

C#程序设计

第7章 继承与多态

杨琦

西安交通大学

计算机教学实验中心

<http://ctec.xjtu.edu.cn>

授课内容

- 7.1 基类和派生类
- 7.2 继承
- 7.3 多态
- 7.4 虚方法
- 7.5 抽象类
- 7.6 接口
- 7.7 静态类和密封类
- 7.8 应用程序举例
- 7.9 运算符重载

7.1 基类和派生类

- 抽象、封装、继承和多态性是面向对象编程的四个主要特性
- 当创建一个类时，不需要完全重新编写新的数据成员和成员方法，只需要设计一个新的类，继承了已有的类的成员即可。这个已有的类被称为的基类，这个新的类被称为派生类。
- 继承就是在一个已存在的类的基础上建立一个新的类
- 已经存在的类称为父类或基类，新建立的类称为子类或派生类。
- 子类或派生类从父类那里获得其特性的现象称为继承

“狗” 和 “黑狗”

- 举个简单的例子：“狗” 和 “黑狗”。当谈论“狗”的时候，知道它是哺乳动物，有4条腿，1条尾巴，喜欢啃肉骨头，……现在谈论“黑狗”，人们会怎么说呢？
- 一种是说“黑狗是一种哺乳动物，有4条腿，1条尾巴，喜欢吃肉骨头，……，并且它的毛是黑色的”。
- 另一种是说：“黑狗就是黑毛狗”。

7.1 基类和派生类

- 派生类只能有一个直接基类。
- C# 不支持多重继承。但是，可以使用接口来实现多重继承。
- 继承的思想实现了 属于 (IS-A) 关系。例如，哺乳动物 属于 (IS-A) 动物，狗 属于 (IS-A) 哺乳动物，因此狗 属于 (IS-A) 动物。
- 继承是可传递的。

7.2 继承

- 派生类的一般定义形式为：
- 访问修饰符 `class` 派生类名称:父类名称
- {
- 成员列表
- }
- 一般来说，派生类的成员列表中只定义派生类新增加的成员。

7.2 继承

- 派生类的成员由两部分构成，一部分是从基类继承得到的，另一部分是自己定义的新成员。
- 派生类的成员访问属性仍然可以 `public`, `private`, `protected`, `internal`, `protected internal` 进行修饰。

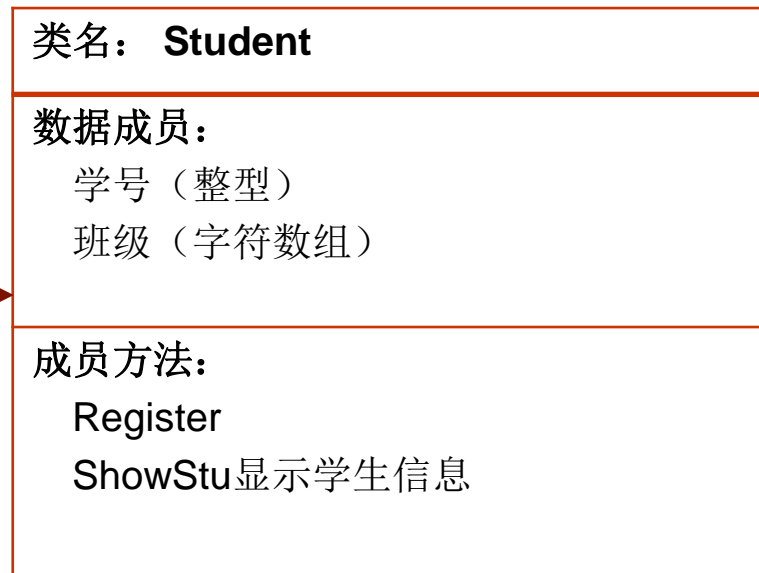
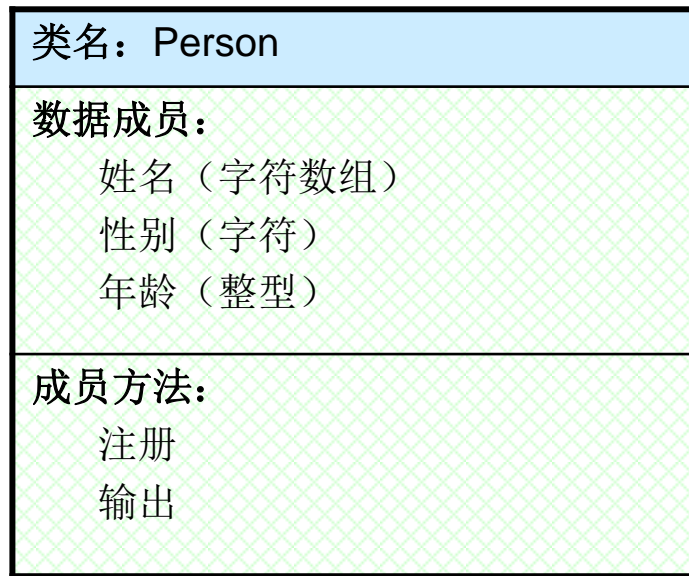
在设计派生类时，应注意：

- ①派生类可以继承基类中除了实例构造方法、析构方法和静态构造方法之外的所有其他成员，无论这些基类成员具有怎样的可访问性；
- ②继承具有传递性，派生类将会继承其所有直接基类和间接基类的成员；
- ③派生类可以根据实际需求在继承基类的基础上添加新的成员。但是，派生类并不能移除继承来的成员的定义。
- ④派生类可以通过声明具有相同名称（数据成员）或签名（方法成员）的新成员来隐藏被继承的同名基类成员，使被隐藏的成员在派生类中不可直接访问。

派生类的继承方式

<div>派生类 基类</div>	public继承		
	public	protected	private
public	√		
protected		√	
private			x

实例:公有继承(public)



例7-1基类（ Person ）及派生类（ Student ）



```
C:\WINDOWS\system32\cmd.exe
071011 计算机 张弓长 18 m
张弓长 18 m
请按任意键继续. . .
搜狗拼音 半:
```

```
using System;
class Person
{
    string Name;
    char Sex;
    int Age;
    public Person(string name, int age, char sex)
    {
        Name = name;
        Age = age;
        Sex = (sex == 'm' ? 'm' : 'f');
    }
    public void ShowMe()
    {
        Console.WriteLine(Name + '\t' + Age + '\t' + Sex);
    }
}
```

