

C# - P1

2023-09-01

C# 专为公共语言基础结构 (common language infrastructure, CLI) 设计
CLI 由可执行代码和运行时环境组成

允许在不同计算机平台和体系结构上使用多种高级语言

using 在 C# 中, using 关键字用于在程序中包含命名空间 (namespace)
类似于 Java 中的 import 关键字
一个程序中通常有多个 using 语句

命名空间 (namespace), 在 C# 中, 命名空间包含一系列类 (class)

基本语法 为 namespace <namespace identifier> {
 [class <class identifier> { ... }]
}

与 Java 中的包 (package) 相似

注释 (comment), 在 C# 中, 注释的语法与 Java 类似

单行的注释用 /* 表示, 编译器忽略双斜杠后面的内容

多行的注释用 /* sd */ 表示, 编译器忽略符号内部的所有内容

Main 方法, 与 Java 类似, Main 方法是所有 C# 程序的入口点 (entry point)

即程序的执行从 Main 方法开始

在 C# 中, 语句 (statement) 和表达式 (expression) 必须以分号 (;) 结尾

标识符 (identifier), 在 C# 中用于识别类, 变量, 函数或其他用户定义项目

必须以字母 (a-z, A-Z), 数字 (0-9), 下划线 (_), @ 组成

只能以字母, 下划线, @ 开头, 不能以数字开头

不能包含嵌入的空格 (embedded space) 或其他特殊字符

不能是 C# 的关键字 (key word)

但是允许加一个 @ 前缀, 如对于关键字 int, @int 是合法标识符

C# 是大小写敏感, 标识符区分大小写字母

不能与 C# 类库名冲突

与 Java 相比, C# 允许文件名与其包含的类名不同

但是保持一致有利于编程的过程

C# - P2

part - second

关键字 (keyword) 在 C# 中编译器预定义的保留字
 关键字不可用作标识符，但是可以通过加 @ 前缀用作标识符

保留关键字 (reserved keyword)	
访问控制	private：私有 protected：受保护 public：公共
类, 方法, 变量	abstract：声明抽象 class：类 interface：接口 new：新实例 override
	static：静态 struct：结构 enum：枚举 volatile：易失 virtual
	object： object sizeof：尺寸 typeof： typeof is
程序控制语句	break：退出循环 continue：继续下一迭代 switch... case... default：switch 结构 return：返回
	if... else：条件结构 do...while：do while 循环 for / foreach：for 循环
错误处理	catch：捕获异常 finally：无论有无异常 throw：抛出异常 try：尝试执行
命名空间	using：包含命名空间 namespace：声明命名空间
基本类型	bool：布尔值 byte：字节型 sbyte：有符号 8-bit 整数 char：字符型 decimal：十进制型
	int：整型 uint：无符号整型 short：短整型 ushort：无符号短整型 string：字符串
	long：长整型 ulong：无符号长整型 float：单精度浮点数 double：双精度浮点数
变量引用	this：本类 this void：无返回值
保留	const：常量 goto：跳转 null：空 true：真 false：假
	as： base checked delegate event
	explicit：显式 implicit：隐式 internal：内部 extern fixed
	in：输入参数 in (generic modifier) out：输出参数 out (generic modifier) lock
	operator：操作符 params：参数 readonly：只读 ref sealed
	stackalloc：堆分配 unchecked：未检查 unsafe

上下文关键字 (contextual keyword) 在 C# 的代码的上下文中具有特殊意义

add	alias	ascending	descending	dynamic
from	get	global	group	into
join	let	orderby	partial (types)	partial (method)
remove	select	set		

