类和对象

北京理工大学计算机学院
金旭亮

主要内容

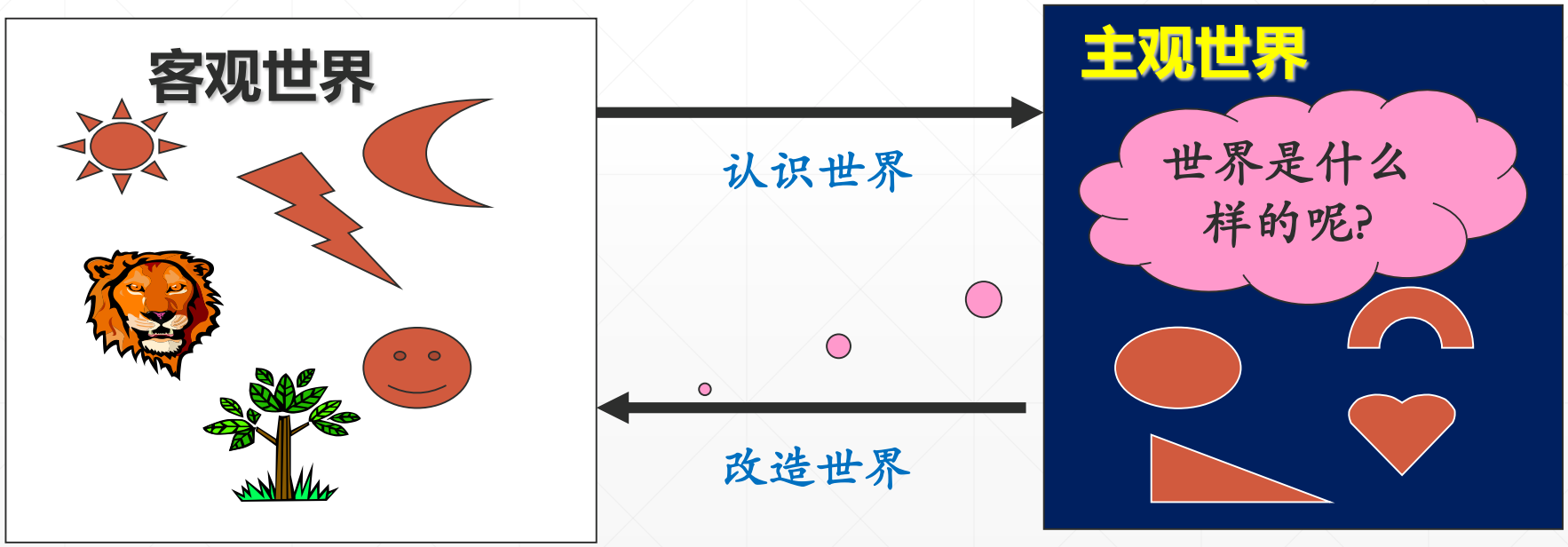
- 理解类和对象的概念
- 学会编写类
- 构造方法
- 分部类与分部方法
- 命名空间与类库