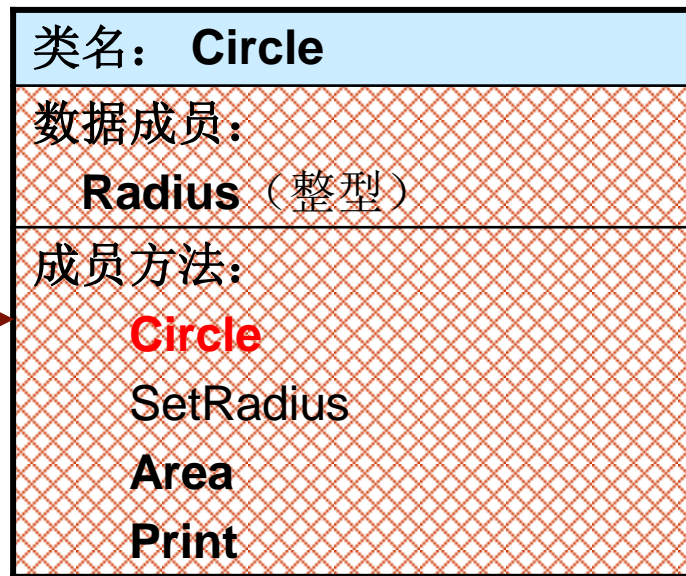
```
class Student : Person
{
    string Number;
    string ClassName;
    public Student(string classname, string number, string name, int age, char sex)
        : base(name, age, sex)
    {
        ClassName = classname;
        Number = number;
    }
    public void ShowStu()
    {
        Console.WriteLine(Number + '\t' + ClassName + '\t');
        base.ShowMe();
    }
}
```

```
class My
{
    static int Main()
    {
        Student stu = new Student("计算机", "071011", "张弓长", 18, 'm');
        stu.ShowStu();
        stu.ShowMe();
        return 0;
    }
}
```

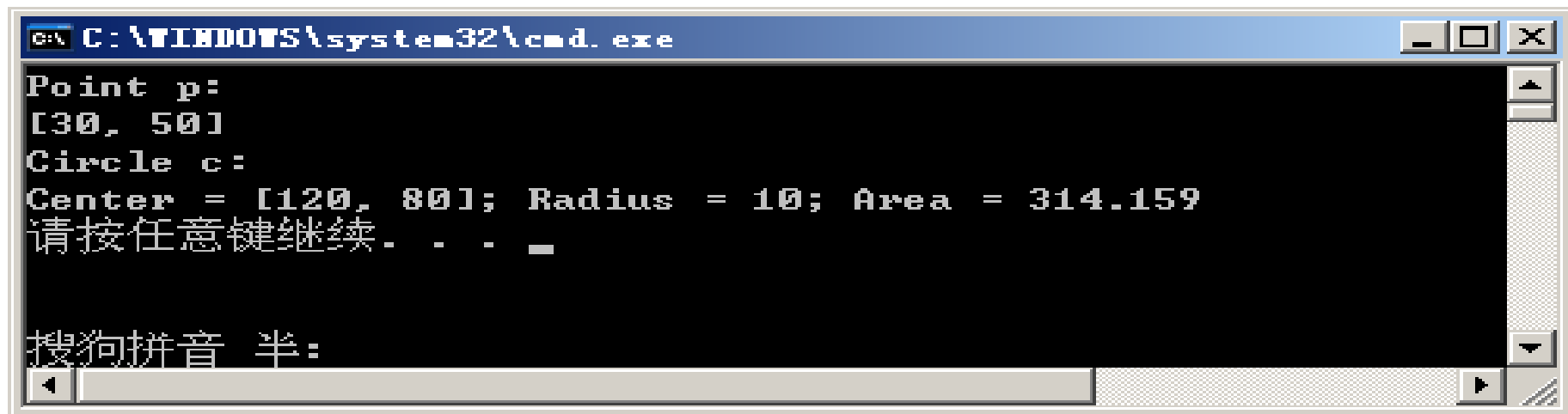
this 和 base

- base 关键字用于从派生类中访问基类的成员。
 1. 调用基类上已被其他方法重写的方法。
 2. 指定创建派生类实例时应调用的基类构造方法。
- this 和 base 关键字不能用在静态方法中

例7-2 Point类派生Circle类



例7-2 Point类派生Circle类



A screenshot of a Windows command prompt window. The title bar shows the path `C:\WINDOWS\system32\cmd.exe`. The window has standard Windows window controls (minimize, maximize, close) on the right. The command prompt displays the following text:

```
Point p:  
[30, 50]  
Circle c:  
Center = [120, 80]; Radius = 10; Area = 314.159  
请按任意键继续. . .  
搜狗拼音 半:
```

The text is displayed in a monospaced font. The prompt is waiting for a key press after the last line.

using System;

class Point

```
{  
    int x, y;  
    public Point( int a, int b ) {  
        SetPoint( a, b );  
    }  
    public void SetPoint( int a, int b ) {  
        x = a;    y = b;  
    }  
    public int GetX() { return x; }  
    public int GetY() { return y; }  
    public void Print() {  
        Console.Write( "[" + x + ", " + y + "]" );  
    }  
}
```

```

class Circle : Point {
    double radius;
    public Circle(int a,int b,double r): base(a,b)
        { SetRadius( r ); }
    public void SetRadius( double r )
        { radius = ( r >= 0 ? r : 0 );}
    public double GetRadius(){ return radius; }
    public double Area()
        { return 3.14159 * radius * radius;}
    new public void Print()
    {
        Console.Write( "Center = " );
        base.Print();
        Console.WriteLine( "; Radius = " + radius+ ";Area = " + Area() );
    }
}

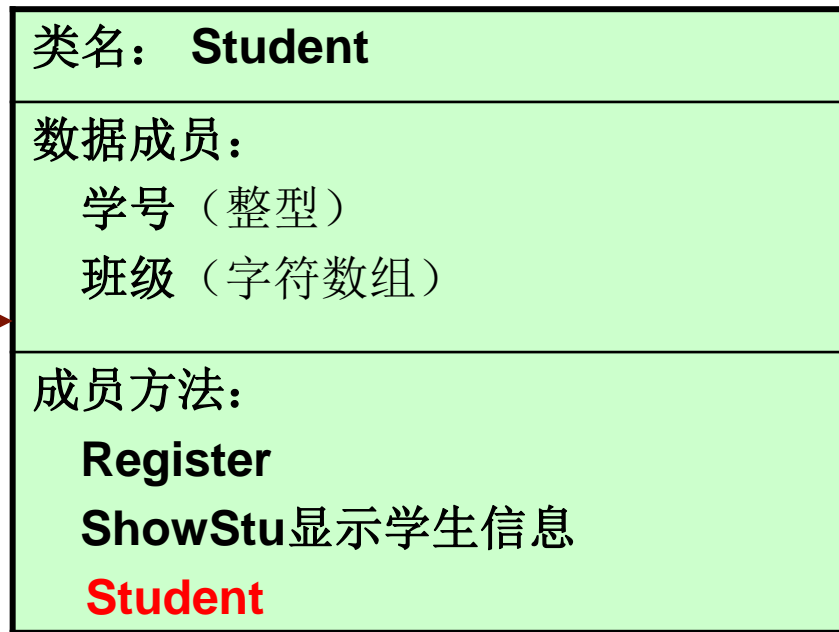
```

```
class My
{
    static int Main()
    {
        Point p=new Point(30,50);
        Circle c=new Circle(120,80,10.0);
        Console.WriteLine( "Point p: " );
        p.Print();
        Console.WriteLine( "\n\nCircle c: " );
        c.Print();
        return 0;
    }
}
```

覆盖性重写

- 在 C#中，如果在基类和派生类中同时声明了名称相同的方法，则视为派生类对基类的此方法进行重写。
- 重写分为覆盖性重写和多态性重写。

派生类构造方法和析构方法



派生类的构造方法与析构方法

- 派生类**没有**继承基类的构造方法和析构方法。
- 如果要对派生类的成员进行**初始化**，需要编写派生类的构造方法。
- 如果要**释放**派生类对象时同样需要调用派生类自己的析构方法。

