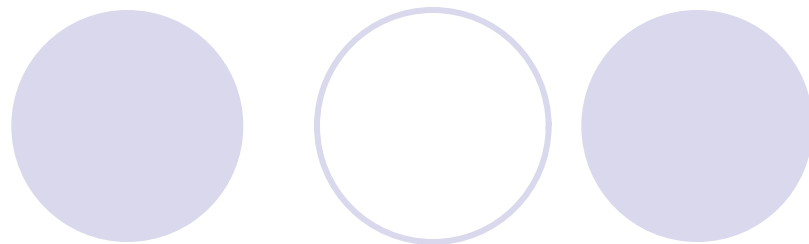


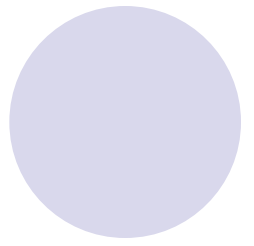
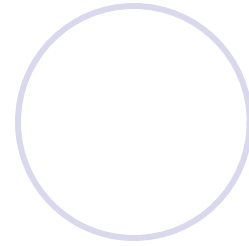
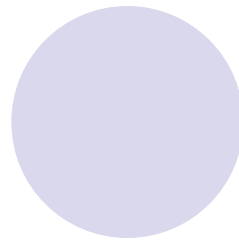
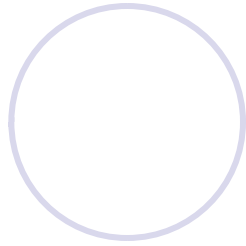
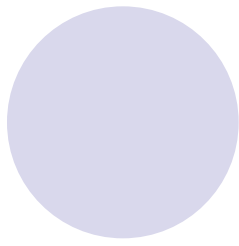
5.5 类的继承与多态



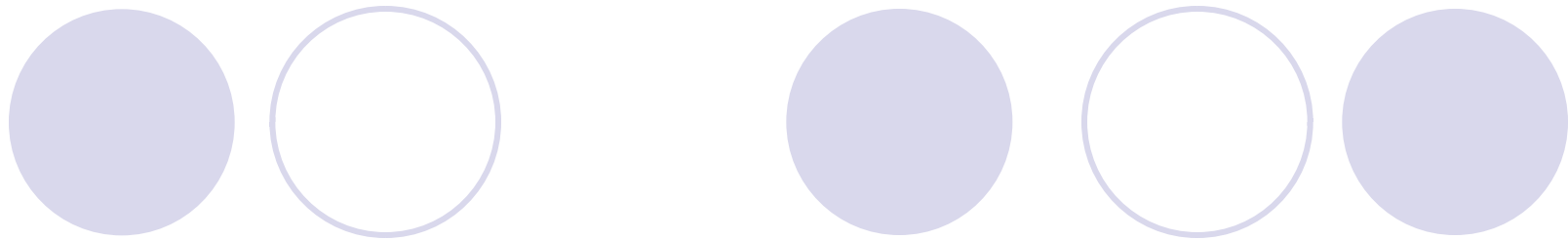
- 5.5.1 类的继承

- 1. 继承

- 继承是面向对象程序设计的主要特征之一。从一个已经存在的类(基类或父类)中获得公有数据成员和方法成员，从而创建新的类(派生类或子类)。派生类可以在继承父类公有成员的基础上增加一些变量和方法。派生类也可以覆盖被继承的方法并重写此方法。通过继承实现代码的共享，可以提高开发效率，并有助于减少错误。



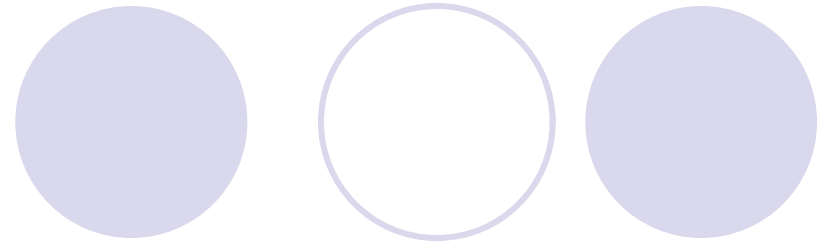
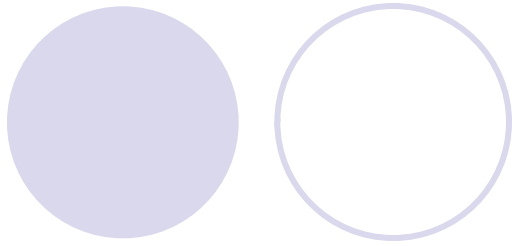
- 从基类(也叫父类)继承一个新的子类(也叫派生类)的语句格式:
- **class** 派生类名: 基类名
- {
- //派生类代码
- }
- 新的类可以从现有类库中继承。
- 基类的私有成员只能被基类的方法访问, 而不能被它所派生的子类的方法访问。



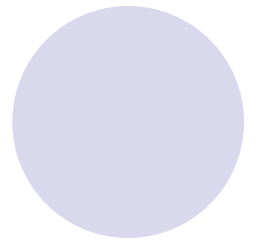
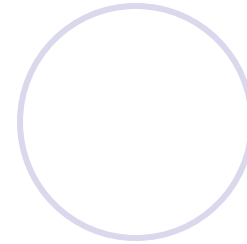
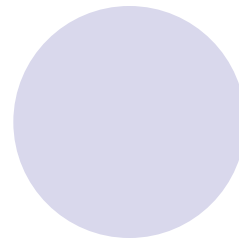
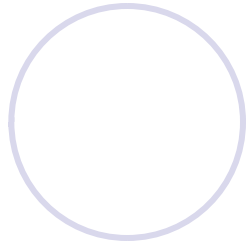
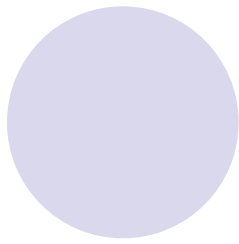
- 在建立子类的实例时，先调用基类的构造函数来初始化派生类对象中的基类成员，接着执行派生类成员对象的构造函数，最后调用派生类构造函数。
- 派生类的对象可以作为基类的对象处理，即派生类对象可以访问基类的公有成员，也允许将派生类对象赋值给基类对象。

【例5-15】 继承示例。最好单步运行，可以清楚查看创建子类对象时，调用基类构造函数和子类构造函数的顺序。运行结果如图5.11所示

```
● public class Jilei                                //基类
● {
●     public Jilei()                                //基类构造函数
●     { Console.WriteLine("基类构造函数"); }
●     public void MethodJ()                          //基类中的方法
●     { Console.WriteLine("调用MethodJ()"); }
● }
● public class Zilei:Jilei                          //派生类Zilei，其父类为Jilei
● {
●     public Zilei()                                //派生类的构造函数
●     { Console.WriteLine("子类构造函数"); }
●     public void MethodZ()                          //派生类中的方法
●     { Console.WriteLine("调用MethodZ()"); }
● }
```



```
● class Test
● {
●     static void Main(string[] args)
●     {
●         Jilei J1 = new Jilei();    //定义一个基类实例
●         J1.MethodJ();              //对象J1调用基类中的方法
●         Zilei Z1 = new Zilei();
●         //定义一派生类实例，先调用基类构造函数，再调用子类构造函数
●         Z1.MethodZ();              //子类的对象Z1调用子类中的方法MethodZ
●         Z1.MethodJ();              //子类的对象Z1调用基类中的方法MethodJ
●         Jilei JJ = (Jilei)Z1;
●         //定义基类对象JJ，地址指向Z1，对Z1进行强制类型转换
●         JJ.MethodJ();              //基类对象调用基类中的方法MethodJ
●         //JJ.MethodZ();            //该句是错的，基类的对象不能调用子类的
方法
●         Zilei ZZ = (Zilei)JJ;
```



