

# Chapter 2. C# Language Basics

## Type Basics

All C# types fall into the following categories:

1. Value types
2. Reference types
3. Generic type parameters
4. Pointer types

## Value Types

- Numeric

Predefined

- Signed integer ( `sbyte`, `short`, `int`, `long` )
- Unsigned integer ( `byte`, `ushort`, `uint`, `ulong` )
- Real number ( `float`, `double`, `decimal` )

Custom

- `enum`
- `struct`

### 1. Numeric suffixes

Category	C# type	Example
F	float	<code>float f = 1.0F;</code>
D	double	<code>double d = 1D;</code>
M	decimal	<code>decimal d = 1.0M;</code>
U	uint	<code>uint i = 1U;</code>
L	long	<code>long i = 1L;</code>
UL	ulong	<code>ulong i = 1UL;</code>

### 2. Range

Table 2-1. Predefined numeric types in C#

C# type	System type	Suffix	Size	Range
<b>Integral — signed</b>				
sbyte	SByte		8 bits	$-2^7$ to $2^7-1$
short	Int16		16 bits	$-2^{15}$ to $2^{15}-1$
int	Int32		32 bits	$-2^{31}$ to $2^{31}-1$
long	Int64	L	64 bits	$-2^{63}$ to $2^{63}-1$
<b>Integral — unsigned</b>				
byte	Byte		8 bits	0 to $2^8-1$
ushort	UInt16		16 bits	0 to $2^{16}-1$
uint	UInt32	U	32 bits	0 to $2^{32}-1$
ulong	UInt64	UL	64 bits	0 to $2^{64}-1$
<b>Real</b>				
float	Single	F	32 bits	$\pm (\sim 10^{-45}$ to $10^{38})$
double	Double	D	64 bits	$\pm (\sim 10^{-324}$ to $10^{308})$
decimal	Decimal	M	128 bits	$\pm (\sim 10^{-28}$ to $10^{28})$

3.e.g.

十进制: `int x = 127`

十六进制: `long y = 0x7F`

二进制: `var b = 0b1010_1011_1100_1101_1110_1111;`

- Logical (`bool`)
- Character (`char`)

自定义一个值类型 (struct) 里面有两个int, 每个Int 4 byte(32 bit) 就消耗4 byte +4 byte = 8 byte  
自定义一个引用类型多消耗4或者8 byte (depends on 32 or 64 bit OS)

## Reference types

- `string`
- `object`
- `delegate`
- `interface`
- `array`
- `dynamic`

## Others

- Conversion

type conversions are *implicit* when the destination type can represent every possible value of the source type. Otherwise, an *explicit* conversion is required. For example

```
int x = 12345; // int is a 32-bit integer
long y = x; // Implicit conversion to 64-bit integral type
short z = (short)x; // Explicit conversion to 16-bit integral type
```

A `float` can be implicitly converted to a `double`, since a `double` can represent every possible value of a `float`. The reverse conversion must be explicit.

所有整型都可以隐式转换为 decimal，其他类型需要显式转换

All integral types may be implicitly converted to all floating-point

```
int i = 1;
float f = i;
//The reverse conversion must be explicit:
int i2 = (int)f;
```

- 可以在数字中任意位置加下划线
- 加减乘除运算符，8bit/16bit的 Numeric Type 不能用,(加减等)运算时自动隐式转换成 int
- Overflow
  - 溢出不会报异常
  - e.g.

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // True
```

- 检查溢出: `checked`

```
int a = 1000000;
int b = 1000000;
int c = checked (a * b); // Checks just the expression.
checked // Checks all expressions
{ // in statement block.
...
c = a * b;
...
}
```

- 

Type	Default value
All reference types	null
All numeric and enum types	0
char type	'\0'
bool type	false

## Variables and Parameters

A variable represents a storage location that has a modifiable value. A variable can be a **local variable**, **parameter** (value, ref, or out), **field** (instance or static), or **array element**.

local variables 需要先赋值再用，field 不需要，数组自动赋值为 0

## Parameters

传入值类型不改变原来的对象，传入引用类型改变引用的对象的值，但是将引用对象的变量设为null，其他引用此对象的变量不会也变成null

