目录

[什么是.Net框架，框架的编译运行原理 2](#_Toc424905491)

[变量的范围，常量表，verbatim，类型转换，操作符的优先级 7](#_Toc424905492)

[Namespace，using，const，访问修饰符，sealed,virtual,override 10](#_Toc424905493)

[Loops，Enumeration，typeof，struct，Array 15](#_Toc424905494)

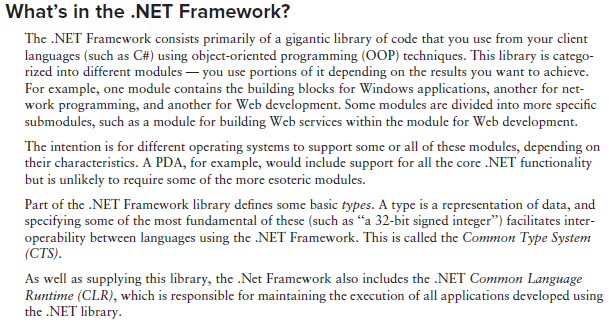
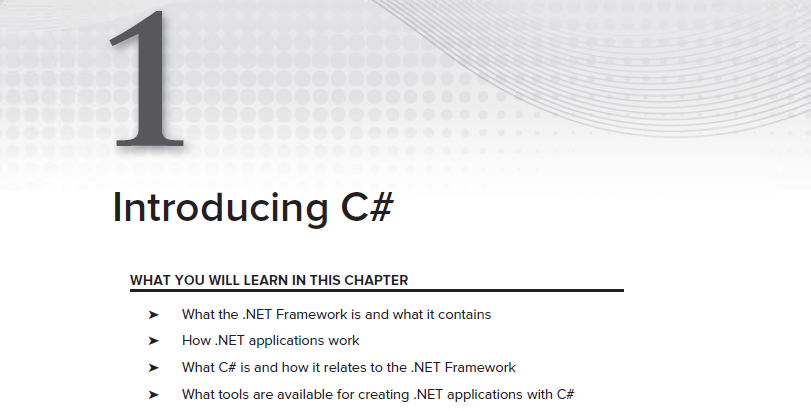
[String manipulation 22](#_Toc424905495)

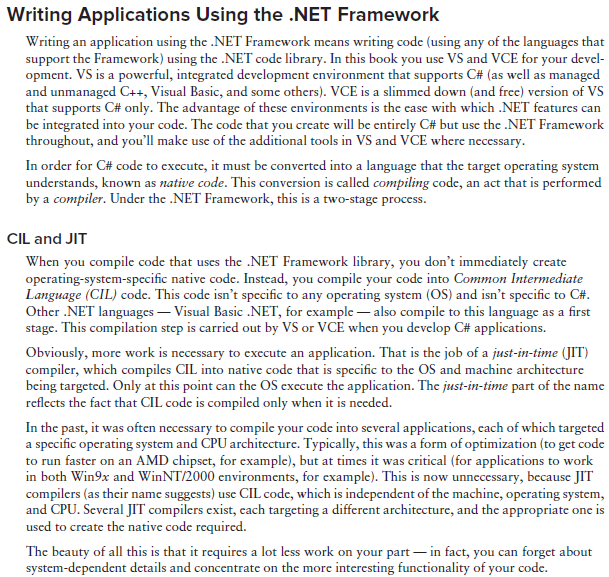
[泛型，类加载问题，Nullable types 25](#_Toc424905496)

[Indexer，using indexer，indexer in interface，Difference between Property and Indexer 27](#_Toc424905497)

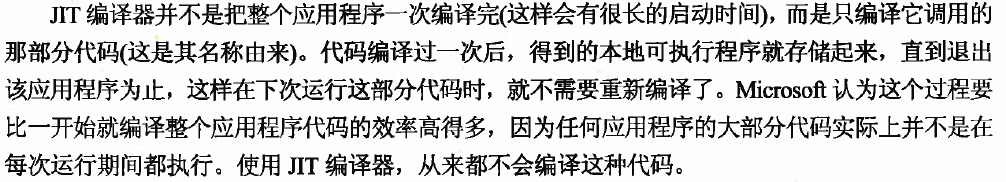
[Property，Using Property，interface Property，Restricting Accessor Accessibility，new as modifier 32](#_Toc424905498)

