一周学会C#（前言）1

C#才鸟(QQ:249178521)

大家好！C#作为微软在 21 世纪推出的新语言，它有着其他语言无法比拟的优势。但如何在短时间内迅速掌握它，却是一个比较难的问题。但如果你看完这个教程后，你一定会理解并掌握 C#。

这个教程共分六个部分，今天先介绍 C#中比较基本的概念。

1.总体框架

Hiker.cs 类名不一定等于文件名

```
using System; //每一个程序必须在开头使用这一语句
public sealed class HitchHiker
{
    public static void Main()//程序从 Main 开始执行
    {
        int result;
        result = 9 * 6;
        int thirteen;
        thirteen = 13;
        Console.Write(result / thirteen); //输出函数
        Console.Write(result % thirteen);
    }
}
//上面各语句的具体用法以后会介绍
/*  这个程序用来
 *  演示 C#的总体框架
 */
```

注意：上面的程序中，符号//表示注释，在//后面的同一行上的内容是注释；
/*和*/ 这间的内容都是注释

你可以在 windows 的命令行提示符下键入：**csc Hiker.cs**
进行编译产生可执行文件 **Hiker.exe**
然后在 windows 的命令行提示符下键入：**Hiker**，你就可以看到在屏幕上显视 42
(注：你必须装有.net framework)

和 Java 不一样，C#源文件名不一定要和 C#源文件中包含的类名相同。C#对大小写敏感，所以 Main 的首字母为大写的 M(这一点大家要注意，尤其是熟悉 C 语言的朋友)。
你可以定义一个返回值为 int 的 Main 函数，当返回值为 0 时表示成功：
public static int Main() { ... return 0; }
你也可以定义 Main 函数的返回值为 void：
public static void Main() { ... }

你还可以定义 Main 函数接收一个 string 数组：

```
public static void Main(string[] args)
{
foreach (string args in args) {
System.Console.WriteLine(arg);
}
}
```

程序中的 Main 函数必须为 static。

2.标识符

标识符起名的规则：

局部变量、局部常量、非公有实例域、函数参数使用 camelCase 规则；其他类型的标识符使用 PascalCase 规则。

privateStyle ——— camelCase 规则（第一个单词的首字母小写，其余单词的首字母大写）

PublicStyle ———— PascalCase 规则（所有单词的首字母大写）

尽量不要使用缩写。

Message，而不要使用 msg。

不要使用匈牙利命名法。

```
public sealed class GrammarHelper     /* sealed  表明该类不能被继承*/
{    ...
        public QualifiedSymbol Optional(AnySymbol symbol)
        { ... }
        private AnyMultiplicity optional =
        new OptionalMultiplicity();
}
```

3.关键字

C#中 76 个关键字：

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

5 个在某些情况下是关键字：

get　　set　　value　　add　　remove

C#中有 76 个在任何情况下都有固定意思的关键字。另外还有 5 个在特定情况下才有固定意思的标识符。例如，*value* 能用来作为变量名，但有一种情况例外，那就是它用作属性/索引器的 set 语句的时候是一关键字。

但你可以在关键字前加@来使它可以用作变量名：

```
int @int = 42;
```

不过在一般情况下不要使用这种变量名。

你也可以使用@来产生跨越几行的字符串，这对于产生正则表达式非常有用。例如：

```
string pattern = @"
(                       # start the group
   abra(cad)?    # match abra and optional cad
)+";                    # one or more occurrences
```

如果你要在字符串中包含双引号，那你可以这样：

```
string quote = @""""quote"""";
```

4.标点符号

{ 和 } 组成语句块
分号表示一个语句的结束

```
using System;
public sealed class Hiker
{
    public static void Main()
    {
        int result;
        result = 9 * 6;
        int thirteen;
        thirteen = 13;
        Console.Write(result / thirteen);
        Console.Write(result % thirteen);
    }
}
```

一个 C#的“类/结构/枚举”的定义不需要一个终止的分号。

```
public sealed class Hiker
{
    ...
} // 没有;是正确的
```

然而你可以使用一个终止的分号，但对程序没有任何影响：

```
public sealed class Hiker
{
    ...
}; //有;是可以的但不推荐
```

在 Java 中，一个函数的定义中可以有一个结尾分号，但在 C#中是不允许的。

```
public sealed class Hiker
{
```

```
        public void Hitch() { ... }; //;是不正确的
    } //  没有;是正确的
```

总结：C#中类可以为分号，但方法的申明是不能有分号;

5.声明

声明是在一个块中引入变量

每个变量有一个标识符和一个类型

每个变量的类型不能被改变

```
using System;
public sealed class Hiker
{
    public static void Main()
    {
        int result;
        result = 9 * 6;
        int thirteen;
        thirteen = 13;
        Console.Write(result / thirteen);
        Console.Write(result % thirteen);
    }
}
```

这样声明一个变量是非法的：这个变量可能不会被用到。例如：

```
    if (...)
        int x = 42; //编译时出错
    else
        ...
```

6.表达式

表达式是用来计算的!

每个表达式产生一个值

每个表达式必须只有单边作用

每个变量只有被赋值后才能使用

```
using System;
public sealed class Hiker
{
    public static void Main()
    {
        int result;
        result = 9 * 6;
        int thirteen;
        thirteen = 13;
        Console.Write(result / thirteen);
        Console.Write(result % thirteen);
    }
}
```

C#不允许任何一个表达式读取变量的值，除非编译器知道这个变量已经被初始化或已经被赋值。例如，下面的语句会导致编译器错误：

```
int m;
if (...) {
        m = 42;
}
Console.WriteLine(m);// 编译器错误，因为 m 有可能不会被赋值
```

7.取值

类型	*取值*	*解释*
bool	**true false**	**布尔型**
float	**3.14**	**实型**
double	**3.1415**	**双精度型**
char	**'X'**	**字符型**
int	**9**	**整型**
string	**"Hello"**	**字符串**
object	**null**	**对象**

8.操作符

操作符	类型
+ - * / %（取余数）	**算术**
&& || ! ?:	**逻辑**
< <= > >=	**关系**
== !=	**相等**
=	**赋值**

9．编程风格

较正规的编程风格

在一个二元操作符的每一边都加一个空格
在每一个逗号后面而不是前面加一个空格
每一个关键字后面加一个空格
一行一个语句
分号前不要有空格
函数的园括号和参数之间不加空格
在一元操作符和操作数之间不加空格

在一个二元操作符的每一边都加一个空格：

```
Console.WriteLine("{0}", result / 13);    //推荐
Console.WriteLine("{0}", result/13);    //不推荐
```

在每一个逗号后面而不是前面加一个空格：

```
Console.WriteLine("{0}", result / 13); //推荐
Console.WriteLine("{0}",result / 13); //不推荐
```

每一个关键字后面加一个空格：

```
if (OneLine(comment)) ...      //推荐
if(OneLine(comment)) ...       //不推荐
```

分号前不要有空格：

```
Console.WriteLine("{0}", result / 13); //推荐
```

Console.WriteLine("{0}", result / 13) ; //不推荐

函数的园括号和参数之间不加空格：

if (OneLine(comment)) ... //推荐

if (OneLine(comment)) ... //不推荐

在一元操作符和操作数之间不加空格：

++keywordCount; //推荐

++ keywordCount; //不推荐

10.找错

```
bool checked;
...                                                    1

public static void main()
{ ... }                                               2

int matched = symbol.Match(input)                     3
if (matched > 0)
{
        ....
}
char optional = "?";
string theory = 'complex';                            4

int matched = 0_or_more(symbol);
...                                                                  5
```

第 1 段程序的错误：**checked** 是一个关键字

第 2 段程序的错误：不是 **main**，而是 **Main**

第 3 段程序的错误：变量声明语句没有分号

第 4 段程序的错误：字符值必须用单引号表示，字符串必须用双引号表示

第 5 段程序的错误：第一个错误是标识符不能以数字开头;第二个错误是不能用下划线作标识符。

一周学会C#（函数）2

C#才鸟（QQ:249178521）

1.前言

C#不支持全局函数
所有的函数必须在类内部声明
无源文件和头文件之分
所有的函数必须声明的时候被实现

```
int NotAllowed()          //错误，C#没有全局函数
{
                ...
}
sealed class Methods
{
      void Inline()
      { ...
      }
      void Error()
      { ...
      };                              //错误，函数不能有结尾分号
      int AlsoError();   //错误，函数必须声明的时候被实现
}
```

和 Java 一样，C#不允许有全局函数。所有的函数必须在类或结构内实现。函数是类或结构的成员，函数也被称为方法。

C#允许可以在类的声明中加入结尾分号，例如：

```
sealed class Methods
{
        ...
};//可以有结尾分号
```

但是，C#不允许在函数的声明中加入结尾分号，例如：

```
sealed class Methods
{
        void NotAllowed() {...} ; //错误，函数不能有结尾分号
}
```

