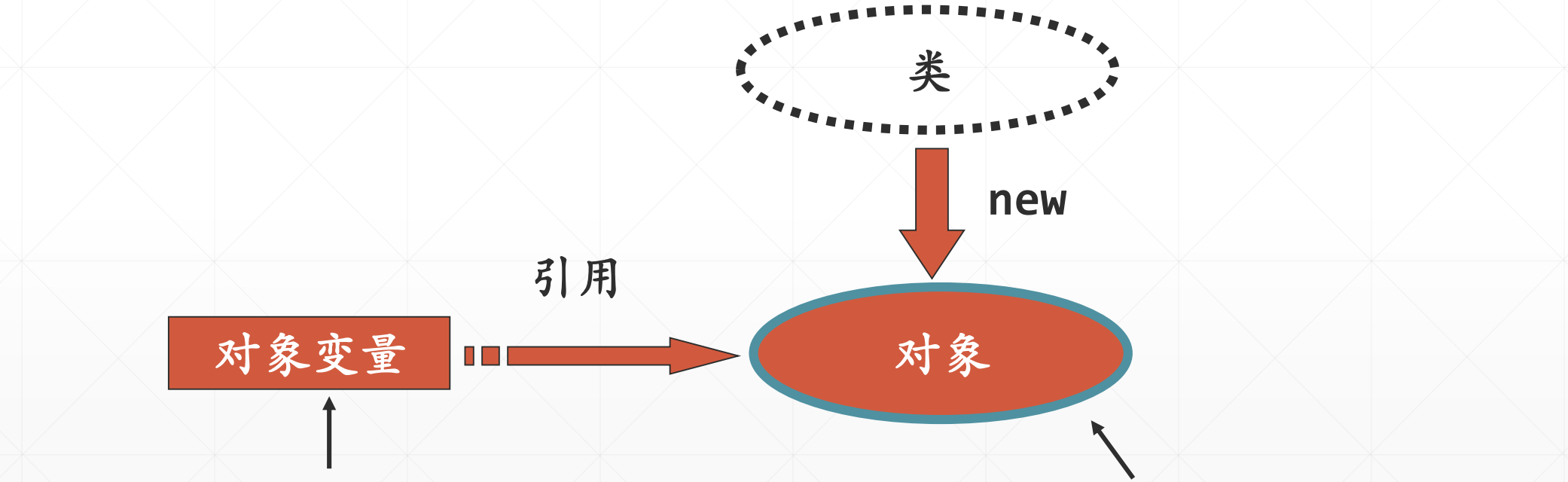
对象的那些事儿

北京理工大学计算机学院
金旭亮

对象与对象变量

“对象变量”与“对象”之间的关系……

对象变量用于“引用”一个真实的对象。



方法中所定义的对象变量保存在线程堆栈（Thread Stack）中。

对象生存于托管堆（Managed Heap）中，当不用时，CLR会自动回收其内存。

线程堆栈 vs 托管堆

1. 程序代码其实是由线程负责执行的，每个线程都拥有一个用于保存临时数据的特定内存区域，称为“线程堆栈（Thread Stack）”。

2. 保存在线程堆栈中的数据，当它所关联的线程运行结束时，这个线程堆栈会被销毁，导致其中的数据“全没了”。

3. 保存在托管堆中的数据，只有当整个程序结束时，才会被全部“销毁”。

C#: 引用类型 vs. 值类型

“类”类型的变量属于“引用类型（Reference Type）”，其引用的对象占用的内存位于“托管堆（managed heap）”中。

int之类简单类型（还包括struct等）的变量属于“值类型（Value Type）”，方法内部所定义的值类型的变量，其占用的内存位于“线程堆栈（thread stack）”中。

