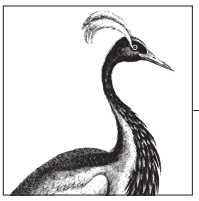

第3章

在C#中创建类

在这一章中，我们将深入讨论类和类成员。

3.1 类

类是最常见的一种引用类型，最简单的类的定义如下：

```
Class YourClassName
{
}
```

复杂的类可能包含以下内容：

类属性	*类属性及类修饰符*。非嵌套的类修饰符有：public、internal、abstract、sealed、static、unsafe、partial
类名	*各种类型参数，唯一基类，多个接口*
花括号内	*类成员（方法、成员属性、索引器、事件、字段、构造方法、运算符函数、嵌套类型和终止器）*

本章涵盖除了类属性、运算符方法和unsafe关键字外的所有上述内容，unsafe关键字将在第4章介绍。以下将逐一介绍各个类成员。

3.1.1 字段

字段是类或结构体中的变量。例如：

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

以下修饰符可以用来修饰字段：

静态修饰符	static
访问权限修饰符	public internal private protected
继承修饰符	new
不安全代码修饰符	unsafe
只读修饰符	readonly
跨线程访问修饰符	volatile

1. 只读修饰符

只读修饰符防止字段值在构造后被更改。只读字段只能在声明时或在其所属的类构造方法中被赋值。

2. 初始化字段

字段不一定要初始化。没有被初始化的字段系统会赋一个默认值（0、\0、null、false）。字段初始化语句在构造方法之前执行：

```
Public int Age =10;
```

3. 同时声明多个字段

为了简便，可以用逗号分隔的列表声明一组同类型的字段，这是声明具有共同属性和修饰符的一组字段的简洁写法。例如：

```
static readonly int legs = 8,
                    eyes = 1;
```

3.1.2 方法

方法是用一组语句实现某个行为。方法能从调用语句的特定类型的传入参数中接收输入数据，并把输出数据以特定的返回值类型返回给调用语句。方法也可以返回void类型，表明这个方法不向调用方返回任何值。此外，方法还可以通过ref/out参数向调用方返回值。

方法签名在整个类中必须是唯一的。方法签名包括方法名、参数类型（但不包括参数名及返回值类型）。

方法可以用以下的修饰符：

静态修饰符	static
访问权修饰符	public internal private protected
继承修饰符	new virtual abstract override sealed
部分方法修饰符	partial
非托管代码修饰符	unsafe extern

1. 重载方法

只要确保方法签名不同，可以在类中重载方法（多个方法共用同一个方法名）。例如，下面的一组方法允许同时出现在同一个类中：

```
void Foo (int x){...}
void Foo (double x){...}
void Foo (int x, float y){...}
void Foo (float x, int y){...}
```

但是，下面的两对方法则不能同时出现在一个类中，因为它们的返回值类型和参数修饰符不属于方法签名的一部分：

```
void Foo (int x){...}
float Foo (int x){...}              // 编译时错误

void Goo (int[] x){...}
void Goo (params int[] x){...}      // 编译时错误
```


2. 值传递和引用传递

参数是按值传递还是按引用传递，也是方法签名的一部分。例如，Foo(int)和Foo(ref int)或Foo(out int)可以同时出现在一个类中。但Foo(ref int)和Foo(out int)不能同时出现在一个类中：

```
void Foo(int x){...}
void Foo(ref int x){...} // 到此处编译正确
void Foo(out int x){...} // 编译时错误
```

3.1.3 类实例构造方法

构造方法执行类或结构体的初始化代码。构造方法的定义和方法的定义类似，区别仅在于构造方法名和返回值只能和封装它的类相同：

```
public class Panda
{
  String name;                  // 定义字段
  public Panda (string n);      // 定义构造方法
  {
    name = n;                   // 初始化代码（给字段赋值）
  }
}
...

Panda p = new Panda ("Petey"); // 调用构造方法
```

构造方法支持以下修饰符：

访问权限修饰符	public internal private protected
非托管代码修饰符	unsafe extern

1. 重载构造方法

类或结构体可以重载构造方法。为了避免重复编码，一个构造方法可以用this关键字调用另一个构造方法：

```
using System;

public class Wine
```

```
{
  public decimail Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this(price) {Year = year; }
}
```

当一个构造方法调用另一个时，被调用的构造方法先执行。

可以向另一个构造方法传递表达式：

```
public Wine (decimal price, DateTime year) : this(price,year.Year) { }
```

表达式内不能再使用this引用，例如，不能调用实例方法。（这是强制性的，因为这个对象当前没有通过构造函数进行实例化，所以调用任何方法都可能失败。）但表达式可以调用静态方法。

2. 隐式无参数构造方法

C#编译器自动为没有显式定义构造方法的类生成构造方法。但是，一旦显式定义了构造方法，系统将不再生成无参数构造方法。

对于结构体来说，无参数构造方法是结构体所固有的，因此，不能自己定义。结构体的隐式构造方法的作用是用默认值初始化每个字段。

3. 构造方法和字段的初始化顺序

首先，在声明字段的时候赋予初始值：

```
class Player
{
  int shields = 50;// 第一个被初始化
  int health = 100;// 第二个被初始化
}
```

字段初始化按声明的先后顺序，在构造方法之前执行。

4. 非公有构造方法

构造方法不一定都是公有的。通常，定义非公有的构造方法的原因是为了在一个静态方法中控制类实例的创建。静态方法可以用于从池中返回类对象，而不必创建一个新对象实例，或用来根据不同的输入属性返回不同的子类。这种使用方式的模板如下：

```
public class Class1
{
  Class1() {}//私有构造方法
  public static Class1 Create(...)
  {
    //在这里定义自己的逻辑，返回Class1的实例
    ...
  }
}
```

3.1.4 对象初始化器

为了简化类对象的初始化，可以在调用构造方法的语句中直接初始化对象的可访问字段或属性。例如下面的类：

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny() {}
  public Bunny(string n) {Name = n; }
}
```

用初始化器初始化Bunny对象的方法如下：

```
//注意无参数构造方法可以省略空括号
Bunny b1 = new Bunny {Name ="Bo",LikesCarrots = true,LikesHumans = false };
Bunny b2 = new Bunny("Bo")      { LikesCarrots = true, LikesHumans = false };
```

确切地说，构造b1和b2的代码等同于：

```
Bunny temp1 = new  Bunny();// temp1是编译器生成的名字
temp1.Name =  Bo ;
temp1.LikesCarrots = true;
temp1.LikesHumans = false;
Bunny b1 = temp1;

Bunny temp2 = new Bunny("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

使用临时变量是为了确保在初始化过程中如果抛出异常，不会得到一个初始化未完成的对象。

对象初始化器是C#3.0引入的新概念。

对象初始化器与可选参数

如果不用对象初始化器，也可以让Bunny类的构造方法接受可选参数：

```
public Bunny ( string name,bool likesCarrots = false,bool likesHumans = false)
{
  Name = name;
  LikesCarrots = likesCarros;
  LikesHumans = LikesHumans;
}
```

这个构造方法可以用如下的语句构造Bunny对象：

```
Bunny b1 = new Bunny(name: "Bo",likesCarrots:true);
```

这样做的优点是我们可以设置Bunny的字段或属性（后面会讲解）为只读。如果在对象的生命周期内，不需要改变字段值或属性值，将字段或属性设为只读是非常有用的。

缺点是所有的可选参数都在调用方处理，换句话说，C#将我们的构造方法调用翻译成：

```
Bunny b1= new Bunny ("Bo", true, false);
```

这使得如果在另一个程序集中实例化Bunny类，而当Bunny类再加一个可选参数（如likesCats）时可能出错。除非引用该类的程序集也重新编译，否则它还将继续调用三个参数的构造方法（现在已经不存在了），并出现运行时错误。（还有一种难以发现的错误是，如果我们修改了某个可选参数的默认值，另一个程序集中的调用方在重新编译前，还会继续使用旧的可选值。）

因此，如果想使程序在不同版本的程序集中保持二进制兼容，最好避免在公有方法中使用可选参数。

3.1.5 this引用

this引用指的是引用类实例自身。在下面的示例中，方法Marry将Partner的mate字段设定为this。

```
public class Panda
{
  public Panda Mate;

  public void Marry(Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

this引用也用来避免类字段和局部变量或属性相混淆。例如：

```
public class Test
{
  string name;
  public Test(string name) { this.name = name; }
}
```

this引用仅对类或结构体的非静态成员有效。

3.1.6 属性

从外表看属性和字段很类似，但属性内部像方法一样包含逻辑。例如，从下面的代码不能判断CurrentPrice到底是字段还是属性：

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

属性和字段的声明很类似，但属性比字段多了一个get/set块。下面是CurrentPrice作为属性的实现方法：

```
public class Stock
{
```

```
  decimal currentPrice;// “背后”的私有字段

  public decimal CurrentPrice// 公有属性
  {
    get { return currentPrice; } set {currentPrice = value; }
  }
}
```

get和set提供属性的访问器。读取属性值时会运行get访问器，它必须返回属性类型的值。给属性赋值时，运行set访问器，它有一个命名为value的隐含参数，类型和属性类型相同，值直接被指定给私有字段（本例中是currentPrice）。

尽管访问属性和字段的方法相同，但不同之处在于，属性在获取和设置值时，给实现者提供了完全的控制能力。这种控制能力使得实现者可以选择所需的任何的内部通信机制，而无需将属性的内部细节暴露给用户。在本例中，set方法可以在value超出有效范围值时抛出异常。

提示：本书中广泛使用私有字段，以免干扰读者注意力。在实际应用中，为了提高封装性，可能更多地在公有字段上应用公有属性。

属性可以用下面的修饰符：

静态修饰符	static
访问权限修饰符	public internal private protected
继承修饰符	new virtual abstract override sealed
非托管代码修饰符	unsafe extern

1. 只读和计算属性

如果只定义了get访问器，属性就是只读的；如果只定义了set访问器，属性就是只写的，但很少用到只写属性。

通常属性会用一个简短的后台字段来存储其所代表的数据，但属性也可以从其他数据计算出来。例如：

