目录

[什么是.Net框架，框架的编译运行原理 2](#_Toc425998644)

[变量的范围，常量表，verbatim，类型转换，操作符的优先级 7](#_Toc425998645)

[Namespace，using，const，访问修饰符，sealed,virtual,override 10](#_Toc425998646)

[Loops，Enumeration，typeof，struct，Array 15](#_Toc425998647)

[String manipulation 23](#_Toc425998648)

[泛型，类加载问题，Nullable types，operators and nullable type，?? Operator, List*<*T*>,* LinkedList<T> 26](#_Toc425998649)

[Sorting and Searching Generic Lists, Dictionary<K,V> 33](#_Toc425998650)

[Define Generic Types, default keyword, constraining types, IEnumerable和IEnumerator的区别，Inheriting from Generic Classes 37](#_Toc425998651)

[Generic interface, Defining Generic Methods 47](#_Toc425998652)

[Indexer，using indexer，indexer in interface，Difference between Property and Indexer 49](#_Toc425998653)

[Property，Using Property，interface Property，Restricting Accessor Accessibility，new as modifier，auto-implemented property 54](#_Toc425998654)

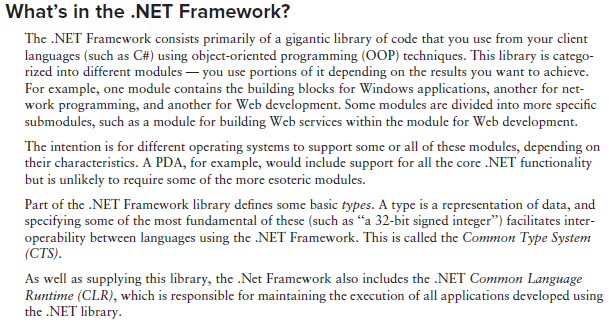
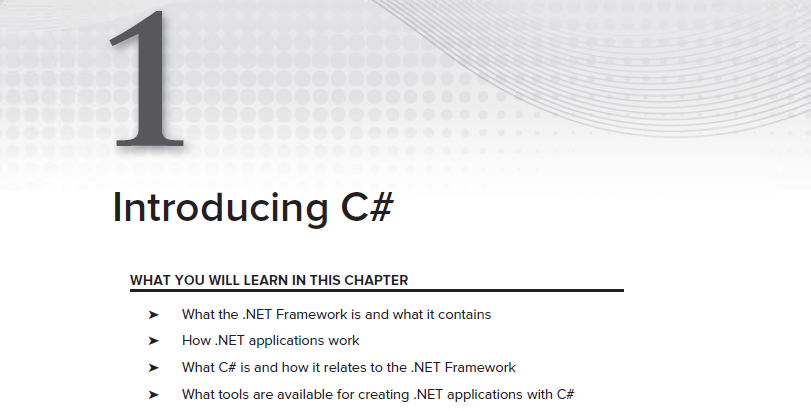
[Delegate, Using delegate, Delegates with Named vs. Anonymous Methods, Variance in Delegate, Variance in Generic Type Parameters 65](#_Toc425998655)

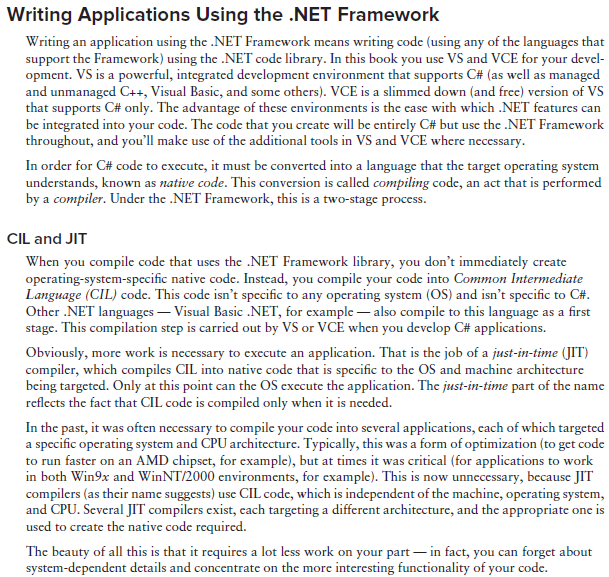
[Collection, abstract class, Static方法不能被重写, IDictionary, Iterator(prime numbers algorithem), Deep copying and shallow copying 75](#_Toc425998656)

[Comparison, is operator, Value Comparison, operator overloading, IComparable and IComparer, Sorting, As operator 87](#_Toc425998657)

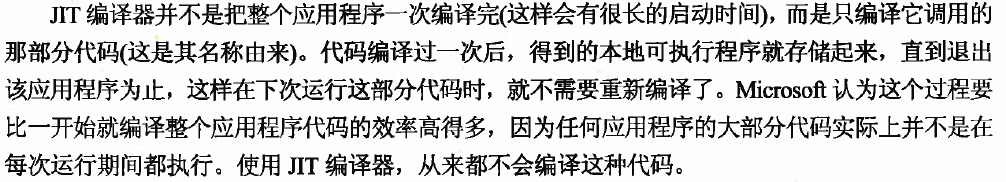
[关键字详解：static 99](#_Toc425998658)

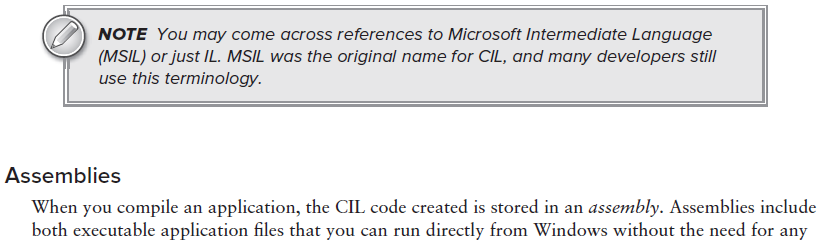
## 什么是.Net框架，框架的编译运行原理

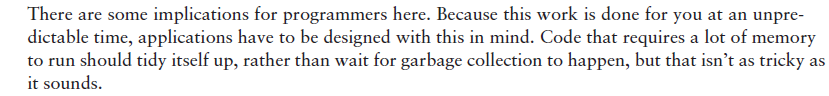
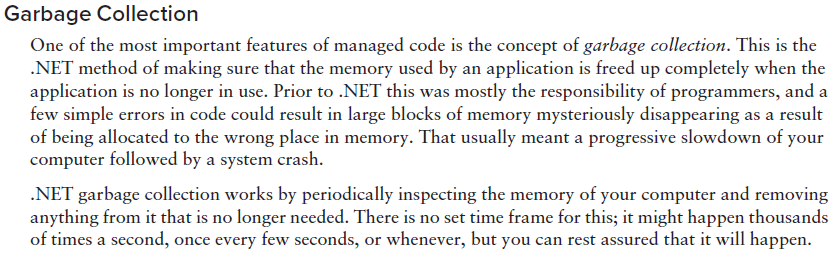
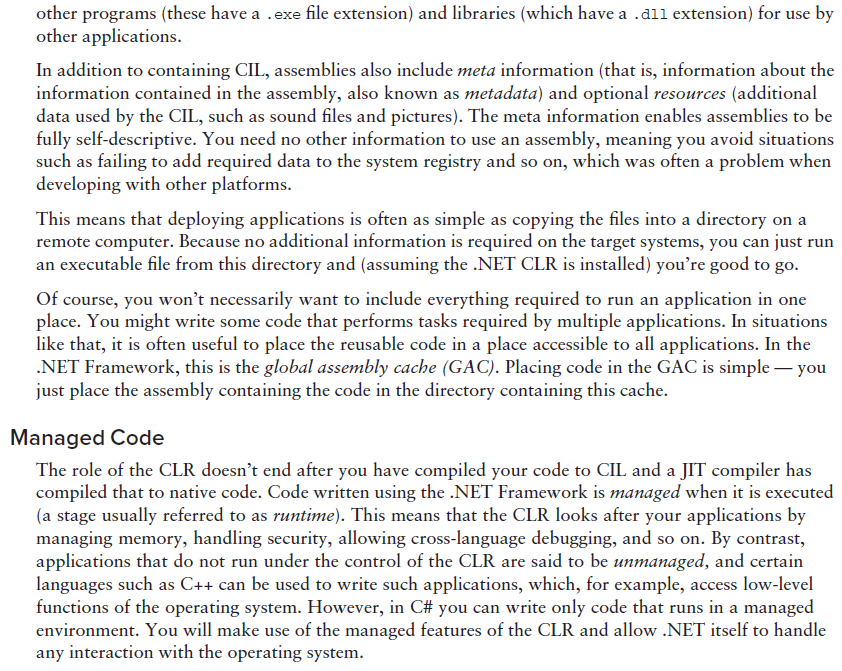


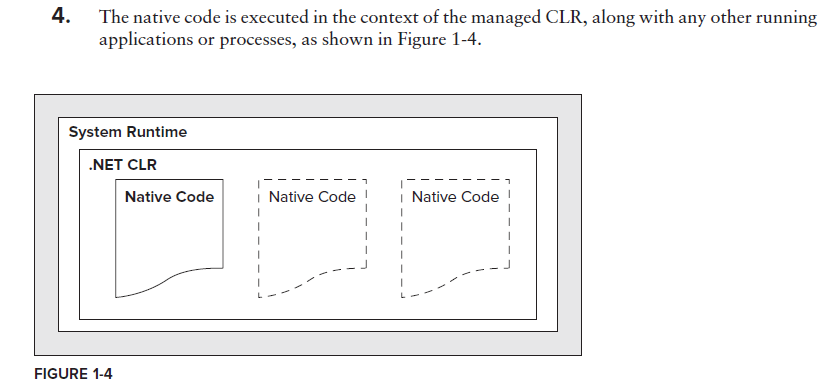
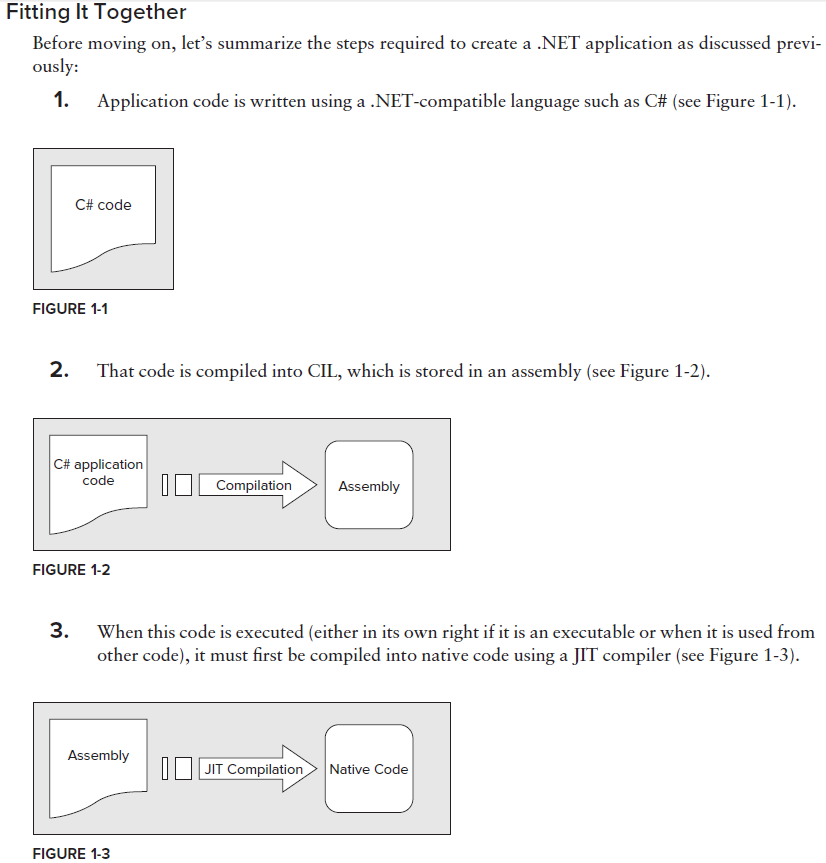


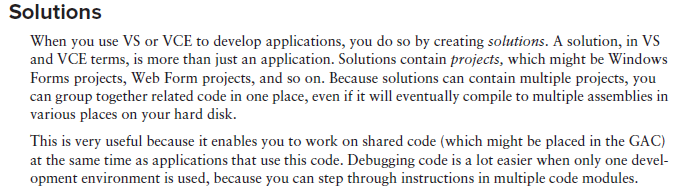
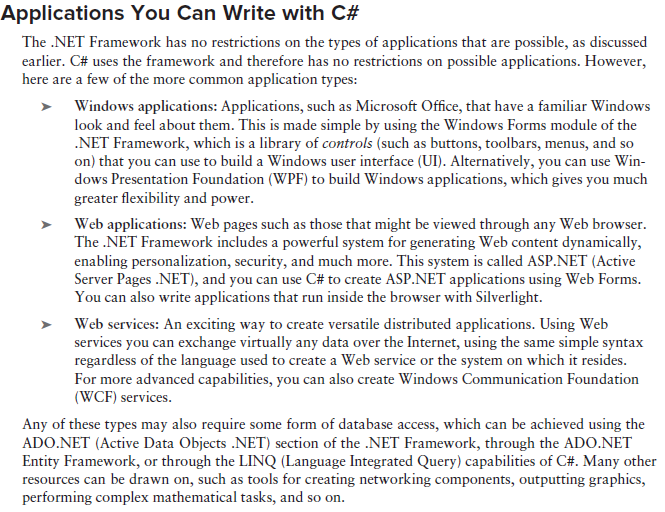
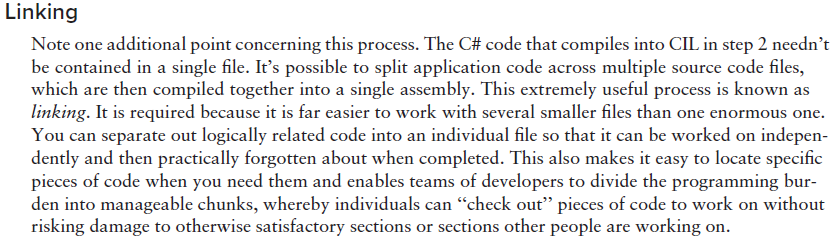
JIT在需要的时候把CIL编译成机器对应的native code，然后就可以运行



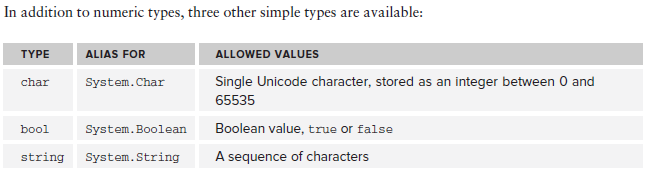
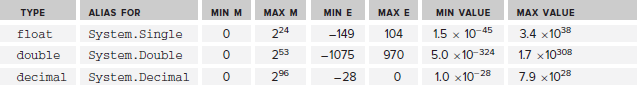
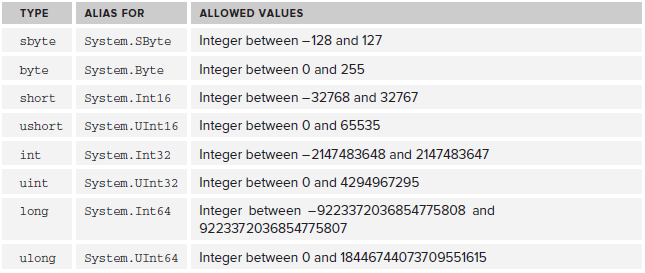




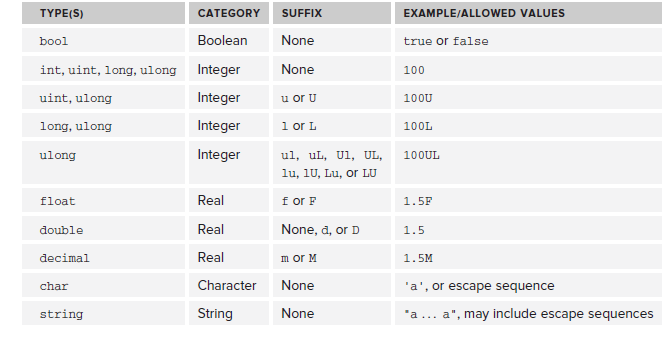




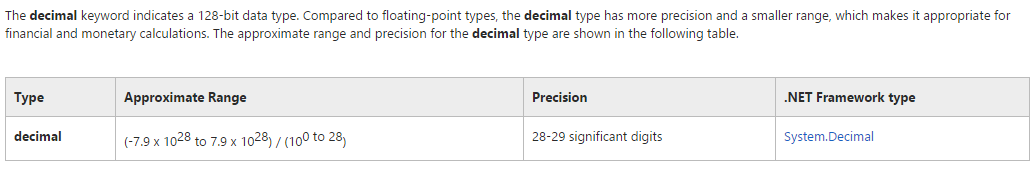
## 变量的范围，常量表，verbatim，类型转换，操作符的优先级



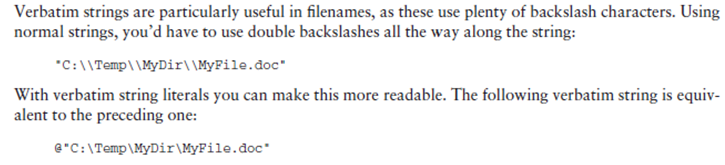
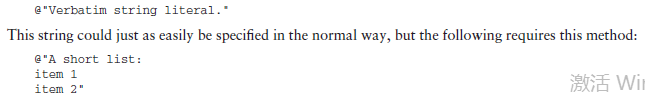
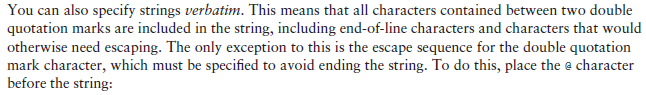
char在内存中其实是以integer的方式存储，每一个0-65535范围内的integer都有他对应的char，所以char比较大小也是比较他的integer的大小，也是因为这样char才能转型成int。



注意，带小数点的常量默认类型是double,所以float f= 1.5是错的，double不能自动转型成float，要写成float f = 1.5f

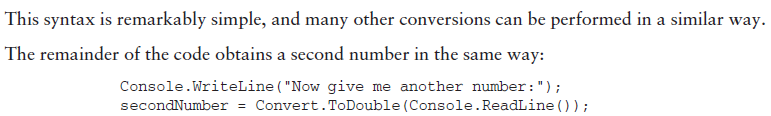


注意decimal，占128bit，他比double精度更高，适合金融或者货币这种高精度计算



Verbatim非常有用，可以让\不被当做转义符来看待。

类型转换：



基本类型的转换和java一样，也是那幅图。

强制转换也和java一样，c#提供了额外的强制转换范围检查，比如byte的最大值为256，当企图把一个short类型，值为281的变量强制转换为byte时，会发生错误。看下面程序

class Program

{

static void Main(string[] args)

{

byte destinationVar;

short sourceVar = 281;

destinationVar = (byte)sourceVar;

Console.WriteLine("destinationVar is {0}", destinationVar);

Console.ReadKey();

}

}

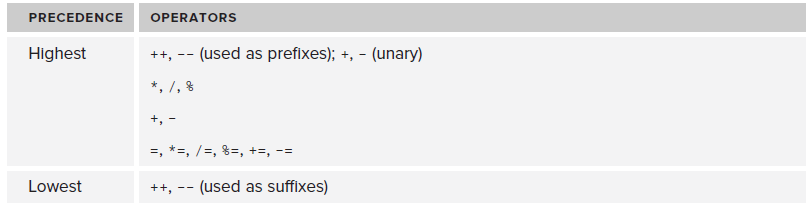
此时sourceVar超过了byte的最大范围，而编译器并不会给出任何警告，打印出了错误的结果destinationVar is 25。如果想让编译器在运行的时候给出警告，则可以使用checked的关键字。如下所示destinationVar = checked((byte)sourceVar)

普通数据类型转换为string:调用ToString方法，或者Convert.ToString()

String转换为普通数据类型：调用Convert.ToInt32(string val),Convert.ToDouble(string val)等等。。

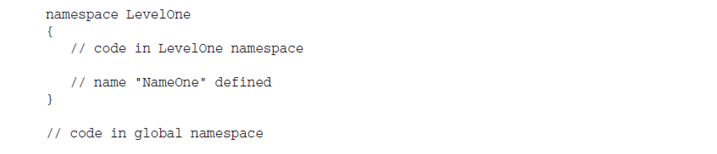


操作符的优先级

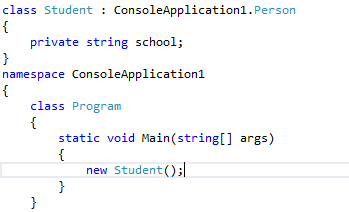


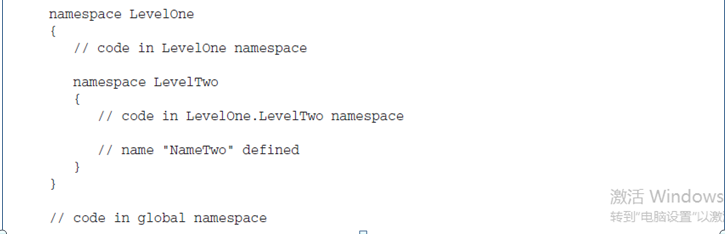
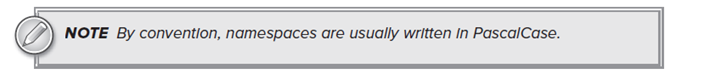
可以用括号来改变优先级

## Namespace，using，const，访问修饰符，sealed,virtual,override

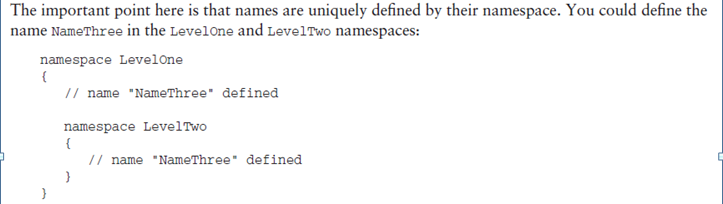


没有在namespace里定义的代码都在global namespace里

从上面程序中可以看到，global namespace里想要访问ConsoleApplication1里的类，需要加上空间名，而1里的类访问Student类就不需要