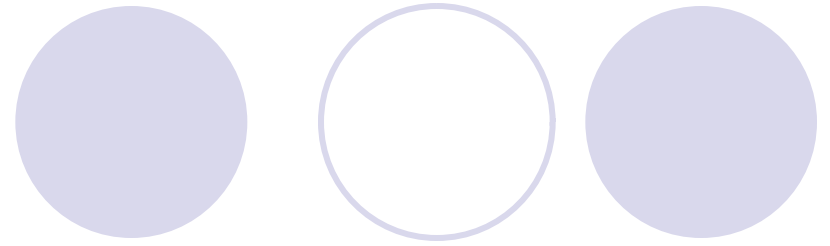
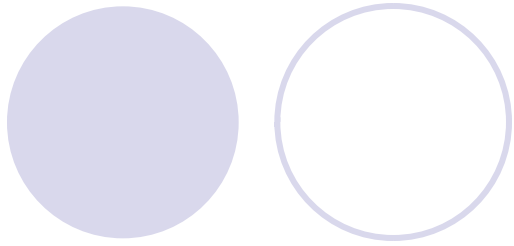
- `ZZ.MethodZ();` //子类对象调用基类的方法
- `ZZ.MethodJ();` //子类对象调用子类中的方法
- `Console.Read ();`
- `}`
- `}`



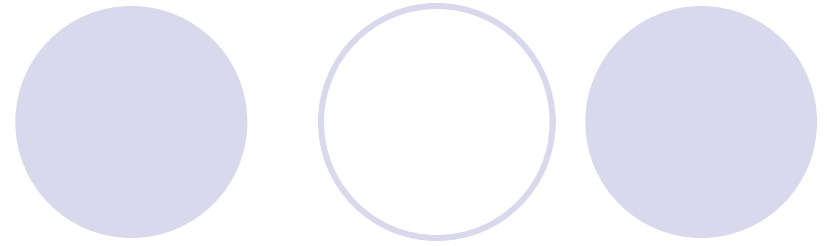
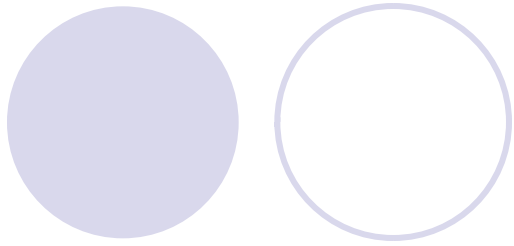
图5.11 例5-15运行结果

2. 访问关键字**this**和**base**

- **this**关键字引用类的当前实例。**this**关键字只能在实例构造函数、实例方法、实例访问器中使用。
- **base**关键字用于派生类中访问基类的成员。**base**关键字只能在实例构造函数、实例方法、实例访问器中使用。
- **【例5-16】** 访问关键字**this**示例。运行结果如图5.12所示，

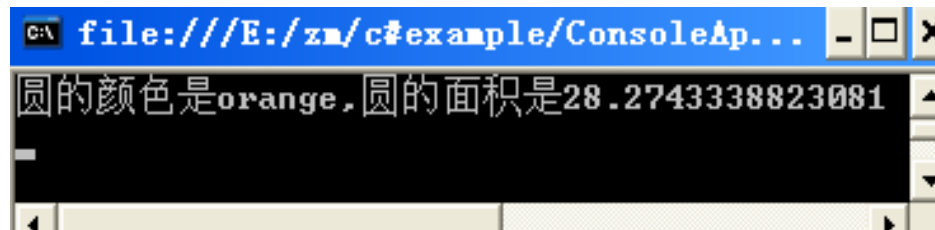


- `public class Shape`
- `{`
- `protected string Color;` //基类中的私有字段
- `public Shape()` //无参构造函数
- `{ ; }`
- `public Shape(string Color)` //有参构造函数
- `{ this.Color = Color; }` //this关键字引用类的当前实例，
等号右边的Color是形参
- `public string GetColor()` //基类中的方法
- `{ return Color; }`
- `}`



- `public class Circle : Shape //圆`
- `{`
- `private double Radius;`
- `public Circle(string Color, double Radius)`
- `{`
- `this.Color = Color;` `//this`关键字引用类的当前实例，等号
右边的`Color`是形参
- `this.Radius = Radius;` `//等号左边的Radius指Radius类中的实`
例字段(私有的)
- `}`
- `public double GetArea()`
- `{ return System.Math.PI * Radius * Radius; }`
- `}`

- class Test
- {
- static void Main(string[] args)
- {
- Circle Cir= new Circle("orange",3.0);
- Console.WriteLine("圆的颜色是{0},圆的面积是{1}", Cir.GetColor(),Cir. GetArea());
- Console.Read ();
- }
- }



The screenshot shows a Windows console window titled "file:///E:/zm/c#example/ConsoleAp...". The output text is "圆的颜色是orange,圆的面积是28.2743338823081". The window has a standard Windows title bar with minimize, maximize, and close buttons.

图5.12 例5-16运行结果

【例5-17】 访问关键字**base**示例。运行结果如图5.13所示，代码如下：

```
public class Person                                //基类
{
    protected string Phone;                        //基类中的字段
    protected string Name;
    public Person(string name, string phone) //基类构造函数
    {
        this.Phone = phone;                       //this关键字引用类的当前实例
        this.Name = name;
    }
    public void GetInfoPerson() //基类中的方法
    {
        Console.WriteLine("Phone: {0}", Phone);
        Console.WriteLine("Name: {0}", Name);
    }
}
```

```
class Employee : Person //派生类，其基类为Person
{
    public string ID; //工号
    public Employee(string name, string phone, string id)
        : base(name, phone) //派生类构造函数，并用:base
调用基类构造函数
    {
        this.ID = id;
    }
    public void GetInfoEmployee() //派生类中的方法
    {
        base.GetInfoPerson(); //调用基类Person的GetInfo方
法
        Console.WriteLine("Employee ID: {0}", ID);
    }
}
```



图5.13 例5-17运行结果

- class Test
- {
- static void Main(string[] args)
- {
- Employee Employees = new Employee("赵敏", "8888888", "19930701");
- //定义一派生类对象
- Employees.GetInfoEmployee(); //调用派生类的方法
- Console.ReadLine();
- }
- }


5.5.2 类的多态

- 1. 虚方法、重写(覆盖)方法和隐藏方法
- 在类的继承中，如果子类中声明了与基类相同名称的方法，并使用了相同的参数个数、相同的参数类型、相同的返回值类型等，就称派生类的成员覆盖了基类中的成员。
- 在基类中声明方法时，使用**virtual**关键字定义虚方法；然后派生类中声明方法时，使用**override**关键字来重写(覆盖)。
- 在派生类中使用关键字**new**修饰定义与基类成员同名的成员，可以实现隐藏基类成员功能的作用，即在派生类中创建与基类中的方法或数据成员同名的方法或数据成员时，将隐藏基类的方法或数据成员。在5.3.3节对方法的重载已做了介绍，表5-6对覆盖与重载进行了比较。

