C#

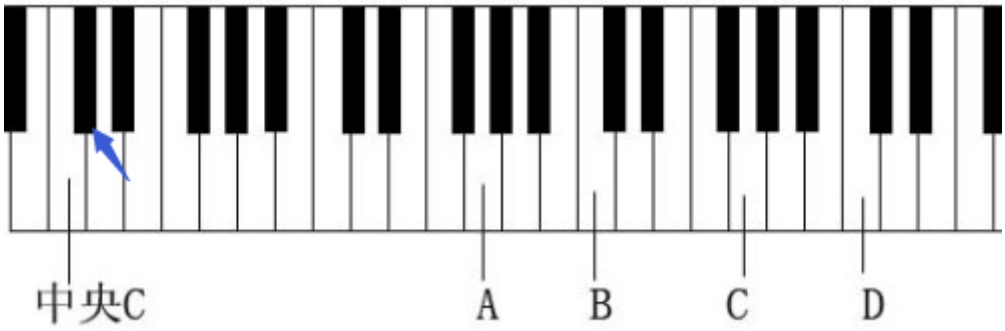
无03 唐昌礼

目录

学习C#之前

“C#”怎么读

C#，读作 “C Sharp”，是一种运行在.NET CLR上的，安全的，稳定的，**优雅的**，面向对象的高级编程语言。C#按理来说写作C♯，横线向上，代表升半音符号（钢琴C键旁边的那个黑键），但后来，官方文档也用 “Shift + 3” 了。



程序的空间资源

堆：用来保存进程运行时动态分配的内存空间。

栈：保存运行的上下文信息；在函数调用时保存被调用函数的形参和局部变量。

代码段、数据段……

内存泄露是发生在堆中的，引入垃圾回收机制可以防止这一问题的发生。

有关垃圾回收算法，可以看这篇weekly，或 书籍《垃圾回收的算法与实现》中村成洋 相川光 著，丁灵 译，北京 : 人民邮电出版社，2016。

SAST Weekly | 垃圾回收——程序员的梦想 (qq.com)

.NET 简介

.NET 是什么

.NET 是微软公司发布的应用程序框架，用以减轻软件开发人员的工作。它包括一系列类库、运行时等内容。

在生成一个 .NET 程序时，程序员编写的代码暂时不翻译成本地的机器语言，而是先翻译成一种其他的语言，叫做“微软中间语言（MSIL, Microsoft Intermediate Language）”。在执行该微软中间语言的可执行文件时，将启动对应 .NET 框架的“公共语言运行时（CLR，Common Language Runtime）”，由该 CLR 将 MSIL 转化为机器码执行。

可见，CLR 就仿佛一台独立于物理机器的“虚拟机”，MSIL 语言的程序可以在 CLR 上运行。由于这个机制，我们可以得到很多便利，例如可以轻松实现垃圾回收（GC, Garbage Collection）、跨平台（虽然微软起初并没有这个目的），等等。

.NET FrameWork？.NET Core？ .NET?

起初，微软推出的框架名字叫“.NET Framework”，只能在 Windows 上运行以推广 Windows，现在 .NET Framework 也是 Windows 10 的预装品。.NET Framework 可以支持多种开发框架，例如 Winform、ASP、WPF，等等。后来，微软改变了策略，拥抱开源与跨平台，推出了“.NET Core”框架，该框架开源，并且支持多种操作系统，并且随着 .NET Core 版本的更迭，.NET Framework 支持的功能也逐渐向 .NET Core 迁移。当 .NET Core 3.1 出现后，功能已经接近完备，接近甚至超过了 .NET Framework，微软决定下个版本扔掉 Core，下个版本直接改名“.NET”，与 .NET 同名以表明它将是以后 .NET 的主要发展方向。况且考虑到 .NET Framework 最新版本已经是 4.x，因此为了防止发生混

淆，.NET 版本跨过 4.x，而直接于 2020 年下半年推出 .NET5，并声称每年推出一个 .NET 版本。2021 年 2 月 17 日，微软的开发者博客中推出了 .NET6 的第一个预览版。2021年11月8日，微软正式发布了 .NET 6 及其一系列内容，推出了 C# 10、F# 6 和 PowerShell 7.2。

此外，在 .NET Core 出现之前，开源社区（非微软官方）自己创造了一个支持 C# 语言的跨平台运行时，称作“mono”。为了便于管理如此繁杂的 .NET 体系，微软推出了 .NET Standard 规范，作为各个 .NET 运行时遵循的准则。不管是 .NET Framework，还是 .NET Core，还是 mono，都必须实现 .NET Standard 作为其公共子集。