理解类和对象的概念

软件系统与真实的世界

软件系统其实是现实世界的一种模拟，可以看成是一个“**模型**”。

构建一个软件系统的活动，我们把它称之为“**建模**”。



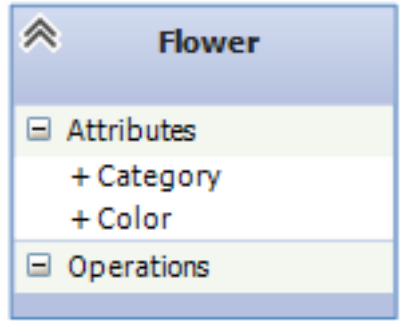
以面向对象的观点看世界

真实的花——**对象**

分析、
归纳和
总结

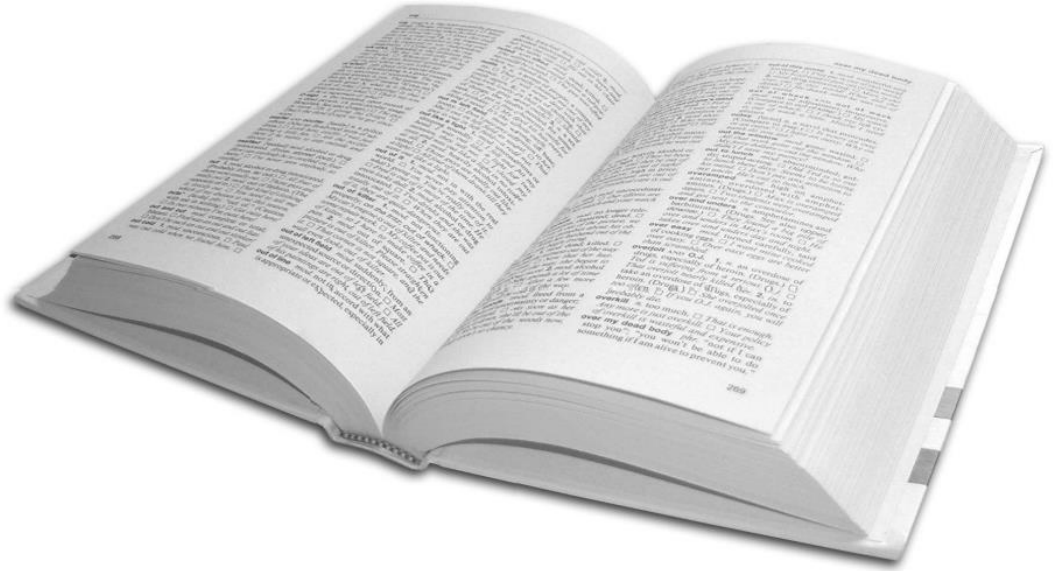
抽象出来的“花”——**类**

用编程语
言实现



软件中的Flower**类**

软件开发中，类的构建主要靠“抽象”