覆盖与重载进行了比较。图5.14 例5-18运行结果

【例5-18】 虚方法的重写，非虚方法的隐藏示例。运行结果如图5.14所示。代码如下

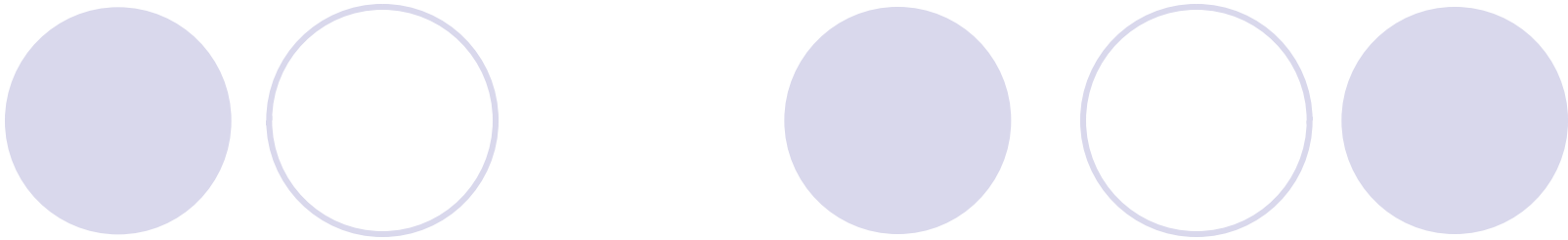
```
● class A
● {
●     public void F()           //A类的非虚方法
●     { Console.WriteLine("A类中的F方法"); }
●     public virtual void G()   //A类的虚方法，用virtual
●     { Console.WriteLine("A类中的G方法"); }
● }
● class B: A                   //B类是由A类派生而来
● {
●     new public void F()       //在B类中对方法F实现了隐藏
●     { Console.WriteLine("B类中的F方法"); }
●     public override void G() //在B类中对虚方法G实现了重写
●     { Console.WriteLine("B类中的G方法"); }
● }
```



```

class Test
{
    static void Main(string[] args)
    {
        B b = new B();           //创建一个派生类的对象
        A a = b;                 //对b进行隐式转换成A类，且a、b指向
        同一个地址
        a.F();
        //在B类中没有对A类中的F函数进行覆盖，只是隐藏了A类中的F，
        A类中的F仍存在
        b.F(); //B类的对象b调用B类中的F方法
        //调用a.G，因a是由b转换而来，B类中已覆盖了A类的虚方法
        G
        //只有B类中的G方法，只能调用B类的G
        a.G();
        b.G(); //B类的对象b调用B类中的G方法
        Console.ReadLine();
    }
}

```


The image features five circles at the top: three solid light purple circles and two hollow light purple circles with thin outlines, arranged in a sequence from left to right.

```
file:///E...
A类中的F方法
B类中的F方法
B类中的G方法
B类中的G方法
```

图5.14 例5-18运行结果

表5-6 覆盖与重载概念的比较

比 较 项 目	覆盖(override)	重载(overload)
特点	派生类函数覆盖基类函数	用于同一类中的成员函数
位置	分别位于派生类与基类	在同一类中
函数名	相同	相同
参数	参数相同	参数不同
虚函数	基类函数必须是虚函数	与是否虚函数无关

2. 抽象类与抽象方法

- 将关键字**abstract**置于关键字**class**的前面，可以将类声明为抽象类。如：
- **abstract class Shape{**
- 抽象类不能实例化，只能作为其他类的基类。
- 在抽象类中可以定义抽象方法。将关键字**abstract**置于方法的返回类型前面。如：
- **public abstract double Area();**
- 抽象方法在基类的声明中没有实现，所以方法声明后面是分号，而不是常规的**}**。抽象类的派生类必须实现所有的抽象方法。要求所有的派生非抽象类都要重写实现抽象方法。引入抽象方法的原因在于抽象类本身是一种抽象的概念，有的方法并不要具体的实现，而是留下来让派生类来重写实现。


【例5-19】 抽象类和抽象方法示例。运行结果如图5.15所示，代码如下：

```
public abstract class tiyu                //定义抽象类“体育”
{
    public abstract void play();          //在抽象类中定义一个抽象方法，
    没有实现部分
}
public class yumaoqiu : tiyu              //定义子类“羽毛球”，
    其基类为tiyu
{
    public override void play()           //重写抽象方法，有实现部分，
    用关键字override
    {
        Console.WriteLine(" 打羽毛球");
    }
}
```

//定义子类“足球”，其基类为tiyu

```
public class zuqiu : tiyu
{
    public override void play()
    {
        Console.WriteLine(" 踢 足 球");
    }
}
```

```
public class youyong : tiyu    //定义子类“游泳”，其基类为tiyu
{
    public override void play()
    {
        Console.WriteLine(" 游  泳");
    }
}
```



```
class Test
{
    static void Main(string[] args)
    {
        int tiyuNum = 3;
        tiyu[] tiyuxm = new tiyu[tiyuNum];
        //定义抽象类的数组，但没有初始化，即没有实例化
        //tiyu aa = new tiyu();//错误，因为抽象类不能实例化
        //创建一个yumaoqiu实例，隐式转换为tiyu类，tiyuxm[0]地址指向
        该yumaoqiu实例
        tiyuxm[0] = new yumaoqiu();
        tiyuxm[1] = new zuqiu();
        tiyuxm[2] = new youyong();
        foreach (tiyu i in tiyuxm)           //遍历数组中的元素
            i.play();                       //调用在子类中已经重写的方法
        Console.ReadLine();
    }
}
```

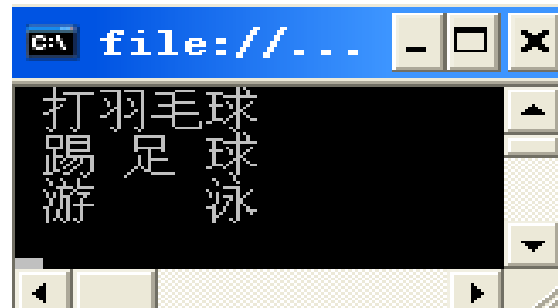
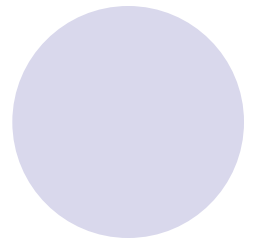
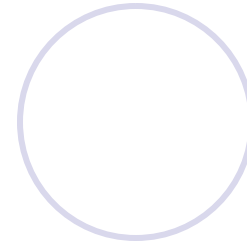
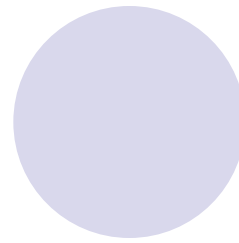
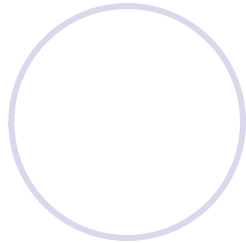
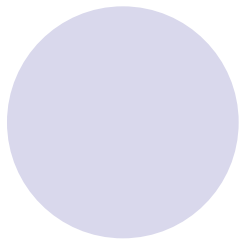
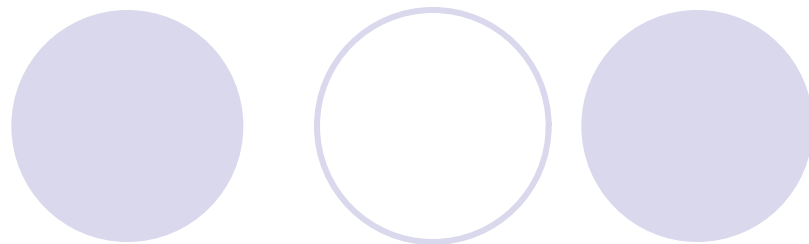


图5.15 例5-19运行结果

3. 密封类与密封方法



- 抽象类作为基类，不能被实例化，而由其他类继承。
- 还有一种不能被其他类继承的类，叫密封类，用**sealed**关键字，并将**sealed**置于关键字**class**的前面。密封类不能用做基类，因此也不能是抽象类。
- 密封类主要用于防止非有意的派生。由于密封类从不用做基类，因此调用密封类成员的效率可能会更高些。
- 如果类的方法声明包含**sealed**修饰符，称该方法为密封方法。类的实例方法声明包含**sealed**修饰符，则必须同时使用**override**修饰符。使用密封方法可以防止派生类进一步重写该方法。

【例5-20】 密封方法的使用(通过该例观察密封方法能否被继承，能否被子类重写)。运行结果如图5.16所示，代码如下：

```
class A //基类
{
    public virtual void F()//虚方法有实现部分，但是抽象方法在基类
    中不能有实现部分
    { Console.WriteLine("A.F"); }
    public virtual void G()
    { Console.WriteLine("A.G"); }
}
class B:A //子类
{
    sealed override public void F()
    { Console.WriteLine("B.F"); }
    public override void G()
    //重写了A类中的虚方法G，同时B类中G方法仍是虚方法
    { Console.WriteLine ("B.G"); }
}
```