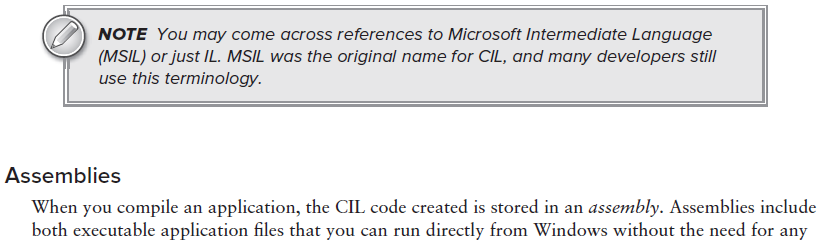
## 什么是.Net框架，框架的编译运行原理

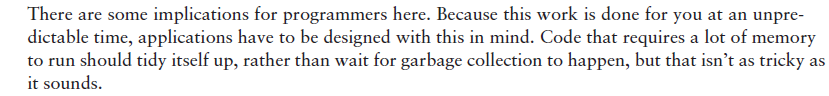
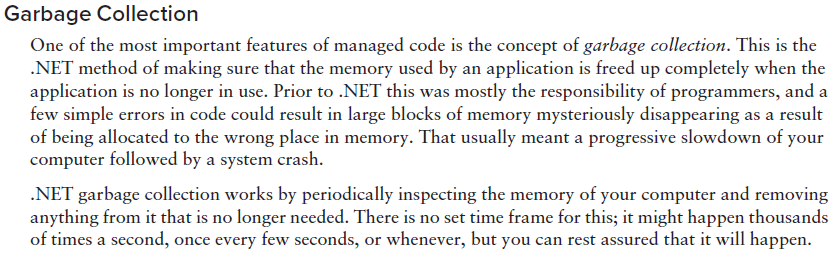
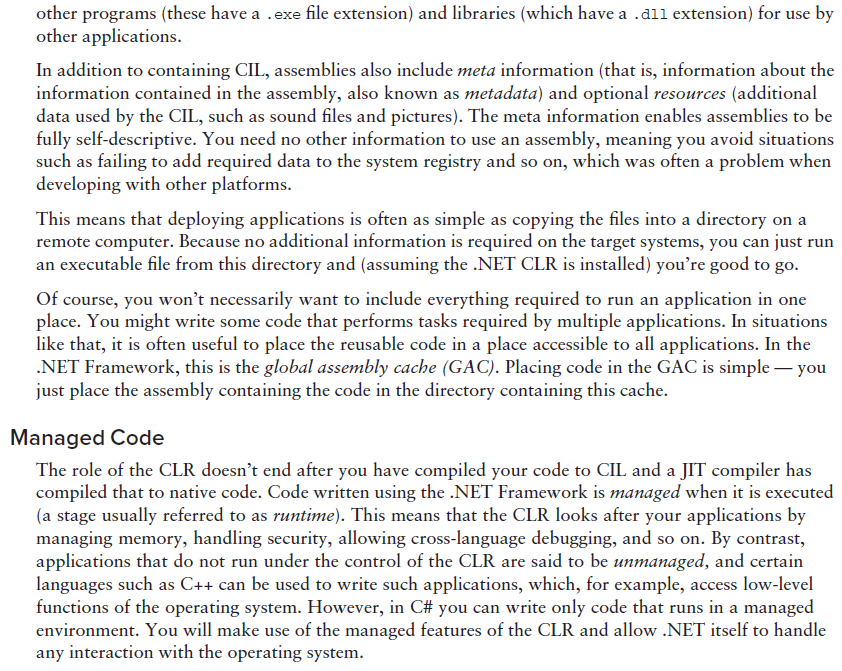