```
static void Foo(Point point)
{
    point.X = 10;           //will change the referenced object's value
    point = null;           //we lose a copy of reference 'point',but the
reference 'p1' outside still exist
}
static void Main()
{
    Point p1 = new Point();
    Foo(p1);
    Console.WriteLine(p1.X);
}
}
class Point
{
    public int X { get; set; } = 5;
}
```

使用 `ref` 关键字可以将外面的引用对象置为null

```
static void Foo(ref Point point)
{
    point.X = 10;
    point = null;
}
static void Main()
{
    Point p1 = new Point();
    Foo(ref p1);
    Console.WriteLine(p1.X);    //System.NullReferenceException: 'Object
reference not set to an instance of an object.'
}
}
class Point
{
    public int X { get; set; } = 5;
}
```

### The `ref` modifier

传入一个引用，定义的变量需要本来有值

```

class Test
{
    static void Foo (ref int p)
    {
        p = p + 1;           // Increment p by 1
        Console.WriteLine (p); // Write p to screen
    }

    static void Main()
    {
        int x = 8;
        Foo (ref x);        // Ask Foo to deal directly with x
        Console.WriteLine (x); // x is now 9
    }
}

```

### The **out** modifier

传入一个变量的引用，在函数里改变这个变量，用来变相return多个值。本来这个变量可以指向null，但是传入某个函数之后必须要给他赋值。

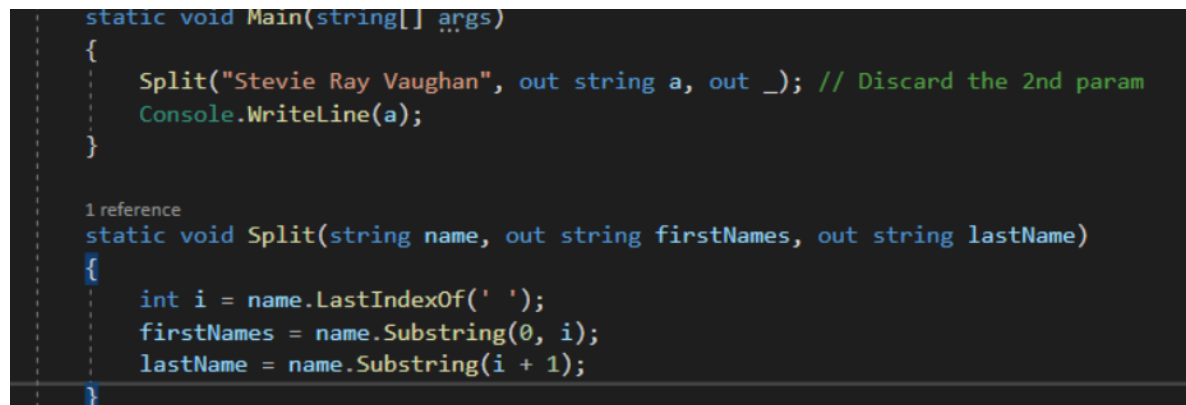
```

class Test
{
    static void Split (string name, out string firstNames,
                      out string lastName)
    {
        int i = name.LastIndexOf (' ');
        firstNames = name.Substring (0, i);
        lastName    = name.Substring (i + 1);
    }

    static void Main()
    {
        string a, b;
        Split ("Stevie Ray Vaughan", out a, out b);
        Console.WriteLine (a);           // Stevie Ray
        Console.WriteLine (b);           // Vaughan
    }
}

```

可以在调用函数的时候声明要out的变量的类型（就不需要提前定义好指向null的变量），可以用 **out** 来diacard out变量。



```

static void Main(string[] args)
{
    Split("Stevie Ray Vaughan", out string a, out _); // Discard the 2nd param
    Console.WriteLine(a);
}