/// 注释，在 C# 中引入 XML 注释，注意 /// 注释会被编译，但不会影响执行速度

在类、属性、方法前使用 ///，VS.NET 自动增加 XML 格式的注释

可以生成一个 XML 文件，程序内容更清晰

在其他位置调用代码时提供智能感知

在其他地方使用注释的类、属性、方法时，黄色的 框显示注释

C# - P3

值类型 - Value Type

值类型

(value type), 在C#中值类型直接包含数据, 如 int / char / float

当声明一个值类型变量时, 系统根据声明类型来分配内存

值类型都是从类 System.ValueType 中派生的

类型	描述	值范围	默认值
bool	布尔值	True / False	False
byte	8位无符号整型	0 ~ 255	0
sbyte	8位有符号整型	-128 ~ 127	0
int	32位有符号整型	- 2^{31} ~ $2^{31}-1$	0
uint	32位无符号整型	0 ~ $2^{32}-1$	0
short	16位有符号整型	-32768 ~ 32767	0
ushort	16位无符号整型	0 ~ 65535	0
long	64位有符号整型	- 2^{63} ~ $2^{63}-1$	0L
ulong	64位无符号整型	0 ~ $2^{64}-1$	0
float	32位单精度浮点数	-3.4×10^{38} ~ $+3.4 \times 10^{38}$	0.0F
double	64位双精度浮点数	$\pm 5.0 \times 10^{-324}$ ~ $\pm 1.7 \times 10^{308}$	0.0D
decimal	128位精确十进制数	$(-7.9 \times 10^{28} \sim +7.9 \times 10^{28}) / 10^{0 \sim 28}$	0.0M
char	16位 Unicode 字符	U+0000 ~ U+FFFF	'\0'

sizeof

在C#中, 不同平台上对于同一类型 / 变量分配的内存是不同的

可以使用 sizeof 方法得到类型 / 变量在特定平台上的准确尺寸

基本语法为 sizeof(type), sizeof object

返回以字节为单位存储对象 / 类型的存储尺寸

引用类型

(reference type), 在C#中引用类型不包含存储在变量中的实际数据

引用类型包含对变量的引用, 即一个内存位置

多个引用类型变量可以指向同一个内存位置

如果内存位置的数据由一个变量修改, 其他变量会自动反映这种变化

内置引用类型有 object, dynamic, string

字符串

对象 (Object) 类型, 动态 (dynamic) 类型, string (String) 类型

用户自定义的引用类型有

类 (class), 接口 (interface), 委托 (delegate)

C# - P4

对象类型 (Object)，在C#中，是通用类型系统 (common type system, CTS) 中所有数据类型的终极基类，也是 System.Object 类的别名。可以被分配给任何其他类型的值。

值类型 / 引用类型 / 预定义类型 / 用户自定义类型

在分配给值前需要先进行类型转换

装箱：将一个值类型转换为对象类型

拆箱：将一个对象类型转换为值类型

动态类型 (Dynamic)，可以在动态数据类型变量中存储任何类型的值。

变量的类型检查在运行时发生。

声明动态类型的基本语法为

dynamic <variable_name> = value;

与对象类型相比，对象类型变量的类型检查在编译时发生。

字符串类型 (String)，在C#中允许给字符串类型变量分配任何字符串值。

从对象 (Object) 类型派生，是 System.String 类的别名。

可以通过两种形式进行分配：

双引号 ("")：与 C++ 和 Java 中的字符串常量类似。

表示字符串的字面值。

@引号 (@ "")：在 C# 中，可以在字符串之前添加 @，称为逐字字符串。

将转义字符 (\) 当作普通字符处理。

例如，通过单引号或双引号如 string str = @"C:\Windows";

等价于 string str = "C:\\Windows";

@字符串中可以任意换行，也可以插入其他双引号。

换行符及缩进空格均计入字符串。

特别注意 String 与 String 的区别：

string 是 C# 中的类，String 是 .NET Framework 的类，C# string 映射为 String。

使用 C# 时，使用 String 比较符合规范，始终代表 System.String (1.x) /:: System.String (2.0)

String 仅在有 using System; 且命名空间中没有名为 String 的类型时代表 System.String。

string 是 C# 的关键字，但 String 不是，String 是 CLR (Common Language Runtime) 的类型。

String 在编译时由 C# 编译器默默地转换为 String，而增加几行转换代码。

直接使用 String 可以减少 C# 编译器的工作。

所以也有建议使用 CLR 类型而不使用 C# 类型 (习惯 / 规范问题)