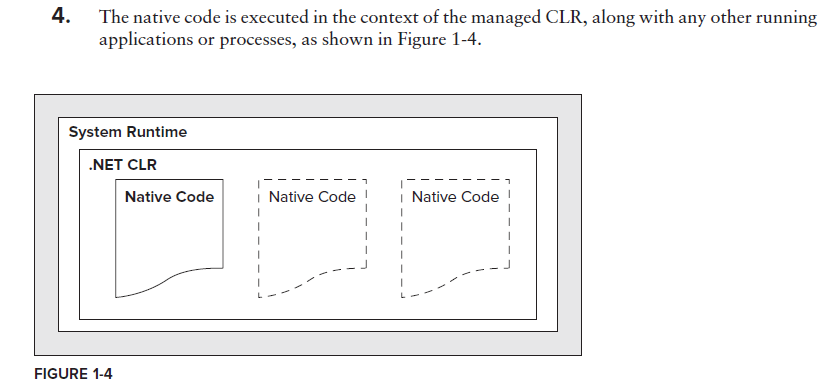
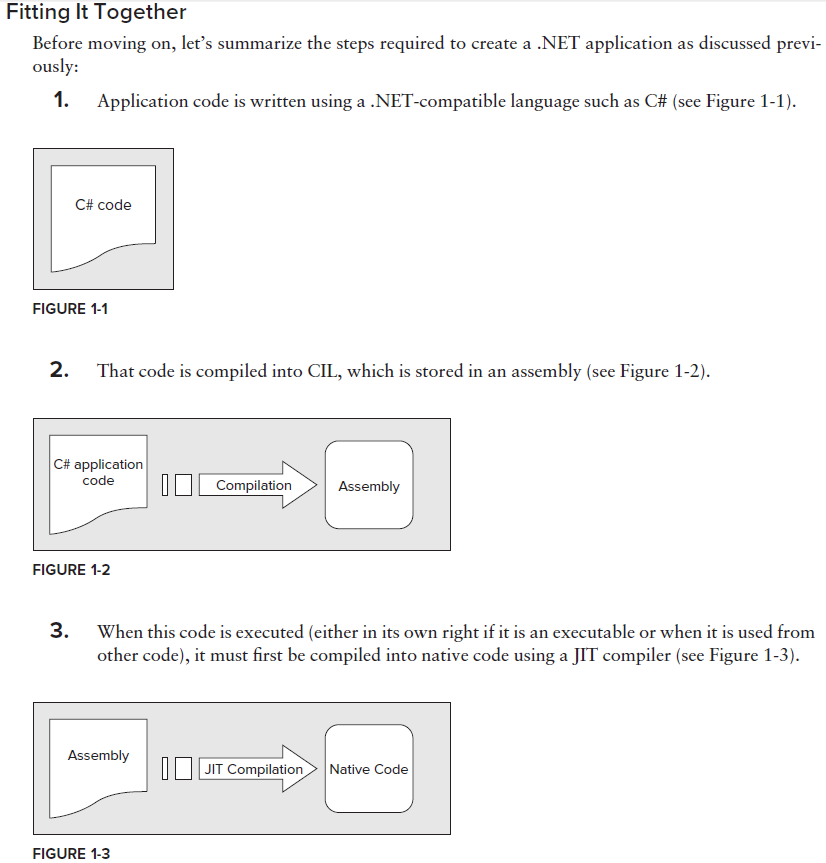


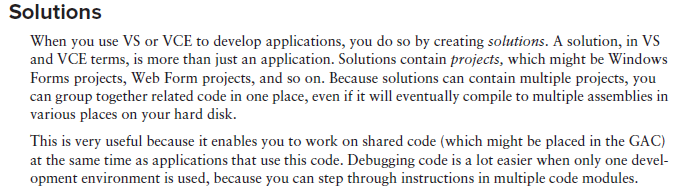
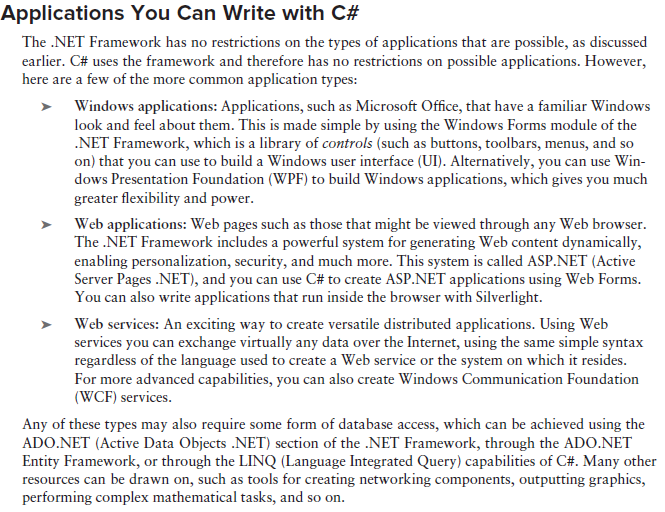
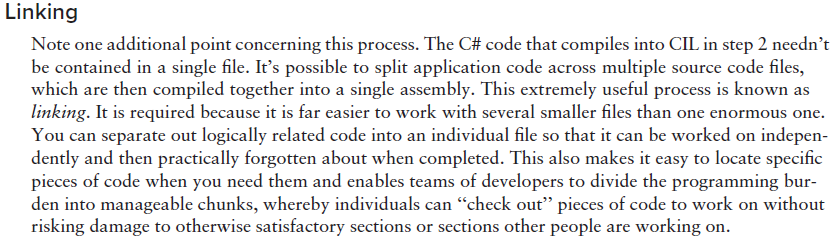
JIT在需要的时候把CIL编译成机器对应的native code，然后就可以运行