1. 派生类的构造方法

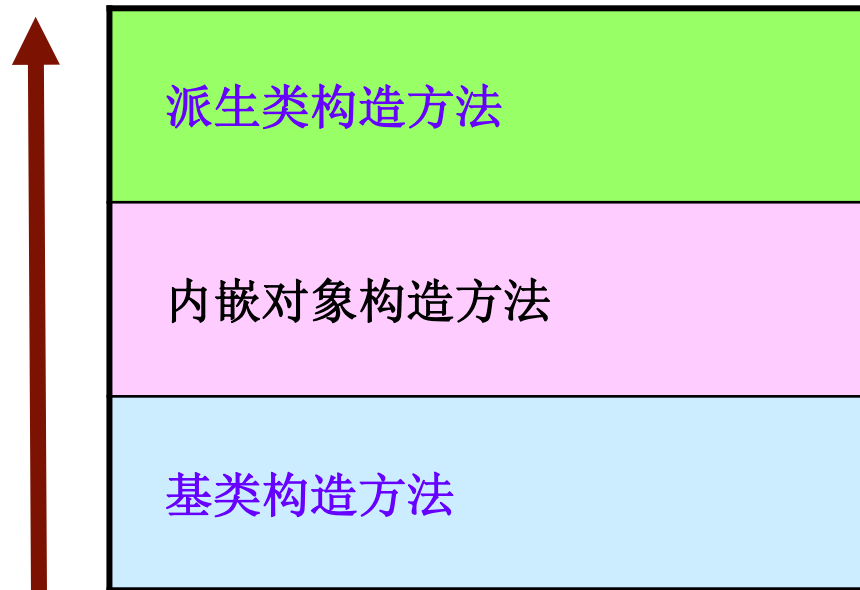
- 在设计派生类的构造方法时，需要考虑派生类新增成员和从基类继承的数据成员的初始化。
- 默认情况下，创建派生类的实例时，基类的无参数构造方法被调用。
- 可设置派生类使用指定的基类构造方法，派生类的构造方法定义如下：

```
public 派生类名称(参数列表):base(基类构造方法的参数列表)  
{ 方法体 }
```

```
public 派生类名称(参数列表)//隐式调用构造方法  
{ 方法体 }
```


派生类构造方法的执行次序

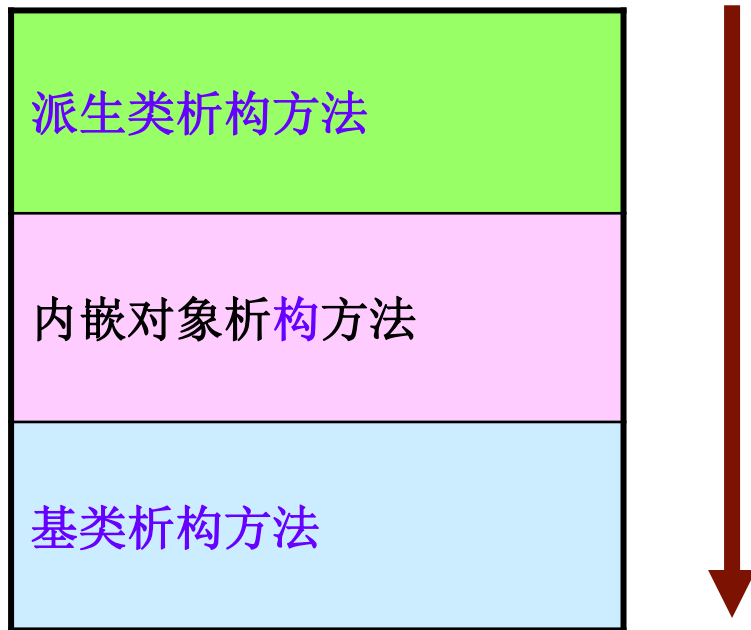
创建派生类的对象时，
会先调用基类的构造
方法，再调用派生类
的构造方法。



2 . 派生类的析构方法

- 在派生类中定义的析构方法用来完成对派生类中新增加的资源的清理工作，基类所申请的资源仍然由基类的析构方法来清理。
- 系统会自动调用派生类的析构方法和基类的析构方法完成对象的清理工作。
- 销毁派生类对象时，会先调用最派生类的析构方法，然后按照备份从低到高依次调用中间基类的析构方法，一直到备份最高的基类析构方法被执行。

析构方法



访问修饰符

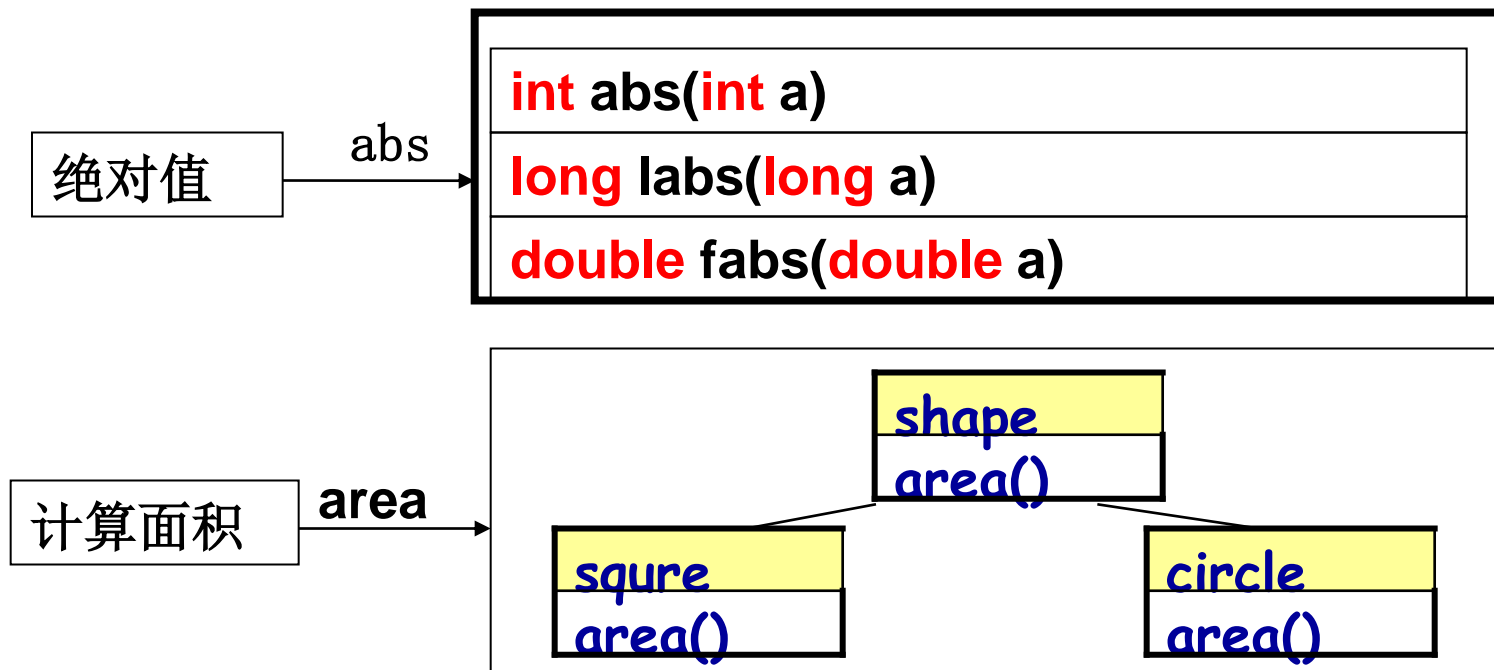
修饰符	应用于	说明
public	所有的类型或成员	任何代码均可以访问
protected	类型和内嵌类型的所有成员	程序集内外子类才可以访问
internal	类型和内嵌类型的所有成员	只能在包含它的程序集中访问
private	所有的类型或成员	本类内可以访问，其他不论程序集内外都不可以
protected internal	类型和内嵌类型的所有成员	只能在包含它的程序集和派生类型的代码中访问

