

C# - P16

方法

在C#中，方法是将相关的语句组织在一起，用于执行一个任务的语句块
每一个C#程序至少有一个带有Main方法的类

在C#中定义方法的基本语法结构为：

```
<Access Specifier> <Return Type> <Method Name> [<Parameter List>]  
{<Method Body>}  
}  
[  
    <Method Body>  
]
```

Access Specifier (访问修饰符)：决定变量/方法对于另一个类的可见性

Return Type (返回类型)：一个方法可以返回至多一个值

参数类型是方法返回值的数据类型，void 表示方法不返回任何值

Method Name (方法名字)：唯一的且大小写敏感的标识符

且不能与类中声明的其他标识符相同

Parameter List (参数列表)：用于传递和接收方法数据的参数

参数列表指示方法的参数类型、顺序和数量

参数是可选的，方法可能不包含参数

Method Body (方法主体)：包含完成任务所需的指令集

在C#中提供了三种传递参数的方式：

值参数：参数传递的默认方式，定义格式为(<parameter type> <parameter name>)

复制参数的 实际值给函数的形式参数：

实参和形参使用的是两个不同的内存中的值

当调用方法时，会为每个值参数创建新的存储位置

当形参的值发生改变时，不影响实参的值，从而保证实参数据的安全

引用参数：对变量的内存位置的引用，定义格式为(ret <type> <name>)

按引用传递参数时，不会为参数创建新的存储位置

引用参数表示与提供给方法的实际参数具有相同的内存位置

当形参的值发生改变时，同时改变实参的值

传递变量前必须初始化

输出参数：用于返回多个值，定义格式为(out <type> <name>)

与 ref 参数类似，但仅用于返回结果

out 参数无法将数据传入方法

out 型参数可以在传入参数前初始化（慎重起见应先初始化）

在方法中必须对 out 型参数赋值，否则编译器错误



数组

(array). 在 C# 中，数组是存储相同类型元素的固定大小的顺序集合。
数组的元素通过索引访问，所有元素由连续内存位置组成。
最低地址对应第一个元素，最高地址对应最后一个元素。

声明数组的基本语法结构为：

`<datatype>[] <arrayname>;`

`<datatype>`：指定存储在数组中的元素类型

`<arrayname>`：指定数组的秩(维度)

`<datatype> <arrayname>`：指定数组名字的标识符

在 C# 中，数组是一种引用类型，声明数组不会在内存中初始化数组。

需要使用 `new` 关键字来创建数组实例。

如 `double[] scores = new double[10];`

可以创建并初始化数组。

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

注意如果创建数组实例时声明了数组的大小

则初始化的数组大小应与声明相同。

注意如果在声明数组时没有指定大小，则会报错：array initializer of length is expected

创建数组实例时，C# 编译器会根据数组类型隐式地初始化每个数组元素。

与 C++ 和 Java 相似，通过 `arrayname[index]` 的形式访问数组元素。

交错数组：在 C# 中，交错数组是数组的数组。

如 `int[][] marks = new int[2][3];`

`marks[0] = new int[] {1};` 每一行的长度不作限制

`marks[1] = new int[] {2, 3};`

注意 C# 的交错数组依旧是一维数组。

如 `int[] marks.Length → 3, marks[1].Length → 2`

`marks[1].GetLength(1) → System.IndexOutOfRangeException`

交错数组访问反映了多维索引：(行数, 列数)。

多维数组：在 C# 中，多维数组也称为矩阵数组，也可以视为矩阵。

如 `int[,] marks = new int[2, 3, 2];`

`marks[0, 0, 0] = {{1, 2}, {3, 4}, {5, 6}}, {{7, 8}, {9, 10}, {11, 12}};`

交错数组访问反映了数组嵌套，`marks[1, 1, 1] → 3`

多维数组访问反映了多维索引，`marks[1, 2, 0] → 11`

且有 `mark.Length → 12`，即 $2 \times 3 \times 2$

`mark.GetLength(1) → 3` 即第 2 个维度的大小。

C# - P18

参数数组

在C#中，参数数组用于传递未知数量的参数给函数

Params 关键字用于使用数组作为形参时

即可以传递数组实参，也可以传递一组数组元素

定义语法为：public <return type> <method name> (params <datatype>[] <array name>)

对于如 public void UseParams (params object[] objs)