2.声明函数

```
                        函数参数列表
            各参数以逗号隔开
            参数必须命名
            没有参数时括号不能省略
sealed class Methods
{
void Error(float) //错误，参数没有命名
      { ...
      }
      void NoError(float delta)
      { ...
      }

 int Error(void) //错误，无参数时不允许使用 void
      { ...
      }
      int NoError()
```

```
    { ...
}
}
```

3. 值型参数

一般的函数参数是实参的一个拷贝
实参必须预先被赋值
实参可以是常量类型

```
sealed class ParameterPassing
{
    static void Method(int parameter)
    {
        parameter = 42;
    }
    static void Main()
    {
        int arg = 0;
        Console.Write(arg); //结果为 0
        Method(arg);
        Console.Write(arg); //结果为 0
    }
}
```

（注：为了叙述的方便，以后所出现的“参数”这个词均指函数参数，也就是所谓的形参）

没有被 ref 或 out 修饰的函数参数是一个值型参数。值型参数只有在该参数所属的函数被调用的时候才存在，并且用调用时所传递的实参的值来进行初始化。当函数调用结束时，值型参数不复存在。

只有被预先赋值的实参才能被传递给值型参数，例如：

```
int arg;      // arg 没有被赋初值
Method(arg);//错误，实参必须预先赋初值
```

传递给函数的实参可以是纯粹的数而不是变量，例如：

```
Method(42);
Method(21 + 21);
```

4.引用型参数(ref、out)

引用型参数是实参的一个别名
没有发生复制
实参必须预先被赋值
实参必须是一个变量类型
实参和函数参数都要有 ref

```
sealed class ParameterPassing
{
    static void Method(ref int parameter)
    {
        parameter = 42;
    }
```

```
    static void Main()
    {
        int arg = 0;
        Console.Write(arg); //结果为 0
        Method(ref arg);
        Console.Write(arg); //结果为 42
    }
}
```

函数参数有 ref 修饰符时，被称为引用型参数。引用型参数不产生新的存储区间。实际上，引用型参数是函数调用时所传递的实参所代表的变量的别名。结果是引用型参数只是实参所代表的变量的另一个名字。

ref 修饰符必须同时出现在函数声明语句和函数调用语句中。

只有被预先赋值的实参才能被传递给引用型参数，例如：

```
int arg;      // arg 没有被赋初值
Method(ref arg);//错误，实参必须预先赋初值
```

传递给引用型参数的实参必须是变量类型，而不能是纯粹的值或常量。

```
Method(ref 42);   //错误，引用型参数的实参不能是纯粹的值
const int arg = 42;
Method(ref arg); //错误，引用型参数的实参不能是常量
```

5.out 型参数

out 型参数是实参的一个别名

没有发生复制

实参不必预先赋值

实参必须是变量类型

函数参数必须被预先赋值才能使用

实参和函数参数都要有 out

```
sealed class ParameterPassing
{
    static void Method(out int parameter)
    {
        parameter = 42;
    }
    static void Main()
    {
        int arg;
        //Console.Write(arg);
        Method(out arg);
        Console.Write(arg); //结果为 42
    }
}
```

函数参数有 out 修饰符时，被称为 out 型参数。out 型参数不产生新的存储区间。实际上，out 型参数是函数调用时所传递的实参所代表的变量的别名。结果是 out 型参数只是实参所代表的变量的另一个名字。

out 修饰符必须同时出现在函数声明语句和函数调用语句中。

没有被预先赋值的实参能够被传递给引用型参数，例如：

```
int arg;      // arg 没有被赋初值
Method(out arg);//正确，实参可以不赋初值
```

传递给 out 型参数的实参必须是变量类型，而不能是纯粹的值或常量。

```
Method(out 42);   //错误，out 型参数的实参不能是纯粹的值
const int arg = 42;
```

Method(out arg); //错误，out 型参数的实参不能是常量

总结：ref 、out

1、 都是地址传递，所以，实参不能是一个常量

2、 Ref 须要初始化，而 out 不须要. 即 ref 有进有出，out 是只出不进。

6.in 型参数?

***readonly*, *const* 和 *in*, 都是 C# 关键字**

它们不能被用于函数参数

***ref*/*out* 型参数总是被赋于写的权力**

7.函数重载

一个类中的函数可以有同一个名字，称为重载

函数名和参数称为标识

标识必须唯一

返回值类型不是标识

```
namespace System
{
    public sealed class Console
    {
        public static void WriteLine()
        { ... }
        public static void WriteLine(int value)
        { ... }
        public static void WriteLine(double value)
        { ... }
        ...
        public static void WriteLine(object value)
        { ... }
        ...
    }
}
```

和 C++与 Java 一样，C#允许一个类声明两个以上的同名函数，只要参数的类型或个数不同。这就是重载。但是，一个类不能包含标识为相同的实例函数和静态函数，例如：

```
sealed class Illegal
{
        void Overload() { ... }
        static void Overload() { ... }//错误
}
```

和 C++与 Java 一样，返回值的类型不是标识的一部分，不能被用作重载的标准，例如：

```
sealed class AlsoIllegal
{
        int Random() { ... }
        double Random() { ... }//错误
}
```

8.ref/out 重载

***ref / out* 在大部分情况下是标识的一部分!**

你可以重载一个 ref 型参数和一个普通参数
你可以重载一个 out 型参数和一个普通参数
你不可以重载一个 ref 型参数和一个 out 型参数

```
sealed class Overloading
{
    void Allowed(      int parameter)
    { ... }
    void Allowed(ref int parameter)
    { ... }
  //正确，重载一个 ref 型参数和一个普通参数

    void AlsoAllowed(      int parameter)
    { ... }
    void AlsoAllowed(out int parameter)
{ ... }
//正确，重载一个 out 型参数和一个普通参数

    void NotAllowed(ref int parameter)
    { ... }
    void NotAllowed(out int parameter)
{ ... }
//错误，不能重载一个 ref 型参数和一个 out 型参数
}
```

ref 和 out 修饰符可以是一个函数的标识。但是你不能同时重载 ref 和 out 型参数。ref 和 out 修饰符在某种意义上是“安全的“，因为只有 ref 型实参才能传递给 ref 型函数参数，只有 out 型实参才能传递给 out 型函数参数。但是，当调用函数的时候，你会非常容易忘记 ref 和 out 修饰符，所以最好不要重载 ref 和 out 型参数。例如：

```
sealed class Overloading
{
    public static void Example(int parameter)
    { ... }
    public static void Example(ref int parameter)
    { ... }
    static void Main()
```

```
        {
            int argument = 42;
            Example(argument);//在这儿非常容易忘记 ref 修饰符
        }
    }
```

9.访问规则

```
函数参数或返回值不能比所属函数的访问级别低
sealed class T { ... } //类的默认访问级别是 internal
public sealed class Bad
{
    public void Parameter(T t)   //错误，函数的访问级别（public）比参数高
    { ... }
    public T Return()                    //错误，函数的访问级别（public）比返回值高
    { ... }
}
public sealed class Good
{
    private void Parameter(T t)   //正确，函数的访问级别（private）比参数低
    { ... }
    private T Return()                   //正确，函数的访问级别（private）比返回值低

    { ... }
}
```

10.找错误

```
sealed class Buggy
{
    void Defaulted(double d = 0.0)                    1
    { ...
    }
    void ReadOnly(const ref Wibble w)                 2
    { ...
    }
    ref int ReturnType()                               3
    { ...
    }
    ref int fieldModifier;                           4
}
```

第 1 个函数的错误是：C#中函数不能拥有缺省参数。

第 2 个函数的错误是：ref 型参数不能用 const 修饰，因为 ref 型参数是可能变化的。

第 3,4 个函数的错误是：ref 和 out 型参数只能用于函数参数和实参。

C#中可以通过函数重载的办法实现缺省参数的功能，以下是实现的方法：

```
sealed class Overload
{
        void DefaultArgument() { DefaultArgument(0.0); }
        void DefaultArgument(double d) { ... }
}
```

一周学会C#（值的类型）3

C#才鸟（QQ:249178521）

1.整型

类型	位数	System.	与 CLS 兼容?	有无符号
sbyte	8	SByte	否	有
ushort	16	UInt16	否	无
uint	32	UInt32	否	无
ulong	64	UInt64	否	无
byte	8	Byte	是	无
short	16	Int16	是	有
int	32	Int32	是	有
long	64	Int64	是	有