- `class C:B` //子类
- `{//在C类中不能对F进行重写，因B类中F方法已是密封的。但方法F在C类中能被继承下来`
- `public override void G() //重写了在B类中的G方法`
- `{ Console.WriteLine ("C.G"); }`
- `}`
- `class Test`
- `{`
- `static void Main(string[] args)`
- `{`
- `A a1 = new A();`
- `B b1 = new B();`
- `C c1 = new C();`
- `a1.F();`
- `a1.G(); b1.F();`
- `b1.G(); c1.F();`
- `//c1对象(调用了B类的F方法)继承了B类中的F方法。但在C类中不能重写F方法`
- `c1.G();`
- `Console.ReadLine();`
- `}`
- `}`

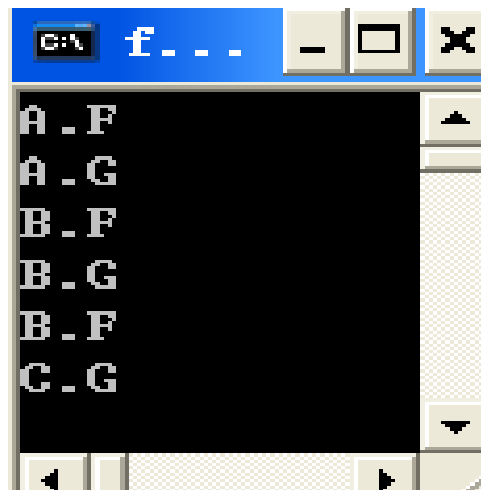
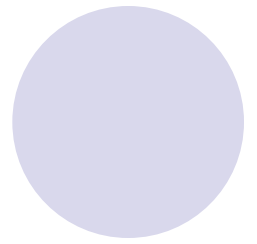
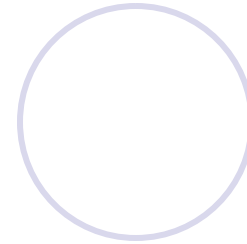
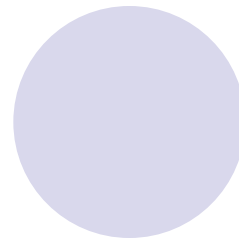
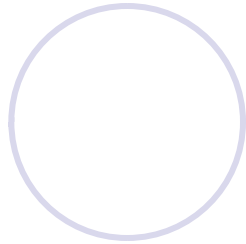
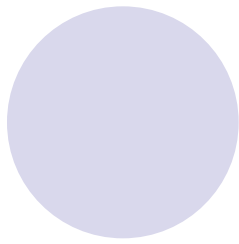
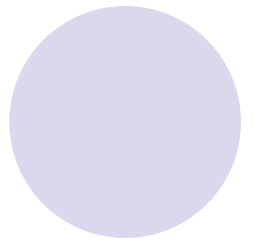
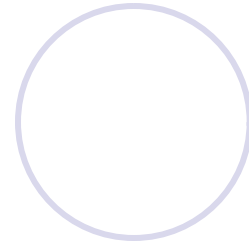
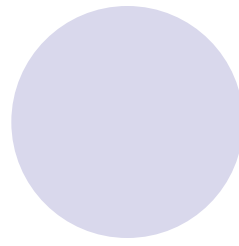
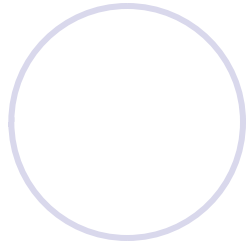
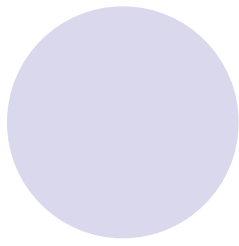


图5.16 例5-20运行结果

5.6 接口

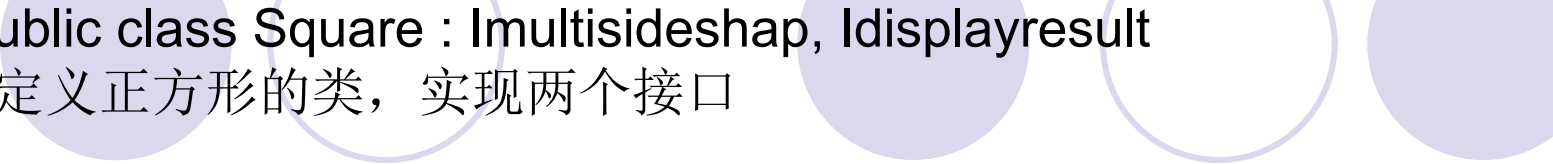
- **C#**只允许单继承机制，而如果希望一个子类继承两个或两个以上更多的父类，**C#**是不支持的。但在**C#**语言中，“多重继承”的功能可通过“接口”技术来实现。
- 接口具有类似于抽象类的地位，它只具有“被继承”的特性，所以接口也像抽象类一样，是一个最高层次的“基类”。但抽象类只能实现单继承，而接口可以实现多继承。
- 接口类似于抽象类的引用类型，主要用来声明要定义的类中将包含哪些功能(方法、属性、索引器或事件)，但不包含这些功能的实现代码(类似于抽象类)，只在继承类中才完成实现代码部分。

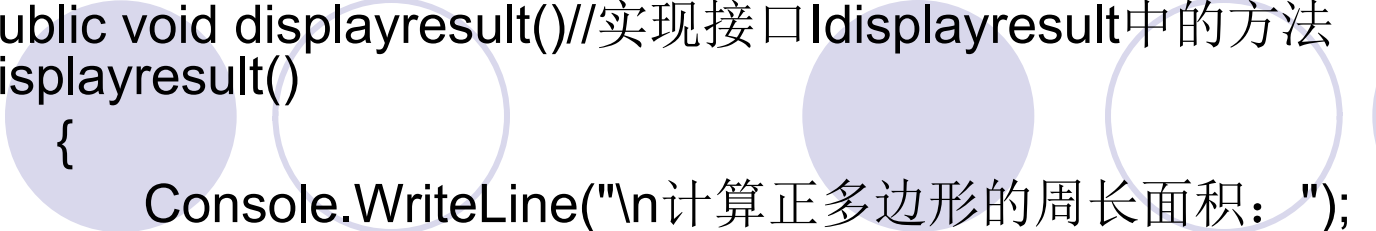


- 接口使用关键字**interface**定义，可以使用的修饰符包括**new**、**public**、**protected**、**internal**、**private**等。接口的命名通常是以**I**开头，如**IPartA**、**IPartB**。接口的成员可以是方法、属性、索引器和事件，但不可以有任意的成员变量，也不能在接口中实现接口成员。接口不能被实例化。接口的成员默认是公共的，因此不允许成员加上修饰符。接口声明的基本格式：
 - **[特性][接口修饰符] interface 接口名 [: 基接口列表]**
 - **{**
 - 接口体
 - **}**

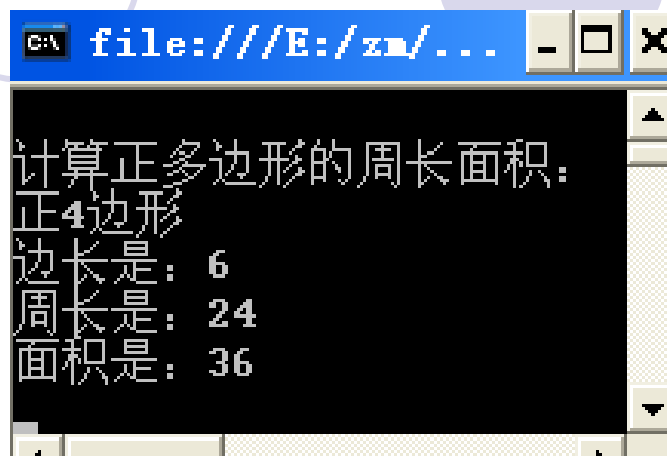
【例5-21】 在一个类中实现多个接口。运行界面如图5.17所示，代码如下：

```
● public interface Imultisideshap           //多边形接口
● {
●     double Area();                       //接口方法，计算多边形面积，
    接口成员前无修饰符
●     double Gramlength();                //接口方法，计算多边形周长
●     int Sides { get;}                  //接口中的属性，多边形的边数
● }
● public interface Idisplayresult //显示计算结果的接口
● {
●     void displayresult();
● }
```