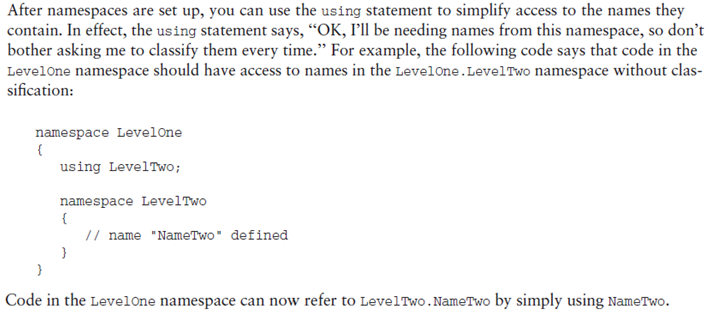


Namespace可以嵌套，在LevelOne中访问LevelTwo，要用LevelTwo.NameTwo，在global中访问则要用LevelOne.LevelTwo.NameTwo

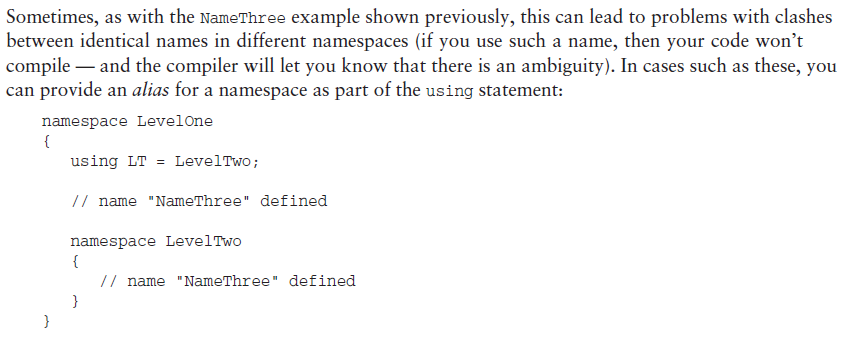


两个NameThree是不同的

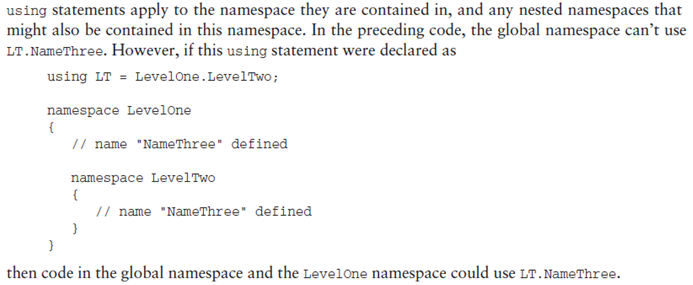
using keyword



在LevelOne里使用using LevelTwo,就可以直接在LevelOne里使用NameTwo，而不需要前缀了

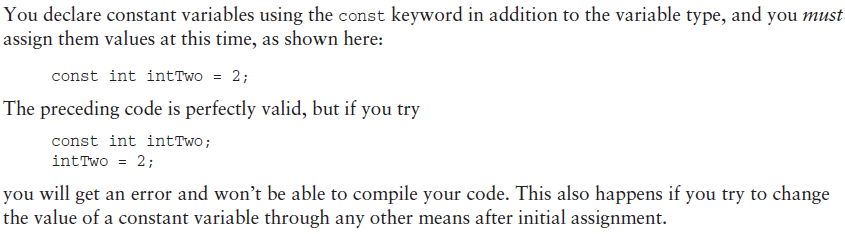


这样做会出现一个问题，看上面程序，LevelOne和LevelTwo里都定义了一个NameThree，如果在LevelOne里用了using LevelTwo，这样两个NameThree就无法区分开来，这时可以给LevelTwo取一个别名，using LT=LevelTwo

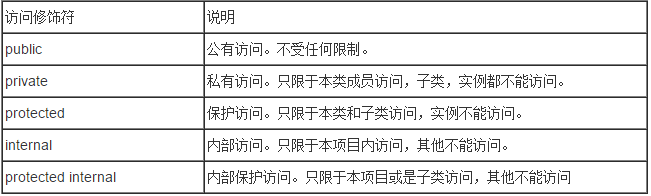


注意，using 影响当前namespace和当前namespace嵌套的namespace，比如上面代码，在global和LevelOne中均可以用LT.Name来代表LevelOne.LevelTwo.NameThree

const keyword



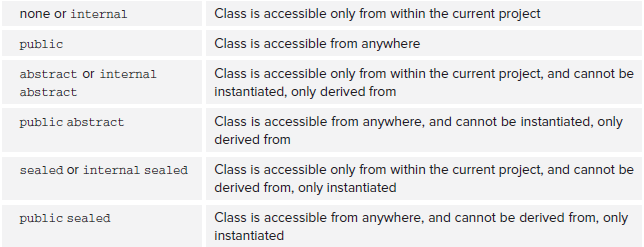
C#中没有final关键字，const类似于final关键字，但是只能变量，不能加在方法和类之前。那么怎么能规定一个类不能被继承，或者一个方法不能被重写呢？下面就会讲到，先来总结一下c#中的访问修饰符。



C#多了一个internal修饰符，类似于java中的default，不过default是包权限，而internal是项目内权限。

protected是项目内权限和子类权限，如果一个类A有一个protected成员。同一个项目的不同类可以访问protected成员，不同项目的不同类不能访问protected成员，无论哪里，只要继承了A，就可以访问这个成员。

再来看看class可以使用的前缀



这里的or代表默认情况，也就是说如果一个class不带修饰符，那他默认就是internal的，如果一个abstract class不带修饰符，那么他默认也是internal的，一个sealed class不带修饰符，那它默认也是internal的。也就是说class的默认修饰符是internal，就像java是default一样。

注意public,internal,proteced,private是权限修饰符，sealed和abstract不是，组合用的时候权限修饰符放在前面。

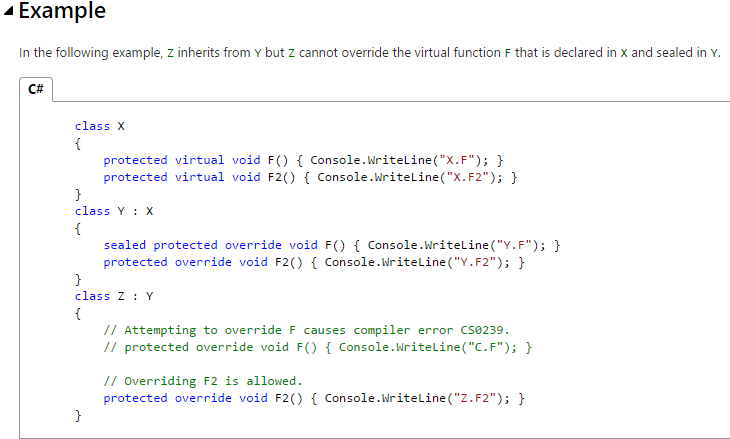
刚才说到防止class被继承，c#中是用sealed关键字，sealed关键字在方法重写中还有一个用途，稍后会讲解

virtual, override

C#中方法重写和java有显著不同，java是默认重写，c#是要显式的指明重写。在父类需要子类重写的方法必须用virtual（或者override）指定，virtual要写在返回值前面，如public virtual void….子类要重写这个方法需要用override关键字，public override void…所以你如果不想一个父类的某方法被重写，那么就不要给这个方法加virtual关键字。

C#中如果一个子类重写了父类方法，那它就用了override关键字，此时它的子类就可以再用override重写这个方法。那么如果不想让这个方法被重写下去了，该怎么办呢？这里就可以用到sealed关键字的另一个用途，看msdn里面的一段描述：

You can also use the **sealed** modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.



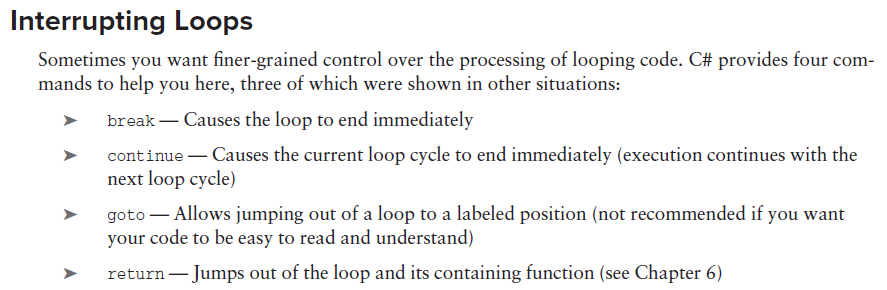
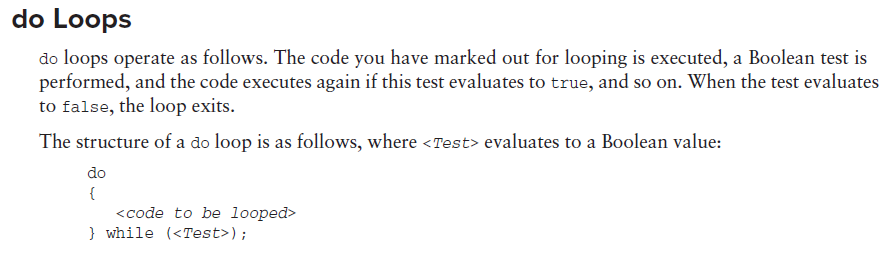
When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as [virtual](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx).

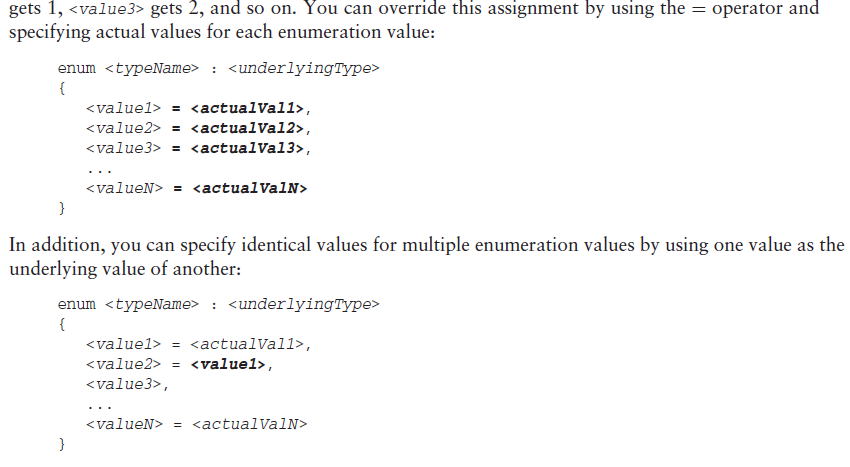
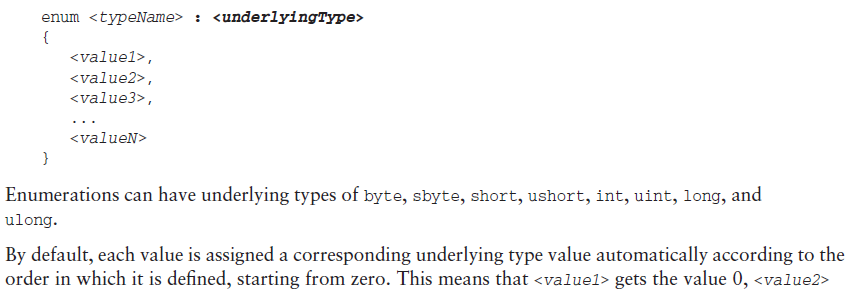
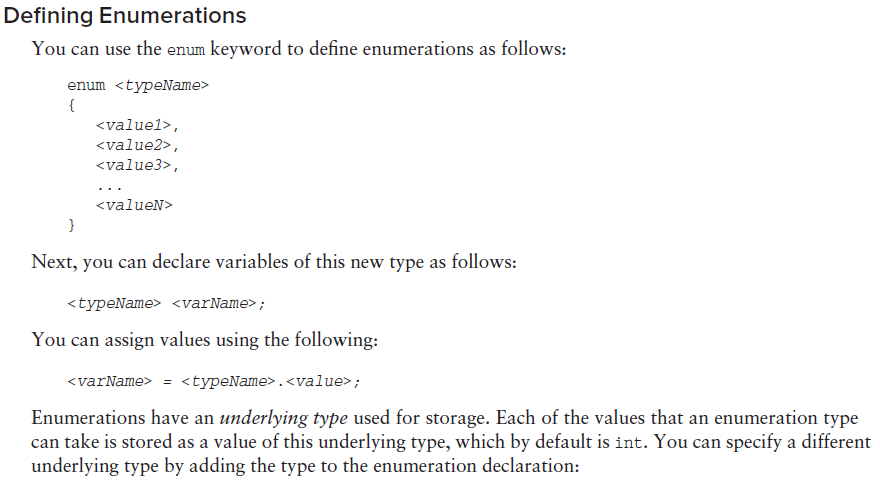
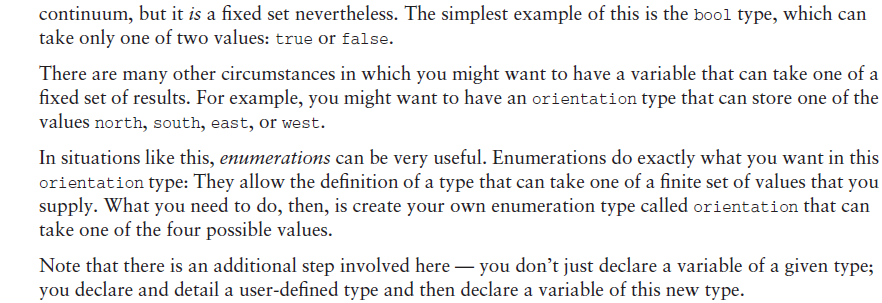
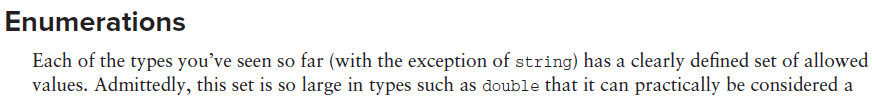
It is an error to use the [abstract](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx) modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

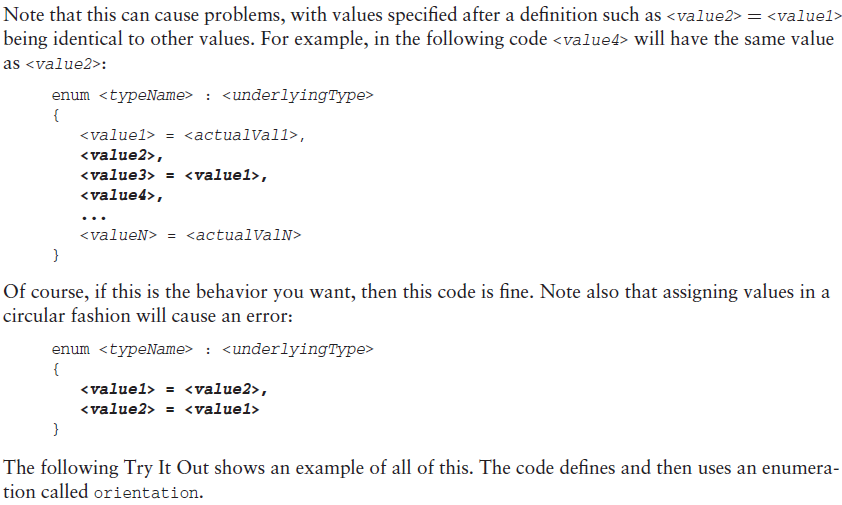
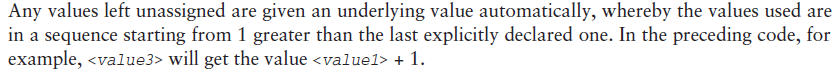
When applied to a method or property, the **sealed** modifier must always be used with [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx).（虽然例子中sealed写在方法声明的最前面，但其实也可以写在protected的后面，推荐写在后面，这样和sealed用在class中的位置相同）

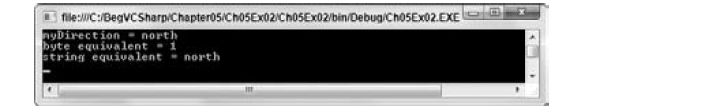
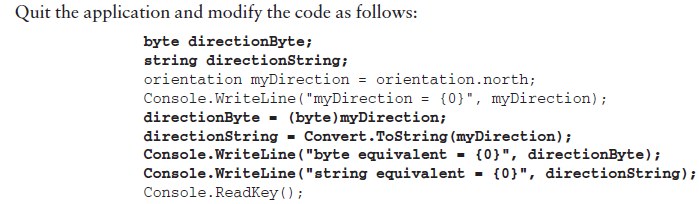
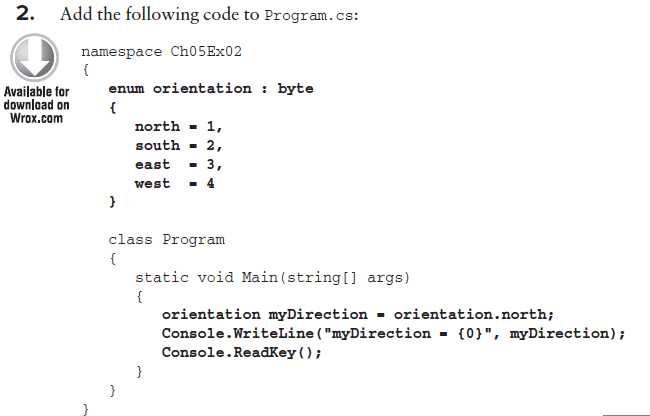
Because structs are implicitly sealed, they cannot be inherited.

## Loops，Enumeration，typeof，struct，Array









Note that you must use explicit conversions here. Even though

the underlying type of orientation is byte, you still have to use the(byte)cast to convert the value of myDirection into a byte type:

directionByte = (byte)myDirection;

The same explicit casting is necessary in the other direction, too, if you want to convert a byte into an orientation. For example, you could use the following code to convert a byte variable called myByte into an orientation and assign this value to myDirection:

myDirection = (orientation)myByte;

注意，上面的转换不限类型，如果enum的underlying type是byte，那么这个myByte的类型不一定非要是byte，可以是 int myByte, double myByte，不过这样做要小心，要注意enum的取值范围

typeof

Used to obtain the **System.Type** object for a type. A **typeof** expression takes the following form:

System.Type type = typeof(int);

To obtain the run-time type of an expression, you can use the .NET Framework method [GetType](https://msdn.microsoft.com/en-us/library/system.object.gettype.aspx), as in the following example:

int i = 0;

System.Type type = i.GetType();

typeof和GetType()的区别：

public class ExampleClass

{

public int sampleMember;

public void SampleMethod() {}

static void Main()

{

Type t = typeof(ExampleClass);

// Alternatively, you could use

// ExampleClass obj = new ExampleClass();

// Type t = obj.GetType();

判断一个对象的类型是否等于某类型：obj.GetType()==typeof(ExampleClass);

Struct

A **struct** type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory. The following example shows a simple struct declaration:

public struct Book

{

public decimal price;

public string title;

public string author;

}

注意，struct的成员要声明成public，因为它们default是private的，而不是internal的。

Struct可以不用new 初始化就能使用，比如：

static void Main(string[] args) {

Book myBook;

myBook.price=12.8;

}

上面程序如果是用class，那么必须用new初始化之后才能赋值。

Structs can also contain [constructors](https://msdn.microsoft.com/en-us/library/ace5hbzh.aspx), [constants](https://msdn.microsoft.com/en-us/library/ms173119.aspx), [fields](https://msdn.microsoft.com/en-us/library/ms173118.aspx), [methods](https://msdn.microsoft.com/en-us/library/ms173114.aspx), [properties](https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx), [indexers](https://msdn.microsoft.com/en-us/library/6x16t2tx.aspx), [operators](https://msdn.microsoft.com/en-us/library/ms173145.aspx), [events](https://msdn.microsoft.com/en-us/library/awbftdfh.aspx), and [nested types](https://msdn.microsoft.com/en-us/library/ms173120.aspx), although if several such members are required, you should consider making your type a class instead.

Structs can implement an interface but they cannot inherit from another struct. For that reason, struct members cannot be declared as **protected**.

注意struct是数值类型，不是reference类型：

Structs vs. Classes

Structs may seem similar to classes, but there are important differences that you should be aware of. First of all, classes are reference types and structs are value types. By using structs, you can create objects that behave like the built-in types and enjoy their benefits as well.

Heap or Stack?

When you call the New operator on a class, it will be allocated on the heap. However, when you instantiate a struct, it gets created on the stack. This will yield performance gains. Also, you will not be dealing with references to an instance of a struct as you would with classes. You will be working directly with the struct instance. Because of this, when passing a struct to a method, it's passed by value instead of as a reference.

The **struct** type is suitable for representing lightweight objects such as **Point**, **Rectangle**, and **Color**. Although it is just as convenient to represent a point as a [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) with [Auto-Implemented Properties](https://msdn.microsoft.com/en-us/library/bb384054.aspx), a [struct](https://msdn.microsoft.com/en-us/library/ah19swz4.aspx) might be more efficient in some scenarios. For example, if you declare an array of 1000 **Point** objects, you will allocate additional memory for referencing each object; in this case, a struct would be less expensive. Because the .NET Framework contains an object called [Point](https://msdn.microsoft.com/en-us/library/system.drawing.point.aspx), the struct in this example is named "CoOrds" instead.

It is an error to define a default (parameterless) constructor for a struct. It is also an error to initialize an instance field in a struct body. You can initialize struct members only by using a parameterized constructor or by accessing the members individually after the struct is declared. Any private or otherwise inaccessible members can be initialized only in a constructor.

When you create a struct object using the [new](https://msdn.microsoft.com/en-us/library/51y09td4.aspx) operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the **new** operator. In such a case, there is no constructor call, which makes the allocation more efficient. However, the fields will remain unassigned and the object cannot be used until all of the fields are initialized.

When a struct contains a reference type as a member, the default constructor of the member must be invoked explicitly, otherwise the member remains unassigned and the struct cannot be used. (This results in compiler error CS0171.)