有符号整形和字节型是属于“通用语言认证系统”(CLS)的。而无符号整形不属于 CLS。

你可以使用原始的类型关键字（如 int）或与之对应的别名（如 System.Int32），这两种方法都是可行的。但唯一例外的情况是：当你把类型名作为.net framework 函数实参的时候，你只能使用 System.Int32，而不能使用 int。例如，你必须这样调用：Type.GetType("System.Int32")，而 Type.GetType("int")语句是错误的。这是因为 int 只是 C#中的关键字，而.net framework 函数是设计成在所有的.net 语言中都通用的。必须注意的是 byte 在 C#中是无符号的。

注意：byte 和 sbyt 只有 8 位，因此它们不能作为数组的元素，因为数组元素的最小尺寸是 16 位（2 字节）。

2.基本操作符

括号	(x)
访问成员	x.y
函数调用	f(x)
访问数组（不是元素）	a[x]
自增	x++
自减	x--
调用构造函数	new
获得类名	typeof

获得尺寸	sizeof（不安全的）
数值检查	(un)checked

基本操作符具有最高的优先级。

new 只能在调用构造函数的时候使用，并且不能被用来重载。使用 new 来调用结构的构造函数会在栈(stack)中分配内存，而用 new 来调用类的构造函数会在堆(heap)中分配内存。在 C#中，结构是值类型的，类是引用类型的。

sizeof 返回类或一个表达式的尺寸，但它只能用在标识为 unsafe 的代码块中。

checked 和 unchecked 操作符用来控制是否检查算术运算溢出。

3. 操作符的优先级别

基本操作符	见上表
一元操作符	**+ - ! ~ ++x –-x (T)x**
乘和除	*** / %（取余数）**
加和减	**+ -**
移位	**<< >>**
关系	**< > <= >= is as**
相等	**== !=**
位操作	& ^ | (注意：左边比右边级别高)
布尔	**&& || ?:** (注意：左边比右边级别高)
赋值	**= *= /= %= += -= ...**

所有的一元操作符都可以重载。

乘和除、加和减、移位、关系（is as 例外）、相等和位操作可以重载。

&&和||只能使用 true/false 转换操作符时才可以重载。

?:和=不能重载。

复合赋值操作符（如*= /=）可以重载。

4.连接

规则 1

除了赋值操作符外的其他二元操作符都是左连接的。

x+y+z 应理解为 （x+y）+z

规则 2

赋值操作符和?: 操作符是右连接的。

x=y=z 应理解为 x=(y=z)

x+=y+=z 应理解为 x+=(y+=z)

a?b:c?d:e 应理解为 a?b:(c?d:e)

5.计算时的顺序

操作数是严格地从左到右被计算的。

```
int m = 2;
int answer = ++m * ++m + ++m * ++m;
```

计算的顺序：

```
3 * ++m + ++m * ++m
3 * 4 + ++m * ++m
12 + ++m * ++m
12 + 5 * ++m
12 + 5 * 6
12 + 30
```

6.整数溢出

溢出错误

un/checked (表达式)

un/checked｛语句 ｝

```
int m = ...
Method(checked(m * 2));
m = checked(m * 2);
checked
{
    Method(m * 2);
    m *= 2;
}
```

以上的每一句语句都进行溢出错误检查

```
Method(m * 2);
m *= 2;
```

以上的每一句语句在用 csc /checked+ *.cs 编译时，进行溢出错误检查

以上的每一句语句在用 csc /checked- *.cs 编译时，不进行溢出错误检查

```
Method(unchecked(m * 2));
m = unchecked(m * 2);
unchecked
{
    Method(m * 2);
    m *= 2;
}
```

以上的每一句语句都不进行溢出错误检查

checked(表达式)检查一个表达式的结果是否溢出。它可以用于任何一个表达式，但只只能对整数操作符起作用，因为只有这些操作符才产生溢出。这些操作符是：++, --, -（负号），+ -（减号），　* ，/ ，%以及整型之间的显式类型转换符。

checked(表达式)的结果也是一个表达式，它可以被用来作为另一个表达式的一部分：

```
int outcome = checked( ... );
```

checked{语句}检查一系列的语句结果是否溢出。它不是一个表达式，没有结果。例如，下面有语句会产生错误：

```
int noOutcome = checked { ... };
```

（注意：checked(表达式)的括号是园括号，而 checked{语句}的括号是花括号）。

unchecked 是不检查是否溢出。

7.整数转换

隐式转换，从小到大的转换

不会丢失精度，不会抛出错误

显式转换，从大到小的转换（强制转换）

可能会丢失精度，可能会抛出错误

```
int m = int.MaxValue;//整数的最大值
short s;
checked { s = (short)m; }//显式转换，会抛出溢出错误
long n;
checked { s = m; }//隐式转换，不会抛出溢出错误
```

8.类型转换表

	sbyte	short	int	long		byte	ushort	uint	ulong
sbyte		I	I	I		E	E	E	E
short	E		I	I		E	E	E	E
int	E	E		I		E	E	E	E
long	E	E	E			E	E	E	E
byte	E	I	I	I			I	I	I
ushort	E	E	I	I		E		I	I
uint	E	E	E	I		E	E		I
ulong	E	E	E	E		E	E	E	

上表中，E 表示显式转换，I 表示隐式转换 。

9.浮点类型

类型	位数	System.	与 CLS 兼容?	后缀
float	32	Single	是	F f
double	64	Double	是	D d

C#默认的浮点类型是 double，所以你要使用 float 型，就必须在数字后面加后缀 F 或 f。

123.F 是错误的，因为 C#认为这会引起歧义。F 究竟是 123.这个浮点数的后缀还是 123 这个 int 类实例的函数？C#编译器认为这是个错误，因为 int 类没有 F 这个方法！

10.浮点数的操作符

- **大部分的操作符和整数的一样**
  - **取余% 是允许的(在 C/C++是不允许的)**
  - **移位操作符是不允许的**
- **浮点数运算不会抛出错误**
  - **很小的结果会转为 0**
  - **很大的结果会转为+/- Infinity**
  - **无效的操作，结果会转为 NaN**
  - **只要有一个操作数是 NaN，结果就转为 NaN**

11. 浮点数的转换

- **float 转为 double**
  - **隐式转换**
  - **不会抛出错误**
- **double 转为 float**
  - **显式转换**
  - **不会抛出错误**
- **整数 转为 浮点数**

隐式转换
不会抛出错误
会损失精度，但不会改变大小

- 浮点数转为整数
  显式转换
  可能会抛出溢出错误

一周学会C#（语句）4

C#才鸟（QQ:249178521）

1. 语句

语句
声明语句
表达式语句
块
是语句的一个无名集合
包含在｛｝之间

```
声明语句;                    //必须要有分号
表达式语句;          //必须要有分号
{
    语句;
    语句;
    ...
}                                  //不需要有分号
```

C#和 C++、Java 一样，都可以把声明语句当作普通语句。换言之，你可以在任何地方使用声明语句，而不必在程序的开头。
一个块定义了一个范围。任何一个在块中声明的变量在块结束时，它就消失了。

2.throw 语句

throw 语句抛出错误
检查先前定义的条件时非常有用
表达式的类型必须是 System.Exception 或是它的派生类

```
string DaySuffix(int days)
{
    if (days < 0 || days > 31)
    {
        throw new
            ArgumentOutOfRangeException("days");
    }
    ...
}
```

3.return 语句

return 语句返回一个值
表达式必须匹配返回值的类型

最好一个函数只有一个 retun 语句
使用 *return*; 来结束一个 *void* 函数

```
string DaySuffix(int days)
{
    string result;
    ...
    return result;
}
```

一个函数通过 return 语句能够返回一个单值。return 语句中的表达式的类型必须和函数声明的返回值的类型相同或可以隐式转换为返回值的类型。
如果你要从一个函数中返回多个值，那你可以使用以下方法：

你可以把返回值放在一个结构中
你可以把返回值放在一个数组或集合类的对象中
你可以使用在函数中使用 out 型参数

4.bool

bool 是一个关键字
它是 System.Boolean 的别名
它的取值只能为 *true* 和 *false*

```
bool love = true;
bool teeth = false;
//正确
System.Boolean love = true;
System.Boolean teeth = false;
//正确
using System;
...
Boolean love = true;
Boolean teeth = false;
//正确
Boolean love = true;
Boolean teeth = false;
//错误，因为没有包含 System 命名空间，请注意大小写
```

5.布尔型操作符

```
1.赋值   =
2.等于   ==   !=
3.逻辑   ! && ||   ^ & |
int tens   = (9 * 6) / 13;
int units = (9 * 6) % 13;
bool isFour = tens == 4;
bool isTwo = units == 2;
bool hhg;
hhg = isFour & isTwo;
hhg = !(isFour & isTwo);
hhg = !isFour | !isTwo;
```

```
hhg = !hhg;
```