1 reference
static void Split(string name, out string firstNames, out string lastName)
{
    int i = name.LastIndexOf(' ');
    firstNames = name.Substring(0, i);
    lastName = name.Substring(i + 1);
}

```

### The **params** modifier

作为方法的最后一个参数，这样方法就可以接受任意个数的参数，可以将 `params` 看作一个普通的数组。

```
class Test
{
    static int Sum (params int[] ints)
    {
        int sum = 0;
        for (int i = 0; i < ints.Length; i++)
            sum += ints[i];           // Increase sum by ints[i]
        return sum;
    }

    static void Main()
    {
        int total = Sum (1, 2, 3, 4);
        Console.WriteLine (total);    // 10
    }
}
```

## Optional parameters

定义函数的时候可以给参数赋默认值，变为 **optional** 参数，默认值必须是一个 **constant expression**，不能用 **ref** 或 **out** 修饰

Mandatory parameters must occur *before* optional parameters in both the method declaration and the method call (the exception is with `params` arguments, which still always come last).

```
void Foo (int x = 0, int y = 0) { Console.WriteLine (x + ", " + y); }

void Test()
{
    Foo(1);    // 1, 0
}
```

## Named arguments

用冒号指定赋给哪个参数，`named arguments` 必须放在 `positional arguments` 之后，调用的时候参数的顺序可以任意

```
void Foo (int x, int y) { Console.WriteLine (x + ", " + y); }

void Test()
{
    Foo (x:1, y:2); // 1, 2
}
```

# Null Operators

## Null Coalescing Operator: 双问号

双引号左边的如果是 `null`，使用右边的值

```
string s1 = null;
string s2 = s1 ?? "nothing"; // s2 evaluates to "nothing"
```

## Null-conditional Operator (C# 6): 单问号加点

如果单引号加点左边是null，表达式为Null而不是throw 一个NullReferenceException

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();    // No error; s instead evaluates to null
```

```
someObject?.SomeVoidMethod();    //If someObject is null, this becomes a “no-
operation” rather than throwing a NullReferenceException.
```

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString() ?? "nothing";    // s evaluates to "nothing"
```

## Statements

---

### Local variables

```
static void Main()
{
    int x;
    {
        int y;
        int x;            // Error - x already defined
    }
    {
        int y;            // OK - y not in scope
    }
    Console.Write (y);    // Error - y is out of scope
}
```

### Selection Statements

- 1.Selection statements (if, switch)
- 2.Conditional operator (?:)
- 3.Loop statements (while, do..while, for, foreach)

### The switch statement

```
static void ShowCard (int cardNumber)
{
    switch (cardNumber)
    {
        case 13:
            Console.WriteLine ("King");
            break;
        case 12:
            Console.WriteLine ("Queen");
            break;
        case 11:
            Console.WriteLine ("Jack");
            break;
        case -1:
            goto case 12;            // Joker is -1
                                     // In this game joker counts as queen
        default:
            Console.WriteLine (cardNumber);    // Executes for any other cardNumber
            break;
    }
}
```

### The switch statement with patterns (C# 7)

```

static void Main()
{
    TellMeTheType (12);
    TellMeTheType ("hello");
    TellMeTheType (true);
}

static void TellMeTheType (object x)    // object allows any type.
{
    switch (x)
    {
        case int i:
            Console.WriteLine ("It's an int!");
            Console.WriteLine ($"The square of {i} is {i * i}");
            break;
        case string s:
            Console.WriteLine ("It's a string");
            Console.WriteLine ($"The length of {s} is {s.Length}");
            break;
        default:
            Console.WriteLine ("I don't know what x is");
            break;
    }
}

```

You can predicate a case with the `when` keyword:

```

switch (x)
{
    case bool b when b == true:    // Fires only when b is true
        Console.WriteLine ("True!");
        break;
    case bool b:
        Console.WriteLine ("False!");
        break;
}

```

## Namespaces

---

两个等价的namespace语法

The `namespace` keyword defines a namespace for types within that block. For example:

```

namespace Outer.Middle.Inner
{
    class Class1 {}
    class Class2 {}
}

```

The dots in the namespace indicate a hierarchy of nested namespaces. The code that follows is semantically identical to the preceding example:

```

namespace Outer
{
    namespace Middle
    {
        namespace Inner
        {
            class Class1 {}
            class Class2 {}
        }
    }
}

```

The `using` Directive



```
using Outer.Middle.Inner;
class Test {
static void Main(){
Class1 c;    // Don't need fully qualified name
    }
}
```

#### using static (C# 6)

```
using static System.Console;

class Test
{
    static void Main() { WriteLine ("Hello"); }
}
```

### Rules Within a Namespace

1. Name scoping : 内层namespace的class可以用外层的namespace不需要qualification
2. Name hiding : If the same type name appears in both an inner and an outer namespace, the inner name wins. To refer to the type in the outer namespace, you must qualify its name.
3. Repeated namespaces : You can repeat a namespace declaration, as long as the type names within the namespaces don't conflict, 哪怕他们在两个不同的source file
4. Nested using directive : 可以在namespace里包含using, 这样被using的namespace就在本namespace里可见。

```
namespace N1
{
    class Class1 {}
}

namespace N2
{
    using N1;

    class Class2 : Class1 {}
}

namespace N2
{
    class Class3 : Class1 {}    // Compile-time error
}
```

### Aliasing Types and Namespaces

可以引用一个namespace里的一部分, 可以给他一个alias

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

### Advanced Namespace Feature

1. extern: 来自两个assemblies的一个namespace含有两个同名的type

2.Namespace alias qualifiers: 内层namespace的names会隐藏外层的names, 有时候写全qualified name 也不行, 用global:: P80

## Chapter 3 Creating Types in C#

### Class

最简形式

```
class YourClassName
{
}
```

在class关键词之前, 之后和大括号里的关键词:

Preceding the keyword <code>class</code>	<i>Attributes and class modifiers. The non-nested class modifiers are <code>public</code>, <code>internal</code>, <code>abstract</code>, <code>sealed</code>, <code>static</code>, <code>unsafe</code>, and <code>partial</code></i>
Following <code>YourClassName</code>	<i>Generic type parameters, a base class, and interfaces</i>
Within the braces	<i>Class members (these are <code>methods</code>, <code>properties</code>, <code>indexers</code>, <code>events</code>, <code>fields</code>, <code>constructors</code>, <code>overloaded operators</code>, <code>nested types</code>, and a <code>finalizer</code>)</i>

### Fields

e.g,

e.g.

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

- *readonly* modifier: 阻止一个field在创建后被修改, 只能在定义field或者构造函数内assign field
- Field initialization is optional

### Methods

- A method's signature must be unique within the type. A method's signature comprises its name and parameter types in order (but not the parameter names, nor the return type).
- Expression-bodied methods:

```
int Foo (int x) { return x * 2; }
int Foo (int x) => x * 2;
```

## Local Method (C# 7.0)

e.g.

```
void WriteCubes()
{
    Console.WriteLine (Cube (3));
    Console.WriteLine (Cube (4));
    Console.WriteLine (Cube (5));

    int Cube (int value) => value * value * value;
}
```

The local method (`Cube`, in this case) is visible only to the enclosing method (`WriteCubes`), 需要在外层函数调用内层函数

- Local methods can appear inside other function kinds, such as property accessors, constructors
- The `static` modifier is invalid for local methods. They are implicitly static if the enclosing method is static.

## Constructor

- 没有返回值，名字和enclosing type一样
- 可以用 expression-bodied members: e.g.

```
public Panda (string n) => name = n;
```

- **Overloading constructors:**

- 一个constructor可以call另一个，用 `this` 关键字，其中price还可以是一个表达式。

```
using System;

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

- C#自动生成一个无参数的constructor,但是一旦自己定义了constructor,就不自动生成
- Field initializations occur before the constructor is executed, and in the declaration order of the fields.
- constructor不一定要public的，可以设为不设置为public后用一个类里的static method来生成 object, e.g.

```
public class Class1
{
    Class1() {} // Private constructor
    public static Class1 Create (...)
    {
        // Perform custom logic here to return an instance of Class1
        ...
    }
}
```

