(overload)的关系。

在一个类中,可以定义多个方法,若方法名相同,参数不同(即参数的个数不同或参数类型顺序不同),则这些方法间就是重载关系。

【例 3-12】 两个 max()方法的参数类型不同形成重载。

```
public class Tools
{
    public double max(double a, double b)
    {
        return a > b ? a : b;
    }
    public int max(int a, int b)
    {
        return a > b ? a : b;
    }
}
```

3.1.7 继承

继承是为了实现类的扩展。C#类是单继承的,即只能有一个父类。

继承的语法结构:

```
[修饰符] 子类 : 父类
{
    类结构体(成员变量,成员方法)
}
```

父类(parent class)又称为基类(base class)、超类(super class)。子类(sub class)又称为派生类(derived class)。

若类没有继承父类,但实际上继承了 Object 这个默认父类,Object 类可认为是所有类的超级父类。编写代码时也可用关键字 object 来替代类名 Object。

【例 3-13】 Shape 父类和 Square 子类示例 1。

```
class Shape                                  //实际上编译器会加上:Object,默认继承 Object 类
{
    public void WhoAmI()
    {
        Console.WriteLine("a shape");
    }
}
class Square : Shape                         //继承了 Shape 中的实例方法 WhoAmI()
{
    public double width;
    public double GetArea()
    {
        return width * width;
    }
}
```

Square 通过“Shape”继承了 Shape 中的成员,包括 WhoAmI()方法。

因此执行:

```
Square s = new Square();
s.WhoAmI();
```

会显示:

```
a shape
```

所以,继承实际上也是代码复用的一种表现形式。

再如:

```
Square s = new Square();
s.width = 3;
Console.WriteLine(s.GetArea());
```

会显示:

```
9
```

即子类可以通过增加方法来扩展父类的结构功能。

sealed(密封)关键字可阻止类派生子类。System 中的 String 类就是 sealed 类。

【例 3-14】 sealed 类阻止被派生。

```
class B1{ }
class E1 : B1 { }                          //正常派生
sealed class B2 { }                        //sealed 类
class E2 : B2 { }                          //编译时出错,无法从密封类 B2 派生
```

3.1.8 方法覆盖、多态、转型

继承时,子类定义了与父类方法签名完全相同的方法(相同名称及相同参数形式),被称为方法的覆盖(override)或重写。

【例 3-15】 Shape 父类和 Square 子类示例 2。

```
class Shape
{
    public virtual void WhoAmI()
    {
        Console.WriteLine("a shape");
    }
}
class Square : Shape
{
    public override void WhoAmI()      //重写父类方法
    {
        Console.WriteLine("a square");
    }
}
```

如上,子类 Square 继承了父类 Shape 中的成员方法 WhoAmI(),但在自己的类结构体中进行了重写。注意,父类方法前用关键字 virtual,子类方法前用关键字 override。

测试代码:

```
Shape shape = new Square();
shape.WhoAmI();
```

结果:

```
a square
```

以上 shape 变量是 Shape 父类类型,实际创建为 Square 对象。当 shape.WhoAmI()调用时,调用的是父类中的 WhoAmI()还是子类重写的 WhoAmI()? 从运行结果看调用的是子类重写的 WhoAmI()。

【总结】 C#的实例成员调用是基于运行时的实际类型的动态调用,而非变量的声明类型。这个非常重要的特性即多态(polymorphism)。

要调用父类中被重写的方法,可用关键字 base。实际上通过 base 可调用父类对象中所有成员。

【例 3-16】 Square 子类中调用 Shape 父类中的实例方法。

```
class Shape
{
    public virtual void WhoAmI()
    {
        Console.WriteLine("a shape");
    }
}
class Square : Shape
{
    public override void WhoAmI()                    //重写父类方法
    {
        base.WhoAmI();                               //a shape, 调用父类 Shape 的实例方法
        Console.WriteLine("a square");
    }
}
```

测试代码：

```
Shape shape = new Square();
shape.WhoAmI();
```

结果：

```
a shape
a square
```

注意，上面 base.WhoAmI()就是在子类中调用父类的实例方法 WhoAmI()。

除了阻止派生子类，使用 sealed 关键字还可阻止方法不被覆盖。

【例 3-17】 使用 sealed 使方法无法在子类中覆盖。

```
class B
{
    public virtual void b1() { }
}
class E : B
{
    public sealed override void b1() { }      //这里覆盖时用 sealed 进行密封
}
class F : E
{
    public override void b1() { }             //会报错。因为父类中 E.b1()是密封的,无法重写
}
```

转型(casting)：实际开发中，经常存在父类转换为子类和子类转换为父类的情况。

向上转型：将子类对象转换为父类对象。即父类变量引用子类对象，属于自动类型转换。

向下转型：把父类对象转换为子类对象。即子类变量引用父类对象，需要进行类型的强制转换。

【例 3-18】 父类 Shape 和子类 Circle、Square 之间的转型。

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("draw a shape");
    }
}
class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("draw a circle");
    }
}
class Square : Shape
{
    public override void Draw()
    {
        Console.WriteLine("draw a square");
    }
}
```

代码调用：