&&的结果是只有当操作符两边的操作数都是 true 时才是 true。如果&&左边的操作数是 false 的话，那么不管右边的操作数是 false 还是 true，整个&&表达式的值为 false。

||的结果是当操作符两边的操作数只要一个是 true 的时候就是 true。如果&&左边的操作数是 true 的话，那么不管右边的操作数是 false 还是 true，整个&&表达式的值为 true。

!表示取反的意思。

有一种称为“短路”的技术非常有用。例如，左边的表达式可以判断一个值是否为 0，然后右边的表达式可以把这个值作为除数。例如：

```
if ((x != 0) && (y / x > tolerance)) ...
```

6.if 语句

```
string DaySuffix(int days)
{
    string result;
    if (days / 10 == 1)
        result = "th";
    else if (days % 10 == 1)
        result = "st";
    else if (days % 10 == 2)
        result = "nd";
    else if (days % 10 == 3)
        result = "rd";
    else
        result = "th";
    return result;
}
```

if 语句的条件表达式必须是纯粹的 bool 型表达式。例如下面的诗句是错误的：

```
if (currentValue = 0) ...
```

c#要求所有的变量必须预先明确赋值后才能使用，因此，下列的程序是错误的：

```
int m;
if (inRange)
    m = 42;
int copy = m; //错误，因为 m 可能不会被赋初值。
```

在 C#中，if 语句中不能包含变量声明语句，例如：

```
if (inRange)
        int useless;//  错误
```

7.switch 语句

```
    语法
    用于整数类类型
    case 后的标志必须是编译时为常数
    没有表示范围的缩略形式
string DaySuffix(int days)
{
```

```
    string result = "th";
    if (days / 10 != 1)
        switch (days % 10)
        {
            case 1 :
                result = "st"; break;
            case 2 :
                result = "nd"; break;
            case 3 :
                result = "rd"; break;
            default: //表示不符合上面条件的情况
                result = "th"; break;
        }
    return result;
}
```

你只能对整型、字符串或可以隐式转换为整型或字符串的用户自定义类型使用 switch 语句。case 标志必须在编译时是常数。

C#中没有 Visual Basic 中的 Is 关键字在 case 中进行比较，例如：

```
        switch (expression())
        {
            case Is < 42      : //错误
            case method()     : //错误
        }
```

C#中没有范围比较符。

```
switch (expression())
{
        case 16 To 21     : //错误
        case 16..21       : //错误
}
```

注意：每个 case 段必须包括 break 语句，default 语句也不例外。

8.while/do

```
int digit = 0;
while (digit != 10)
{
    Console.Write("{0} ", digit);
    digit++;
}//没有分号
int digit = 0;
do
{
    Console.Write("{0} ", digit);
    digit++;
}
while (digit != 10);//有分号
```

9.for 语句

```
            for 语句
            for 块中声明的变量是局部的，只在 for 块中有效
            可以省略 for 语句中的任何一部分
for (int digit = 0; digit ！＝ 10; digit++)
{
    Console.Write("{0} ", digit);
}
//屏幕上显示 0 1 2 3 4 5 6 7 8 9
```

在 for 块中声明的变量只在 for 块中有效。例如：

```
    for (int digit = 0; digit != 10; digit++)
    { ... }
    Console.WriteLine(digit); //错误，digit 只在 for 块中有效
```

可以通过逗号在 for 语句中声明多个变量和多个变化语句：

```
    for (int i = 0, j = 0; i + j < 20; i++, j++)
    { ... }
```

10.foreach

```
            来源于 shell, VB, PERL
            用于任一集合，包括数组
using System;
sealed class Foreach
{
    static void Main(string[] args)
    {
        foreach (string arg in args)
        {
            Console.Write("{0} ", arg);
        }
        Console.WriteLine();
    }
}
```

把以上程序在 windows 命令行进行编译：**C:\Sharp>csc Foreach.cs**
然后键入：**C:\Sharp>Foreach 0 1 2 3 4 5 6 7 8 9**
屏幕上显示 **0 1 2 3 4 5 6 7 8 9**
foreach 用来遍历集合或数组中的元素。

11.foreach 的几点注意

```
            foreach ( 类型 标识符 in 表达式 )
            类型和标识符声明一个循环变量
            循环变量隐含为 readonly，不能有 ref 或 out 修饰
            表达式是可列举的集合
sealed class ForeachTranslation
{
    static void Example(MyWibbleCollection wibbles)
    {
```

```
            MyEnumerator pos = wibbles.GetEnumerator();
            while (pos.MoveNext())
            {
                Wibble current = pos.Current;
                Console.Write("{0} ", current);
            }
        }
}
```

集合类型的定义规则：（假设集合的名字是 C）

C 必须定义一公有函数 GetEnumerator(),该函数的返回值是一结构类型或是一类类型或是一接口类型（假设返回值的名字是 E）；

返回值 E 的定义规则：

E 包含一公有函数 MoveNext()，用来使 E 指向集合中的下一个元素，返回值的类型是 bool。

E 包含一公有属性 Current，用来读取当前值。这个属性的类型是集合元素的类型。

12.break/continue

break 用来结束一个循环，continue 用来重新启动一个循环。

13.找错

```
{
    for (int i = 0; i != 12; i++)                    1
    {
        ...
    }
    int i = 0;
}

for (int i = 0, i == 12, i++)                        2
{
    ...
}

foreach (int i in array)                                3
{
    i++;
}
```

第 1 个程序的错误：不能在一个父块和一个子块中声明两个相同名字的变量

第 2 个程序的错误：不是使用逗号分割 for 语句的三个部分，而应使用分号。for 语句的判断条件应为逻辑语句。

第 3 个程序的错误：foreach 的循环变量是只读的

C#才鸟（QQ:249178521）

1.问题

```
越大的程序包含的名字越多
越多的名字-越大的可能性导致命名冲突
你是怎样命名以反映结构
显式的前缀不是一个很好的解决方法
sealed class Book
{
    ...
}
sealed class GUIBook
{
    ...
}
//传统的命名方法
```

2.解决的方法

```
一个命名空间是一个逻辑的命名系统
命名空间表示一个范围
任何.cs 文件中都可以在任一命名空间中插入类
单独的一个.cs 文件可以访问多个命名空间
namespace GUI
{
    sealed class Book
    {
        ...
    }
}
//命名空间的解决方法
```

使用命名空间的方法可以反映程序中的逻辑关系。上面的例子说明你在 GUI 命名空间中声明了一个类叫 Book，而不是使用 GUIBook 这么长的名字。

3.嵌套的命名空间

```
一个命名空间可以包含其他的命名空间
使用嵌套来反映程序的结构
命名空间总是隐含为 public

namespace RainForest
{
  namespace GUI
  {
    sealed class Book
    {
        ...
```

```
    }
  }
}
```

```
namespace RainForest.GUI
{
    sealed class Book
    {
        ...
    }
}
```

上面两个程序是等价的。
命名空间可以包含类和其他的命名空间，但不能包含数据。
嵌套的命名空间之间的结构反映了程序组织的逻辑结构。
命名空间隐含为 public，也就是说命名空间的声明不能包含任何访问修饰符，连 public 也不能加。命名空间之所以隐含为 public 是因为这样任何一个程序的任何一部分都可以访问它。还必须注意的是，因为命名空间是隐含为 public，所以它的命名应使用 PascalCase 命名规则，即所有单词的首字母大写。
嵌套的命名空间可以非常有效地组织大型程序的逻辑结构。但是每一层都重复键入关键字 namespace 的话，那将是非常繁的。但正如上面的例子中所示的，你可以使用简写的方法。

4.全名

```
命名空间反映的是逻辑结构
带点的全名称是冗长和讨人厌的
但它比不带点的名字要好
```

```
namespace RainForest.GUI
{
    sealed class Book
    {
        ...
        private System.Collections.Hashtable pages;
    }
}
```

命名空间的使用可以避免命名冲突，但是它会导致名字的长度增加。例如，.NET framework 有一个类叫 Hashtable。这个类位于 Collections 命名空间，而 Collections 命名空间又位于 System 命名空间，这就意味着 Hashtable 类的全名是 System.Collections.Hashtable（好长啊）。

5.using 标记

```
using 标记使类在命名空间中可见
只能在一个命名空间的开头使用
namespace RainForest.GUI
{
```

```
    using System.Collections;
    ...
    sealed class Book
    {
        ...
        private Hashtable pages;
    }
}
```

using 标识只能用在命名空间的开头，放在任何类声明语句的开头。 using 标识符的作用是能够使用简写的形式来调用该命名空间中的类。例如在上面的例子中，Book 类声明了一个私有字段 pages，它是 Hashtable 类的。Book 类声明这个字段时使用的是简写形式，而不是它的全名 System.Collections.Hashtable。

```
        using 标记也可以被用在.cs 文件的开头
using System; //表示全局空间
sealed class Global
{
    static void Main(string[] args)
    {
        Console.Write(args[0]);
    }
}
```