指针类型 (pointer type), 在C#中指针类型变量存储另一种类型变量的内存地址
 C#中的指针与C/C++中的指针有相同的功能

声明指针类型的基本语法为

`<type>* <identifier>;`

类型转换 在C#中根本上是类型铸造 (type casting), 即把数据从一种类型转换为另一种类型

隐式类型转换：是C#默认的以安全方式进行的转换

不会导致数据丢失，也不会损失精度

如小整数类型到大整数类型，派生类转换为其基类

显式类型转换：即强制类型转换

需要强制转换运算符，如 `intVar = (int) doubleVar`

强制转换会造成数据丢失

内置类型转换方法 method description

<code>To Boolean</code>	转换为布尔型 (如果可能)
<code>To Byte</code>	类型转换为字节类型
<code>To Char</code>	类型转换为单个 Unicode 字符类型 (如果可能)
<code>To SByte</code>	类型转换为有符号字节类型
<code>To Int16</code>	类型转换为 16 位有符号整型值
<code>To Int32</code>	类型转换为 32 位有符号整型值
<code>To Int64</code>	类型转换为 64 位有符号整型值
<code>To UInt16</code>	类型转换为 16 位无符号整型值
<code>To UInt32</code>	类型转换为 32 位无符号整型值
<code>To UInt64</code>	类型转换为 64 位无符号整型值
<code>To Single</code>	类型转换为单精度浮点数类型
<code>To Double</code>	类型转换为双精度浮点数类型
<code>To Decimal</code>	将整型值 / 浮点型 转换为十进制类型
<code>To String</code>	类型转换为字符串类型
<code>To DateTime</code>	将整型值 / 字符串转换为日期-时间结构
<code>To Type</code>	类型转换为指定 Type 类型

如 `Enum.ToObject(Type, Int64)`

~~UAS 中~~ ~~ATP + CF~~
 通用类库 <T> T[] ~~ToArray(T[] a)~~

C# - P6

类型转换

在C#中，有不同的方法可以将其他类型的值转换为整型值

如 Convert.ToInt32(arg) / int.Parse(str) / (int) d

从传入参数的类型看：

Convert.ToInt32(arg)：可以传入多种类型的参数，继承自Object的对象

除数字类型外还有 bool, DateTime 等

int.Parse(str)：只能传入字符串，且只能是整型字符串

即可以通过整型 ToString() 方法得到的字符串

如果传入的字符串是浮点型的

int.Parse() 抛出错误 FormatException

(int)：只能转换其他数字类型，如 float, double, uint

从对空值 NULL 的处理方式看：

Convert.ToInt32() / int.Parse() 不接受传入参数为空值 NULL

会抛出异常，可以捕获异常并处理

Convert.ToInt32()：实际上是在转换前先做判断

如果传入参数为空值 NULL

则 Convert.ToInt32() 返回 0

int.TryParse()：实际上是对 int.Parse() 做异常处理

如果传入参数不是整型字符串，如浮点型或空值 NULL

则捕获抛出的异常并返回 0

从对数字进行四舍五入的处理方式看：

int.Parse() 不接受浮点型字符串，只接受整型字符串

于是不存在对四舍五入的处理

(int)：对于传入参数包含小数部分的情形

不进行四舍五入而直接忽略小数部分

Convert.ToInt32()：如果传入参数包含小数部分，则进行四舍五入

但是对于参数正好处于两个整数中间

则返回两个相邻整数中的偶数

如 Convert.ToInt32(3.5) → 4

Convert.ToInt32(4.5) → 4

从处理溢出的方式看：

int.Parse() / Convert.ToInt32()：对于值相比 Int32 过大或过小的情形

抛出溢出错误

(int)：对于溢出，不报错而直接返回 -1

C# - P7

时间 - 2023.8.1

变量

在 C# 中，变量只是供程序操作的存储区的名。每个变量都有一个特定的类型，决定变量的内存大小和布局。范围内的值可以存储在内存中，可以对变量进行一系列操作。

值类型：整数类型：sbyte/byte/short/ushort/int/uint/long/ulong/char
浮点型：float / double
进制类型：decimal
布尔类型：bool
空类型：可为空值的数据类型