## Deconstructors (C# 7)

原本给构造函数参数赋值创建对象，此时将赋的值返回

*A deconstruction method must be called Deconstruct, and have one or more out parameters, such as in the following class*

```
class Rectangle
{
    public readonly float Width, Height;

    public Rectangle (float width, float height)
    {
        Width = width;
        Height = height;
    }

    public void Deconstruct (out float width, out float height)
    {
        width = Width;
        height = Height;
    }
}
```

To call the deconstructor, we use the following special syntax:

```
var rect = new Rectangle (3, 4);
(float width, float height) = rect;           // Deconstruction
Console.WriteLine (width + " " + height);     // 3 4
```

第二行相当于：

```
rect.Deconstructor (out var width, out var height );
```

## Object Initializers

要么用initializers直接给field等赋值，要么用constructor

Using object initializers, you can instantiate Bunny objects as follows:

```
// Note parameterless constructors can omit empty parentheses
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo")          { LikesCarrots=true, LikesHumans=false };
```

The code to construct b1 and b2 is precisely equivalent to:

```
Bunny temp1 = new Bunny();           // temp1 is a compiler-generated name
temp1.Name = "Bo";
temp1.LikesCarrots = true;
```

## The `this` Reference

1.用来引用这个对象本身：P88底

2.用来避免参数和自身的properties/field同名

## Properties

---

`get / set` 作为properties accessors，有一个隐含的参数value

### Expression-bodied properties (C# 6, C# 7)

e.g.1 一个readonly Property(相当于只有get)

```
public decimal Worth => currentPrice * sharesOwned;
```

e.g.2 一个既可以读又可以写的property

```
public decimal Worth
{
    get => currentPrice * sharesOwned;
    set => sharesOwned = value / currentPrice;
}
```

自动属性：相当于在field后面加{get;set}

### Property initializers (C# 6)

e.g.

```
public decimal CurrentPrice { get; set; } = 123;
```

可以只含get，这样相当于创建了一个immutable (read-only) types

### get and set accessibility

get和set可以有不同的accessibility

e.g.

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

## Indexers

---

像string有indexers可以访问里面的每个char，通常用来访问一个class/type里面的list/dictionary value.

To write an indexer, define a property called **this**, specifying the arguments in square brackets. For instance:参数列表用方括号而不像普通的函数一样用圆括号

```

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

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

```

Here's how we could use this indexer:

```

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

```

如果不用set只要获取,可以用expression-body syntax

```

public string this [int wordNum] => words [wordNum];

```

## Constants:P93

---

是一个static field, 值永远不会变

A constant is declared with the `const` keyword and must be initialized with a value.

```

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

```

Constants can also be declared local to a method.

```

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

    ...
}

```

## Static Constructors

---

1.A static constructor executes once per type, rather than once per instance. A type can define only one static constructor, and it must be parameterless and have the same name as the type

2.触发static constructor:实例化这个type/访问这个type里面的静态成员

3.如果有静态field initializer, Static field initializers run just before the static constructor is called

# Static Classes

---

静态类只能含有静态成员，例如 `System.Console` and `System.Math` class

## Finalizers

---

1. Finalizers are class-only methods that execute before the garbage collector reclaims the memory for an unreferenced object. The syntax for a finalizer is the name of the class prefixed with the `~` symbol:

2. From C# 7, single-statement finalizers can be written with expression-bodied syntax:

```
~Class1() => Console.WriteLine ("Finalizing");
```

## Partial Types and Methods:P96

---

1. allow type definition to be split — typically across multiple files. *A common scenario is for a partial class to be auto-generated from some other source (such as a Visual Studio template or designer), and for that class to be augmented with additional hand-authored methods.* For example:

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }

// PaymentForm.cs - hand-authored
partial class PaymentForm { ... }
```

2. 每个partial type's participant must have the partial declaration， 下面的是错误的：

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

3. Participants cannot have conflicting members. e.g. A constructor with the same parameters  
4. each participant can independently specify interfaces to implement.  
5. The compiler makes no guarantees with regard to field initialization order between partial type declarations.

## Partial methods

Partial methods must be **void** and are implicitly **private**.

一般用在一部分是自动生成的，一部分是手写的情况

1. 含两部分：**definition, implemention**

e.g.