There is no inheritance for structs as there is for classes. A struct cannot inherit from another struct or class, and it cannot be the base of a class. Structs, however, inherit from the base class [Object](https://msdn.microsoft.com/en-us/library/system.object.aspx). A struct can implement interfaces, and it does that exactly as classes do.

**Array:**

Array的初始化1）静态初始化：int[] myIntArray = { 5, 9, 10, 2, 99 };编译器更加数组内容自动设定数组长度

2）动态初始化：int[] myIntArray = new int[5];定义一个长度为5的数组，此时数组的每个元素都默认为0，或者：, int[] myIntArray = new int[size]，注意，数组一旦初始化，长度就不能改变。所以建议把size定义为常量,const int size。

3）组合初始化，也可以把两种方式综合起来：int[] myIntArray = new int[5] { 5, 9, 10, 2, 99 };注意，数组的大小要和初始化的值的个数相等。int[] myIntArray = new int[10] { 5, 9, 10, 2, 99 };这样的初始化会引起错误

**Multidimensional Arrays:**

A two-dimensional array such as this is declared as follows:

<*baseType*>[,] <*name*>;

Arrays of more dimensions simply require more commas:

<*baseType*>[,,,] <*name*>;

动态初始化：double[,] hillHeight = new double[3,4];

静态初始化：double[,] hillHeight = { { 1, 2, 3, 4 }, { 2, 3, 4, 5 }, { 3, 4, 5, 6 } };

取得其中的一个元素：hillHeight[2,1]

遍历数组：double[,] hillHeight = { { 1, 2, 3, 4 }, { 2, 3, 4, 5 }, { 3, 4, 5, 6 } };

foreach (double height in hillHeight)

{

Console.WriteLine("{0}", height);

}

注意这里和java不同，二维数组并没有被当做数组的数组，所以不需要两个foreach嵌套循环

**Arrays of Arrays（jagged array）：**

上面介绍的多维数组是“长方形”的数组，所谓长方形的数组，是说它的维度是固定的，比如 double[3,4],是一个三行四列的二维数组。

那么如果不想要一个维度“长方形”的数组呢？这时可以用数组的数组

The syntax for declaring arrays of arrays involves specifying multiple sets of square brackets in the declaration of the array, as shown here:

int[][] jaggedIntArray;

初始化问题：

Unfortunately, initializing arrays such as this isn’t as simple as initializing multidimensional arrays.

You can’t, for example, follow the preceding declaration with this:

jaggedIntArray = new int[3][4];

or jaggedIntArray = { { 1, 2, 3 }, { 1 }, { 1, 2 } };

那么怎么初始化呢？

You have two options. You can initialize the array that contains other arrays (we’ll call these sub-arrays for clarity) and then initialize the sub-arrays in turn:

jaggedIntArray = new int[2][];

jaggedIntArray[0] = new int[3];

jaggedIntArray[1] = new int[4];

Alternately, you can use a modified form of the preceding literal assignment:

int[][] jaggedIntArray = { new int[] { 1, 2, 3 }, new int[] { 1 },

new int[] { 1, 2 } };

遍历数组，需要用多重for循环

foreach (int[] divisorsOfInt in divisors1To10)

{

foreach(int divisor in divisorsOfInt)

{

Console.WriteLine(divisor);

}

}

也可以用index来遍历

int[][] jaggedIntArray = { new int[] { 1, 2, 3 }, new int[] { 1 },

new int[] { 1, 2 } };

for(int i=0; i<jaggedIntArray.Length; i++)

{

for(int j=0; j<jaggedIntArray[i].Length; j++)

{

Console.WriteLine(jaggedIntArray[i][j]);

}

}

## String manipulation

1. ToCharArray:

string myString = "A string";

char[] myChars = myString.ToCharArray();

和java中不同的地方，c#可以用foreach来直接遍历一个string的每个字符

string myString = "A string";

foreach (char character in myString)

{

Console.WriteLine("{0}", character);

}

而这段在代码在java中是不可以的

1. 得到string的长度:C#中是myString.Length，在java中是myString.Length()
2. 两个有用的方法，ToLower()，ToUpper()

当你在判读一个用户输入的类容是不是yes的时候，要考虑到用户有可能会输入”YES”,”Yes”,”yes”等多种可能，所以要先用ToLower把输入转换为全小写

string userResponse = Console.ReadLine();

if (userResponse.ToLower() == "yes")

{

// Act on response.

}

注意，和java里一样，string是不可变的，所以ToLower不会改变原string的类容，只是会返回一个全小写的新string，需要一个变量来储存它，也可以直接用原变量储存userResponse = userResponse.ToLower();

1. Trim()方法：

在上面的情况中，如果用户不小心在yes前面或者后面多输入了一个空格，那么这个程序将无法运行，所以可以用Trim方法来去掉一个string前面或者后面的空格

string userResponse = Console.ReadLine();

userResponse = userResponse.Trim();

if (userResponse.ToLower() == "yes")

{

// Act on response.

}

注意，Trim只能去掉string开始和结尾的空格，并不能去掉中间的空格。

如” adafs sdf safas ”调用Trim之后得到”adafs sdf safas”

You can also use these commands to remove any other characters, by specifying them in a char array,

for example:

char[] trimChars = {’ ‘, ‘e’, ‘s’};

string userResponse = Console.ReadLine();

userResponse = userResponse.ToLower();

userResponse = userResponse.Trim(trimChars);

if (userResponse == "y")

{

// Act on response.

}

注意，这样也是去掉string开始和结尾的’ ’，’e’，和’s’。简单的说，就是把空格，e，s都当做空格，比如” yesseyeee sssyses”调用Trim(trimChars)之后得到”yesseyeee sssy”

Trim从开始和结尾去掉空格，如果想从开始去，可以用TrimStart()，只想从结尾去，可以用TrimEnd()，这两个方法同样也可以加一个char[]参数

1. PadLeft()和PadRight()方法：

You can use two other string commands to manipulate the spacing of strings: <*string*>.PadLeft() and <*string*>.PadRight(). They enable you to add spaces to the left or right of a string to force it to the desired length. You use them as follows:

<*string*>.PadX(<*desiredLength*>);

**Here is an example:**

myString = "Aligned";

myString = myString.PadLeft(10);

注意，不是在Aligned左边加10个空格，而是加一些空格，直到这个string长度到10

These methods can be helpful when aligning strings in columns, which is particularly useful for positioning strings containing numbers.

As with the trimming commands, you can also use these commands in a second way, by supplying the character to pad the string with. This involves a single char, not an array of chars as with trimming:

**myString = "Aligned";**

**myString = myString.PadLeft(10, ‘-’);**

This would add three dashes to the start of myString.

## 泛型，类加载问题，Nullable types，operators and nullable type，?? Operator, List*<*T*>,* LinkedList<T>

you can create generic interfaces, generic methods (which can be defined on nongeneric classes), and even generic delegates.

Behind the scenes, the .NET runtime allows generic classes to be dynamically generated as and when you need them. A given generic class A of B won’t even exist until you ask for it by instantiating it. 注意，泛型类之间没有默认的转化，如Farm<Animal> a=new Farm<Cow>是不行的，尽管Cow是Animal的子类。

类加载问题

class Table

{

static Bowl b1=new Bowl(1);

static Bowl b2 = new Bowl(2);

Bowl b3 = new Bowl(3);

public Table()

{

Console.WriteLine("Table");

}

static public void showTable()

{

Console.WriteLine("static method of table");

}

}

class Bowl

{

public Bowl(int marker)

{

Console.WriteLine("Bowl({0})", marker);

}

}

class Program

{

static Table t;

static void Main(string[] args)

{

Table.showTable();

Console.WriteLine(t);

}

}

在Main方法执行时，Program类会被加载，此类有一个静态Table类变量t，由于这个静态变量并没有被实例化，所以Table类此时并没有被加载。在main方法中调用了Table里的静态方法，此时必须加载Table类，加载Table类后，Table类有两个静态Bowl变量，这两个变量被实例化了，所以会加载Bowl类以及其构造方法。注意到Table类还有一个非静态的Bowl变量，因为Table并没有被实例化，所以这个变量也不会被实例化。

可以说明，如果一个类含有static成员，此成员会在实例化或者调用它的static方法时才会被加载

Nullable types

有时在和数据库打交道时，比如查找符合条件的一组记录的id，返回值是id，如果查找没有结果，你会希望返回null，然而int类型的id不能是null，这时候就需要nullable type

Generics give you a way to do this using the System.Nullable<T> type, as shown in this example:

**System.Nullable<int> nullableInt;**

**nullableInt = new System.Nullable<int>();**

You can test nullable types to determine whether they are null, just like you test reference types:

if (nullableInt == null) { ... }

Alternatively, you can use the HasValue property:

if (nullableInt.HasValue) { ... }注意，HasValue只能用在nullable type上

You can also look at the value of a nullable type by using the **Value** property. If **HasValue** is true, then you are guaranteed a non-null value for Value; but if HasValue is false — that is, null has been assigned to the variable — then accessing Value will result in an exception of type System.InvalidOperationException.

简写：int? nullableInt;

看下面这个例子：

class Program

{

static void Main(string[] args)

{

Vector v = new Vector();//error, no such constructor.

Console.ReadKey();

}

}

class Vector

{

public double? R;

public double? Theta;

public Vector(double? r, double? theta)

{

R = r;

Theta = theta;

}

}

注意到，Vector的constructor需要两个nullable double作为参数，而在Main中没有任何参数，没有参数并不代表着null，需要显示的传入两个null，像这样Vector v=new Vector();

Operators and nullable type:

With simple types, such as **int**, you can use operators such as +, -, and so on to work with values. With

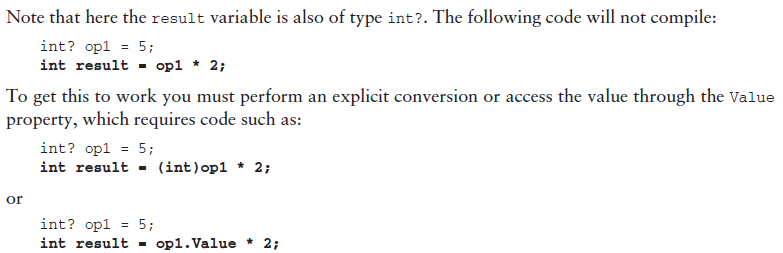
nullable type equivalents, there is no difference: The values contained in nullable types are implicitly

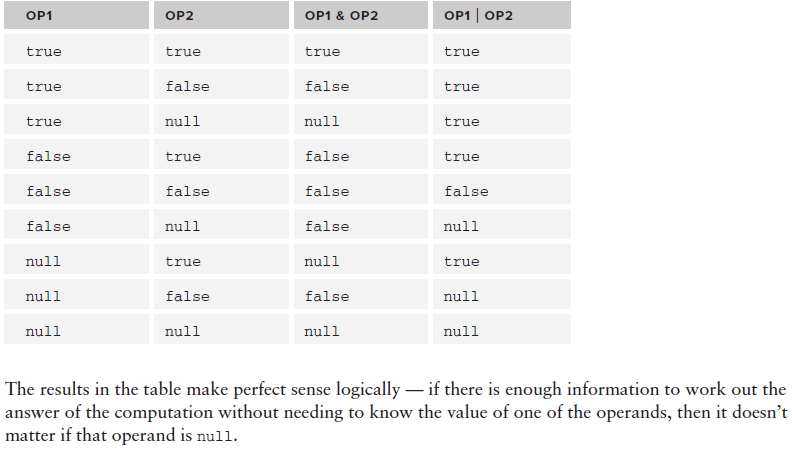
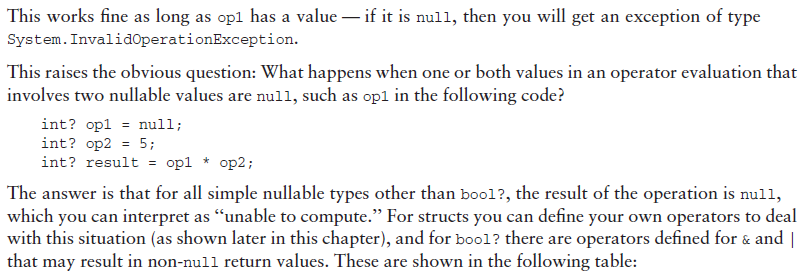
converted to the required type and the appropriate operators are used. This also applies to structs with

operators that you have supplied:

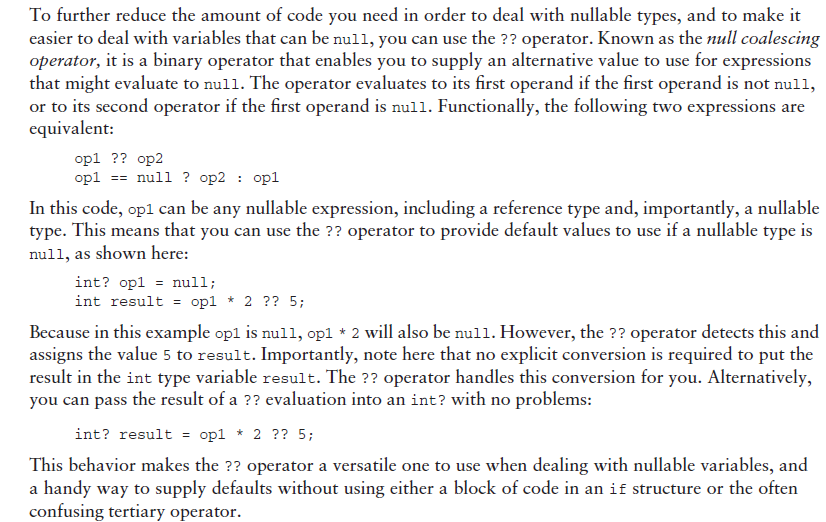
**int? op1 = 5;**

**int? result = op1 \* 2;**

****

****

**?? operator:**

****

List<T>:

Creating a collection of type T objects requires the following code:

List<T> myCollection = new List<T>();

List<int> lists = new List<int> { 3, 4, 5, 6 };

Console.WriteLine(lists[2]);

foreach (int l in lists)

{

Console.WriteLine(l);

}

注意，List<T>在C#中可以用像array的indexer访问，lists[i]

List<T>是ArrayList的generic版本

**LinkedList<T>**

public class LinkedList<T> : ICollection<T>,

IEnumerable<T>, IEnumerable, IReadOnlyCollection<T>, ICollection

可以看到LinkedList<T>并没有实现IList<T>这个接口，说明了LinkedList和List的区别

LinkedList<T> is a general-purpose linked list. It supports enumerators and implements the [ICollection](https://msdn.microsoft.com/en-us/library/system.collections.icollection(v=vs.110).aspx) interface, consistent with other collection classes in the .NET Framework.

LinkedList<T> provides separate nodes of type [LinkedListNode<T>](https://msdn.microsoft.com/en-us/library/ahf4c754(v=vs.110).aspx), so insertion and removal are O(1) operations.

You can remove nodes and reinsert them, either in the same list or in another list, which results in no additional objects allocated on the heap. Because the list also maintains an internal count, getting the [Count](https://msdn.microsoft.com/en-us/library/t56y0h99(v=vs.110).aspx) property is an O(1) operation.

Each node in a LinkedList<T> object is of the type [LinkedListNode<T>](https://msdn.microsoft.com/en-us/library/ahf4c754(v=vs.110).aspx). Because the LinkedList<T> is doubly linked, each node points forward to the [Next](https://msdn.microsoft.com/en-us/library/b4t9w6be(v=vs.110).aspx) node and backward to the [Previous](https://msdn.microsoft.com/en-us/library/de80z2x6(v=vs.110).aspx) node.

Lists that contain reference types perform better when a node and its value are created at the same

time. LinkedList<T> accepts **null** as a valid [Value](https://msdn.microsoft.com/en-us/library/86397t7k(v=vs.110).aspx) property for reference types and allows duplicate values.

If the LinkedList<T> is empty, the [First](https://msdn.microsoft.com/en-us/library/ms132187(v=vs.110).aspx) and [Last](https://msdn.microsoft.com/en-us/library/ms132188(v=vs.110).aspx) properties contain **null**.

The LinkedList<T> class does not support chaining, splitting, cycles, or other features that can leave the list in an inconsistent state. The list remains consistent on a single thread. The only multithreaded scenario supported by LinkedList<T> is multithreaded read operations.

下面一些使用LinkedList的例子：

string[] words =

{ "the", "fox", "jumped", "over", "the", "dog" };

LinkedList<string> sentence = new LinkedList<string>(words);

Display(sentence, "The linked list values:");

Console.WriteLine("sentence.Contains(\"jumped\") = {0}",

sentence.Contains("jumped"));

// Add the word 'today' to the beginning of the linked list.

sentence.AddFirst("today");

Display(sentence, "Test 1: Add 'today' to beginning of the list:"); //today the fox jumped over the dog

// Move the first node to be the last node.

LinkedListNode<string> mark1 = sentence.First;

sentence.RemoveFirst();

sentence.AddLast(mark1);

Display(sentence, "Test 2: Move first node to be last node:"); //the fox jumped over the dog today

// Change the last node be 'yesterday'.

sentence.RemoveLast();

sentence.AddLast("yesterday");

Display(sentence, "Test 3: Change the last node to 'yesterday':"); //the fox jumped over the dog yesterday

// Move the last node to be the first node.

mark1 = sentence.Last;

sentence.RemoveLast();

sentence.AddFirst(mark1);

Display(sentence, "Test 4: Move last node to be first node:"); //yesterday the fox jumped over the dog

// Indicate, by using parentheisis, the last occurence of 'the'.

sentence.RemoveFirst();

LinkedListNode<string> current = sentence.FindLast("the");

IndicateNode(current, "Test 5: Indicate last occurence of 'the':"); //the fox jumped over (the) dog

// Add 'lazy' and 'old' after 'the' (the LinkedListNode named current).

sentence.AddAfter(current, "old");

sentence.AddAfter(current, "lazy");

IndicateNode(current, "Test 6: Add 'lazy' and 'old' after 'the':"); //the fox jumped over (the) lazy old dog

// Indicate 'fox' node.

current = sentence.Find("fox");

IndicateNode(current, "Test 7: Indicate the 'fox' node:"); //the (fox) jumped over the lazy old dog

// Add 'quick' and 'brown' before 'fox':

sentence.AddBefore(current, "quick");

sentence.AddBefore(current, "brown");

IndicateNode(current, "Test 8: Add 'quick' and 'brown' before 'fox':"); //the quick brown (fox) jumped over the lazy old dog

// Keep a reference to the current node, 'fox',

// and to the previous node in the list. Indicate the 'dog' node.

mark1 = current;

LinkedListNode<string> mark2 = current.Previous;

current = sentence.Find("dog");

IndicateNode(current, "Test 9: Indicate the 'dog' node:"); //the quick brown fox jumped over the lazy old (dog)

// The AddBefore method throws an InvalidOperationException

// if you try to add a node that already belongs to a list.

Console.WriteLine("Test 10: Throw exception by adding node (fox) already in the list:"); //Exception message: The LinkedList node belongs a LinkedList.

try

{

sentence.AddBefore(current, mark1);

}

catch (InvalidOperationException ex)

{

Console.WriteLine("Exception message: {0}", ex.Message);

}

Console.WriteLine();

// Remove the node referred to by mark1, and then add it

// before the node referred to by current.

// Indicate the node referred to by current.

sentence.Remove(mark1);

sentence.AddBefore(current, mark1);

IndicateNode(current, "Test 11: Move a referenced node (fox) before the current node (dog):"); //the quick brown jumped over the lazy old fox (dog)

// Remove the node referred to by current.

sentence.Remove(current);

IndicateNode(current, "Test 12: Remove current node (dog) and attempt to indicate it:"); //Node 'dog' is not in the list.

// Add the node after the node referred to by mark2.

sentence.AddAfter(mark2, current);

IndicateNode(current, "Test 13: Add node removed in test 11 after a referenced node (brown):"); //the quick brown (dog) jumped over the lazy old fox

// The Remove method finds and removes the

// first node that that has the specified value.

sentence.Remove("old");

Display(sentence, "Test 14: Remove node that has the value 'old':"); //the quick brown dog jumped over the lazy fox

// When the linked list is cast to ICollection(Of String),

// the Add method adds a node to the end of the list.

sentence.RemoveLast();

ICollection<string> icoll = sentence;

icoll.Add("rhinoceros");

Display(sentence, "Test 15: Remove last node, cast to ICollection, and add 'rhinoceros':"); //the quick brown dog jumped over the lazy rhinoceros

Console.WriteLine("Test 16: Copy the list to an array:");

// Create an array with the same number of

// elements as the inked list.

string[] sArray = new string[sentence.Count];

sentence.CopyTo(sArray, 0);

foreach (string s in sArray)

{

Console.WriteLine(s);

}

// Release all the nodes.

sentence.Clear();

Console.WriteLine();

Console.WriteLine("Test 17: Clear linked list. Contains 'jumped' = {0}",

sentence.Contains("jumped"));//Test 17: Clear linked list. Contains 'jumped' = False

Console.ReadLine();

}

private static void Display(LinkedList<string> words, string test)

{

Console.WriteLine(test);

foreach (string word in words)

{

Console.Write(word + " ");

}

Console.WriteLine();

Console.WriteLine();

}

private static void IndicateNode(LinkedListNode<string> node, string test)

{

Console.WriteLine(test);

if (node.List == null)

{

Console.WriteLine("Node '{0}' is not in the list.\n",

node.Value);

return;

}

StringBuilder result = new StringBuilder("(" + node.Value + ")");

LinkedListNode<string> nodeP = node.Previous;

while (nodeP != null)

{

result.Insert(0, nodeP.Value + " ");

nodeP = nodeP.Previous;

}

node = node.Next;

while (node != null)

{

result.Append(" " + node.Value);

node = node.Next;

}

Console.WriteLine(result);

Console.WriteLine();

}

注意：一个LinkedListNode remove掉就找不到了，如果想实现换位置，就先让一个reference指向这个Node。

Remove这个方法可以Remove(LinkedListNode<T> someNode)，也可以Remove(T someValue),比如sentence.Remove("old"); remove the first node that that has the specified value old.

## Sorting and Searching Generic Lists, Dictionary<K,V>

Sorting a generic list is much the same as sorting any other list. The last chapter described how you can use the IComparer and IComparable interfaces to compare two objects and thereby sort a list of that type of object. The only difference here is that you can use the generic interfaces IComparer<T> and IComparable<T>, which expose slightly different, type-specific methods.