变量定义的基本语法：`<data-type> <variable-list>;`

`<data-type>`：必须是一个有效的 C# 数据类型

`<variable-list>`：由一个或多个以逗号分隔的标识符组成

变量通过在等号后跟随一个常量表达式进行初始化（赋值）。

基本语法为：`<data-type> <variable-list> = <constant-expression>;`

方法的局部变量必须在代码中显式地初始化

随后才能在语句中使用变量的值。

编译器会通过方法检查所有可能的路径。

如果检测到局部变量在初始化之前使用了变量的值。

即抛出错误 `use of unassigned local variable`

在 C# 中，表达式被分为两种 lvalue / rvalue

`lvalue (左值)`：表达式可以出现在赋值语句的左边或右边。

`rvalue (右值)`：表达式只能出现在赋值语句的右边，不能出现在左边。

`lvalue` 可以被赋值，而 `rvalue` 不可被赋值。

`lvalue` 出现在赋值语句的右边时，`lvalue` 是已被赋值的状态。

静态变量

在 C# 中，没有全局变量 (global variable) 的概念。

所有变量必须由所属类的实例进行操作。

虽然提升了安全性，但某些情形下则不方便。

静态变量 (static variable) 用于保存类的公共信息。

基本语法为：`static <data-type> <variable-list> = <constant-expression>;`

C# - P8

常量

在C#中，常量是程序执行期间不会改变的固定值

可以看作常规的变量，区别是值在定义后不能被修改

可以是任何基本数据类型：整型/浮点数/字符/字符串

也可以是枚举常量

整数常量

在C#中，整数常量可以是十进制/八进制/十六进制的常量

从C#7.0开始支持二进制表示的整数常量

前缀：用于指定整数常量的基数 0x/0X

0表示八进制，0b表示二进制，0x表示十六进制

十进制没有前缀，但是注意以0开头的整数常量视为八进制

后缀：u/U表示无符号(unsigned)整数

l/L表示长整型值(long)

后缀可以以任意顺序组合，但不能出现重复后缀

浮点常量

在C#中，浮点常量由整数部分、小数点、小数部分、指数部分组成

如 6.022 e 23，阿伏伽德罗常数(Avogadro constant)

e/E：用于表示有符号的指数部分

后缀：f/F表示单精度浮点数(float)

d/D表示双精度浮点数(double)

m/M表示精确十进制值(decimal)

也可以不加后缀而由变量类型指定，默认指定为 double

注意：当使用小数形式表示的浮点常量时

必须包含小数点，和指数部分至少一种

即如 210 为非法的浮点常量(现在也是合法的)

可以省略整数部分

当使用指数形式表示的浮点常量时

必须包含整数部分和小数部分至少一种

即如 .e23 属于无意义的表达式 invalid expression

分隔符

在C#中，数字分隔符用于提高数字常量的可读性

C#中的数字分隔符为下划线“_”

可以出现在数字中间的任何位置，可以连续出现，但不能作为部分的前缀/后缀

可以用于整数常量和浮点常量

C# - Pg

字符常量

在 C 中，字符串常量表示为括在单引号('')里
可以存储在一个简单的字符类型变量 (char) 中

字符常量可以是：普通字符，如'j'\n'Windows'

转义字符，如 '\n'