```
Decimal currentPrice, shareOwned;

public decimal Worth
{
  get{ return currentPrice * sharesOwned; }
}
```

2. 自动属性

属性最常见的实现方法是get访问器和set访问器，对一个同类型的私有字段进行简单的读写操作。自动属性的声明表明由编译器提供上述实现方法。可以把本节的第一个示例重新定义为：

```
public class Stock
{
  ...
  public decimal CurrentPrice { get; set;}
}
```

编译器会自动产生一个后台的私有字段，该字段名由编译器生成，且不能被引用。如果希望属性对外暴露成只读属性，set访问器可以标注为private的。在C# 3.0中引入了自动属性。

3. get和set的访问权限

get和set访问器可以有不同的访问级别。典型的用法是，将一个public的属性中的set访问器设置成internal或private的：

```
public class Foo
{
  private decimal x;
  public decimal x
  {
    get { return x;}
    private set { x = Math.Round (value, 2); }
  }
}
```

注意，属性本身被声明具有较高的访问权限（本例中是public），然后在需要较低级别的访问器上添加较低级别的访问权限修饰符。

4. CLR属性的实现

C#属性访问器在系统内部被编译成名为get_*XXX*和set_*XXX*的方法:

```
public int get_CurrentPrice { ... }
public void set_CurrentPrice (decimal value) { ... }
```

简单的非虚拟属性访问器被JIT（即时）编译器编译成内联的，消除了属性和字段访问方法的性能差别。内联是一种优化方法，它用方法的函数体替代方法调用。

通过WinRT的属性，编译器就可以假定是put_*XXX*命名转换，而不是set_*XXX*。

3.1.7 索引器

索引器为访问类或结构体中封装的列表或字典型数据元素提供了自然的访问接口。索引器和属性很相似，但索引器通过索引值而非属性名访问数据元素。string类具有索引器，可以通过int索引访问其中的每一个char值：

```
string s = "hello";
Console.WriteLine(s[0]);// 'h'
Console.WriteLine(s[3]);// '1'
```

当索引是整型时，使用索引器的方法类似于使用数组。

提示：索引器和属性具有相同的修饰符（详见“属性”一节中的介绍）。

---

1. 实现索引器

要编写一个索引器，首先定义一个名为this的属性，将参数定义放在一对方括号中。例如：

```
class Sentence
{
  sting[] words ="The quick brown fox".Split();

  public string this [int wordNum]   // 索引器
  {
    get { return words [wordNum]; }
    set { words [wordNum] = value; }
  }
}
```

下面是使用索引器的方法：

```
Sentence s= new Sentence();
Console.WriteLine (s[3]);//fox
S[3] = "kangaroo";
Console.WriteLine (s[3]);//Kangaroo
```

一个类可以定义多个参数类型不同的索引器，索引器也可以有多个参数：

```
public string this [int arg1, string arg2]
{
  get { ... } set { ... }
}
```

如果省略set访问器，索引器就变成只读的。

2. CLR索引器的实现

索引器在系统内部被编译成名为get_Item和set_Item的方法，如下所示：

```
public string get_Item (int wordNum) { ... }
public void set_Item (int wordNum, string value) { ... }
```

3.1.8 常量

常量是值永远不会改变的字段。常量在编译时静态赋值，并且在使用时，编译器直接替换该值，类似于C++中的宏。常量可以是内置的数据类型：bool、char、string或枚举类型。

常量用关键字const定义，并且必须以特定值初始化。例如：

```
public class Test
{
  public const string Message = "Hello World";
}
```

常量在使用时比静态只读字段有更多限制——不仅能使用的类型有限，而且初始化字段的语句含义也不同。常量和静态只读变量的不同之处还有，常量是在编译时赋值的。例如：

```
public static double Circumference (double radius)
{
  return 2 * System.Math.PI * radius;
}
```

编译成：

```
public static double Circumference (double radius)
{
  return 6.2831853071795862 * radius;
}
```

这样做是合理的，因为PI是常量，它的值永远不变。相反，静态只读字段可以在每个应用中有不同的值。

提示：静态只读字段的好处还有，当提供给其他程序集时，可以更新数值。例如：假定程序集X提供一个如下常量：

```
public const int MaximumThreads = 20;
```

如果程序集Y引用X并使用该常量，值20在编译时就固定在Y程序集中了。这样，若X后来将常量设成50并重新编译，Y如果不重新编译，还会使用旧值20。静态只读字段可以避免这一问题。
从另一角度看，将来可能发生变化的任意值都不受其定义约束，所以不应该表示为一个常量。

常量也可以在方法内声明，例如：

```
static void Main()
{
  Const double twoPI = 2 * System.Math.PI;
  ...
}
```

常量可以使用以下修饰符：

访问权限修饰符	public internal private protected
继承修饰符	new

3.1.9 静态构造方法

静态构造方法是每个类执行一次，而不是每个类实例执行一次。一个类只能定义一个静态构造方法，并且必须没有参数，必须和类同名：

```
class Test
{
  static Test() { Console.WriteLine("Type Initialized"); }
}
```

运行时在使用类之前自动调用静态构造方法，下面两种行为可以触发静态构造方法：

- 实例化类
- 访问类的静态成员

静态构造方法只有两个修饰符：unsafe和extern。

警告：如果静态构造方法抛出一个未处理异常（第4章），类在整个应用程序的生命周期内都是不可用的。

静态构造方法和字段初始化顺序

静态字段在调用静态构造方法之前执行初始化。如果一个类没有静态构造方法，字段在类被使用前初始化或在运行时随机选一个更早的时间执行初始化（这说明静态构造方法的存在可能使字段初始化比正常时间晚执行）。

静态字段按字段声明的先后顺序初始化。下例中X初始化为0，Y初始化为3。

```
class Foo
{
  public static int X = Y;//0
  public static int Y = 3; //3
}
```

如果我们调换两个字段的初始化顺序，两个字段都将被初始化为3。下例中先打印0后打印3，因为字段初始化器在X被初始化为3前实例化Foo：

```
class program
{
  staic void Main() { Console.WriteLine (Foo.X); } // 3
}

class Foo
{
  public static Foo Instance = new Foo();
  public static int X=3;

  Foo() { Console.WriteLine(X); } // 0
}
```

如果交换两行粗体语句的顺序，上例输出两个3。

3.1.10 静态类

类可以标记为`static`，表明它必须仅由静态成员组成，并且不能产生子类。`System.Console`和`System.Math`类就是静态类的最好示例。

3.1.11 终止器

终止器是只能在类中使用的方法，它在垃圾收集器回收没有被应用的对象前执行。终止器的语法是类名加前缀~：

```
class Class1
{
  ~Class1()
  {
    ...
  }
}
```

实际上，这是重载对象的Finalize方法的C#语法，编译器将其扩展成如下的方法声明：

```
protected override void Finalize()
{
   ...
```

```
    base.Finalize();
}
```

我们在第12章详细讨论垃圾回收。

终止器允许使用以下修饰符：

非托管代码修饰符	unsafe

3.1.12 局部类和方法

局部类允许一个类分开定义，典型的用法是分开在多个文件中。从其他源文件自动生成的类（如：XSD文件）需要和自定义的方法交互时，通常使用partail类。例如：

```
// PaymentFormGen.cs -自动生成的
partial class PaymentForm { ... }

// PaymentForm.cs -自定义的
partial class PaymentForm { }
```

每个部分必须由partial声明，下面的写法是不合法的：

```
partial class PaymentForm { }
class PaymentForm { }
```

局部类的各组成部分不能有冲突的成员。例如具有相同参数的构造方法，不能重复出现。局部类完全由编译器处理，也就是说，各组成部分在编译时必须可用，并必须编译在同一个程序集中。

有两个方法为partial类定义基类：

- 在每个部分定义同一个基类。例如：

```
partial class PaymentForm : ModalForm { }
partial class PaymentForm : ModalForm { }
```

- 仅在其中一部分定义基类。例如：

```
partial class PaymentForm : ModalForm {}
partial class PaymentForm {}
```

另外，每个部分都可以独立定义并实现接口。我们将在后面“继承”和“接口”中详细介绍基类和接口。

局部方法

局部类可以包含局部方法，这些方法使自动生成的局部类可以为自定义方法提供自定义钩子（hook）。例如：

```
partial class PaymentForm// 自动生成的类文件中
{
  ...
  partial void ValidatePayment (decimal amount);
}

partial class PaymentForm// 自定义的文件中
{
```

```
    ...
    partial void ValidatePayment (decimal amount)
    {
      if(amount > 100)
        ...
    }
}
```

局部方法由两部分组成：定义和实现。定义一般由代码生成器产生，而实现多为手工编写。如果没有提供方法的实现，方法的定义会被编译器清除。这使得自动代码生成可以自由提供钩子（hook），而不用担心代码过于臃肿。局部方法必须是void型，并且默认是private的。

局部方法在C# 3.0中引入。

3.2 继承

为了扩展或自定义原类，类可以继承另一个类。继承类让你可以重用另一个类的方法，而无需重新构建。一个类只能继承自唯一的类，但可以被多个类继承，从而形成类的层次。在本例中，我们定义一个名为Asset的类：

```
public class Asset
{
  public string Name;
}
```

然后定义一个Stock类和一个House类，他们都继承自Asset类。他们具有Asset类的所有特征，而各自又有增加的定义：