.NET 现在可以支持多种语言或被多种语言进行调用，例如 C#、Visual Basic.NET、F#、PowerShell、C++/CLI，等等。

初识C#

使用.NET CLI创建C#应用程序

我们首先需要在计算机上安装.NET Core SDK，之后就可以使用.NET CLI创建C#程序。

要查看计算机中.NET的详细信息，可以在命令行中输入

```
dotnet --info
```

要创建一个新的项目，.NET为我们提供了许多模板。比如我们要创建一个控制台应用程序，只需在命令行输入

```
dotnet new console --output ./Exp
```

dotnet就为我们创建一个控制台应用程序的模板。

然后可以使用run命令编译并执行该程序。

```
dotnet run --project Exp
```

更多细节请参考微软官方文档：dotnet command - .NET CLI | Microsoft Docs

使用.NET CLI在日常开发中是一件相对麻烦的事。如果是在Windows上开发，我们可以使用 “宇宙第一IDE” Visual Studio 来完成上述过程，而无须使用命令行。

第一个C#程序

Main方法

一个基本的C# HelloWorld程序如下

```
using System;

namespace hello
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

运行上述代码，程序将输出：Hello World!

`using System`指的是使用`System`命名空间。`System`是.NET众多类库所在的命名空间，例如`Console`类。我们使用了该命名空间，就能够去使用该命名空间下的众多类。如果不写`using System`，如果想要使用`Console`类，一般来说，就需要加上前缀，即：`System.Console.WriteLine("Hello World!");`。

> 注：.NET 6 默认开启 implicit using 选项，会默认 using System 等一系列命名空间

`Main`方法是程序的主入口。由于程序开始运行时没有实例化任何对象，因此`Main`方法必须被定义为静态方法。常见的`Main`方法形式为

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

一般来说，习惯上`Main`方法都被定义在`Program`类中。

顶级语句

打开Visual Studio 2022，创建一个C#控制台应用程序，使用.NET 6框架，你就会发现Program.cs中的代码如下

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");

```

这是C# 9.0（.NET 5.0）之后的新特性，即顶级语句。C#应用程序可以没有`Main`方法，程序也不使用`Main`方法作为入口点，而是将顶级代码段作为入口点。

C# 类型

C# 是一种面向对象的强类型语言，具有一个庞大的类型系统。C# 类型系统的特点是，一切类型（除指针类型）均继承自`object`（`System.Object`）类，即`object`类型是一切类型（除指针类型）的基类。

C# 的类型分为两种：值类型和引用类型（以及指针类型）。

值类型

值类型是一类比较简单的类型，直接在内存栈上或其他内存位置储存数值。值类型分为两种：结构类型和枚举类型。

内置数据类型

内置的数值类型均属于结构类型（如 `int` 实际上是 `struct Int32`）。C# 内置的数值类型有：

C# 类型名称	范围	对应的 .NET 类型	备注
`sbyte`	-128 ~ 127	`System.SByte`	8 位有符号整数
`byte`	0 ~ 255	`System.Byte`	8 位无符号整数
`short`	-32,768 ~ 32,767	`System.Int16`	16 位有符号整数
`ushort`	0 ~ 65535	`System.UInt16`	16 位无符号整数
`int`	-2,147,483,648 ~ 2,147,483,647	`System.Int32`	32 位有符号整数
`uint`	0 ~ 4,294,967,295	`System.UInt32`	32 位无符号整数
`long`	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807	`System.Int64`	64 位有符号整数
`ulong`	0 ~ 18,446,744,073,709,551,615	`System.UInt64`	64 位无符号整数
`nint`	取决于平台	`System.IntPtr`	32 或 64 位有符号整数
`nuint`	取决于平台	`System.UIntPtr`	32 或 64 位无符号整数
`float`	—	`System.Single`	IEEE754 单精度浮点数
`double`	—	`System.Double`	IEEE754 双精度浮点数
`decimal`	±1.0E-28 ~ ±7.9228E28	`System.Decimal`	16 个字节小数
`bool`	true, false	`System.Boolean`	布尔类型，1 字节
`char`	U+0000 ~ U+FFFF	`System.Char`	Unicode UTF-16 字符类型；2 字节

内置的数字类型有对应的文本类型，例如整数 `2021`、`2021_0714UL`，双精度浮点数 `3.1415926`，单精度浮点数 `3.56f`、`3.14F`、小数 `3.50m`、`3.75M`、字符型 `'c'`，等等。

此外在整数文本的前面加上前缀可以指定进制：十六进制为 `0x` 或 `0X`，二进制（C# 7.0）`0b` 或 `0B`。

自定义结构类型

可以使用 `struct` 自定义结构类型

```
public struct Point
{
    public int x;
    public int y;
}
```

使用 `struct` 定义的类型均为值类型。

结构体成员x，y都被赋予了 `public` 访问权限。对于结构体，其成员默认为 `private`。结构体不能被继承，因此其成员不能使用 `protected` 修饰。

`struct` 前的 `public` 修饰符下面再讲。

枚举类型

枚举类型是一种不同于结构类型的值类型。需要用 `enum` 关键字进行定义。

```
public enum Color
{
    Red = 0,
    Blue = 1,
    Yello = 2,
    Purple = 8,
    Black   // Equivalent to "Black = 9", 8 + 1 by default.
}
```