To sort a List<T>, you can supply an IComparable<T> interface on the type to be sorted, or supply an IComparer<T> interface. Alternatively, you can supply a *generic delegate* as a sorting method. From the perspective of seeing how things are done, this is far more interesting because implementing the interfaces shown above is really no more effort than implementing their nongeneric cousins.

➤ Comparison<T>: A delegate type for a method used for sorting, with the following return type and parameters:

int method(T objectA, T objectB)

public delegate int Comparison<in T>(

T x,

T y

)

This type parameter T is contravariant. That is, you can use either the type you specified or any type that is less derived. For more information about covariance and contravariance

➤ Predicate<T>: A delegate type for a method used for searching, with the following return type and parameters:

bool method(T targetObject)

public delegate bool Predicate<in T>(

T obj

)

You can define any number of such methods, and use them to ‘‘snap-in’’ to the searching and sorting methods of List<T>.

public class Vectors : List<Vector>

{

public Vectors()

{

}

public Vectors(IEnumerable<Vector> initialItems)

{

foreach (Vector vector in initialItems)

{

Add(vector);

}

}

}

public static class VectorDelegates

{

public static int Compare(Vector x, Vector y)

{

if (x.R > y.R)

{

return 1;

}

else if (x.R < y.R)

{

return -1;

}

return 0;

}

public static bool TopRightQuadrant(Vector target)

{

if (target.Theta >= 0.0 && target.Theta <= 90.0)

{

return true;

}

else

{

return false;

}

}

}

class Program

{

static void Main(string[] args)

{

Vectors route = new Vectors();

route.Add(new Vector(2.0, 90.0));

route.Add(new Vector(1.0, 180.0));

route.Add(new Vector(0.5, 45.0));

route.Add(new Vector(2.5, 315.0));

Comparison<Vector> sorter = new Comparison<Vector>(

VectorDelegates.Compare);

route.Sort(sorter);

Predicate<Vector> searcher =

new Predicate<Vector>(VectorDelegates.TopRightQuadrant);

Vectors topRightQuadrantRoute = new Vectors(route.FindAll(searcher));

Console.ReadKey();

}

}

这里需要注意的是：

As an aside to this example, note that the code

Comparison<Vector> sorter = new Comparison<Vector>(VectorDelegates.Compare); route.Sort(sorter);

can be simplified to the following:

route.Sort(VectorDelegates.Compare);

This removes the necessity to implicitly reference the Comparison<Vector> type. In fact, an instance of this type is still created, but it is created implicitly. The Sort() method obviously needs an instance of this type to work, but the compiler realizes this and creates one for you from the method that you supply.

也就是说，在需要delegate的时候，可以直接传入一个能有此delegate代理的方法，注意此时是传的方法的名称，没有后面的括号

**Dictionary<K,V>:**

The Dictionary<K, V> type enables you to define a collection of key-value pairs. Unlike the other generic collection types you’ve looked at in this chapter, this class requires instantiating two types: the types for both the key and the value that represent each item in the collection.

Once a Dictionary<K, V> object is instantiated, you can perform much the same operations on it as you can on a class that inherits from DictionaryBase, but with type-safe methods and properties already in place. You can, for example, add key-value pairs using a strongly typed Add() method:

Dictionary<string, int> things = new Dictionary<string, int>();

things.Add("Green Things", 29);

things.Add("Blue Things", 94);

things.Add("Yellow Things", 34);

things.Add("Red Things", 52);

things.Add("Brown Things", 27);

You can iterate through keys and values in the collection by using the Keys and Values properties:

foreach (string key in things.Keys)

{

Console.WriteLine(key);

}

foreach (int value in things.Values)

{

Console.WriteLine(value);

}

In addition, you can iterate through items in the collection by obtaining each as a KeyValuePair<K, V> instance, much like you can with the DictionaryEntry objects shown in the last chapter:

foreach (KeyValuePair<string, int> thing in things)

{

Console.WriteLine("{0} = {1}", thing.Key, thing.Value);

}

One thing to note about Dictionary<K, V> is that the key for each item must be unique. Attempting to add an item with an identical key to one already added will cause an ArgumentException exception to be thrown. Because of this, Dictionary<K, V> allows you to pass an IComparer<K> interface to its constructor. This may be necessary if you use your own classes as keys and they don’t support an IComparable or IComparable<K> interface, or if you want to compare objects using a nondefault process. For instance, in the preceding example, you could use a case-insensitive method to compare string keys:

Dictionary里面的key不能相同，所以如果用自定义类作为key，程序必须知道怎么比较这个自定义类，所以可以选择让这个自定义类实现IComparable<T>或者IComparer<T>接口，也可以在Dictionary的构造器中传入一个实现了这两个接口其中之一的实例

Dictionary<string, int> things =

new Dictionary<string, int>(StringComparer.CurrentCultureIgnoreCase);

Now you’ll get an exception if you use keys such as this:

things.Add("Green Things", 29);

things.Add("Green things", 94);

You can also pass an initial capacity (with an int) or set of items (with an IDictionary<K, V> interface) to the constructor.

## Define Generic Types, default keyword, constraining types, IEnumerable和IEnumerator的区别，Inheriting from Generic Classes

You’ve now learned enough about generics to create your own. You’ve seen plenty of code involving generic types and have had plenty of practice using generic syntax. This section looks at defining the following:

➤ Generic classes

➤ Generic interfaces

➤ Generic methods

➤ Generic delegates

You’ll also look at the following more advanced techniques for dealing with the issues that come up when defining generic types:

➤ The default keyword

➤ Constraining types

➤ Inheriting from generic classes

➤ Generic operators

**Defining Generic Classes**

To create a generic class, merely include the angle bracket syntax in the class definition:

class MyGenericClass<T> { ... }

Here, T can be any identifier you like, following the usual C# naming rules, such as not starting with a number and so on. Typically, though, you can just use T. A generic class can have any number of type parameters in its definition, separated by commas:

class MyGenericClass<**T1, T2, T3**> { ... }

Once these types are defined, you can use them in the class definition just like any other type. You can use them as types for member variables, return types for members such as properties or methods, and parameter types for method arguments:

class MyGenericClass<T1, T2, T3>

{

**private T1 innerT1Object;**

**public MyGenericClass(T1 item)**

**{**

**innerT1Object = item;**

**}**

**public T1 InnerT1Object**

**{**

**get**

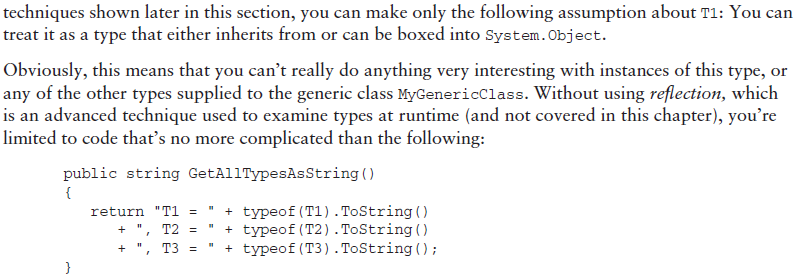
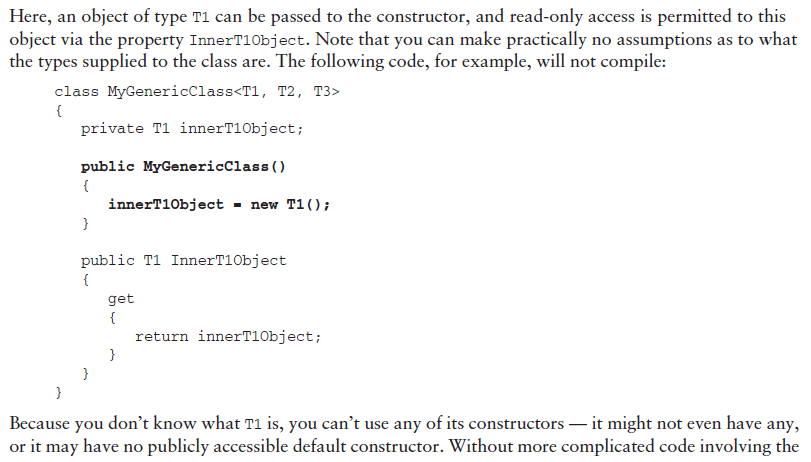
**{**

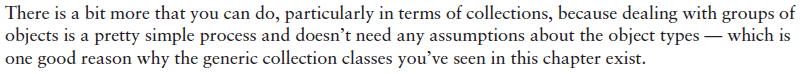
**return innerT1Object;**

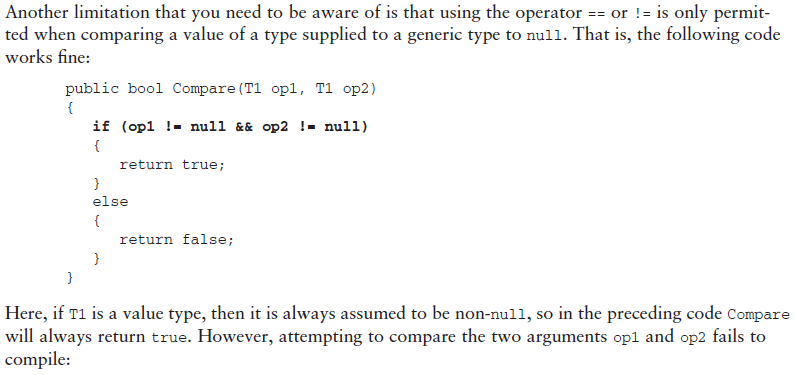
**}**

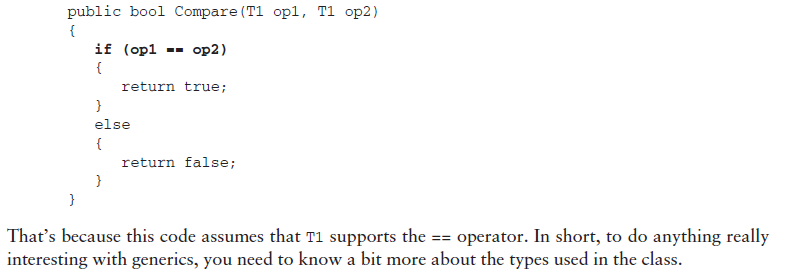
**}**

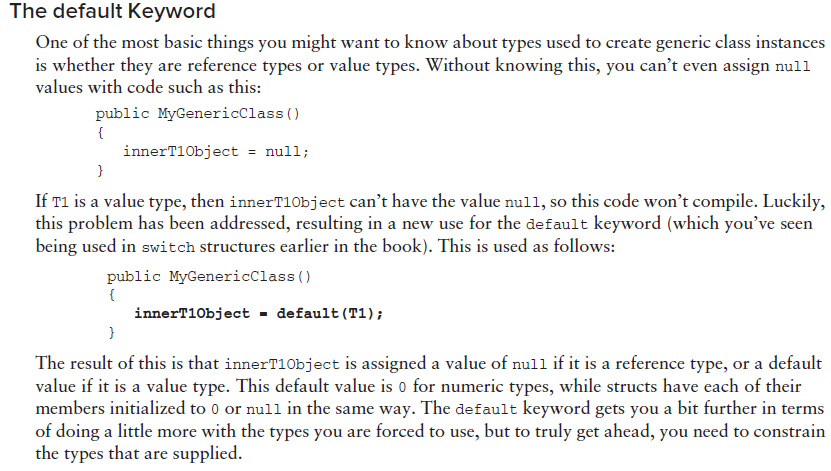
}











**Constraining Types:**

The types you have used with generic classes until now are known as *unbounded* types because no restrictions are placed on what they can be. By *constraining* types, it is possible to restrict the types that can be used to instantiate a generic class. There are a number of ways to do this. For example, it’s possible to restrict a type to one that inherits from a certain type. Referring back to the Animal, Cow, and Chicken classes used earlier, you could restrict a type to one that was or inherited from Animal, so this code would be fine:

MyGenericClass<Cow> = new MyGenericClass<Cow>();

The following, however, would fail to compile:

MyGenericClass<**string**> = new MyGenericClass<**string**>();

In your class definitions this is achieved using the where keyword:

class My GenericClass<T> **where T : *constraint***

{...}

Here, *constraint* defines what the constraint is. You can supply a number of constraints in this way by separating them with commas:

class MyGenericClass<T> where T : ***constraint1*, *constraint2***

{...}

You can define constraints on any or all of the types required by the generic class by using multiple where statements:

class MyGenericClass<**T1, T2**> **where T1 : *constraint1* where T2 : *constraint2***

{...}

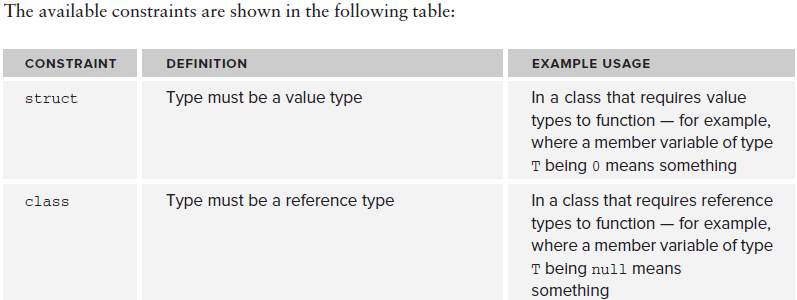
注意，两个where条件之间没有其他符号

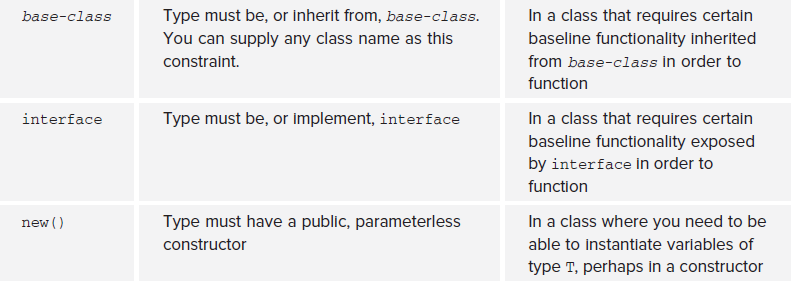
Any constraints that you use must appear after the inheritance specifiers:

class MyGenericClass<T1, T2> **: MyBaseClass, IMyInterface**

where T1 : constraint1 where T2 : constraint2 {...}

where条件要放在继承后面，且没有任何符号





It is possible to use one type parameter as a constraint on another through the base-class constraint as follows:

class MyGenericClass<T1, T2> **where T2 : T1**

{...}

Here, T2 must be the same type as T1 or inherit from T1. This is known as a *naked type constraint*, meaning that one generic type parameter is used as a constraint on another.

Circular type constraints, as shown here, are forbidden:

class MyGenericClass<T1, T2> where T2 : T1 **where T1 : T2**

public class Farm<T> : IEnumerable<T> where T : Animal

{

private List<T> animals = new List<T>();

public List<T> Animals

{

get

{

return animals;

}

}

public IEnumerator<T> GetEnumerator()

{

return animals.GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return animals.GetEnumerator();

}

public void MakeNoises()

{

foreach (T animal in animals)

{

animal.MakeANoise();

}

}

public void FeedTheAnimals()

{

foreach (T animal in animals)

{

animal.Feed();

}

}

public Farm<Cow> GetCows()

{

Farm<Cow> cowFarm = new Farm<Cow>();

foreach (T animal in animals)

{

if (animal is Cow)

{

cowFarm.Animals.Add(animal as Cow);

}

}

return cowFarm;

}

static void Main(string[] args)

{

Farm<Animal> farm = new Farm<Animal>();

farm.Animals.Add(new Cow("Jack"));

farm.Animals.Add(new Chicken("Vera"));

farm.Animals.Add(new Chicken("Sally"));

farm.Animals.Add(new SuperCow("Kevin"));

farm.MakeNoises();

Farm<Cow> dairyFarm = farm.GetCows();

dairyFarm.FeedTheAnimals();

foreach (Cow cow in dairyFarm)

{

if (cow is SuperCow)

{

(cow as SuperCow).Fly();

}

}

Console.ReadKey();

}

}

The GetCows() method is interesting. This method simply extracts all the items in the collection that are of type Cow (or that inherit from Cow, such as the new SuperCow class):

public Farm<Cow> GetCows()

{

Farm<Cow> cowFarm = new Farm<Cow>();

foreach (T animal in animals)

{

if (animal is Cow)

{

cowFarm.Animals.Add(animal as Cow);

}

}

return cowFarm;

}

What is interesting here is that this method seems a bit wasteful. If you wanted other methods of the same sort, such as GetChickens() and so on, you’d need to implement them explicitly too. In a system with many more types, you’d need many more methods. A far better solution here would be to use a *generic* *method,* which you’ll implement a little later in the chapter.

**IEnumerable和IEnumerator的区别：**

先来看定义

http://www.cnblogs.com/Images/OutliningIndicators/None.gifpublic interface IEnumerable  
 2http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
 3http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    IEnumerator GetEnumerator();  
 4http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}  
 5http://www.cnblogs.com/Images/OutliningIndicators/None.gif   
 6http://www.cnblogs.com/Images/OutliningIndicators/None.gifpublic interface IEnumerator  
 7http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
 8http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    bool MoveNext();  
 9http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    void Reset();  
10http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif   
11http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    Object Current { get; }  
12http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

IEnumrable用来暴露一个IEnumerator对象，它只包含一个简单的GetEnumerator。

IEnumerator接口包含了遍历集合的方法。

　　1、一个Collection要支持foreach方式的遍历，必须实现IEnumerable接口（亦即，必须以某种方式返回IEnumerator object）。   
　　2、IEnumerator object具体实现了iterator（通过MoveNext()，Reset()，Current）。   
　　3、从这两个接口的用词选择上，也可以看出其不同：IEnumerable是一个声明式的接口，声明实现该接口的class是“可枚举（enumerable）”的，但并没有说明如何实现枚举器（iterator）；IEnumerator是一个实现式的接口，IEnumerator object就是一个iterator。   
　　4、IEnumerable和IEnumerator通过IEnumerable的GetEnumerator()方法建立了连接，client可以通过IEnumerable的GetEnumerator()得到IEnumerator object，在这个意义上，将GetEnumerator()看作IEnumerator object的factory method也未尝不可。  
　　IEnumerator   是所有枚举数的基接口。

来看一个列子：

public class Person  
 4http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
 5http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public Person(string fName, string lName)  
 6http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
 7http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        this.firstName = fName;  
 8http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        this.lastName = lName;  
 9http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
10http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public string firstName;  
11http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public string lastName;  
12http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}  
13http://www.cnblogs.com/Images/OutliningIndicators/None.gifpublic class People : IEnumerable  
14http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
15http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    private Person[] \_people;  
16http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public People(Person[] pArray)  
17http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
18http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        \_people = new Person[pArray.Length];  
19http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        for (int i = 0; i < pArray.Length; i++)  
20http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
21http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            \_people[i] = pArray[i];  
22http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
23http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
24http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public IEnumerator GetEnumerator()  
25http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
26http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return new PeopleEnum(\_people);  
27http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
28http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}  
29http://www.cnblogs.com/Images/OutliningIndicators/None.gifpublic class PeopleEnum : IEnumerator  
30http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
31http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public Person[] \_people;  
32http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    // Enumerators are positioned before the first element   
33http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    // until the first MoveNext() call.   
34http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    int position = -1;  
35http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public PeopleEnum(Person[] list)  
36http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
37http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        \_people = list;  
38http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
39http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public bool MoveNext()  
40http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
41http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        position++;  
42http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return (position < \_people.Length);  
43http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
44http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public void Reset()  
45http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
46http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        position = -1;  
47http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
48http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    public object Current  
49http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
50http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        get  
51http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
52http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            try  
53http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif            {  
54http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif                return \_people[position];  
55http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif            }  
56http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            catch (IndexOutOfRangeException)  
57http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif            {  
58http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif                throw new InvalidOperationException();  
59http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif            }  
60http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
61http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
62http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}  
63http://www.cnblogs.com/Images/OutliningIndicators/None.gifclass App  
64http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
65http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif    static void Main()  
66http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
67http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        Person[] peopleArray = new Person[3]  
68http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif        {  
69http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            new Person("John", "Smith"),  
70http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            new Person("Jim", "Johnson"),  
71http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            new Person("Sue", "Rabon"),  
72http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        };  
73http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        People peopleList = new People(peopleArray);  
74http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        foreach (Person p in peopleList)  
75http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif            Console.WriteLine(p.firstName + " " + p.lastName);  
76http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
77http://www.cnblogs.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

观察上面的程序：public IEnumerator GetEnumerator()  
25http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockStart.gif    {  
26http://www.cnblogs.com/Images/OutliningIndicators/InBlock.gif        return new PeopleEnum(\_people);  
27http://www.cnblogs.com/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }

返回了一个PeopleEnum实例，此类必定实现了IEnumerator接口，并且实现了该接口的方法。这个程序有一点可以改进的地方，PeopleEnum这个类需要用到People类的数据，而且PeopleEnum又只会在People类中被使用，所以完全可以把PeopleEnum定义成内部类

再来看一个列子：

1. **public** **class** Garage:IEnumerable
2. {
3. Car[] carArray = **new** Car[4];
5. //启动时填充一些Car对象
6. **public** Garage()
7. {
8. carArray[0] = **new** Car("Rusty", 30);
9. carArray[1] = **new** Car("Clunker", 50);
10. carArray[2] = **new** Car("Zippy", 30);
11. carArray[3] = **new** Car("Fred", 45);
12. }
13. **public** IEnumerator GetEnumerator()
14. {
15. **return** **this**.carArray.GetEnumerator();
16. }
17. }

这里并没有手动自己实现一个继承了IEnumerator的类，而是直接使用了carArray的IEnumerator实现，因为carArray实现了IEnumerable，所以它肯定有GetEnumerator方法。

**Inheriting from Generic Classes：**

在c#中定义一个generic类继承于另一个generic类，不需要显式的指定T的类型。比如

class SuperFarm<T> : Farm<T> where T : SuperCow

{…}

这里定义一个SuperFarm generic类，继承于Farm generic类，Farm generic的类的constraint也会应用在这个子类里，比如这里Farm的T必须是Animal或者Animal的子类。所以SuperFarm的T默认也是Animal或者Animal的子类。SuperFarm可以unconstraint这个限制，然而它可以定义一个自己的限制，这个限制必须遵守一定的规则，如这个限制必须至少和原限制一样，也就说上面的程序where T: SuperCow是可以的，因为SuperCow是Animal的子类。

Also note that if you inherit from a generic type, then you must supply all the required type information, either in the form of other generic type parameters, as shown above, or explicitly. This also applies to nongeneric classes that inherit from generic types, as you’ve seen elsewhere. Here’s an example:

public class Cards : List<Card>, ICloneable

{ }

This is fine, but attempting the following will fail:

public class Cards : List<T>, ICloneable

{ }

Here, no information is supplied for T, so no compilation is possible.