可以传入以逗号隔开的参数：UseParams (1, 'i', "Apple");

传入指定类型的参数数组：object[] objs = {2, 'j', "Boom"};

UseParams (objs);

不传入任何参数：UseParams ();

注意：带 params 关键字的参数类型必须是一维数组，不能使用多维数组

不允许与 ref / out 同时使用

带 params 关键字的参数必须是方法的最后一个参数

方法声明中至多只能有一个带 params 关键字的参数

不能仅使用 params 关键字来使用重载方法

没有 params 关键字的方法优先级高于带有 params 关键字的方法

即不允许存在仅对不带 params 关键字的数组形参加上 params 的重载方法

如 static void OverloadParams (int[] lst)

Static void OverloadParams (params int[] lst)] 不是合法的重载

Array 类

在C#中，System命名空间中定义的Array类是所有数组的基类

属性：IsFixedSize：指示数组是否带有固定大小

IsReadOnly：指示数组是否只读

Length / LongLength：32-bit / 64-bit 整数表示所有维度的数组中的元素总数

Rank：指示数组的秩（维度）

方法：Clear：设置数组某个范围的元素为 0 / false / null

Copy (Array, Array, Int32)：从数组首个元素开始复制某个范围到另一个数组的首个元素位置

CopyTo (Array, Int32)：从当前数组中复制所有元素到另一个数组的指定索引

GetLength / GetLongLength：32-bit / 64-bit 整数表示指定维度的数组中元素总数

GetLowerBound / GetUpperBound：获取数组指定维度的上界

GetType：获取当前实例的类型，从对象 (Object) 继承

GetValue (Int32) / SetValue (Object, Int32)：获取 / 设置指定索引的一维数组的值

IndexOf (Array, Object)：搜索对象在一维数组中首次出现的位置

Sort (Array) / Reverse (Array)：以 IComparable 排序 / 逆转一维数组

ToString：返回字符串表示，从对象 (Object) 继承

C# - P19

结构体 (struct)，在C#中，结构体是使得单一变量可以存储各种数据类型数据的值类型数据结构
（`struct` 关键字用于创建结构体，定义一个带有多个成员的新数据类型）

基本定义结构为： `struct <struct name> { ... };`

与 C/C++ 中的 `public <data type> <property name>;`

C# 的结构与 C/C++ 中的 struct 不同

可以带有方法、字段、索引、属性、运算符方法、事件

可以定义构造函数 (constructor)，但不能定义析构函数 (destructor)