定义值类型变量

定义一个结构类型变量有两种方式，一是直接使用“类型名+变量名”的方式，二是使用 `new` 关键字，两者完全等价。

```
int x;              // Equivalent to "int x = 0;" or "int x = default(int);"
int y = new int();  // Equivalent to "int y;"
int z = 4;
Point pt1;
Point pt2 = new Point();    // Equivalent to "Point pt2;"
```

如果值类型变量具有构造方法，可以用 `new` 关键字。

```
public struct Point
{
    public int x;
    public int y;
    public Point(int x_, int y_)
    {
        x = x_;
        y = y_;
    }
}
```

```
Point pt = new Point(2, 3);
```

关于值类型

`System.ValueType` 类（该类派生自 `object` 类）是所有值类型的直接或间接基类，其中结构类型（`int`、自定义结构类型，等等）都直接派生自 `System.ValueType` 类，而枚举类型则直接派生自 `System.Enum` 类（该类派生自 `System.ValueType` 类）。

值类型不可被继承。

引用类型

什么是引用类型

引用类型对象实际上是两部分：一部分是对象真正的数据，开辟在托管堆上，另一部分是真正对象的引用，储存在内存栈或其他任何位置。

常用的引用类型包括`object`、`string`，或是程序员自己定义的类（`class`），接口（`interface`），委托（`delegate`），事件（`event`）等等。

实例化一个引用类型对象，需要使用`new`关键字

```
class Exp
{
    public int x;
}
Exp A1 = new Exp(); // 创建一个Exp对象，A1是该对象的引用
var A3 = new Exp(); // 同上
Exp A2 = new(); // 同上，C# 9.0 新增
Exp A4; // 创建了一个Exp类型的引用，但并没有指向任何Exp对象！
```

装箱与拆箱

`object`是C#的一个关键字，表示类型`System.Object`，它是所有类型（除指针类型）的公共基类，是一个引用类型。因此，object类型可以指向所有类型的托管对象。

我们知道，值类型不存在引用，那么一个`object`类型该如何指向值类型呢？

这就是装箱（Boxing）与拆箱（Unboxing）的机制。

当我们把一个值类型强制转换为 `object` 类型时，会在托管堆上 `new` 出一个 `object` 对象，用来存储这个值类型，这样就把值类型转换成了 `object` 类型。这个过程叫做“装箱”。

```
int x;
object o = x;
int y = (int)o;
```

将一个装箱产生的`object`对象转回值类型的过程，就叫“拆箱”，这个过程会将储存在托管堆上的数据拿出来。

这样，通过装箱和拆箱，我们就实现了让每一个值类型都可以被转换为 `object` 类型。这种统一的类型系统为我们编程带来了巨大的便利。

但是，频繁的装箱和拆箱也会有较大的性能损失，所以尽量避免大量的装箱与拆箱操作。

字符串

`string` 是一个 C# 关键字，表示 .NET 类型 `System.String` 类，即是一个类类型。

字符串文本采用双引号 `""` 引起来，其中 `\` 是转义字符，一些字符需要用 `\` 实现。例如 `\n` 为换行符、`\t` 为制表符，`\\` 为字符 `'\'` 本身、`\"` 是双引号，例如：

```
Console.WriteLine("I said, \"I am happy.\"");
```

将会输出：

```
I said, "I am happy."
```

如果不想使用转义字符，那么可以使用 C# 的逐字字符串。逐字字符串以 `@` 开头，并用双引号引起。逐字字符串中 `\` 不再被解释为转义字符，而且逐字字符串中可以含有换行。唯一的例外是逐字字符串中不能由双引号 `""`。

此外 C# 还支持字符串内插（C# 6.0），可以在字符串中插入变量。内插字符串以 `$` 开头，用双引号引起。插入到字符串中的变量放在一对花括号 `{}` 中。

```
Console.WriteLine(@"D:\new\input.txt");
Console.WriteLine(@"#include <stdio.h>
int main()
{
    return 0;
}");
int x = 4;
int y = 5;
Console.WriteLine($"The position is ({x}, {y}).");
```

程序输出：

```
D:\new\input.txt
#include <stdio.h>
int main()
{
    return 0;
}
The position is (4, 5).
```

值类型与引用类型的对比

引用类型必须使用`new`关键字，才能真正意义上的创建一个对象。

```
string s1;
string s2 = "123";
string s3 = new string("456");
```

上述代码中，s1仅仅创建了一个`string`的引用，并没有实例化任何对象，也没有在托管堆上开辟空间。s2，s3都实例化了一个字符串对象，并在托管堆上开辟了空间。

值类型则不存在引用

```
int x;
int y = new int();
```

尽管`y`是用`new`创建的，但是其实和直接写`int y`没什么区别。`x,y`都是真正的`int`变量，不是引用。

值类型进行赋值是将该值类型的对象进行复制，而引用类型的复制是复制引用，而非对象本身。典型的例子就是函数传参，有如下示例代码

```
A a = new A(1);
B b = new B(1);
AddOne(a, b);
Console.WriteLine($"a.x={a.x}, b.x={b.x}");

void AddOne(A a, B b)
{
    a.x++;
    b.x++;
}

public class A
{
    public int x;
    public A(int x) { this.x = x; }
}
public struct B
{
    public int x;
    public B(int x) { this.x = x; }
}