```
//向上转型,属于自动类型转换,如
Shape s = new Circle();
//向下转型,需要强制转换,如
Circle c = (Circle)s;              //(Circle)强制将 s 类型(父类 Shape)转换为子类型 Circle
//向下转型,与真实类型不匹配,会产生异常,如
Square sq = (Square)s;             //s 的真实类型 Circle 转换为 Square,会产生异常
```

向下转型时，为了让代码更强壮，可先加上 is 操作符预判，只有符合类型才转型。

【例 3-19】 用 is 操作符预判类型是否可转型。

```
Shape c = new Circle();
if (c is Square)                          //用 is 预判 c 变量类型
{
    Circle circle = (Circle)c;            //向下转型
}
```

向下转型时，建议用 as 操作符进行转型。其好处是：当无法转型时返回 null，不会产生异常而导致程序执行中断。

【例 3-20】 用 as 操作符进行转型，若转型失败则返回 null。

```
Shape c = new Circle();
Circle circle = c as Circle;
Console.WriteLine(circle);
Square square = c as Square;
Console.WriteLine(square); //null(控制台上无显示)
```

3.1.9 抽象类

抽象类是类的一种，同样有成员变量、属性和方法。它与普通类最大的区别，就在于其只能作为基类，无法直接实例化为对象。

抽象类使用关键字 abstract 表示。即 class 前加 abstract，说明是抽象类。

【例 3-21】 抽象类 Shape 含有抽象方法 Draw()。

```
abstract class Shape                    //class 前加 abstract,说明是抽象类
{
    public abstract void Draw();        //方法前加 abstract,说明是抽象方法
}
```

直接创建抽象类对象，编译将通不过，例如：

```
Shape c = new Shape(); //编辑报错:无法创建抽象类型的实例
```

抽象类的本质是用抽象方法定义功能规范，具体功能或业务逻辑由子类来实现。

【例 3-22】 抽象类 Shape 的实现子类 Circle。

```
class Circle : Shape
{
    public override void Draw()
    {
        //覆盖了父类中抽象方法 Draw(),实现其功能逻辑
        Console.WriteLine("draw a circle");
    }
}
```

创建抽象类的具体子类，代码如下(编译将通过)：

```
Shape c = new Circle();
```

抽象的几点理解：

(1) 只要 abstract 关键字在类前，就是抽象类。抽象类中是可以有属性和具体方法的。

(2) 抽象类是不可以直接创建的类。如上面的 Shape 类，进行 new Shape()操作不允许。

(3) 抽象方法是没有实现的方法，有“{ }”存在就是空现，即使方法体中为空，也是不允许的。

(4) 抽象方法存在,则说明类的部分已抽象,所以该类必须标注 abstract 使其成为抽象类。

在抽象类中,有时方法前用 virtual 来标注,则说明该方法是虚方法。不同于抽象方法,虚方法是具体实现的方法,若子类用 override 覆盖该虚方法,则子类调用方法时将调用自己定义的方法,否则就调用父类中的虚方法。

【例 3-23】 父类中的虚方法及子类对虚方法的覆盖。

```
abstract class Shape
{
    public virtual void Draw()                  //虚方法有具体实现
    {
        Console.WriteLine("draw a shape");
    }
}
class Circle : Shape
{
    public override void Draw()                 //覆盖父类中虚方法 Draw(),实现自己的功能
    {
        Console.WriteLine("draw a circle");
    }
}
class Rectangle : Shape
{
    //未覆盖父类中虚方法 Draw(),会继承该虚方法
}
```

调用:

```
Shape c = new Circle();
c.Draw();                          //draw a circle
Shape r = new Rectangle();
r.Draw();                          //draw a shape
```

结果:

```
draw a circle
draw a shape
```

3.1.10 接口

接口可认为是更为抽象的抽象类。接口本身并不做任何功能(方法)实现,仅需声明必须实现哪些功能(方法),具体由派生类实现。

接口中可以定义属性(有 get 和 set 的方法)和抽象方法。接口成员始终是公共的,不能应用任何访问修饰符。一个接口可以同时继承多个接口。

接口的语法结构:

```
[访问修饰符] interface 接口名 [ : 接口 1,接口 2,… ] {
        [属性; ]
```

```
        [静态字段; ]
        [抽象方法; ]
}
```

【例 3-24】 用 interface 定义接口 IfcShape。

```
interface IfcShape
{
        string Name { get; set; }          //属性
        static double PI = 3.14;           //静态字段。实例字段不允许定义在接口中
        void Draw();                       //抽象方法
}
```

将 interface 关键字放在名称前,说明定义的是接口;接口中定义了属性 Name 和静态字段 PI,以及抽象方法 Draw()。注意,此时抽象方法前不用加 abstract。

接口可以看成是定义了功能的契约。若有子类想实现该接口,就必须实现接口中所有的抽象方法(定义的功能)。如果不实现或部分实现,则说明子类是抽象的,必须用关键字 abstract 将其声明为抽象类。

【例 3-25】 Circle 子类实现 IfcShape 接口。

```
class Circle : IfcShape
{
    //这里隐式实现接口中的属性,也可完全重写 set 和 get
    public string Name { get; set; }
    public void Draw()
    {
        Console.WriteLine("draw a circle named " + Name);
    }
}
```

创建接口子类对象,执行:

```
IfcShape s = new Circle();
s.Name = "CircleOne";
s.Draw(); //draw a circle named CircleOne
```

结果:

```
draw a circle named CircleOne
```

子类中使用符号“:”实现接口。实现接口所定义的所有抽象方法,也包括实现属性的 set、get。值得注意的是,上面 Name 属性前用 public 访问修饰符,起到了隐式实现的功能,也可以正式重写属性的 set 和 get 逻辑代码。

C#中,类之间是单继承的,但接口允许是多继承的。

【例 3-26】 接口可继承多个接口。

```
interface IfcFlyable { void Fly(); }
interface IfcSingable { void Sing(); }
interface IfcBird : IfcFlyable, IfcSingable { }
```

如上,IfcBird 接口继承了 IfcFlyable 和 IfcSingable 两个接口。若父接口中有抽象方法,则子接口也将继承到。若有类要实现该子接口,则应实现其父接口中所有的抽象方法。

【例 3-27】 子接口的实现类,必须实现所有抽象方法。

```
class Swan : IfcBird
{
    public void Fly()                    //必须实现,Fly()在 IfcBird 的父接口中定义
    {
        Console.WriteLine("swan flying");
    }
    public void Sing()                   //必须实现,Sing()在 IfcBird 的父接口中定义
    {
        Console.WriteLine("swan singing");
    }
}
```

【总结】 C#中,子类只能继承一个父类,但可以实现多个接口。

【例 3-28】 Flamingo 子类继承 Animal 类实现 Flyable 和 Walkable 接口。

```
class Animal { }
interface Flyable { }
interface Walkable { }
class Flamingo : Animal, Flyable, Walkable{ }
```

3.2 面向对象编程进阶

面向对象语法中,还包括名称空间、程序集、内部类、Lambda 表达式、异常处理等内容。

3.2.1 名称空间、程序集

名称空间(namespace)是一种代码逻辑分组,用以解决类、接口等资源的名称冲突。可将名称冲突的类或者接口放在不同名称空间中,若有名称冲突时,用完整名称(名称空间名.类名)加以区分。

程序集(assembly)是程序的物理分组。将项目中类、接口等资源打包为程序集,对应一个.dll 或.exe 文件。开发中可引用程序集,代码中使用 namesapce 关键字引入程序集名称空间,然后就可调用程序集内部的类、接口。

【例 3-29】 设计类库项目并生成程序集文件,引用程序集并调用其内部类。

实现的目标:学会创建程序集和引用程序集。

实现的步骤:

(1) 创建程序集。

本处创建的程序集是一个类库(Tools.dll 文件),内部创建一个 Math 类。具体步骤如下。

① 启动 Visual Studio。

② 选择“文件”→“新建”→“项目”选项,弹出“创建新项目”窗口。

③ “语言”选择 C#,“项目类型”选择“库”,在列表中选择“类库”选项,单击“下一步”按钮,弹出“配置新项目”窗口。创建“类库”项目界面如图 3-1 所示。

图 3-1 创建“类库”项目界面

④ 在“项目名称”文件框中输入 Tools,单击“下一步”按钮,单击“创建”按钮。打开 Visual Studio 开发“类库”项目界面如图 3-2 所示。

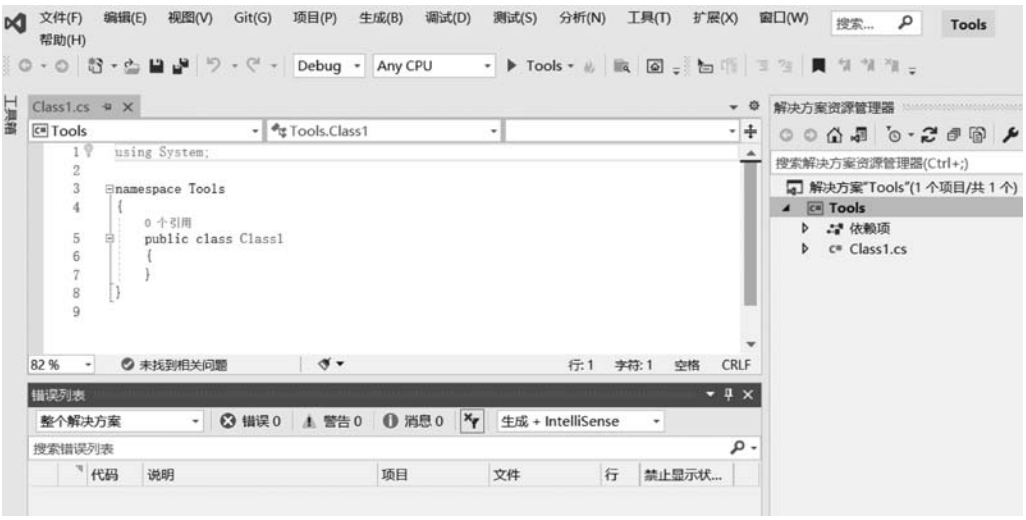

图 3-2 开发“类库”项目界面

⑤ 在“解决方案资源管理器”窗口中,右击项目 Tools,在弹出的快捷菜单中选择“添加”→“类”选项,在弹出的对话框中设置名称为 Math.cs,单击“添加”按钮。在生成的 Math 类中编写如下代码:

```
namespace Tools
{
    public class Math
    {
        public static int Add(int a, int b)
        {
            return a + b;
        }
        public static int Substract(int a, int b)
        {
            return a - b;
        }
    }
}
```

⑥ 右击项目 Tools,在弹出的快捷菜单中选择“生成”选项,会生成程序集 Tools.dll 文件。

⑦ 右击项目 Tools,在弹出的快捷菜单中选择“文件资源管理器中打开文件夹”选项,打开 bin/Debug/net5.0/子目录,可看到相应程序集文件 Tools.dll,如图 3-3 所示。可将该文件保存到其他文件夹中,如 C:\dlls 中。

图 3-3 生成了程序集文件 Tools.dll

(2) 引用程序集。

可通过“添加项目引用”引用程序集,然后在代码中调用程序集中的类。具体步骤如下。

① 创建控制台应用 ConsoleAppUseAssambly,具体过程参见 1.5.1 节。

② 右击项目 ConsoleAppUseAssambly 下方的“依赖项”,在弹出的快捷菜单中选择“添加项目引用”选项,如图 3-4 所示。

图 3-4 添加项目引用

③ 在弹出的“引用管理器”对话框中，通过“浏览”按钮添加程序集 Tools.dll，如图 3-5 所示。

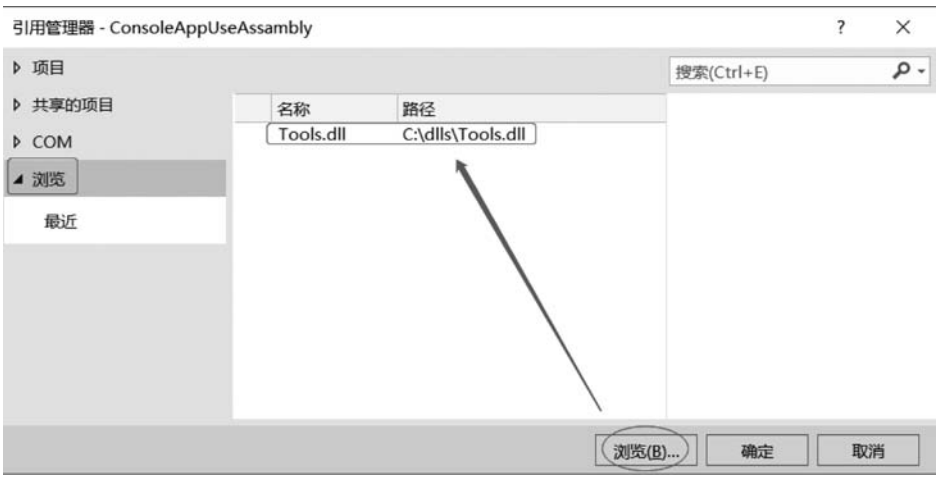

图 3-5　添加程序集 Tools.dll

④ 编辑 ConsoleAppUseAssambly 项目中的 Program.cs 文件，代码如下：

```
using System;
using Tools;                                    //引入名称空间 Tools(即程序集中的 namespace)
namespace ConsoleAppUseAssambly
{
    class Program
    {
        static void Main(string[] args)
        {
            //如下，为区分 System.Math，加上名称空间 Tools
            int a = Tools.Math.Add(1, 2);
            int b = Tools.Math.Substract(2, 1);
            Console.WriteLine(a);          //3
            Console.WriteLine(b);          //1
        }
    }
}
```

上述 Main()方法中，使用“using Tools;”引入了名称空间 Tools，然后就可用“Tools.Math.Add(1, 2);”调用名称空间中的类和类中的方法了。

小结：

(1) 创建类库项目，在其中定义名称空间和类。类库项目编译后生成程序集.dll 文件。.dll 文件可在其他项目中被引用，起到代码重用的效果。

(2) 上述使用“引用管理器”浏览方式找到自己开发的程序集文件。实际工作中可能引用其他组织开发的程序集，只需要将其下载或复制到本地进行引用即可。例如本地 C# 项目

需要与 MySQL 数据库交互,可在 MySQL 官方网站获取相应.dll 文件,并引用到本地项目中。

3.2.2 访问修饰符

访问修饰符又称为访问控制符,用来控制对类、类成员的访问权限。C# 支持 6 种不同的访问权限。

public:访问不受限制,都可访问;

protected:访问限于所在类或所在类的派生类;

internal:访问限于当前程序集;

protected internal:访问限于当前程序集或所在类的派生类;

private:访问限于所在类;

private protected:访问限于所在类或当前程序集中所在类的派生类。

图 3-6 创建两个项目及相应 5 个类

【例 3-30】 测试修饰符访问性。

具体步骤:

(1) 创建 prj1、prj2 两个项目,分别在两个项目中创建 A、B、C 和 D、E 5 个类,然后将 prj1 项目生成的程序集引入 prj2 中,如图 3-6 所示。

(2) 在项目 prj1 和 prj2 中编写源代码,测试访问修饰符的访问性,如下所示。

```
namespace Prj1{
  public class A {     //首先放开类的访问
    private static int pri;
    public static int pub;
    protected static int pro;
    internal static int inr;
    protected internal static int proInr;
    private protected static int priPro;
    void Test(){      //自己内部,都可访问
       A.pri = 1;
       A.pub = 2;
       A.pro = 3;
       A.inr = 4;
       A.proInr = 5;
       A.priPro = 6;
    }
    }
    class B {
       void Test() {
    //同一程序集内不同类,private、protected
    无法访问
          //A.pri = 1;
          A.pub = 2;
          //A.pro = 3;
```