思索

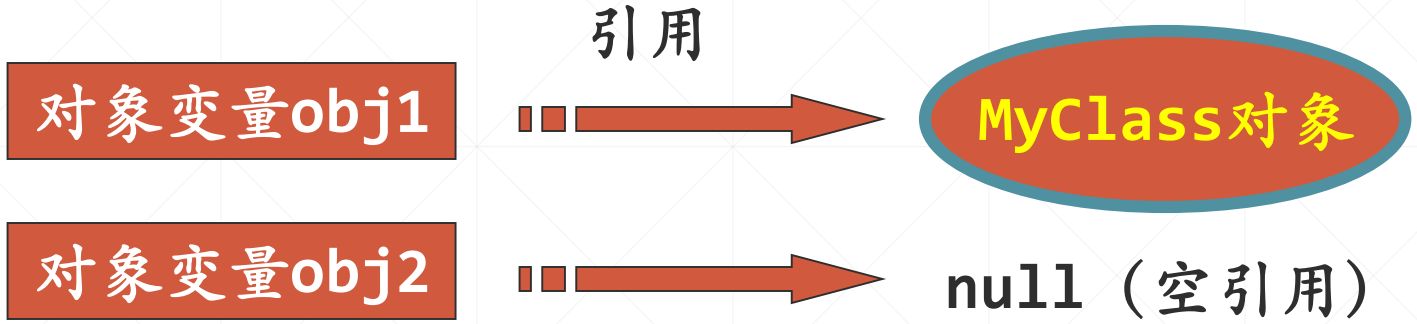
假设MyClass是一个类，请看以下C#代码：

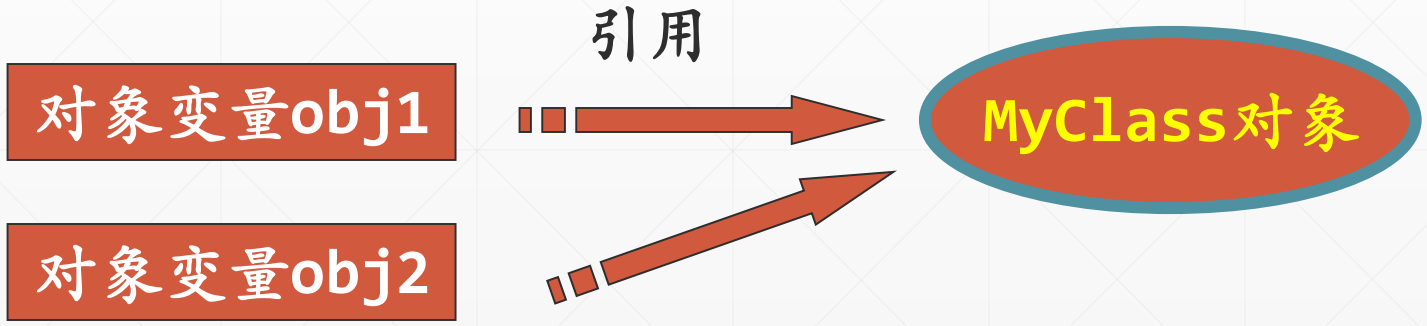
```
MyClass obj1 = new MyClass();
MyClass obj2 = null;
obj2 = obj1;
```

上述代码执行之后，obj1 和 obj2 引用两个不同的对象吗?

对象变量“相互赋值”的真实含义

```
MyClass obj1 = new MyClass();
MyClass obj2 = null;
obj2 = obj1;
```





对象变量引发的故事

equals与 “==”
this

聊聊“Equals”和“==”

请看一个有趣的示例：EqualsDemo

1

```
int i = 100;
int j = 100;
Console.WriteLine(i==j);
//int类型居然有equals方法？
Console.WriteLine(i.Equals(j));
```

2

```
MyClass obj1 = new MyClass();
MyClass obj2 = new MyClass();
Console.WriteLine(obj1==obj2);
Console.WriteLine(obj1.Equals(obj2));
```

3

```
//String类型是引用类型还是值类型？
String str1 = "Hello";
String str2 = "Hello";
Console.WriteLine(str1==str2);
Console.WriteLine(str1.Equals(str2));
```

从上面这三段代码的输出，我们能得到什么样的结论呢？

重要结论：

1. 当“==”运算符施加于两个值类型变量时，实际上是比对两个变量的内容（值）是否一样。
2. 当“==”运算符施加于两个引用类型变量时，实际上是比对这两个变量是否引用同一对象！
3. 要“按值比较”对象，需重写其Equals()和GetHashCode()方法。
4. String是引用类型，但它的“==”经过了重写，其功能与“Equals()”方法一样，都是比较两个字符串的“内容”是否一样。

一个奇特的关键字：this

this是一个特殊的对象变量，它引用“对象自己”。

“this”之与“对象”，就如“某人”自称“我”一样，用“我”来代表“自己”，比如张三会说：“我今天很高兴”，但张三不会说：“张三今天很高兴”，真要这么说，别人一定会觉得你比较奇怪

关于this，你需要知道……

类的实例方法可以直接访问同一个类的实例字段，其中隐藏着一个this引用

示例： ButtonCounterInSingleForm

```
//定义一个内部实例字段
private int counter = 0;
//Click事件响应代码是类的实例方法
1 个引用
private void btnClickMe_Click(object sender, EventArgs e)
{
    //实例方法内部可以直接访问实例字段
    counter++;
    //上面这句其实相当于：this.counter++;

    //标签实际上也是类的实例字段
    lblCount.Text = counter.ToString();
    //上面这句其实相当于：this.lblCount.Text=this.counter.ToString();
}
```

“this”小结：

C#中的**this**，是一个特殊的对象引用，它代表对象自身。

this = **me（我）**

位于同一类内部的成员彼此访问，本质上是通过**this**这一特殊引用来完成的。只不过这个关键字通常被省略了。

结论：

通过对象变量来访问对象的实例成员，是面向对象编程的一个基本准则。

对象的“装箱”与“拆箱”