```

输出结果为：a.x=2, b.x=1

自动类型推导

使用`var`关键字可以让编译器自动推导变量类型。

```
var x = 4;  // x 是 int 类型
var s = "Hello, world"; // s 是 string 类型
var o = new object();   //  o 是 object 类型
```

杂项

类型转换

C# 默认可以将低精度类型隐式转换到高精度类型，而将高精度类型转换到低精度类型需要强制类型转换：

```
int x; ulong y;
y = x;      // ok
// x = y;   // error
x = (int)y; // ok
```

变量定义

C# 变量名只能包含数字、字母（汉字等文字也属于字母）、下划线，并且不能以数字开头。也可以在变量名前加上 `@` 以强调它是变量名，但是一般情况下不加，例如 `int @val = 4;` 与 `int val = 4;` 完全等价。

C# 包含一系列关键字（Keyword），例如 `int`、`double`、`if`、`while`，等等，如果要把变量名定义成关键字，则必须加上 `@`：

```
int @int;   // 一个名字叫 int 的 int 型变量
```

GC

动态内存开辟在托管堆上。所谓托管堆，顾名思义，就是被托管了，开辟的空间资源不需要程序员手动回收，.NET提供了垃圾回收（GC）的机制完成GC。

C# 基础

输入输出

通过控制台的输入输出需要用到`System.Console`类。

输入

使用`Console.ReadLine`，`Console.Read`方法在控制台输入数据。

```
string? in1 = Console.ReadLine() // 读入一行数据，返回string?类型
int in2 = Console.Read()  // 读取下一个字符，返回int类型（ascii码）
```

输入时，我们有时希望输入一个整数（或浮点数），在程序中用整数（或浮点数）存储，而不是字符串。可以使用`System.Convert`类进行类型转换。

```
// using System;
string s = Console.ReadLine();
int x = Convert.ToInt32(Console.ReadLine());
double d = Convert.ToDouble(Console.ReadLine());
```

输出

使用`Console.WriteLine`，`Console.Write`方法在控制台输入数据。区别是`WriteLine`会在末尾加一个换行而`Write`不会。

输出变量的值，一般可以使用字符串内插输出，也可以使用老式写法格式输出。

```
int x = 4, y = 5, z = 6;
Console.WriteLine($"z = {z}, x = {x}, y = {y}");
Console.WriteLine("z = {2}, x = {0}, y = {1}", x, y, z);
```

流程控制与基本运算

C# 控制结构：

- 条件分支：`if`、`switch`
- 循环：`while`、`do while`、`for`、`foreach`
- 跳转：`goto`

特别说明：`switch` 语句中，如果 `case` 标签后有语句，则必须要写 `break`，除非 `case` 后没有任何语句。

C# 提供的运算符有：算术运算符（+、-、*、/）、逻辑运算符（&&、||、!）、位运算符、三目运算符、null 合并运算符，等等。

有趣的？

C#提供`?`的多种使用方式

- 三目运算符：`a ? b : c`
- `null`检查运算符：`?.`
  - 语法：`obj?.xxx`，表示如果`obj`不为`null`，则使用`obj`的`xxx`属性或方法；否则值为`null`

```
void Print(object? obj)
{
    Console.WriteLine(obj?.ToString());
}
```

- `null`合并运算符：`??`
  - 语法：如果左边的表达式不为`null`，则值为左边的值；若左边的表达式为`null`，则值为右边的值。

```
void Print(object? obj)
{
    Console.WriteLine(obj?.ToString() ?? "It's null!");
}
```

- `null`合并赋值运算符`??=`
  - 语法：若左边的值是`null`，则给它赋右边的值。

```
o ??= new object();
```

- 可为`null`的类型
  - 可以使用`?`定义一个可为`null`的类型

```
string s1;   // s1理论上不允许为null
string? s2;  // s2允许为null
```

  - 值类型也可定义为可为`null`的类型，如`int?, double?`。可为`null`的值类型在.NET中都是泛型结构体`System.Nullable<T>`的实例。

数组

C# 中，数组属于引用类型。数组均继承自 `System.Array` 类。

一维数组

定义一个一维数组的基本语法如下

```
type[] arr = new type[array_size]
```

可以在定义数组时为其赋初值，以 `int` 为例

```
int[] arr1 = new int[5];        // 定义一个长度为 5 的数组，每个元素初始值均为默认值 0
int[] arr2 = new int[5] { 1, 2, 3, 4, 5 };  // 定义一个长度为 5 的数组，初始值分别为 1, 2, 3, 4, 5
int[] arr3 = { 1, 2, 3, 4, 5 };     // 简略写法
```

可以通过 `[]` 访问其元素，用 `Length` 获取元素个数：

```
for (int i = 0; i < arr1.Length; ++i)
{
    Console.WriteLine(arr1[i]);
}
```

也可以使用 `foreach` 语句遍历：

```
foreach (var i in arr1)
{
    Console.WriteLine(i);
}
```