## Generic interface, Defining Generic Methods

**Defining Generic Interfaces:**

You’ve now seen several generic interfaces in use — namely, those in the Systems.Collections. Generic namespace such as IEnumerable<T> used in the last example. Defining a generic interface involves the same techniques as defining a generic class:

interface MyFarmingInterface<T>

where T : Animal

{

bool AttemptToBreed(T animal1, T animal2);

T OldestInHerd { get; }

}

Here, the generic parameter T is used as the type of the two arguments of AttemptToBreed() and the type of the OldestInHerd property.

The same inheritance rules apply as for classes. If you inherit from a base generic interface, you must obey the rules, such as keeping the constraints of the base interface generic type parameters.

**Defining Generic Methods:**

The last chaptor used a method called GetCows(), and in the discussion of the example it was stated that you could make a more general form of this method using a *generic method*. In this section you’ll see how this is possible. A generic method is one in which the return and/or parameter types are determined by a generic type parameter or parameters:

public T GetDefault<T>()

{

return default(T);

}

This trivial example uses the default keyword you looked at earlier in the chapter to return a default value for a type T. This method is called as follows:

int myDefaultInt = GetDefault<int>();

The type parameter T is provided at the time the method is called.

This T is quite separate from the types used to supply generic type parameters to classes. In fact, generic methods can be implemented by nongeneric classes:

If the class is generic, though, then you must use different identifiers for generic method types. The following code won’t compile:

public class Defaulter**<T>**

{

public T GetDefault<T>()

{

return default(T);

}

}

The type T used by either the method or the class must be renamed. Constraints can be used by generic method parameters in the same way that they are for classes, and in this case you can make use of any class type parameters:

public class Defaulter<**T1**>

{

public **T2** GetDefault<**T2**>() **where T2 : T1**

{

return default(**T2**);

}

}

Here, the type T2 supplied to the method must be the same as, or inherit from, T1 supplied to the class. This is a common way to constrain generic methods.

In the Farm<T> class shown earlier, you could include the following method

public Farm<U> GetSpecies<U>() where U : T

{

Farm<U> speciesFarm = new Farm<U>();

foreach (T animal in animals)

{

if (animal is U)

{

speciesFarm.Animals.Add(animal as U);

}

}

return speciesFarm;

}

This can replace GetCows() and any other methods of the same type. The generic type parameter used here, U, is constrained by T, which is in turn constrained by the Farm<T> class to Animal. This enables you to treat instances of T as instances of Animal, should you wish to do so. In the client code for previous captor, in Program.cs, using this new method requires one modification:

Farm<Cow> dairyFarm = farm.**GetSpecies<Cow>()**;

You could equally write

Farm<Chicken> poultryFarm = farm.GetSpecies<Chicken>();

## Indexer，using indexer，indexer in interface，Difference between Property and Indexer

Indexers allow instances of a **class** , **struct** or **interface** to be indexed just like arrays. Indexers resemble [properties](https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx) except that their accessors take parameters.

In the following example, a generic class is defined and provided with simple [get](https://msdn.microsoft.com/en-us/library/ms228503.aspx) and [set](https://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor methods as a means of assigning and retrieving values. The Program class creates an instance of this class for storing strings.

class SampleCollection<T>

{

// Declare an array to store the data elements.

private T[] arr = new T[100];

// Define the indexer, which will allow client code

// to use [] notation on the class instance itself.

// (See line 2 of code in Main below.)

public T this[int i]

{

get

{

// This indexer is very simple, and just returns or sets

// the corresponding element from the internal array.

return arr[i];

}

set

{

arr[i] = value;

}

}

}

// This class shows how client code uses the indexer.

class Program

{

static void Main(string[] args)

{

// Declare an instance of the SampleCollection type.

SampleCollection<string> stringCollection = new SampleCollection<string>();

// Use [] notation on the type.

stringCollection[0] = "Hello, World";

System.Console.WriteLine(stringCollection[0]);

}

}

// Output:

// Hello, World.

* Indexers enable objects to be indexed in a similar manner to arrays.
* A **get** accessor returns a value. A **set** accessor assigns a value.
* The [this](https://msdn.microsoft.com/en-us/library/dk1507sz.aspx) keyword is used to define the indexers.
* The [value](https://msdn.microsoft.com/en-us/library/a1khb4f8.aspx) keyword is used to define the value being assigned by the **set** indexer.
* Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
* Indexers can be overloaded.
* Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Using indexer

 Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array. For example, suppose you have a class named TempRecord that represents the temperature in Farenheit as recorded at 10 different times during a 24 hour period. The class contains an array named "temps" of type float to represent the temperatures, and a [DateTime](https://msdn.microsoft.com/en-us/library/system.datetime.aspx) that represents the date the temperatures were recorded. By implementing an indexer in this class, clients can access the temperatures in a TempRecord instance as float temp = tr[4] instead of as float temp = tr.temps[4]. The indexer notation not only simplifies the syntax for client applications; it also makes the class and its purpose more intuitive for other developers to understand.

To declare an indexer on a class or struct, use the [this](https://msdn.microsoft.com/en-us/library/dk1507sz.aspx) keyword, as in this example:

public int this[int index] // Indexer declaration

{

// get and set accessors

}

The following example shows how to declare a private array field, temps, and an indexer. The indexer enables direct access to the instance tempRecord[i]. The alternative to using the indexer is to declare the array as a [public](https://msdn.microsoft.com/en-us/library/yzh058ae.aspx) member and access its members, tempRecord.temps[i], directly.

Notice that when an indexer's access is evaluated, for example, in a **Console.Write** statement, the [get](https://msdn.microsoft.com/en-us/library/ms228503.aspx) accessor is invoked. Therefore, if no **get** accessor exists, a compile-time error occurs.

当你在程序中使用instance[i]的时候，其实就是调用了这个public int this[int index] 的get accessor。所以没有get accessor的话会出现编译错误

下面的关于温度的例子

class TempRecord

{

// Array of temperature values

private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,

61.3F, 65.9F, 62.1F, 59.2F, 57.5F };

// To enable client code to validate input

// when accessing your indexer.

public int Length

{

get { return temps.Length; }

}

// Indexer declaration.

// If index is out of range, the temps array will throw the exception.

public float this[int index]

{

get

{

return temps[index];

}

set

{

temps[index] = value;

}

}

}

class MainClass

{

static void Main()

{

TempRecord tempRecord = new TempRecord();

// Use the indexer's set accessor

tempRecord[3] = 58.3F;

tempRecord[5] = 60.1F;

// Use the indexer's get accessor

for (int i = 0; i < 10; i++)

{

System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);

}

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

C# does not limit the index type to integer. For example, it may be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string in the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can co-exist.

In this example, a class is declared that stores the days of the week. A **get** accessor is declared that takes a string, the name of a day, and returns the corresponding integer. For example, Sunday will return 0, Monday will return 1, and so on.

// Using a string as an indexer value

class DayCollection

{

string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

// This method finds the day or returns -1

private int GetDay(string testDay)

{

for (int j = 0; j < days.Length; j++)

{

if (days[j] == testDay)

{

return j;

}

}

throw new System.ArgumentOutOfRangeException(testDay, "testDay must be in the form \"Sun\", \"Mon\", etc");

}

// The get accessor returns an integer for a given string

public int this[string day]

{

get

{

return (GetDay(day));

}

}

}

class Program

{

static void Main(string[] args)

{

DayCollection week = new DayCollection();

System.Console.WriteLine(week["Fri"]);

// Raises ArgumentOutOfRangeException

System.Console.WriteLine(week["Made-up Day"]);

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

Indexer in interface

Indexers can be declared on an [interface (C# Reference)](https://msdn.microsoft.com/en-us/library/87d83y5b.aspx). Accessors of interface indexers differ from the accessors of [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) indexers in the following ways:

* Interface accessors do not use modifiers.
* An interface accessor does not have a body.

Thus, the purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only.

The following is an example of an interface indexer accessor:

public interface ISomeInterface

{

//...

// Indexer declaration:

string this[int index]

{

get;

set;

}

}

在接口中只给出indexer的定义，它是只有get方法，set方法，还是两个方法都有。具体的实现由实现了这个接口的class完成，如下面所示：

public interface ISomeInterface

{

// Indexer declaration:

int this[int index]

{

get;

set;

}

}

// Implementing the interface.

class IndexerClass : ISomeInterface

{

private int[] arr = new int[100];

public int this[int index] // indexer declaration

{

get

{

// The arr object will throw IndexOutOfRange exception.

return arr[index];

}

set

{

arr[index] = value;

}

}

}

In the preceding example, you could use the explicit interface member implementation by using the fully qualified name of the interface member. For example:

public int ISomeInterface.this

{

}

However, the fully qualified name is only needed to avoid ambiguity when the class is implementing more than one interface with the same indexer signature. For example, if anEmployee class is implementing two interfaces, ICitizen and IEmployee, and both interfaces have the same indexer signature, the explicit interface member implementation is necessary. That is, the following indexer declaration:

public int IEmployee.this

{

}

implements the indexer on the IEmployee interface, while the following declaration:

public int ICitizen.this

{

}

在java中没有解决的问题：一个class实现了两个接口，而这两个接口有一个一模一样的方法签名，那么怎么区分class里面覆写的是哪一个呢，这时候，在C#中可以用fully qualified name来区分，比如public int Interface1.GetInfo(){…}和Interface2.GetInfo(){…}。在调用方法的时候，利用多态，就能很清楚的知道调用的是哪个方法。

注意：

A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface.

Difference between Property and Indexer:

Indexers are like properties. Except for the differences shown in the following table, all the rules that are defined for property accessors apply to indexer accessors also.

|  |  |
| --- | --- |
| **Property** | **Indexer** |
| Allows methods to be called as if they were public data members. | Allows elements of an internal collection of an object to be accessed by  using array notation on the object itself. |
| Accessed through a simple name. | Accessed through an index. |
| Can be a static or an instance member. | Must be an instance member. |
| A [get](https://msdn.microsoft.com/en-us/library/ms228503.aspx) accessor of a property has no parameters. | A **get** accessor of an indexer has the same formal parameter list as the indexer. |
| A [set](https://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor of a property contains the implicit **value** parameter. | A **set** accessor of an indexer has the same formal parameter list as the  indexer, and also to the[value](https://msdn.microsoft.com/en-us/library/a1khb4f8.aspx) parameter. |
| Supports shortened syntax with [Auto-Implemented Properties (C# Programming Guide)](https://msdn.microsoft.com/en-us/library/bb384054.aspx). | Does not support shortened syntax. |

## Property，Using Property，interface Property，Restricting Accessor Accessibility，new as modifier，auto-implemented property

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

In this example, the TimePeriod class stores a time period. Internally the class stores the time in seconds, but a property named Hours enables a client to specify a time in hours. The accessors for the Hours property perform the conversion between hours and seconds.

class TimePeriod

{

private double seconds;

public double Hours

{

get { return seconds / 3600; }

set { seconds = value \* 3600; }

}

}

class Program

{

static void Main()

{

TimePeriod t = new TimePeriod();

// Assigning the Hours property causes the 'set' accessor to be called.

t.Hours = 24;

// Evaluating the Hours property causes the 'get' accessor to be called.

System.Console.WriteLine("Time in hours: " + t.Hours);

}

}

// Output: Time in hours: 24

这里Hours像一个方法，但是却像一个data member一样被使用。在内存中Hours也是像方法一样储存

* Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
* A [get](https://msdn.microsoft.com/en-us/library/ms228503.aspx) property accessor is used to return the property value, and a [set](https://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor is used to assign a new value. These accessors can have different access levels. For more information, see [Restricting Accessor Accessibility (C# Programming Guide)](https://msdn.microsoft.com/en-us/library/75e8y5dd.aspx).
* Unlike fields, properties are not classified as variables. Therefore, you cannot pass a property as a [ref (C# Reference)](https://msdn.microsoft.com/en-us/library/14akc2c7.aspx) or [out (C# Reference)](https://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameter.
* The [value](https://msdn.microsoft.com/en-us/library/a1khb4f8.aspx) keyword is used to define the value being assigned by the **set** accessor.
* Properties that do not implement a **set** accessor are read only.
* For simple properties that require no custom accessor code, consider the option of using auto-implemented properties. For more information, see [Auto-Implemented Properties (C# Programming Guide)](https://msdn.microsoft.com/en-us/library/bb384054.aspx).

Using Property

Properties have many uses: they can validate data before allowing a change; they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database; they can take an action when data is changed, such as raising an event, or changing the value of other fields.

public class Date

{

private int month = 7; // Backing store

public int Month

{

get

{

return month;

}

set

{

if ((value > 0) && (value < 13))

{

month = value;

}

}

}

}

In this example, Month is declared as a property so that the **set** accessor can make sure that the Month value is set between 1 and 12. The Month property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property.

When you reference the property, except as the target of an assignment, the **get** accessor is invoked to read the value of the property. For example:

Person person = new Person();

//...

System.Console.Write(person.Name); // the get accessor is invoked here

It is a bad programming style to change the state of the object by using the **get** accessor. For example, the following accessor produces the side effect of changing the state of the object every time that the number field is accessed.

private int number;

public int Number

{

get

{

return number++; // Don't do this

}

}

The **get** accessor can be used to return the field value or to compute it and return it. For example:

class Employee

{

private string name;

public string Name

{

get

{

return name != null ? name : "NA";

}

}

}

When you assign a value to the property, the **set** accessor is invoked by using an argument that provides the new value. For example:

Person person = new Person();

person.Name = "Joe"; // the set accessor is invoked here

System.Console.Write(person.Name); // the get accessor is invoked here

注意：

A property may be declared as a static property by using the **static** keyword. This makes the property available to callers at any time, even if no instance of the class exists. For more information

A property may be marked as a virtual property by using the [virtual](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) keyword. This enables derived classes to override the property behavior by using the [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) keyword.

A property overriding a virtual property can also be [sealed](https://msdn.microsoft.com/en-us/library/88c54tsw.aspx), specifying that for derived classes it is no longer virtual. Lastly, a property can be declared [abstract](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx). This means that there is no implementation in the class, and derived classes must write their own implementation.

Interface Property:

Properties can be declared on an [interface (C# Reference)](https://msdn.microsoft.com/en-us/library/87d83y5b.aspx). The following is an example of an interface property

public interface ISampleInterface

{

// Property declaration:

string Name

{

get;

set;

}

}

 Interface members are automatically public, and they can't include any access modifiers.

The accessor of an interface property does not have a body. Thus, the purpose of the accessors is to indicate whether the property is read-write, read-only, or write-only.

In this example, the interface IEmployee has a read-write property, Name, and a read-only property, Counter. The class Employee implements the IEmployee interface and uses these two properties. The program reads the name of a new employee and the current number of employees and displays the employee name and the computed employee number.

interface IEmployee

{

string Name

{

get;

set;

}

int Counter

{

get;

}

}

public class Employee : IEmployee

{

public static int numberOfEmployees;

private string name;

public string Name // read-write instance property

{

get

{

return name;

}

set

{

name = value;

}

}

private int counter;

public int Counter // read-only instance property

{

get

{

return counter;

}

}

public Employee() // constructor

{

counter = ++counter + numberOfEmployees;

}

}

class TestEmployee

{

static void Main()

{

System.Console.Write("Enter number of employees: ");

Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();

System.Console.Write("Enter the name of the new employee: ");

e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");

System.Console.WriteLine("Employee number: {0}", e1.Counter);

System.Console.WriteLine("Employee name: {0}", e1.Name);

}

}

注意，在interface里面也不用注明是virtual，可以在class中直接覆写,default就是public virtual

Restricting Accessor Accessibility：

The [get](https://msdn.microsoft.com/en-us/library/ms228503.aspx) and [set](https://msdn.microsoft.com/en-us/library/ms228368.aspx) portions of a property or indexer are called *accessors*. By default these accessors have the same visibility, or access level: that of the property or indexer to which they belong.

However, it is sometimes useful to restrict access to one of these accessors. Typically, this involves restricting the accessibility of the **set** accessor, while keeping the **get** accessor publicly accessible. For example:

private string name = "Hello";

public string Name

{

get

{

return name;

}

protected set

{

name = value;

}

}

In this example, a property called Name defines a **get** and **set** accessor. The **get** accessor receives the accessibility level of the property itself, **public** in this case, while the **set** accessor is explicitly restricted by applying the [protected](https://msdn.microsoft.com/en-us/library/bcd5672a.aspx) access modifier to the accessor itself.

Using the accessor modifiers on properties or indexers is subject to these conditions:

* You cannot use accessor modifiers on an interface or an explicit [interface](https://msdn.microsoft.com/en-us/library/87d83y5b.aspx) member implementation.
* You can use accessor modifiers only if the property or indexer has both **set** and **get** accessors. In this case, the modifier is permitted on one only of the two accessors.
* If the property or indexer has an [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) modifier, the accessor modifier must match the accessor of the overridden accessor, if any.
* The accessibility level on the accessor must be more restrictive than the accessibility level on the property or indexer itself.

When you override a property or indexer, the overridden accessors must be accessible to the overriding code. Also, the accessibility level of both the property/indexer, and that of the accessors must match the corresponding overridden property/indexer and the accessors. For example:

public class Parent

{

public virtual int TestProperty

{

// Notice the accessor accessibility level.

protected set { }

// No access modifier is used here.

get { return 0; }

}

}

public class Kid : Parent

{

public override int TestProperty

{

// Use the same accessibility level as in the overridden accessor.

protected set { }

// Cannot use access modifier here.

get { return 0; }

}

}

When you use an accessor to implement an interface, the accessor may not have an access modifier. However, if you implement the interface using one accessor, such as **get**, the other accessor can have an access modifier, as in the following example:

public interface ISomeInterface

{

int TestProperty

{

// No access modifier allowed here

// because this is an interface.

get;

}

}

public class TestClass : ISomeInterface

{

public int TestProperty

{

// Cannot use access modifier here because

// this is an interface implementation.

get { return 10; }

// Interface property does not have set accessor,

// so access modifier is allowed.

protected set { }

}

}

记住，property像方法一样，accessor也就是这一个特殊的方法，所以尽管在interface里没有set，在class里也可以自己有一个set accessor。

If you use an access modifier on the accessor, the [accessibility domain](https://msdn.microsoft.com/en-us/library/zd74a9ww.aspx) of the accessor is determined by this modifier.

If you did not use an access modifier on the accessor, the accessibility domain of the accessor is determined by the accessibility level of the property or indexer.

Accessor的 accessibility和property的发生冲突的情况，就像上面的程序，property是public的，但是set却是protected，所以下面的代码会产生错误

public class Program

{

static void main(string[] args)

{

TestClass test = new TestClass();

test.TestProperty = 20;//error, can not access set method

Console.WriteLine(test.TestProperty);

}

}

The following example contains three classes, BaseClass, DerivedClass, and MainClass. There are two properties on the BaseClass, Name and Id on both classes. The example demonstrates how the property Id on DerivedClass can be hidden by the property Id on BaseClass when you use a restrictive access modifier such as [protected](https://msdn.microsoft.com/en-us/library/bcd5672a.aspx) or[private](https://msdn.microsoft.com/en-us/library/st6sy9xe.aspx). Therefore, when you assign values to this property, the property on the BaseClass class is called instead. Replacing the access modifier by [public](https://msdn.microsoft.com/en-us/library/yzh058ae.aspx) will make the property accessible.

The example also demonstrates that a restrictive access modifier, such as **private** or **protected**, on the **set** accessor of the Name property in DerivedClass prevents access to the accessor and generates an error when you assign to it.

public class BaseClass

{

private string name = "Name-BaseClass";

private string id = "ID-BaseClass";

public string Name

{

get { return name; }

set { }

}

public string Id

{

get { return id; }

set { }

}

}

public class DerivedClass : BaseClass

{

private string name = "Name-DerivedClass";

private string id = "ID-DerivedClass";

new public string Name

{

get

{

return name;

}

// Using "protected" would make the set accessor not accessible.

set

{

name = value;

}

}

// Using private on the following property hides it in the Main Class.

// Any assignment to the property will use Id in BaseClass.

new private string Id

{

get

{

return id;

}

set

{

id = value;

}

}

}

class MainClass

{

static void Main()

{

BaseClass b1 = new BaseClass();

DerivedClass d1 = new DerivedClass();

b1.Name = "Mary";

d1.Name = "John";

b1.Id = "Mary123";

d1.Id = "John123"; // The BaseClass.Id property is called.

System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);

System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

/\* Output:

Base: Name-BaseClass, ID-BaseClass

Derived: John, ID-BaseClass

\*/

new 关键字在这里的用法将在下一节介绍，这里derived class里有一个和base class一样的属性ID,但是在derived class里，这个属性是private，那么就不能从外部访问，所以在main中访问d1的ID时，实际上会访问被隐藏的base class的ID属性

New as modifier:

In C#, a method in a derived class can have the same name as a method in the base class. You can specify how the methods interact by using the [new](https://msdn.microsoft.com/en-us/library/51y09td4.aspx) and [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) keywords. The**override** modifier *extends* the base class method, and the **new** modifier *hides* it. The difference is illustrated in the examples in this topic.

In a console application, declare the following two classes, BaseClass and DerivedClass. DerivedClass inherits from BaseClass.

class BaseClass

{

public void Method1()

{

Console.WriteLine("Base - Method1");

}

}

class DerivedClass : BaseClass

{

public void Method2()

{

Console.WriteLine("Derived - Method2");

}

}

In the Main method, declare variables bc, dc, and bcdc.

