写在前面的话
    先修知识
.NET 概述
    什么是 .NET
C# 概览
    创建一个 C# 程序
    `Main` 方法
    程序的空间资源
    C# 类型系统
        值类型
            内置数值类型
            自定义结构类型
            枚举类型
            元组类型
            值类型变量的定义
        引用类型
                装箱和拆箱
            `string` 初探
                字符串文本
        关于类型的几点说明
            自动推导类型
            关于赋值
            关于垃圾回收
C# 入门
    输入输出
        输出
        输入
    C# 控制结构：
    有趣的？
    数组
        一维数组
        多维数组
        交错数组
C# 初步
    命名规范
    类
        定义一个类
            类的命名规范
            字段
                字段的命名规范
                字段的默认值
                静态字段与常量
                只读字段
            方法
                方法的定义
                方法的命名规范
                静态方法
                方法的参数传递方式
                参数的缺省值
                构造方法
                简化的函数体
                运算符重载
            属性
                属性的定义

            属性的命名规范
            自动实现属性
        `partial` 类
        继承
            `is` 运算符
    自定义结构类型
    多态：虚方法、抽象方法与抽象类
        虚属性与抽象属性
    多态：接口
        接口的命名规范
    泛型
        泛型初探
        泛型的约束
    委托
        定义、实例化并调用一个委托
        内置委托
        多播委托
    事件
    lambda 表达式
    异常处理
    .NET 数据结构
        `System.Collections`
        `System.Collections.Generic`
        `System.Collections.Concurrent`
C# 精通
作业
    题目及要求
    提交方式
    截止日期
参考文献

写在前面的话

先修知识

阅读本文档要求有 C 语言的基础，并初步了解面向对象的基本知识，掌握至少一门支持面向对象的语言（例如 C++、Java 等）。

.NET 概述

什么是 .NET

.NET 是微软公司发布的应用程序框架，包括一系列类库、运行时等内容。

在生成一个 .NET 程序时，程序员编写的代码先被翻译成“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 编译为机器码执行，称作 **JIT 编译**（just-in-time compilation）。

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 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。

C# 概览

创建一个 C# 程序

基于 .NET Core 的 C# 程序可以通过命令行创建，在安装了 .NET Core SDK 后，可以在命令行中输入：

```
dotnet new console --output hello
```

来创建一个 .NET Core 控制台应用程序

输入下面的命令可以运行该控制台程序：

```
dotnet run --project hello
```

程序输出：

```
Hello World!
```

如果在 Windows 上开发，我们可以使用 Visual Studio 来完成上述过程，而无须使用命令行。

`Main` 方法

一个基本的C# HelloWorld程序如下

```
using System;

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

`using System;` 即使用 `System` 命名空间，`System` 是很多 .NET 类库所在的命名空间。其中，`Console` 是 `System` 命名空间的一个类，因此该语句应为 `System.Console.WriteLine("Hello World!");`。由于我们用了 `using System;` ，因此可省去 `System.`。

`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，`Main` 必须是静态方法（也就说它必须是 `static` 修饰的）

`Main` 方法的形式通常有以下几种：

```
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。

> 在 C# 9.0（.NET 5.0）中，应用程序也可以没有 `Main` 方法，但是需要有且仅有一个文件具有“顶级语句（Top-level statements）”。

> 有兴趣的同学可以查阅微软官方文档：https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statements

程序的空间资源

C#程序在CLR上运行时，内存从逻辑上划分两大块：栈、堆

- 栈，在程序运行时，每个线程(Thread)都会维护一个自己的专属线程堆栈。作为栈，数据只能从栈的顶端插入或删除。存放方法的参数、局部变量、返回地址等值，当一个方法执行完毕后立刻自动清除。
- 堆：用来保存进程运行时请求操作系统分配给自己的内存空间，他们被调用完毕不会立即被清理掉。内存泄露是发生在堆中的。