```
public class Stock : Asset// 从Asset继承
{
  public long SharesOwned;
}

Public class House : Asset// 从Asset继承
{
  public decimal Mortgage;
}
```

下面是两个类的使用方法：

```
Stock msft = new Stock { Name = "MSFT", SharesOwned = 1000 };
Console.WriteLine (msft.Name);                // MSFT
Console.WriteLine (msft.SharesOwned);         // 1000

House mansion = new House { Name = "Mansion",Mortgage = 250000 };
console.WriteLine (mansion.Name);             // Mansion
Console.WriteLine (mansion.Mortage);          // 250000
```

子类Stock和House都从基类Asset继承了Name属性。

提示： 子类也被称为派生类；基类也被称为超类。

3.2.1 多态

引用是多态的。意味着*x*类型的变量可以指向*x*子类的对象。例如：

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

这个方法可以用来显示Stock和House类，因为这两个类都继承自Asset类。

```
Stock msft = new Stock ...;
House mansion = new House ...;

Display (msft);
Display (mansion);
```

多态性之所以能实现，是因为子类（Stock和House）具有基类（Asset）的全部特征。反过来，则不正确。如果Display改成接收House类，就不能把Asset类传给它：

```
static void Main() { Display (new Asset()); }// 编译时错误

public satic void Display (House house)      // 不能接收Asset类
{
  System.Console.WriteLine (house.Mortgage);
}
```

3.2.2 类型转换和引用转换

对象引用可以被：

- 隐式向上转换成基类的引用
- 显式向下转换为子类的引用

在可兼容的类型引用之间向上类型转换或向下类型转换即为引用转换：生成一个新的引用指向同一个对象。向上转换总是能成功，而向下转换只有在对象的类型符合要求时才能成功。

1. 向上类型转换

向上类型转换创建一个基类指向子类的引用。例如：

```
Stock msft = new Stock();
Asset a = msft;      // 向上转换
```

向上转换以后，变量a还是指向msft指向的Stock对象。被引用的对象本身不会被替换或改变：

```
Console.WriteLine (a == msft);       // 正确
```

虽然a和msft指向同一个对象，但a指向对象时有更严格的要求：

```
Console.WriteLine (a == msft);       // 正确
Console.WriteLine (a.ShareOwned);    // 错误：SharesOwned未定义
```

上面最后一行产生一个编译错误，因为变量a是Asset类型，虽然它指向的是Stock类型的对象。如果要访问ShareOwned字段，你必须先把Asset类型向下转换成Stock类型。

2. 向下类型转换

向下类型转换创建一个子类指向基类的引用。例如：

```
Stock msft = new Stock();
Asset a = msft;                        // 向上转换
Stock s = (Stock)a;                    // 向下转换
Console.WriteLine (s.ShareOwned);      // 没有错误
Console.WriteLine (s == a);            // 输出True
Console.WriteLine (s == msft)          // 输出True
```

对于向上转换，只影响了引用，被引用的对象没有变化。而向下转换必须是显式转换，因为它可能导致运行时错误：

```
House h = new House();
Asset a = h;              // 向上转换永远会成功
Stock s = (Stock)a;       // 向下转换出错：a不是Stock类型
```

如果向下转换出错，会抛出InvalidCastException。这是一个运行时类型检查的例子（我们后面还会在“静态和运行时类型检查”中详细介绍。

3. as运算符

as运算符在向下类型转换出错时为变量赋值null（而不是抛出异常）：

```
Asset a = new Asset();
Stock s = a as Stock//s是null，不抛出异常
```

这个操作相当有用，接下来只需判断结果是否为null。

```
if(s!= null) Console.WriteLine(s.SharesOwned);
```

提示： 如果不用判断结果是否为null，使用cast更好，因为如果发生错误，cast会抛出描述更清楚的异常。我们可以通过比较下面两行代码看出：

```
int shares = ((Stock)a).ShareOwned;    //方法#1
int shares = (a as Stock).ShareOwned;  //方法#2
```

如果a不是Stock类型，第一行代码抛出InvlidCastException，很清楚地描述了错误。第二行代码抛出NullReferenceException，这就比较模糊，到底是因为a不是Stock类型还是因为a为null呢？

从另一个角度看，使用cast操作符的意思是告诉编译器：“已确定这个值的类型；如果判断错误，那么代码可能有bug，所以要抛出一个异常！”而如果使用as操作符，则表示不确定其类型，需要根据运行时输出结果来确定其类型。

---

as运算符不能用来实现自定义转换（见第4章“运算符重载”），也不能用于数值型转换：

```
long x = 3 as long;    //编译时错误
```

提示： as和cast运算符也可以用来实现向上类型转换，但不常用，因为隐式转换就可以实现。

---

4. is运算符

is运算符用于检查引用的转换能否成功，换句话说，它是检查一个对象是否是从某个特定类派生

（或是实现某个接口），经常在向下类型转换前使用。

```
if(a is Stock)
  Console.WriteLine (((Stock)a).SharesOwned);
```

is运算符不能用于自定义类型转换和数值型类型转换，但它可以用于拆箱机制的类型转换（详见“object类型”）。

3.2.3 虚函数成员

标识为virtual的函数可以被提供特定实现的子类重载。方法、属性、索引器和事件都可以被声明为virtual：

```
public class Asset
{
   public string Name;
   public virtual decimal Liability { get { return 0; } }
}
```

子类通过override修饰符重载虚方法：

```
public class Stock : Asset
{
  public long SharesOwned;
}

public class House : Asset
{
  public decimal Mortgage;
  public override decimal Liability { get { return Mortgage; } }
}
```

Asset类的Liability默认值是0。Stock类不用限定这一行为，而House类限定让Liability属性返回Mortgage的值：

```
House mansion = new House { Name =  McMansion , Mortgage = 250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability); // 250000
Console.WriteLine (a.Liability); // 250000
```

虚方法和重载方法的标识、返回值以及访问权限必须完全一致。重载方法可以通过base关键字调用其基类的实现（我们在“base关键字”中详细介绍）。

警告：从构造方法调用虚方法可能很危险，因为编写子类的人在重写方法时不可能知道正在操作一个未完全实例化的对象。换而言之，重写方法最终会访问到一些依赖于未被构造方法初始化的域的方法或属性。

3.2.4 抽象类和抽象成员

被声明为*abstract*的抽象类不能被实例化。只有抽象类的具体实现子类才能被实例化。

抽象类中可以定义抽象成员，抽象成员和虚成员相似，但抽象成员不提供默认的实现。实现必须由子类提供，除非子类也被声明为抽象类：

```
public abstract class Asset
{
  // 注意实现为空
  public abstract decimal NetValue { get; }
}

public class Stock : Asset
{
  public long ShareOwned;
  public decimal CurrentPrice;

  // 像虚方法一样重载
  public override decimal NetValue
  {
    get { return CurrentPrice * ShareOwned; }
  }
}
```


3.2.5 隐藏继承成员

基类和子类可能定义相同的成员，例如：

```
public class A{ public int Counter = 1; }
public class B : A { public int Counter = 2; }
```

类B中的字段Counter隐藏了类A中的字段Counter。通常，这种情况的产生是由于编程时，在定义子类的成员之后又把相同的成员加到基类中。因此，编译器会产生一个警告，可以用下面的方法避免二义性：

- A类的引用（在编译时）绑定到A.Counter
- B类的引用（在编译时）绑定到B.Counter

有时需要故意隐藏一个成员，这种情况下，可以在子类中使用new修饰符。new修饰符的作用仅为防止编译器发出警告。写法如下：

```
public class A     { public     int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

修饰符new把你的意图传达给编译器以及其他编程人员，即重复的成员不是无意的。

提示： C#在不同的上下文环境中使用new关键字表达完全不同的含义。特别要注意new运算符和new成员修饰符的不同。

---

new和virtual的比较

思考下面的类层级：

```
public class BaseClass
{
  public virtual void Foo() { Console.WriteLine ("BaseClass.Foo"); }
}

public class Overrider : BaseClass
{
```

```
  public override void Foo() { Console.WriteLine ("Overrider.Foo"); }
}

public class Hider : BaseClass
{
  Public new void Foo() { Console.WriteLine ("Hider.Foo"); }
}
```

下面的代码说明了Overrider和Hider类的不同：

```
Overrider over = new Overrider ();
BaseClass b1 = over;
over.Foo();          // Overrider.Foo
b1.Foo();            // Overrider.Foo

Hider h = new Hider();
BaseClass b2 = h;
h.Foo();             // Hider.Foo
b2.Foo();            // BaseClass.Foo
```

3.2.6 密封方法和类

重载的方法成员可用sealed关键字密封它的实现，以防止该方法被它的更深层次的子类再次重载。在前面的虚方法成员示例中，我们可以密封House类的Liability实现，以防止继承自House的子类重载Liability。写法如下：

```
public sealed override decimal Liability { get { return Mortgage; } }
```

也可以在类中使用sealed修饰符来密封整个类，含义是密封类中所有的虚方法。密封类比密封方法成员更常见。

3.2.7 base关键字

关键字base和关键字this很类似。它有两个重要目的：

- 从子类访问重载的基类方法成员
- 调用基类的构造方法（见下节）

本例中，House类用关键字base访问Asset类对Liability的实现：

```
public class House : Asset
{
  ...
  public override decimal Liability
  {
    get { return base.Liability + Mortgage; }
  }
}
```