注：通过 `foreach` 是无法修改数组中的元素的，只能访问其值。

多维数组

多维数组的定义方式如下（以二维数组为例）

```
type[, ] = new type[m, n];
```

同样可以在定义时初始化

```
int[,] arr = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };
```

可以通过 `Length` 获取数组的长度，通过 `GetLength(n)` 获取数组第 `n` 维的长度。

交错数组

C# 支持嵌套的数组，即数组的每个元素都是一个数组。这样的数组称为“交错数组”。以一个每个元素都是一维数组的一维交错数组为例（其他维度的数组类似）：

```
int[][] arr = new int[2][]
{
    new int[3]{ 1, 2, 3 },
    new int[2]{ 4, 5 }
};
```

上述定义了一个具有两个元素的交错数组。

需要注意的是，数组也是引用类型，因此交错数组的每个元素都要用 `new` 产生，例如下面的：

```
int[][] arr = new int[2][];
```

此处 `arr` 具有两个元素，每个元素都是一个 `int[]` 的引用。但是由于并没有用 `new` 为每个元素创建托管对象，因此每个引用都是 `null`，并没有指向任何数组。

C# 面向对象编程

命名规范

大驼峰命名法

名称中各单词首字母大写。

小驼峰命名法

名称中第一个单词首字母小写，后续每个单词首字母大写。

类 class

定义一个类

类属于引用类型，可以由class关键字定义。

```
public class ClassName { ... }
```

这里定义了一个类，用访问修饰符 `public` 修饰，与其对应的是 `internal`。

`internal`：该类只能在本程序集内访问。

`public`：该类可以被任意访问。

若该类为嵌套类，可以使用更多访问修饰符，详见“字段”部分。

类命名规范

类一般采用大驼峰命名法。

字段 field

在类中可以定义自己的字段，类的字段即为类"内部的变量"。

```
public class Person
{
    public string name;
    private int age;
}
```

上述定义中，访问修饰符`public`、`private`分别指定`name`为公有字段，`age`为私有字段。

访问修饰符共包括

- `private`：这个字段只能够被本类所访问
- `public`：这个字段可以被随意访问
- `protected`：这个字段可以被本类及其派生类访问
- `internal`：这个字段可以在本程序集内随意访问
- `protected internal`：既可以被本来及其派生类访问，又可以在本程序集内随意访问

字段命名规范

字段一般采用小驼峰命名法。

字段的默认值

C# 的字段可以定义默认值。若未指定默认值，则对于值类型默认值为`0`，对于引用类型默认值为`null`。

```
public class Person
{
    public string name;
    private int age = 18;
}
```

静态字段与常量

类可以含有静态字段，静态字段用`static`修饰。静态字段需要用“类名.字段名”的方式访问。

类内也可以定义常量，用`const`修饰，常量不允许被修改。与静态字段一样，常量也必须由“类名.字段名”的方式访问。

```
class Person
{
    public string name;
    private int age = 18;
    public static int population;
    public const string school = "THU";
}
```

```
Console.WriteLine(Person.population);
Console.WriteLine(Person.school);
```

属性 Property

属性是C# 的一大特色，可以设置不同访问器来简化代码书写。

属性命名规范

属性一般使用大驼峰命名法。

属性的访问器

下面的例子定义了一个Age属性，并实现了`get`访问器和`set`访问器。

```
class Person
{
```

```
    private int age;
    public int Age
    {
        get
        {
            Console.WriteLine("Get Age!");
            return age;
        }
        set
        {
            Console.WriteLine("Set Age!");
            age = value >= 0 ? value : 0;
        }
    }
}
```

上述代码中，字段`age`是属性`Age`背后真正的数据，属性可用来控制访问字段的方式。

访问器包括

- `get`访问器：当外部需要读取属性的值时，会调用`get`访问器。
- `set`访问器：当外部要设置属性的值时，会调用`set`访问器，`value`是要设置的值。
- `init`访问器（C# 9.0）：只允许在对象构造期间设置属性的值。

只有`get`访问器的属性是只读的，例如

```
class Person
{
    private int age;
    public int Age { get => age; }
}
```

自动实现属性

有时，定义一个字段+一个属性太过繁琐，我们不希望再额外定义一个字段，并且也没有复杂的处理逻辑，这是我们就可以用自动实现属性：

```
class Person
{
    public double Name { get; set; }
}
```

这是编译器会为我们隐式生成一个字段，绑定到这个属性上。

方法 Method

类内可以定义方法。访问修饰符与字段相同。方法的定义一般来说由访问权限、方法名、返回值、参数列表几部分组成。其中，“返回值、方法名、参数列表”一般称为“方法的签名”。例如：

```
class Person
{
    public string name;
    public void PrintName()
    {
        Console.WriteLine($"My name is {name}");
    }
}
```

方法命名规范

方法一般采用大驼峰命名法。

静态方法

一般的方法由对象调用，当然也可以声明类的静态方法，用`static`关键字声明，调用时需要使用"类名.方法名"调用。

参数缺省与指定

可以为方法定义缺省值

```
class Person
{
    public static void Print(string name, int age = 18)
    {
        Console.WriteLine($"My name is {name}. My age is {age}.");
    }
}
```