通用字符，如 'lu02c0'。

转义字符	description	参考中文字典
\\"	字符 \	单引号
\'	字符 '	双引号
\"	字符 "	单引号
\?]	字符 ?	问号
\a	Alert 或 bell	警报
\b	退格键 (backspace)	退格
\f	换页符 (form feed)	换页
\n	换行符 (newline, line feed)	换行
\r	回车 (return, carriage return)	回车
\t	水平制表符 tab	制表符
\v	垂直制表符 tab	制表符
\ooo	至三位八进制数 (C语言支持, C#不支持)	八进制数
\xhh...	一个或多个数字的十六进制数 (超过两位可能无法显示)	十六进制数

字符串常量

在 C# 中，字符串常量表示为括在双引号 (" ") 或 @ 双引号 (@ " ") 里

④“中为延字字符串，转义字符串会看作是一般字符串”

并且可以使用回车 / 换行 / 其他特殊字符

静态常量

也能编译时常量，在C#中通过关键字 const 定义

基本语法结构为：主语 + 谓语 + 补语 + 状语 + 题旨句尾

const <data-type> <constant-name> = <constant-expression>;

静态常量在编译时即确定了常量的值

必须在声明常量时进行初始化

表初始化之后不能进行更改

可以在类和方法中定义常量

适用于：取值永远不变的常数

C# - P10

动态常量

也称运行时常量，在C#中通过关键字 readonly 声明和定义

基本语法结构为：

`readonly <data-type> <constant-name> [= <expression>];`

动态常量在程序运行时才确定常量的值

可以在常量声明或构造函数中进行动态常量的初始化

只要在类中定义

通常应该优先使用 readonly 常量，少数情况使用 const 常量

算术运算符

在C#中，提供了与C++和Java类似 的算术运算符

加法(+)，减法(-)，乘法(*)，除法(/)，取模(%)

前缀自加/自减(++a/-a)：先进行自加/自减，再进行表达式运算

后缀自加/自减(a++/a--)：先进行表达式运算，再进行自加/自减

关系运算符

在C#中，提供了与C++和Java类似 的关系运算符

LT(<)，LE(<=)，EQ(==)，NE(!=)，GE(>=)，GT(>)

逻辑运算符

在C#中，提供了与C++和Java类似 的逻辑运算符

逻辑与(&&)，逻辑或(||)，逻辑非(!)

位运算符

在C#中，提供了与C++和Java类似 的位运算符

按位与(&)，按位或(|)，按位非(~)，按位异或(^)

左移(<<)，右移(>>)

赋值运算符

在C#中，提供了与C++和Java类似 的赋值运算符(=)，并同样可与其他运算符组合使用

与算术运算符：+=，-=，*=，/=，%=\n与位运算符：&=，|=，^=，<<=，>>=

operator

description

sizeof()

返回数据类型以byte表示的尺寸

typeof()

返回参数对象的类型

&a

返回变量的地址

*a

返回地址指向的变量

a is class

判断变量是否为类的对象

a as class

等同于Java的instanceof

强制类型转换，即使转换失败也不会抛出异常

C# - P11

条件运算符 在C#中，提供与C++和Java类似的基本语法为 `<boolean expression> ? <value if true> : <value if false>`，
注意，虽然C#中的条件运算符不需要像Java一样要求`<value>`的类型相同，但是类型间必须有隐式的类型转换。

否则无法确定条件表达式的类型

type of conditional expression cannot be determined

because there is no implicit conversion

运算符优先级	运算符	操作符	结合性
高	后缀	<code>()</code> , <code>[]</code> , <code>-></code> , <code>.</code> , <code>++</code> , <code>--</code>	左结合
	一元	<code>+</code> , <code>-</code> , <code>!</code> , <code>~</code> , <code>++</code> , <code>--</code> , 类型转换(type), <code>sizeof</code>	右结合
	乘除	<code>*</code> , <code>/</code> , <code>%</code>	左结合
	加减	<code>+</code> , <code>-</code>	左结合
	移位	<code><<</code> , <code>>></code>	左结合
	关系	<code><</code> , <code><=</code> , <code>>=</code> , <code>></code>	左结合
	相等	<code>==</code> , <code>!=</code>	左结合
	按位与	<code>&</code>	左结合
	按位异或	<code>^</code>	左结合
	按位或	<code> </code>	左结合
	逻辑与	<code>&&</code>	左结合
	逻辑或	<code> </code>	左结合
	条件	<code>?:</code>	右结合
低	赋值	<code>=</code> , <code>+=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , <code><<=</code> , <code>>>=</code> , <code>&=</code> , <code>^=</code> , <code> =</code>	右结合
	逗号	<code>,</code>	左结合