C# 类型系统

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# 的类型分为三种：值类型、引用类型以及指针类型。

> C# 保留了指针类型，但只能定义非托管类型的指针。指针在日常的使用中并不多见，在此不做过多介绍。此外，C# 9.0 开始支持函数指针。

值类型

值类型作为字段时，跟随其所属对象存储在堆上，作为局部变量时，存储在栈上。值类型分为两种：结构类型和枚举类型。

内置数值类型

内置的数值类型均属于结构类型，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

C# 类型名称	范围	对应的 .NET 类型	备注
`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

与C++不同（取决于平台与编译器），C#的数值类型的字节大小是确定的。

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

结构类型也可以自定义：

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

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.
}
```

元组类型

元组功能提供了简洁的语法来将多个数据元素分组成一个轻型数据结构。

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
```

元组最常见的用例之一是作为方法返回类型。

更多内容：值类型元组、弃元与析构元组

值类型变量的定义

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);
```

引用类型

引用类型与值类型不同。引用类型分两部分，一部分是存储对象真正数据的部分，是开辟在“托管堆”上的，另一个是指向该数据块的“引用”，是在内存栈或其他任何位置的。

创建引用类型的对象时，一般需要通过 `new` 关键字来在内存堆上开辟数据空间，创建对象的过程通常称为“实例化”。

例如字符串类型（`string`）就是一个引用类型：

```
string s1 = new string("Hello, world");      // s1 是一个引用，指向一个 string 对象，该对象为 "Hello, world".
string s2 = "Hello, world";                  // string s2 = new string("Hello, world"); 的简略写法
string s3;                                   // 仅仅定义了一个 string 引用，并没有创建和指向任何 string 对象!
int y = new int();                           // 与 int y; 完全等价
```

没有指向任何对象的引用的值是默认值 `null`，即上述代码等价于 `string s3 = null;`。

引用类型有很多，例如内置的引用类型 `object`、`string`、`dynamic`，数组类型、类类型（`class`）、委托类型（`delegate`）、接口类型（`interface`）、记录类型（`record`，C# 9.0）

装箱和拆箱

`object` 是一个 C# 关键字，表示 .NET 类型 `System.Object` 类，即是一个类类型，它是所有类型（除指针类型）的公共基类。而它是个引用类型，这也意味着 `object` 的引用可以指向所有类型的托管对象（关于托管的概念在后文中会有提到）。

既然值类型没有引用，那么一个 `object` 引用是如何指向值类型的呢？这个机制就叫“装箱（Boxing）”和“取消装箱（Unboxing）”（也叫“拆箱”）。

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”。

当我们想把装箱产生的 `object` 类型转换回该值类型的时候，存储在托管堆上的数据就会被拿出来，这个过程叫做“拆箱”。

通过装箱和拆箱可以让每一个值类型都被转换为 `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。

```
int y;
object o = y;   //Boxing
int x = (int)o; //Unboxing
```

所有类都可以使用 `object` 类的方法。`object` 类有一些比较有用的方法：

- `ToString`：`ToString` 是一个虚方法，这意味着任何类和结构都可以重写 `ToString` 方法。它的作用是把一个对象转换为字符串，例如 `5.ToString()` 会返回字符串 `"5"`。
- `GetHashCode`：返回一个 `int` 值。自定义的类可以重写 `GetHashCode` 方法，返回对象的哈希值，就可以将对象用于 .NET 库自带的哈希表当中。
- `GetType`：获取对象的类型信息，返回一个 `System.Type` 对象，储存这个对象所属类型的信息。

`string` 初探

`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

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# 的逐字文本。逐字文本以 `@` 开头，并用双引号引起。逐字文本中 `\` 不再被解释为转义字符，而且逐字文本中可以含有换行。唯一的例外是逐字文本中的双引号 `"` 需要使用两个连续的双引号 `""` 来表示。

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（C# 11 开始，表示内插的花括号内可以换行，这有利于在内插中使用模式匹配等），而内插字符串中的花括号 `{` 和 `}` 需要分别使用两个连续的花括号 `{{` 和 `}}` 来表示。

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

C# 也支持逐字内插字符串，以 `$@` 开头，即可以将逐字文本和内插字符串的用法结合起来。

> C# 11（.NET 7）开始支持原始字符串文本，以 `"""` 开始并以 `"""` 结束，允许多行字符串，若为多行字符串则以单独的一行 `"""` 结束，且字符串的缩进以 `"""` 的起始位置为基准。原始字符串文本不进行任何转义操作，但允许字符串内插（开头的 `$` 数量代表需要内插的花括号数）：