* bc is of type BaseClass, and its value is of type BaseClass.
* dc is of type DerivedClass, and its value is of type DerivedClass.
* bcdc is of type BaseClass, and its value is of type DerivedClass. This is the variable to pay attention to.

Because bc and bcdc have type BaseClass, they can only directly access Method1, unless you use casting. Variable dc can access both Method1 and Method2. These relationships are shown in the following code.

class Program

{

static void Main(string[] args)

{

BaseClass bc = new BaseClass();

DerivedClass dc = new DerivedClass();

BaseClass bcdc = new DerivedClass();

bc.Method1();

dc.Method1();

dc.Method2();

bcdc.Method1();

}

// Output:

// Base - Method1

// Base - Method1

// Derived - Method2

// Base - Method1

}

Next, add the following Method2 method to BaseClass. The signature of this method matches the signature of the Method2 method in DerivedClass.

public void Method2()

{

Console.WriteLine("Base - Method2");

}

Because BaseClass now has a Method2 method, a second calling statement can be added for BaseClass variables bc and bcdc

bc.Method1();

bc.Method2();

dc.Method1();

dc.Method2();

bcdc.Method1();

bcdc.Method2();

you see that the addition of the Method2 method in BaseClass causes a warning. The warning says that the Method2 method in DerivedClass hides the Method2 method in BaseClass. You are advised to use the **new** keyword in the Method2 definition if you intend to cause that result. Alternatively, you could rename one of the Method2 methods to resolve the warning, but that is not always practical.

Before adding **new**, run the program to see the output produced by the additional calling statements. The following results are displayed.

// Output:

// Base - Method1

// Base - Method2

// Base - Method1

// Derived - Method2

// Base - Method1

// Base - Method2

这里，如果在derived class把method2变为private，那么dc.Method2()就会无法访问derived class里的Method2，所以会找到被隐藏起来的base class里的Method2。

所谓的new keyword hides the Method2 in BaseClass，是说如果有一个derived class的实例，那么你用这个实例调用Method2这个方法，永远调用的都是derived class的Method2方法，所以base class的Method2方法就被隐藏起来了，也可以显示的调用被隐藏起来的成员。看下面例子：

public class BaseC

{

public static int x = 55;

public static int y = 22;

}

public class DerivedC : BaseC

{

// Hide field 'x'.

new public static int x = 100;

static void Main()

{

// Display the new value of x:

Console.WriteLine(x);

// Display the hidden value of x:

Console.WriteLine(BaseC.x);这里显示的调用了base class被隐藏的成员

// Display the unhidden member y:

Console.WriteLine(y);

}

}

下面的例子以内部类作为成员：

public class BaseC

{

public class NestedC

{

public int x = 200;

public int y;

}

}

public class DerivedC : BaseC

{

// Nested type hiding the base type members.

new public class NestedC

{

public int x = 100;

public int y;

public int z;

}

static void Main()

{

// Creating an object from the overlapping class:

NestedC c1 = new NestedC();

// Creating an object from the hidden class:

BaseC.NestedC c2 = new BaseC.NestedC();显示调用base class的内部类成员

Console.WriteLine(c1.x);

Console.WriteLine(c2.x);

}

}

下面来看一个关于多态的例子：

class Car

{

public void DescribeCar()

{

System.Console.WriteLine("Four wheels and an engine.");

ShowDetails();

}

public virtual void ShowDetails()

{

System.Console.WriteLine("Standard transportation.");

}

}

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails hides the base class method.

class ConvertibleCar : Car

{

public new void ShowDetails()

{

System.Console.WriteLine("A roof that opens up.");

}

}

// Class Minivan uses the override modifier to specify that ShowDetails extends the base class method.

class Minivan : Car

{

public override void ShowDetails()

{

System.Console.WriteLine("Carries seven people.");

}

}

public static void TestCars1()

{

System.Console.WriteLine("\nTestCars1");

System.Console.WriteLine("----------");

Car car1 = new Car();

car1.DescribeCar();

System.Console.WriteLine("----------");

// Notice the output from this test case. The new modifier is used in the definition of ShowDetails in the ConvertibleCar

ConvertibleCar car2 = new ConvertibleCar();

car2.DescribeCar();

System.Console.WriteLine("----------");

Minivan car3 = new Minivan();

car3.DescribeCar();

System.Console.WriteLine("----------");

}

观察output

// TestCars1

// ----------

// Four wheels and an engine.

// Standard transportation.

// ----------

// Four wheels and an engine.

// Standard transportation.

// ----------

// Four wheels and an engine.

// Carries seven people.

// ----------

上面程序中，ConvertibleCar中的ShowDetail方法并没有覆写Car中的ShowDetail方法，而且程序并没有直接调用ShowDetail方法（如果直接调用ShowDetail方法，那么car2.ShowDetail()就会调用ConvertibleCar类中自己方法），而是通过一个通用的基类方法DescribeCar来调用ShowDetail方法。当car2调用DescribeCar方法时，由于没有覆写ShowDetail（可以这么想，DescribeCar调用的是ShowDetail方法，而基类本身就实现了ShowDetail方法，而且在ConvertibleCar中是一个new ShowDetail方法，可以把这个方法想成一个名字不同的方法，比如newShowDetail）。所以调用的是基类的方法。再看一个例子：

public static void TestCars2()

{

System.Console.WriteLine("\nTestCars2");

System.Console.WriteLine("----------");

var cars = new List<Car> { new Car(), new ConvertibleCar(),

new Minivan() };

foreach (var car in cars)

{

car.DescribeCar();

System.Console.WriteLine("----------");

}

}

// TestCars2

// Four wheels and an engine.

// Standard transportation.

// Four wheels and an engine.

// Standard transportation.

// Four wheels and an engine.

// Carries seven people.

// ----------

这里用父类引用指向子类实例，执行结果还是和上面程序一样。

Auto-implemented Property:

auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. They also enable client code to create objects. When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's get and set accessors.

也就是其实是有一个自动生成的private的变量来储存这个property的值

class Customer

{

// Auto-Impl Properties for trivial get and set

public double TotalPurchases { get; set; }

public string Name { get; set; }

public int CustomerID { get; set; }

// Constructor

public Customer(double purchases, string name, int ID)

{

TotalPurchases = purchases;

Name = name;

CustomerID = ID;

}

// Methods

public string GetContactInfo() {return "ContactInfo";}

public string GetTransactionHistory() {return "History";}

// .. Additional methods, events, etc.

}

class Program

{

static void Main()

{

// Intialize a new object.

Customer cust1 = new Customer ( 4987.63, "Northwind",90108 );

//Modify a property

cust1.TotalPurchases += 499.99;

}

}

The class that is shown in the previous example is mutable. Client code can change the values in objects after they are created. In complex classes that contain significant behavior (methods) as well as data, it is often necessary to have public properties. However, for small classes or structs that just encapsulate a set of values (data) and have little or no behaviors, it is recommended to make the objects immutable by declaring the set accessor as [private](https://msdn.microsoft.com/en-us/library/st6sy9xe.aspx).

## Delegate, Using delegate, Delegates with Named vs. Anonymous Methods, Variance in Delegate, Variance in Generic Type Parameters

A *delegate* is a type that enables you to store references to functions. Although this sounds quite

involved, the mechanism is surprisingly simple. The most important purpose of delegates will become

clear later in the book when you look at events and event handling, but it will be useful to briefly consider them here.

Delegates are declared much like functions, but with no function body and using the delegate keyword. The delegate declaration specifies a return type and parameter list. After defining a delegate, you can declare a variable with the type of that delegate. You can then initialize the variable as a reference to any function that has the same return type and parameter list as that delegate. Once you have done this, you can call that function by using the delegate variable as if it were a function.

public delegate int PerformCalculation(int x, int y);

When you have a variable that refers to a function, you can also perform other operations that would

be otherwise impossible. For example, you can pass a delegate variable to a function as a parameter,

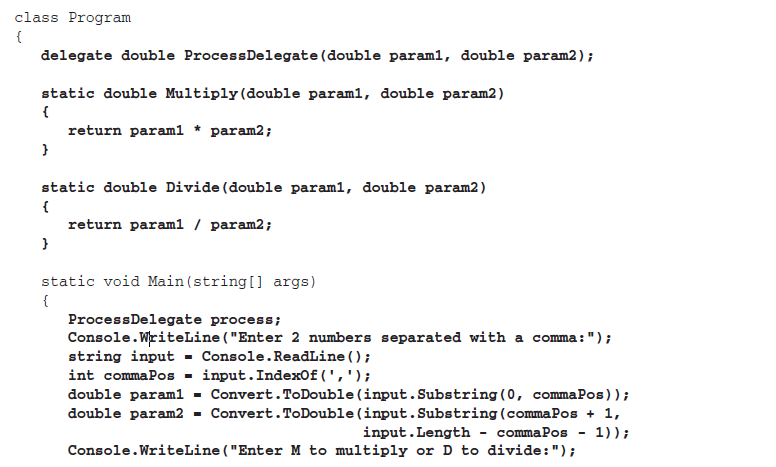
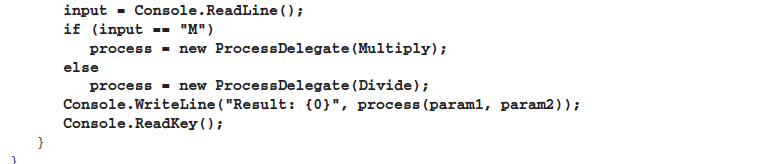
and then that function can use the delegate to call whatever function it refers to, without knowing what

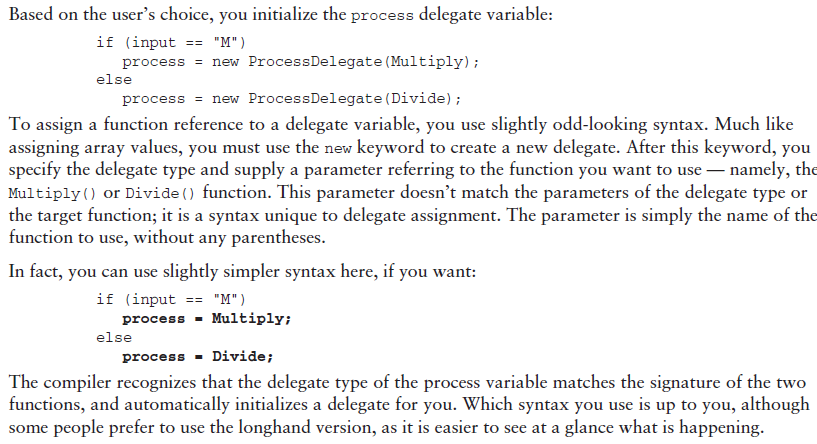
function will be called until runtime.

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This makes it possible to programmatically change method calls, and also plug new code into existing classes.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. For example, a reference to a method that compares two objects could be passed as an argument to a sort algorithm. Because the comparison code is in a separate procedure, the sort algorithm can be written in a more general way.

The following Try It Out demonstrates using a delegate to access one of two functions.



Here, you treat the delegate variable as if it were a function name. Unlike a function, though, you can also perform additional operations on this variable, such as passing it to a function via a parameter, as shown in this simple example:

static void ExecuteFunction(ProcessDelegate process)

{

process(2.2, 3.3);

}

This means that you can control the behavior of functions by passing them function delegates, much

like choosing a ‘‘snap-in’’ to use. For example, you might have a function that sorts a string array

alphabetically. You can use several techniques to sort lists, with varying performance depending on the

characteristics of the list being sorted. By using delegates, you can specify the function to use by passing a sorting algorithm function delegate to a sorting function.

**Using Delegate:**

1. 定义一个delegate: public delegate void Del(string message);特别注意！delegate可以不在class中定义，看下面的例子：

delegate void Del(int i, double j);

class MathClass

{

static void Main()

{

MathClass m = new MathClass();

// Delegate instantiation using "MultiplyNumbers"

Del d = m.MultiplyNumbers;

// Invoke the delegate object.

System.Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");

for (int i = 1; i <= 5; i++)

{

d(i, 2);

}

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

// Declare the associated method.

void MultiplyNumbers(int m, double n)

{

System.Console.Write(m \* n + " ");

}

}

1. 创建一个方法：

public static void DelegateMethod(string message)

{

System.Console.WriteLine(message);

}

1. 使用delegate:

A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with an [anonymous Method](https://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx).

// Instantiate the delegate.

Del handler = DelegateMethod;

或者 Del handler= new Del(DelegateMethod);

// Call the delegate.

handler("Hello World");

Delegate types are derived from the [Delegate](https://msdn.microsoft.com/en-us/library/system.delegate.aspx) class in the .NET Framework. Delegate types are [sealed](https://msdn.microsoft.com/en-us/library/88c54tsw.aspx)—they cannot be derived from— and it is not possible to derive custom classes from [Delegate](https://msdn.microsoft.com/en-us/library/system.delegate.aspx). Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property. This allows a method to accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous callback, and is a common method of notifying a caller when a long process has completed. When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.

Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the Del type as a parameter:

public void MethodWithCallback(int param1, int param2, Del callback)

{

callback("The number is: " + (param1 + param2).ToString());

}

You can then pass the delegate created above to that method:

MethodWithCallback(1, 2, handler);

When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature. When a delegate is constructed to wrap a static method, it only references the method. Consider the following declarations:

public class MethodClass

{

public void Method1(string message) { }

public void Method2(string message) { }

}

Along with the static DelegateMethod shown previously, we now have three methods that can be wrapped by a Del instance.

A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

MethodClass obj = new MethodClass();

Del d1 = obj.Method1;

Del d2 = obj.Method2;

Del d3 = DelegateMethod;

//Both types of assignment are valid.

Del allMethodsDelegate = d1 + d2;

allMethodsDelegate += d3;

可以用+号把几个delegate引用加起来，比如Del allMethodsDelegate = d1+d2;如果此后调用allMethodsDelegate的话，会依次调用d1和d2两个方法。

At this point allMethodsDelegate contains three methods in its invocation list—Method1, Method2, and DelegateMethod. The original three delegates, d1, d2, and d3, remain unchanged. When allMethodsDelegate is invoked, all three methods are called in order.

If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method.

When any of the methods throws an exception that is not caught within the method, that exception is passed to the caller of the delegate and no subsequent methods in the invocation list are called.

If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked. To remove a method from the invocation list, use the decrement or decrement assignment operator ('-' or '-='). For example:

//remove Method1

allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2

Del oneMethodDelegate = allMethodsDelegate - d2;

Because delegate types are derived from **System.Delegate**, the methods and properties defined by that class can be called on the delegate. For example, to find the number of methods in a delegate's invocation list, you may write:

int invocationCount = d1.GetInvocationList().GetLength(0);

Delegates with more than one method in their invocation list derive from [MulticastDelegate](https://msdn.microsoft.com/en-us/library/system.multicastdelegate.aspx), which is a subclass of **System.Delegate**. The above code works in either case because both classes support **GetInvocationList**.

Multicast delegates are used extensively in event handling. Event source objects send event notifications to recipient objects that have registered to receive that event. To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source. The source calls the delegate when the event occurs. The delegate then calls the event handling method on the recipient, delivering the event data. The delegate type for a given event is defined by the event source. For more, see [Events (C# Programming Guide)](https://msdn.microsoft.com/en-us/library/awbftdfh.aspx).

**Delegates with Named vs. Anonymous Methods:**

A [delegate](https://msdn.microsoft.com/en-us/library/900fyy8e.aspx) can be associated with a named method. When you instantiate a delegate by using a named method, the method is passed as a parameter, for example:

// Declare a delegate:

delegate void Del(int x);

// Define a named method:

void DoWork(int k) { /\* ... \*/ }

// Instantiate the delegate using the method as a parameter:

Del d = obj.DoWork;

This is called using a named method. Delegates constructed with a named method can encapsulate either a [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) method or an instance method. Named methods are the only way to instantiate a delegate in earlier versions of C#. However, in a situation where creating a new method is unwanted overhead, C# enables you to instantiate a delegate and immediately specify a code block that the delegate will process when it is called. The block can contain either a lambda expression or an anonymous method. For more information, see [Anonymous Functions (C# Programming Guide)](https://msdn.microsoft.com/en-us/library/bb882516.aspx).

Create delegate with an anonymous method:

// Create a delegate.

delegate void Del(int x);

// Instantiate the delegate using an anonymous method.

Del d = delegate(int k) { /\* ... \*/ };

更多关于匿名方法将在以后讨论

In the following example, one delegate is mapped to both static and instance methods and returns specific information from each.

// Declare a delegate

delegate void Del();

class SampleClass

{

public void InstanceMethod()

{

System.Console.WriteLine("A message from the instance method.");

}

static public void StaticMethod()

{

System.Console.WriteLine("A message from the static method.");

}

}

class TestSampleClass

{

static void Main()

{

SampleClass sc = new SampleClass();

// Map the delegate to the instance method:

Del d = sc.InstanceMethod;

d();

// Map to the static method:

d = SampleClass.StaticMethod;

d();

}

}

/\* Output:

A message from the instance method.

A message from the static method.

\*/

注意，虽然两个方法的签名都有public，但是delegate的签名并没有public，所以上面说的一个delegate能代理和这个delegate签名相同的方法，这里的签名指的是返回值，方法名和方法参数。

**Variance in Delegate:**

.NET Framework 3.5 and Visual Studio 2008 introduced variance support for matching method signatures with delegate types in all delegates in C# and Visual Basic. This means that you can assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. This includes both generic and non-generic delegates.

比如一个delegate的return type是Object,那它代理的方法的return type可以是Object或者Object的子类。

比如一个delegate接受的参数是string,那它代理的方法接受的参数可以是string或者string的父类

For example, consider the following code, which has two classes and two delegates: generic and non-generic.

public class First { }

public class Second : First { }

public delegate First SampleDelegate(Second a);

public delegate R SampleGenericDelegate<A, R>(A a);

When you create delegates of the SampleDelegate or SampleGenericDelegate<A, R> (SampleDelegate(Of A, R) in Visual Basic) types, you can assign any one of the following methods to those delegates.

// Matching signature.

public static First ASecondRFirst(Second first)

{ return new First(); }

// The return type is more derived.

public static Second ASecondRSecond(Second second)

{ return new Second(); }

// The argument type is less derived.

public static First AFirstRFirst(First first)

{ return new First(); }

// The return type is more derived

// and the argument type is less derived.

public static Second AFirstRSecond(First first)

{ return new Second(); }

The following code example illustrates the implicit conversion between the method signature and the delegate type.

// Assigning a method with a matching signature

// to a non-generic delegate. No conversion is necessary.

SampleDelegate dNonGeneric = ASecondRFirst;

// Assigning a method with a more derived return type

// and less derived argument type to a non-generic delegate.

// The implicit conversion is used.

SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.

// No conversion is necessary.

SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;

// Assigning a method with a more derived return type

// and less derived argument type to a generic delegate.

// The implicit conversion is used.

SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;

Example of covariance:

This example demonstrates how delegates can be used with methods that have return types that are derived from the return type in the delegate signature. The data type returned by DogsHandler is of type Dogs, which derives from the Mammals type that is defined in the delegate.

class Mammals{ }

class Dogs : Mammals{ }

class Program

{

// Define the delegate.

public delegate Mammals HandlerMethod();

public static Mammals MammalsHandler()

{

return null;

}

public static Dogs DogsHandler()

{

return null;

}

static void Test()

{

HandlerMethod handlerMammals = MammalsHandler;

// Covariance enables this assignment.

HandlerMethod handlerDogs = DogsHandler;

}

}

This example demonstrates how delegates can be used with methods that have parameters of a type that are base types of the delegate signature parameter type. With contravariance, you can use one event handler instead of separate handlers. For example, you can create an event handler that accepts an EventArgs input parameter and use it with a Button.MouseClick event that sends a MouseEventArgs type as a parameter, and also with a TextBox.KeyDown event that sends a KeyEventArgs parameter.

// Event hander that accepts a parameter of the EventArgs type.

private void MultiHandler(object sender, System.EventArgs e)

{

label1.Text = System.DateTime.Now.ToString();

}

public Form1()

{

InitializeComponent();

// You can use a method that has an EventArgs parameter,

// although the event expects the KeyEventArgs parameter.

this.button1.KeyDown += this.MultiHandler;

// You can use the same method

// for an event that expects the MouseEventArgs parameter.

this.button1.MouseClick += this.MultiHandler;

}

Delegate 可以互相assign

delegate Animal Del();

public Animal GetAnimal()

{

return new Animal();

}

public Cat GetCat()

{

return new Cat();

}

Del d1;

Del d2;

Animal a = new Animal();

Cat c = new Cat();

d1 = a.GetAnimal;

d2 = c.GetCat;

d1 = d2;

也可以d2=d1，这里的delegate d1,d2拥有同样的签名，所以可以互相assign。如果此时有一个delegate Animal Del2();Del2 d2=a.GetAnimal。此时尽管d1和d2签名完全一样，代理的方法也完全一样，这两个delegate还是视为不同的类型，不能互相assign。Generic Delegate也是一样，Del<string>和Del<Object>是两个不同的delegate,不能互相assign,除非在delegate的定义中使用了variance，下面将介绍。

**Variance in Generic Type Parameters:**

In .NET Framework 4, you can enable implicit conversion between delegates, so that generic delegates that have different types specified by generic type parameters can be assigned to each other, if the types are inherited from each other as required by variance.

To enable implicit conversion, you must explicitly declare generic parameters in a delegate as covariant or contravariant by using the **in** or **out** keyword.

The following code example shows how you can create a delegate that has a covariant generic type parameter.

// Type T is declared covariant by using the out keyword.

public delegate T SampleGenericDelegate <out T>();

public static void Test()

{

SampleGenericDelegate <String> dString = () => " ";

// You can assign delegates to each other,

// because the type T is declared covariant.

SampleGenericDelegate <Object> dObject = dString;

}

If you use only variance support to match method signatures with delegate types and do not use the **in** and **out** keywords, you may find that sometimes you can instantiate delegates with identical lambda expressions or methods, but you cannot assign one delegate to another.

In the following code example, SampleGenericDelegate<String> cannot be explicitly converted to SampleGenericDelegate<Object> (SampleGenericDelegate(Of String) to SampleGenericDelegate(Of Object) in Visual Basic), although String inherits Object. You can fix this problem by marking the generic parameter T with the**out** keyword.

public delegate T SampleGenericDelegate<T>();

public static void Test()

{

SampleGenericDelegate<String> dString = () => " ";

// You can assign the dObject delegate

// to the same lambda expression as dString delegate

// because of the variance support for

// matching method signatures with delegate types.