抽象

类

书
-作者 -出版社 -页数 -出版时间 -定价

“抽象”的具体实施方法：

提取出事物的本质特性，忽略所有不相关的不重要的信息，以“类”的形式表达出来

盲人摸象

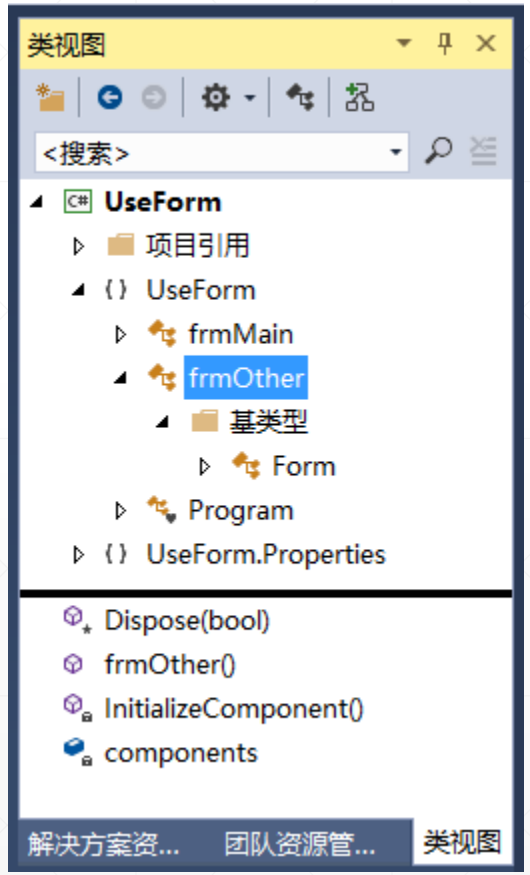
观察的角度不同，目的不同，对同一种事物抽象，会得到不同的抽象模型。

程序中的窗体与项目中的类

示例UseForm很形象地说明了软件中“类”和“对象”的差别。

窗体

窗体与类之间存在着对应关系



UseForm示例项目中的类

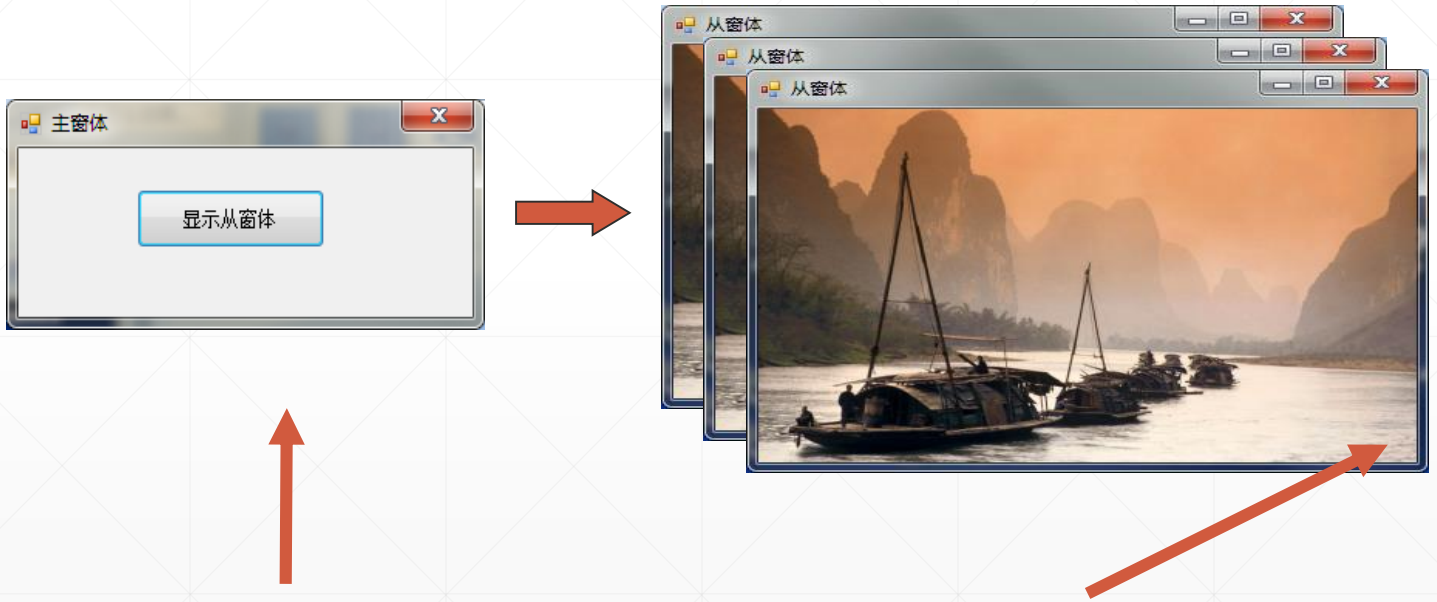
“窗体类和窗体对象”


```
private void btnShowOtherForm_Click(object sender, EventArgs e)
{
    //创建从窗体对象
    frmOther frm = new frmOther();
    //显示从窗体
    frm.Show();
}
```

1. 屏幕上看到的每个窗体，实际上都是特定窗体类创建出来的对象。
2. 主窗体是frmMain类创建的对象，从窗体是frmOther类创建的对象。
3. C#中要创建对象，使用new关键字。

对象也分“高低贵贱”……

“主窗体”和“从窗体”都是窗体，但“地位”不一样。



主窗体一关闭，整个程序结束

从窗体关闭，整个程序仍然会继续运行

从示例中我们知道，C#中.....

(1) 用new关键字创建“类的实例（即对象）”。

```
frmOther frm = new frmOther();
```

(2) 用“类”类型的变量（称为“对象变量”）来保存对创建出来的对象的引用。

```
frmOther frm = new frmOther();
```

(3) 通过对象变量来访问对象的公有成员（包括方法与字段）。

```
frm.Show();
```

UseForm中调用窗体对象的Show()方法显示窗体

```
public void Show();
```

Show()方法由Form的基类Control所定义

打破砂锅问到底！

类和对象到底是什么？！

类是印章，对象是印章沾上印泥后盖出的印！

蒙阴宰之印（真）北京故宫博物院藏

为什么这一切如此重要？

深刻理解类和对象这两个概念，是面向对象编程的基础，在此基础之上，才可以介绍更多的面向对象编程知识。

学会编写类

“类”放在哪儿？

所有代码放在类中，类是编程的基本单元。

类的定义方法

C#使用**class**关键字定义一个类。类中常见的成员有：

类的成员
- **字段（field）**：即变量
- **方法（method）**：即函数

C#中的字段与方法，可以加上“**public**、**private**、**protected**”关键字控制其存取权限。

类的定义与使用实例

类的定义

```
2 个引用
public class MyClass
{
    public int i = 0;
    private int j = 0;
    1 个引用
    public void PrintValue()
    {
        //在方法中定义的变量j，将使定义在类中的同名字段j“失效”
        int j = 1;
        Console.WriteLine("i={0},j={1}" , i,j);
        Console.ReadKey();
    }
}
```

字段

方法

类的使用

```
MyClass obj = new MyClass();   //创建对象

obj.i = 100;      //通过对象变量访问公有字段

obj.j = 200;   //错！ 不允许直接访问类的私有成员

obj.PrintValue();  //通过对象变量访问公有方法
```