6.引用别名

```
        using 别名 产生一个别名用于：
        类或命名空间
        只能被用在命名空间的开头
namespace RainForest.GUI
{
    using Hashtable = System.Collections.Hashtable;

    sealed class Book
    {
        ...
        Hashtable pages;
    }
}
```

一周学会C#（枚举）5

C#才鸟（QQ:249178521）

1.类型

```
        值类型
        变量直接包含它们自己的数据
```

局部变量总是放在栈(stack)中
引用类型
变量间接指向它们的数据
局部变量指向堆(heap)中的对象

枚举（enum） 值类型
结构（struct） 值类型
类（class） 引用类型
接口（interface） 引用类型
数组（[ ]array ） 引用类型
委托（delegate） 引用类型

你可能对上面的例子感到奇怪，c#中的内在类如 int,double 怎么没有。C#规定这些内在类属于结构，C#称之为简单类型。简单类型和用户自定义类型之间的最大区别是前者有字面表达式（如 42），而后者没有。
当然，还有第三种类型：指针。但指针只用在由 unsafe 关键字标识的非安全的代码中。

2.枚举类型

它是一个用户声明的值类型

```
enum Suit
{
    Clubs, Diamonds, Hearts, Spades
}
//Suit 表示一副牌，它有 4 个花色：梅花(Clubs)，方块(Diamonds)，红心(Hearts)，//黑桃(Spades)
sealed class Example
{
    static void Main()
    {
        ...
        Suit lead = Spades; //错误
        ...
        Suit trumps = Suit.Clubs; //正确
        ...
    }
}
```

枚举的声明可以出现在类声明的相同地方。
枚举的声明包括名字、访问权限、内在的类型和枚举的成员。
枚举中声明的常量的范围是定义它们的枚举，换言之，下面的例子是错误的：

```
Suit trumps = Clubs;
```

Clubs 必须被限制为 Suit 的一个成员，就如下面：

```
Suit trumps = Suit.Clubs;
```

3.枚举的注意点

枚举值缺省为 int
你可以选择任一内在的整数类型
但不能是字符型

```
enum Suit : int //内在类型是 int，可以省略
    {
        Clubs,
        Diamonds,
        Hearts = 42, //成员的取值缺省为前一个成员取值+1，但可以自己赋初值
        Spades, //最后一个分号是可选的
    };//可以有结尾分号
```

枚举类可以显式的声明它的内在类型是 sbyte, byte, short, ushort, int, uint, long, ulong。如果一个枚举类没有显式声明它的内在类型，则缺省为 int。

成员的取值必须和枚举声明的内在类型相同，并且必须在内在类型的范围之内（例如，你不能让成员的取值为负数，而枚举的内在类型是 uint）。

如果成员没有被赋值， 那么它的取值是前一个成员取值+1，第一个成员的缺省值是 1。枚举的成员的取值可以有相同的取值。

最后一个枚举成员可以使用一个结尾分号，这使得你将来可以很方便地加入更多的成员。

枚举成员的访问权限隐含为 public。

4.使用枚举

```
枚举隐式派生于 System.Enum
namespace System
{
    public abstract class Enum ...
    {
        // 静态函数
        public static string[] GetNames(Type);
        ...
        // 实例函数
        public override string ToString();
        // 构造函数
        protected Enum();
    }
}
Suit trumps = Suit.Clubs;
string[] names = System.Enum.GetNames(trumps.GetType());
...
Console.Write(names[0]);                // Clubs
Console.Write(trumps);                  // Clubs
Console.Write("{0}", trumps);           // Clubs
Console.Write(trumps.ToString()); // Clubs
Console.Write((Suit)24);                // 24
```

System.Enum 是 System 命名空间的一个抽象类。它实现了一系列的接口：

```
public abstract class Enum
        : IComparable, IFormattable, IConvertible
{
```

...
}
枚举（例如 Suit）隐式派生于 System.Enum。
System.Enum 是一个特殊的类，它只能被用作一个枚举类型的隐含基类。
你不能自己从 System.Enum 显式派生自己的类。
你不能创造一个 System.Enum 的实例。（它的构造函数是 protected，并且它是一个抽象类）。

5.枚举的操作符

枚举变量当作整型变量看待，但在大部分情况下不能使用移位操作符。

6.枚举的转换

隐式转换
0 可以转为任一 *enum* 类型
从不抛出错误
显式转换
从 *enum* 到 *enum* 通过内在类型转换
从 *enum* 转为数值类型(包括 *char*)
从数值类型(包括 *char*) 转为 *enum*
从不抛出错误

0 可以被转为任一 enum 类型，不管 enum 类型包不包括 0。
如果你使用 Console.WriteLine 显视一个枚举值，它好像被隐式转换为一个字符串。但这是一种错觉，下面的例子说明了这一点：

```
enum Suit { Clubs, Diamonds, Hearts, Spades }
Suit trumps = Suit.Clubs;
Console.WriteLine（trumps）;//显视为 Clubs
string s = trumps;//错误，trumps 不是字符串
```

Console.WriteLine 完成的从 enum 到 string 的类型转换是通过 System.Enum 的 IFormattable 接口实现的。

一周学会C#（结构）6

C#才鸟（QQ:249178521）

1．结构的声明

结构是用户自定义的值类型

```
struct Pair
{
    public int X, Y; //公有变量名单词的首字母大写（PascalCase 规则）
}
struct Pair
{
```

```
    private int x, y; //非公有变量名第一个单词的首字母小写（camelCase 规则）
}
struct Pair
{
    int x, y; //缺省的访问修饰符是 private
};//可以有结尾分号
```

结构是 C#程序员用来定义自己的值类型的最普遍的机制。结构比枚举更强大，因为它提供函数、字段、构造函数、操作符和访问控制。结构成员的缺省访问权限是 private（在 C++中是 public）。当你定义结构的成员名时，不要忘了对公有成员使用 PascalCase 规则，而对非公有成员使用 camelCase 规则。

结构类的声明中虽然可以使用结尾分号，但建议你不要使用，这只不过是为了照顾 C++程序员的习惯。

2.值的产生

一个结构类的变量存在于栈（stack）中
字段不是被预先赋值的
字段只有被赋值后才能读
使用点操作符来访问成员

下面的例子假设 Pair 是一结构，它有两公有整数类成员 X,Y

```
static void Main()
{
    Pair p;
    Console.Write(p.X); //错误
    ...
}
```

```
static void Main()
{
    Pair p;
    p.X = 0;
    Console.Write(p.X); //正确
    ...
}
```

结构类的变量存在于栈中。在上面的例子中，虽然声明了一个叫 p 的 Pair 类结构变量，但实际上只是声明两个局部变量 p.X 和 p.Y 的一种简写形式。

上面例子中的第一段程序的 Console.Write 试图使用 p.X 的值，但它是错误的，因为 p.X 没有被赋初值。

3.值的初始化

一个结构变量：
总是能使用缺省构造函数来进行初始化
缺省构造函数把字段初始化为 0/*false/null*

```
static void Main()
{
    Pair p;
```

```
        Console.Write(p.X); //错误，p.X 没有初始化
        ...
}
static void Main()
{
        Pair p = new Pair();
        Console.Write(p.X); //正确,p.X=0
        ...
}
```

除了上面介绍的初始化方法外，还可以使用缺省构造函数来初始化一个结构变量。调用构造函数总是使用 new 关键字。一个结构变量是值类型的，它直接存在于栈中，new 关键字的使用不会在堆中开辟内存。结构的缺省构造函数总是把结构变量中的所有字段初始化（你不能改变这一行为，在下面一节会讲到）。

如果你有 C++或 Java 背景，你可能会很难相信使用 new 关键字来调用构造函数不会在堆中分配内存，但在 C#中就是这样。结构变量存在于栈中，调用构造函数初始化它的字段，没有发生堆的内存分配。

C++程序员注意：在 C#中调用缺省构造函数必须使用括号。

Pair p = new Pair; //错误

Pair p = new Pair();//正确

4.值的构造函数

一般规则

编译器声明缺省构造函数

你不能声明缺省构造函数

缺省构造函数把所有的实例字段初始化为 0/*false*/*null*

```
struct Pair
{
}
//编译器声明一缺省构造函数
struct Pair
{
        public Pair()
        { ... }
}
//错误，不能自己声明缺省构造函数
struct Pair
{
        public Pair(int x, int y)
        { ... }
}
//正确，但编译器声明的缺省构造函数仍存在
```