但不能定义无参（默认）构造函数，无参构造函数自动定义且不能改变

不能继承其他的结构或类，也不能作为其他结构或类的基础结构

可以实现一个或多个接口

成员不能被指定为 `abstract / virtual / protected`

当使用 `New` 操作符创建结构对象时，会调用适当的构造函数用于创建结构

与类不同，结构可以不使用 `New` 操作符即可实例化

如 `<struct name> <variable name>;`

如果不使用 `New` 操作，则需要对所有字段进行初始化

字段赋值之后，才可以使用对象

注意：结构体中声明的字段无法在声明时初始化

`cannot have instance property / field initializer in struct`

如果定义构造函数，则必须在构造函数中对所有字段赋值

`field must be fully assigned before control returned to caller`

在C#中，`struct` 类型与 `class` 类型在语法上相似

结构是值类型，在栈中分配空间，直接存储成员数据

类是引用类型，数据位于堆中的分配空间，栈中保存变量为指向堆中数据对象的引用

当对象的主要成员为数据且数据量不大的情况下，使用结构性能更好

为结构分配内存 / 当离开作用域或删除结构时，性能非常好

如果结构成员非常多且复杂时，使用结构会造成损失

当堆栈空间有限，且有大量逻辑对象时，创建类优于创建结构

当表现抽象和多级别的对象层次时，类优于结构，由于结构不支持继承

通常结构的目的只包含有数据，或以数据为主

C# 中的简单类型 `int / double / bool` 都是结构类型的

可以结合结构类型与运算符重载，为C#创造新的值类型

C# - P20

枚举

(enum). 在C#中，枚举是一组命名整型常量。

C#的枚举是值类型，包含自己的值，且不能继承或传递继承。

使用enum关键字声明枚举类型的基本语法为：

```
enum <enum_name> {  
    <enumeration_list>  
}
```

enumeration_list：以逗号分隔的标识符列表。

枚举类型中的每个标识符都指派一个整型值。

默认情况下，第一个标识符的值为0。

第k个标识符的值为第k-1个标识符的值+1。

```
如 enum TestEnum {a, b, c} → a:0, b:1, c:2
```

也可以自定义标识符的值。

```
如 enum TestEnum {a=1, b=3, c=5} → a:1, b:3, c:5
```

如果部分有定义，而部分没有定义，则未定义的标识符服从默认情形。

```
如 enum TestEnum {a, b=11, c, d=1, e, f}
```

a:0, b:11, c:12, d:1, e:13, f:14

在C#中，枚举类型允许不同的标识符指派相同的整型值。

```
如 enum TestEnum {a, b=0, c=0} → a:0, b:0, c:0
```

可以通过类型转换将整型值转换为对应的枚举类的标识符。

```
如 ((TestEnum)0).ToString() → a
```

注意如果有多个标识符指派了相同的整型值。

则返回枚举列表中第一个符合的。

如果枚举类中没有匹配的整型值，则返回该整型值。

```
如 ((TestEnum)10).ToString() → 10
```

枚举类可以用于为数组中的索引赋予含义，提高程序可读性。

```
如 enum Shape {Length, Width, Height}
```

则有 Length:0, Width:1, Height:2

对于传入的数组 int[] cube = new int[3] {1, 3, 9}

相比于用 cube[0], cube[1], cube[2] 分别表示 Length, width, height

可以使用 cube[(int)Shape.Length] → 1

cube[(int)Shape.Width] → 3

cube[(int)Shape.Height] → 9

C# - P21

枚举 在C#中，提供了位标志枚举(flags enumeration)，

在 enum 关键字前加上 [Flags]，用于表示枚举字段支持位运算

FlagsAttribute.FlagsAttribute()

indicate enumeration can be treated as bit field (set of flags)

```
40 [Flags] enum Weekday {
```

None = 0,

Sunday = 0x01, first bit of input! 0000 0001, second

Monday = 0x02, bit register flag 0000 0010

Tuesday = 0x04, 0000 0100

Wednesday = 0x08 : 0000 | 0008

Thursday = 0x10, 00 0000 0000 0000

Friday = 0x20

Saturday = 0x4D 0100 0000

Weekend = Sunday | Saturday 01.00.0001

Week day = Monday | Tuesday | Wednesday | Thursday |

Wednesday - Monday Tuesday Wednesday Thursday Friday

0011 1110

类 (class)，在C#中，类定义了一个数据类型的蓝图，但没有定义任何的数据

对象是类的实例，类定义了对象由什么组成以及在对象上可执行的操作

类的成员包含类的变量(属性)和类的方法

通过 class 关键字定义类，一般的语法结构为

<access specifier> class <class name> {
 { // code segment to the left system.out.println

定义属性： [<access specifier> <data type> <variable> [= <initializer>]] *

not denied, it becomes better placing the state in America.

<access specifier> <return types> <method>

statements

multiple ports for the same server

the } additional tiny ton the piano

access specifier(访问标识符) 用于指定对类及其成员的访问规则

类的默认访问标识符为 internal，成员的默认访问标识符为 private

`datatype` 指定变量类型, `return type` 指定方法返回的数据类型

点运算符(.)用于访问类的成员，链接对象的名称与成员的名称

The use of such a system of classification is the first step in the development of every library.

2.9 - C# 构造函数

类

在C#中，类的成员函数为类中定义或提供原型的函数。可以作为类的成员在类的任何对象上操作，且可以访问该类的所有成员。成员变量通常保持私有来实现封装，只能通过公共成员函数访问。