7.3 多态、多态性重写、虚方法

- **同一操作**作用于**不同类**的实例、不同的类型将进行不同的解释，产生不同的结果，即为**多态**。
- C#可以通过继承实现多态性，实现的办法为：**对基类的成员进行隐藏或覆盖**。



7.3 多态、多态性重写、虚方法



1. 隐藏基类的成员

- C#规定派生类不能删除它继承的任何成员，但是在派生类中可以隐藏从基类继承的成员。
- 要隐藏一个基类的成员：
- ①对于数据成员来说，需要声明一个新的相同类型的成员，并使用相同的名称；对于方法成员来说，需要声明新的带有相同签名的方法成员；
- ②在派生类中声明用来隐藏基类成员的同名成员时需要使用new修饰符。
- 基类的静态成员也可以被派生类隐藏。

- 访问基类被隐藏的成员，可以使用基类访问表达式。基类访问表达式如下所示：
- **base.** 基类成员名

2. 覆盖基类的成员

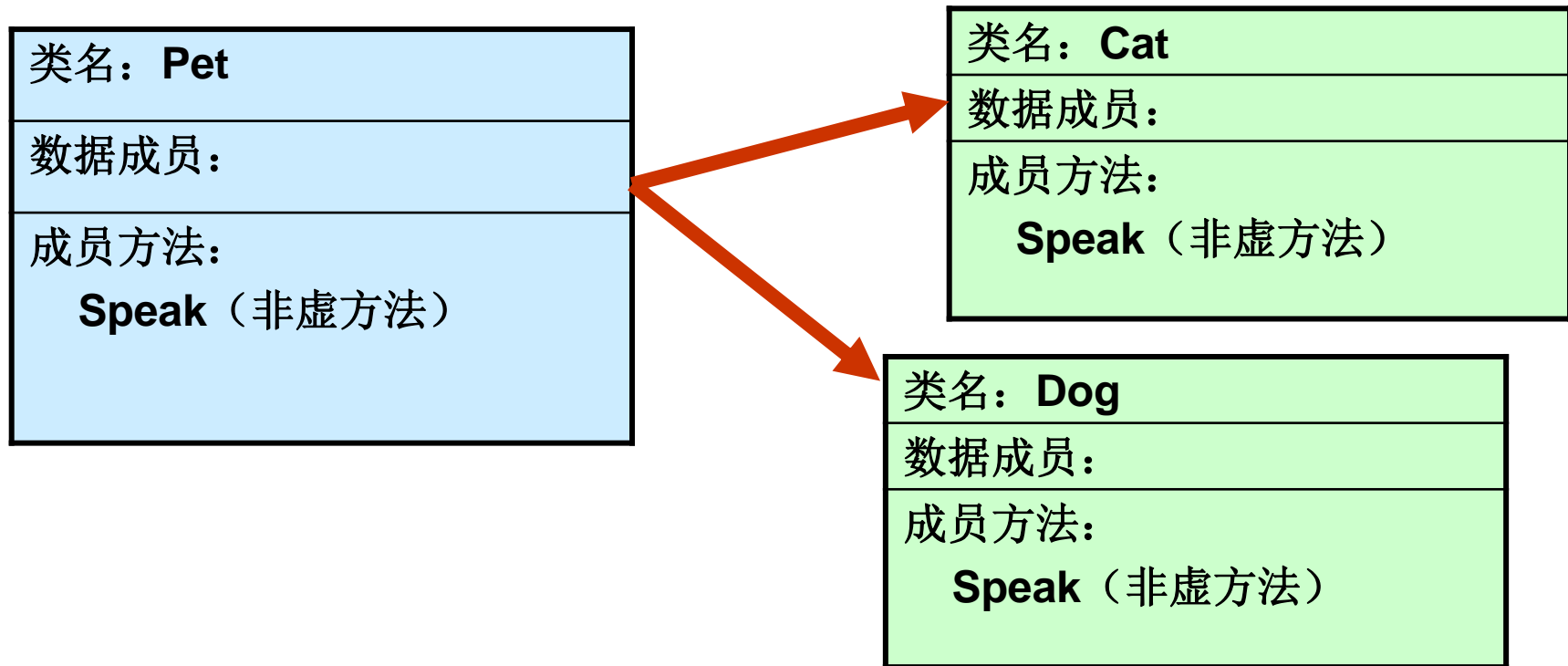
- 如果有一个派生类对象的引用，使用类型转换运算符把该引用转换成基类类型，就可以获取该对象基类部分的引用，格式如下所示：
- (基类名)派生类对象名
- 将派生类对象的引用转换成基类引用来访问派生类对象时，得到的是基类的成员。
- 虚方法提供了使基类引用访问“升至”派生类内的方法。
- 虚方法即为使用关键字virtual声明的实例方法；没有使用关键字virtual声明的方法称为非虚方法。

动态多态性

假设定义了基类Pet，从Pet类派生了Cat、Dog类

```
static void Main() {  
    Pet p;  
    int x;  
    x=Convert.ToInt32(Console.ReadLine());  
    if(x>0)  
        p=new Cat();  
    else  
        p=new Dog();  
    p.Speak();  
    .....  
}
```

例7-4 覆盖性重写



例7-4 覆盖性重写

```
1.  using System;
2.  class Pet {
3.      public void Speak() {
4.          Console.WriteLine("How does a pet speak ?");
5.      }
6.  };
7.  class Cat : Pet {
8.      new public void Speak() {
9.          Console.WriteLine("miao!miao!");
10.     }
11. };
```

例7-4 覆盖性重写

```
1.  class Dog : Pet  {  
2.      new public void Speak()  {  
3.          Console.WriteLine("wang!wang!");  
4.      }  
5.  };
```

例7-4 覆盖性重写

```
1.  class My{
2.      static int Main() {
3.          Pet p1 = new Pet();
4.          Dog dog1 = new Dog();
5.          Cat cat1 = new Cat();
6.          p1 = dog1;
7.          p1.Speak();
8.          p1 = cat1;
9.          p1.Speak();
10.         return 0;
11.     }
12. }
```

例7-4 覆盖性重写

- 输入和输出
- How does a pet speak ?
- How does a pet speak ?