结构类总有一编译器声明的公有的缺省构造函数。不管你有没有声明构造函数，编译器声明的公有的缺省构造函数总是存在的。所以你不能定义缺省构造函数，这样会出现两个缺省构造函数，这是不允许的。但要注意的是，这只适合于结构，对于类是不适用的。编译器产生的缺省构造函数把所有的实例字段归零化：

bool 型化为 false
整型（包括字符型）化为 0
实型化为 0.0
枚举型化为 0
引用型（包括字符串）化为 null

用户自定义的结构类的构造函数的默认访问权限是 private，和结构类的字段一样。

C#不允许你声明一个和构造函数名字一样的函数。

5.:this(…)

一个构造函数可以调用另一构造函数

```
struct ColouredPoint
{
    public ColouredPoint(int x, int y)
        : this(x, y, Colour.Red)
    {
    }

    public ColouredPoint(int x, int y, Colour c)
    {
        ...
    }
    ...
    private int x, y;
    private Colour c;
}
```

6.实例字段

实例字段...
缺省初始化是调用编译器声明的缺省构造函数
在用户自定义的构造函数中必须显式初始化
不能在它们声明时初始化

```
struct Pair
{
    public Pair(int x, int y)
    {
        this.x = x;
        this.y = y;     //正确，所有的实例字段都被显式初始化
    }
    ...
    private int x, y; //声明时没有被初始化
}
```

结构的自定义构造函数必须显式初始化结构中的所有实例字段。（在类的构造函数中不是这样的）

```
public BadPair(int x, int y)
                {
```

```
            this.x = x;//没有定义 this.y
        }
        private int x, y;
```

结构的实例字段只能在构造函数中进行初始化，而不能使用赋值的方法。（在类的构造函数中不是这样的）

```
private int x;
        private int y = 0;//在结构中是非法的
```

7.静态字段

静态字段...
被缺省初始化为 0/*false*/*null*
可以在声明时初始化
只能通过类名访问

```
struct Pair
{
    public Pair(int x, int y)
    {
        ...
    }
    private static Pair origin = new Pair(0,0);
    ...
private int x, y;
}
Pair p = new Pair();
...
Method(p.origin);      //错误，只能通过类名访问
Method(Pair.origin); //正确
```

由 static 修饰符声明的字段称为静态变量。当类的声明装载时，静态变量就开始存在，直到程序结束时才消失。

静态变量的初值：

整型变量为 0（包括枚举）
实型变量为 0.0
bool 型变量为 false
引用型变量为 null

8.只读字段

只读字段...
不能被赋值
不能被用作 *ref/out* 型参数

```
struct Pair
{
    public static readonly Pair Origin = new Pair(0,0);
    public Pair(int x, int y)
    {
        this.x = x;
        this.y = y;
```

```
    }
    public void Reset()
    {
        x = 0; //错误
        Origin.x = 0; //错误
    }
    private readonly int x, y;
}
```

9.术语

两种值类型
枚举类型
结构类型
两种结构类型
简单结构类型
有一个关键字别名(例如 System.Int32 == *int*)
有数值表达式 (例如, 42)
用户自定义结构类型
没有一个关键字别名
没有数值表达式

10.常量字段

常量字段...
隐含为 ***static***
必须在声明时初始化
必须被初始化为编译时常量值
只有简单类型，枚举，字符串才可以是常量

```
struct Pair
{
    public Pair(int x, int y)
    {
        // ???
    }
    ...
    private const int x = 0, y = 0;
}
```

在 C#中，常量字段隐含为 static，但你不能显式声明一个常量字段是 static：

```
static const int x = 0;//错误
```

常量必须被初始化，并且只能在声明时初始化：

```
const int x;//错误
```

常量必须被初始化为编译时常量值：

```
const int x = Method();//错误
```

只有简单类型，枚举，字符串才能被声明为常量：

```
const Pair p = new Pair();//错误
```

11.找错

```
struct HHG
```

```
{
    HHG(int toTheGalaxy)
    {
        question = 9 * 6;                                    1
        ...
    }
    string cover = "Don't Panic";                            2
    static const int answer = 42;                            3
    const Hiker arthur = new Hiker();                        4
    const int question;                                          5
    ...
}
```

第 1 个语句的错误：不能在构造函数中初始化常量，常量只能在声明时初始化。
第 2 个语句的错误：结构中不能在声明实例字段时进行初始化，但在类中是可以的。
第 3 个语句的错误：不能显式声明常量字段为 static，常量字段只能隐式为 static。
第 4 个语句的错误：常量字段只能用于简单类型，枚举，字符串。
第 5 个语句的错误：常量字段在声明时没有被初始化。

12. 局部变量

```
        局部变量...
        可以被声明为 const (规则同字段)
        不能被声明为 static  或 or
struct Pair
{
    void okay()
    {
        const int answer = 42;
        ...
    }
    void compileTimeErrors()
    {
        const int local = call(); //错误，必须被初始化为编译时常量值
        const Pair origin = ...; //错误，常量只能为简单类型，枚举，字符串
        readonly Pair p = ...; //错误，只有字段才能声明为 readonly
        ...
    }
    ...
}
```

13.静态构造函数

```
        静态构造函数初始化类
        可以初始化 static  字段而不是 const  字段
        当类被装载时由.net 调用
        不能被调用：没有参数，没有访问修饰符
struct Pair
```

```
{
    public static readonly Pair Origin;
    public Pair(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    static Pair()
    {
        Origin = new Pair(0, 0);
    }
    private int x, y;
}
```

静态构造函数只能由.net 调用，而不能由程序员调用。这保证它会被调用，只被调用一次，并且在恰当的时候被调用（在任何类或结构被使用前）。因为程序员不能调用静态构造函数，所以静态构造函数没有参数。出于同样的原因，静态构造函数不能有访问修饰符。

静态构造函数不能被用来初始化常量字段，即使常量字段隐式为静态的。因为前面说过，常量字段必须被初始化，而且只能在它声明的时候初始化。

一周学会C#（属性）7

C#才鸟（QQ:249178521）

1. 问题

- 原始的封装是麻烦的

```
struct Time
{
    ...
    public int GetHour()
    {
        return hour;
    }
    public void SetHour(int value)
    {
        hour = value;
    }
    ...
    private int hour, minute, second;
}
static void Main()
{
    Time lunch = new Time();
```

```
        lunch.SetHour(12);
        Console.WriteLine(lunch.GetHour());
}
```

封装把一些不重要的细节隐藏起来，这样你可以集中精力处理那些重要的内容。但封装很难被掌握，一个典型的封装误用是盲目地把公有字段转为私有字段。例如在上面的例子中，程序定义了一个私有字段 hour 和 SetHour 函数和 GetHour 函数，而不是定义一个公有的 hour 字段。如果 GetHour 函数只是返回私有字段的值而 SetHour 函数只是设置私有字段的值的话，那么你除了使 Time 类更难使用外，你不会得到任何好处。

2. 不是解决的办法

```
            如果字段是公有的，那使用起来是简单的
            但如果你使用公有字段的话，你会失去控制权
            要简化而不是简单
struct Time
{
        ...
        public int Hour;
        public int Minute;
        public int Second;
}
static void Main()
{
        Time lunch = new Time();
        lunch.Hour = 30;
        lunch.Minute = 12;
        lunch.Second = 0;
        ...
}
```

上面的例子使用公有字段来使字段的使用比较简单。例如，你不用写：

```
        lunch.SetHour(lunch.GetHour() + 1);
```

而只要写：

```
        ++lunch.Hour;
```

但是，这种简单的表达式是有代价的。考虑上面的例子，程序给 Hour 和 Minute 字段分别赋值为 30 和 12。问题是 30 不在 Hour 的范围（0-23）内。但如果字段是公有的话，你就没有办法捕获这个错误。

所以虽然 get 和 set 函数比较麻烦，但它们在这方面比公有字段具有优势是很明显的。get 和 set 函数允许程序员控制类的内在字段的读和写。这是非常有用的，例如你可以检查 set 函数的参数范围。

当然最理想的方法是保留公有字段提供的简单而直接的表达式和 get 和 set 函数提供的控制权。（呵呵，人总是既想偷懒又想得到很多）

3. 解决的办法

```
            属性
            自动使用 get  标识符进行读
            自动使用 set  标识符进行写
```

```
struct Time
{
    ...
    public int Hour //没有（），是 H 而不是 h
    {
        get { ... }
        set { ... }
    }
    private int hour, minute, second;
}
Time lunch = new Time();
...
lunch.Hour = 12;
...
Console.WriteLine(lunch.Hour);
```

C#提供了一个解决上述问题的好办法。你可以把 get 和 set 函数组合成一个简单的属性。属性的声明包括一个可选的访问修饰符(在例子中是 public)、返回值(int)、属性的名字(Hour)和一个包含 get 和 set 语句的属性体。特别要注意的是属性没有括号，因为属性不是函数。属性的命名规则应符合一般的命名规则，即公有的使用 PascalCase 规则，而非公有的使用 camelCase 规则。在上面的例子中，Hour 属性是公有的，所以命名为 Hour 而不是 hour。