C#是一种“强类型”的编程语言

（1）强类型的编程语言，要求变量“先定义后使用”，并且变量要拥有明确的特定的类型。

（2）特定类型的变量，只能接收特定类型的值。

做一个试验……

如果我们干些奇怪的事，把**值类型**数值赋给**引用类型**变量，像这样：

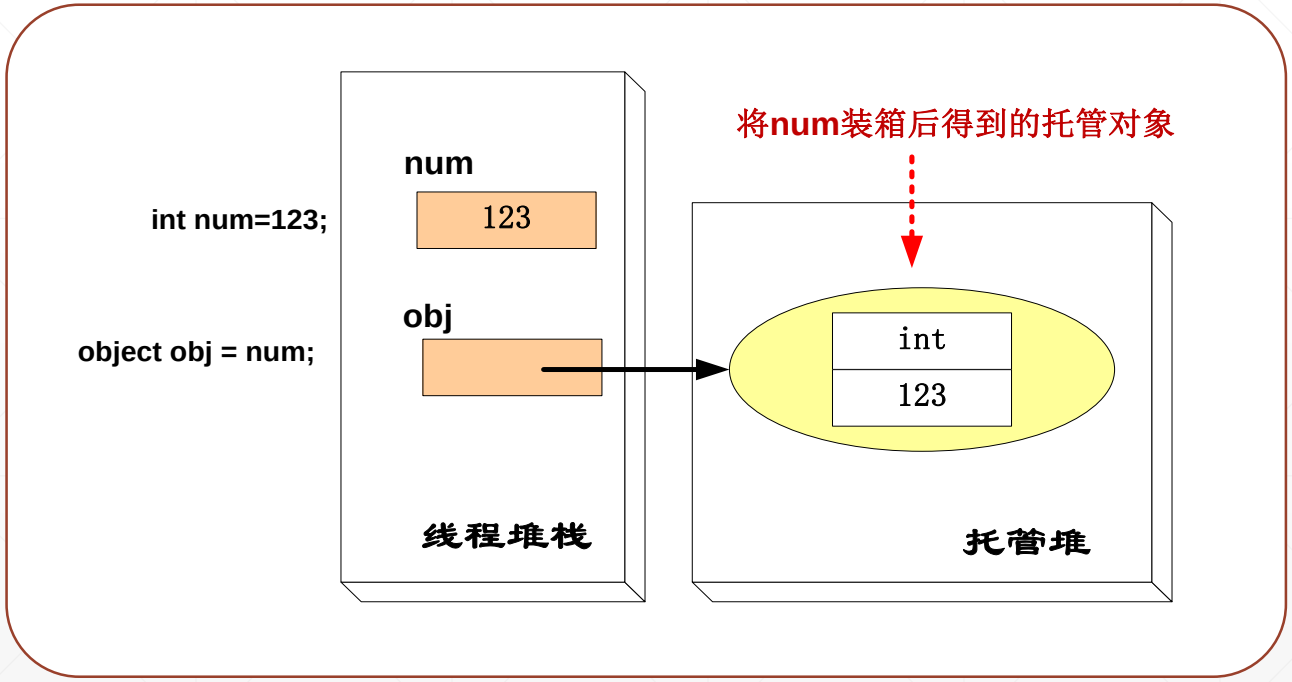
```
int num = 123;
object obj = num;
```

这样的代码能编译通过吗？如果能顺利编译，在程序运行时，又会发生什么呢？

一切皆对象！

```
int num = 123;
object obj = num;
```

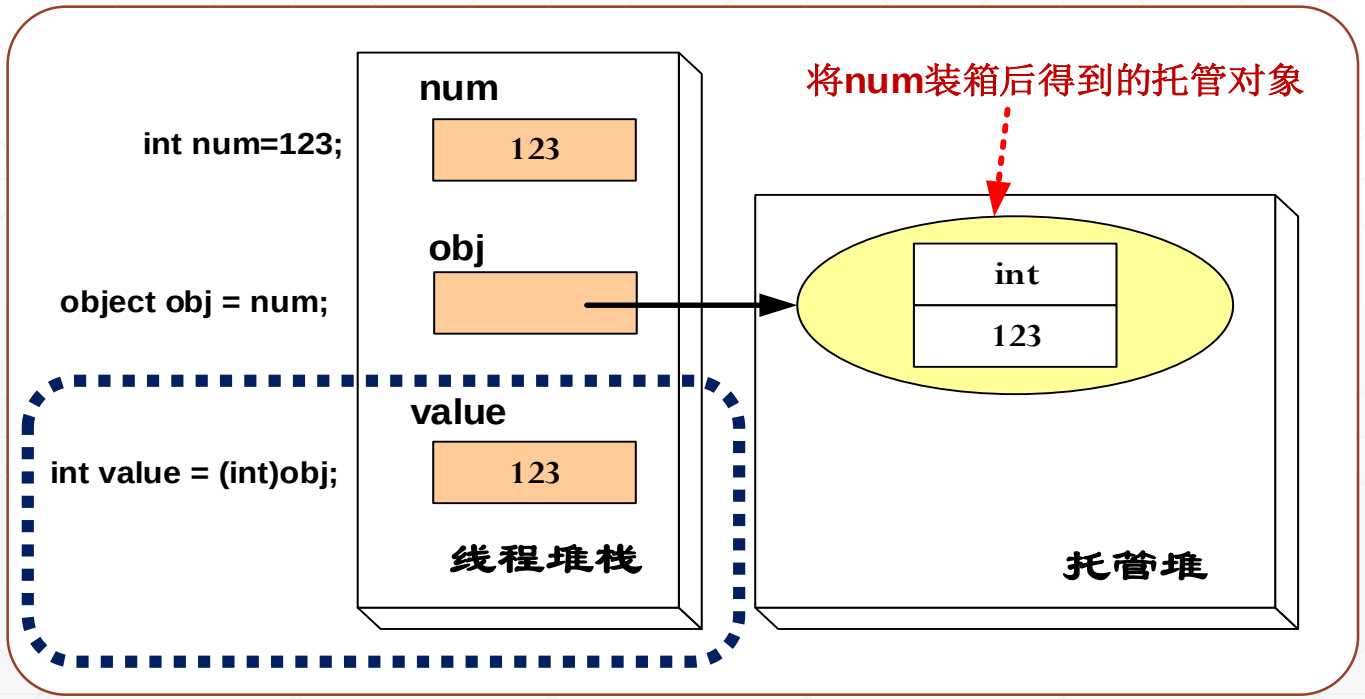
装箱（boxing）



再倒过来，对象赋值给值类型变量

```
int num = 123;
object obj = num;
int value = (int)obj;
```

拆箱（unboxing）



代码“擂台赛”

```
long resultLong = 0;

for(int i = 1; i <= 1000000; i++)
{
        resultLong += i;
}
```

