Static

单例设计模式

在项目中 有很多类仅仅只需要一个实例

例如：玩家信息类 游戏管理类 资源管理 存档数据类

满足单一职责的原则

即 某一个类 只有一个静态对象会被实例化 实现数据的互通

实现流程：

1 私有的构造方法

2 静态的自身类型的对象

3 静态访问器或方法 仅第一次调用是在类的内部去new

C#单例的实现

```
public class Admin
{
    public int x = 0;
    //静态的自身类型的对象
    private static Admin instance;

    //静态访问器 只有第一次调用时 对instance进行实例化 以后调用直接返回
    public static Admin Instance
    {
        get {
            if (instance == null)
                instance = new Admin();
            return instance;
        }
    }

    //私有的构造方法 保证外部不能实例化对象 防止多个对象存在
    private Admin()
    {

```

```
        }
    }

        //调用
        static void Main(string[] args)
        {
            //Admin a = Admin.Instance;
            //a.x= 100;
            //Admin b = Admin.Instance;
            //Console.WriteLine(b.x);
            Admin.Instance.x = 200;
            Admin a = Admin.Instance;
            Console.WriteLine(a.x);
        }
```

运算符重载

operator 关键字

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

      public static MyClass operator +(MyClass mc1,MyClass mc2)
      {
          MyClass mc = mc1;
          mc.x = mc1.x + mc2.x;
          mc.y = mc1.y + mc2.y;
          return mc;
      }
  }

      static void Main(string[] args)
      {
          MyClass mc1 = new MyClass();
          MyClass mc2 = new MyClass();
          mc1.x = 100;
          mc2.x = 200;
```

```
            MyClass mc3 = mc1 + mc2;
            Console.WriteLine(mc3.x);
        }
```

== ！=

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

        public static bool operator ==(MyClass mc1,MyClass mc2)
        {
            if (mc1.x == mc2.x && mc1.y == mc2.y)
                return true;
            return false;
        }

        //重载了==或者!= 必须重载另一个 否则编译错误
        public static bool operator !=(MyClass mc1, MyClass mc2)
        {
            if (mc1.x == mc2.x && mc1.y == mc2.y)
                return true;
            return false;
        }
   }
```

true false

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

        public static bool operator true(MyClass mc1)
```

```
        {
            if (mc1.x > 0)
                return true;
            return false;
        }
        public static bool operator false(MyClass mc1)
        {
            if (mc1.x <= 0)
                return true;
            return false;
        }
    }

        static void Main(string[] args)
        {
            MyClass mc1 = new MyClass();
            MyClass mc2 = new MyClass();
            mc1.x = 1;
            if (mc1) {

                Console.WriteLine("对象的x结果 >0");
            }
        }
```

可以重载的运算符：

数学运算符：+-*/%

比较运算符 ： == != > < >= <= 成对重载

单目运算符： ！ ~ ++ -- true false(成对重载)

位运算 ： & | ^ << >>

特殊：[ ] 不能重载 但是可以定义索引器

索引器

索引器(Indexer)

访问类的成员 使用set和get语句

允许一个对象可以像数组一样被索引

当定义一个索引器后 该类的行为就像是一个虚拟数组(virtual arrary)

对象名为 a a中有成员变量x 访问除了使用a.x以外 还可以a[x]

例：

```
public class MyClass
   {
        public string[] names;

        public MyClass()
        {
            names = new string[4];
            names[0] = "张三";
            names[1] = "李四";
            names[2] = "张三";
            names[3] = "张三";
        }

        public string this[int index]
        {
            get {
                return names[index];
            }
        }

        public int this[string name]
        {
            get {

                for (int i = 0; i < names.Length; i++)
                {
                    if (names[i].Equals(name))
                        return i;
                }
                return -1;
            }
        }


```

```
35
36              MyClass mc1 = new MyClass();
37              Console.WriteLine(mc1.names[1]);
38              Console.WriteLine(mc1[1]);
39              Console.WriteLine(mc1["李四"]);
40
41              string x = "asdf";
42
43              for (int i = 0; i < x.Length; i++)
44              {
45                  Console.WriteLine(x[i]);
46              }
```

继承

继承是面向对象程序设计的最重要的概念之一

用一个类定义另一个新的类

新定义的类可以不用编写新的成员变量和成员方法 而是继承拥有已有类的成员

目的：节省开发时间 实现代码复用 让程序的创建和维护更简单

父类（基类）

子类（派生类）

动物类是一个父类

动物类有自己的的成员变量和成员方法

{

名字 体重 颜色等

吃东西 跑 叫等

}

狗类 猫类则是一个子类 都继承与动物类 也都应该包括动物类所有成员

```
1  /// <summary>
```

```
    /// 父类  动物类
    /// </summary>
    public class Animal
    {
        public string name; //最高访问权限 可以在任何地方被访问到
        protected int x; //保护的 只有父类自身 或者子类内部可以访问
        private int y; //最低的访问权限 只有父类自身可以访问

        public void Eat()
        {
            Console.WriteLine("吃东西");
        }
    }

    /// <summary>
    /// 子类 狗类 继承于动物类
    /// </summary>
    public class Dog : Animal
    {
        public int a;
        public Dog()
        {
            a = 200;
            name = "旺财";
            x = 100; //在子类内部可以访问父类的保护成员
            //y = 200; 无法访问父类的私有成员
        }

        public void Fun()
        {
            Console.WriteLine("狗会看家");