```
using Prj1;

namespace Prj2
{
  class D
  {
      void Test()
      {
        //跨程序集,仅 public 可访问
            //A.pri = 1;
            A.pub = 2;
            //A.pro = 3;
            //A.inr = 4;
            //A.proInr = 5;
            //A.priPro = 6;
        }
    }


    class E : A
    {
        void Test(){ //不同程序集内子类访问
//继承的成员,仅 public、protected 和 protected
//internal 可继承访问
```

```
        A.inr = 4;
        A.proInr = 5;
       //A.priPro = 6;
    }
}
class C : A{
  void Test(){//同一程序集内子类,访问继
//承的成员仅 private 无法继承访问
      //pri = 1;
       pub = 2;
       pro = 3;
       inr = 4;
       proInr = 5;
       priPro = 6;
  }
}
```

```
              //A.pri = 1;
              A.pub = 2;
              A.pro = 3;
              //A.inr = 4;
              A.proInr = 5;
              //A.priPro = 6;
          }
      }
}
```

成员所属不同类型时,可用的修饰符和默认的访问修饰是不同的,如表 3-1 所示。

表 3-1　成员所属不同类型时可用的修饰符和默认的访问修饰符

所在的类型	成员可用的修饰符	成员默认的访问修饰
enum	不写	public
struct	public、internal、private	private
class	public、protected、internal、private、protected internal、private protected	private
interface	public、protected、internal、private、protected internal、private protected	public

3.2.3　异常处理

程序在运行过程中,可能遇到各种异常情况。如:用户输入数值或日期时格式有错;读取文件但文件却不存在;执行的 SQL 语句有语法错误;数组访问时下标越界等。此时需要编写代码做出合理处理,而不是任其发生,造成程序运行崩溃。C＃中引入了 System. Exception 类及其子类,并设计了异常处理机制代码编写构架:用 try、catch、finally、throw 关键字处理。

1. 异常类

C＃中 System. Exception 为所有异常的根类,下面派生了 System. ApplicationException 和 System. SystemException 两个子类。其中,System. ApplicationException 支持应用程序产生的异常,所以程序中自定义的异常都应该派生自该类;System. SystemException 是系统预定义异常的基类,如 IO 异常、下标越界异常等都派生自该类。

C＃中常见预定义的异常类如表 3-2 所示。

表 3-2　C#中常见预定义的异常类

预定义的异常类	描　　述
System. IO. IOException	输入输出异常
System. IndexOutOfRangeException	当索引超出了下标范围时产生的异常
System. ArrayTypeMismatchException	当数组类型不匹配时产生的异常
System. NullReferenceException	当引用一个空对象时产生的异常
System. DivideByZeroException	当除以零时产生的异常
System. FormatException	按照格式转换类型时产生的异常
System. InvalidCastException	在类型转换期间产生的异常
System. OutOfMemoryException	内存不足时产生的异常
System. StackOverflowException	栈溢出时产生的异常

2. 异常发生时不处理

异常发生时,若不处理,则应用运行时会中断。

【例 3-31】 访问数组元素,发生下标越界异常时运行被中断。

```
int[] iAry = { 1, 2, 3 };
Console.WriteLine(iAry[3]);
Console.WriteLine("End");
```

执行第 2 行时,抛出 IndexOutOfRangeException 异常,运行中断,无法执行到输出"End"所在行,如图 3-7 所示。

图 3-7　访问数组下标越界时抛出 IndexOutOfRangeException 异常

【例 3-32】 读取不存在文件,抛出文件找不到异常,运行被中断。

```
using System.IO;
…
StreamReader sr = new StreamReader("C:/math.txt");
string line = null;
while ((line = sr.ReadLine()) != null)
{
    Console.WriteLine(line);
}
sr.Close();
```

执行 new StreamReader("C:/math.txt")时,抛出 FileNotFoundException 异常,运行

中断，无法执行后面几行代码，如图 3-8 所示。

图 3-8　FileNotFoundException 异常

3. 异常处理——用 try … catch … finally 语句

如上，当异常发生时，程序可能无法再正常执行。为保障程序能继续执行，C＃提供了 try … catch … finally 语句。

try … catch … finally 语句的语法格式：

```
try {
    业务逻辑语句块;                    //可能发生异常
} catch[(异常类 1 e)]{                 //若异常发生,但不想引起异常时可用 catch{ }处理
    异常处理语句块;
}[catch(异常类 2 e){
    异常处理语句块;
} …
catch(异常类 N e){
    异常处理语句块;
}] finally {
    不管是否产生异常,都执行的语句块;
}
```

try 语句块就是可能产生异常的语句块。在执行过程中，当产生异常时，会产生并抛出相应类型的异常对象，后面的 catch 语句段可分别对异常做相应处理。异常处理结束以后，不会转回执行 try 语句段中未执行的代码。一个 try 语句必须带有至少一个 catch 语句块或一个 finally 语句块。

catch 语句块可以有一个或多个，用于处理可能产生的不同类型的异常。catch 对异常的捕获顺序是：从上至下，先捕获子类异常再捕获父类异常。也就是越是父类越应放在下方。若上方写了父类异常处理，则子类异常就无法在下面捕获了，这在语法上是错误的。

不管是否发生异常，finally 语句块都必须要执行。通常在该语句块中释放打开的资源，如关闭文件流、关闭数据库连接等。

【例 3-33】　用 try … catch … finally 语句处理 FileNotFoundException 异常。