例子中演示了属性的用法。属性使用的语法和字段的一样，没有括号。如果你要写一个属性，那你可以这样写：

```
lunch.Hour = 12;
```

属性的 set 语句自动被执行。

如果你要读一个属性，你可以这样写：

```
int hour = lunch.Hour;
```

属性的 get 语句自动被执行。

4. get 语句

get 语句

必须返回一个有确定类型的值

功能上就像一个 “get 函数”

```
struct Time
{
    ...
    public int Hour
    {
        get
        {
            return hour;
        }
        ...
    }
    private int hour, minute, second;
```

```
}
Time lunch = new Time();
... Console.WriteLine(lunch.Hour);
//请注意，get 和 set 不是关键字
```

当读一个属性的时候，属性的 get 语句自动运行。

get 语句必须返回一个有确定类型的值。在上面的例子中，Time 结构类有一个整型属性 Hour，所以它的 get 语句必须返回一个整型值。

属性的返回值不能是 void（从这里可以推断出字段的类型也不能是 void）。这就意味着 get 语句必须包含一个完整的 return 语句（retun;这种形式是错误的）。get 语句可以在 retun 语句前包含任何其他的语句（比如，可以检查变量的类型），但 return 语句不能省略。

注意，get 和 set 不是关键字，所以你可以在任何地方包括 get/set 语句中声明一个局部变量、常量的名字是 get 或 set，但最好不要这样做。

5. set 语句

```
        set 语句
        是通过 value 标识符来进行赋值的
        可以包含任何语句（甚至没有语句）
struct Time
{
    ...
    public int Hour
    {
        ...
        set {
            if (value < 0 || value > 24)
                throw new ArgumentException("value");
            hour = value;
        }
    }
    private int hour, minute, second;
}
Time lunch = new Time();
...
lunch.Hour = 12;
```

当写一个属性的时候，属性的 set 语句自动运行。

在上面的例子中，Time 结构类有一个整型属性 Hour，所以赋给这个属性的值必须是一个整型值。例如：

```
lunch.Hour = 12;
```

把一个整型值 12 赋给了 lunch 的 Hour 属性，这个语句会自动调用属性的 set 语句。set 语句是通过 value 标识符来获得属性的赋值的。例如，如果 12 被赋给了 Hour 属性，那么 vaue 的值就是 12。注意的是 value 不是一个关键字。value 只是在 set 语句中才是一个标识符。你可以在 set 语句外的任何语句声明 value 为一变量的名字。例如：

```
public int Hour
```

```
{
    get { int value; ... }//正确
    set { int value; ... }//错误
}
```

6. 只读属性

只读属性只有 get 语句
任何写操作都会导致错误
就像一个只读字段

```
struct Time
{
    ...
    public int Hour
    {
        get
        {
            return hour;
        }
    }
    private int hour, minute, second;
}
Time lunch = new Time();
...
lunch.Hour = 12; //错误
...
lunch.Hour += 2;//错误
```

一个属性可以不必同时声明 get 语句和 set 语句。你可以只声明一个 get 语句。在这种情况下，属性是只读的，任何写的操作都会导致错误。例如，下面的语句就会导致一个错误：

```
lunch.Hour = 12;
```

因为 Hour 是只读属性。

但要注意的是，属性必须至少包括一个 get 或 set 语句，一个属性不能是空的：

```
public int Hour { }//错误
```

7. 只写属性

只写属性只能有 *set* 语句
任何读操作都是错误的

```
struct Time
{
    ...
    public int Hour
    {
        set {
            if (value < 0 || value > 24)
                throw new OutOfRangeException("Hour");
            hour = value;
```

```
            }
        }
        private int hour, minute, second;
}
Time lunch = new Time();
...
Console.WriteLine(lunch.Hour); //错误
...
lunch.Hour += 12;//错误
```

一个属性可以不必同时声明 get 语句和 set 语句。你可以只声明一个 set 语句。在这种情况下，属性是只写的，任何读的操作都会导致错误。例如，下面的语句就会导致一个错误：

```
Console.WriteLine(lunch.Hour);
```

因为 Hour 是只写属性。

而下面的例子则看上去好像是对的：

```
lunch.Hour += 2;
```

这句语句的实际运作是这样的：

```
lunch.Hour = lunch.Hour + 2;
```

它执行了读的操作，因此是错误的。因此，像+=这种复合型的赋值操作符既不能用于只读属性，也不能用于只写属性。

8. 静态属性

```
        静态属性是和类联系在一起的
          只能通过类名使用
sealed class Error
{
        ...
        public static TextWriter Log
        {
                get { return log; }
        }
        ...
        private static Stream sink
                = new FileStream("error.log", FileMode.Append);
        private static TextWriter log
                = new StreamWriter(sink);
}
Error.Log.WriteLine("time out");
```

字段可以是静态的，所以属性也可以是静态的。声明静态属性的语法很简单，只要在属性名前加入 static 关键字。静态函数中的机制和限制同样适用于静态属性。静态属性可以同一般的属性一样声明为只读或只写。

静态属性没有隐含的 this 参数。例如，上面的例子中，Log 这个静态属性之所以能访问 log 这个字段，是因为 log 是一个静态字段。如果 log 是一个实例字段，那么 Log 这个静态属性就不能访问它。例如：

```
public sealed class Error
```

```
    {
        public static TextWriter Log
        {
            get { return log; }
        }
        private Stream sink = ...;
        private TextWriter log = ...;
    }
```

9. 属性 vs.字段

属性和字段的比较：
属性不能使用 *ref/out* 型参数
属性使用前必须赋值

```
//属性
struct Time
{
    ...
    public int Hour
    {
        set { ... }
    }
    private int hour;
}
Time lunch;
Method(out lunch.Hour); //错误
lunch.Hour = 12;//错误

//字段
struct Time
{
    ...
    public int Hour;
    ...
}
Time lunch;
Method(out lunch.Hour); //正确
lunch.Hour = 12;
```

属性使用前必须赋值，例如：

```
Time lunch;
lunch.Hour = 12;//错误，lunch 没有初始化
```

10. 属性 vs.函数

相似点
都包含执行代码
都可以有访问修饰符
都可以有 *virtual, abstract, override* 修饰符

都可以用在接口中
不同点
属性只能拥有 get/set 语句
属性不可以是 *void* 型
属性不能使用参数
属性不能使用[ ] 参数
属性不能使用括号

一周学会C#（索引）8

C#才鸟（QQ:249178521）

1.[]

索引提供[ ] 类的语法
总是一个实例成员，可以是虚拟的
没有 *ref/out* 参数

```
struct StringSection
{
    ...
    public char this [int at]
    {
        ...
    }
    ...
}
StringSection cord("csharp", 2, 6);
...
char first = cord[0];       // 'h'
...
Console.WriteLine(cord); // harp
```

索引的声明包括：可选的访问修饰符（例子中是 public），返回值的类型（char），关键字 this（不能省略），和函数参数类似的参数（不过是方括号，而不是函数的圆括号），然后是索引体。你不能使用静态索引，所以你不能在索引的声明中使用 static 关键字。

索引可以被声明为虚拟的，因此它可以在它的派生类中被重载。

索引的参数不能使用 ref/out 型参数。例如：

```
struct StringSection
{
    public char this [ref int at]//错误
    ...
}
```

2.读和写

索引只能包含下列语句
get { } 用来读

```
            set { } 用来写
struct StringSection
{
    ...
    public char this [int at]
    {
        get { return ...; }
        set { ... = value; }
    }
    ...
}
cord[3] = 'e';
if (cord[3] == 'e') ...
```

索引的声明和属性一样：只能含有 set/get 语句。

当使用一个索引表达式进行读操作时，索引的 get 语句自动运行，例如在上面的例子中，表达式：

cord[3] == 'e'

系统会自动调用索引的 get 语句（3 传递给索引的整型参数），返回一个 char 型值，这个 char 型值然后和'e'进行比较。

当使用一个索引表达式进行写操作时，索引的 set 语句自动运行，例如在上面的例子中，表达式：

cord[3] = 'e'

系统会自动调用索引的 set 语句（3 传递给索引的整型参数），set 语句可以取得表达式右边的值，它是通过 value 这个“关键字”来获得的。

3.只读或只写

```
            只有 get 语句的索引是
            只读索引
            只有 set 语句的索引是
            只写索引
struct StringSection
{
    ...
    public char this [int at]
    {
        get { return ... }
    }
    ...
}
StringSection cord("csharp", 1, 6);
...
if (cord[4] == 'k') {
    cord[4] = 'k'; //错误
}
```

