C#笔记

枚举

- 定义：被命名的整型常量的集合
- 声明枚举数据类型的位置
  - 可以：namespace/class/struct语句块
  - 不可以：函数语句块
- 声明枚举数据类型

```
enum E_PlayerType
{
    Main,
    Other,
}
```

- 声明枚举变量

```
E_PlayerType playerType = E_PlayerType.Main;
```

- 用途
  - 常和条件分支语句（if-else/switch)一起使用
  - 用来表示对象的类型/状态等（如：1-行走，2-跑步，3-待机，用整型常量不能直观明了表示具体含义，故给表示每一种状态的数字命名，使它们成为枚举类型）
- 枚举的转换
  - 和int转换

```
int i = (int)playerType;
```

  - 和string转换

```
string str = playerType.ToString();
Console.WriteLine(str);
>>Main //把枚举类型转换成字符串的结果是枚举项的名字
```

  - 把string转换成枚举类型

```
playerType = (E_PlayerType)Enum.Parse(typeof(E_PlayerType),"Other");
Console.WriteLine(str);
>>Other //默认打印的是枚举项的名字
```

一维数组

- 声明变量

```
//变量类型[] 数组名;
int[] arr1;
arr1 = new int[5];

int[] arr2 = new int[5];
```

- 数组的使用
  - 获取长度：变量名.Length
  - 获取数组中的元素：通过下标/索引
  - 修改数组中的元素：直接赋值
  - 遍历数组：通过循环（For/While/DoWhile）
  - 增减元素：需要通过一个新的数组作为“中介”进行赋值

二维数组

- 声明变量
  - ```
    int[,] arr1;
    arr1 = new int[3,3];

    int[,] arr2 = new int[3,3];
    int[,] arr3 = new int[3,3]{{1,2,3},
                               {4,5,6},
                               {7,8,9}};
    ```

- 数组的使用
  - 获取长度：
    - 获取行数：变量名.GetLength(0)
    - 获取列数：变量名.GetLength(1)
  - 获取数组中的元素：通过下标/索引
  - 修改数组中的元素：直接赋值
  - 遍历数组：通过嵌套循环（For/While/DoWhile）
  - 增减元素：需要通过一个新的数组作为“中介”进行赋值

交错数组

- 声明变量

```
// 变量类型[][] 数组名;
int[][] arr1;
int[][] arr2 = new int [3][];
int[][] arr3 = new int [3][]{ new int[]{1,2,3},
                              new int[]{1,2},
                              new int[]{1}};
```

- 数组的使用
  - 获取长度：
    - 获取行数：变量名.GetLength(0)
    - 获取列数：变量名[0].Length，某一行的列数
```

- 获取数组中的元素：通过下标/索引
- 修改数组中的元素：直接赋值
- 遍历数组：通过嵌套循环（For/While/DoWhile）
- 增减元素：需要通过一个新的数组作为“中介”进行赋值

值类型和引用类型

- 数据类型
  - 值类型：整型/浮点型/字符型/布尔类型/结构体
  - 引用类型：string/数组/class
- 区别
  - 值类型在相互赋值时把内容拷贝给对方
  - 引用类型在赋值时是让两者指向同一个值（只是地址的拷贝）
  - 特殊：string虽然是引用类型（储存在堆中），但具有值类型的特征。这是因为当string类型的变量重新赋值时，会在堆中重新分配空间，因此频繁给string赋值会产生内存垃圾。
- 本质
  - 值类型和引用类型存储在内存区域的位置是不同的
    - 值类型储存在栈空间——系统分配，自动回收，小而快
    - 引用类型储存在堆空间——手动申请和释放，大而快
  - 引用类型的地址存在栈中，内容存在”堆“中
  - ”new”相当于开了一个新的空间

函数

- 概念
  - 函数=方法，本质是一块具有名称的代码块
  - 可以使用函数的名称来执行该代码块
  - 函数是封装代码进行重复使用的一种机制
  - 调用时执行过程依然符合线性流程
- 作用
  - 封装代码
  - 提升代码的复用率
  - 抽象行为
- 位置
  - class语句块中
  - struct结构体中

    #**函数不可以在namespace中！**面向过程编程的思路需要纠正！
- 命名：函数名要用帕斯卡命名法，参数名要用驼峰命名法
- 分类：
  - 无参无返
  - 有参无返
  - 无参有返
  - 有参有返
  - 有参多返（善用数组）