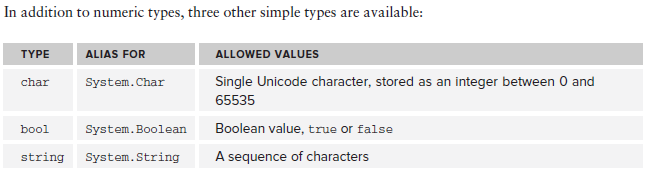
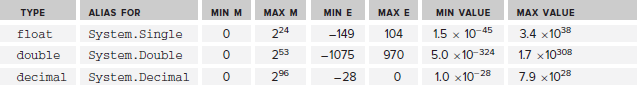
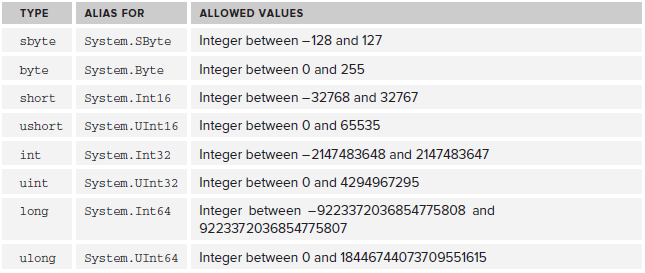




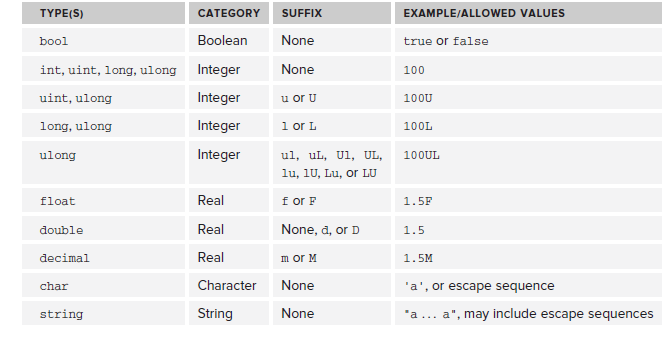




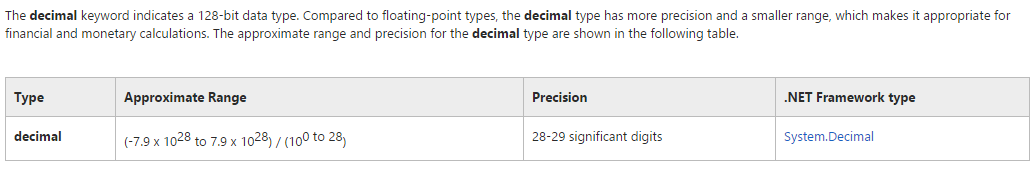
## 变量的范围，常量表，verbatim，类型转换，操作符的优先级



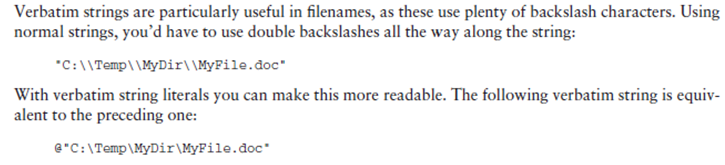
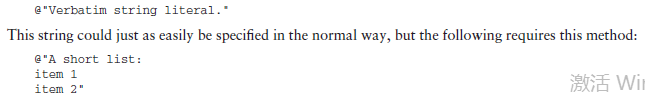
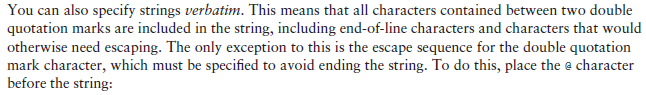
char在内存中其实是以integer的方式存储，每一个0-65535范围内的integer都有他对应的char，所以char比较大小也是比较他的integer的大小，也是因为这样char才能转型成int。



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

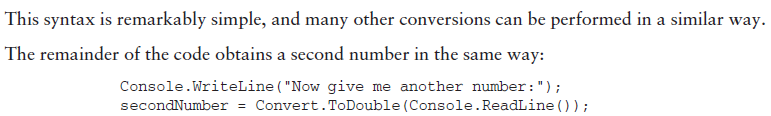


注意decimal，占128bit，他比double精度更高，适合金融或者货币这种高精度计算



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();

}

}

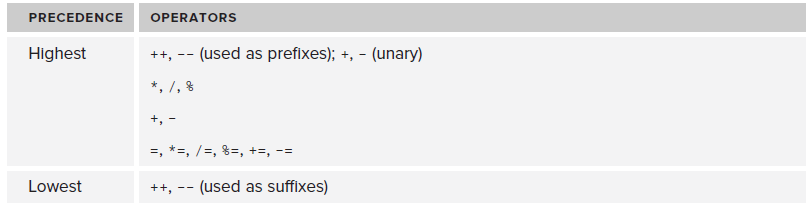
此时sourceVar超过了byte的最大范围，而编译器并不会给出任何警告，打印出了错误的结果destinationVar is 25。如果想让编译器在运行的时候给出警告，则可以使用checked的关键字。如下所示destinationVar = checked((byte)sourceVar)