- 
- `public class Square : Imultisideshap, Idisplayresult`
 - `//定义正方形的类，实现两个接口`
 - `{`
 - `private int sides;` `//边数`
 - `public int sidelength;` `//边长`
 - `public Square()` `//构造函数`
 - `{ sides = 4; }`
 - `public int Sides` `//实现接口Imultisideshap中的`
属性
 - `{`
 - `get { return sides; }`
 - `}`
 - `public double Area()` `//实现Imultisideshap接口中的`
方法Area()
 - `{ return ((double)(sidelength * sidelength)); }`
 - `public double Gramlength()` `//实现接口Imultisideshap中的方法`
Gramlength()
 - `{ return ((double)(Sides * sidelength)); }`

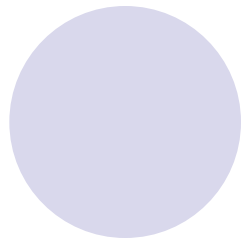
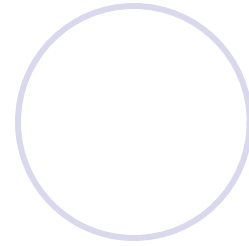
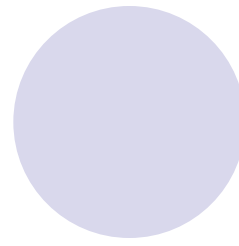
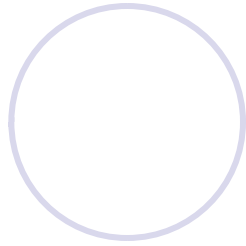
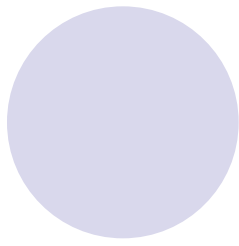


```
public void displayresult()//实现接口IDisplayresult中的方法
displayresult()
{
    Console.WriteLine("\n计算正多边形的周长面积: ");
    Console.WriteLine("正{0}边形",this .Sides);
    Console.WriteLine("边长是: {0}",this .sidelength);
    Console.WriteLine("周长是: {0}",this.Gramlength());
    Console.WriteLine("面积是: {0}",this .Area ());
}
}
class Test
{
    static void Main(string[] args)
    {
        Square tt = new Square();
        tt.sidelength = 6;
        tt.displayresult();
        Console.ReadLine();
    }
}
```

```
file:///E:/zm/...
计算正多边形的周长面积:
正4边形
边长是: 6
周长是: 24
面积是: 36
```

图5.17 例5-21 运行结果



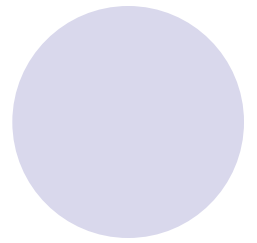
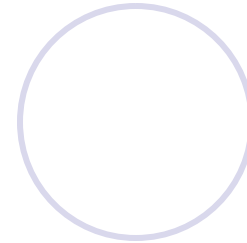
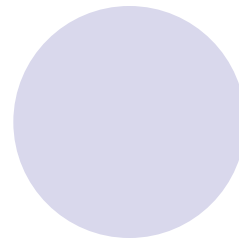
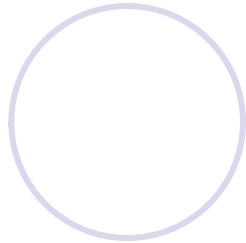
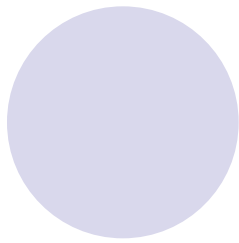
- 类可以同时有一个基类和零个以上的接口，并要将基类写在前面：
- `class ClassB:ClassA,IA,IB`
- `{ }`
- 指出接口成员所在的接口，则为显式接口成员。显式接口成员只能通过接口来调用。显式接口成员没被声明为**public**，这是因为这些方法都有着双重的身份。当在一个类中使用显式接口成员时，该方法被认为是私有方法，因此不能用类的实例调用它。但是，当将类的引用转型为接口引用时，接口中定义的方法就可以被调用，这时它又成为了一个公有方法。

【例5-22】 显示接口成员实现示例。运行结果如图5.18所示，代码如下：

```
• public interface Imultisideshap           //多边形接口
• {
•     double Area();                       //接口方法，计算多边形面积
•     double Gramlength();                 //接口方法，计算多边形周长
•     int Sides { get;}                   //接口中的属性，多边形的边数
•     void displayresult();                //接口方法
• }
• public interface Idisplayresult           //显示计算结果的接口
• { void displayresult(); }               //在Imultisideshap接口中也有同名方法
• public class Square : Imultisideshap, Idisplayresult
• //定义正方形的类，实现两个接口
• {
•     private int sides;                   //边数
•     public int sidelength;               //边长
•     public Square()                       //构造函数
•     { sides = 4; }
```

- public int Sides //实现接口Imultisideshap中的属性
- {
- get { return sides; }
- }
- public double Area() //实现Imultisideshap接口中的方法
- Area()
- { return ((double)(sidelength * sidelength)); }
- public double Gramlength() //实现接口Imultisideshap中的方法
- Gramlength()
- { return ((double)(Sides * sidelength)); }
- void Imultisideshap.displayresult() //这个显式实现是私有成员，不能用public
- {
- Console.WriteLine("\n计算正多边形的周长面积: ");
- Console.WriteLine("正{0}边形",this .Sides);
- Console.WriteLine("边长是: {0}",this .sidelength);
- Console.WriteLine("周长是: {0}",this.Gramlength());
- Console.WriteLine("面积是: {0}",this .Area ());
- }

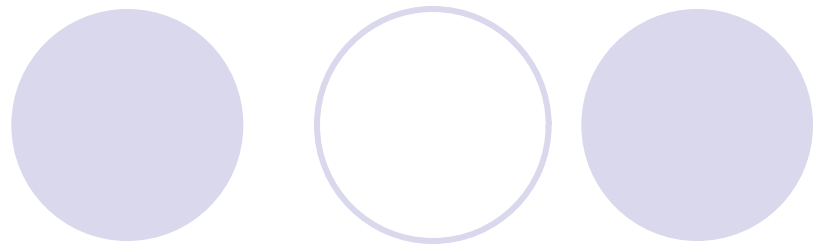
```
• void Idisplayresult.displayresult()  
• //因为两个接口中都有displayresult()方法，所以使用显式接口成员  
• //显式实现接口是私有成员，无public，注意是哪个接口的  
• { Console.WriteLine("显示图形"); }  
• }  
• class Test  
• {  
•     static void Main(string[] args)  
•     {  
•         Square tt = new Square();  
•         tt.sidelength = 6;  
•         //定义一个Imultisideshap 接口变量sh，对tt进行强制类型转换  
•         Imultisideshap sh=(Imultisideshap )tt;  
•         //定义一个Idisplayresult接口变量dis1，对tt进行强制类型转换  
•         Idisplayresult dis1=(Idisplayresult)tt;  
•         sh.displayresult(); //作为接口引用时为公有的  
•         dis1.displayresult();  
•         //tt.displayresult(); //类的对象不可以调用displayresult()，因为是  
私有的  
•         Console.ReadLine();  
•     }  
• }
```



```
file:///E:/zm/c...  
计算正多边形的周长面积:  
正4边形  
边长是: 6  
周长是: 24  
面积是: 36  
显示图形
```

图5.18 例5-22运行结果

5.7 委托与事件



- 1. 委托
- **C#**的委托相当于**C/C++**中的函数指针。函数指针用指针获取函数的入口地址，实现对函数的操作。委托与**C/C++**中的函数指针不同在于，委托是面向对象的，是引用类型，因此对委托的使用要先定义后实例化，最后才调用。
- 委托的声明使用关键字**delegate**，具体格式如下：
- **[委托修饰符] delegate**返回值类型 委托名(**[形参列表]**)
- 其中，委托修饰符包括访问修饰符和**new**，不能在同一个委托内多次使用同一个修饰符。修饰符**new**用于隐藏从基类继承而来的同名委托。访问修饰符**public**、**protected**、**internal**和**private**用于控制委托类型的可访问性。