Ref和Out

- 作用
  - 在函数内部改变外部传入的内容
- 使用
  - 函数参数修饰符，写在实参和形参的前面
  - 当传入的值类型在函数内部改变，或引用类型在函数内部重新声明时，外部的值也会发生变化

```
static void ChangeValue(ref int value)
{
    value = 3;
}

static void ChangeArray(ref int[] arr)
{
    arr = new int[]{100,200,300};
}
```

```
static void ChangeValue(out int value)
{
    value = 3;
}

static void ChangeArray(out int[] arr)
{
    arr = new int[]{100,200,300};
}
```

- 区别
  - ref传入的变量必须初始化，out不用
  - out传入的变量必须在内部赋值，ref不用
  - ref传入的变量必须初始化，但是在函数内部可改可不改
  - out传入的变量不用初始化，但是在函数内部必须修改（赋值）
  - 上述规则是出于安全性考虑

```
//变量必须先初始化才能用ref传入
int a = 1;
ChangeValue(ref a);
```

变长参数关键字

- 作用：可以传入n个同类型的参数
- 写法：加入关键字params

```
// params int[]意味着可以传入n个int类型的参数，传入的参数会保存在arr数组中
static int Sum(params int[] arr)
{
    int sum = 0;
    for(int i = 0; i < arr.Length; i++)
    {
        sum += arr[i];
    }
    return sum;
}
// 调用
int sum1 = Sum();
int sum2 = Sum(1,2,3,4,5);
```

- 规则
  - 关键字params后面必为数组
  - 数组的类型可以是任意类型
  - 函数的参数可以有params关键字修饰的参数和其他类型的参数共存
  - 函数参数中最多只能出现一个params关键字修饰的参数，并且要放在最后

参数默认值

- 可选参数：有参数默认值的参数
- 作用：当函数被调用时可以不传入参数，不传入参数就会使用默认值作为参数的值
- 规则
  - 支持多参数默认值
  - 如果要混用可选参数和普通参数，可选参数必须写在普通参数后面

函数重载

- 概念
  - 在同一语句块（class或struct）中
  - 函数名相同
  - 参数不同：数量不同；或者数量相同，但类型或顺序不同
- 作用
  - 用来总合处理不同参数的同一类型的逻辑
  - 命名一组功能相似的函数，减少函数名的数量，避免命名空间的污染
  - 提升程序的可读性
  - 一组函数名相同的重载函数相当于一个函数的"多重身"，如WriteLine函数就是重载函数
- 规则
  - 函数重载和返回值类型无关（即，一组同名的重载函数返回值类型可以不同），只和参数个数/类型/顺序有关
  - ref/out，params修饰的参数也算不同类型的参数，但可变参数不算
  - 调用时程序会自己根据传入的参数类型判断使用哪一个重载

```
// 以下函数是一组重载函数
static int CalsSum(int a, int b){
    return a + b;
}
```

```
static float CalsSum(float a, float b){
    return a + b;
}

static float CalsSum(float a, int b){
    return a + b;
}

static float CalsSum(int a, float b){
    return a + b;
}

static float CalsSum(ref float a, float b){
    return a + b;
}

static int CalsSum(params int[] arr){
    int sum = 0;
    for(int i = 0; i < arr.Length; i++)
    {
        sum += arr[i];
    }
    return sum;
}
```

结构体

- 概念
  - 是一种自定义的变量类型
  - 是数据和函数的集合
  - 在结构体中，可以声明各种变量和方法
- 作用
  - 用来表现存在关系的数据集合，如学生/动物等
- 规则
  - 写在namespace语句块中
  - 前缀关键字struct
  - 结构体的名字用帕斯卡命名法
  - 在结构体中声明的变量不能初始化
  - 在结构体中的函数不需要加static，函数里面能使用之前在结构体中声明过的变量
- 组成部分
  - 声明变量
  - 构造函数
  - 函数方法（表现这个数据结构的行为）
- 访问修饰符：public/private
  - public：公共的，可以被外部访问
  - private：私有的，只能在内部使用
  - 不写则默认为private
  - 加在结构体中的变量和函数之前，用来规定变量和函数能否被外部使用
- 构造函数

- 没有返回值
- 函数名必须和结构体名相同
- 必须有参数且必须在函数中对的所有变量初始化（赋初值）
- 为了方便在结构体外部对结构体初始化
- 构造函数时可以重载改的（不常用）

```
struct Student
{
    //声明变量
    int age;
    bool sex;
    string name;

    //构造函数
    public Student(int age, bool sex, string name){
        this.age = age;
        this.sex = sex;
        this.name = name;
    }
}

//在结构体外部对结构体初始化
Student s2 = new Student(18, true, "Tom");
```