SampleGenericDelegate<Object> dObject = () => " ";

// The following statement generates a compiler error

// because the generic type T is not marked as covariant.

// SampleGenericDelegate <Object> dObject = dString;

}

[**Generic Delegates That Have Variant Type Parameters in the .NET Framework**](javascript:void(0))

.NET Framework 4 introduces variance support for generic type parameters in several existing generic delegates:

* **Action** delegates from the [System](https://msdn.microsoft.com/en-us/library/system.aspx) namespace, for example, [Action<T>](https://msdn.microsoft.com/en-us/library/018hxwa8.aspx) and [Action<T1, T2>](https://msdn.microsoft.com/en-us/library/bb549311.aspx)
* **Func** delegates from the [System](https://msdn.microsoft.com/en-us/library/system.aspx) namespace, for example, [Func<TResult>](https://msdn.microsoft.com/en-us/library/bb534960.aspx) and [Func<T, TResult>](https://msdn.microsoft.com/en-us/library/bb549151.aspx)
* The [Predicate<T>](https://msdn.microsoft.com/en-us/library/bfcke1bz.aspx) delegate
* The [Comparison<T>](https://msdn.microsoft.com/en-us/library/tfakywbh.aspx) delegate
* The [Converter<TInput, TOutput>](https://msdn.microsoft.com/en-us/library/kt456a2y.aspx) delegate

[**Declaring Variant Type Parameters in Generic Delegates**](javascript:void(0))

If a generic delegate has covariant or contravariant generic type parameters, it can be referred to as a *variant generic delegate*.

You can declare a generic type parameter covariant in a generic delegate by using the **out** keyword. The covariant type can be used only as a method return type and not as a type of method arguments. The following code example shows how to declare a covariant generic delegate.

public delegate R DCovariant<out R>();

You can declare a generic type parameter contravariant in a generic delegate by using the **in** keyword. The contravariant type can be used only as a type of method arguments and not as a method return type. The following code example shows how to declare a contravariant generic delegate.

public delegate void DContravariant<in A>(A a);

It is also possible to support both variance and covariance in the same delegate, but for different type parameters. This is shown in the following example.

public delegate R DVariant<in A, out R>(A a);

[**Instantiating and Invoking Variant Generic Delegates**](javascript:void(0))

You can instantiate and invoke variant delegates just as you instantiate and invoke invariant delegates. In the following example, the delegate is instantiated by a lambda expression.

DVariant<String, String> dvariant = (String str) => str + " ";

dvariant("test");

[**Combining Variant Generic Delegates**](javascript:void(0))

You should not combine variant delegates. The [Combine](https://msdn.microsoft.com/en-us/library/system.delegate.combine.aspx) method does not support variant delegate conversion and expects delegates to be of exactly the same type. This can lead to a run-time exception when you combine delegates either by using the [Combine](https://msdn.microsoft.com/en-us/library/system.delegate.combine.aspx) method (in C# and Visual Basic) or by using the **+** operator (in C#), as shown in the following code example.

Action<object> actObj = x => Console.WriteLine("object: {0}", x);

Action<string> actStr = x => Console.WriteLine("string: {0}", x);

// All of the following statements throw exceptions at run time.

// Action<string> actCombine = actStr + actObj;

// actStr += actObj;

// Delegate.Combine(actStr, actObj);

[**Variance in Generic Type Parameters for Value and Reference Types**](javascript:void(0))

Variance for generic type parameters is supported for reference types only. For example, DVariant<int> cannot be implicitly converted toDVariant<Object> or DVaraint<long>, because integer is a value type.

The following example demonstrates that variance in generic type parameters is not supported for value types.

// The type T is covariant.

public delegate T DVariant<out T>();

// The type T is invariant.

public delegate T DInvariant<T>();

public static void Test()

{

int i = 0;

DInvariant<int> dInt = () => i;

DVariant<int> dVariantInt = () => i;

// All of the following statements generate a compiler error

// because type variance in generic parameters is not supported

// for value types, even if generic type parameters are declared variant.

// DInvariant<Object> dObject = dInt;

// DInvariant<long> dLong = dInt;

// DVariant<Object> dVariantObject = dVariantInt;

// DVariant<long> dVariantLong = dVariantInt;

}

## Collection, abstract class, Static方法不能被重写, IDictionary, Iterator(prime numbers algorithem), Deep copying and shallow copying

Several interfaces in the System.Collections namespace provide basic collection functionality:

➤ IEnumerable— Provides the capability to loop through items in a collection

➤ ICollection— Provides the capability to obtain the number of items in a collection and copy

items into a simple array type (inherits from IEnumerable)

➤ IList—Provides a list of items for a collection along with the capabilities for accessing these

items, and some other basic capabilities related to lists of items (inherits from IEnumerable

and ICollection)

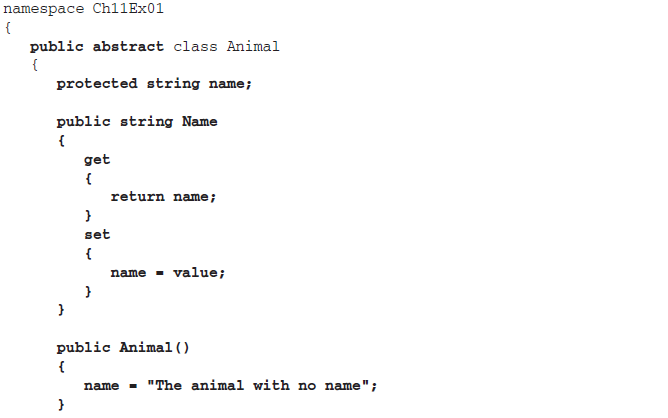
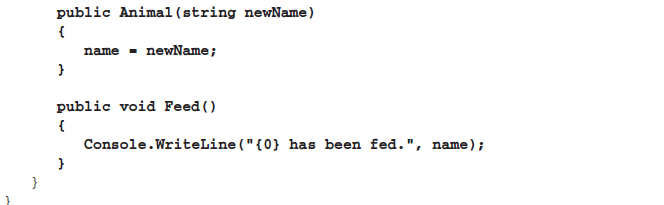
➤ IDictionary— Similar to IList, but provides a list of items accessible via a key value, rather

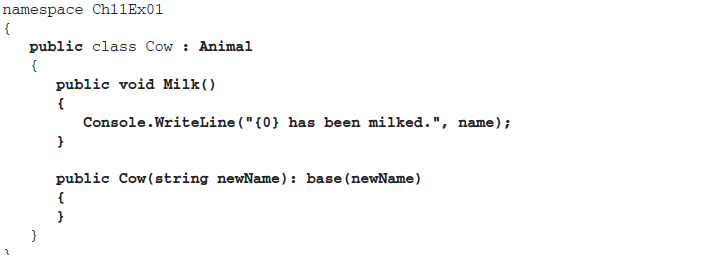
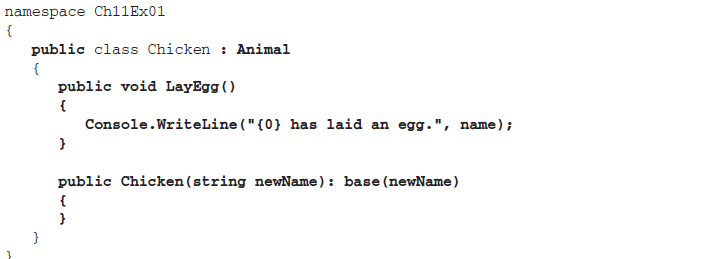
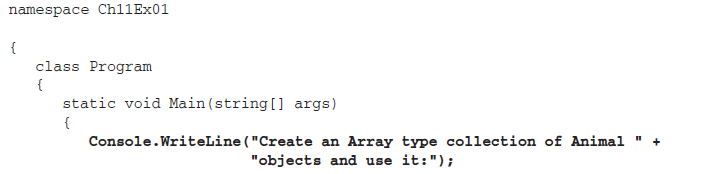
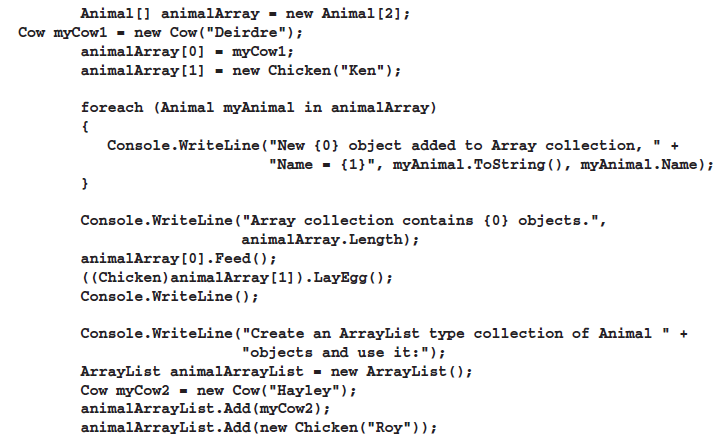
than an index (inherits from IEnumerable and ICollection)

The System.Array class implements IList, ICollection, and IEnumerable. However, it doesn’t support

some of the more advanced features of IList, and it represents a list of items by using a fixed size.

看下面的列子：



**Abstract class:(more about abstract class will discussed later)**

这个程序有几个需要注意的地方：

1. abstract class不一定要有abstract方法，但是有abstract方法的class必须申明为abstract class.
2. abstract class不能被实例化
3. 一个实现了abstract class的类必须重写所有的abstract方法
4. An abstract method is implicitly a virtual method.
5. Abstract method declarations are only permitted in abstract classes.
6. It is an error to use the [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) or [virtual](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) modifiers in an abstract method declaration.（下面解释为什么）

Static方法不能被覆写（不能用virtual和abstract修饰）：

在java中：

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

* The version of the overridden instance method that gets invoked is the one in the subclass.
* The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Consider an example that contains two classes. The first is Animal, which contains one instance method and one static method:

public class Animal {

public static void testClassMethod() {

System.out.println("The static method in Animal");

}

public void testInstanceMethod() {

System.out.println("The instance method in Animal");

}

}

The second class, a subclass of Animal, is called Cat:

public class Cat extends Animal {

public static void testClassMethod() {

System.out.println("The static method in Cat");

}

public void testInstanceMethod() {

System.out.println("The instance method in Cat");

}

public static void main(String[] args) {

Cat myCat = new Cat();

Animal myAnimal = myCat;

Animal.testClassMethod();

myAnimal.testInstanceMethod();

}

}

The Cat class overrides the instance method in Animal and hides the static method in Animal. The main method in this class creates an instance of Cat and invokes testClassMethod()on the class and testInstanceMethod() on the instance.

The output from this program is as follows:

The static method in Animal

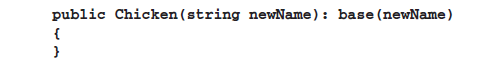
The instance method in Cat

As promised, the version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

可以看到，在java中，如果一个static方法在父类和子类中有同样的签名，并不是重写，而是看具体引用的是哪个方法，比如Animal.testClassMethod，也可以调用 Cat.testClassMethod.

而在C#中，这一切在语言层面上就自然实现了，在C#中要覆写一个方法，需要覆写的方法必须在父类中定义为virtual或者abstract，由于static方法不能覆写，所以规定static方法不能加virtual或者abstract关键字就理所当然。

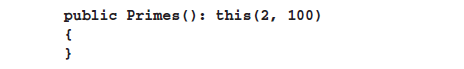
Collection章节的例子程序还有一个需要注意的地方，看下面的代码段：



在java中，如果一个子类想调用其直接父类的构造器，可以在子类构造器中使用super()，如果想调用父类的方法，可以用super关键字，super代表直接父类的对象的引用。

而在c#中，子类要调用父类构造器，则是采用上面的代码段的写法，此代码段调用了父类带一个string参数的构造器。

如果要调用本类的其他构造器的话，可以用下面代码段：



**IDictionary:**

Instead of the IList interface, it is also possible for collections to implement the similar IDictionary

interface, which allows items to be indexed via a key value (such as a string name), rather than an index.

This is also achieved using an indexer, although here the indexer parameter used is a key associated with

a stored item, rather than an int index, which can make the collection a lot more user-friendly.

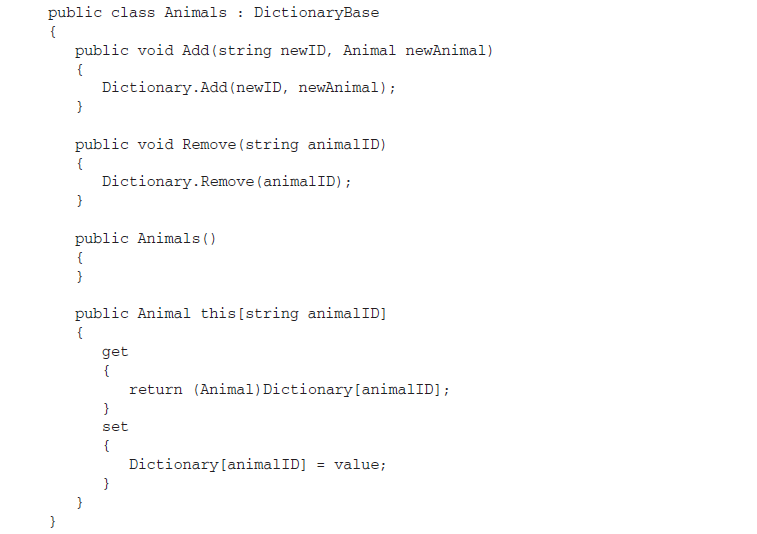
As with indexed collections, there is a base class you can use to simplify implementation of the

IDictionary interface: DictionaryBase. This class also implements IEnumerable and ICollection,

providing the basic collection manipulation capabilities that are the same for any collection.

The following code shows an alternative version of the Animals class, this time derived from

DictionaryBase. Implementations are included for Add(), Remove(), and a key-accessed indexer:

****

One other difference between collections based on DictionaryBase and collections based on

CollectionBase is that foreach works slightly differently. The collection from the last section allowed

you to extract Animal objects directly from the collection. Using foreach with the DictionaryBase

derived class gives you DictionaryEntry structs, another type defined in the System.Collections namespace. To get to the Animal objects themselves, you must use the Value member of this struct, or you can use the Key member of the struct to get the associated key. To get code equivalent to the earlier

foreach (Animal myAnimal in animalCollection)

{

Console.WriteLine("New {0} object added to custom collection, " +

"Name = {1}", myAnimal.ToString(), myAnimal.Name);

}

you need the following:

foreach (DictionaryEntry myEntry in animalCollection)

{

Console.WriteLine("New {0} object added to custom collection, " +

"Name = {1}", myEntry.Value.ToString(),

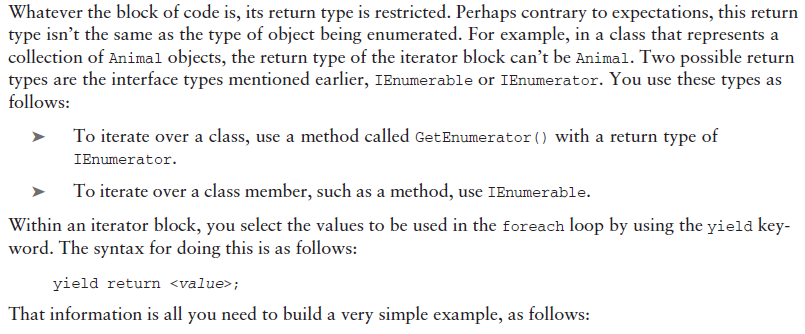
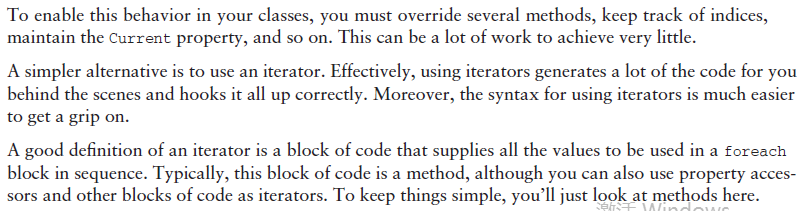
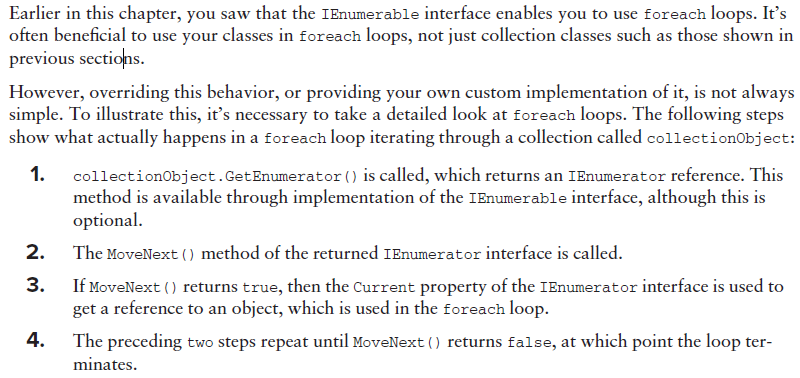
((Animal)myEntry.Value).Name);

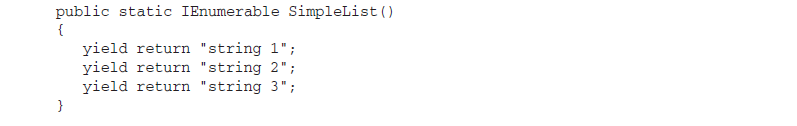
}

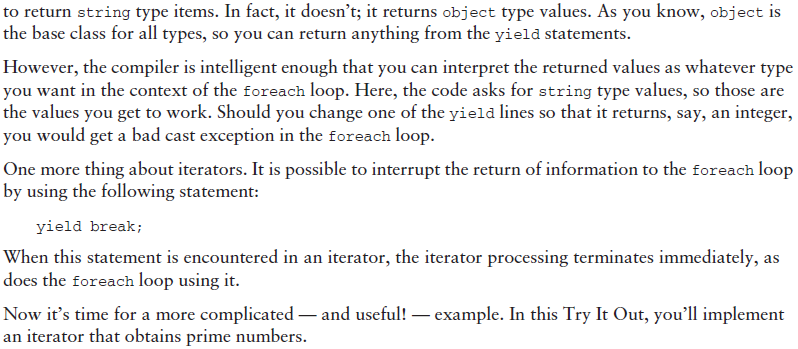
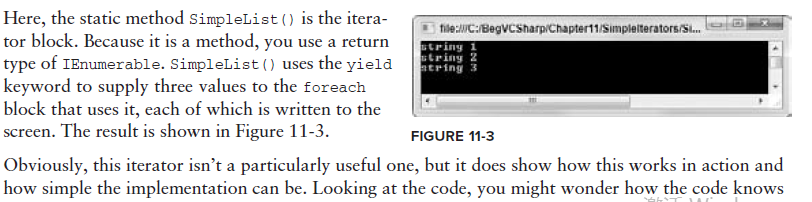
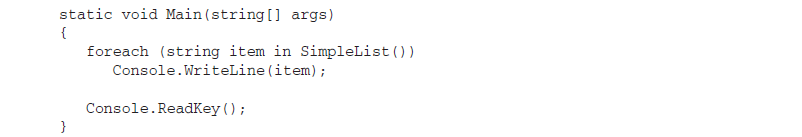
It is possible to override this behavior so that you can access Animal objects directly through foreach.

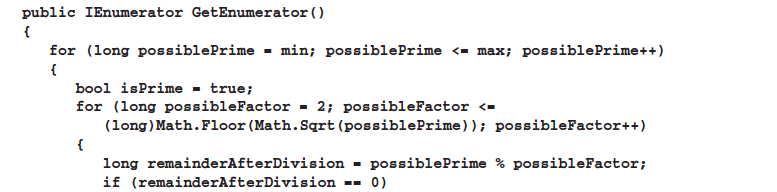
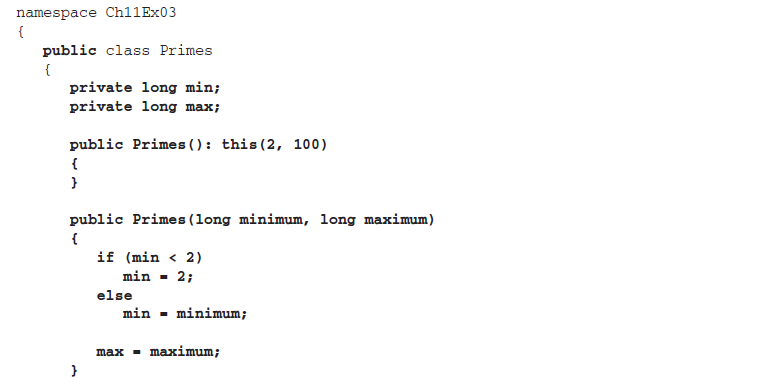
There are several ways to do this, the simplest being to implement an iterator.

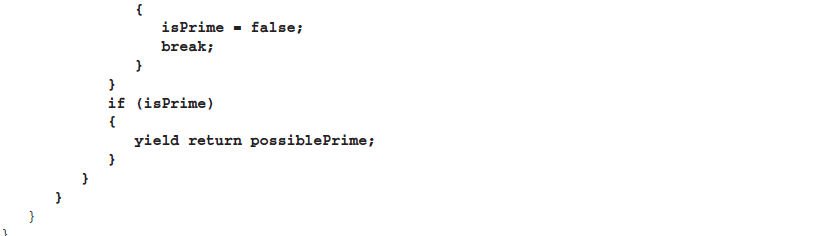
**Iterator:**

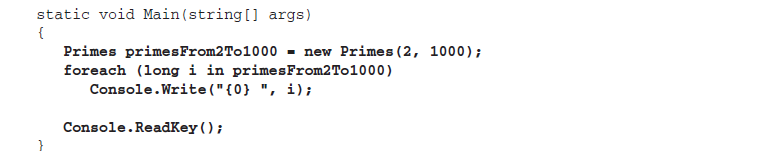
****

****

****

****

****

****

这个程序要注意的地方是他判断prime number的算法：判断一个数是否素数，只需判断它是否有非1，非本身的正因子。一般算法都是从2开始判断，设该数是N，假如N有大于 根号N 的因子，那么它的另一个因子必小于 根号N，那么计算机运算时查到这个因子时就可判断它不是素数，因此只需到[平方根](https://www.baidu.com/s?wd=%E5%B9%B3%E6%96%B9%E6%A0%B9&hl_tag=textlink&tn=SE_hldp01350_v6v6zkg6)，而不必查到 N-1

证明：n=ab, let b> , prove that: a<

因为: b> 所以ab>a

因为: n=ab 所以 n> a

n> a 两边同除 得到a< ，得证

**Deep copying and shallow copying:**

The MemberwiseClone method creates a shallow copy by creating a new object, and then copying the nonstatic fields of the current object to the new object. If a field is a value type, a bit-by-bit copy of the field is performed. If a field is a reference type, the reference is copied but the referred object is not; therefore, the original object and its clone refer to the same object.