```
var x = 1;
var y = 2;
var code1 = """int i = 0;""";
```

```
var code2 = $"""int x = {x};""";
var code3 = $$"""
    #include <stdio.h>
    int main(void) {
        const char *s = "{y} = {{y}}";  // {y} = 2
        return 0;
    }
    """;
Console.WriteLine($"code1:\n{code1}\n");
Console.WriteLine($"code2:\n{code2}\n");
Console.WriteLine($"code3:\n{code3}\n");
```

关于类型的几点说明

自动推导类型

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 类型的引用
var o = new object();   //  o 是 object 类型的引用
```

关于赋值

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。

关于垃圾回收

在托管堆上使用 `new` 实例化一个引用类型的对象后，不需要将其 `delete` 掉，.NET 的垃圾回收（GC）机制会自动完成这件事情。

C# 入门

输入输出

输出

C# 的控制台输出需要使用 `System` 命名空间的 `Console` 类的 `Write`、`WriteLine` 方法。区别是 `WriteLine` 会在末尾加一个换行而 `Write` 不会。

此外，可以进行格式输出或字符串内插

```
System.Console.Write("Hello, ");
System.Console.WriteLine("world!");
int x = 4, y = 5, z = 6;
Console.WriteLine("z = {2}, x = {0}, y = {1}", x, y, z); // {n} 代表该处应换成第 n 个参数的值
Console.WriteLine($"z = {z}, x = {x}, y = {y}"); //字符串内插更为常见
```

输入

C# 提供两种控制台输入：`System.Console.Read` 和 `System.Console.ReadLine`。其中 `System.Console.Read` 读取一个字符返回（返回值为 `int`）。而 `System.Console.ReadLine` 可以读入一行字符串。可以用手动转换为其他类型：

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

C# 控制结构：

- 条件分支：`if`、`switch`
- 循环：`while`、`do...while`、`for`、`foreach`
  - `foreach` 语法格式如下：

    ```
    foreach(数据类型 迭代变量 in 数组或者集合)
    {
        代码
    }
    ```

  - 你不可以修改迭代变量，不能使数组或者集合增减元素（请使用 `for`）
- 分支：`goto`

特别说明：`switch` 语句中，如果 `case` 标签后有语句，则必须要有 `break` 或 `return` 等跳转语句，除非 `case` 后没有任何语句。

有趣的？

- 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
  ```

- null 检查运算符：`?` 常用于检查一个引用是否为 `null`，若不为 `null` 则对其进行访问：

  ```
  static public void Func(SomeType foo)
  {
      foo?.DoSomething(); // 如果 foo 不为 null，则调用 DoSomething 方法
  }
  ```

- null 合并运算符：`??`。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

  ```
  static public void PrintString(string? s) // string后面的?表示允许为null
  {
      Console.WriteLine(s ?? "Null string");
  }
  ```

  此外还有 null 合并赋值运算符 `??=`（C# 8.0），若左边的值是 `null`，则给它赋右边的值：

  ```
  static public void ConvertObject? o)
  {
      o ??= new object();
  }
  ```

该方法实现功能：若 `o` 为 `null`，则创建一个新的 `object` 对象赋给 `o`。

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最基本形式如下：

```
// type[] arr = new type[array_size];
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
var arr3 = new int[] { 1, 2, 3, 4, 5 };     // 数组的长度可以由编译器自动推导
int[] arr4 = { 1, 2, 3, 4, 5 };             // 简略写法
```

其中，各个数组的类型均为 `int[]`。可以通过 `[]` 访问其元素，`Length` 获取元素个数：

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

多维数组

C# 可以定义多维的数组，以二维数组为例，二维数组的类型为 `int[,]`：