构造函数(constructer)是类的特殊成员函数，用于创建类的新对象。

构造函数的名字与类的名字相同，且没有任何返回类型。

如果类中没有显式地定义实例构造函数，则编译器提供隐式的默认构造函数。

特点是不传入参数且方法体为空。

等价于 `public test() {}`

也可以定义需要传入参数的构造函数。

称为参数化构造函数。

特别注意：当定义了参数化构造函数时，编译器不再隐式地提供默认构造函数。

此时如果需要以不传入参数的方式构造实例，则需要显式地定义非参数化的构造函数。

析构函数(deconstructor)是类的特殊成员函数。

用于在类的对象离开其作用域时释放资源。

析构函数的名字为类的名字加前缀~，不传入参数也没有任何返回类型。

如 `~test()`

在C#中，可以使用 static 关键字定义类的静态成员。

无论类创造了多少个实例，类的静态成员都只有一个副本。

通常用于定义类所使用的常量。

静态成员的值可以直接通过类调用，而不用通过类的对象。

可以在类的内部初始化静态变量。

也可以在成员函数或类的外部进行初始化或修改（外部需要声明为 public / ...）。

如果成员函数被定义为静态成员，则其内部只能调用成员的其他静态成员。

否则抛出 `object reference required for non-static field / method`

示例代码：
`private static int num = 11;`

`public static int getStch() { return num; }`

C# - P23

3. Inheritance

8+9 = 17

继承

继承允许根据一个类来定义另一个类，使得创建/维护应用程序变得容易。已有的类称为基类 (base class)，或称父类 (parent class)。

继承已有类的新的类称为已有类的派生类 (derived class)。

在C#中，一个类可以派生自一个类和多个接口。

基本的基类/派生类的语法结构为：

```
<access specifier> class <parent class>
  { <parent property list>
    <parent method list>
    {
      <access specifiers> class <derived class> : <parent class>
        { <derive property list>
          → <derive property list>
        }
    }
  }
```

派生类继承基类的所有成员变量与成员方法。

注意在C#中不允许多重继承，即派生类从多个基类继承是禁止的。
class cannot have multiple base classes.

但是在C#中可以使用接口 (interface) 实现多重继承。

在C#中，当创建子类对象调用子类的构造函数前，会先依顺序调用基类的无参数构造函数。

特别注意：当声明为父类而初始化为子类实例时，对于在子类中使用 override / new 关键字的成员有不同的行为。

```
public class ParentClass {
  public virtual void someMethod() {
    Console.WriteLine("Parent");
  }
}

public class DeriveClass1 : ParentClass {
  public override void someMethod() {
    Console.WriteLine("Derive1");
  }
}

public class DeriveClass2 : ParentClass {
  public new void someMethod() {
    Console.WriteLine("Derive2");
  }
}
```

继承

在 C# 中，对于声明为父类类型的子类实例对象，~~无法访问子类的成员~~

用正常方法无法访问子类本身成员，而只能访问从父类继承的成员

~~或用 share override 关键字：重写父类中用 virtual 关键字声明的虚方法~~

~~或用 override 关键字：重写父类中用 virtual 关键字声明的虚方法~~

在 C# 中，如果需要使子类可以完全访问被隐藏的继承成员

可以使用基类访问表达式访问被隐藏的继承成员

如在子类 DeriveClass1 中定义新的方法

public void testBase() { ~~调用 base : 基类~~

someMethod(); ~~子类调用父类方法~~

~~或直接调用 base : 基类~~

~~或直接调用 base : 基类~~

~~或直接调用 base : 基类~~

~~或直接调用 base : 基类~~

依赖反转规则 (dependency inversion principle, DIP)

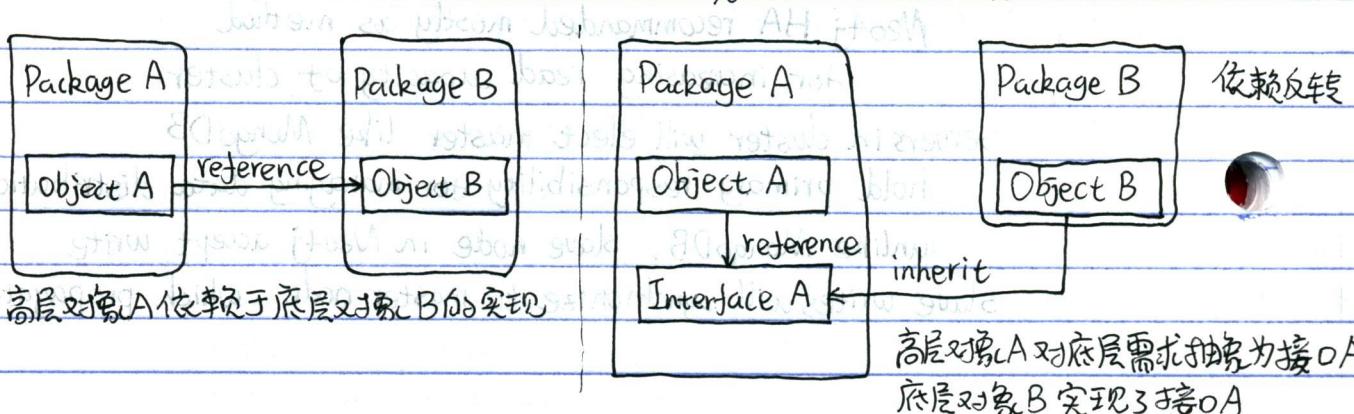
指特定的解耦 (传统的依赖关系创建在高层次上，而具体策略设置则应用在低层次模块上)