举个例子：

public class Cloner

{

public int Val;

public Cloner(int newVal)

{

Val = newVal;

}

public object GetCopy()

{

return MemberwiseClone();

}

}

public class Program

{

static void Main(string[] args)

{

Cloner source = new Cloner(5);

Cloner dest = (Cloner)source.GetCopy();

source.Val = 8;

Console.WriteLine("source value is: {0} and dest value is :{1}", source.Val, dest.Val);

//output: source value is 8 and dest value is 5

Console.ReadKey();

}

}

这里Cloner类只有一个value type的成员变量，所以shallow copying可以满足程序需要，来看此时的内存情况：

Cloner

Cloner

dest

source

Val=5

Val=5

此时如果改变source指向的Cloner的Val的值，并不会影响其他的对象，下面看另一种情况。

Suppose you have fields that are reference types, rather than value types (for example, objects):

public class Content

{

public int Val;

}

public class Cloner

{

public Content MyContent = new Content();

public Cloner(int newVal)

{

MyContent.Val = newVal;

}

public object GetCopy()

{

return MemberwiseClone();

}

}

public class Program

{

static void Main(string[] args)

{

Cloner mySource = new Cloner(5);

Cloner myTarget = (Cloner)mySource.GetCopy();

Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);

mySource.MyContent.Val = 2;

Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);

Console.ReadKey();

}

}

来看这时的内存图：

dest

source

Cloner

MyContent

Content

Val=5

Cloner

MyContent

可以看到，由于Cloner对象包含一个reference type的成员变量，shallow copying并不会为这个成员变量所指向的对象创建新对象。所以source和dest对象的MyContent成员指向同一个对象，改变source的MyContent.Val的值也会影响dest。

To get around this, you need to perform a deep copy. You could just modify the GetCopy() method

used previously to do this, but it is preferable to use the standard .NET Framework way of doing

things: Implement the ICloneable interface, which has the single method Clone(). This method takes

no parameters and returns an object type result, giving it a signature identical to the GetCopy() method

used earlier.

To modify the preceding classes, try using the following deep copy code:

public class Content

{

public int Val;

}

public class Cloner : ICloneable

{

public Content MyContent = new Content();

public Cloner(int newVal)

{

MyContent.Val = newVal;

}

public object Clone()

{

Cloner clonedCloner = new Cloner(MyContent.Val);

return clonedCloner;

}

}

public class Program

{

static void Main(string[] args)

{

Cloner mySource = new Cloner(5);

Cloner myTarget = (Cloner)mySource.Clone();

Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);

mySource.MyContent.Val = 2;

Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);

Console.ReadKey();

}

}

This created a new Cloner object by using the Val field of the Content object contained in the original

Cloner object (MyContent). This field is a value type, so no deeper copying is necessary.

Using code similar to that just shown to test the shallow copy but using Clone() instead of GetCopy()

gives you the following result:

myTarget.MyContent.Val = 5

myTarget.MyContent.Val = 5

This time, the contained objects are independent. Note that sometimes calls to Clone() are made recursively,

in more complex object systems. For example, if the MyContent field of the Cloner class also

required deep copying, then you might need the following:

public object Clone()

{

Cloner clonedCloner = new Cloner();

clonedCloner.MyContent = MyContent.Clone();

return clonedCloner;

}

You’re calling the default constructor here to simplify the syntax of creating a new Cloner object. For

this code to work, you would also need to implement ICloneable on the Content class.

## Comparison, is operator, Value Comparison, operator overloading, IComparable and IComparer, Sorting, As operator

This section covers two types of comparisons between objects:

➤ Type comparisons

➤ Value comparisons

**Type Comparisons:**

When comparing objects, you often need to know their type, which may enable you to determine whether a value comparison is possible. In Chapter 9 you saw the GetType() method, which all classes inherit from System.Object, and how this method can be used in combination with the typeof() operator to determine (and take action depending on) object types:

if (myObj.GetType() == typeof(MyComplexClass))

{

// myObj is an instance of the class MyComplexClass.

}

这样的比较必须是两个类型完全match, 那么如果想知道myObj是不是MyComplexClass类型或者其子类该怎么办呢，下面将会讲到。

**The is Operator**

Despite its name, the is operator isn’t a way to determine whether an object *is* a certain type. Instead, the is operator enables you to check whether an object either is or *can be converted into* a given type. If this is the case, then the operator evaluates to true.

注意这里的convert是指自动转换

Earlier examples showed a Cow and a Chicken class, both of which inherit from Animal. Using the is operator to compare objects with the Animal type will return true for objects of all three of these types, not just Animal. This is something you’d have a hard time achieving with the GetType() method and typeof() operator shown previously.

The is operator has the following syntax:

<*operand*> is <*type*>

The possible results of this expression are as follows:

➤ If <*type*> is a class type, then the result is true if <*operand*> is of that type, if it inherits from that type, or if it can be boxed into that type.

➤ If <*type*> is an interface type, then the result is true if <*operand*> is of that type or it is a type that implements the interface.

➤ If <*type*> is a value type, then the result is true if <*operand*> is of that type or it is a type that can be unboxed into that type.

class A

{

}

class B : A , I

{

}

class C : A

{

}

interface I

{

}

class Program

{

static void Main(string[] args)

{

object o = new Object();

A a = new A();

B b = new B();

Console.WriteLine(a is B);//false a不能默认转型为B

a = b;

Console.WriteLine(a.GetType());//B GetType取得的是run time type B

Console.WriteLine(a is B);//true a的type已经是B了

Console.WriteLine(a is object);//true a的类型是Object的子类

Console.WriteLine(o is A);//false Object不能默认转型为A

Console.WriteLine(b is A);//true B是A的子类

Console.WriteLine(b is I);//true B实现了I

Console.WriteLine(a is int);//false A不能unbox为int

Console.ReadKey();

}

}

**Value Comparisons:**

Consider two Person objects representing people, each with an integer Age property. You might want to compare them to see which person is older. You can simply use the following code:

if (person1.Age > person2.Age)

{

...

}

This works fine, but there are alternatives. You might prefer to use syntax such as the following:

if (person1 > person2)

{

...

}

This is possible using *operator overloading*, which you’ll look at in this section. This is a powerful technique, but it should be used judiciously. In the preceding code, it is not immediately obvious that ages are being compared — it could be height, weight, IQ, or just general ‘‘greatness.’’

Another option is to use the IComparable and IComparer interfaces, which enable you to define how objects will be compared to each other in a standard way. This technique is supported by the various collection classes in the .NET Framework, making it an excellent way to sort objects in a collection.

**Operator Overloading:**

*Operator overloading* enables you to use standard operators, such as +, >, and so on, with classes that you design. This is called ‘‘overloading’’ because you are supplying your own implementations for these operators when used with specific parameter types, in much the same way that you overload methods by supplying different parameters for methods with the same name.

Operator overloading is useful because you can perform whatever processing you want in the implementation of the operator overload, which might not be as simple as, for example, +, meaning ‘‘add these two operands together.’’

To start with, though, here’s a look at the basic syntax for operator overloading. Operators may be overloaded by adding operator type members (which must be static) to a class. Some operators have multiple uses (such as -, which has unary and binary capabilities); therefore, you also specify how many operands you are dealing with and the types of these operands.

In general, you will have operands that are the same type as the class in which the operator is defined, although it’s possible to define operators that work on mixed types, as you’ll see shortly.

As an example, consider the simple type AddClass1, defined as follows:

public class AddClass1

{

public int val;

}

This is just a wrapper around an int value but it illustrates the principles. With this class, code such as the following will fail to compile:

AddClass1 op1 = new AddClass1();

op1.val = 5;

AddClass1 op2 = new AddClass1();

op2.val = 5;

AddClass1 op3 = op1 + op2;

The error you get informs you that the + operator cannot be applied to operands of the AddClass1 type. This is because you haven’t defined an operation to perform yet. Code such as the following works, but it won’t give you the result you might want:

AddClass1 op1 = new AddClass1();

op1.val = 5;

AddClass1 op2 = new AddClass1();

op2.val = 5;

**bool op3 = op1 == op2;**

Here, op1 and op2 are compared by using the == binary operator to determine whether they refer to the same object, *not* to verify whether their values are equal. op3 will be false in the preceding code, even though op1.val and op2.val are identical.

To overload the + operator, use the following code:

public class AddClass1

{

public int val;

**public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)**

**{**

**AddClass1 returnVal = new AddClass1();**

**returnVal.val = op1.val + op2.val;**

**return returnVal;**

**}**

}

As you can see, operator overloads look much like standard static method declarations, except that they use the keyword operator and the operator itself, rather than a method name. You can now successfully use the + operator with this class, as in the previous example:

AddClass1 op3 = op1 + op2;

Overloading all binary operators fits the same pattern. Unary operators look similar but have only one parameter:

**public static AddClass1 operator -(AddClass1 op1)**

**{**

**AddClass1 returnVal = new AddClass1();**

**returnVal.val = -op1.val;**

**return returnVal;**

**}**

Both these operators work on operands of the same type as the class and have return values that are also of that type. Consider, however, the following class definitions:

public class AddClass1

{

public int val;

public static AddClass3 operator +(AddClass1 op1, AddClass2 op2)

{

AddClass3 returnVal = new AddClass3();

returnVal.val = op1.val + op2.val;

return returnVal;

}

}

public class AddClass2

{

public int val;

}

public class AddClass3

{

public int val;

}

This will allow the following code:

AddClass1 op1 = new AddClass1();

op1.val = 5;

AddClass2 op2 = new AddClass2();

op2.val = 5;

AddClass3 op3 = op1 + op2;

When appropriate, you can mix types in this way. Note, however, that if you added the same operator to AddClass2, then the preceding code would fail because it would be ambiguous as to which operator to use. You should, therefore, take care not to add operators with the same signature to more than one class.

In addition, if you mix types, then the operands *must* be supplied in the same order as the parameters to the operator overload. If you attempt to use your overloaded operator with the operands in the wrong order, the operation will fail. For example, you can’t use the operator like:

AddClass3 op3 = op2 + op1;

unless, of course, you supply another overload with the parameters reversed:

public static AddClass3 operator +(AddClass2 op1, AddClass1 op2)

{

AddClass3 returnVal = new AddClass3();

returnVal.val = op1.val + op2.val;

return returnVal;

}

The following operators can be overloaded:

➤ **Unary operators:** +, -, !, ~, ++, --, true, false

➤ **Binary operators:** +, -, \*, /, %, &, |, ˆ , <<, >>

➤ **Comparison operators:** ==**,** !=**,** <**,** >**,** <=**,** >=

Some operators, such as < and >, must be overloaded in pairs. That is, you can’t overload < unless you also overload >. In many cases, you can simply call other operators from these to reduce the code required (and the errors that might occur), as shown in this example:

public class AddClass1

{

public int val;

**public static bool operator >=(AddClass1 op1, AddClass1 op2)**

**{**

**return (op1.val >= op2.val);**

**}**

**public static bool operator <(AddClass1 op1, AddClass1 op2)**

**{**

**return !(op1 >= op2);**

**}**

**// Also need implementations for <= and > operators.**

}

The same applies to == and !=, but with these operators it is often worth overriding Object.Equals() and Object.GetHashCode(), because both of these functions may also be used to compare objects. By overriding these methods, you ensure that whatever technique users of the class use, they get the same result. This isn’t essential, but it’s worth adding for completeness. It requires the following nonstatic override methods:

public class AddClass1

{

public int val;

public static bool operator ==(AddClass1 op1, AddClass1 op2)

{

return (op1.val == op2.val);

}

public static bool operator !=(AddClass1 op1, AddClass1 op2)

{

return !(op1 == op2);

}

**public override bool Equals(object op1)**

**{**

**return val == ((AddClass1)op1).val;**

**}**

**public override int GetHashCode()**

**{**

**return val;**

**}**

}

GetHashCode() is used to obtain a unique int value for an object instance based on its state. Here, using val is fine, because it is also an int value.

Note that Equals() uses an object type parameter. You need to use this signature or you will be overloading this method, rather than overriding it, and the default implementation will still be accessible to users of the class. Instead, you must use casting to get the required result. It is often worth checking the object type using the is operator discussed earlier, in code such as this:

public override bool Equals(object op1)

{

**if (op1 is AddClass1)**

**{**

return val == ((AddClass1)op1).val;

**}**

**else**

**{**

**throw new ArgumentException(**

**"Cannot compare AddClass1 objects with objects of type "**

**+ op1.GetType().ToString());**

**}**

}

**The IComparable and IComparer Interfaces:**

The IComparable and IComparer interfaces are the standard way to compare objects in the .NET Framework. The difference between the interfaces is as follows:

➤ IComparable is implemented in the class of the object to be compared and allows comparisons between that object and another object.

➤ IComparer is implemented in a separate class, which allows comparisons between any two objects.

IComparable exposes the single method CompareTo(), which accepts an object. You could, for example, implement it in a way that enables you to pass a Person object to it and determine whether that person is older or younger than the current person. In fact, this method returns an int, so you could also determine how much older or younger the second person is:

if (person1.CompareTo(person2) == 0)

{

Console.WriteLine("Same age");

}

else if (person1.CompareTo(person2) > 0)

{

Console.WriteLine("person 1 is Older");

}

else

{

Console.WriteLine("person1 is Younger");

}

IComparer exposes the single method Compare(), which accepts two objects and returns an integer result just like CompareTo(). With an object supporting IComparer, you could use code like the following:

if (personComparer.Compare(person1, person2) == 0)

{

Console.WriteLine("Same age");

}

else if (personComparer.Compare(person1, person2) > 0)

{

Console.WriteLine("person 1 is Older");

}

else

{

Console.WriteLine("person1 is Younger");

}

**Sorting:**

Many collection classes allow sorting, either by default comparisons between objects or by custom methods. ArrayList is one example. It contains the method Sort(), which can be used without parameters, in which case default comparisons are used, or it can be passed an IComparer interface to use to compare pairs of objects.

When you have an ArrayList filled with simple types, such as integers or strings, the default comparer is fine. For your own classes, you must either implement IComparable in your class definition or create a separate class supporting IComparer to use for comparisons.

Note that some classes in the System.Collection namespace, including CollectionBase, don’t expose a method for sorting. If you want to sort a collection you have derived from this class, then you have to do a bit more work and sort the internal List collection yourself.

来看下面的列子：

class Person : IComparable

{

public string Name;

public int Age;

public Person(string name, int age)

{

Name = name;

Age = age;

}

public int CompareTo(object obj)

{

if (obj is Person)

{

Person otherPerson = obj as Person;

return this.Age - otherPerson.Age;

}

else

{

throw new ArgumentException(

"Object to compare to is not a Person object.");

}

}

}

public class PersonComparerName : IComparer

{

public static IComparer Default = new PersonComparerName();

public int Compare(object x, object y)

{

if (x is Person && y is Person)

{

return Comparer.Default.Compare(

((Person)x).Name, ((Person)y).Name);

}

else

{

throw new ArgumentException(

"One or both objects to compare are not Person objects.");

}

}

}

class Program

{

static void Main(string[] args)

{

ArrayList list = new ArrayList();

list.Add(new Person("Jim", 30));

list.Add(new Person("Bob", 25));

list.Add(new Person("Bert", 27));

list.Add(new Person("Ernie", 22));

Console.WriteLine("Unsorted people:");

for (int i = 0; i < list.Count; i++)

{

Console.WriteLine("{0} ({1})",

(list[i] as Person).Name, (list[i] as Person).Age);

}

Console.WriteLine();

Console.WriteLine( "People sorted with default comparer (by age):");

list.Sort();

for (int i = 0; i < list.Count; i++)

{

Console.WriteLine("{0} ({1})",

(list[i] as Person).Name, (list[i] as Person).Age);

}

Console.WriteLine();

Console.WriteLine(

"People sorted with nondefault comparer (by name):");

list.Sort(PersonComparerName.Default);

for (int i = 0; i < list.Count; i++)

{

Console.WriteLine("{0} ({1})",

(list[i] as Person).Name, (list[i] as Person).Age);

}

Console.ReadKey();

}

}

有几个需要注意的地方：

1. Comparer.Default, The .NET Framework includes a default implementation of the IComparer interface on a class called Comparer, found in the System.Collections namespace. This class is capable of performing culturespecific comparisons between simple types, as well as any type that supports the IComparable interface.

This uses the Comparer.Default static member to obtain an instance of the Comparer class, and then uses the Compare() method to compare first two strings,

1. list.Sort(PersonComparerName.Default);Sort方法可以接收一个IComparer类型的参数。并会根据具体实现的Compare方法来比较对象

**As operator:**

As operator的作用也是强制类型转换，和强制类型转换的区别在于，如果不能成功转换，则会得到null值。看下面的例子：

class Program

{

static void Main(string[] args)

{

ClassA obj1 = new ClassA();

ClassD obj2 = obj1 as ClassD;//null

ClassD obj3 = (ClassD)obj1;//exception

ClassD \_obj1 = new ClassD();

ClassA \_obj2 = \_obj1;

ClassD \_obj3 = \_obj2 as ClassD;//Ok

Console.ReadKey();

}

}

这里obj1 as ClassD将会返回一个null而ClassD obj3 = (ClassD)obj1将会抛出异常。注意，向上转型是自动进行的，比如Object o = a, where a is string. String is an Object，所以可以转型。 向下转型要强制转换，然而并不是所有转换都能成功的。我们经常看到有的方法的签名是method(Object a),说明这个方法可以接受任何对象，如果我调用方法的时候放一个string进去，method(“Hello”)。那么在方法里，string b=(string)a这样的statement就完全可以,但是如果我放一个Object进去，method(o),尽管Object是String的父类，但是这样的转型是不安全的，因为o指向的是一个Object对象，它不能转换成String对象。此时将抛出ClassCasetException。在java中可以通过instanceof来防止出现异常。在C#中则是用is operator

## 关键字详解：static

In UML syntax, static members of classes appear with underlining: 

**Static Constructors:**

When using static members in a class, you may want to initialize these members beforehand. You can

supply a static member with an initial value as part of its declaration, but sometimes you may want to

perform a more complex initialization, or perhaps perform some operations before assigning values or

allowing static methods to execute.

You can use a static constructor to perform initialization tasks of this type. A class can have a single static constructor, which must have no access modifiers and cannot have any parameters. A static constructor can never be called directly; instead, it is executed when one of the following occurs:

* An instance of the class containing the static constructor is created.
* A static member of the class containing the static constructor is accessed.

In both cases, the static constructor is called first, before the class is instantiated or static members accessed. No matter how many instances of a class are created, its static constructor will only be called once. To differentiate between static constructors and the constructors described earlier in this chapter, all nonstatic constructors are also known as *instance constructors.*

A static constructor is used to initialize any [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced.

class SimpleClass

{

// Static variable that must be initialized at run time.

static readonly long baseline;

// Static constructor is called at most one time, before any

// instance constructor is invoked or member is accessed.

static SimpleClass()

{

baseline = DateTime.Now.Ticks;

}

}

* A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
* Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the **LoadLibrary** method.
* If a static constructor throws an exception, the runtime will not invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain in which your program is running.

In this example, class Bus has a static constructor. When the first instance of Bus is created (bus1), the static constructor is invoked to initialize the class. The sample output verifies that the static constructor runs only one time, even though two instances of Bus are created, and that it runs before the instance constructor runs.

public class Bus

{

// Static variable used by all Bus instances.

// Represents the time the first bus of the day starts its route.

protected static readonly DateTime globalStartTime;

// Property for the number of each bus.

protected int RouteNumber { get; set; }

// Static constructor to initialize the static variable.

// It is invoked before the first instance constructor is run.

static Bus()

{

globalStartTime = DateTime.Now;

// The following statement produces the first line of output,

// and the line occurs only once.

Console.WriteLine("Static constructor sets global start time to {0}",

globalStartTime.ToLongTimeString());

}

// Instance constructor.

public Bus(int routeNum)

{

RouteNumber = routeNum;

Console.WriteLine("Bus #{0} is created.", RouteNumber);

}

// Instance method.

public void Drive()

{

TimeSpan elapsedTime = DateTime.Now - globalStartTime;

// For demonstration purposes we treat milliseconds as minutes to simulate

// actual bus times. Do not do this in your actual bus schedule program!

Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",

this.RouteNumber,

elapsedTime.TotalMilliseconds,

globalStartTime.ToShortTimeString());

}

}

class TestBus

{

static void Main()

{

// The creation of this instance activates the static constructor.

Bus bus1 = new Bus(71);

// Create a second bus.

Bus bus2 = new Bus(72);

// Send bus1 on its way.

bus1.Drive();

// Wait for bus2 to warm up.

System.Threading.Thread.Sleep(25);

// Send bus2 on its way.

bus2.Drive();

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

/\* Sample output:

Static constructor sets global start time to 3:57:08 PM.

Bus #71 is created.

Bus #72 is created.

71 is starting its route 6.00 minutes after global start time 3:57 PM.

72 is starting its route 31.00 minutes after global start time 3:57 PM.

\*/

**Static Classes：**

Often, you will want to use classes that contain only static members and cannot be used to instantiate

objects (such as Console). A shorthand way to do this, rather than make the constructors of the class

private, is to use a *static class.* A static class can contain only static members and can’t have instance

constructors, since by implication it can never be instantiated. Static classes can, however, have a static

constructor, as described in the preceding section.

Static可以用来修饰Property。在interface中，不能有static修饰的成员