面向对象概述

- 三大特性
  - 封装：用程序语言来形容对象
  - 继承：复用封装对象的代码
  - 多态：同样行为的不同表现
- 七大原则
  - 开闭原则
  - 依赖倒转原则
  - 里氏替换原则
  - 单一职责原则
  - 接口隔离原则
  - 合成复用原则
  - 迪米特法则

类和对象

- 类的概念
  - 具有相同特征和行为的一类事物的抽象
  - 类是对象的模板
  - 可以通过类创建出对象
  - 类的关键词是class
- 类和结构体的区别
  - 结构体是值类型（Value Types），而类则是引用类型（Reference Types）
  - 结构体使用栈存储（Stack Allocation），而类使用堆存储（Heap Allocation）

    - 结构体成员不能继承自其他类或结构体，也不能被其他类或结构体继承（即不具有继承的多态性），而类可以
    - 结构体成员变量不能在声明时赋值初始值，而类可以
    - 结构体没有默认的构造函数，添加构造函数时不能定义无参构造函数，只能定义带参构造函数；类有默认的构造函数，可以自定义无参构造函数或者带参构造函数
    - 类是反映现实事物的一种抽象，而结构体只是包含了不同类别数据的一种包装
- 类声明的位置
    - namespace语句块中
    - class语句块中（内部类）
- 类声明的语法

```
class 类名
{
    // 特征-变量成员
    // 行为-方法成员
    // 保护特征-成员属性

    // 构造函数和析构函数
    // 索引器
    // 运算符重载
    // 静态成员
}
class Person
{

}

// 实例化对象的形式与内存空间的关系
// 没有分配堆内存，只是在栈中开辟了内存空间用于储存堆内存的地址，但地址为null
Person p0;
Person p1 = null;
// 分配了堆内存，栈中的地址不为null
Person p2 = new Person();
```

变量成员

- 特性
    - 声明在class语句块中
    - 用来描述对象的特征
    - 是否赋值根据需求来定
    - 自带构造函数，函数名为类名
    - 类中类不能实例化成和外面的类相同类型的类的实例（不能new），否则会死循环；但是可以初始化为null
    - 访问修饰符：public/private/protected（只有内部和子类可以使用）

方法成员（函数）

- 特性
    - 声明在class语句块中
    - 是用来描述对象的行为的
    - 规则和函数声明规则相同

- 收到访问修饰符的影响
- 前面不能加static修饰符

```
class Person
{
    //变量成员
    public string name;
    public int age;

    //方法成员
    public void Speak(string str)
    {
        Console.WriteLine("{0}speaks{1}", name, str);
    }
}
```

类的构造函数

- 概念
  - 在实例化对象时会调用的用于初始化的函数
  - 如果不写，默认存在一个无参构造函数
- 特性
  - 普通构造函数在new一个新对象的时候会被调用
  - 没有返回值
  - 函数名和类名相同
  - 没有特殊需求时一般都用public访问修饰符修饰
  - 类的构造函数可以重载
  - 如果自定义了带参的构造函数，则会失去默认的无参构造函数（可理解为被顶替了）

```
class Person
{
    //变量成员
    public string name;
    public int age;
    public float money;

    //自定义构造函数可以重载
    public Person(){
        name = "Tom";
        age = 18;
    }
    public Person(string name, int age){
        this.name = name;
        this.age = age;
    }
}
```

- 构造函数的特殊写法：关键字this
  - 后面加": this()"可以调用其他的同名重载构造函数，"()"内可以加变量

- 程序根据"()"内的变量类型/数量/顺序信息自动识别该调用哪个重载函数
- 执行完"this()"后会继续执行当前函数

```
public Person(){
        name = "Tom";
        age = 18;
    }
public Person(int age){
        this.age = age;
    }
// 使用: this()先传入变量age，程序根据变量类型和个数判断应当先执行前面一个重载构造函数
// 执行完它以后继续执行这个构造函数
// 这么做可以少写重复的代码：this.age = age;
public Person(int age, string name):this(int age)
{
    this.name = name;
}
```