通过关键字base，我们非显式地访问了Asset类的Liablility非虚属性。这表明，不管实例的运行时类型如何，都将访问Asset类的该属性。

如果Liability是隐藏属性而非重载的属性，该方法也同样有效。（也可以在调用方法前，转换成基类，以访问隐藏的成员。）

3.2.8 构造和继承

子类必须声明自己的构造方法。例如，如果定义如下子类：

```
public class Baseclass
{
  public int X;
  public Baseclass() {}
  public Baseclass(int x) { this.X = x; }
}

public class Subclass : Bassclass { }
```

下面的语句是不合法的：

```
Subclass s = new Subclass (123);
```

子类必须重新定义它想对外公开的任何构造方法。不过，定义子类的构造方法，也可以通过使用关键字base调用基类的某个构造方法实现：

```
public class Subclass : Baseclass
{
  public Subclass (int x) : base (x) { }
}
```

关键字base和this用法类似，但base关键字调用的是基类中的构造方法。

基类的构造方法总是先执行，这保证了*base*的初始化发生在作为子类的特例初始化之前。

1. 隐式调用基类无参数的构造方法

如果子类中的构造方法省略base关键字，那么基类的无参数构造方法将被隐式调用：

```
public class BaseClass
{
  public int X;
  public BaseClass () { X = 1; }
}

public class Subclass : BaseClass
{
  public Subclass() { Console.WriteLine (x); } // 1
}
```

如果基类没有无参数的构造方法，子类的构造函数中就必须使用base关键字。

2. 构造方法和字段初始化的顺序

当对象被实例化时，初始化按以下顺序进行：

（1）从子类到基类：

a. 初始化字段

b. 指定被调用基类的构造方法中的变量

（2）从基类到子类：

a. 构造方法体执行

代码如下：

```
public class B
{
  int x = 0; // 第三个执行
  public B (int x)
  {
    ...// 第四个执行
  }
}
public class D : B
{
  int y = 0; // 第一个执行
  public D(int x)
    : base (x+1)//第二个执行
  {
    ...// 第五个执行
  }
}
```

3.2.9 重载和解析

继承对方法的重载有特殊的影响。看下面的两个重载：

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

当重载被调用时，类型最明确的优先匹配：

```
House h = new House( );
Foo(h);              //调用Foo(House)
```

具体调用哪个重载是静态决定的（编译时）而不是在运行时决定。下面的代码调用Foo(Asset)，尽管a在运行时是House类型的：

```
Asset a = new House ( );
Foo(a);              //调用Foo(Asset)
```

提示： 如果把Asset类转换成动态的（dynamic）（见第4章），会在运行时决定哪个重载被调用，这样就会基于对象的实际类型进行选择：

```
Asset a = new House(...);
Foo ((dynamic)a);   // 调用Foo(House)
```

3.3 object类型

object类（System.Object）是所有类型的最终基类。任何类型都可以向上转换成object类型。

为了说明这个类型的重要性，首先介绍通用栈。栈是一种遵循LIFO（Last-In First-Out，后进先出法）

的数据结构。栈有两种操作：*push*表示一个元素进栈和*pop*表示一个元素出栈。下面是能容纳10个对象的栈的简单实现：

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj)      { data[position++] = obj;}
  public object Pop()                { return data[ -- position]; }
}
```

因为栈操作的对象是object类，所以可以实现任意类型的对象实例的进栈和出栈：

```
Stack stack = new Stack();
stack.Push("sausage");
string s = (string) stack.Pop();     // 向下类型转换，需要显式转换

Console.WriteLine(s);                // sausage
```

承载了类的优点，object是引用类型。尽管如此，int等数值类型也可以和object类型相互转换，并加入栈中。C#的这个特性称为类型一致化，代码演示如下：

```
stack.Push(3);
int three = (int) stack.Pop();
```

当数值类型和object类型之间相互转换时，公共语言运行时（CLR）必须作一些特定的工作，实现数值类型和引用类型的转换这个过程被称为装箱和拆箱。

提示： 下面的“泛化”章节，我们会讲述如何改进Stack类，使之能更好地处理同类型数据。

3.3.1 装箱和拆箱

装箱是将数值类型实例转换成引用类型实例的行为。引用类型可以是object类或接口（本章后面将介绍接口）（注1）。本例中，我们将int类型装箱成一个object对象。

```
int x = 9;
object obj = x;      // 把int类型装箱
```

拆箱操作正好相反，它把object类型转换成原始的数值类型：

```
int y = (int)obj;    // 把int类型拆箱
```

拆箱需要显式进行。运行时检查提供的值类型是否与真正的对象类型相匹配，并在检查出错误时，抛出InvalidCastException。例如，下面的例子抛出异常，因为long类型和int类型不匹配：

```
object obj = 9;              // 9被自动识别为int类型
long x = (long) obj;         // 抛出InvalidCastException
```

下面的语句正确：

```
object obj = 9;
```

注1：引用类型也可以是System.ValueType或System.Enum（见第6章）。

```
long x = (int)obj;
```

以下语句也正确：

```
object obj = 3.5;                    // 3.5被自动识别为double类型
int x = (int) (double) obj;          // x 现在是3
```

最后一行代码中，（double）是拆箱操作，（int）是数值转换操作。

提示：装箱转换对系统提供一致的数据类型至关重要。但这个体系并不是完美的：在“泛化”章节会介绍，数组和泛型的变量只能支持引用转换，不能支持装箱转换：

```
object[] a1 = new string [3]; //合法
object[] a2 = new int[3]      //出错
```

装箱和拆箱的实质是复制

装箱是把数值类型的实例复制到新对象中，而拆箱是把对象的内容复制回数值类型的实例中。下面的示例修改了i的值，但并不会改变它先前被装箱时拷贝的值：

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine(boxed);    // 3
```

3.3.2 静态和运行时类型检查

C#在静态（编译时）和运行时都会进行类型检查。

静态类型检查使编译器能在程序没有运行的情况下检查正确性。下面代码会出错，因为编译器强制进行静态类型检查：

```
int x = "5";
```

在引用或拆箱操作的向下类型转换时，由CLR执行运行时类型检查。例如：

```
object y = "5";
int z = (int) y;             //运行时错误，向下类型转换失败
```

可以进行运行时类型检查，是因为堆栈中的每个对象都在内部存储了类型标识，这个标识可以通过调用object类的GetType方法读取。

3.3.3 GetType方法和typeof运算符

所有C#的类型在运行时都会维护System.Type类的实例。有两个基本方法可以获得System.Type对象：

- 在类实例上调用GetType方法
- 在类名上使用typeof运算符

GetType在运行时赋值；typeof在编译时静态赋值（如果使用泛型类型，那么它将由即时编译器解析）。

System.Type有针对类型名、程序集、基类等的属性。例如：

```
using System;

public class Point { public int X,Y; }

class Test
{
  static void Main()
  {
    Point p = new Point();
    Console.WriteLine(p.GetType().Name);                    // Point
    Console.WriteLine(typeof(Point).Name);                  // Point
    Console.WriteLine(p.GetType() == typeof(Point));        // True
    Console.WriteLine(p.X.GetType().Name);                  // Int32
    Console.WriteLine(p.Y.GetType().FullName);              // System.Int32
  }
}
```

同时System.Type还有作为运行时反射模式的访问器。在第19章介绍该内容。

3.3.4 ToString方法

ToString方法返回类实例的默认文本表述。这个方法被所有内置类型重载。下面是对int类使用ToString方法的示例：

```
int x = 1;
string s = x.ToString();// s 是"1"
```

可以用下面的方式重载自定义类的ToString方法：

```
public class Pansda
{
  public string Name;
  public override string ToString() { return Name; }
}
...

Panda p = new Panda { Name = "Petey"};
Console.WriteLine(p);// Petey
```

如果不重写ToString，那么这个方法会返回类型名称。

提示： 当直接在数值型对象上调用像ToString这样的重载的object成员时，不会发生装箱。只有进行类型转换时，才会执行装箱操作：

```
Int x = 1;
String s1 = x.ToString();      // 调用没有装箱的值
Object box = x;
String s2 = box.ToString();    // 调用装箱后的值
```

3.3.5 列出object成员

列出object的所有成员：

```
Public class Object
```

```
{
  public Object();

  public extern Type GetType();

  public virtual bool Equals (object obj);
  public static bool Equals (object objA, object objB);
  public static bool ReferenceEquals (object objA, object objB);

  public virtual int GetHashCode();

  public virtual string ToString();
  protected override void Finalize();
  protected extern object MemberwiseClone();
}
```

Equals，ReferenceEquals和GetHashCode方法将在第6章的“等值比较”一节讲述。

3.4 结构体

结构体和类相似，不同之处在于：

- 结构体是值类型，而类是引用类型。
- 结构体不支持继承（除了隐式派生自object类的，更精确些说，是派生自System.ValueType）。

除了以下三项内容，结构体可以包含类的所有成员：

- 无参数的构造方法
- 终止器
- 虚成员

当表示值类型时使用结构体更理想而不用类。数值类就是很好的例子，此时，指派一个值的副本比指派一个引用更自然。因为结构体是值类型，每个实例不需要在堆栈上实例化，创建一个类型的多个实例时可以节约空间。例如，创建一个数值类型的数组，只需要分配一个堆栈空间。

3.4.1 结构体构造语义

结构体的构造语义如下：

- 隐含存在一个无法重载的无参数构造方法，将字段按位置零。
- 定义结构体的构造方法时，必须显式指定每个字段。
- 不能在结构体内初始化字段。

下面是一个声明和调用结构体构造方法的示例：

```
public struct Point
{
  int x, y;
  public Point(int x, int y) { this.x = x; this.y = y; }
```

```
}

...
Point p1 = new Point();              // p1.x和p1.y都是0
Point p2 = new Point(1,1);           // p2.x和p2.y都是1
```