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。。

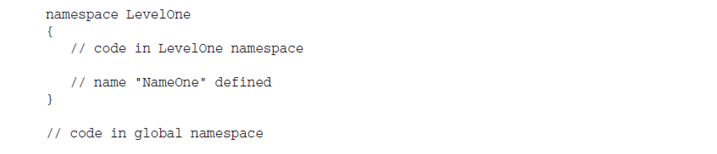


操作符的优先级

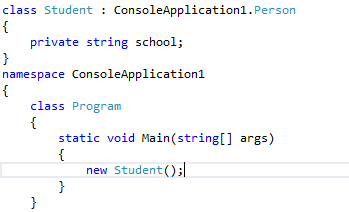


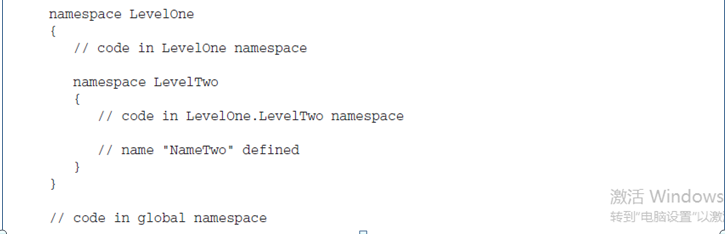
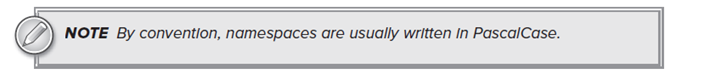
可以用括号来改变优先级

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

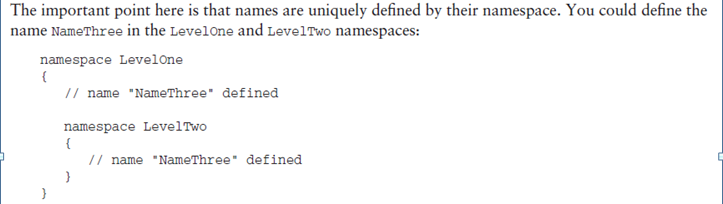


没有在namespace里定义的代码都在global namespace里

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

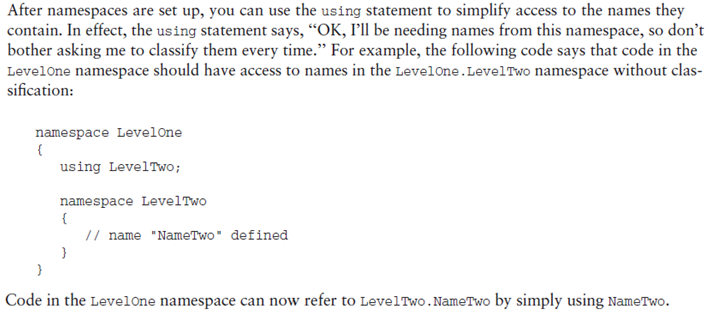


Namespace可以嵌套，在LevelOne中访问LevelTwo，要用LevelTwo.NameTwo，在global中访问则要用LevelOne.LevelTwo.NameTwo