7.4 虚方法

- 虚方法声明:

- virtual 返回值类型 方法名 (参数表)
- {
- 方法体
- }

- 注意事项:

- (1) 只有类的成员方法才能说明为虚方法
- (2) 静态成员方法不能是虚方法
- (3) 内联方法不能是虚方法

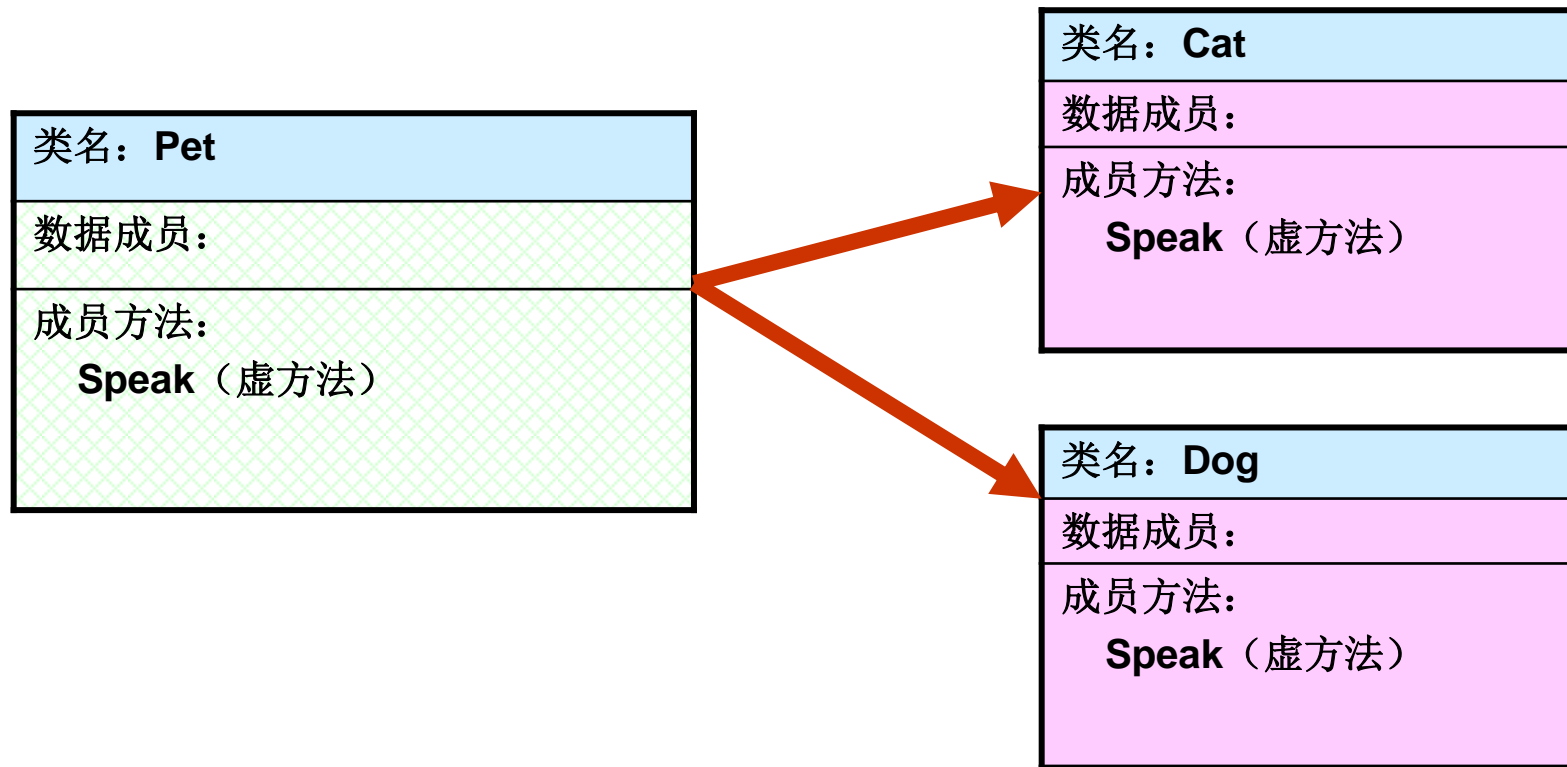
7.4 虚方法

- 借助虚方法使得基类引用调用派生类的方法需要满足下列条件：
- ①派生类的方法和基类的方法具有相同的方法签名和返回类型；
- ②基类的方法必须使用`virtual`声明为虚方法；
- ③派生类的方法需要使用`override`标注为覆写方法。

7.4 虚方法

- 使用虚方法和覆写方法时还需注意：
- ①虚方法和覆写方法必须具有相同的可访问性；
- ②不能覆写静态方法和非虚方法；
- ③方法、属性、索引以及事件都可以使用virtual和override进行声明和覆写；
- ④覆写方法可以出现在继承的任意层次上。
- 当使用对象的基类引用调用一个覆写方法时，方法的调用被沿派生层次追溯到标记为override的方法的最派生版本，或称为继承序列中辈分最小的版本。

例7-5 虚方法实现多态性



【例7-5】虚方法实现多态性

- 输入和输出
- wang!wang!
- miao!miao!

【例7-5】 虚方法实现多态性

```
1.  using System;
2.  class Pet {    //基类
3.      public virtual void Speak()
4.      {
5.          Console.WriteLine("How does a pet speak ?");
6.      }
7.  }
8.  class Cat : Pet { //派生类
9.      public override void Speak()
10.     {
11.         Console.WriteLine("miao!miao!");
12.     }
```

【例7-5】虚方法实现多态性

```
1. class Dog : Pet {  
2.     public override void Speak() {  
3.         Console.WriteLine("wang!wang!");  
4.     }  
5. }
```

【例7-5】 虚方法实现多态性

```
1.  class My{
2.      static int Main()  {
3.          Dog dog1 = new Dog();
4.          Cat cat1 = new Cat();
5.          Pet p1=new Pet() ;
6.          p1 = dog1;
7.          p1.Speak();
8.          p1 = cat1;
9.          p1.Speak();
10.         return 0;
11.     }
12. }
```

7.5 抽象类

- 抽象类即为使用`abstract`关键字修饰的类。
- 抽象类的内部可能包括使用`abstract`修饰的没有实现的虚方法，称为抽象方法。
- 抽象类是不完整的类，它只能用做基类来派生出其他类，其中包含的抽象方法必须在每个非抽象派生类中重写。

7.5 抽象类

- 使用抽象类时需注意：
- ①抽象类不能直接实例化，并且对抽象类使用 `new` 运算符会导致编译时错误；
- ②允许但不要求抽象类包含抽象成员，但是包含抽象成员类必须声明为抽象类；
- ③抽象类声明时不能使用 `sealed` 修饰符；
- ④当从抽象类派生非抽象类时，这些非抽象类必须具体实现所继承的所有抽象成员，从而重写那些抽象成员；
- ⑤对于每个抽象类，至少应提供一个具体的继承类型。