```
int[,] arr1 = new int[2, 3] { { 1, 2, 3 }, { 5, 4, 9 } };
int[,] arr2 = { { 1, 2, 3 }, { 5, 4, 9 } };
```

通过 `Length` 获取数组的元素个数，通过 `GetLength(n)` 获取数组第 `n` 维的长度：

```
var arr = new int[,] { { 1, 2, 3 }, { 5, 4, 9 } };
Console.WriteLine($"{arr.Length} {arr.GetLength(0)} {arr.GetLength(1)}");
```

输出：

```
6 2 3
```

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# 初步

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

定义一个类

类类型属于“引用类型”，由 `class` 关键字定义。

类的命名规范

类名习惯上采用大驼峰命名法

字段

一个类可以含有它自己的“字段（field）”，一个“字段”即它内部含有的变量：

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

可以看到，这个 `Person` 类含有两个字段，一个是 `int` 类型的字段，一个是`string` 类型的引用作为字段。

前面的 `private` 是访问修饰符，它规定了这个字段的访问级别。

`private` 可以换成：

- `private`：只能够被本类所访问
- `public`：可以被随意访问
- `protected`：可以被本类及其派生类访问
- `internal`：可以在本程序集内随意访问
- `protected internal`：既可以被本类及其派生类访问，又可以在本程序集内随意访问
- `private protected`：可以被本类及其在**本程序集**内的派生类访问

访问一个对象的字段需要用“对象的引用名.字段名”的方式访问。

字段的命名规范

字段习惯上采用小驼峰命名法。

字段的默认值

C# 的字段可以定义默认值：

```
public class Foo
{
    private int bar = 233;
}
```

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`。

静态字段与常量

类可以含有静态字段，静态字段用 `static` 修饰。

类内也可以定义常量，用const修饰，常量不允许被修改。

静态字段与常量需要用“类名.字段名”的方式访问。

```
public class Person
{
    public static int population;
    public const string school="xfggtql";
}
```

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

只读字段

在字段前加上 `readonly` 修饰符代表该字段只能在构造方法里被赋值。一旦构造方法里被赋值后，便不能再修改它的值：

方法

方法的定义

类内可以含有方法（method）。

```
class Person
{
    private int age = 0;
    private string name = "Tom";

    public void Print(int n)
    {
        for (int i = 0; i < n; ++i)
        {
            Console.WriteLine($"age: {age}, name: {name}");
        }
    }
```

```
}

var ps = new Person();
ps.Print(2);
```

方法的命名规范

方法习惯上采用大驼峰命名法。

静态方法

类可以含有静态方法，静态方法是在方法前面加上 `static` 关键字：

```
class MathTool
{
    static public int Add(int x, int y)
    {
        return x + y;
    }
}
```

调用时需要用“类名.方法名”的方式调用：

```
Console.WriteLine(MathTool.Add(3, 5)); // 输出 8
```

静态方法只能访问静态成员。静态方法在执行时，并不一定存在对象。

方法的参数传递方式

方法的参数默认采用值传递，即将实参复制一份给形参。对于值类型来说，复制的是值类型的所有数据，对于引用类型来说，复制的是一个引用。

```
class Person
{
    public int age;
}

class Utility
{
    static public void Swap(Person x, Person y)
    {
        Person tmp = x;
        x = y;
        y = tmp;
    }

    static public void SwapAge(Person x, Person y)
    {
        int tmp = x.age;
        x.age = y.age;
        y.age = tmp;
    }
}
```

调用：

```
Person p = new Person(), q = new Person();
p.age = 555; q.age = 666;
Utility.Swap(p, q);
Console.WriteLine($"{p.age} {q.age}"); // 555 666
Utility.SwapAge(p, q);
Console.WriteLine($"{p.age} {q.age}"); // 666 555
```

此外，参数可以改成按引用方式传递，需要加上关键字 `ref`。如果加上关键字 `ref`，则值类型的形参和实参指代的是同一个值类型对象，而引用类型的实参和形参指代的是同一个引用，例如上述代码改成：