可空类型 (`nullable`)，在C#中，提供了一个特殊的 `System.Nullable` 类型

`Nullable<type>` 表示在 type 类型的取值正常范围之外，附加一个 null
如 `Nullable<bool>` 的取值范围是 `true / false / null`

单问号? 为可空类型修饰符，编译器会将`<T>?` 编译为 `System.Nullable<T>` 的形式

提高可空类型的语义结构为 `<T>? <variable_name> = <value>;`

如 `int? i = 11;` 等价于 `Nullable<int> i = new Nullable<int>(11);`

数值/布尔值的可空类型常用于处理数据库或其他包含可能未赋值元素的数据类型时，即数据库中的某些字段可以存储值，也可以为空值 null

C# - P12

CS9 - 2020

null 合并运算符

在 C# 中, null 合并运算符用于定义可空类型和引用类型的默认值。

为类型转换定义一个预设值, 防止可空类型的值为 null。

null 合并运算符 将操作数隐式地转换为另一个可空/不可空的值类型的操作数的类型。

基本语法结构为 <nullable-operand> ?? <expression>

如果 nullable-operand 为 null, 则计算并返回 expression 的值。

如果 nullable-operand 不为 null, 则返回其自身。

例 System.Nullable<Int32> int1 = 11;

int1 ?? int2 = null; → int1 ?? int2 = 11

int1 ?? int2 = 11; → int1 ?? int2 = 11

null 合并运算符是右结合运算符, 也可以看作是一种短路设计。

例 a ?? b ?? c 等价于 a ?? (b ?? c)

null 检查运算符

在 C# 中, 提供了 null 检查运算符 (?.)。

基本语法结构为

<nullable-object> ?. <member>

如果 nullable-object 为 null, 则不进行调用成员的运算, 而直接返回 null。

条件结构

在 C# 中, if ... else if ... else 结构的基本语法结构为:

if (<boolean-expression>) {
 <Statements>
}

[else if (<boolean-expression>) {
 <Statements>
}] * 可以有 0 个或多个 else if 结构

[else {
 <Statements>
}] 可以有 0 个或 1 个 else 结构

当任何一个 if / else if 结构被执行, 则之后的 else if / else 被忽略。

switch 结构

在 C# 中, switch 结构的基本语法结构为:

switch (<expression>) {

[case <constant-expression> : {
 <Statements>
}] + 可以有任意数量的 case 结构

[break;]

[default : {
 <Statements>
}]

[break;]

C# - P13

switch 结构 在 C# 中，switch 结构必须遵循如下规则

switch 语句中的 <expression> 必须或者是整型 / 枚举类型

或者是一个 class 类型，且必须包含一个单一的转换函数

转换函数将其转换为整型 / 枚举类型

case 后的 <constant-expression> 必须是一个常量

且类型必须与 switch 语句中的 <expression> 相同

当被测试的变量等于 case 中的常量时

执行 case 后跟随的语句，直到遇到 break 为止

当遇到 break 语句时，终止 switch 结构，并跳转至 switch 结构的下一行

并非每个 case 都必须包含 break 语句

如果 case 中没有执行的语句，则可以不包含 break 语句

控制流将继续判断后续的 case，直到遇到 break 语句

switch 结构的结尾可选地有一个 default case

当 switch 中所有的 case 均不执行时，执行默认的语句块

default case 中可以不包含 break 语句 (VS 中必须包含)