VS.

```
object resultObj = 0;

for (int i = 1; i <= 1000000; i++)
{
    resultObj = Convert.ToInt64(resultObj) + i;
}
```

测试结果：

使用long变量求和，计算结果为500000500000，
花费时间3毫秒。
使用object变量求和，计算结果为500000500000，
花费时间16毫秒。

一段多么“愚蠢”的
代码啊！

示例项目：HowSlowlyIsBoxing

小心陷阱！

在实际开发中，装箱与拆箱会带来较大的性能损失，一定要尽量减少这种情况的发生。

一个基本的编程规范就是：

始终为变量定义明确的类型，并且只把对应类型的数据赋给它。

方法的参数

两种类型的方法参数

值类型参数

```
static void ModifyValue ( int value)
{
    value *= 2;
}
```

引用类型参数

```
static void ModifyValue ( MyClass obj)
{
    obj.Value *= 2;
}
```

方法参数示例

用于测试的类：

```
3 references
class MyClass
{
    public int Value = 100;
}
```

两个测试方法：

```
//参数类型：值类型
1 reference
static void ModifyValue(int value)
{
    value *= 2;
}
//参数类型：引用类型
1 reference
static void ModifyValue(MyClass obj)
{
    obj.Value *= 2;
}
```

测试代码及输出：

```
static void Main(string[] args)
{
    MyClass obj = new MyClass();
    Console.WriteLine(obj.Value); //输出：100

    ModifyValue(obj.Value);
    Console.WriteLine(obj.Value);  //输出：100

    ModifyValue(obj);
    Console.WriteLine(obj.Value);  //输出：200

}
```

这个实例很清楚地展示出了两种类型方法参数的不同特性。

两种类型的方法参数的不同作用

值类型参数

实参值被复制一份，方法内容对参数的修改，不会影响到原始值

引用类型参数

对象引用被复制一份，方法内部通过引用访问的对象，与方法外部的对象，是同一个！

“只读”的对象

多学一点

通常情况下，对象的字段值是可以修改的.....

```
2 个引用
class MyClass
{
    private int Value = 100;
    1 个引用
    public void add(int step)
    {
        this.Value += step;
    }
    1 个引用
    public void printValue()
    {
        Console.WriteLine("Value={0}", Value);
    }
}
```

```
MyClass obj = new MyClass();
obj.add(1);  //字段值加一
obj.printValue();  //输出:101
```

示例： ReadOnlyType

但是，在.NET基类库中，我们发现……

值类型DateTime好象不一样……

```
DateTime date = new DateTime(2015, 10, 1);
date.AddDays(1);  //增加一天
Console.WriteLine(date);  //输出：2015/10/1
```

字符串类型好象也不一样……

```
string str = "abcd";
str.ToUpper();  //改为大写
Console.WriteLine(str);  //输出：abcd
```

DateTime和string类型变量居然都是**只读**的！一旦创建之后，内容不可改！

为什么要设计“只读”的类？

现在的计算机和手机、平板，都是**多核**的。

为发挥多核CPU的计算能力，应用程序应该是“**多线程**”的

在“多线程”环境下，多个线程访问同一个对象时，因为对象是只读的，无需互斥（一次只允许一个访问，一个在访问时，其他等待），就可以保证数据读取不会出错（想一想，如果不是只读的，一个线程正在读，另一个线程正在写，一切就乱套了）。

所以，在“多线程”环境下，使用只读对象可以提升程序的性能。

怎样设计“只读”的类？

当外界期望修改对象的字段值时，不是修改原有对象的字段值，而是**新建**一个对象，让它的字段值符合要求，然后把这个新对象返回给外界！

```
6 references
class MyReadOnlyClass
{
    private int Value = 100;

    1 reference
    public MyReadOnlyClass add(int step)
    {
        MyReadOnlyClass obj = new MyReadOnlyClass();
        obj.Value = this.Value;
        obj.Value += step;
        return obj;
    }

    2 references
    public void printValue()
    {
        Console.WriteLine("Value={0}", Value);
    }
}
```

只读类的使用示例：

```
MyReadOnlyClass readonlyObj = new MyReadOnlyClass();
MyReadOnlyClass readonlyObj2 = readonlyObj.add(1);
Console.WriteLine(readonlyObj2 == readonlyObj); //false
readonlyObj.printValue(); //输出：100
readonlyObj2.printValue(); //输出：101
```

类的静态成员

典型开发场景

在实际开发中，我们可能会有一些“**到处都要使用**”的功能需要实现，比如各种标准的数学函数以及圆周率等数学常量，在C#中如何定义并实现它们?