- 如果一个方法和某委托相兼容，则这个方法必须具备如下条件：两者具有相同的参数数量、类型、顺序和参数修饰符，两者返回值类型也相同。例如：
- 定义一个委托：`delegate int D(int y, string zz);`
- 实例化：`D d1 = new D(a1.InstanceMethod);`
- 最后调用：`d1(5, "aaa");`图5.19 例5-23运行结果
- 通过委托D实现对方法InstanceMethod的调用，调用还必须有一个前提条件是：方法InstanceMethod的参数和定义D的参数一致，并且返回值都为int。
- 方法InstanceMethod定义：`public int InstanceMethod(int x, string tt)。`
- 委托的实例化中的参数既可以是非静态方法，也可以是静态方法。
- 可以在类中定义委托

【例5-23】 委托的使用。运行界面如图5.19所示，代码如下

[illegible]



```
class APP
```

```
{
```

```
    static void Main(string[] args)
```

```
    { A a1 = new A();      //创建一个A类对象
```

```
      Mydelegate d1 = new Mydelegate(A.staticmethod);
```

```
      //实例化一个delegate, 指向一静态方法
```

```
      d1(10);              //调用委托实例
```

```
      Mydelegate d2 = new Mydelegate(a1instancemethod);
```

```
      //指向一个实例方法
```

```
      d2(20);              //调用委托实例
```

```
      a1.calldele(new Mydelegate(a1instancemethod));
```

```
      //将委托实例作为参数来传递
```

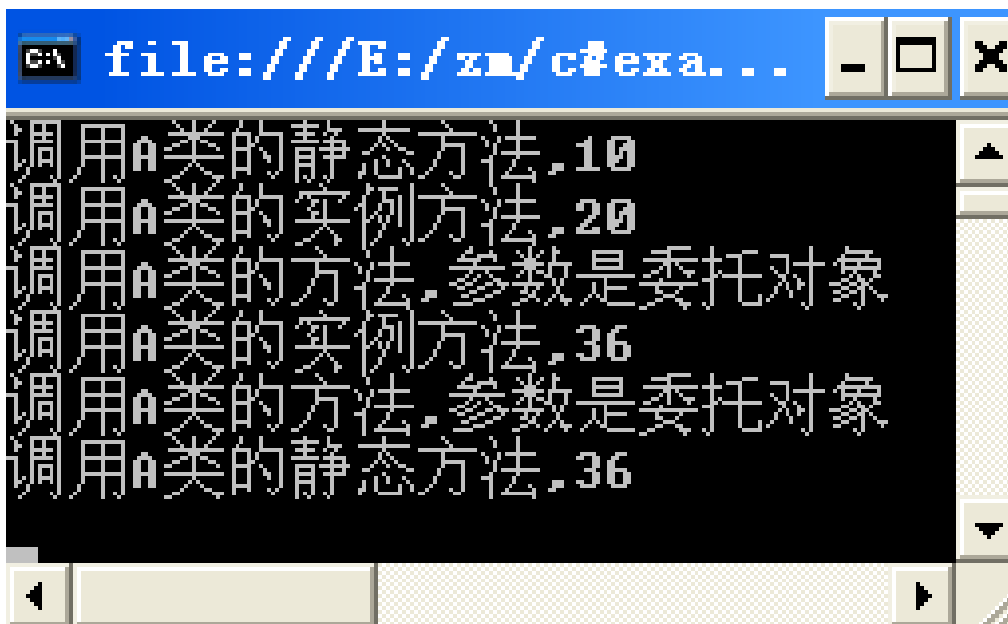
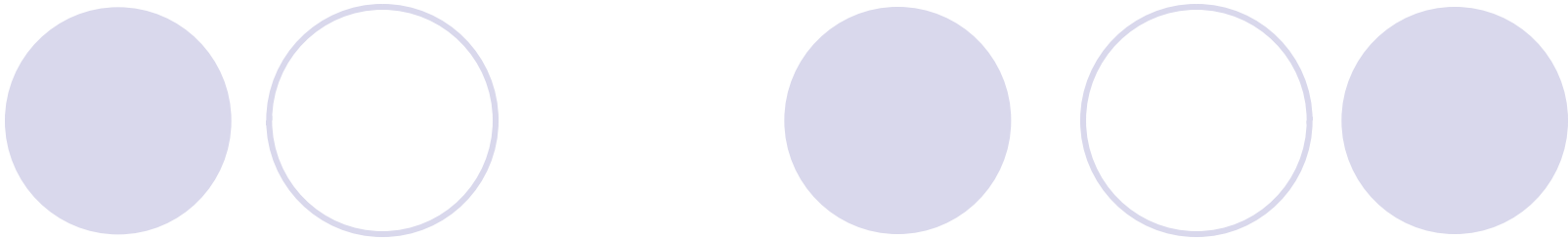
```
      a1.calldele(new Mydelegate (A.staticmethod));
```

```
      Console.ReadLine();
```

```
    }
```

```
  }
```

```
}
```



```
file:///E:/zm/c#exa...
调用A类的静态方法,10
调用A类的实例方法,20
调用A类的方法,参数是委托对象
调用A类的实例方法,36
调用A类的方法,参数是委托对象
调用A类的静态方法,36
```

图5.19 例5-23运行结果

相对于例5-23中一次委托只调用一个方法，一次委托也可以调用多个方法，称为多播。通过+和-运算符实现多播的增加或减少

- **【例5-24】** 在例5-23的基础上实现多播的增加或减少。
- `Mydelegate d3=d1+ d2; //多播d3由两个委托d1和d2组成`
- `d3(78); //调用d3，执行完d1所指向的方法后，执行d2所指向的方法`
- `d3= d3-d2; //多播d3减去委托d2，d3只有d1所指向的方法`
- `d3(96);`
- 使用委托匿名方法：在前面介绍的实例中，委托实例的方法都有具体的名称。**C#**支持匿名方法，即允许委托的关联代码以内联方式写入使用委托的位置，从而方便地将代码块直接绑定到委托实例中。
- 委托匿名方法的具体使用格式如下：
- `delegate(参数表)`
- `{处理代码块};`

【例5-25】 使用命名方法和匿名方法创建委托实例。运行结果如图5.20所示，代码如下：

```
namespace exweituo2
{
    public delegate void Daili(string s);
    class AA
    {
        public void fangfa(string kk)
        { Console.WriteLine(kk); }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Daili p1 = delegate(string j)
            //用匿名方法创建的委托实例，参数里面不是方法名
            {
                Console.WriteLine(j);
            };
            p1("调用匿名方法创建的委托\n"); //调用委托
            AA a1 = new AA(); //创建一个AA类的实例
            Daili p2 = new Daili(a1.fangfa); //用命名方法创建的委托实例
            p2("调用命名方法创建的委托");
            Console.ReadLine();
        }
    }
}
```