在调用该方法时，可以指定参数进行传参。使用 " : " 进行指定。

```
Person.Print(name: "TeamStyle", age: 24);
```

构造方法

每个类可以定义构造方法。构造方法不具有返回值，且方法名与类名相同，是在一个对象被构造的时候调用的方法，由`new`表达式传递参数：

```
class Person
{
    private int age;
    public Person(int age)
    {
        this.age = age;
    }
}
Person ps = new Person(18) ;
```

简化函数体

如果函数体非常简短，可以使用`=>`运算符：

```
class Adder
{
    static public int Add(int x, int y) => x + y;
}
```

重载

一个类里可以定义多个同名方法，但是它们的参数列表必须不同。调用时根据传递的实参决定调用哪个重载方法。

部分运算符也可以进行重载，例如：+、-、*、/ 等，但只能重载为静态方法。

```
namespace Math
{
    public class Vector2
    {
        public double X { get; private set; }
        public double Y { get; private set; }
        public Vector2(double x, double y)
        {
            this.X = x;
            this.Y = y;
        }
        public static double operator*(Vector2 v1, Vector2 v2)
        {
            return v1.X * v2.X + v1.Y * v2.Y;
        }
    }
}
```

接口 Interface

C# 提供接口类型，属于引用类型。接口与类不同，接口中只允许含有方法和属性，不允许含有字段。接口也不允许被实例化。

引入接口，是为了提供一套标准的属性与方法，用于继承。

命名接口时，通常以大写字母“I”开头，以表示这是一个接口，并采用大驼峰命名法。

```
public interface ICallable
{
    void Call()
    {
        Console.WriteLine("Call");
    }
}
```

需要注意的是，接口内方法与属性的默认访问权限是 `public`，并且，类在实现接口内的方法时，需要与接口内定义的权限相同。

接口还可以为其内的方法定义一个默认的实现（C# 8.0），派生类可以选择不显式实现接口的方法，而采用其默认实现。

面向对象初探

封装

用类封装，略。

继承

C# 继承的语法如下：

```
public class Base { }
public class Derived : Base { }
```

`Derived` 类继承自基类 `Base`。如果一个类没有显式继承另一个类，那么它默认继承自 `object` 类。

C# 不支持类的多继承，即一个类有且仅有一个基类（object类除外）。但一个类可以继承多个接口。

可以使用`base`关键字代表它的基类，同样构造方法也需要通过 `base` 关键字来为它的基类提供构造方法的参数。

```
class Animal
{
    private string name;
    public Animal(string name)
    {
        this.name = name;
    }
}
class Dog : Animal
{
    public Dog(string name) : base(name) { }
}
```

密封类

如果你希望一些类不允许被继承，可以将其设为密封类。C# 提供了`sealed`关键字声明密封类。密封类不能再派生出其他任何类。

```
public sealed class Foo { }
```

`is` 运算符

.NET支持在运行时进行类型检查，可以使用`is`或`is not`检查对象的类型，也可检查是否为`null`。

```
class Person { ... }
class Student : Person { ... }
class EEStudent: Student { ... }
...
var stu = new Student();
Console.WriteLine(stu is object); // True
Console.WriteLine(stu is Person); // True
Console.WriteLine(stu is Student); // True
Console.WriteLine(stu is EEStudent); // False
Console.WriteLine(stu is null);  // False
```

注意派生类与基类的兼容性。

多态

虚方法与抽象方法

虚方法用`virtual`关键字声明，抽象方法用`abstract`关键字声明。虚方法与抽象方法的区别是，虚方法必须为其定义函数体，而抽象方法不能定义函数体。

派生类中重写虚方法或抽象方法时，需要使用`override`关键字。否则只会"隐藏"基类方法，而不会重写，因此不能实现多态。

```
class Person
{
    public virtual void Print()
    {
        Console.WriteLine("Person!");
    }
}
class Student : Person
{
    public override void Print()
    {
        Console.WriteLine("Student!");
    }
}
```

虚属性与抽象属性

属性也可以声明为虚的或抽象的，相当于派生类需要重写属性的访问器，如`get`、`set`等。用法与虚方法和抽象方法完全相同。

抽象类

含有抽象方法或抽象属性的类必须是抽象类。抽象类用`abstract`关键字声明。

不能创建抽象类的实例。

接口多态

接口可以被继承，且允许多继承，但不能被实例化。继承接口的类必须实现所有方法与属性（除非接口中已提供默认值）。

接口也可用于实现多态。

```
public static void Main(string[] args)
{
    IPerson stu = new Student();
    Console.WriteLine(stu.Job);
    stu = new Teacher();
    Console.WriteLine(stu.Job);
}
interface IPerson
{
    public string Job { get; }
}
class Student : IPerson
{
    public string Job => "Student";
}
```

```
class Teacher : IPerson
{
    public string Job => "Teacher";
}
```