```
using System.IO;
…
StreamReader sr = null;
try
{
   sr = new StreamReader("C:/math.txt");
```

```
    string line = null;
    while ((line = sr.ReadLine()) != null)
    {
          Console.WriteLine(line);
    }
}
catch(FileNotFoundException e)
{
    Console.WriteLine("发生文件找不到异常: " + e.Message);
    //发生文件找不到异常: Could not find file 'C:\math.txt'
}
finally
{
    if (sr != null)
    {
        sr.Close();
    }
}
Console.WriteLine("End");      //End
```

运行程序,控制台输出:

```
发生文件找不到异常: Could not find file 'C:\math.txt'.
End
```

上面程序的执行过程:当执行 sr = new StreamReader("C:/math.txt")时,抛出 FileNotFoundException 异常;该异常由 catch(FileNotFoundException e)捕获并交由 e 变量引用,接着执行 Console.WriteLine("发生文件找不到异常:" + e.Message);再接着执行 finally 语句块,在 finally 中关闭 sr 文件流资源。

若在 C:\盘下创建 math.txt 文件,并输入文本 9/3,再执行以上程序,输出:

```
9/3
End
```

一般场合中,针对可能产生的不同的异常,会使用多个 catch 语句捕获多种异常。

【例 3-34】 分析文件内容,并执行程序,输出每行数学表达式的值。

```
using System.Data;
using System.IO;
...
StreamReader sr = null;
try
{
     sr = new StreamReader("C:/math.txt");
     string line = null;
     while ((line = sr.ReadLine()) != null)
     {
          Console.Write(line);
```

```
            DataTable eval = new DataTable();      //System.Data
            Console.WriteLine(" = " + eval.Compute(line, ""));
        }
    }
    catch(FileNotFoundException e)
    {
           Console.WriteLine("发生文件找不到异常: " + e.Message);
    }
    catch (SyntaxErrorException e)
    {
          Console.WriteLine("表达式运算时异常: " + e.Message);
    }
    finally
    {
           if (sr != null)
           {
               sr.Close();
           }
    }
    Console.WriteLine("End");
```

修改 C:\math.txt 内容为：

```
9/3
3/
```

运行程序，控制台输出：

```
9/3 = 3
3/表达式运算时异常: Syntax error: Missing operand after '/' operator.
End
```

分析以上代码，catch 代码块有 2 个，分别为 catch(FileNotFoundException e)和 catch (SyntaxErrorException e)。当 eval.Compute(line, "")语句执行到 math.txt 第二行“3/”时，产生了 SyntaxErrorException 异常，此时因为有第二个 catch 语句块，所以该异常被捕获，规避了程序中断情况的发生。

4. 用 using 处理异常

在 C# 中，关键字 using 除了可以引用名称空间外，还可以使用 using 语法结构来代替 try … catch … finally 语句处理。在执行完毕后，会自动关闭在 using 后面括号“()”中创建的资源，使相应代码得以简化。

语法结构：

```
using( 资源创建 )      //在 using 代码块运行结束后，此处创建资源会自动关闭
{
    代码块
}
```

【例 3-35】 用 using 自动关闭资源。

```
using (StreamReader sr = new StreamReader("C:/math.txt"))
{
    string line = null;
    while ((line = sr.ReadLine()) != null)
    {
        Console.Write(line);
        System.Data.DataTable eval = new System.Data.DataTable();
        Console.WriteLine(" = " + eval.Compute(line, ""));
     }
}
```

在 using 后面括号中创建的资源,在 using 结构执行完毕后会自动调用资源的关闭方法。需要注意的是,using 代码块中异常并未处理,所以执行过程中依然可能出现异常,如遇到 eval.Compute(line, ""),执行到"3/"时,还是会抛出 SyntaxErrorException 异常,造成中断。

5. 用 throw 抛出异常

有时需要用代码主动抛出异常,以便调用者对此进行有针对性的处理。此时可使用 throw 关键字抛出异常。

【例 3-36】 检查用户名,若为空,则抛出异常。

```
string name = "";
if (String.IsNullOrWhiteSpace(name))     //等价 if (name == null || name.Trim().Length == 0)
{
    throw new Exception("姓名不能为空.");     //代码主动抛出异常
}
```

6. 自定义异常

除了系统预定义异常外,有时项目需要自定义异常,以处理特有问题。自定义异常需要继承自 Exception 类,但实际工作中习惯用 Exception 类的子类 ApplicationException 作为自定义异常类的父类。

【例 3-37】 创建自定义异常。

```
public class ParamException : Exception
{
    string message;
    Exception innerException;
    public ParamException() { }
    public ParamException(string message) : base(message)     //此构造最常用
    {
        this.message = message;
    }
    public ParamException(string message, Exception innerException) : base(message, innerException)
    {
        this.message = message;
        this.innerException = innerException;
```

```
    }
}
```

调用：

```
string name = "";
if (String.IsNullOrWhiteSpace(name))
{
    throw new ParamException("姓名不能为空.");      //主动抛出自定义异常
}
```

3.2.4 递归

递归的简单定义：方法直接或者间接调用自己。

递归不是面向对象语言独有的，但递归比较抽象，并不直观。对于初学者而言，它确实难以理解，学习有一定难度。

还是从问题出发来理解递归的使用。

问题 1：求解 10 的阶乘(factorial)。

这个问题可以用 for 循环语句实现。

【例 3-38】 用 for 循环语句求解 10 的阶乘。