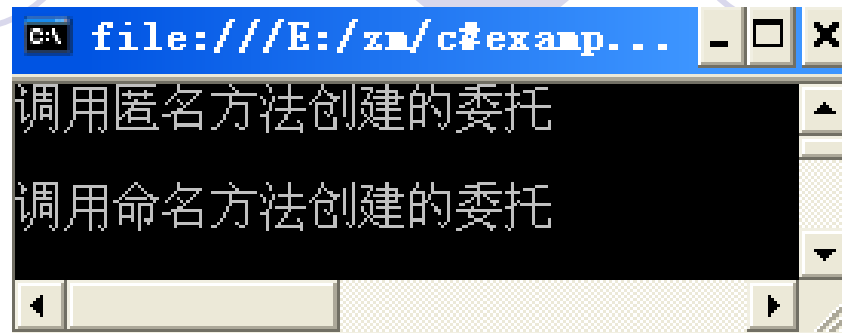
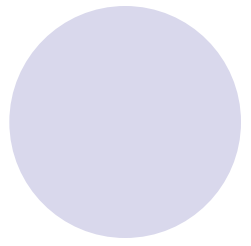
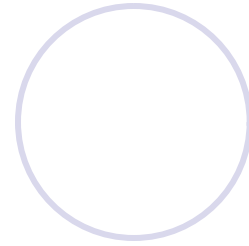
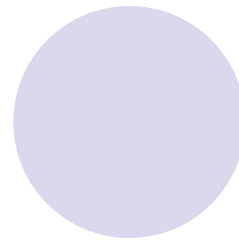
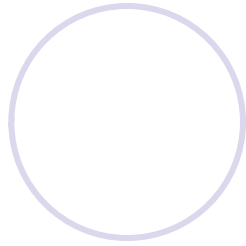
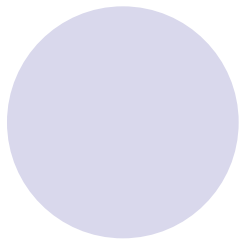


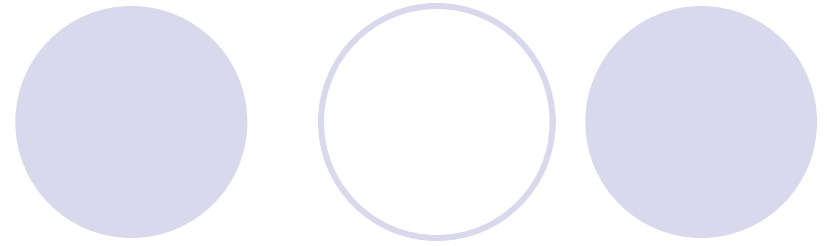
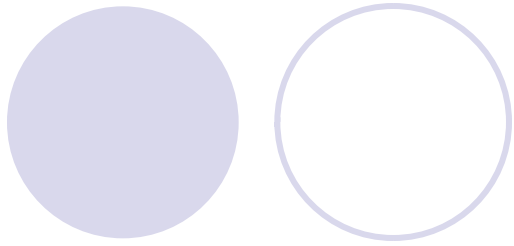
图5.20 例5-25运行结果

2. C#事件

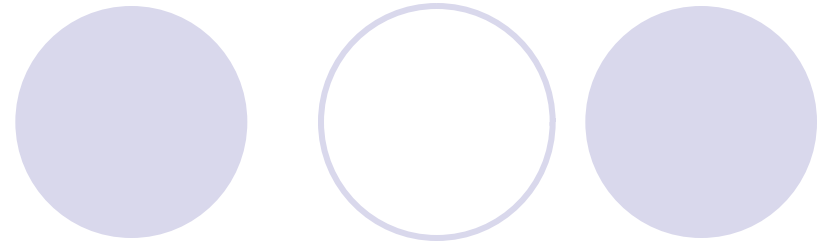
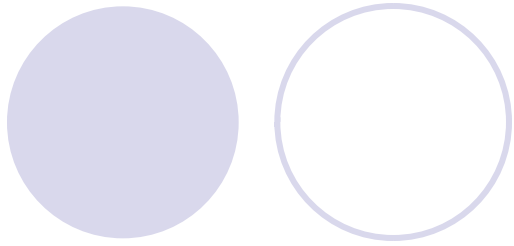
- 假设员工小李是张三的委托人，专门负责为张三发送资料给客户。当然小李可以采取多种措施来完成这个任务，比如发电子邮件或发传真。这些措施都是命名或匿名方法，必须在使用委托之前和委托关联。在这个场景中，可以采取另外的方式完成张三的任务。例如，张三可以对他的所有员工说“当我有要求时随时为我服务”，这样，一旦张三需要发送资料，张三的所有员工都会得到这个消息并为他服务。当然，如果不是张三的员工，他是不会为张三做任何事的。



- 从程序员的角度来看这个场景。当张三向外宣布需要发送资料时，事件发生了，张三就是事件的发布者，向外发布一个事件。张三的所有员工为事件的订户。他们依据事件进行相应的处理，如发邮件或传真，这个过程叫事件处理程序。同时，不是张三的下属则根本不关心事件发布者的话，因为他们没有预订这个事件。
- **C#**的事件处理采用了同样的方式。把事件信息通知给其他对象的对象称为发布方，注册到事件的对象称为订户。事件即为事件发布方向外界发出的通知，而事件处理程序是事件订户应事件发生而做出的反应(具体到程序上，就是针对事件而写的那些处理代码)。这里需要注意的是，订户必须预订该事件，位于订户对象内的事件处理程序才可能会被执行。孤立的事件处理程序不会被执行。



- C#的事件具有如下特点：
- 事件是基于委托的。
- 发布者确定何时引发事件，订户确定何时操作来响应该事件。
- 一个事件可以有多个订户，一个订户可处理来自多个发布者的多个事件。
- 没有订户的事件永远不会被调用。
- 事件通常用于通知用户操作(如图形用户界面中的按钮单击或菜单选择操作)。
- 如果一个事件有多个订户，当引发该事件时，会同步调用多个事件处理程序。
- 在.NET Framework类库中，事件是基于EventHandler委托和EventArgs基类的。