垃圾回收机制

- 概念：垃圾回收的过程是遍历堆上动态分配的所有对象，通过识别它们是够被引用来确定哪些对象是垃圾，哪些对象仍要被使用，然后释放垃圾的过程
- 注意点
  - 垃圾回收只负责堆内存的回收
  - 栈上的内存是由系统自动管理的，系统会自动分配和释放栈内存
- 原理
  - 堆内存分为三代：0代/1代/2代，通过分代算法来实现
  - 新分配的对象都会被配置在第0代内存中
  - 0代内存满时会触发0代垃圾回收机制，1代内存满时会触发0代和1代垃圾回收机制
  - 在回收过程开始时，垃圾回收器默认堆中全是垃圾，会进行以下3步骤
    - 标记对象：从根开始检查引用对象，把可达的对象标记了，不可达的对象不会被标记
    - 搬迁对象：把标记过的可达对象搬迁到下一代内存中并修改引用地址
    - 释放垃圾：未标记的不可达对象都是垃圾，会被释放

属性成员

- 作用
  - 用于保护变量成员
  - 为成员属性的获取和赋值添加逻辑处理
  - 可以让变量成员在外部只能获取不能修改，或只能修改不能获取（通过在关键字get和set之前添加访问修饰符来实现）
  - 可以通过set和get关键字进行加密和解密
- 命名：使用帕斯卡命名法
- 关键字
  - get：返回内容，返回的类型要和属性类型相同
  - set：设置内容，set中用value表示传入的值

- value：表示外部传入的值

```
// money是class中的变量成员
// 通过在get和set前添加访问修饰可以让money在外部只能获取不能修改，或只能修改不能获取

//只能获取不能修改
public int Money
{
    get{
        return money;
    }
    private set{ //set被设置成了私有，故外部不能修改money
        money = value;
    }
}

//只能修改不能获取
public int Money
{
    private get{ //get被设置成了私有，故外部不能获取money
        return money;
    }
    set{
        money = value;
    }
}

//get和set可以只有一个
public int Money
{
   get{
        return money;
    }
}
```

- 自动属性
  - 可以使用自动属性以减少代码量
  - 前提是属性不需要通过逻辑处理

```
//外部能得到但不能修改的属性
public float Height
{
    get;
    private set;
}
```

索引器

- 概念
  - 让对象可以像数组一样通过索引访问其中的元素，使程序更简单容易编写
  - 索引器可以重载，重载规则和函数重载规则相同
  - 可以让我们以"实例[]"形式访问类中的元素，比较适合在类中有数组变量时使用，方便访问
  - 除了class，结构体中也可以写索引器

```
class Person
{
    private string name;
    private int age;
    private Person[] friends;

    //返回的是Person类型的值，传入的是表示索引的整型，this用来指代使用索引器的实例本身
    public Person this[int index]
    {
        get
        {
            return friends[index];
        }
    }
    //索引器可以重载
    public string this[string str]
    {
        get
        {
            switch(str)
            {
                case "name":
                    return this.name;
                case "age":
                    return this.age.ToString();
            }
            return "";
        }
    }
}
```

静态成员

- 概念
  - 用静态关键字static修饰的变量成员/方法成员/属性成员等成为静态成员
  - 静态成员直接用类名点出使用而不用通过实例化
    - 静态成员在程序开始运行时就会被分配内存空间，因此能直接使用
    - 静态成员和程序同生共死
    - 静态成员具有唯一性和全局性
  - 静态函数中不能直接使用非静态成员
- 作用
  - 常用的唯一的变量的声明
  - 方便他人获取的对象的声明
  - 常用的唯一的方法的声明，如相同规则的数学计算函数

静态类

- 概念：用关键字static修饰的类
- 特点
  - 只能包含静态成员
  - 不能被实例化

- 作用
  - 将常用的静态成员写在静态类中，方便使用
  - 静态类不能被实例化，更能体现工具的唯一性（如Console就是一个静态类）

```
static class TestStatic
{
    // 静态变量成员
    public static int testIndex = 0;

    // 静态方法成员
    public static void TestFunc()
    {

    }

    // 静态属性成员
    public static int TestIndex
    {
        get;
        set;
    }
}
```

静态构造函数

- 概念：在构造函数前加上关键字static修饰
- 特点
  - 静态类和普通类都可以有静态构造函数
  - 不能使用访问修饰符
  - 不能有参数
  - 当第一次使用类中的成员时会自动调用，且只自动调用一次
  - 在普通类中，静态构造函数和普通构造函数不构成重载函数
- 作用：在静态构造函数中初始化静态变量

```
// 在静态类中使用静态构造函数初始化静态变量
static class StaticClass
{
    public static int testInt1 = 100;
    public static int testInt2 = 200;

    static StaticClass()
    {
        testInt1 = testInt2 = 300;
        Console.WriteLine("静态构造函数被调用");
    }
}

// 普通类中使用静态构造函数
class Test
{
    public static int testInt = 200;
```

```
    static Test()// 只要用到类中的成员就会被调用
    {
        Console.WriteLine("静态构造函数被调用");
    }

    public Test()// new实例的时候才被调用
    {
        Console.WriteLine("普通构造函数被调用");
    }
}
```