.NET基类库中Math类所封装的部分数学常量与函数

```
public static class Math
{
    public const double E = 2.7182818284590451;
    public const double PI = 3.1415926535897931;

    public static short Abs(short value);
    public static decimal Abs(decimal value);
    public static double Abs(double value);
    public static float Abs(float value);
    public static long Abs(long value);
    public static int Abs(int value);
    public static sbyte Abs(sbyte value);
    public static double Acos(double d);
    public static double Asin(double d);
    public static double Atan(double d);
    public static double Atan2(double y, double x);
    public static long BigMul(int a, int b);
    public static double Ceiling(double a);
```

使用const定义数学常量
使用static定义数学函数

应用实例

```
//计算90度的余弦值
int angle = 90;
//转为弧度
double radian = angle * Math.PI / 180.0;
//调用数学库函数进行计算
Console.WriteLine(Math.Cos(radian));
```

使用static关键字来定义类的静态成员

```
// 静态类
6 个引用
static class StaticClass
{
    // 公有静态字段
    public static int staticField = 100;
    // 公有静态属性（同时初始化为“Hello”，适用于C# 6）
    2 个引用
    public static string staticProp
    {
        get; set;
    } = "Hello";

    // 静态方法
    1 个引用
    public static void staticFunc()
    {
        Console.WriteLine("调用静态方法");
    }

    // 私有静态字段，不能被外界访问
    // 初始化主要通过静态构造方法或直接赋初值。
    private static int SecretField = 0;

    // 静态构造方法
    0 个引用
    static StaticClass()
    {
        SecretField = 100;
    }
}
```

```
0 个引用
static void Main(string[] args)
{
    //静态类型无法创建对象
    //var obj = new StaticClass();

    StaticClass.staticField += 100;
    Console.WriteLine("staticField={0}",StaticClass.staticField);
    //访问静态属性
    StaticClass.staticProp +=" World";
    Console.WriteLine("staticProp={0}",StaticClass.staticProp);
    //访问静态方法
    StaticClass.staticFunc();
    Console.ReadKey();
}
```

使用“**类名.成员名**”来访问类的静态成员。

示例StaticMember

两种类型的类成员

静态（static）方法/字段/属性　　使用static定义

实例（instance）方法/字段/属性　　不使用static定义

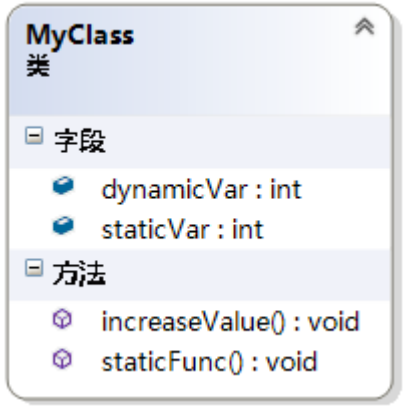
静态字段 vs. 实例字段

```
5 个引用
class MyClass
{
    //静态字段
    public static int staticVar=0;
    //静态方法
    0 个引用
    public static void staticFunc()
    {
        MyClass obj = new MyClass();
        obj.increaseValue();
        obj.dynamicVar++;
    }
    //实例字段
    public int dynamicVar=0;
    //实例方法
    2 个引用
    public void increaseValue()
    {
        staticVar++;
        dynamicVar++;
    }
}
```

```
0 个引用
class Program
{
    0 个引用
    static void Main(string[] args)
    {
        StaticMembers obj = null;
        //创建100个对象
        for (int i = 0; i < 100; i++)
        {
            obj = new StaticMembers();
            obj.increaseValue();
        }
        //查看静态字段与实例字段的值
        Console.WriteLine("dynamicVar=" + obj.dynamicVar);
        Console.WriteLine("staticVar=" + StaticMembers.staticVar);
        //程序暂停，敲任意键继续
        Console.ReadKey();
    }
}
```

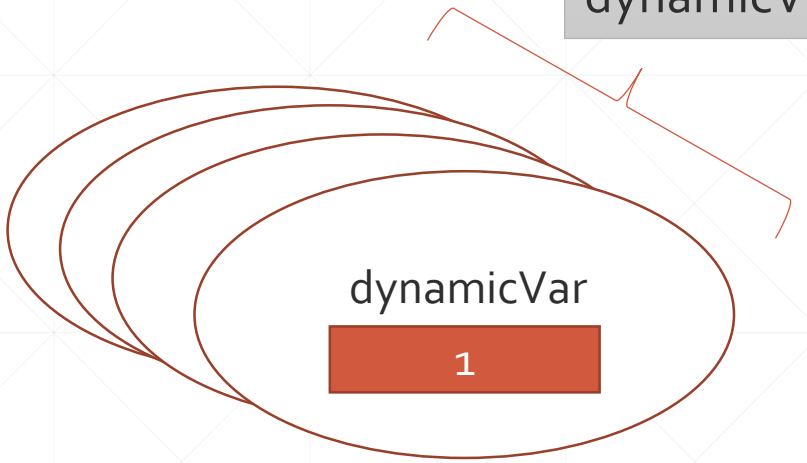
示例StaticAndInstanceField

StaticAndInstanceField实例分析




```
MyClass obj = null;
//创建100个对象
for (int i = 0; i < 100; i++)
{
    obj = new MyClass();
    obj.increaseValue();
}
```