```

partial class PaymentForm    // In auto-generated file
{
    ...
    partial void ValidatePayment (decimal amount);
}

partial class PaymentForm    // In hand-authored file
{
    ...
    partial void ValidatePayment (decimal amount)
    {
        if (amount > 100)
            ...
    }
}

```

## Polymorphism

---

1. This means a variable of type x can refer to an object that subclasses x.
2. 一个方法的参数是父类，可以传给他子类，参数是子类传给他父类不行(subclass is more competent, can be used in scenarios require the less competent one)

## Casting and Reference Conversions

an object reference can be 隐式转换成一个父类 (**up cast**), Explicitly **downcast** to a subclass reference

### Upcasting

e.g.

An upcast operation creates a base class reference from a subclass reference. For example:

```

Stock msft = new Stock();
Asset a = msft;           // Upcast

```

```

Console.WriteLine (a == msft); // True

```

尽管引用a指向(引用)了一个子类对象，不能完全当成子类来用，因为a的reference type是父类,即使指向的object是子类，只能当作父类来用，但是可以之后成功的downcast:

e.g.

```

Console.WriteLine (a.Name);           // OK
Console.WriteLine (a.SharesOwned);    // Error: SharesOwned undefined

```

The last line generates a compile-time error because the variable a is of type Asset, even though it refers to an object of type Stock. To get to its SharesOwned field, you must *downcast* the Asset to a Stock.

### Downcasting

A downcast operation **creates a subclass reference from a base class reference**.

e.g.



```

Stock msft = new Stock();
Asset a = msft;                                // Upcast
Stock s = (Stock)a;                          // Downcast
Console.WriteLine (s.SharesOwned);             // <No error>
Console.WriteLine (s == a);                    // True
Console.WriteLine (s == msft);                 // True

```

子类的引用赋值给父类的引用，然后再downcast给另一个子类引用不可以。即Downcast要求原先是那个子类，被upcast到父类之后可以downcast回来

```

0 references
static void Main(string[] args)
{
    Asset asset = new Asset();
    Stock stock = (Stock)asset;
    Console.WriteLine(stock.sharehold); //Compile error:Unable to cast
}

```

其中第四行输出为默认初始值0

If a downcast fails, an InvalidCastException is thrown

## The **as** operator

The **as** operator performs a downcast that evaluates to **null** (rather than throwing an exception) if the downcast fails.

```

Asset a = new Asset();
Stock s = a as Stock;           // s is null; no exception thrown

```

## The **is** operator

The **is** operator **tests whether a reference conversion would succeed**; in other words, whether an object derives from a specified class (or implements an interface). It is often used to test before downcasting.

```

Stock s1 = new Stock();
s1.SharesOwned = 1000;
Asset a1 = s1;           // up cast
Stock s2 = (Stock)s1;    //downcast
WriteLine(s2.SharesOwned); // >1000. after downcasted back the property is still
WriteLine(s2 is Stock);   //True
WriteLine(s2 is Asset);   //True

```

The **is** operator and pattern variables (C# 7):P100

From C# 7, you can introduce a variable while using the `is` operator:

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

This is equivalent to:

```
Stock s;
if (a is Stock)
{
    s = (Stock) a;
    Console.WriteLine (s.SharesOwned);
}
```

## Virtual Function Members

- 定义格式如 `public virtual decimal xxx`, 定义了可以被子类用 `override` 关键词改写, 用来实现更加具体的功能。改写了之后不仅子类的这个成员值改变, 其被 `upcast` 之后的指向父类的引用的这个值也改变
- Methods, properties, indexers, and events can all be declared, Field can not
- The signatures, return types, and accessibility of the virtual and overridden methods must be identical. An overridden method can call its base class implementation via the `base` keyword

e.g.