【例7-6】抽象宠物类的另一种用法



```
using System;
abstract class Pet
{
    protected int Age;
    protected string Name;
    protected string Color;
    protected string Type;
    public Pet(string name, int age, string color) {
        Name = name;
        Age = age;
        Color = color;
        Type = "pet";
    }
    public abstract void Speak();
    public abstract void GetInfo();
}
```

```
class Cat : Pet{
    public Cat(string name, int age, string color) : base(name, age, color) {}
    public override void Speak() {
        Console.WriteLine(" Sound of speak : miao!miao!");
    }
    public override void GetInfo() {
        Console.WriteLine(" The cat's name : " + Name);
        Console.WriteLine(" The cat's age : " + Age);
        Console.WriteLine(" The cat's color: " + Color);
    }
}

class Dog : Pet
{
    public Dog(string name, int age, string color)
        : base(name, age, color) {}
}
```

```
public override void Speak() {  
    Console.WriteLine(" Sound of speak : wang!wang!");  
}  
  
public override void GetInfo() {  
    Console.WriteLine(" The dog's name : " + Name);  
    Console.WriteLine(" The dog's age : " + Age);  
    Console.WriteLine(" The dog's color: " + Color);  
}  
}  
  
class My{  
    static int Main() {  
        Pet p1 = new Cat("MiKey", 1, "Blue");  
        p1.GetInfo();    p1.Speak();  
        Pet p2 = new Dog("BenBen", 2, "Black");  
        p2.GetInfo();  
        p2.Speak();  
        return 0;  
    }  
}
```

例7.6 抽象宠物类的实现

- 输入和输出
- The cat's name : MiKey
- The cat's age : 1
- The cat's color: Blue
- Sound of speak : miao!miao!
- The dog's name : BenBen
- The dog's age : 2
- The dog's color: Black
- Sound of speak : wang!wang!

7.6 接口 (Interface)

- 接口定义了所有类继承接口时应遵循的语法合同。接口定义了语法合同 "是什么" 部分，派生类定义了语法合同 "怎么做" 部分。
- 接口定义了属性、方法和事件，这些都是接口的成员。接口只包含了成员的声明。成员的定义是派生类的责任。接口提供了派生类应遵循的标准结构。
- 抽象类在某种程度上与接口类似。
- C# 不支持多重继承。但是，您可以使用接口来实现多重继承。

7.6.1 接口的声明

- 接口把所有的方法和属性都掏空了，接口内部只有声明没有实现代码。接口的声明语法如下：
- [修饰符] **interface** 接口名[:父接口列表]
- {
- //接口体
- }
- 接口就是一组类或结构的一种实现形式约定。

7.6.1 接口的声明

- 声明接口时，应该遵守下面的规则：
- ①接口声明**不能包括**数据成员；
- ②接口声明**只能包括**成员类型为方法、属性、事件和索引的非静态成员方法的声明；
- ③接口的方法成员声明**不能包括**任何实现代码，每个方法成员声明的主体后必须使用分号；
- ④按照惯例，接口的名称必须从大写的I开始； ⑤与类和结构相似，接口声明可以分割成分部接口声明； ⑥接口声明可以使用所有的访问修饰符**public**、**protected**、**internal**和**private**；
- ⑦接口的成员是隐式**public**的，不允许有任何访问修饰符。

7.6.2 接口的实现

- 如果类从基类继承并实现接口，则基类列表中基类名称必须放在任何接口之前。
- C#支持一个类或结构实现**多个接口**，所有实现的接口必须在基类列表中用逗号分隔。
- 如果一个类实现了多个接口，并且其中一些接口有相同签名和返回类型的成员，则类可以实现单个成员来满足所有包含重复成员的接口，也可以通过**显式实现**方式分别实现每个接口的成员。

【例7-6】定义一个宠物接口

- 输入和输出
- 我的声音: miao!miao!
- 我的声音: wang!wang!

【例7-6】 定义一个宠物接口

```
using System;
interface Pet
{
    void Speak();    void ShowMe();}
class Cat : Pet
{
    public void Speak()
    { Console.WriteLine("miao!miao!"); }
    public void ShowMe()
    { Console.Write("我的声音: ");
      Speak();
    }
}
```

【例7-6】 定义一个宠物接口

```
class Dog : Pet
{
    public void Speak()
    { Console.WriteLine("wang!wang!"); }
    public void ShowMe()
    {
        Console.Write("我的声音: ");
        Speak();
    }
}
```

【例7-6】定义一个宠物接口

```
class Program
{
    static void Main()
    {
        Cat cat1 = new Cat();
        Dog dog1 = new Dog();
        cat1.ShowMe();
        dog1.ShowMe();
    }
}
```

7.7 静态类和密封类

- 使用static修饰符声明的类称为静态类。
- 静态类是密封的，不能实例化，不能用作类型，而且仅可以包含静态成员。
- 使用静态类时需要遵守下面的规则，违反这些规则时将导致编译时错误：
 - ①静态类不可包含 sealed 或 abstract 修饰符；
 - ②静态类不能显式指定基类或所实现接口的列表，它隐式的从object类继承；
 - ③静态类只能包含静态成员；
 - ④静态类不能含有声明的可访问性为protected或protected internal的成员。

7.7 静态类和密封类

- 静态类没有实例构造方法。
- 静态类可以包含静态构造方法。如果非静态类包含需要进行重要的初始化的静态成员，也应定义静态构造方法。
- 静态类的成员并不会自动成为静态的，成员声明中必须显式包含一个 **static** 修饰符（常量和嵌套类型除外）。

7.7 静态类和密封类

- 如果类定义时使用 `sealed` 修饰符进行修饰，则说明该类是一个密封类。
- 密封类只能被用作独立的类，不能从密封类派生出其他类。
- 如果一个密封类被指定为其他类的基类，则会发生编译时错误。密封类的一般定义形式：
- `sealed class` 类名
- { 成员列表 }
- 密封类不能同时声明为抽象类。
- 当调用密封类实例的虚方法成员时可以转换为非虚调用来处理。
- 可以使用密封来限制开发人员扩展某些指定的框架。

密封类

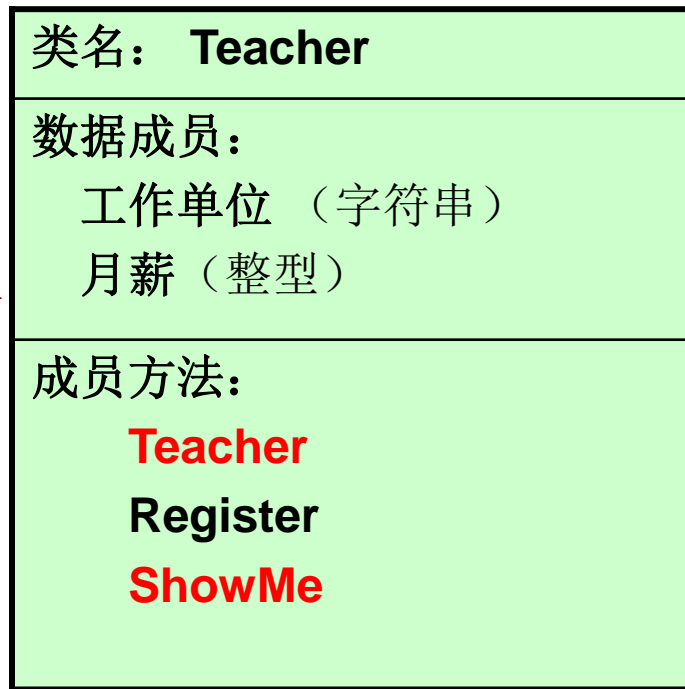
- 如果类满足如下条件，则应将其密封：
 - ① 类是静态类；
 - ② 类包含带有安全敏感信息的继承的受保护成员；
 - ③ 类继承多个虚成员，并且密封每个成员的开发和测试开销明显大于密封整个类；
 - ④ 类是一个要求使用反射进行快速搜索的属性。密封属性可提高反射在检索属性时的性能。
- **不要在密封类中声明受保护成员或虚成员。** 因为如果类是密封的，则它不能有派生类。受保护成员只能从派生类进行访问，虚成员也只能在派生类中重写。