以下的示例包含三个编译时错误：

```
public struct Point
{
  int x = 1;                                  // 不合法：不能初始化字段
  int y;
  public Point() {}                           // 不合法：不能有无参数的构造方法

  public Point(int x) { this.x = x; }// 不合法：必须指定y值
}
```

如果把struct替换成class，上面的写法都合法。

3.5 访问权限修饰符

为了提高封装性，类或类成员会在声明中添加五个访问权限修饰符之一，来限制其他类和其他程序集对它的访问权限：

public
: 完全访问权限；枚举类型成员或接口隐含的访问权限。

internal
: 仅可访问程序集和友元程序集；非嵌套类型的默认访问权限。

private
: 仅在包含类型可见；类和结构体成员的默认访问权限。

protected
: 仅在包含类型和子类中可见。

protected internal
: protected和internal的访问权限并集Eric Lippert是这样解释的：默认情况下尽可能将所有成员定义为私有，然后每一个修饰符都会提高其访问级别。所以用protected internal修饰的成员在两个方面的访问级别都提高了。

提示： CLR有对protected和internal访问权限交集的定义，但C#并不支持。

3.5.1 举例

Class2在本程序集外可访问，而Class1不可以：

```
class Class1 {}                 // Class1是internal访问权限的（默认）
public class Class2 {}
```

ClassB有对本程序集的其他类提供的字段x；ClassA没有：

```
class ClassA { int x;        } // x是private访问权限的（默认）
class ClassB { internal int x; }
```

Subclass里的函数可以调用Bar但不能调用Foo：

```
class BaseClass
{
  void Foo(){}                  // Foo是private访问权限的（默认）
  protected void Bar() {}
}

class Subclass: BaseClass
{
  void Test1() { Foo(); }       // 出错：不能访问Foo
  void Test2() { Bar(); }       // 正确
}
```

3.5.2 友元程序集

在高级语义应用中，加上System.Runtime.CompilerServices.InternalsVisibleTo属性，就可以把internal成员提供给其他的友元程序集，用如下方法指定友元程序集：

```
[assembly: InernalsVisibleTo("Friend")]
```

如果友元程序集有强命名（见第18章），必须指定其完整的160字节公共键值：

```
[assembly: InternalsVisibleTo( "StrongFriend, PublicKey = 0024f000048c...")]
```

可以用LINQ查询从强命名的程序集中提取完整的公共键值（第8章详细介绍LINQ）：

```
String key = string.Join("",
Assembly.GetExecutingAssembly().GetName().GetPublicKey()
.Select(b=>b.ToString("x2"))
.ToArray());
```

提示： 在LINQPad中和本例相辅相成的一个例子，是浏览程序集后复制程序集的完整公共键值到剪贴板。

3.5.3 程序集的权限封顶

类权限是它内部声明的成员访问权限的封顶。关于权限封顶最常用的示例是internal类中的public成员。例如：

```
class C { public void Foo() {}}
```

类C的（默认）访问权限是internal，它作为Foo的最高访问权限，把Foo的权限改为internal。Foo指定为public的原因一般是，如果类C的访问权限要改成public的，重构会更容易。

3.5.4 访问权限修饰符的限制

当重载基类的函数时，重载函数的访问权限必须一致。例如：

```
class BaseClass { protected virtual void Foo() {} }
```

```
class Subclass1 : BaseClass { protected override void Foo() {} } // 正确
class Subclass2 : BaseClass { public override void Foo() {} }    // 错误
```

（一个例外情况是，如果在另一个程序集中重写一个protected internal方法，那么重写方法必须修饰为protected。）

编译器会阻止使用任何不一致的访问权限修饰符。例如，子类可以比基类访问权限低，但不能比基类访问权限高：

```
internal class A {}
public class B: A {}         //错误
```

3.6 接口

接口和类相似，但接口只为成员提供定义而不提供实现。接口和类的不同之处有：

- 接口的成员都是隐含抽象的。相反，类可以包含抽象成员和有具体实现的成员。
- 一个类（或结构体）可以实现多个接口。相反，类只能继承一个类，而结构体完全不支持继承（只能从System.ValueType派生）。

接口声明和类声明很类似，但接口不提供其成员的实现，因为它的所有成员都是隐式定义为抽象的，这些成员将由实现接口的类或结构体实现。接口只能包含方法、属性、事件、索引器，这些正是类中可以定义为抽象的成员。

下面是System.Collections命名空间中IEnumerator接口的定义：

```
public interface IEnumerator
{
  pool MoveNext();
  object Current { get; }
  void Reset();
}
```

接口成员总是隐式地定义成public的，并且不能用访问权限修饰符声明。实现接口意味着为其所有成员提供public的实现：

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext() { return count -->0;                }
  public object Current  { get { return count; }             }
  public void Reset()    { throw new NotSupportedException(); }
}
```

可以把对象隐式转换为它实现的任意一个接口。例如：

```
IEnumerator e = new Countdown();
While(e.MoveNext())
  Console.Write(e.Current);      // 10987654210
```

提示：尽管Countdown是internal权限的类，通过把Countdown实例转换成IEnumerator，其内部实现IEnumerator接口的成员可以作为public成员访问。例如，如果同程序集中的一个公有类定义了如下方法：

```
Public static class Util
{
  Public static object GetCountDown()
  {
    Return new Countdown();
  }
}
```

另一个程序集的调用者可以执行：

```
IEnumerator e = (IEnumerator) Util.GetCountDown();
e.MoveNext();
```

如果IEnumerator被定义成internal，上面的方法则不正确。

3.6.1 扩展接口

接口可以从其他接口派生。例如：

```
public interface IUndoable              { void Undo(); }
public interface IRedoable : IUndoable  { void Redo(); }
```

IRedoable继承所有IUndoable的成员。换而言之，实现IRedoable的类型还必须实现IUndoable的成员。

3.6.2 显式接口实现

当实现多个接口时，有时成员标识符会有冲突。显式实现接口成员可以解决冲突。看下面的例子：

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget: I1,I2
{
   public void Foo()
  {
    Console.WriteLine("Widget's implementation of I1.Foo");
  }
  int I2.Foo()
  {
    Console.WriteLine("Widget's implementation of I2.Foo");
    Return 42;
  }
}
```

因为I1和I2都有Foo标识符，Widget显式实现了I2的Foo方法。这是两个同名方法在同一个类中同时存在。调用显式实现成员的唯一方法是先转换为相应的接口：

```
Widget w = new Widget();
w.Foo();                    // Widget对I1.Foo的实现
((I1)w).Foo();              // Widget对I1.Foo的实现
((I2)w).Foo();              // Widget对I2.Foo的实现
```

另一个使用显式实现接口成员的原因是，隐藏那些和类的正常用法差异很大或有严重干扰性的成员。例如：实现ISerializable接口的类型，通常需要避免强调它的ISerializable成员，除非显式转换成这个接口。

3.6.3 虚方法实现接口成员

默认情况下，接口成员的实现是隐式定义为sealed。为了能重载，必须在基类中标识为virtual或者abstract。例如：

```
public interface IUndoable { void Undo(); }

public class TextBox: IUndoable
{
  Public virtual void Undo()
  {
    Console.WriteLine("TextBox.Undo");
  }
}

public class RichTextBox : TextBox
{
  public override void Undo()
  {
    Console.WriteLine("RichTextBox.Undo");
  }
}
```

不管从基类还是接口中调用接口成员，调用的都是子类的实现：

```
RichTextBox r = new RichTextBox();
r.Undo();                          //RichTextBox.Undo
((IUndoable)r).Undo();             // RichTextBox.Undo
((TextBox)r).Undo();               // RichTextBox.Undo
```

显式实现的接口成员不能标识为virtual的，也不能实现通常意义的重载。但是它可以被重新实现。

3.6.4 在子类中重新实现接口

子类可以重新实现基类中已经被实现的任意一个接口。不管基类中该成员是不是virtual的，当通过接口调用时，重新实现都能够屏蔽成员的实现。它不管接口成员是隐式还是显式实现都有效，但后者效果更好。

下面的例子中，TextBox显式实现IUndoable.Undo，所以不能标识为virtual。为了实现重载，RichTextBox必须重新实现IUndoable的Undo方法：

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  void IUndoable.Undo() { Console.WriteLine("TextBox.Undo"); }
}

public class RichTextBox : TextBox, IUndoable
{
```

```
  public new void Undo() { Console.WriteLine ("RichTextBox.Undo"); }
}
```

从接口调用成员的重新实现时，调用的是子类的实现：

```
RichTextBox r = new RichTextBox();
r.Undo();               // RichTextBox.Undo   例1
((IUndoable)r).Undo();  // RichTextBox.Undo   例2
```

假定RichTextBox定义不变，如果TextBox隐式实现Undo：

```
public class TextBox : IUndoable
{
  public void Undo() { Console.WriteLine( TextBox.Undo ); }
}
```

这样，为我们提供了另一种调用Undo的方法，如例3所示，它将“中断”系统。

```
RichTextBox r = new RichTextBox();
r.Undo();                   // RichTextBox.Undo      例1
((IUndoable)r).Undo();      // RichTextBox.Undo      例2
((TextBox)r).Undo();        // TextBox.Undo          例3
```

例3显示出，重新实现屏蔽仅当通过接口调用成员时有效，从基类调用时无效。这个特性通常不尽人意，因为它有二义性。重新实现主要是为了重载显式实现的接口成员。

接口重新实现的替代方法

即使在显式成员实现中，接口重新实现还是容易出问题，原因如下：

- 子类无法调用基类的方法
- 定义基类时不能预测方法是否会被重新实现，可能不允许这个潜在的功能

重新实现是未知子类时最不理想的方法，更好的选择是，在定义基类时不允许使用重新实现。有两种方法可以做到：

- 当隐式实现成员时，如果合适，标为virtual
- 当显式实现成员时，如果能预测子类可能要重载某些逻辑，用下面的模式：