        }
    }

    /// <summary>
    /// 猫类 子类  继承于动物类
    /// </summary>
    public class Cat : Animal
    {

        public Cat()
```

```
        {

        }

        public void Fn()
        {
            Console.WriteLine("猫会抓老鼠");
        }

    }
```

子类调用父类的成员

```
/// <summary>
    /// 父类  动物类
    /// </summary>
    public class Animal
    {
        public string name;
        protected int x;
        private int y;

        public void Eat()
        {
            Console.WriteLine("吃东西");
        }
    }

    /// <summary>
    /// 子类 狗类 继承于动物类
    /// </summary>
    public class Dog : Animal
    {
        public int a;
        public string name;
        public Dog(string name)
        {
```

```
            Console.WriteLine(name);//形参的name
            Console.WriteLine(this.name);//自身类的name
            Console.WriteLine(base.name);//父类的name
        }

        public void Eat()
        {
            //使用base.访问父类的方法
            base.Eat();
        }

    }
```

子类和父类的构造顺序问题：

先调用父类的构造方法 然后调用子类的构造方法

当子类有有参构造方法时 也会调用父类的默认构造方法

如果需要在子类中调用父类的有参构造方法 需要显式调用

```

/// <summary>
    /// 父类  动物类
    /// </summary>
    public class Animal
    {
        public string name;
        protected int x;
        public Animal(string name)
        {
            Console.WriteLine("父类的有参构造");
        }
    }

    /// <summary>
    /// 子类 狗类 继承于动物类
    /// </summary>
    public class Dog : Animal
    {
       public int a;
```

```
        //父类有参构造方法的显式调用
        public Dog(string name):base(name)
        {
            Console.WriteLine("狗类的构造");
        }
    }
```

析构调用顺序

子类先析构 父类再析构

类型判断和类型转换

is as

子类类型可以被看做是父类类型

但是父类类型不可以被看做是子类类型

原因：子类类型>父类类型 子类可能包括父类所没有的数据

```
    public class ClassA
    {

    }

    public class ClassB :ClassA
    {
        public int x;
    }

    public class ClassC:ClassA
    {
    }


    //调用

     static void Main(string[] args)
    {
```

```
            ClassB c = new ClassB();
            ClassC c2 = new ClassC();
            Fun1(c2);

            ClassA c1 = new ClassA();
            //Fun2(c1); 要求子类类型 父类传递报错
        }

        static void Fun1(ClassA w)
        {
            //把父类类型 强行转化为子类类型
            //判断对象的类型 是不是classB类型 或者是不是classB的父类类型
            if (w is ClassB)
            {
                //不会抛异常的转换 当转换失败 返回null 适用于引用类型
                ClassB x = w as ClassB;
            }

            if (w is ClassC)
            {
                //不会抛异常的转换 当转换失败 返回null 适用于引用类型
                ClassC x = w as ClassC;
            }
        }

        static void Fun2(ClassB c)
        {

        }
```

多态

静态多态 ：编译时发生

1 函数重载 2 运算符重载

运行多态(动态多态) ：运行时发生

同样的操作作用于不同的对象 可以出现不同的解释 产生不同的结果

一个接口 多个功能

多态的实现：

1 以继承为前提

2 子类重写父类的虚方法或抽象方法

3 父类的引用指向子类的对象

```
  public class Ball
{
    //父类定义的虚方法
    public virtual void Play()
    {
        Console.WriteLine("玩球");
    }

    public virtual void Fun()
    {
        Console.WriteLine("Ball:Fun");
    }
}

public class BasketBall : Ball
{
    public override void Play()
    {
        Console.WriteLine("打篮球");
    }
}

public class FootBall : Ball
{
    //public override void Play()
    //{
    //    Console.WriteLine("踢足球");
    //}
```

```
    }

    class Program
    {
        static void Main(string[] args)
        {
            //父类的引用b  指向子类basketball的对象
            //upcasting 向上转型  子类可以自动通过向上转型转换为父类类型
            Ball b = new BasketBall();
            b.Play();//子类重写了Play 调用子类的方法
            b.Fun();//子类没有重写Fun 调用父类的方法
            Ball b2 = new FootBall();
            b2.Play();
        }

    }
```

抽象类和抽象方法

抽象类 不能被实例化对象 由abstract关键字修饰

抽象类只是为子类服务 一般用于继承

子类继承于抽象类时 必须全部实现父类的抽象方法 否则 子类也不能被实例化

抽象类决定了子类的规范和格式

```
    //使用abstract定义一个抽象类
    public abstract class MyClass
    {
        protected int w;
        protected int l;
        protected int s;

        //抽象方法 不能实现
        protected abstract void AFun();
        protected abstract int X { get; set; }

        protected void Fn() { }
```

```
    }

    public class MyClass1 : MyClass
    {
        //AFun 和 X访问器 在子类都必须有实现 { }
        protected override void AFun()
        {
            base.Fn();
        }

        protected override int X
        {
            get
            {
                return 100;
            }
            set
            {

            }
        }
    }
```

面向对象的程序设计 三大特征

封装 继承 多态

封装：

封装可以把一个对象的属性私有化(private)

仅仅自己在内部访问 不给外部提供访问的权限

或者只能按照开发者规定的方法来进行访问 对使用者的操作进行限制

对成员进行精确的控制 隐藏信息 实现细节

良好的封装可以减少代码的耦合

继承：

继承提高了复用性和可维护性

让类与类之间产生了耦合关系（弊端）

不可避免使用继承 因为继承是多态实现的前提条件

开发者不必为每个类都编写功能的实现 只需要对父类进行处理

多态

子类的功能可以被父类的方法或引用变量所调用 向后兼容

提高的代码的可扩展性和可维护性

把不同的子类都当作父类来看 可以屏蔽不同子类对象之间的差异

写出通用的代码 做出通用的编程 以适应需求的不断变化