创建了100个MyClass对象，每个对象都拥有一个独立的dynamicVar字段



创建了100个MyClass对象，共享同一个staticVar字段

- 类的实例成员只能通过对象来访问。每个对象都有一份自己独享的实例成员，是“**个人财产**”。
- 类的静态成员归所有对象所共享，是“**国有资产**”。

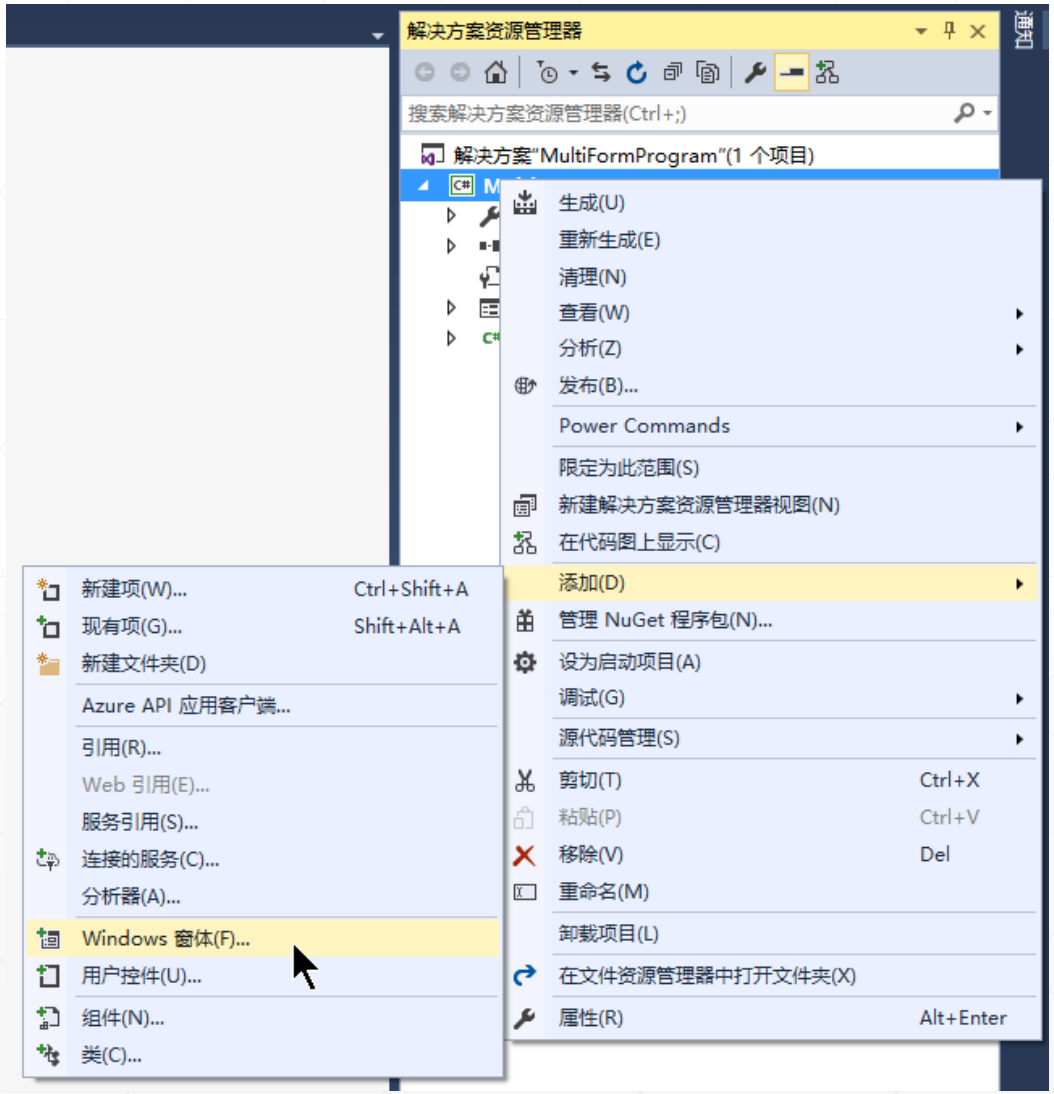
类成员的访问规则

- 类的**实例方法**可以访问类的**实例字段**；
- 类的**实例方法**可以访问类的**静态成员**；
- 类的**静态方法**只能访问类的**静态字段**。