```
static public void Swap(ref Person x, ref Person y)
{
    Person tmp = x;
    x = y;
    y = tmp;
}
```

调用：

```
Person p = new Person(), q = new Person();
p.age = 555; q.age = 666;
Utility.Swap(ref p, ref q);  // 调用时也必须加 ref 关键字!
Console.WriteLine($"{p.age} {q.age}"); // 666 555
```

参数的缺省值

C# 具有参数缺省值：

```
class MathTool
{
    static public int Div(int x = 1, int y = 1)
    {
        return x / y;
    }
}
```

调用时，默认会把末尾未赋实参的参数赋以缺省值。但是也可以自行指定：

```
MathTool.Div(5);        // x = 5, y = 1
Math.Div(y: 9);         // x = 1, y = 9
Math.Div(y: 5, x: 4);   // x = 4, y = 5
```

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_)
    {
        age = age_;
    }
}
```

```
Person ps = new Person(4);
```

简化的函数体

如果函数体非常简短，可以使用 `=>` 运算符：

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

运算符重载

一些运算符可以进行重载，例如：`+`、`-`、`*`、`&`、`|`、`true`、`false`，等等。

运算符重载只能将运算符重载为**静态方法**，例如计算向量内积：

```
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;
        }
    }
}
```

属性

属性（**property**）是 C# 的一个特色，它简化了许多代码的书写。属性本质上也是通过方法实现的。

属性的定义

一个属性包含至多两个访问器，一个是 `get` 访问器，一个可以为 `set` 访问器或 `init` 访问器（C# 9.0）。

- `get` 访问器：用于外部读取字段的值。

- `set` 访问器：用于外部设置字段的值。
- `init` 访问器（C# 9.0）：只允许在对象构造期间设置属性的值。

如下，`Age` 是一个属性，它用于设置和访问 `age` 字段的值。

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

其中，`set` 被定义成是 `private` 的，而 `get` 没有显式写出，则默认与属性的访问权限相同，即 `public`。

属性的命名规范

习惯上，字段采用小驼峰命名法并为 `private`，而属性与它设置的字段同名并采用大驼峰命名法供外部访问。

自动实现属性

如果属性没有复杂的处理逻辑，我们可采用自动实现属性：

```
class Complex
{
    public double Real { get; set; }
    public double Imag { get; init; }
}
```

这是编译器会为我们隐式生成两个字段，分别绑定到两个属性上。

`partial` 类

.NET 支持将类在一个程序集内分成很多块来定义，甚至放在多个文件里，这时是需要将每个部分的定义都加上 `partial` 关键字即可。

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。

在类里可以通过 `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) {}
}
```

`is` 运算符

.NET 支持在运行期进行类型检查。使用 `is` 或 `is not` 运算符可以检查对象的类型，还可以检查是否为 `null`，例如：

```
// 变量声明：
// Animal ani = new Animal("");
// Dog dog = new Dog("");
// Animal ani2 = dog;  // 用 ani2 引用指向 dog 指向的对象
// object o = 1;   // 将整数 1 装箱

Console.WriteLine(ani is Animal);   // True，ani 是 Animal 类的对象
Console.WriteLine(ani is Dog);      // False，ani 不是 Dog 类的对象
Console.WriteLine(dog is Animal);   // True，Animal 是 Dog 类的基类
Console.WriteLine(ani2 is Dog);     // True，ani2 指向的确实是 Dog 类的对象
Console.WriteLine(o is int);        // True，拆箱
```

自定义结构类型

与类类型类似，有字段、方法、属性

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

多态：虚方法、抽象方法与抽象类

“抽象类”是指不能使用 `new` 实例化出对象的类，抽象类在定义时在 `class` 前加上 `abstract` 关键字。

虚方法必须为其定义函数体，而抽象方法不能定义函数体，且具有抽象方法的类必须是抽象类。

派生类中重写基类的虚方法或抽象方法时，需要在前面加上 `override` 关键字，否则只会“隐藏”基类方法。