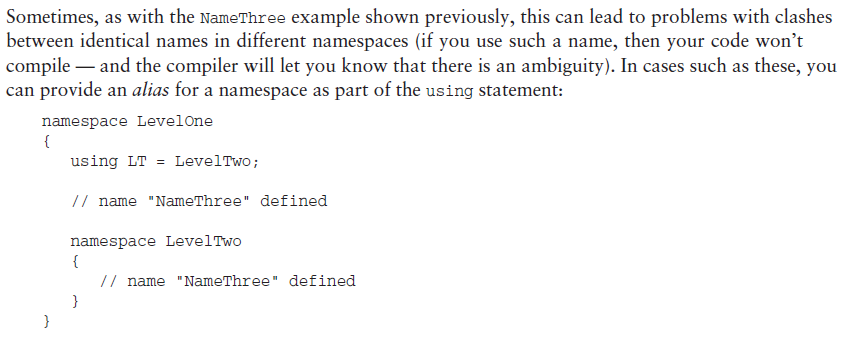


两个NameThree是不同的

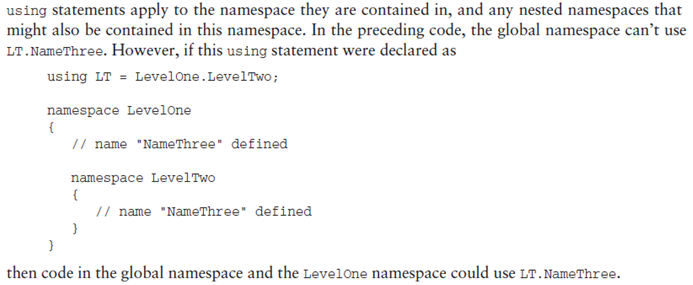
using keyword



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

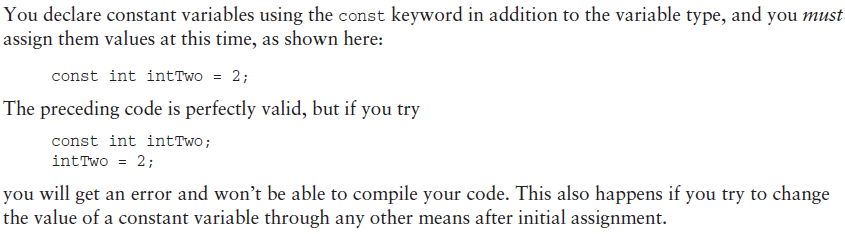


这样做会出现一个问题，看上面程序，LevelOne和LevelTwo里都定义了一个NameThree，如果在LevelOne里用了using LevelTwo，这样两个NameThree就无法区分开来，这时可以给LevelTwo取一个别名，using LT=LevelTwo

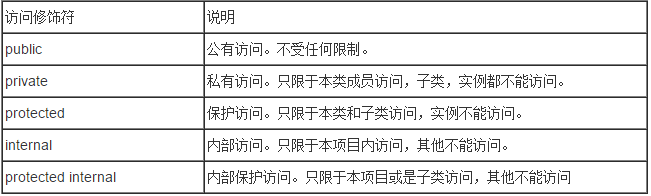


注意，using 影响当前namespace和当前namespace嵌套的namespace，比如上面代码，在global和LevelOne中均可以用LT.Name来代表LevelOne.LevelTwo.NameThree

const keyword

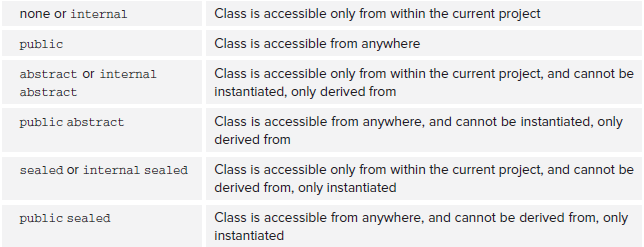


C#中没有final关键字，const类似于final关键字，但是只能变量，不能加在方法和类之前。那么怎么能规定一个类不能被继承，或者一个方法不能被重写呢？下面就会讲到，先来总结一下c#中的访问修饰符。



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

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。

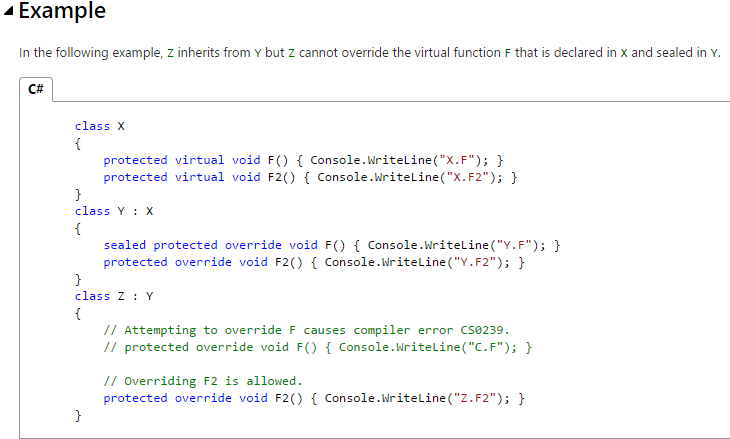
刚才说到防止class被继承，c#中是用sealed关键字，sealed关键字在方法重写中还有一个用途，稍后会讲解

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.



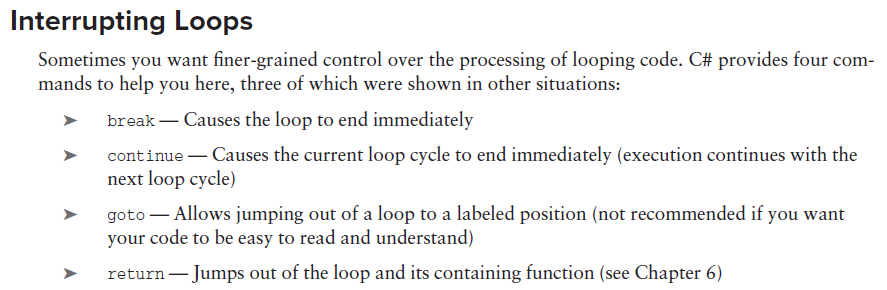
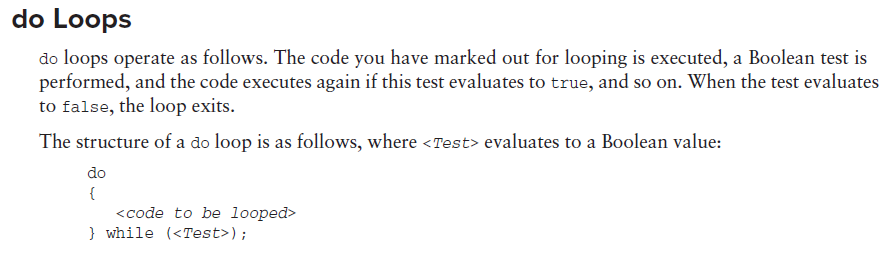
When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as [virtual](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx).