4.索引 vs.数组

索引和数组的比较
索引可以使用非整型参数
索引可以被重载
索引可以是私有的
索引不能有 ***ref/out*** 型参数

```
struct Matrix
{
    ...
    public double this [int row, int col]
    {
        get { ... }
        set { ... }
    }
    public Row this [int row]
    {
        get { ... }
        set { ... }
    }
    ...
}
```

5.索引 vs.属性

索引和属性的比较
都只有 **get** 和 **set** 语句
没有静态索引
可以有静态属性
索引是在类的层次上声明的
属性是在字段的层次上声明的

```
struct Row
{
    ...
    public double this [int col]
    {
        get { ... }
        set { ... }
    }
    public int Length
    {
        get { ... }
    }
    ...
}
```

属性和索引都能在接口中声明。

interface IRow

```
{
    double this [int index] { get; set; }
    int Length { get; }
}
```

6.索引 vs.函数

索引和函数的比较
函数可以没有参数
函数可以有 void 型返回值
索引只能包含 get/set 语句

一周学会C#（类）9

C#才鸟（QQ:249178521）

1．类的声明

类是用户自定义的引用类型

```
class Pair
{
    public int X, Y; //公有变量名单词的首字母大写（PascalCase 规则）
}
class Pair
{
    private int x, y; //非公有变量名第一个单词的首字母小写（camelCase 规则）
}
class Pair
{
    int x, y; //缺省的访问修饰符是 private
};//可以有结尾分号
```

你可以使用关键字 class 来声明类这一用户自定义的引用类型。类的语法和结构的非常类似。它提供函数、字段、构造函数、操作符和访问控制。类成员的缺省访问权限是 private。当你定义类的成员名时，不要忘了对公有成员使用 PascalCase 规则，而对非公有成员使用 camelCase 规则。

类的声明中虽然可以使用结尾分号，但建议你不要使用，这只不过是为了照顾 C++程序员的习惯。

2.对象的产生

一个类的局部变量存在于栈（stack）中
不是被预先赋值的
可以被初始化为 null 或调用构造函数进行初始化

(下面的例子中 Pair 类的声明请看前面，右边显示的是内存中的情况，@表示指向)

```
  栈              堆
static void Main()
{
    Pair p;                                  p ?
```

```
}
static void Main()
{
    Pair p = null;                    p null
}
static void Main()
{
    Pair p = new Pair();          p @              0   .X
}                                                    0   .Y
```

虽然类的声明和结构的声明非常类似，但类与结构是两个不同的类型。结构是值类型，而类是引用类型。无论类的实例多大，类的局部变量只是这个类实例的一个引用。

上面例子中的最上面的那段程序定义了一个 Pair 类的局部变量 p。不管 Pair 类包含什么成员，p 只是存在存在于栈中的一个引用。因为 p 没有被初始化，所以这个引用没有被赋值，p 也就不能被使用。

上面例子中的中间的那段程序定义了一个 Pair 类的局部变量 p。p 被初始化为 null，所以 p 没有指向任何对象。p 已经被赋值，所以 p 也就能被使用。

上面例子中的最下面的那段程序定义了一个 Pair 类的局部变量 p。由 new 新生成一 Pair 类对象，这个新产生的对象产生在堆(heap)中，然后 p 通过赋值指向这一堆中的对象，而 p 是被定义为存在于栈中。p 已经被赋值，所以 p 也就能被使用。

new 对于类来说是产生一堆中的对象，而对于结构来说是在栈中产生一个值，这可能需要一段时间的适应。

3.对象的构造函数

```
        类的构造函数!＝结构的构造函数
        编译器声明缺省构造函数
        你可以声明缺省构造函数
        如果你声明构造函数，那么编译器不会声明构造函数
class Pair
{
}
//编译器声明一缺省构造函数
class Pair
{
    public Pair()
    { ... }
}
//正确，可以自己声明缺省构造函数
class Pair
{
    public Pair(int x, int y)
    { ... }
}
```

//正确，但编译器声明的缺省构造函数不存在，不存在缺省构造函数

类的缺省构造函数的规则与结构的缺省构造函数的规则是不同的。你可以回忆一下，结构总有一编译器声明的公有的缺省构造函数。不管你有没有声明构造函数，编译器声明的公有的缺省构造函数总是存在的。所以你不能定义缺省构造函数，这样会出现两个缺省构造函数，这是不允许的。

但这只适合于结构，对于类是不适用的。如果你没有声明任何构造函数，那编译器会产生缺省构造函数。但如果你定义了一个构造函数，那么编译器就不会产生缺省构造函数。这也意味着如果你声明了一个或多个构造函数，那么你能拥有缺省构造函数的唯一途径是你声明的构造函数中必须有一个是缺省构造函数。这样的结果是如果你只定义了一非缺省构造函数，而你又要使用缺省构造函数的话，那你只有重载这个构造函数。如果你必须手动初始化类中的每一个字段（就像在结构中一样），那将是非常麻烦的。但幸运的是你可以不必这么做，你将在下面看到这一点。

4.:this(…)

一个构造函数可以调用另一构造函数

```
sealed class Pair
{
    public Pair(int x, int y)
        : this(x, y, Colour.Red)
    {
    }

    public Pair(int x, int y, Colour c)
    {
        ...
    }
    ...
    private int x, y;
    private Colour c;
}
```

5.实例字段

实例字段...
在所有的构造函数中被初始化为缺省值（0/false/null）
可以在一构造函数中显式初始化
可以在它们声明时初始化

```
sealed class Pair
{
    public Pair(int x, int y)
    {
        this.x = x;
        y = y;
    }
    ...
    private int x;
```

```
    private int y = 42;
}
```

回忆一下：在结构的自定义构造函数必须显式初始化类中的所有实例字段。结构的实例字段只能在构造函数中进行初始化，而不能使用赋值的方法。
而类则比结构方便的多。
类中的所有字段都缺省初始化为缺省值。
在类的构造函数中，你可以在字段声明时进行初始化。
在上面的例子中，构造函数的参数 y 用字段 y 来赋值。编译器通过的原因是字段 y 已经被赋初值。其实，如果没有对字段 y 进行初始化，编译器仍然会通过，因为字段 y 已有缺省值 0，所以在构造函数中没有显式初始化字段 y 是不会发生错误的。

6.静态字段

```
        静态字段...
        被缺省初始化为 0/false/null
        可以在声明时初始化
        只能通过类名访问
class Pair
{
    public Pair(int x, int y)
    {
        ...
    }
    private static Pair origin = new Pair(0,0);
    ...
private int x, y;
}
Pair p = new Pair();
...
Method(p.origin);      //错误，只能通过类名访问
Method(Pair.origin); //正确
```

由 static 修饰符声明的字段称为静态变量。当类的声明装载时，静态变量就开始存在，直到程序结束时才消失。
静态变量的初值：

整型变量为 0（包括枚举）
实型变量为 0.0
bool 型变量为 false
引用型变量为 null

7.只读字段

```
        只读字段...
        不能被赋值
         不能被用作 ref/out  型参数
class Pair
{
    public static readonly Pair Origin = new Pair(0,0);
```

```
    public Pair(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public void Reset()
    {
        x = 0; //错误
        Origin.x = 0; //错误
    }
    private readonly int x, y;
}
```

8.常量字段

常量字段...
隐含为 ***static***
必须在声明时初始化
必须被初始化为编译时常量值
只有简单类型，枚举，字符串才可以是常量

```
class Pair
{
    public Pair(int x, int y)
    {
        // ???
    }
    ...
    private const int x = 0, y = 0;
}
```

在 C#中，常量字段隐含为 static，但你不能显式声明一个常量字段是 static：

```
static const int x = 0;//错误
```

常量必须被初始化，并且只能在声明时初始化：

```
const int x;//错误
```

常量必须被初始化为编译时常量值：

```
const int x = Method();//错误
```

只有简单类型，枚举，字符串才能被声明为常量：

```
const Pair p = new Pair();//错误
```

9.静态构造函数

静态构造函数初始化类
可以初始化 ***static*** 字段而不是 ***const*** 字段
当类被装载时由.net 调用
不能被调用：没有参数，没有访问修饰符

```
class Pair
{
    public static readonly Pair Origin;
    public Pair(int x, int y)
```

```
    {
        this.x = x;
        this.y = y;
    }
    static Pair()
    {
        Origin = new Pair(0, 0);
    }
    private int x, y;
}
```

静态构造函数只能由.net 调用，而不能由程序员调用。这保证它会被调用，只被调用一次，并且在恰当的时候被调用（在任何类或类被使用前）。因为程序员不能调用静态构造函数，所以静态构造函数没有参数。出于同样的原因，静态构造函数不能有访问修饰符。

静态构造函数不能被用来初始化常量字段，即使常量字段隐式为静态的。因为前面说过，常量字段必须被初始化，而且只能在它声明的时候初始化。