```
public abstract class Animal
{
    public abstract void Call();
}

public sealed class Dog : Animal
{
    public override void Call()
    {
        Console.WriteLine("Wang wang!");
    }
}
```

```
Animal ani = new Dog();
ani.Call();                      // 输出 Wang wang
// Animal ani2 = new Animal();  // 编译错误，Animal 是抽象类
```

虚属性与抽象属性

由于属性的 `get` 与 `set` 访问器底层用方法实现，因此也可以将属性声明为 `abstract` 与 `virtual`。用法与虚方法和抽象方法完全相同。

多态：接口

C# 支持接口（interface），属于引用类型。

接口中只允许含有方法和属性，而不允许含有字段。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。

一个结构（struct）或一个类（class）可以实现多个接口，即接口允许多继承，实现一个接口的结构或类必须实现其所有方法与属性，例如：

```
public interface IObject
{
    public int X { get; private set; }
    public int Y { get; private set; }
    public void GetPosition();
}

public interface IMoveable
{
    public void Move(int x,int y);
}

public class Cat : IMoveable , IObject
```

```
{
    public int X { get; private set; }
    public int Y { get; private set; }

    public void Move(int x,int y)
    {
        X = x;
        Y = y;
    }
    public void GetPosition()
    {
        Console.WriteLine($"X:{X},Y:{Y}");
    }
}
```

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。

接口还可以定义属性。但是，接口不会为属性提供默认实现，也不会定义自动实现属性，只能约束该属性至少具有的访问器。例如：

```
public interface INamable
{
    string Name { get; }
}

public class Cat : ICallable, INamable
{
    void ICallable.Call()
    {
        Console.WriteLine("Meow");
    }

    private string name;
    public string Name
    {
        get => name;  // 实现接口中 Name 的 get 访问器
        private set
        {
            name = value;
        }
    }
}
```

接口的命名规范

接口一般采用大驼峰命名法，并以大写字母“I”开头。例如 .NET 库的 `IEnumable`、`ICollection`，等等

泛型

泛型初探

C# 支持泛型。可以创建泛型类、泛型方法、泛型接口、泛型委托、泛型记录（C# 9.0），等等。

泛型类和泛型方法的定义方法很简单，只需要在类或方法名后使用尖括号 `<>` 括住泛型的名称即可：

```
class Point<T>
{
    public T X { get; private set; }
    public T Y { get; private set; }
    public Point(T x, T y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

```
Point<int> pt = new Point(0, 0);
```

泛型的约束

C# 可以对泛型的性质进行约束。不满足约束性质的泛型参数会在编译期检查出来。

泛型约束的办法是使用 `where` 关键字，例如：

```
class Point<T>
    where T : struct
{
    public T X { get; private set; }
    public T Y { get; private set; }
    public Point(T x, T y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

其中，`where T : struct` 表示 `T` 必须是一个不可为 `null` 的值类型，否则会报错。此外，`struct` 还可换成（关于可为 `null` 和不可为 `null` 的类型将在之后说明）：

- `struct`：不可为 `null` 的值类型（C# 8.0 及以上）；值类型（C# 8.0 以前）
- `class`：不可为 `null` 的引用类型（C# 8.0 及以上）；引用类型（C# 8.0 以前）
- `class?`：引用类型（C# 8.0 以上）
- `new()`：具有无参构造方法
- 一个类名或接口名：`T` 必须从该类继承或实现了该接口
- ……

委托

定义、实例化并调用一个委托

“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 BinaryFunctor(int x, int y);
```

该段代码定义了一个委托类型，名字叫 `BinaryFunctor`。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。

```
int Add(int a, int b) => a + b;  // Add 方法的定义
BinaryFunctor bf = new BinaryFunctor(Add);
```