```
long factorial = 1;
for (int i = 1; i <= 10; i++)
{
    factorial *= i;
}
Console.WriteLine(factorial);
```

结果：

```
3628800
```

换个解题思路：假设有一个方法 $f(i)$为求 i 的阶乘，那么求 n 的阶乘就可表示为 $n * f(n-1)$；而求 $f(n-1)$可表达为$(n-1) * f(n-2)$；以此类推，最后求 $f(1)$，结果为 1。可用如下公式表示：

$$f(n)=\begin{cases}1, & n=1\\ n * f(n-1), & n>1\end{cases}$$

对此公式，可以用 C# 写出求 $f(n)$的方法来。具体如下。

【例 3-39】 用递归方法实现 n 的阶乘求解。

```
static long f(int n)
{
    if (n == 1)                    //不可以无限制处理,必须有退出条件
    {
        return 1;
    }
```

```
    return n * f(n - 1);      //n的阶乘 = n*(n-1的阶乘)
}
```


调用:

```
Console.WriteLine(f(10));
```

其结果与前面第一种 for 循环语句解答结果相同,为:

```
3628800
```

总结递归写法,分为两部分:

(1) 正常的逻辑处理代码,如"return n*f(n-1);"。

(2) 以免陷入死递归,需有退出条件,如"if(n==1) return 1;"。

有时某些问题用递归思路解题更为清晰。例如,斐波那契数列:0,1,1,2,3,5,8,13,21,34,55,89,144…… 其规律是后一个数等于前面两个数的和。即,若求第 n 项的值,可用公式表示为:

$$f(n)=\begin{cases} n, & n\leqslant 1 \\ f(n-1)+f(n-2), & n>1 \end{cases}$$

使用递归思路,可定义一个方法求解第 n 项对应的斐波那契数列的值,具体如下。

【例 3-40】 用递归方法实现求斐波那契数列第 n 项数值。

```
static int f(int n)
{
    if (n <= 1)                          //(2)以免陷入死递归,需有退出条件
    {
        return n;
    }
    return f(n - 1) + f(n - 2);          //(1)正常的逻辑处理代码
}
```

调用:

```
Console.WriteLine(f(4));
```

结果:

```
3
```

n=4 时,斐波那契数列的值为 3,测试结果与实际一致。

3.3 项目案例——中华文明,魅力永恒

文化是一个国家、一个民族的灵魂,文化自信是实现中华民族伟大复兴的精神力量。中华文化经过历史长河的洗练、峥嵘岁月的磨砺、伟大实践的锻造,是最有韧劲、最具内涵、最

富生机的文化，是凝聚亿万人民为新中国发展不懈奋斗的精神力量。

中华文明是世界上唯一没有中断的既古老又年轻的文明，是人类文明灿烂星空中最绚丽的星宿。五千多年文明江河奔流到如今，涌现出老子、孔子、庄子、孟子、屈原、李白、苏轼、曹雪芹等灿若星辰的伟大人物，诞生了诗经、楚辞、汉赋、唐诗、宋词、元曲、明清小说等浩如烟海的文学经典，为中华民族生生不息、薪火相传提供了精神滋养。这些文化基因和精神标识，历经千年风雨的洗礼依然挺立、生机勃勃。中华文化跨越时空的永恒价值和魅力，是我们的自信之根。

在此，假设未来需实现一个“中国文化代表人物风采演示系统”，要求事先做出如下部分设计。

3.3.1 设计一：设计用户类

设计说明：为登录和管理系统，需要设计一个用户类 User。该类应该有用户名 Name、密码 Pwd 两个属性，有登录时判断用户是否有效的方法 IsValid()。

注意，对于方法中的逻辑实现，因为尚未掌握数据库相关技术，可进行代码模拟实现，此处假定有效用户名为 admin，密码为@dm1n。

设计实现步骤：

(1) 启动 Visual Studio。

(2) 选择“文件”→“新建”→“项目”选项，弹出“创建新项目”窗口。

(3) “语言”选择 C#，“项目类型”选择“库”，在列表中选择“类库”选项，单击“下一步”按钮，弹出“配置新项目”窗口。

(4) 在“项目名称”文本框中输入 CulCelebrity，单击“下一步”按钮，单击“创建”按钮，打开 Visual Studio 开发“类库”项目界面。

(5) 在“解决方案资源管理器”窗口中，右击项目 CulCelebrity，在弹出的快捷菜单中选择“添加”→“类”选项，在弹出的对话框中设置名称为 User.cs，单击“添加”按钮。在类中编写如下代码：