拓展方法

- 概念
  - 为现有非静态的变量类型添加的新方法
- 作用
  - 提升程序的拓展性
  - 不需要在对象中重新写方法
  - 不需要通过继承来添加方法
  - 为封装的类型（如系统自带的int/string等）写额外的方法
- 特点
  - 一定写在静态类中
  - 一定是个静态函数
  - 第一个参数为拓展目标（变量类型名），且前面用关键词this修饰
  - 后面的参数正常写（变量名+参数名）

```
// 为现有的非静态变量类型int拓展了新的方法成员SpeakValue
// 方法成员是需要实例化对象后才能使用
// value为类int的实例化对象
static class Tools
{
    public static void SpeakValue(this int value)
    {
        Console.WriteLine(value + "New Function Added");
    }
}

int i = 10;
i.SpeakValue();
```

运算符重载

- 作用：让自定义的类和结构体对象可以进行运算
- 关键字：operator
- 特点
  - 一定是一个公共的静态的方法
  - 返回值写在operator前
  - 一个运算符可以有多个重载

- 逻辑处理自定义
- 不能使用ref和out

```
class Point
{
    public int x;
    public int y;

    // 为运算符+写重载，使两个Point类型的变量可以用+进行运算
    public static Point operator +(Point p1, Point p2)
    {
        Point p = new Point();
        p.x = p1.x + p2.x;
        p.y = p1.y + p2.y;
    }

    // 一个运算符可以有多个重载
    public static Point operator +(Point p1, int value)
    {
        Point p = new Point();
        p.x = p1.x + value;
        p.y = p1.y + value;
    }
}
```

- 可重载和不可重载的运算符
  - 可重载的运算符：重载函数的参数个数必须和运算符的目数相同
    - 算数运算符：全部
    - 逻辑运算符：只有"!"（逻辑非）
    - 位运算符：全部
    - 条件运算符：必须成对出现（如 > 和 < 为一对，写了一个的重载必须写另一个）
  - 不可重载的运算符
    - &&（逻辑与）和||（逻辑或）
    - []（索引符）
    - ()（强转运算符）
    - ? ：（条件运算符）
    - .（点）
    - =（赋值等号）

内部类

- 概念：在一个类中再声明一个类
- 特点
  - 使用时要使用包裹者点出自己
  - 只有当前面添加public访问修饰符时才能被外面使用

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

```
    public Body body;

    public class Body
    {
        public Arm leftArm;
        public Arm rightArm;
        public class Arm
        {

        }
    }
}
```

继承

- 概念
  - 一个类A继承一个类B
    - 类A会继承类B的所有成员（变量/属性/方法）
    - 类A将拥有类B的所有特征和行为
    - 子类可以有自己的特征和行为
- 语法

```
class 类名: 被继承的类名
{

}
```

- 特征
  - 单根性：只能继承自一个父类（只能有一个爹）
  - 传递性：可以间接继承父类的父类中的所有内容（孙子可以继承外公的血脉）
- 访问修饰符：protected
  - 只在继承的时候起作用
  - 只有内部和子类可以访问

```
// 老师类
class Teacher
{
    public string name;
    public int number;

    public void speakName()
    {
        Console.WritLine(name);
    }
}

// 教学老师属于老师，是老师类的子类
class TeachingTeacher: Teacher
{
    public string subject;

    public void SpeakName()
```

```
    {
        Console.WritLine(subject + " teacher");
    }
}

// 语文老师属于教学老师，是教学老师类的子类
class ChineseTeacher: TeachingTeacher
{
    public void Skill()
    {
        Console.WriteLine("一行白鹭上青天");
    }
}
```