然后我们便可以像调用方法一样调用这个委托所绑定的方法：

```
Console.WriteLine(bf(3, 5)); // 输出 8。 bf(3, 5)相当于bf.Invoke(3, 5)
Console.WriteLine(bf?.Invoke(3, 5)); //这样更合理,在多播委托中常用
```

内置委托

多数情况下，我们并不需要自定义委托类型，.NET 中已经定义好了一些内置的委托类型：

- `Action` 是返回值为 `void` 类型的委托，泛型参数列表内为参数列表，例如 `Action` 为无参且返回值为 `void` 的委托、`Action<int>` 为参数是 `int` 且返回值为 `void` 的委托。`Action` 最多可以有 16 个参数类型。
- `Func` 是既有参数又有返回值的委托。泛型参数列表中最后一个为返回值类型。例如 `Func<int, double>` 为参数是 `int`、返回值是 `double` 的委托。`Func` 最多可达 16 个参数类型和 1 个返回值类型。

多播委托

一个委托不仅可以绑定一个方法，还可以绑定多个方法。

可以用 `+` 或 `+=` 来将新的方法附加到已有的委托上，称为“订阅”了该委托，例如：

```
// static public void Call1() => Console.WriteLine("Call1");
// static public void Call2() => Console.WriteLine("Call2");
// static public void Call3() => Console.WriteLine("Call3");

var caller = new Action(Call1);
caller += Call2;
caller = caller + Call3;
caller?.Invoke();
```

同样也可以用 `-` 或 `-=` 来讲方法从委托中删除，例如 `caller -= Call1;`。

一般调用顺序与订阅的顺序相同，但官方不予保证。

事件

事件一般定义方式：

```
public event EventHandler<任意类型> 事件名称;
```

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"。

而类内对事件做的操作一般是`?.Invoke()`。

例如下面的例子：

```
partial class Compiler
{
    public class WarningMessage : EventArgs
    {
        public string Message { get; }
        public WarningMessage(string message)
        {
            Message = message;
        }
    }

    public event EventHandler<WarningMessage> onWarning;

    private void Warn(string message)
    {
        onWarning?.Invoke(this, new WarningMessage(message));
    }
}

var compiler = new Compiler();
compiler.onWarning += (obj, eventArg) => Console.WriteLine(eventArg.Message);
```

第一个参数常用于传递引发事件的对象，因此通常传递`this`引用，而第二个参数则通常传递事件发生需要传递的参数。

此外，事件还可以自定义`add`与`remove`访问器，再此不做过多展开。

lambda 表达式

lambda 表达式可以看成是一个匿名方法。

lambda 表达式的基本语法是：

```
(参数列表) => { 函数体 }
```

如果函数体只有一条语句，可以省略花括号：

```
(x, y) => x + y;
```

特别地，如果只有一个参数，则括号可以省略：

```
o => o + 1;
```

lambda 表达式可以直接当作方法赋值给委托，例如：

```
var output = new Action
(
    () =>
    {
        Console.WriteLine("Hello, world!");
    }  // 或 () => Console.WriteLine("Hello, world!");
);
var getAddOne = new Func<int, int>(x => x + 1);
```

从 C# 10（.NET 6.0）开始，lambda 表达式可以自动推导出 `Action` 或 `Func` 类型，也可以手动指定参数类型和返回值类型，因此上述代码可以用以下更简洁的方式写出：

```
var output = () =>
{
    Console.WriteLine("Hello, world!");
};
var getAddOne = (int x) => x + 1;
```

以及下面的代码：

```
var f = object (int x) => x + 1; // f 是 Func<int, object> 而非 Func<int, int>
```

异常处理

C# 支持异常处理。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*/
```

举一个简单的例子，`Exception` 是 `System` 命名空间的一个类，是一切异常类的基类，抛出的异常也必须从 `Exception` 类中派生出来。`Message` 是该类的虚属性，通常储存着异常信息。