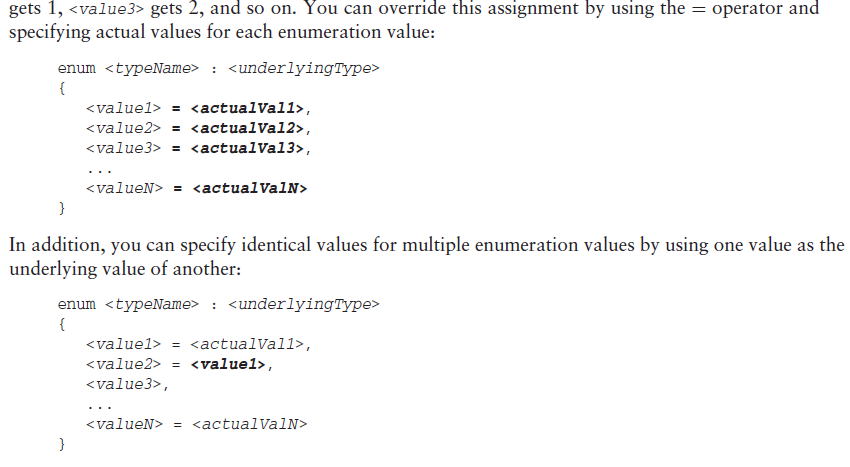
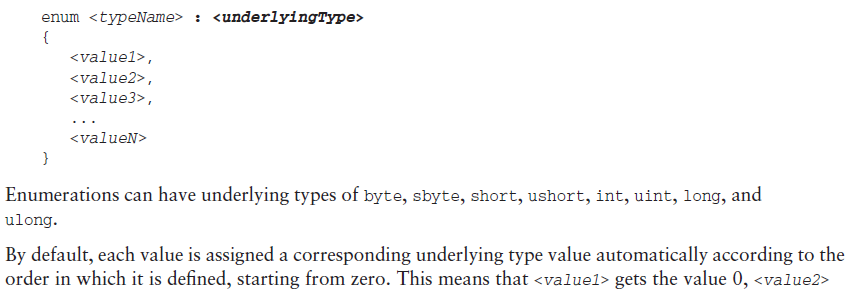
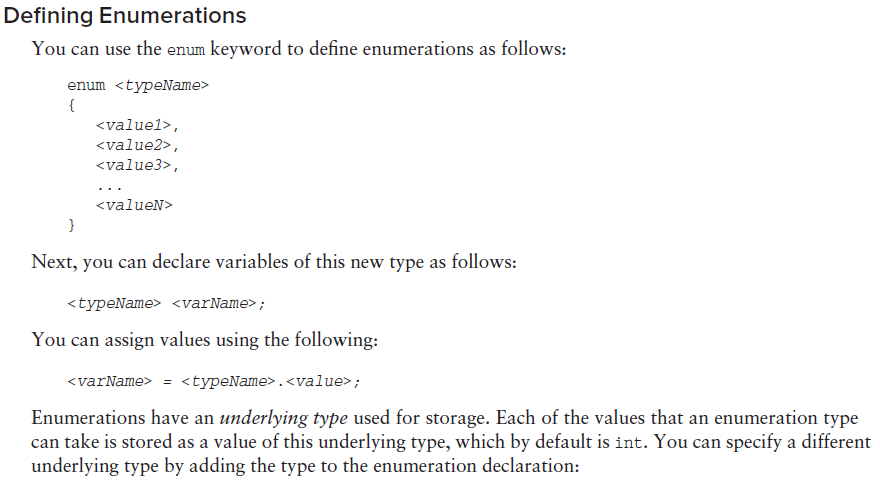
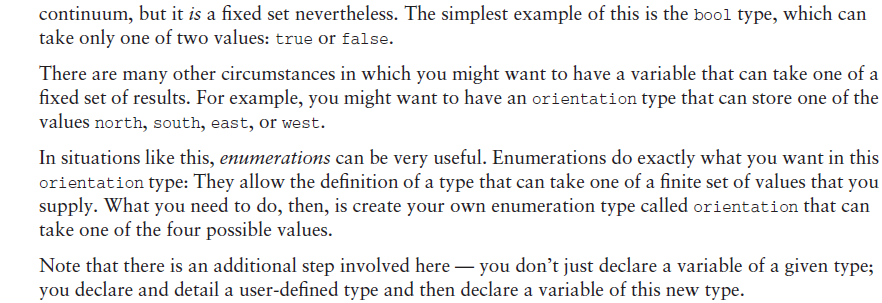
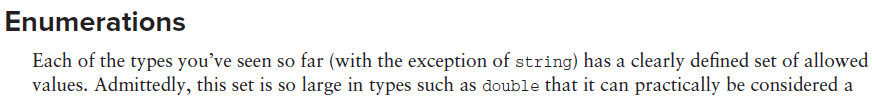
It is an error to use the [abstract](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx) modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