- 作用
  - 让两个类产生联系
  - 提高代码的复用率

里氏替换原则

- 概念
  - 任何父类出现的地方，子类都可以替代
  - 父类容器装子类对象，因为子类对象包含了父类的所有内容
- 声明类和实例类
  - 声明类：对象声明时前面写的类
  - 实例类：对象执行时实例化的类
- is和as
  - is
    - 判断一个对象是否为某个类的实例化对象
    - 返回值是bool类型的，是为True，不是为False
  - as
    - 将一个对象转换为某个类的实例化对象
    - 返回值：成功则返回这个类的对象，失败则返回null
  - is和as往往配合使用
- 作用
  - 方便对象的存储和管理（如，用父类数组作为容器装子类对象）

```
class GameObject
{

}

// 子类Player
class Player: GameObject
{
    public void PlayerAttack()
    {
        Console.WriteLine("玩家攻击");
    }
}
```

```
// 子类Monster
class Monster: GameObject
{
    public void MonsterAttack()
    {
        Console.WriteLine("怪物攻击");
    }
}

// 主函数中
// 任何父类出现的地方，子类都可以替代（里氏替换原则--父类容器装子类对象）
// 其中GameObject是对象player的声明类，Player是对象player的实例类
GameObject player = new Player();
GameObject[] objects = new GameObject[]{new Player(), new Monster};

// 判断player是否为类Player的实例化对象（看new的是什么类型的）
if(player is Player)
{
    // 若是，则将player转换成真正的类Player的实例化对象
    // 虽然它new的是Player类型的对象但它装载了父类GameObject的容器中，还要用as转换
    Player p = player as Player;
    p.PlayerAttack();
}

// 等价的写法，更省代码
if(player is Player)
{
    (player as Player).PlayerAttack();
}

// 判断数组中的对象是玩家还是怪物，然后启动它们对应的攻击方式
if(int i = 0; i < objects.Lenth; i++)
{
    if(objects[i] is Player)
    {
        (player as Player).PlayerAttack();
    }
    else if(objects[i] is Monster)
    {
        (monster as Monster).MonsterAttack;
    }
}
```

继承中的构造函数

- 特点
  - 当声明一个子类对象时，先执行父类的构造函数再执行子类的构造函数
  - 父类构造函数必须被执行，这是规则
- 执行顺序：……父类的父类的构造函数——父类的构造函数——子类的构造函数
- 父类的无参构造函数
  - 子类在实例化时默认自动调用的是父类的无参构造，如果父类的无参构造被带参构造顶替掉了，那么在声明子类时会报错

    - 解决方法：1.在父类中手动重载一个无参构造；2.使用base调用父类的带参构造函数
- 使用base调用指定的父类的构造函数

```
class Father
{
    public Father(int i)
    {
        Console.WriteLine("父类的带参构造函数被调用");
    }
}

class Son: Father
{
    //程序根据()中的变量类型和个数判断应当调用父类中的哪一个重载构造函数
    // this指的是自身类，base指的是父类
    // 执行完base指代的父类构造函数语句后，再执行子类的构造函数
    public Son(int i): base(i)
    {
        Console.WriteLine("子类的带参构造函数被调用");
    }

    public Son(int i, string str): this(i)
    {
        Console.WriteLine("子类的带两个参数的构造函数被调用");
    }
}

// 主函数中
Son s = new Son(1, "eee");

// 控制台输出
父类的带参构造函数被调用
子类的带两个参数的构造函数被调用
```

Object——万类之父

- 概念：object是所有类型的基类，本质是一个类/class（引用类型）
- 作用
    - 可以利用里氏替换原则，用object容器装所有的对象
    - 可以来表示不确定型，作为函数参数类型
- 使用
    - 若object用来装引用类型的对象，则用is和as进行转换（上面讲过）
    - 若object用来装值类型的对象，则通过直接赋值和“()"（强转运算符）进行转换
    - 若object用来装特殊的string类型（是引用类型但是具有值类型的特征）的对象，则
        - 用.ToString()转化成string
        - 用as转化成string

```
object str = "123";

// 以下两种方式都可以将object装的string类型的变量转化成真正的string
string str1 = str.ToString();
string str2 = str as string;
```

- 若object用来装特殊的数组类型的对象，则
  - 用“()"（强转运算符）转换成数组
  - 用as转换成数组

```
object arr = new int[10];

int[] arr1 = arr as int[];
int[] arr2 = (int[])arr;
```

万类之父Object中的方法

- 静态方法
  - public static bool Equals(Object? objA, Object? objB);
    - 用于比较两个对象是否相等
  - public static bool ReferenceEquals(Object? objA, Object? objB);
    - 比较两个对象是否有相同的引用，用来比较引用类型的对象
    - 值类型对象的返回值始终是false
- 虚方法
  - public virtual bool Equals(Object? obj);
    - 默认实现比较两者是否为同一个引用，即相当于方法ReferenceEquals
    - 微软在所有值的基类Sysyem.ValueType中重写了该方法，用来比较值相等
  - public virtual int GetHashCode();
    - 获取对象的哈希码
  - public virtual string? ToString();
    - 用于返回当前对象代表的字符串
    - 我们可以重写它来定义自定义类型的对象的转字符串规则
- 成员方法
  - public Type GetType();
  - protected Object MemberwiseClone();
    - 获取对象的浅拷贝对象，即返回一个新的对象
    - 但是新对象中的引用类型变量会和老对象一致