示例： DefAndUseClass

我们现在知道了……

（1）类中的方法，可以直接访问类中的字段。

（2）类中的方法定义的局部变量，将屏蔽掉类中的同名字段。

（3）有两种最基本的数据存取权限：

1. **public（公有）**：通过对象变量外界可以直接访问它
2. **private（私有）**：除了类内部的方法，外界无法直接访问它们

在设计一个类时，仅有需要被外界访问的成员才设置为public的。

类的“属性（property）”

属性的经典实现方法

定义

```
public class MyTestClass
{
    //私有变量用于存储数据
    private string _myprop = "";
    public string Myprop
    {
        get  //读访问器
        {
            return _myprop;
        }
        set  //写访问器
        {
            _myprop = value;
        }
    }
}
```

value是一个拥有特殊含义的关键字，在此处，它代表外界传入的值。

使用

```
MyTestClass obj = new MyTestClass();
//向属性赋值
obj.Myprop = "Hello";
//读取属性值
Console.WriteLine(obj.Myprop);
```

实现自定义属性的要点：

（1）定义一个**私有**字段用于存储属性数据。

（2）设计一个**get**方法，当读取属性值时，向外界返回私有字段的当前值。

（3）设计一个**set**方法，当向属性赋值时，其自动隐含的value参数保存外界传入的值，应将此值传给前面定义的私有字段。

“属性”经典实现方法的弊端

```
public class MyTestClass
{
    //私有变量用于存储数据
    private string _myprop = "";
    public string Myprop
    {
        get  //读访问器
        {
            return _myprop;
        }
        set  //写访问器
        {
            _myprop = value;
        }
    }
}
```

存在的问题：

当一个类中存在有很多的属性，而这些属性又采用类似的方法编写时，这是一个烦人的工作，要敲很多的代码！

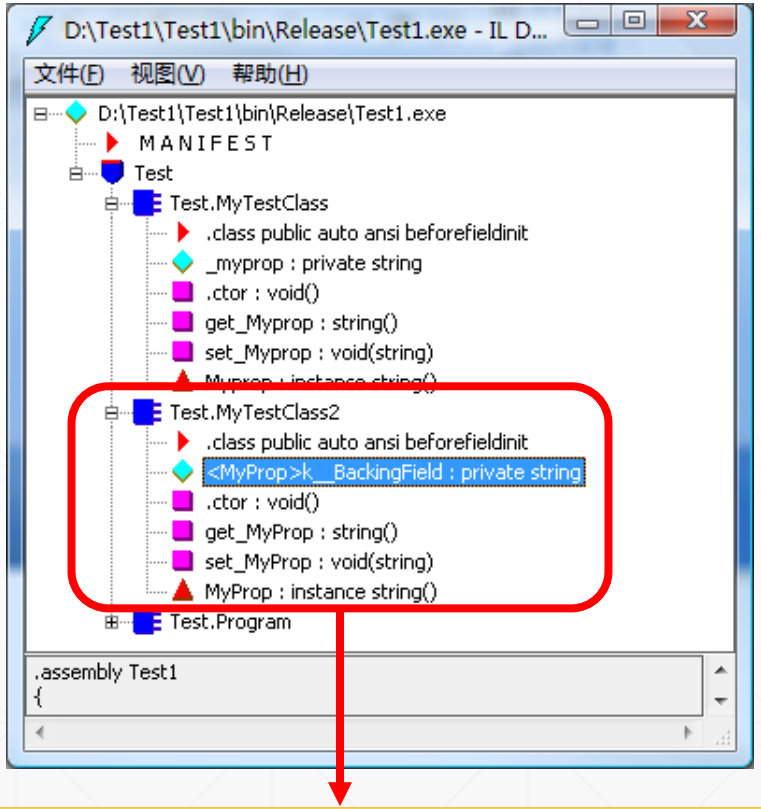
能不能想办法偷一些懒？……

C# 3.0的改进——自动实现的属性

C#编译器偷偷“搞了什么鬼”？

```
public class MyTestClass2
 {
     public string MyProp
     {
         get;
         set;
     }
}
```

编译



编译器会自动添加一个私有字段，并按照经典方法生成属性的读/写访问器。

“属性”比“字段”多了哪些东西？

```
public class Person
{
        private int _age = 0;
        public int Age
        {
            get
            {
                return _age;
            }
            set
            {
                if (value < 0)
                    _age = 0;
                else
                    _age = value;
            }
        }
}
```