委托

“委托”是一种引用类型，作用与函数指针类似。需要用 `delegate` 关键字定义一个委托类型。委托类型的定义格式与方法类似，只是在返回值类型前加上 `delegate` 关键字：

```
delegate int Operate(int x, int y);
```

该段代码定义了一个委托类型，名字叫 `Operate`。该委托可以接收参数为 `(int, int)`，返回类型为 `int` 的方法。

与其他引用类型一样，我们需要用 `new` 关键字创建一个委托，并将一个方法赋给这个委托。

```
public static int Add(int x, int y)
{
    var res = x + y;
    Console.WriteLine($"{x} + {y} = {res}");
    return res;
}
Operate Op = new Operate(Add);
```

然后可以像调用方法一样调用这个委托。

```
Op(2, 1);
```

或者可以使用 `Invoke` 方法调用。

```
Op?.Invoke(2, 1);
```

单播与多播

一个委托不仅可以绑定一个方法，还可以绑定多个方法。这种委托我们称为“多播委托”。多播委托这个词的来源在于，最初微软设计的委托分为“单播委托”和“多播委托”，但是后来发现这种设计并没有什么用处，因此将两种委托合并成一种。但不幸的是当时已经设计出两种委托类了，因此多播委托类就一直传了下来。现在，所有的委托都是多播委托，不必再区分单播与多播委托。

可以用 `+` 或 `+=` 来将新的方法附加到已有的委托上，也可以用 `-` 或 `-=` 来将方法从委托中删除。使用 `+` 或 `+=` 将新的方法绑定到委托上，这称为“订阅”了该委托。

```
public delegate int Operate(int x, int y);
public static int Add(int x, int y)
{
    var res = x + y;
    Console.WriteLine($"{x} + {y} = {res}");
    return res;
}
public static int Sub(int x, int y)
{
    var res = x - y;
    Console.WriteLine($"{x} - {y} = {res}");
```

```
    return res;
}
public static void Main(string[] args)
{
    Operate Op = new Operate(Add);
    Op += Sub;
    Op?.Invoke(2, 1);
    Op = Op - Add;
    Op?.Invoke(2, 1);
}
```

当委托不绑定任何一个方法时，调用委托是非法的，因此一般都使用`?.Invoke(...)`的方法调用委托，这种写法会先判断委托是否绑定了方法，如果是再调用`Invoke`方法。

事件

为什么要有事件

一般的多播委托存在一个问题：外部对象可以任意修改已发布的委托（如果该委托是一个公有成员），这就导致外部对象的订阅不再可靠。

例如，外部对象在订阅该委托时，直接将该委托置为`null`，这就直接导致其他订阅者的订阅失效，订阅因此不再可靠。

```
interface IOperator
{
    public int Operate(int x, int y);
}
class Adder: IOperator
{
    public int Operate(int x, int y)
    {
        var res = x + y;
        Console.WriteLine($"{x} + {y} = {res}");
        return res;
    }
}

class Subber: IOperator
{
    public int Operate(int x, int y)
    {
        var res = x - y;
        Console.WriteLine($"{x} - {y} = {res}");
        return res;
    }
}

class Program
{
    public delegate int Operate(int x, int y);
    public static void Main(string[] args)
    {
        var add = new Adder();
        Operate Op = new Operate(add.Operate);

        var sub = new Subber();
```

```
        Op += sub.Operate;      // 订阅

        // ......

        // 外部对象订阅时，导致所有订阅均失效
        // Op = null;

        // ...

        var res = Op.Invoke(2, 1);   // Exception !
    }
}
```

究其原因是多播委托除了能被订阅外，还能够被直接赋值。

因此，C# 提供 “事件” 机制，以解决这一问题。

事件简介

事件使用`event`关键字定义，其本质是一个特殊的多播委托。

事件只支持“订阅”与“取消订阅”两种操作，不允许直接对事件赋值，这保证了用户订阅的可靠性。

外部对象只能使用`+=`、`-=`运算符对事件进行"订阅"、"取消订阅"。即事件只能放在`+=`、`-=`运算符的左侧。

```
public event EventHandler onWarning;
// ...
onWarning += Function1; // Correct
onWarning -= Function1; // Correct
onWarning = null;       // Error !
onWarning = onWarning + Function2; // Error !
onWarning?.Invoke();
```

C# 杂项

Lambda表达式

lambda 表达式可以看成是一个匿名方法，基本语法为

```
(参数列表) => { 函数体 }
```

- 当参数只有一个时，参数列表的括号可以不写。
- 当函数体只有一句时，或只有返回值时，可以简写，如：

  ```
  () => 1;
  ```

lambda 表达式可以直接当作方法赋值给委托，例如

```
var getAddOne = new Func<int, int>(x => x + 1);
```

从 C# 10（.NET 6）开始，Lambda 表达式可能具有自然类型。 编译器不会强制你为 Lambda 表达式声明委托类型（例如`Func<...>`或`Action<...>`），而是根据 Lambda 表达式推断委托类型。例如

```
var f = () => 1;
```