- **C#**处理事件的步骤(如图5.21所示)如下:
- (1) 定义事件;
- (2) 订阅事件;
- (3) 引发事件。

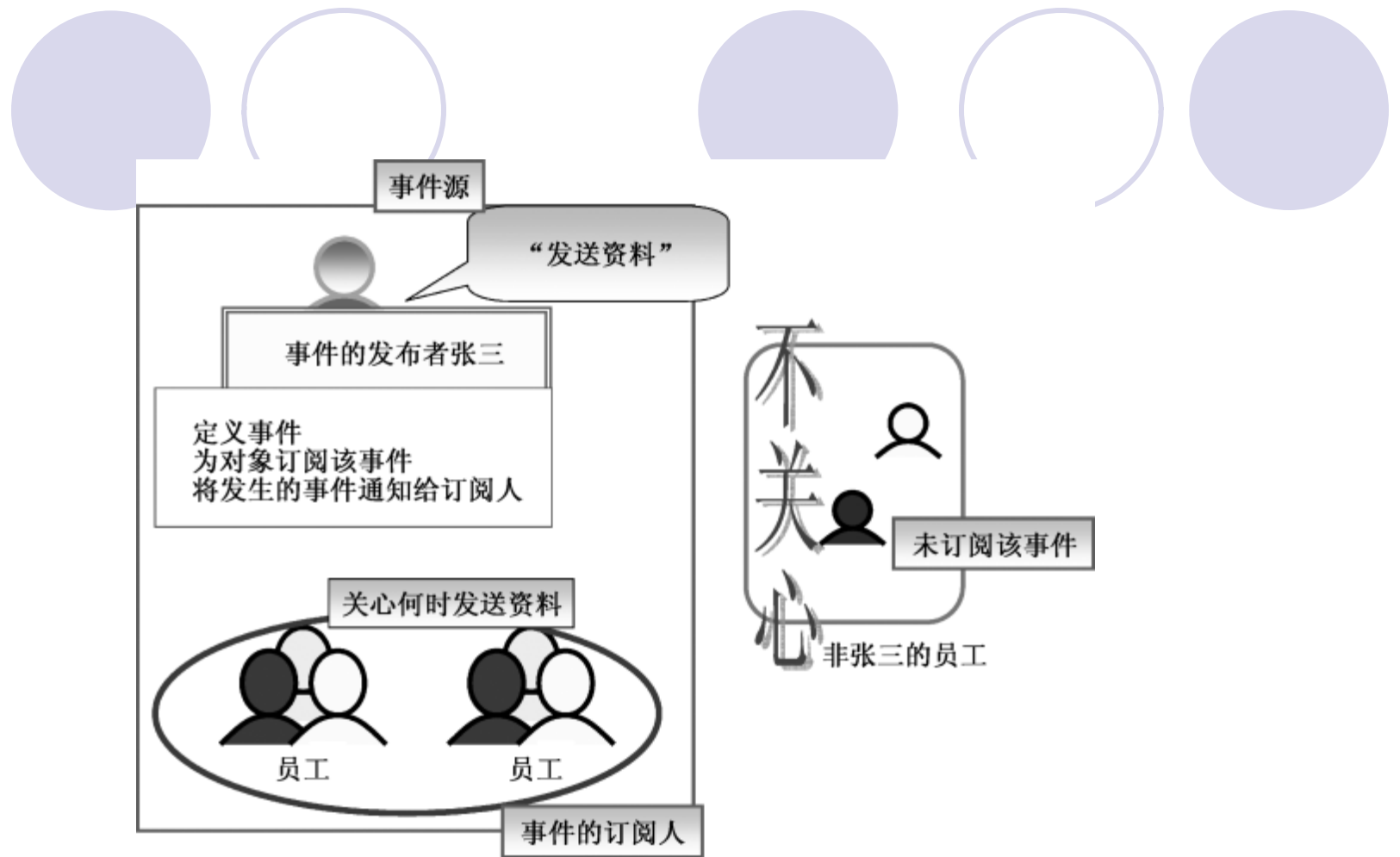



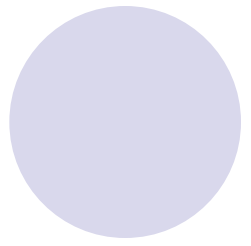
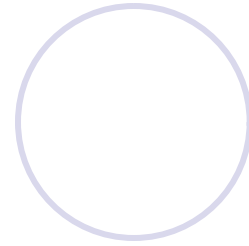
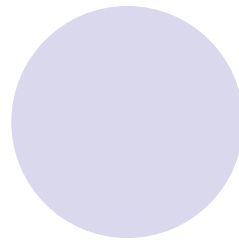
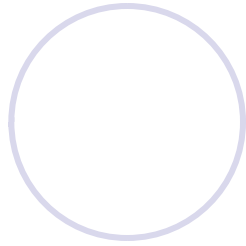
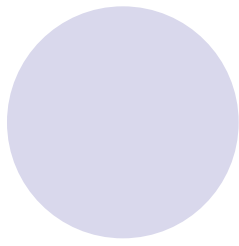
图5.21 处理事件的步骤

- 
- 定义事件：**C#**中的事件需要借助于委托，使用委托调用预订该事件的对象中的方法。当发布方引发事件时，可能会调用许多委托。一般而言，有多少订户，就应该调用多少委托。定义事件之前，需由发布方先定义委托，事件取决于委托。
 - 订阅事件：将发布方的事件和订户的事件处理程序进行绑定的过程，这样当事件被引发时，将执行响应的绑定代码。能否订阅事件取决于事件是否存在。如果事件存在，那么要给对象订阅事件，只需使用加法赋值运算符(+=)绑定一个当该瞬间被引发时将调用方法的委托即可；要取消预订，可以采用减法赋值运算符(-=)。

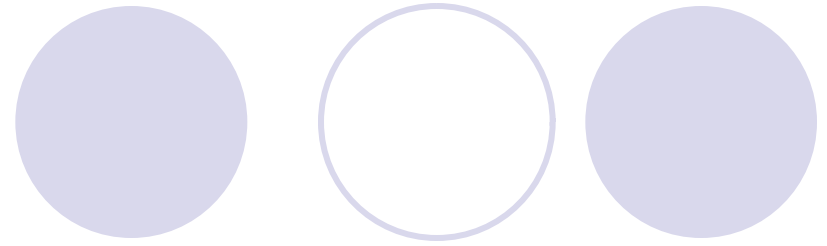
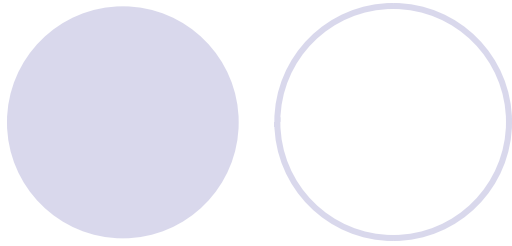
- 引发事件：要通知所有已预订了事件的对象，只需引发事件。引发事件与调用方法相似。
- 定义事件语法：[访问修饰符] **event** 委托名 事件名;
- 例如：
- 定义事件：
- `public delegate void delegateMe();`
- `private event delegateMe eventMe;`
- 订阅事件：
- `eventMe += new delegateMe(objA.Method);`
- `eventMe += new delegateMe(objB.Method);`
- 通知订阅对象：
- `if(condition)`
- `{`
- `eventMe(); //调用订阅特定事件的对象的所有委托`
- `}`

【例5-26】 使用事件实例，该程序用单步运行能更好地理解事件调用的全过程。运行结果如图5.22所示。

```
public class Master
{
    public delegate void AskEventHandler(); //定义委托
    public event AskEventHandler AskEvent; //定义事件
    public void Ask() //激活事件的方法，通知订阅对象
    {
        if (AskEvent != null)
        {
            Console.WriteLine("请帮我把资料发送给客户");
            AskEvent(); //调用订阅特定事件的对象的所有委托，即触发事件
        }
    }
}
```



- public static void Main()
- {
- Master me = new Master();
- Employee x1 = new Employee("小王");
- Employee x2 = new Employee("小李");
- me.AskEvent += new AskEventHandler(x1.DoWorkByTele);
- //订阅事件
- me.AskEvent += new AskEventHandler(x2.DoWorkByEmail);
- //订阅事件
- me.Ask(); //调用触发事件的方法
- me.AskEvent -= new AskEventHandler(x2.DoWorkByEmail);
- //取消订阅
- me.Ask(); //调用触发事件的方法
- }
- }



- public class Employee
- {
- private string name;
- public Employee(string n)
- {
- name = n;
- }
- public void DoWorkByTele()
- {
- Console.WriteLine("{0}将资料传真给用户",name);
- }
- public void DoWorkByEmail()
- {
- Console.WriteLine("{0}将资料Email给用户",name);
- }
- }

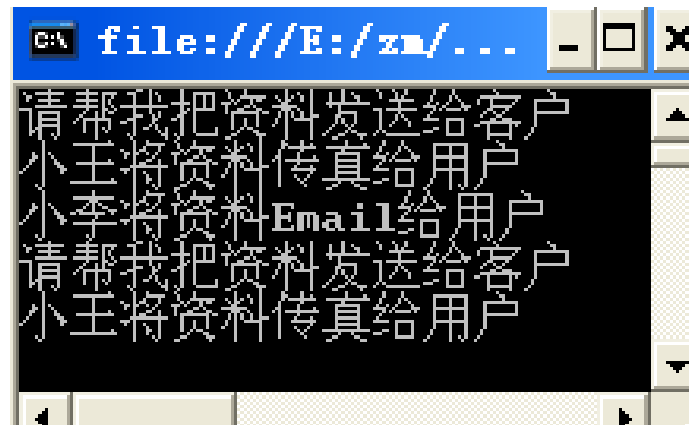
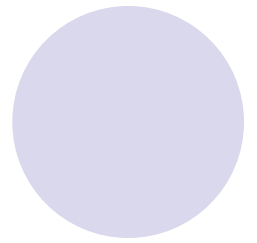
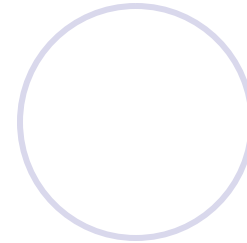
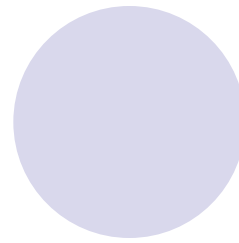
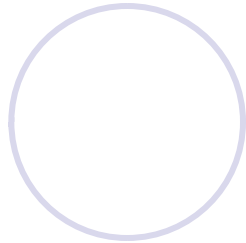
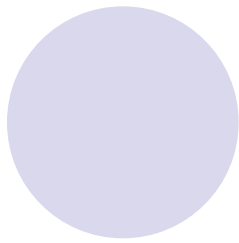


图5.22 例5-26运行结果