装箱和拆箱

- 概念：值类型和object进行互相转化时发生的行为
  - 装箱：用object储存值类型
    - 把值类型用引用类型存储
    - 栈内存会迁移到堆内存中
  - 拆箱：把object转换成值类型

        - 把引用类型储存的值类型取出来
        - 堆内存会迁移到栈内存中
- 优劣
    - 优点：不确定数据类型时可以方便参数的存储和传递
    - 缺点：栈和堆之间存在内存迁移，增加性能消耗
- 要尽量避免使用

```
static void SimpleBoxUnbox()
{
  int myInt = 25;

  // 装箱操作，把值类型转换成引用类型
  object boxedInt = myInt;

  // 拆箱操作，把引用类型用强转符号转换成值类型
  int unboxedInt = (int)boxedInt;
}
```

密封类

- 概念
    - 用关键字sealed修饰的类（结扎印章）
    - 无法被继承（断子绝孙）
- 作用
    - 不允许最底层子类被继承
    - 保证程序的安全性

多态

- 多态的分类
    - 编译时的多态（如，函数的重载）
    - 运行时的多态（虚函数重写/抽象类和抽象函数/接口）
- 概念：让继承同一父类的子类们在执行相同方法/函数时有不同的表现
- 目标：让同一个对象调用同一个方法有唯一的行为
- 关键字
    - virtual——虚函数
    - override——重写，常和关键字virtual配合使用实现虚函数重写
    - base——父类，要保留父类函数中的行为时可以使用

```
class GameObject
{
    public string name;

    // 为该类写一个构造函数
    public GameObject(string name)
    {
        this.name = name;
    }

    // 为父类写一个虚函数，GameObject是该函数的声明类
```

```
    public virtual void Attack()
    {
        Console.WriteLine("游戏对象进行攻击");
    }
}

class Player: GameObject
{
    public Player(string name): base(name)
    {

    }

    // 重写其父类的虚函数Attack()
    public override void Attack()
    {
        // 可通过base来保留父类的行为，能少些代码
        base.Attack();
        Console.WriteLine("玩家对象进行攻击");
    }
}
```

- 虚函数和一般函数的区别
  - 一般函数在编译时就静态地编译到了执行文件中，其相对地址在程序运行期间是不发生变化的
  - 而虚函数在编译期间是不被静态编译的，它的相对地址是不确定的，它会根据运行时期对象实例来动态判断要调用的函数
- 虚函数的使用
  - 声明类和实例类（里氏替换原则中的两个重要概念，再次复习一下）
    - 对象声明时定义的类叫声明类，执行时实例化的类叫实例类
  - 当调用一个对象的函数时，程序会直接去检查这个对象的声明类，看所调用的函数是否为虚函数
    - 如果不是虚函数则执行该函数
    - 如果是虚函数（有virtual关键字），就不会立刻执行该函数了，而去检查对象的实例类
    - 在实例类里，如果有重新实现该虚函数（有override关键字修饰的和虚函数同名的函数），则马上执行这个函数
    - 如果没有，程序会不停地往上找实例类的父类，并对父类重复刚才在实例类里的检查，直到找到第一个重写了该虚函数的父类为止，然后执行这个函数
  - 如果声明类中有虚函数且实例类和声明类相同，则仍然执行虚函数（因为找不到override关键字）
  - 有new关键字修饰的同名函数不是重写，而是覆盖父类中的函数

```
class A
{
    public virtual void Fun()
    {
        Console.WriteLine("Fun in A");
    }
}

class B: A
{
```

```
    public override void Fun()
    {
        Console.WriteLine("Fun in B");
    }
}

class C: B
{

}

class D: A
{
    // new表明覆盖父类里的同名类，而不是重新实现
    public new void Fun()
    {
        Console.WriteLine("Fun in D");
    }
}

// 主函数中
A a = new A();
A b = new B();
A c = new C();
A d = new D();

a.Fun();// Fun in A，声明类的实例类相同，抽象函数依然会执行
b.Fun();// Fun in B，执行的是实例类中的重写抽象函数
c.Fun();// Fun in B，执行的是实例类的父类中的重写抽象函数
d.Fun();// Fun in D，子类中的同名函数覆盖父类中的函数
```