```
using System;
namespace Cn.Edu.Common.CultrueCharacter
{
    public class User
    {
        public String Name { get; set; }
        public String Pwd { get; set; }
        public bool IsValid()
        {
            if("admin" == Name && "@dm1n" == Pwd)
            {
                return true;
            }
            return false;
        }
    }
}
```

设计小结：

(1) User 类可以使用于控制台应用或者 Windows 窗体应用中。为此创建类库项目，并将 User 类存放于类库项目中，作为程序集. dll 形式引用比较适合实际开发情景。

(2) 本处 User 类中使用了模拟方式，“写死”了用户名 admin 和密码@dm1n，在实际项目开发中这是不可取的，不仅不利于账号维护，同时存在安全隐患。在掌握了数据库相关技术后，用户信息应存放在数据表中。

3.3.2 设计二：设计类别类

设计说明：文化人物信息应该进行分类管理，如李白属于诗人，孔子属于教育家，等等。为此设计一个类别类 Classification，该类应该有类别名 Name、类别描述 Description 两个属性，同时设计无参构造和带参构造，以方便创建分类对象。注意，带参构造应该传入 Name 和 Description 两个参数值，并对属性值进行初始化。

设计实现步骤：在“解决方案资源管理器”窗口中，右击项目 CulCelebrity，在弹出的快捷菜单中选择“添加”→“类”选项，在弹出的对话框中设置名称为 Classification. cs，单击“添加”按钮。在类中编写如下代码：

```
using System;
namespace Cn.Edu.Common.CultrueCharacter
{
    public class Classification
    {
        public string Name{ get; set;}
        public string Description { get; set; }
        public Classification()
        {
        }
        public Classification(string name,string description)
        {
            Name = name;
            Description = description;
        }
    }
}
```

设计小结：

(1) Classification 类可使用于不同类型中。为此将 Classification 类存放于类库项目 CulCelebrity 中相对合适。

(2) 此处 Classification 类中定义了两个构造方法。带参数构造在便于创建对象的同时，对属性值进行初始化。因为定义了带参构造，编译器不再生成默认的无参构造了，所以若需保留无参构造则应显式编写代码加入。

3.3.3 设计三：设计人物类

设计说明：展现文化人物，需要姓名、介绍、图片等元素，以及所属类别。

为此设计一个人物类 Celebrity，该类应该有姓名 Name、介绍 Description、图片路径 ImageURL 三个属性。此外，人物的所属分类 Classification 的值可能有多个，如孔子既是思想家也是教育家，可用数组表示（待后续学过集合，可用更合适的 List < T >类替代）。

设计实现步骤：在“解决方案资源管理器”窗口中，右击项目 CulCelebrity，在弹出的快捷菜单中选择“添加”→“类”选项，在弹出的对话框中设置名称为 Celebrity. cs，单击“添加”按钮。在类中编写如下代码：

```
using System;
namespace Cn. Edu. Common. CultrueCharacter
{
    public class Celebrity
    {
        public string Name { get; set; }
        public string Description { get; set; }
        public string ImageURL { get; set; }
        public Classification[ ] Classifications { get; set;}
    }
}
```

设计小结：

（1）同样，Celebrity 类可使用于不同类型应用中，为此将 Celebrity 类存放于类库项目 CulCelebrity 中比较合适。

（2）人物类 Celebrity 中，属性 Classifications 类型被定义为 Classification[ ]，原因是 Classification 类与 Celebrity 类形成了组合关系。实际上，类之间除了继承关系外，还有其他关系，如组合、依赖、关联、聚合等。在实践开发过程中可自行学习和体会，此处不做过多介绍。

3.3.4 设计四：创建控制台应用引用类库，并检验设计类

设计说明：创建控制台应用，引用类库，并检验类库中的设计类。创建思想家和教育家两个分类对象；创建文化人物对象孔子，并将其加入创建的分类对象（思想家、教育家）中。

设计实现步骤：

（1）右击项目 CulCelebrity，在弹出的快捷菜单中选择“生成”选项，生成程序集文件 CulCelebrity. dll。

（2）创建控制台应用 ConsoleAppCulCelebrity。

（3）右击应用 ConsoleAppCulCelebrity 下方的“依赖项”，在弹出的快捷菜单中选择“添加项目引用”选项。

（4）在弹出的“引用管理器”对话框中，通过“浏览”按钮添加程序集 CulCelebrity. dll。

（5）在 Program. cs 中引用类库 CulCelebrity 中相关类的名称空间：

```
using Cn. Edu. Common. CultrueCharacter;
```

(6) 在 Main()方法内,编写如下代码:

```
using Cn.Edu.Common.CultrueCharacter;
using System;
namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            //创建分类
            Classification c1 = new Classification("思想家", "富有独特思想和智慧的人可称之为思想家,但是一般指是那些影响特别大的哲学家。");
            Classification c2 = new Classification("教育家", "以教育作为学科知识进行系统研究的学问家。");
            //创建人物
            Celebrity kongZi = new Celebrity();
            kongZi.Name = "孔子";
            kongZi.Description = "名丘,字仲尼,世称"圣人",春秋时期鲁国人。我国古代伟大的思想家、教育家,儒家学派创始人,私学创办人。孔子及其弟子的主要言行思想由孔子的弟子及再传弟子记录在«论语»20 篇中。";
            kongZi.ImageURL = "images/kongzi.png";
            kongZi.Classifications = new Classification[] { c1, c2 }; //设置分类
            Console.ReadLine();
        }
    }
}
```

设计小结:通过对类库项目程序集的“添加项目引用”操作,以及代码中“using 对应名称空间”,应用项目就可实现访问类库中的相关设计类。

实际上,在 Visual Studio 开发工具中,一个解决方案下可以有多个项目,每个项目各自完成相对独立的功能。通过“引用”,项目之间就能实现功能的调用。