```
public class Asset
{
    public string Name;
    public virtual decimal Liability => 0;    // Expression-bodied property
}

public class Stock : Asset
{
    public long SharesOwned;
}

public class House : Asset
{
    public decimal Mortgage;
    public override decimal Liability => Mortgage;
}

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

## Abstract Classes and Abstract Members

1. 可以定义抽象类, 只有抽象类里可以定义抽象成员(一般的类不能包含抽象方法等), 只有这些抽象成员都被 `override` 之后这个类才可以被实例化。(抽象类里只可以定义抽象函数, 属性等可以是非

抽象的，但是也可以被override。)Abstract members are like virtual members, except they don't provide a default implementation.

2. abstract methods have no actual code in them, and subclasses HAVE TO override the method. Virtual methods can have code, which is usually a default implementation of something, and any subclasses CAN override the method using the override modifier and provide a custom implementation.

## Hiding Inherited Members( `new` keyword)

1. B是A的子类，A,B类里面有一样的成员C则：References to A (at compile time) bind to A.C  
References to B (at compile time) bind to B.C编译器会产生一个warning，可以用new关键词 suppress warning
2. new versus override:P102底当一个override的对象up cast到父类的时候，这个对象实现的是override之后的功能，当一个new 对象upcast到父类的时候，这个对象实现的是之前父类里的功能。

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

public class Overrider : BaseClass
{
    public override void Foo() { Console.WriteLine ("Overrider.Foo"); }
}

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

The differences in behavior between Overrider and Hider are demonstrated in the following code:

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

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

## Sealing Functions and Classes

1. 可以seal一个方法，seal一个类(seal类里的所有virtual function),In our earlier virtual function member example, we could have sealed `House`'s implementation of `Liability`, preventing a class that derives from `House` from overriding `Liability`, as follows:

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

2. Although you can seal against overriding, you can't seal a member against being hidden.

## The `base` Keyword

1. Accessing an overridden function member from the subclass.

In this example, House uses the `base` keyword to access Asset's implementation of Liability:

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

现在访问的就不是被virtual过的，父类的 `Liability` 属性，同样也可以用来访问被hidden(用new)的成员，但是被new的成员也可以用直接cast到父类的方法来访问

2. 可以用来Calling a base-class constructor.

## Constructors and Inheritance

如果子类没写**constructor**,不会自动继承父类的**constructor**，父类的无参构造函数总是默认被调用

1. 父类的构造方法不自动被继承，可以用base来访问

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

public class Subclass : Baseclass { }
```

the following is illegal:

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

Subclass must hence “redefine” any constructors it wants to expose. In doing so, however, it can call any of the base class's constructors with the `base` keyword:

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

Base-class constructors always execute first; this ensures that *base* initialization occurs before *specialized* initialization.

2. If a constructor in a subclass omits the base keyword, the base type's parameterless constructor is implicitly called

e.g.

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

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

If the base class has no accessible parameterless constructor, subclasses are forced to use the `base` keyword in their constructors.

## Constructor and field initialization order

When an object is instantiated, initialization takes place in the following order:

1. From subclass to base class:
  - a. Fields are initialized.
  - b. Arguments to base-class constructor calls are evaluated.
2. From base class to subclass:
  - a. Constructor bodies execute.

The following code demonstrates:

```
public class B
{
    int x = 1;           // Executes 3rd
    public B (int x)
    {
        ...             // Executes 4th
    }
}
public class D : B
{
    int y = 1;           // Executes 1st
    public D (int x)
        : base (x + 1)   // Executes 2nd
    {
        ...             // Executes 5th
    }
}
```

子类class D构造函数接收到一个参数x时，调用父类的构造函数，把x+1作为参数。

## Overloading and Resolution: P106

两个函数同名且一个的参数是另一个的子类，When an overload is called, the most specific type has precedence

e.g.

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

When an overload is called, the most specific type has precedence:

```
House h = new House (...);
Foo(h);                               // Calls Foo(House)
```

The particular overload to call is determined statically (at compile time) rather than at runtime. The following code calls `Foo(Asset)`, even though the runtime type of `a` is `House`:

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