抽象类

- 概念：用抽象关键字abstract修饰的类
- 特点
    - **不能被实例化**（不能new）
    - 但遵循里氏替换原则，可以用父类容器装子类
    - 可以包含抽象方法
    - 继承抽象类的类必须重写其抽象方法

抽象函数

- 概念：用抽象关键字abstract修饰的函数
- 特点
    - **只能在抽象类中声明**
    - 没有函数体，“()”后直接用“;”结束，没有“{}”
    - 不能是私有的（前面必须加public或proteced访问修饰符，保证子类可以访问）
    - 抽象类的子类继承后必须实现用override重写

接口

- 概念

    - 行为的抽象规范
    - 是一种自定义的类型
    - 关键字interface
- 特点
    - 不包含变量成员
    - 只包含**方法/属性**/索引器/事件
    - 接口中的成员不能被实现
    - 接口中的成员不能是私有的（不写访问修饰符**默认为public**，若要写只能用public或protected）
- 继承
    - 接口不能继承类，但是可以继承另一个接口或多个接口
    - 接口继承接口后，不需要实现里面的内容
    - 接口不能实例化对象，但是可以作为容器存储对象（也遵循**里氏替换原则**）
    - 类可以继承多个接口，也可以继承1个类和n个接口
    - 类继承接口后，必须实现接口中的所有成员，并且前面必须加public（一般不加protected）
    - 类在实现函数时可以写虚函数（virtual）
- 命名
    - 帕斯卡命名法
    - 前面加字母“I”

```
// 接口1，所有账户都要继承
public interface IBankAccount
{
    // 接口中写的函数都不需要写方法体
    void PayIn(decimal amount);// 存钱函数
    bool WithDraw(decimal amount);// 取钱函数
    decimal Balance{get;}// 余额属性
}

// 类1为普通账户，继承了接口1
public class CommonAccount: IBankAccount
{
    private decimal balance;

    // 在类中实现接口中声明的函数
    public void PayIn(decimal amount)
    {
        balance += amount;
    }

    public bool WithDraw(decimal amount)
    {
        if(balance >= amount)
        {
            balance -= amount;
            return true;
        }
        Console.WriteLine("余额不足");
        return false;
    }

    //在类中实现接口中声明的属性
```

```
    public decimal Balance
    {
        get
        {
            return balance;
        }
    }
}

// 接口2，高级账户要继承这个接口
public interface IBankAdvancedAccount
{
    void DealStartTip();
    void DealEndTip();
}

// 类2为高级账户，继承了接口1和接口2
public class AdvancedAccont: IBankAccount, IBankAdvancedAccount
{
    private decimal balance;

    // 在类中实现接口1中声明的函数
    public void PayIn(decimal amount)
    {
        balance += amount;
    }

    public bool WithDraw(decimal amount)
    {
        if(balance >= amount)
        {
            balance -= amount;
            return true;
        }
        Console.WriteLine("余额不足");
        return false;
    }

    //在类中实现接口1中声明的属性
    public decimal Balance
    {
        get
        {
            return balance;
        }
    }

    // 在类中实现接口2中声明的函数
    public void DealStartTip()
    {
        Console.WriteLine("交易开始，请注意周围环境");
    }

    public void DealEndTip()
    {
```

```
        Console.WriteLine("交易结束，请带好随身物品欢迎下次光临");
    }
}
```

- 显示实现接口（不常用）
  - 当一个类继承了两个接口但是接口中存在同名方法时使用
  - 显示实现接口时不能写访问修饰符
  - 接口名.方法名实现
  - 类继承接口后，其实例若要执行方法，必须转成父类接口后才能执行对应的方法（as 接口名）

密封函数

- 概念：用关键字sealed修饰的重写函数
- 特点
  - 让虚方法（virtual）和抽象方法（abstract）在之后的子类中不能被重写
  - 一定和override一起出现，加在override前面
- 作用：提高程序的安全性

命名空间

- 概念
  - 命名空间是用来组织和重用代码的语句块
  - 就像一个工具包，类是一件件的工具
  - 关键字namespace，命名空间名用帕斯卡命名法
- 使用
  - 不同命名空间相互使用，需要引用命名空间或指明出处
    - 代码最上方写“**using 命名空间名**”
    - 在引用对象前面写“**命名空间.**”
  - 不同命名空间中允许有同名的类
  - 命名空间可以包裹命名空间，引用子命名空间时必须要点出来（父命名空间.子命名空间）