```
public class TextBox: IUndoable
{
  void IUndoable.Undo(){ Undo(); }            // 调用下面的方法
  protected virtual void Undo() { Console.WriteLine("TextBox.Undo"); }
}

public class RichTextBox : TextBox
{
  protected override void Undo() { Console.WriteLine("RichTextBox.Undo"); }
}
```

如果不添加子类，可以把类标注成sealed，以占用接口的重新实现。

3.6.5 接口和装箱

将结构体转换成接口会引发装箱机制。调用结构体的隐式实现接口成员不会引发装箱。

```
Interface I { void Foo(); }
struct S : I { public void Foo() {} }

...
S s = new S();
s.Foo();// 没有装箱

I i = s;// 当转换为接口时引发装箱
i.Foo();
```

写类和写接口的对比

指导原则：

- 当能自然地共享实现时，使用类和子类
- 当实现是独立的，为类定义接口

观察下面的类：

```
abstract class Animal {}
abstract class Bird : Animal {}
abstract class Insect : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore : Animal {}

// 具体实现类：

class Osrich : Bird {}
class Eagle : Bird, FlyingCreature, Carnivore {}  // 不合法
class Bee : Insect, FlyingCreature  {}// 不合法
class Flea : Insect, Carnivore {}// 不合法
```

Eagle类、Bee类和Flea类无法编译，因为它们继承自多个类，这是非法的。为了解决这个问题，我们必须把其中的某些类转换成接口。问题是转换哪个类呢？遵照一般准则，我们看出所有昆虫类共享实现，所有鸟类共享实现，所以Insect和Bird还保持为类。相反，“能飞的生物”的“能飞”是独立的机制，“食肉动物”的“食肉”是独立的机制，所以我们把FlyingCreature和Carnivore转换成接口：

```
interface IFlyingCreature {}
interface ICarnivore {}
```

在特定的语义中，Bird和Insect可以对应Windows控件和Web控件，FlyingCreature和Carnivore对应IPrintable和IUndoable。

3.7 枚举类型

枚举类型是一种特殊的数值类型，可以在枚举类型中定义一组命名的数值常量。例如：

```
public enum BorderSide { Left, Right, Top, Bottom }
```

使用枚举类型的方法如下：

```
BorderSide topside = BorderSide.Top;
bool isTop = (topside == BorderSide.Top); // true
```

每个枚举成员都对应一个整型数，默认情况下：

- 对应的数值是int型的
- 按枚举成员的声明顺序，自动指定的常量为0、1、2……

可以指定其他的整数类型代替默认类型，例如：

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

也可以显式指定每个枚举成员对应的值：

```
public enum BorderSide : byte { Left = 1, Right = 2, Top = 10, Bottom = 11 }
```

提示： 编译器还支持显式指定部分枚举成员。没有指定的枚举成员，在最后一个显式指定的值的基础上递增。上例等价于：

```
public enum BorderSide : byte
 { Left = 1, Right, Top = 10, Bottom }
```

3.7.1 枚举类型转换

枚举类型的实例可以和它对应的整型值互相显式转换：

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftorRight = (int) side <= 2;
```

也可以显式地将一个枚举类型转换成另一个。假设HorizontalAlignment定义如下：

```
public enum HorizontalAlignment
{
  Left = BorderSide.Left,
  Right = BorderSide.Right,
  Center
}
```

两个枚举类型之间的转换通过对应的数值进行：

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;
// 等价于：
HorizontalAlignment h = (HorizontalAlignment) (int)BorderSide.Right;
```

在枚举表达式中，编译器对数值0进行特别处理，不需要显式转换：

```
BorderSide b = 0;   // 不用转换
if(b == 0) ...
```

对0进行特别管理原因有两个：

- 第一个枚举成员经常被用作“默认”值
- 在合并枚举类型中，0表示不标识类型

3.7.2 标志枚举类型

枚举类型成员可以合并。为了避免混淆，合并枚举类型的成员要显式指定值，典型的增量为2。例如：

```
[Flags]
public enum BorderSides { Left = 0, Right = 2, Top = 4, Bottom = 8 }
```

使用位运算符操作合并枚举类型的值，例如|和&，它们作用在对应的整型数值上。

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;

if((leftRight & BorderSides.Left) != 0)
  Console.WriteLine ("Includes Left");       // Includes Left

string formatted = leftRight.ToString();     // "Left, Right"
BorderSides s = BorderSides.Left;
s|=BorderSides.Right
Console.WriteLine(s==leftRight);      // True

s ^=BorderSides.Right;                // 切换BorderSides.Right
Console.WriteLine(s);                 // Left
```

依照惯例，当枚举类型元素被合并时，一定要应用Flags属性。如果声明了一个没有标注Flags属性的枚举类型，枚举类型的成员仍然可以合并，但是当在该枚举实例上调用ToString方法时，输出一个数值而非一组名字。

一般来讲，合并枚举类型通常用复数名而不用单数名。

为了方便起见，可以把合并的成员直接放在枚举的声明内：

```
[Flags]
public enum BorderSides
{
    None=0,
    Left = 1, Right = 2, Top = 4, Bottom = 8
    LeftRight = Left | right,
    TopBottom = Top | Bottom,
    All = LeftRight | TopBottom
}
```

3.7.3 枚举运算符

枚举类型可以使用的运算符有：

```
=   ==   !   =   <   >   <=   >=   +   -   ^   &   |   ~
+=   -=   ++   --   sizeof
```

位运算符、算术运算符和比较运算符都返回对应整型值的运算结果。枚举类型和整型之间可以做加法，但两个枚举类型之间不能做加法。

3.7.4 类型安全问题

请看下面的枚举类型：

```
public enum BorderSide { Left, Right, Top, Bottom }
```

因为枚举类型可以和它对应的整型值相互转换，枚举的真实值可能超出枚举类型成员的数值范围。例如：

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);//12345
```

位操作和算术操作也会产生非法值：

```
BorderSide b = BorderSide.Bottom;
b++;// 不会报错
```

不合法的BorderSide枚举值会破坏如下程序：

```
void Draw(BorderSide side)
{
  if(side == BorderSide.Left)        {...}
  else if (side == BorderSide.Right) {...}
  else if (side == BorderSide.Top)   {...}
  else                               {...} // 这里被当成BorderSide.Bottom
}
```

其中一个解决方案是再加一个else子句：

```
...
else if (side == BorderSide.Bottom)...
else throw new ArgumentException ("Invalid BorderSide:" + side, "side");
```

另一个解决方案是，先检查枚举值的合法性。静态方法Enum.IsDefined有此功能：

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine (Enum.IsDefined (typeof (BorderSide), side));   // False
```

但是，Enum.IsDefined对标志枚举类型不起作用，然而下面的方法（使用Enum.ToString()），可以在标志枚举类型合法时返回true。

```
static bool IsFlagDefined (Enum e)
{
  decimal d;
  return !decimal.TryParse(e.ToString(), out d);
}

[Flags]
public enum BorderSides { Left =1, Right = 2, Top =4, Bottom = 8 }

static void Main()
{
  for(int i = 0; i<=16; i++)
  {
    BorderSide side = (BorderSide)i;
    Console.WriteLine(IsFlagDefined(side) + "  " + side);
  }
}
```

3.8 嵌套类型

嵌套类型是声明在另一个类型内部的类型。例如：

```
public class TopLevel
{
  public class Nested { }                    // 嵌套类型
  public enum Color { Red, Blue, Tan }       // 嵌套枚举类型
}
```

嵌套类型有如下特征：

- 可以访问包含它的外层类中的私有成员，以及外层类所能访问的所有内容。
- 可以使用所有的访问权限修饰符修饰，而不仅限于public和internal。
- 嵌套类型的默认访问权限是private而不是internal。
- 从外层类以外访问嵌套类型，需要用外层类名称限定（就像访问静态成员一样）。

例如，为了从TopLevel类以外访问Color.Red，我们必须：

```
TopLevel.Color color = TopLevel.Color.Red;
```

所有类型都可以被嵌套，但只有类和结构体才能嵌套其他类型。

下面是从嵌套类型访问外层私有成员的例子：

```
public class TopLevel
{
  static int x;
  class Nested
  {
    static void Foo() { Console.WriteLine(TopLevel.x); }
  }
}
```

下面是对嵌套类型使用protected访问权限修饰符的例子：

```
public class TopLevel
{
  protected class Nested { }
}

public class SubTopLevel : TopLevel
{
  static void Foo() { new TopLevel.Nested(); }
}
```

下面是从外层类以外引用嵌套类的例子：

```
public class TopLevel
{
  Public class Nested { }
}

Class Test
```

```
{
  TopLevel.Nested n;
}
```

嵌套类型在编译器中的应用也很普遍，如编译器用于生成捕获迭代和匿名方法结构状态的私有类。

提示： 如果使用嵌套类型的主要原因，是避免一个命名空间中类型定义杂乱无章，那么可以考虑使用嵌套命名空间。使用嵌套类型的原因，应该是利用它较强的访问控制能力，或者是因为嵌套类必须访问其外层类的私有成员。

3.9 泛化

C#对书写能跨类型复用的代码，有两个不同的支持机制：继承和泛化。但继承的复用性来自基类，而泛化的复用性是通过带有“占位符”类的“模板”。和继承相比，泛化能提高类型的安全性以及减少类型的转换和装箱。

提示： C#的泛化和C++的模板是相似的概念，但它们的工作方法不同。我们将在“C#泛化和C++模板的比较”章节讲解。

3.9.1 泛型

泛型中声明类型参数——占位符类型，由泛型的使用者填充，它支持类型变量。下面是一个泛型Stack<T>用于在栈中存放T类型的实例。Stack<T>声明了单个类型参数T：