分部类

- 如果一个类的内容很长，则可以将类的声明分割成几个部分来声明，每个部分称为一个分部类。
- 每个分部类的声明中都含有一些类成员的声明，这些分部类可以在一个文件中，也可以在不同文件中。
- 将类分割成几个分部类声明时，每个局部必须被标为 `partial class`，而不是单独的关键字 `class`。
- 除了必须添加类型修饰符 `partial` 之外，分部类的声明和普通类声明相同。
- 组成所有类的分部类声明必须一起编译，并且这些分部类分开声明和在一起声明应该具有相同的含义。
- 分部类的各个部分必须具有相同的可访问性，如 `public`、`private` 等。

7.8 应用程序举例

例7-7 Person->Teacher



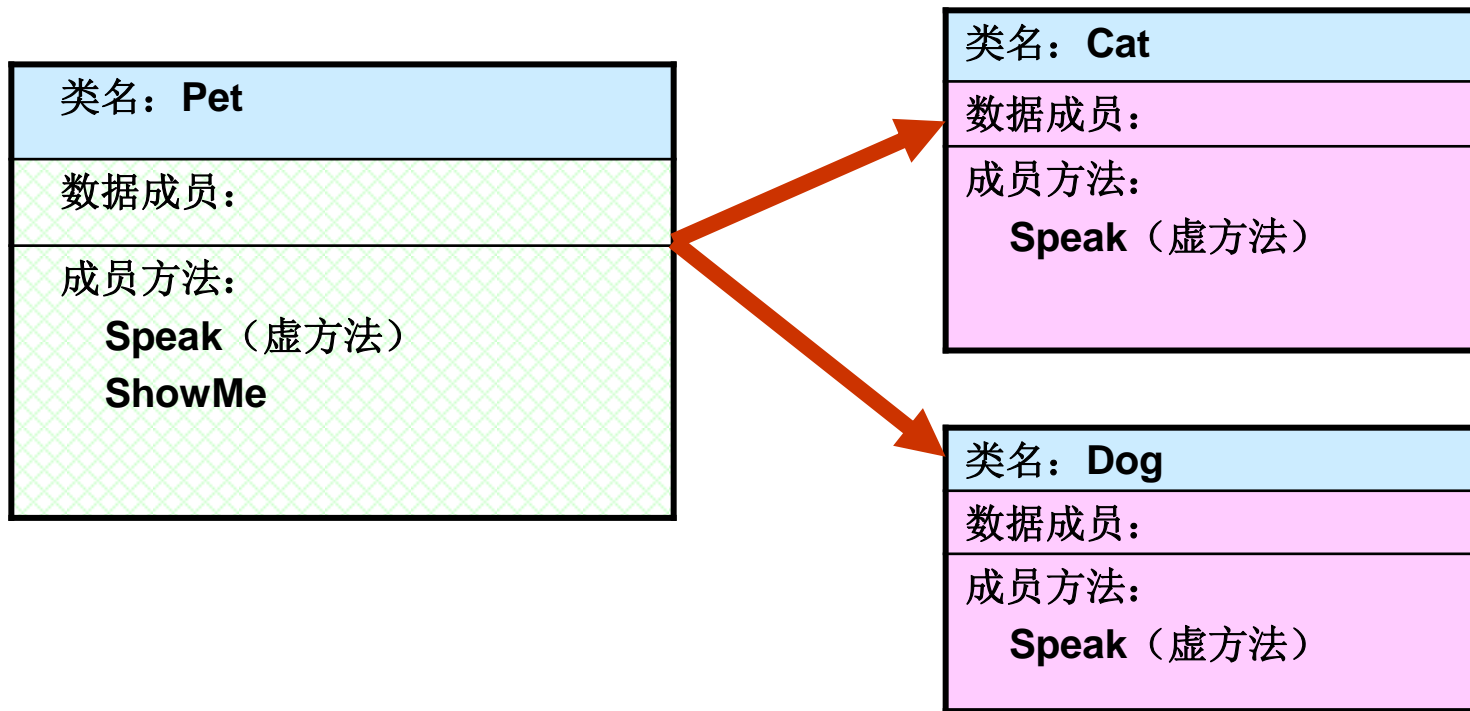
```
using System;
class Person {
    string Name;
    char Sex;
    int Age;
    public Person(string name, int age, char sex) {
        Name = name;
        Age = age;
        Sex = (sex == 'm' ? 'm' : 'f');
    }
    public void ShowMe() {
        Console.WriteLine(" 姓 名: " + Name);
        Console.WriteLine(" 性 别: " + (Sex == 'm' ? "男" : "女"));
        Console.WriteLine(" 年 龄: " + Age);
    }
}
```

```
class Teacher : Person
{
    string Dept;
    int Salary;
    public Teacher(string name, int age, char sex, string dept, int salary) : base(name,
age, sex)
    {
        Dept = dept;
        Salary = salary;
    }
    new public void ShowMe()
    {
        base.ShowMe();
        Console.WriteLine(" 工作单位: " + Dept);
        Console.WriteLine(" 月 薪: " + Salary + "\n");
    }
}
```

```
class My
{
    static int Main()
    {
        Teacher emp1 = new Teacher("章立早", 38, 'm', "电信学院", 8500);
        emp1.ShowMe();
        return 0;
    }
}
```



【例7-8】抽象宠物类的另一种用法



【例7-8】抽象宠物类的另一种用法

- 输入和输出
- wang!wang!
- miao!miao!