使得高层次的模块不依赖于低层次的模块的实现 (细节)

依赖关系被反转，使得低层次模块依赖于高层次模块的需求抽象

规定：1. 高层次的模块不应该依赖于低层次的模块，两者都应该依赖于抽象接口

2. 抽象接口不应该依赖于具体实现，而具体实现应该依赖于抽象接口



多态性

在C#中，多态性意味着有多重形式，同一个行为具有多个不同表现形式的能力

在面向对象编程范式中通常表现为“一个接口，多个功能”

C#的每个类型都是多态的，因为包括用户定义类型的所有类型都继承自Object

I = AA = A(78A) A = (A78)A

静态多态性：编译时函数和对象的连接机制称为早期绑定，也称为静态绑定

在C#中提供了两种技术来实现静态多态性

函数重载 (function overloading)

运算符重载 (operator overloading)

函数重载：在同一作用域内对相同函数名有多个不同定义

可以是参数列表中的参数类型不同

也可以是参数列表中的参数个数不同

但不可以重载仅有返回类型不同的函数声明

如 public int add (int a, int b)

public float add (float a, float b)

注意：不能创建抽象类的实例

动态多态性：在C#中可以通过 abstract 关键字创建抽象类，用于提供接口的实现

当派生类继承抽象类时，即完成了实现

抽象类包含抽象方法，抽象方法必须被派生类实现

动态多态性中，函数响应在运行时发生

注意：不能创建抽象类的实例

不能在抽象类外部声明抽象方法

可以通过 sealed 关键字将类声明为密封类

声明为 sealed 的类不能被继承；抽象类不可声明为 sealed

如 public abstract class Shape { }

public abstract double area();

public class Rectangle : Shape { }

public double length;

public double width;

public override double area() { }

return length * width;

}

}

动态多态性 在C#中，动态多态性是通过抽象类 (abstract class) 和虚方法 (virtual method) 实现。当定义在类中的函数需要在派生类中实现时可以使用虚方法。

在C#中，通过 virtual 关键字声明虚方法。

虚方法在类的不同派生类中可以有不同的实现。

对虚方法的调用是在运行时发生的。

与抽象方法 (abstract method) 相比：

virtual 和 abstract 都是用于修饰父类的关键字。

子类通过覆盖父类的定义进行重新定义。

{return 0;}

虚方法必须有函数体的实现，但是可以为空，如 public virtual double area();

抽象方法不允许进行函数体的实现，如 public abstract double area();

虚方法可以被子类重写，但也可以选择不重写。

抽象方法必须在子类中进行重写，与接口 (interface) 类似。

虚方法可以定义在抽象类中，也可以定义在非抽象类中。

抽象方法必须定义在抽象类中。

包含虚方法的非抽象类可以实例化。

但包含抽象方法的类（抽象类）只能被继承但无法实例化。

隐藏方法：在派生类中定义的与基类中方法同名的方法。

在C#中，通过 new 关键字定义。

如 public new double area() { return length * width; }

不仅可以隐藏基类中的虚方法，也可以隐藏基类的非虚方法。

对于虚方法的调用：

使用子类 / 父类构造的对像实例调用虚方法，则会调用子类 / 父类的方法。

对于隐藏方法的调用：

使用子类 / 父类声明的对像实例调用虚方法，则会调用子类 / 父类的方法。

重载 (overload)：相同函数名通过不同的参数列表来区分的机制。

在同一个作用域中的两个 / 多个方法函数名相同，但参数列表不同。

方法名必须相同，参数列表必须不同，返回类型可以不同。

重写 (override)：使用 override 关键字在子类中重写基类的虚方法。

方法名相同，参数列表相同，返回类型相同。