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)    { data[position++] = obj; }
  public T pop()              { return data[--position]; }
}
```

使用Stack<T>的方法如下：

```
Stack<int> stack = new Stack<int>();
stack.Push(5);
stack.Push(10);
int x = stack.Pop();        // x是10
int y = stack.Pop();        // y是5
```

Stack<int>用类型参数int填充T，运行时隐式创建类型。Stack<int>具有如下的定义（为了防止混淆类名用#代替，替换类用黑体显示）：

```
public class ###
{
  int position;
  int[] data;
  public void Push(int obj)   { data [position++] = obj; }
  public int Pop()            { return data[--position]; }
}
```

技术上，我们称Stack<T>是开放类型，称Stack<int>是关闭类型。在运行时，所有泛型的实例都是关闭的——占位符类型填充。下面的语句是不合法的：

```
var stack = new Stack<T>();   //不合法：T是什么类型?
```

只有在类或方法的内部，T才可以被定义为类型参数：

```
public class Stack<T>
{
  ...
  public Stack<T> Clone()
  {
    Stack<T> clone = new Stack<T>();    // 合法
    ...
  }
}
```


3.9.2 为什么存在泛化

泛化是为了代码能跨类型复用而设计的。假定我们需要一个类型完整的栈，如果没有泛型，解决方法之一是为每个需要的元素类型硬编码类的不同版本（如IntStack、StringStack等）。显然，这将导致产生大量的重复代码。另一个解决方法是写一个用object作为元素类型的栈：

```
public class ObjectStack
{
  int position;
  object[] data = new object[10];
  public void Push(object obj)  { data[position++] = obj; }
  public object Pop()           { return data[--position]; }
}
```

但是ObjectStack类不会像硬编码的IntStack类一样只处理整型元素。而且，ObjectStack需要用到装箱和向下类型转换，这些都不能在编译时检查：

```
// 假设规定只存储整型元素
ObjectStack stack = new ObjectStack();

stack.Push("s");// 错误类型，但不会报错!
int i = (int)stack.Pop();// 向下类型转换－运行时错误
```

我们需要的栈是既能对各种不同元素类型都支持，又要有一种方法能很容易地限定栈的元素为特定类型，以提高类的安全性和减少类型转换和装箱。泛化恰好通过允许参数化元素类型，提供了这些功能。Stack<T>具有ObjectStack和IntStack的全部优点。与ObjectStack的共同点是，Stack<T>只要写一次，就可以在所有类型上都工作。和IntStack的共同点是，Stack<T>的元素是特定的某个类型，它的独特之处在于操作的类是T，我们可以在编程时任意替换。

提示： ObjectStack在功能上等价于Stack<object>。

3.9.3 泛化方法

泛化方法指在方法的标识符内声明类参数。

使用泛化方法，许多基本数学函数可以用一个通用的方法实现。下面是交换两个任意类型值的泛化方法：

```
Static void Swap<T> (ref T a, ref T b)
{
  T temp = a;
  a = b;
  b = temp;
}
```

Swap<T>的使用方法如下：

```
int x = 5;
int y = 10;
Swap(ref x, ref y);
```

通常不需要提供参数的类型给泛化方法，因为编译器可以在后台推断出类型。如果有歧义，可以用下面的方法调用泛化方法：

```
Swap<int> (ref x, ref y);
```

在泛型中，只有新引入类型参数的方法才被归为泛化方法（用尖括号标出）。泛化类Stack中的Pop方法仅仅使用了类中已经存在的类型参数T，所以不属于泛化方法。

唯有方法和类可以引入类型参数。属性、索引器、事件、字段、构造方法、运算符等都不能声明类型参数，虽然它们可以参与使用所在的类中已经声明的类型参数。例如，可以写一个索引器返回一个泛化项：

```
public T this [int index] { get { return data[index]; } }
```

类似的，构造方法可以参与使用已存在的类型参数，但不能引入新的类型参数：

```
public Stack<T>() { }    // 不合法
```

3.9.4 声明类型参数

可以在声明类、结构体、接口、委托（见第4章）和方法时引入类型参数。其他的结构，如属性，不能引入类型参数，但可以使用类型参数。例如属性Value使用T：

```
public struct Nullable<T>
{
  public T Value { get; set; }
}
```

泛型或方法可以有多个参数。例如：

```
class Dictionary<TKey,TValue> {...}
```

实例化方法：

```
Dicionary<int,string> myDic = new Dictionary<int,string>();
```

或者：

```
var myDic = new Dictionary<int,string>();
```

泛型名和泛化方法名可以被重载，只要类型参数的数量不同即可。例如，下面的两个类名不会冲突：

```
class A<T> { }
class A<T1,T2> { }
```

提示：习惯上，泛型和泛化方法如果只有一个类型参数，只要参数的含义明确，一般把这个类型参数命名为T。当使用多个类型参数时，每个类型参数都使用T作为前缀，后面跟一个更具描述性的名称。

3.9.5 typeof和无绑定泛型

在运行时不存在开放的泛型：开放泛型被汇编成程序的一部分而关闭。但运行时可能存在无绑定（unbound）泛型，只用作类对象。C#中唯一指定无绑定泛型的方法是使用typeof运算符：

```
class A<T> { }
class A<T1,T2> { }
...

Type a1 = typeof(A<>);     // 无绑定类型（注意没有指定类参数）
Type a2 = typeof(A<,>);    // 用逗号表明有多个类参数
```

开放泛型类型一般与反射API（第19章）一起使用。

也可以用typeof运算符指定关闭的类：

```
Type a3 = typeof(A<int,int>);
```

或在运行时关闭的开放类：

```
class B<T> { void X() { Type t = typeof(T); } }
```

3.9.6 泛化的默认值

可以用default关键字获取赋给泛型类参数的默认值。引用类型的默认值是null，数值类型的默认值是将类的所有字段按位置0：

```
static void Zap<T> (T[] array)
{
  For(int i = 0; i<array.Length; i++)
    array[i] = default(T);
}
```

3.9.7 泛化的约束

默认情况下，类型参数可以被任何类型替换。在类型参数上应用约束，可以定义类型参数为指定类型。下面是可用的约束：

```
where T : base-class  //基类约束
where T : interface   //接口约束
where T : class       //引用类型约束
```

```
where T : struct        //数值类型约束（排除可空类型）
where T : new()         //无参数构造方法约束
where U : T             //裸类型约束
```

下面的示例GenericClass<T,U>中，要求T派生自SomeClass且实现接口Interface1，要求U提供无参数构造方法：

```
class     SomeClass { }
interface Interface1 { }

class GenericClass<T,U> where T : SomeClass, Interface1
                        where U : new()
{ }
```

约束可以应用在方法和类的任何类型参数的定义中。

基类约束或接口约束规定类型参数必须是某个类的子类或实现特定类或接口。这允许参数类可以被隐式转换成特定类或接口。例如，假定写一个泛化方法Max，返回两个值中的较大者。我们可以利用框架中定义的泛化接口IComparable<T>：

```
public interface IComparable<T>    // 接口的简化写法
{
  int CompareTo(T other);
}
```

如果other比this大，CompareTo方法返回正值。用此接口作为约束，我们可以写如下的Max方法（为了避免分散注意力，省略了null值检查）：

```
static T Max<T> (T a, T b) where T : IComparable<T>
{
  return a.CompareTo(b) > 0 ? a : b;
}
```

Max方法可以接收任意类型的参数，实现IComparable<T>接口（该接口包含大部分内置的类，如int和string）：

```
int z = Max(5,10);// 10
string last = Max("ant","zoo");// zoo
```

类约束和结构体约束规定T必须是引用类型或数值类型（不能为空）。结构体约束的一个很好的示例是System.Nullable<T>结构体（我们在第4章“可空类型”一节详细介绍）：

```
struct Nullable<T> where T: struct{...}
```

无参数构造方法约束要求T有一个公有的无参数构造方法。如果定义了这个约束，就可以在T中调用new()：

```
static void Initialize<T> (T[] array) where T : new()
{
  for (int i = 0; i< array.Length; i++)
    array[i] = new T();
}
```

裸类型约束要求一个类型参数从另一个类型参数派生。本例中，方法FilteredStack返回另一个Stack，返回的Stack只包含原类中满足U条件的部分元素。

```
class Stack<T>
{
  Stack<U> FilterStack<U>() where U:T {...}
}
```

3.9.8 子类泛型

泛型类和非泛型的类一样，都可以作为子类。子类可以让基类中的类型参数保持开放，如下所示：

```
class Stack<T>                    {...}
class SpecialStack<T> : Stack<T> {...}
```

子类也可以用具体类型关闭泛型参数：

```
class IntStack : Stack <int> {...}
```

子类还可以引入新的类型变量：

```
class List<T>                       {...}
class KeyedList<T,Tkey> : List<T> {...}
```

提示：技术上，子类型中所有类型参数都是新的：可以说子类型关闭后又重新开放了基类的类型参数。这表明子类可以为其重新打开的类型参数使用更有意义的新名称：

```
class List<T> {...}
class KeyedList<TElement, TKey> : List<TElement> {...}
```

3.9.9 自引用泛化声明

当关闭类型参数时，类可以用自己作为实体类：

```
public interface IEqutable<T> { bool Equals (T obj); }

public class Ballon : IEquatable<Balloon>
{
  public string Color { get; set; }
  public int CC { get; set; }

  public bool Equals(Balloon b)
  {
    If(b == null) return false;
    Return b.Color == Color && b.CC = CC;
  }
}
```

下面的写法也是合法的：