异常处理

C# 异常处理由一个 `try` 语句块加上至少一个 `catch` 或 `finally` 语句块组成。

```
try
{
    /*Some code*/
}
// catch { }
// finally { }
```

`catch` 块会捕获 `try` 块中抛出的异常。单独的一个 `catch` 可以捕获全部异常，单独的一个 `throw` 可以将捕获的异常再次抛出。

```
try {}
catch       // 捕获全部异常
{
    throw;  // 将捕获到的异常再次抛出
}
```

`finally` 块表示：无论 `try` 块中是否发生异常，执行完全部的 `try` 或 `catch` 后，都将进入 `finally` 块执行，然后再执行其他工作。因此 `finally` 块常用于进行一些恢复或清理的工作。

```
var rwlock = new ReaderWriterLockSlim();
rwlock.EnterWriteLock();        // 锁住
try
{
    /*Some code*/
}
finally
{
    rwlock.ExitWriteLock();         // 解锁
}
```

partial类

C# 允许将类分开定义，可以使用 `partial` 关键字。

```
// in file 1:
public partial class A
{
    //...
}

// in file 2
public partial class A
{
    //...
}
```

MSTest简介

MSTest是.NET提供的一套测试框架，可用于程序的单元测试。

使用Visual Studio可以帮我们方便地使用MSTest框架。当然也可以用.NET CLI。

在MSTest中，可以使用`[TestClass]`定义一个单元测试类，使用`[TestMethod]`定义一个单元测试方法。MSTest会自动运行`[TestMethod]`进行测试。可以使用`Assert`类对测试结果进行评判。一个简单的测试程序如下

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Test
{
    [TestClass]
    public class UnitTest
    {
        [TestMethod]
        public void TestMethod1
        {
            int a = 1, b = 2;
            int res = a + b;
            int testRes = Adder.Add(a, b);  // 假设我们要测试一个加法器，是否完成了加法功能
            Assert.IsTrue(res == testRes);
        }
    }
}
```

在.NET CLI中，使用`dotnet test`命令可以对指定测试项目进行测试。

其他

由于时间与篇幅限制，我们没有介绍C#的其他众多语法与功能，例如

- 泛型
- 托管与垃圾回收
- .NET 数据结构
- 指针类型
- 索引器
- 迭代器
- 文件IO
- 语言集成查询（LINQ）
- ……

略去上述内容不会对我们后续的学习产生太大影响。有兴趣的同学可以查阅微软官方文档，深入学习C#。

作业说明

作业文件

GitHub地址：[TCL606/BasicCSharp: 2022 THU EESAST 软件部暑期培训 CSharp作业 (github.com)]

或者在云盘“作业”文件夹中下载，暑培云盘链接：https://cloud.tsinghua.edu.cn/d/0d8895f41a4a4dcaa0a4/

题目及要求

我们希望编写一个程序，用于统计某个变量被修改与被读取的次数。请在`BasicCSharp/hw1/hw1/Counting.cs`中完成对变量`Variable`被修改与被读取的次数统计。只允许修改`Counting`类中的代码，且满足以下要求

- `Variable` 不能被外部程序赋值为负数。若被赋为负数，则将它代表的值置为0。
- `ReadTimes` 为 `Variable` 被外部程序读取的次数。
- `WriteTimes` 为 `Variable` 被外部程序修改的次数。

使用工具

VS 2022

作业的 C# 程序运行在.NET 6平台上，建议使用Visual Studio 2022进行编写。

使用 Visual Studio 2022直接打开`BasicCSharp/hw1/hw1.sln`进行编写。

VScode

也可以使用 VScode，使用 VScode 请保证你的计算机上有.NET 6 环境。

使用 VScode 直接修改`BasicCSharp/hw1/hw1/Counting.cs`即可。若要检验结果是否正确，可以使用命令启动`Counting.cs`程序

```
dotnet run --project BasicCSharp/hw1/hw1
```

或直接用我们写好的测试程序进行测试

```
dotnet test BasicCSharp/hw1/hw1.sln
```

提交方式

提供两种提交方式：

1. GitHub 提交
    - fork 仓库： TCL606/BasicCSharp (github.com)到个人仓库，按要求修改好后，从个人仓库提pr到原本的仓库TCL606/BasicCSharp: 2022 THU EESAST 软件部暑期培训 CSharp作业 (github.com)。pr信息填写为：`CSharp_姓名_班级` （如：`CSharp_小明_无19`）。
2. 邮箱提交
    - 只允许修改`BasicCSharp/hw1/hw1/Counting.cs`代码，修改好后，将`BasicCSharp`整个文件夹打包成常见压缩格式（如`.rar`、`.zip`等），并命名为：`CSharp_姓名_班级` （如：`CSharp_小明_无19`）发送到邮箱 **tcl606_thu@163.com**。

截止日期

2022.7.20

参考文献

1. C# 程序设计基础 | EESAST Docs (eesast.com)，2022年7月2日
2. https://docs.microsoft.com/en-us/
3. 《复变函数与数理方程》教案，吴昊（男）