【例7-8】 抽象宠物类的另一种用法

```
1.  using System;
2.  abstract class Pet{
3.      public abstract void Speak();
4.      public void ShowMe() {
5.          Console.WriteLine( "我的声音： ");
6.          Speak();
7.      }
8.  }
9.  class Cat : Pet{
10.     public override void Speak()
11.     { Console.WriteLine( "miao!miao!" ); }
12. }
```

【例7-8】 抽象宠物类的另一种用法

```
1.  class Dog : Pet {  
2.      public override void Speak()  
3.      { Console.WriteLine( "wang!wang!"); }  
4.  }  
5.  class My {  
6.      static void Main()  {  
7.          Cat cat1 = new Cat();  
8.          Dog dog1 = new Dog();  
9.          cat1.ShowMe();  
10.         dog1.ShowMe();  
11.     }  
12. }
```

7.11 运算符重载

- 背景

- C#中预定义的运算符的操作对象只能是基本数据类型，而用户自定义类型（比如类）也需要有类似的运算操作
- 能否对运算符重新定义，赋予已有符号以新的功能？

- 运算符重载

- “借用”已有的运算符，对其赋予多重含义，对自定义的类对象实现自定义的新功能！
- ——同一个运算符作用于不同类型数据导致不同的行为

- 运算符重载实质就是方法重载！

自定义类中运算符重载的形式(2种)

- 重载为类的公有成员方法

<方法类型> **operator** <运算符>(<形参表>)
{ <方法体>; }

- 重载为类的友元方法

friend <方法类型> **operator** <运算符>(<形参表>)
{ <方法体>; }

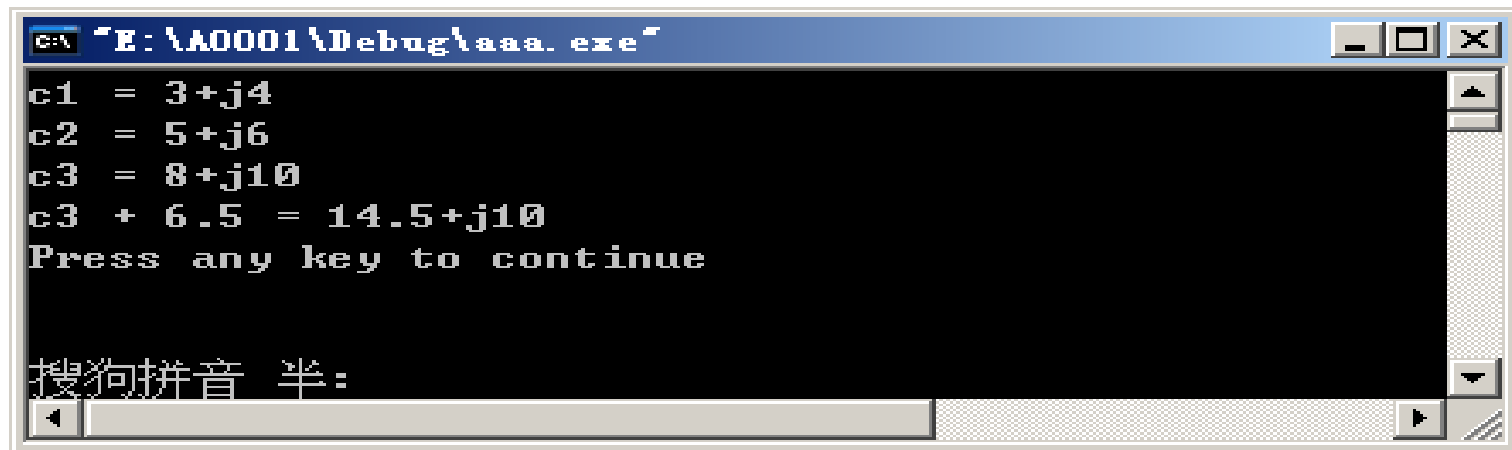
- 二者区别——参数个数不同!

- 成员方法方式重载：参数个数比原来运算数少一个(后缀++、--除外)
- 友元方法方式重载：参数个数与原运算数个数相同,且至少有一个参数是说明该友元的类或是该类的引用

可重载和不可重载运算符

运算符	描述
<code>+, -, !, ~, ++, --</code>	这些一元运算符只有一个操作数，且可以被重载。
<code>+, -, *, /, %</code>	这些二元运算符带有两个操作数，且可以被重载。
<code>==, !=, <, >, <=, >=</code>	这些比较运算符可以被重载。
<code>&&, </code>	这些条件逻辑运算符不能被直接重载。
<code>+=, -=, *=, /=, %=</code>	这些赋值运算符不能被重载。
<code>=, ., ?:, ->, new, is, sizeof, typeid</code>	这些运算符不能被重载。

【例7-9】复数加法运算



A screenshot of a Windows command prompt window. The title bar shows the file path "E:\A0001\Debug\aaa.exe". The command prompt displays the following text:
c1 = 3+j4
c2 = 5+j6
c3 = 8+j10
c3 + 6.5 = 14.5+j10
Press any key to continue
Below this text, the Chinese characters "搜狗拼音 半:" are visible, likely from an input method editor. The window has standard Windows controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

```
C:\E:\A0001\Debug\aaa.exe  
c1 = 3+j4  
c2 = 5+j6  
c3 = 8+j10  
c3 + 6.5 = 14.5+j10  
Press any key to continue  
搜狗拼音 半:
```


【例7-9】 复数加法运算

```
1.  using System;
2.  class Complex {
3.      double real = 0, imag = 0;
4.      public Complex(double r, double i)  {
5.          real = r;
6.          imag = i;
7.      }
8.      public double Real() { return real; }
9.      public double Imag() { return imag; }
10.     public static Complex operator +(Complex c1, Complex c2)
11.     {
```

【例7-9】 复数加法运算

```
1.      Complex temp = new Complex(0, 0);
2.      temp.real = c1.real + c2.real;
3.      temp.imag = c1.imag + c2.imag;
4.      return temp;
5.  }
```

```
6.  public static Complex operator +(Complex c1, double d)
7.  {   Complex temp = new Complex(0, 0);
8.      temp.real = c1.real + d;
9.      temp.imag = c1.imag;
10.     return temp;
11. }
```

【例7-9】复数加法运算

```
1.  class My {  
2.      static int Main() {  
3.          Complex c1 = new Complex(3, 4);  
4.          Complex c2 = new Complex(5, 6);  
5.          Complex c3 = new Complex(0, 0);  
6.          Console.WriteLine("c1 = " + c1.Real() + "+j" + c1.Imag());  
7.          Console.WriteLine("c2 = " + c2.Real() + "+j" + c2.Imag());
```

【例7-9】复数加法运算

```
1.      c3 = c1 + c2;  
2.      Console.WriteLine("c3 = " + c3.Real() + "+j" + c3.Imag());  
3.      c3 = c3 + 6.5;  
4.      Console.WriteLine("c3 + 6.5 = " + c3.Real() + "+j" + c3.Imag());  
5.      return 0;  
6.  }
```

7. }

C# 命名空间 (Namespace)

- 命名空间的设计目的是为了提供一种让一组名称与其他名称分隔开的方式。
- 在一个命名空间中声明的类的名称与另一个命名空间中声明的相同的类的名称不冲突。
- **using** 关键字
- **using** 关键字表明程序使用的是给定命名空间中的名称。

嵌套命名空间

- 命名空间可以被嵌套，即您可以在一个命名空间内定义另一个命名空间，如下所示：
- `namespace namespace_name1`
- `{`
- `// 代码声明`
- `namespace namespace_name2`
- `{`
- `// 代码声明`
- `}`
- `}`

嵌套命名空间

- 可以使用点 (.) 运算符访问嵌套的命名空间的成员
- `using System;`
- `using first_space;`
- `using first_space.second_space;`

结 束 语

- 学好程序设计语言的唯一途径是

上机练习

- 你的编程能力与你在计算机上投入的时间成

正比