```
class Foo<T> where T : IComparable<T> {...}
class Bar<T> where T : Bar<T> {...}
```

3.9.10 静态数据

对每个封装的类来说，静态数据是全局唯一的：

```
class Bob<T> { public static int Count; }

class Test
{
  static void Main()
  {
    Console.WriteLine (++Bob<int>.Count);     // 1
    Console.WriteLine (++Bob<int>.Count);     // 2
    Console.WriteLine (++Bob<string>.Count);  // 1
    Console.WriteLine (++Bob<object>.Count);  // 1
  }
}
```

3.9.11 类型参数的转换

C#的类型转换运算符可以进行多种转换，包括：

- 数值型转换
- 引用型转换
- 装箱/拆箱转换
- 自定义转换（通过运算符重载，见第4章）

根据原数据的类型，在编译时决定转换成何种类型，并实现转换。因为编译时还不知道原数据的确切类型，使得泛型参数具有有趣的语义。如果导致二义性，编译器会产生一个错误。

最常见的应用是实现一个引用的类型转换：

```
StringBuilder Foo<T> (T arg)
{
  if(arg is StringBuilder)
    return (StringBuilder) arg;   // 编译不通过
   ...
}
```

因为不知道T的确切类型，编译器认为是自定义转换。最简单的解决方法就是改用as运算符，因为它不能进行自定义类型转换，所以不会产生二义性：

```
StringBuilder Foo<T> (T arg)
{
  StringBuilder sb = arg as StringBuilder;
  if(sb!= null) return sb;
  ...
}
```

一般的做法是，先转换成object类型。这种方法能够实现，是因为转换自或转换到object类型，被假定为不是自定义转换而是引用或装箱/拆箱转换。下例中，StringBuilder是引用类型，所以必须是引用转换：

```
Return (StringBuilder)(object) arg;
```

拆箱转换也会产生二义性。下面可能是拆箱转换、数值转换或自定义转换：

```
int Foo<T> (T x) { return (int)x; }   // 编译时错误
```

下面的方法先转换成object再转换成int（很明显这是一个拆箱转换）：

```
int Foo<T> (T x) { return (int)(object)x; }
```

3.9.12 协变

假定S是B的子类，如果X<S>允许引用转换成X<B>，那么称X为协变类。

提示：由于C#符号的共变性（和逆变性），所以“可改变”表示可以通过隐式引用转换进行改变——如A是B的子类，或者A实现B。数字转换、装箱转换和自定义转换都不包含在内。

换句话说，如果下面的语句合法，那么类IFoo<T>是协变类：

```
IFoo<string> b = ... ;
IFoo<object> s = b;
```

C#4.0中，泛化接口支持协变（泛化委托也支持，见第4章），但泛化类不支持。数组也支持协变（如果S是B的子类，S[]可以转换成B[]），这里作一些讨论和比较。

提示： 协变性和逆变性（或简称变性）是高级概念。在C#中引入和强化协变的背后动机在于，允许泛化接口和泛型（尤其是框架中定义的那些，例如IEnumerable<T>）像人们所期待的那样，做更多的工作。编程人员可以利用协变性和逆变性这些概念的优点，而无需掌握它们背后的细节。

---

1. 类

为了保证静态类的安全性，泛化类不是协变的。看下面的例子：

```
class Animal { }
class Bear : Animal { }
class Camel: Animal { }

public class Stack<T>   //简单的栈实现
{
  int position;
  T[] data = new T[100];
  public void Push(T obj)    { data[position++] = obj; }
  public T Pop()             { return data[- - position]; }
}
```

下面的语句编译通不过：

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears;        // 编译时错误
```

这个限制避免了下面代码发生运行时错误：

```
animal.Push(new Camel());      // 试图把Camel类加入Bear栈
```

但是，协变性的缺失可能妨碍复用性。假定下例中，我们想写一个Wash方法来操作整个Animal栈：

```
public class ZooCleaner
```

```
{
  public static void Wash ( Stack<Animal> animals) {...}
}
```

如果调用Wash方法操作Bear栈，将会导致编译时错误。解决方法是，重新定义一个带有约束的Wash方法：

```
class ZooCleaner
{
  public static void Wash<T> (Stack<T> animals) where T : Animal {...}
}
```

这样，我们就可以用如下方法调用Wash了：

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash(bears);
```

另一种解决方法是让Stack<T>实现一个协变的泛化接口，后面会举出示例。

2. 数组

由于历史原因，数组array类型具有协变性。这表明如果B是A的子类（A和B都是引用类型），那么B[]可以转换成A[]。例如：

```
Bear[] bears = new Bear[3];
Animal[] animals = bears;            // 正确
```

这种可复用性的缺点在于，指定元素值可能导致运行时错误：

```
animal[0] = new Camel();     // 运行时错误
```

3. 接口

在C#4.0中，泛化接口对用out修饰符标注的类型参数支持协变。和数组不同，out修饰符保证了协变性的接口是完全类型安全的。为了阐述这一点，假定Stack类实现了下面的接口：

```
public interface IPoppable<out T> { T Pop(); }
```

T前的out修饰符是C#4.0的新特性，表明T只用在输出的位置（例如：方法的返回值）。out修饰符将接口标识为具有协变性的并允许以下操作：

```
var bears = new Stack<Bear>();
bears.Push (new Bear());
// Bears实现IPoppable<Bear>。我们可以把它转换成IPoppable<Animal>：
IPoppable<Animal> animals = bears;   // 合法
Animal a = animal.Pop();
```

基于接口有协变性的优点，将bears转换成animals是编译器允许的。它是类型安全的，因为编译器试图避免发生Camel类进栈，因为T仅能在输出的位置出现，所以不可能将Camel类输入接口。

提示： 接口中的协变和逆变的典型应用是使用接口：很少需要向协变性接口写入。确切地说，由于CLR的限制，为了协变性将方法参数标注为out是不合法的。

如前所述，我们可以利用类型转换的协变性解决复用性问题：

```
public class ZooCleaner
{
  public static void Wash(IPoppable<Animal> animals) {...}
}
```

提示： 第7章所讲的IEnumerator<T>和IEnumerable<T>接口在Framwork4.0中都标识为协变的。这样，如果需要，可以将IEnumerable<string>转换成IEnumerable<object>。

如果在输入的位置使用协变类型参数（例如：方法的参数或可写属性），编译器会产生错误。

提示： 不管泛型还是数组，协变（逆变）仅对引用转换的元素有效而对装箱转换无效。因此，如果写了一个接收IPoppable<object>类为参数的方法，可以用IPoppable<string>作为参数调用，而不能用IPoppable<int>。

3.9.13 逆变

前面我们已经知道，如果S是B的子类且X(S)允许引用类型转换到X(B)，则类X是协变的。那么能反方向转换的类是逆变的——从X(B)到X(S)。泛化接口支持逆变当泛型参数只出现在输入的位置，且被指定了in修饰符时。扩展我们前面的示例，如果Stack<T>类实现了如下接口：

```
public interface IPushble<in T> { void Push(T obj); }
```

下面的语句是合法的：

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;      // 合法
bears.Push (new Bear());
```

IPushable中没有成员输出类型为T，所以不会出现将animals转换成bears的问题（但是通过这个接口不能实现Pop方法）。

提示： Stack<T>类既可以实现IPushable<T>接口也可以实现IPoppable<T>接口——尽管T在两个接口中有不同的协变注解。它可以实现的原因是，只能从一个接口中激活协变，因此在协变转换之前，必须先选定IPoppable或IPushable。选定的接口会限制在合适的协变规则下进行合法的操作。

这也说明了为什么将类（如Stack<T>）定义为协变是没有意义的。

再看一个例子，下面的接口是.NET框架中的定义：

```
Public interface IComparer<in T>
{
  // 返回值为a和b的相对顺序
  int Compare(T a, T b);
}
```

因为这个接口是逆变的，我们可以用IComparer<object>比较两个字符串：

```
var objectComparer = Comparer<object>.Default;
// objectComparer实现了IComparer<object>接口
IComparer<string> stringComparer = objectComparer;
int result = stringComparer.Compare("Brett", "Jemaine");
```

与之相对的协变，如果将逆变的参数用在输出的位置（例如，作为返回值或在可读属性中），编译器将会报错。

3.9.14 C#泛化与C++模板的比较

C#的泛化与C++的模板在应用上很相似，但它们的工作原理却大不相同。两者都发生了生产者和消费者的语义关联，其中生产者的占位符类型由消费者填充。但是C#的泛化中，生产者类（例如开放类型List<T>）可以被编译到程序库中（如*mscorlib.dll*），因为在生产者和产生关闭类型的消费者间的语义交换直到运行时才实际发生。而C++模板中，这一语义交换是在编译时进行的。这表明我们不能在C++中以*.dll*形式部署模板库，因为生产者类仅存在于源代码中。这使得动态语法检查很难实现，更不用说联机创建或参数化类了。

为了深究这一情形的产生原因，我们重新观察C#的Max方法：

```
static T Max<T> (T a, T b) where T : IComparable<T>
{
  return a.CompareTo(b) > 0 ? a:b;
}
```

为什么我们不能像下面这样做呢？

```
static T Max<T> (T a, T b)
{
  return a > b ? a:b;                  //编译时错误
}
```

原因是，Max需要编译一次后对所有可能的T值都能工作。上例编译不能通过，因为对于任意类型T，“>”没有统一的含义。实际上，并不是所有的类型都支持“>”运算符。与之对应，下面的代码是用C++模板写的同样Max方法。该代码会为每个T值分别编译，对特定T呈现“>”不同语义，如果某个T不支持“>”运算符，则编译通不过：

```
Template <class T> T Max (T a, T b)
{
  return a > b ? a : b;
}
```