请自己编写一些测试代码，验证以上这些特性

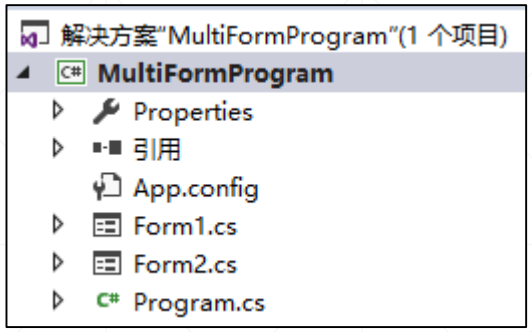
扩充：多窗体编程初步

向项目中添加新窗体



在项目节点上右击，可以向Windows窗体应用项目中添加新的窗体。

如何设定启动窗体？



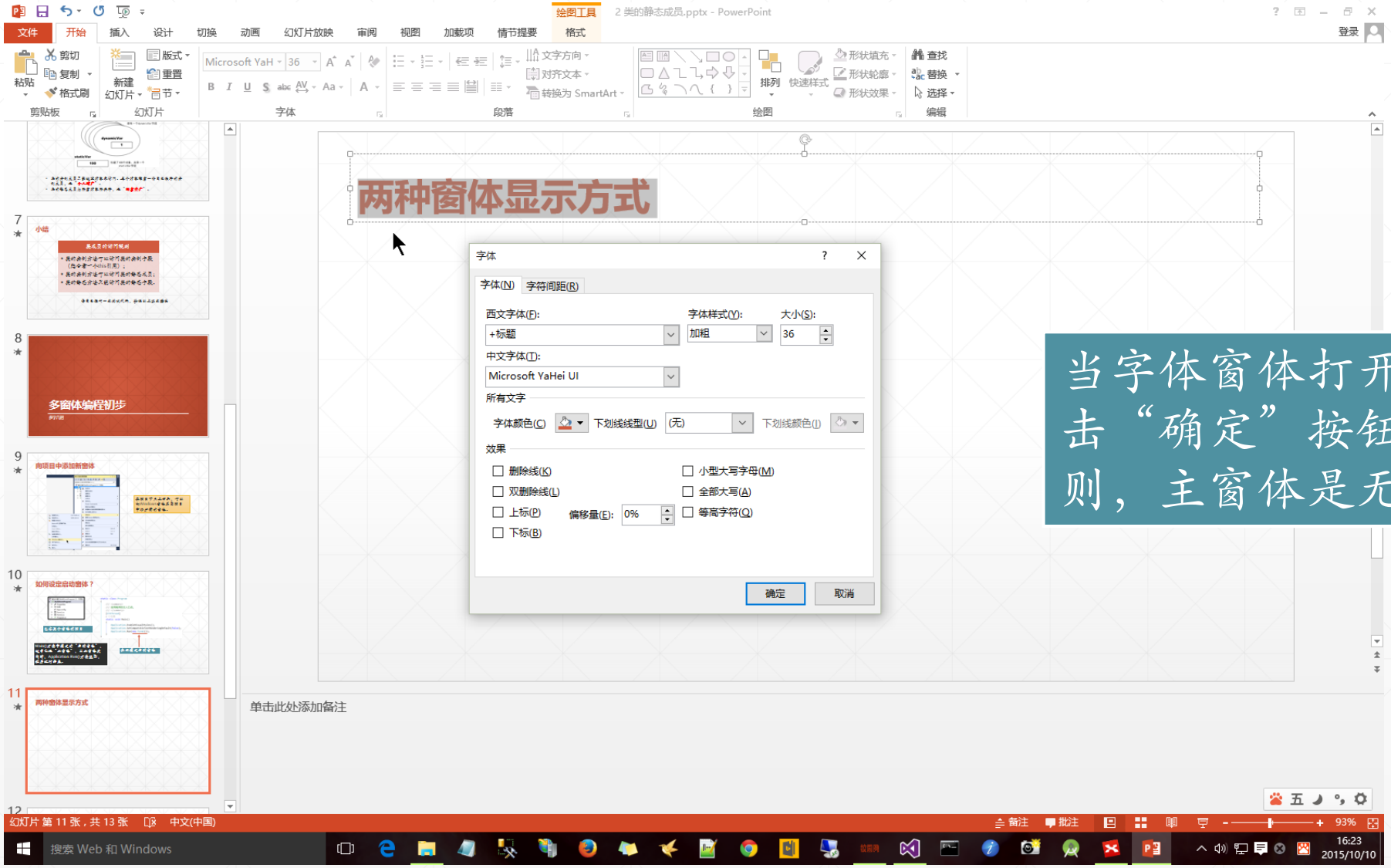
包容两个窗体的项目

```
static class Program
{
    /// <summary>
    /// 应用程序的主入口点。
    /// </summary>
    [STAThread]
    0 个引用
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}
```