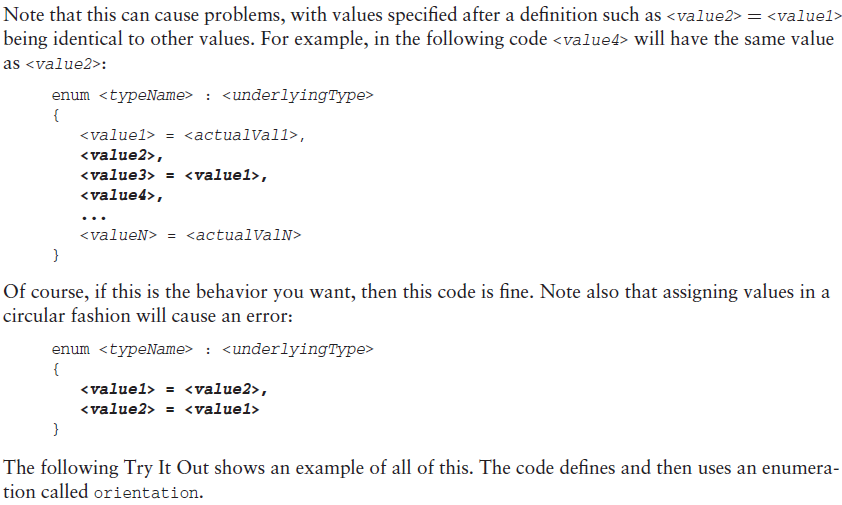
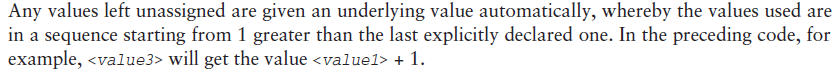
When applied to a method or property, the **sealed** modifier must always be used with [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx).（虽然例子中sealed写在方法声明的最前面，但其实也可以写在protected的后面，推荐写在后面，这样和sealed用在class中的位置相同）

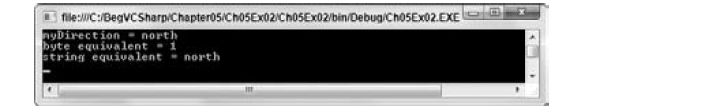
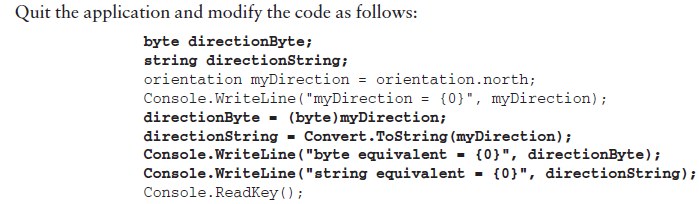
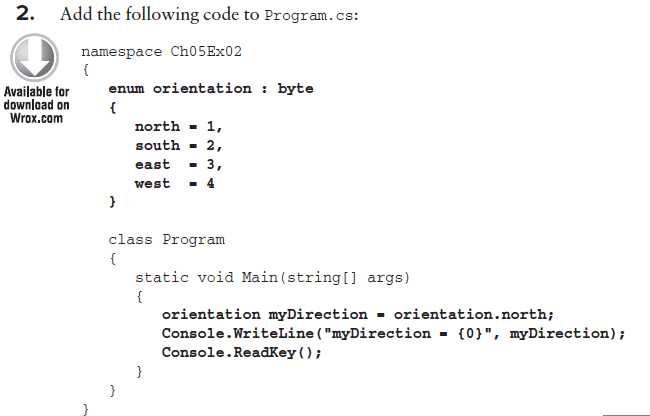
Because structs are implicitly sealed, they cannot be inherited.

## 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();

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。

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

**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

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.

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;

## 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).