```
try
{
    /*Some code 1*/
    throw new Exception("Throw an exception!");
    /*Some code 2*/
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

`try` 块下可能有多个 `catch` 块，则抛出异常时会一次查找下面每个 `catch` 块所捕获的内容。如果找到了可以匹配的类型，则执行该 `catch` 块，如果没找到则异常向上抛出。

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
{
    Console.WriteLine("Caught");
    throw; // 将捕获到的异常再次抛出
}
```

`throw` 语句可以引发异常：

```
if (shapeAmount <= 0)
{
    throw new ArgumentOutOfRangeException(nameof(shapeAmount), "Amount of shapes
must be positive.");
}
```

C# 支持 `finally` 语句块，放在所有 `catch` 块之后。在执行所有的 `try` 或 `catch` （即便有 `return` ）后，都将进入 `finally` 块执行。

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

.NET 数据结构

.NET 提供了很多数据结构可供使用：

`System.Collections`

位于 `System.Collections` 命名空间中的集合的每个元素都是 `object` （少数集合如 `BitArray` 除外），这意味着它可以容纳任何类型：

- `ArrayList`：列表（可变长的数组，线性结构）
- `SortedList`：有序列表
- `Stack`：栈
- `Queue`：队列
- `Hashtable`：哈希表（键值对）

`System.Collections.Generic`

位于 `System.ollections.Generic` 命名空间中的集合都是泛型集合，其容纳的元素类型由泛型参数决定：

- `List<T>`：列表（可变长的数组，线性结构）
- `SortedSet<T>`：有序列表
- `SortedList<T>`：有序列表（键值对）
- `LinkedList<T>`：双向链表
- `Stack<T>`：栈
- `Queue<T>`：队列
- `PriorityQueue<T>`：优先级队列
- `HashSet`：哈希集合
- `Dictionary`：字典（键值对）
- `SortedDictionary`：有序字典（键值对，二叉搜索树）

`System.Collections.Concurrent`

位于 `System.Collections.Concurrent` 命名空间中的集合时并发安全的，将在其他文档中进行介绍。

C# 精通

由于篇幅的限制和课时的影响，很多有趣且非常有用的内容我们没有做过多展开，例如：

- 匿名类型、记录类型（C# 9.0）和记录值类型（C# 10.0）
- 模式匹配
- 索引器（Indexer）
- 特性（Attribute）
- 迭代器与可迭代对象
- 反射（Reflection）
- 语言集成查询（LINQ）
- 字符编码操作（`System.Text.Encoding`）、.NET 正则表达式语法与.NET 正则表达式库、JSON 序列化（`System.Text.Json`）、XML 文件操作与XML 序列化
- .NET 的流与文件读写（`StreamReader`、`StreamWriter`、`BinaryReader`、`BinaryWriter`）
- 使用栈空间提升程序性能：`ref struct` 类型（C# 7.2）、`Span<T>`（C# 7.2）与 `stackalloc`
- 类似 C 语言的非托管代码编写：`System.Runtime.InteropServices.Marshal`
- AOT：本地代码生成（.NET 7）
- 表达式树
- .NET HTTP 客户端框架（`System.Net.Http.HttpClient`）
- .NET Web 开发——ASP.NET 与 Blazor
- ……

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：

https://learn.microsoft.com/zh-cn/dotnet/csharp/

深入学习 C#。

我们将在下一节学习使用 C# 进行多线程程序与异步程序的编写。接下来请进一步学习“多线程与异步”单元。

作业

题目及要求

- 根据Program.cs中IProgress的要求修改Progress类中的代码
- 作业地址https://github.com/shangfengh/EESAST-hw2023-CSharp1/tree/main

提交方式

GitHub 提交

- fork 仓库：shangfengh/EESAST-hw2023-CSharp1(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，pr信息填写为：`CSharp_姓名_班级`（如：`CSharp_大佬_无29`）。

截止日期

由yxgg决定

参考文献

1. https://learn.microsoft.com/en-us/
2. 《复变函数与数理方程》教案，吴昊（男）