在此指定启动窗体

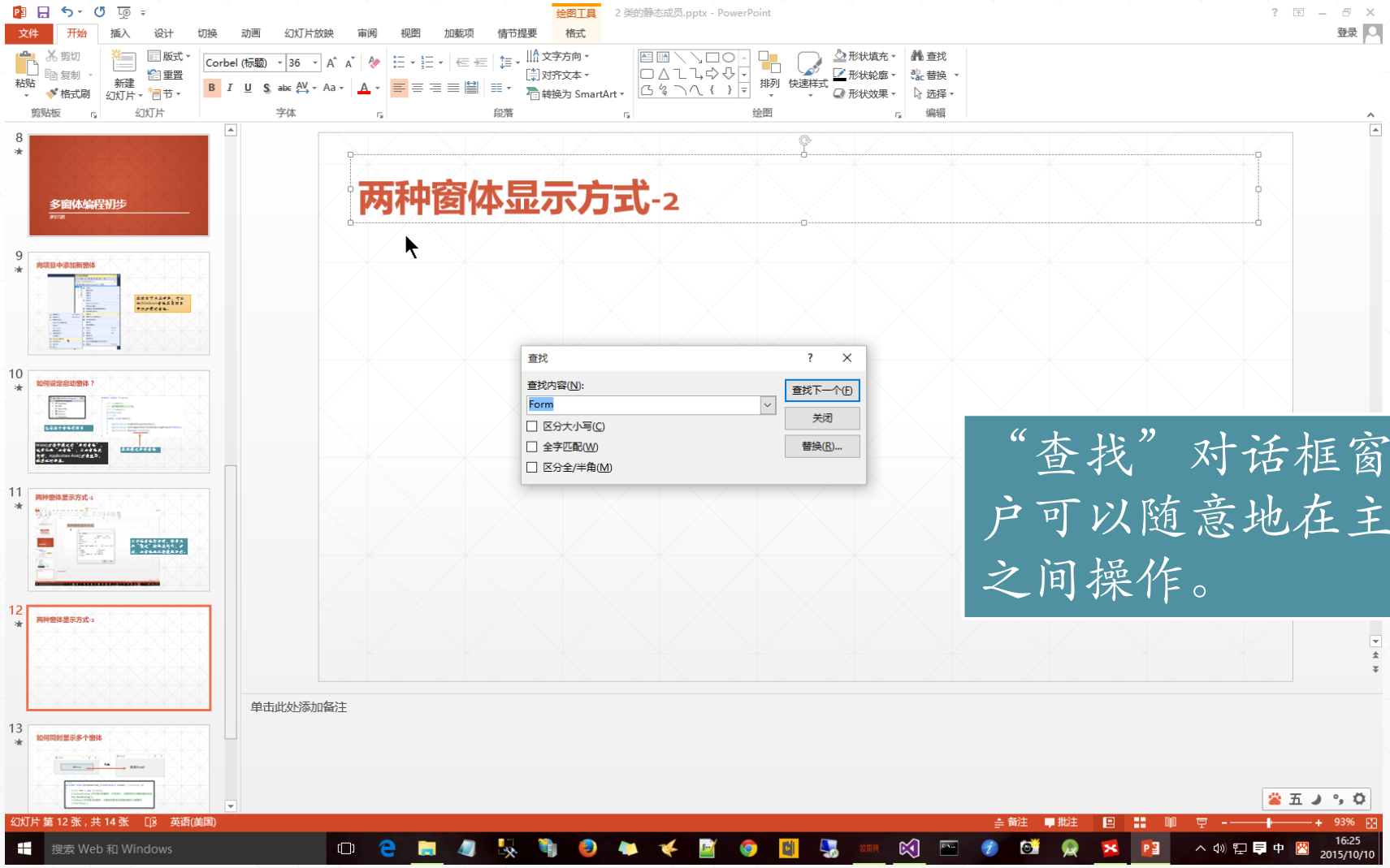
Main()方法中指定的“启动窗体”，通常也是“主窗体”，当主窗体关闭时，Application.Run()方法返回，程序运行结束。

两种窗体显示方式-1



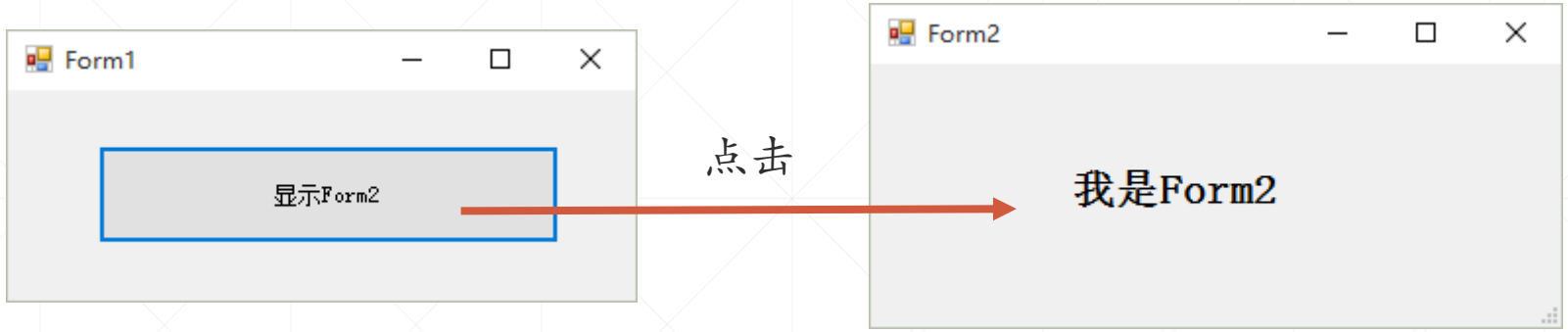
当字体窗体打开时，除非点击“确定”按钮关闭它，否则，主窗体是无法被激活的。

两种窗体显示方式-2



“查找”对话框窗体显示时，用户可以随意地在主窗体和从窗体之间操作。

编程以两种方式显示新窗体




```
1 个引用
private void btnShowForm2_Click(object sender, EventArgs e)
{
    Form2 frm = new Form2();
    //以showDialog()方式显示的窗体，不关闭它，主窗体将无法响应鼠标点击
    frm.ShowDialog();
    //以Show()方法显示的窗体，主窗体和新显示的窗体都可以被激活
    //frm.Show();
}
```