注意在 C# 中，不允许从一个 case 标签显示地贯穿到另一个 case 标签

即如果 case 中有执行的语句，则不允许继续执行到下一个 case

必须在下一个 case 之前包含 break 语句或其他跳转语句

如 goto case <constant-expression> / goto default

否则会出现编译错误

control cannot fall through from one case label to another

while 循环结构，在 C# 中，while 循环结构的基本语法结构为

while (<condition>) { }

<Statements>

}

condition 可以是任意表达式，且当为任意非零值都为真

do...while 循环，在 C# 中，do...while 循环结构的基本语法结构为

do { } while (<condition>);

<Statements>

}

与 while 循环相比，do...while 循环中的语句块至少会执行一次

C# - P14

for 循环

在 C# 中, for 循环的基本结构为

```
for (<init>; <condition>; <increment>) {  
    <Statements>  
}
```

init: 可选地初始化循环控制变量, 即可以留空

在循环开始时执行且只执行一次

condition: 循环的控制条件, 可选地定义

注意仅当 condition 计算值为 false 时退出循环

跳转至 for 循环结构之后的下一条语句

(注意如果留空时, 则视为 true 并开始执行下次迭代)

此时需要在循环中显式地定义退出循环

increment: 可选地定义如何更新循环控制变量

在 for 循环体中的 statements 结束后被执行

执行之后即进行下一次迭代前的控制条件判断

于是可以构造无限循环

即 for (;;) { 等价于 while (True) {

foreach 循环 在 C# 中, foreach 循环的基本结构为

```
foreach (<data-type> <element> in <iterable>){  
    <Statements>
```

<element> 为类型为 data-type 的对象

<iterable> 为包含数据为 data-type 类型的对象

break 语句 在 C# 中, break 语句可以用于退出当前的语句块

在循环结构中时, 停止循环并跳转至循环结构之后的下一条语句

在 switch 结构中, 从 switch 的一个 case 中退出

如果存在循环的嵌套, 即只退出其所处的循环结构

continue 语句 在 C# 中, continue 语句用于跳转至循环迭代的开始

对于 for 循环, 会跳转至循环增量与条件次试的语句

C# - P15

封装

在C#中，封装定义为“将一个或多个项目封闭在一个物理的或者逻辑的包中”

在面向对象程序设计方法论中，封装防止对实现细节的访问

抽象允许相关信息可视化，封装使开发者实现所需级别的抽象

C# 封装根据需要 ~~通过~~ 通过访问修饰符设置使用者的访问权限

Public：所有对象都可以访问

允许一个类将标记为 Public 的成员变量 / 成员函数暴露给其他函数 / 对象
任何 Public 成员均可以被外部的类访问

Private：对象本身在对象内部可以访问

允许一个类将标记为 Private 的成员变量 / 成员函数 对其他函数 / 对象进行隐藏
只有同一个类中的函数可以访问其 Private 成员
即使是类的实例也不能直接访问其内部的 Private 成员

Protected：只有该类的对象及其子类的对象可以访问

允许子类访问其基类中标记为 Protected 的成员变量 / 成员函数

Protected 访问修饰符有助于实现继承

internal：同一个程序集内的对象可以访问

允许一个类将标记为 internal 的成员变量 / 成员函数暴露给当前程序中的其他函数
带有 internal 访问修饰符的任何成员

可以被定义在该成员所定义的应用程序内的任何类或方法访问

protected internal：访问限于当前程序集或派生自所在类的类型

允许标记为 protected internal 的成员变量 / 成员函数

在类，派生类或者包含该类的程序集中访问

可以视为 protected 和 internal 的并集

protected internal 访问修饰符也用于实现继承

不严格地描述为：父亲 A，儿子 B，妻子 C，私生子 D，以及一件关于父亲 A 的事

public：所有人都知道， private：只有 A 自己知道

protected：A, B, D 知道（父亲 A 及其所有儿子 B, D 知道，但妻子 C 不知道）

internal：A, B, C 知道（同一家庭中的父亲 A, 儿子 B, 妻子 C 知道，不在家庭的私生子 D 不知道）

protected：AB, C, D 知道，而其他人不知道