当使用经典写法时，我们可以很容易地在读/写属性时“插入一些”特定的代码，完成诸如“检验数据”、“显示信息”、“触发事件”等工作，这是单纯的字段所不具备的特性。

人的年龄只能是一个正数，因此，当外界传入一个**负数**时，set方法会自动将其改为**0**。

“属性”的各种“玩法……

定义一个属性时，get/set方法并不需要同时存在，它们的存取权限也可以不一样，各种情况的不同组合，将影响到属性的存取特性……

```
/// <summary>
/// 只读属性
/// </summary>
0 个引用
public string ReadOnlyProp
{
    get;
}
```

```
/// <summary>
/// 只写属性
/// </summary>
private string _value;
0 个引用
public string WriteOnlyProp
{
    set
    {
        _value = value;
    }
}
```

```
/// <summary>
/// 外界可读，内部可写属性
/// </summary>
0 个引用
public string PublicReadPrivateSetProp
{
    get;
    private set;
}
```

```
/// <summary>
/// 直接给定初始值的属性（适用于C# 6.0）
/// </summary>
0 个引用
public string AutoInitProp
{
    get;
} = "Hello";
```

开发建议

在实际开发中，多用属性少用字段，尤其是杜绝“**公有（public）**”字段，过多的公有字段，可能会损害代码的可维护性。

构造方法

类中的一种特殊方法

当我们通过new关键字创建一个对象时，一个特殊的函数被调用，此函数被称为——**构造函数（构造方法）**。

```
MyClass obj = new MyClass();
```

所谓“构造方法”，就是在创建对象时被**自动调用**的方法。

构造方法长得什么样？

构造方法与类名相同，没有返回值。

```
public class MyClass
{
    public MyClass()
    {
        Console.WriteLine("无参构造方法被调用");
    }
    public MyClass(string info)
    {
        Console.WriteLine("调用MyClass(string):" + info);
    }
}
```

构造方法的重载

一个类可以有多个构造方法，这些构造方法构成“**重载（overload）**”关系。在程序实际运行时，依据参数决定调用哪个构造方法。

```
class Program
{
        static void Main(string[] args)
        {
            //调用无参构造方法
            MyClass obj = new MyClass();

            //调用有一个字符串参数的构造方法
            obj = new MyClass("Hello");

        }
}
```

为什么要定义一个构造方法？

构造方法主要用于在创建对象时给它的相关字段一个有意义的初始值。

定义一个类时，即使你没有显式地定义一个构造方法，C#编译器也会“偷偷”地给你的类加上一个没有参数的“**缺省构造方法**”。

字段的初始化方式

简化代码的小窍门

创建对象并给其字段/属性相应初始值的基本方法

请看以下类：

```
class MyClass
{
    public int IntValue;
    public string StrValue
    {
        get;
        set;
    }
}
```

- 以下代码创建MyClass的对象并给其字段或属性赋值：

```
MyClass obj = new MyClass();

obj.IntValue = 100;
obj.StrValue = "Hello";
```

问题：

当一个类定义了多个字段或属性时，代码变得很冗长……

改进方式一：

给类添加构造函数

```
class MyClass
{
    public int IntValue;
    public string StrValue
    {
        get;
        set;
    }
    public MyClass(int iValue, string strValue)
    {
        IntValue = iValue;
        StrValue = StrValue;
    }
}
```

使用代码：

```
MyClass obj = new MyClass ( 100 , "Hello" );
```

存在的问题：

如果我只想给**部分**属性和字段一个有效的初始值，需要提供多个重载的构造函数。这就让类的定义变得复杂了。

改进方式二：

使用C# 3.0所提供的“**对象初始值设定项**”特性，不需要给MyClass添加重载的构造函数即可实现相同目的：

类的原始定义不变

```
class MyClass
{
    public int IntValue;
    public string StrValue
    {
        get;
        set;
    }
}
```

简化后的字段/属性初始化代码

```
//对象字段和属性直接使用其默认值
MyClass obj1 = new MyClass();
```

```
//设定对象所有的字段和属性初始值
MyClass obj2 = new MyClass {
      IntValue = 100,
      StrValue = "Hello"
};
```

```
//设定对象部分的字段和属性初始值
MyClass obj3 = new MyClass {
        IntValue = 100
 };
```

还可以直接初始化集合对象

- 直接初始化基本数据类型的集合对象

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6 };
```

- 初始化包含引用类型元素的集合对象

```
List<MyClass> objs = new List<MyClass>{
                new MyClass{ IntValue=100,StrValue="Hello" },
                new MyClass{ IntValue=200,StrValue="World" }
            };
```

分部类与分部方法

分部类（partial class）

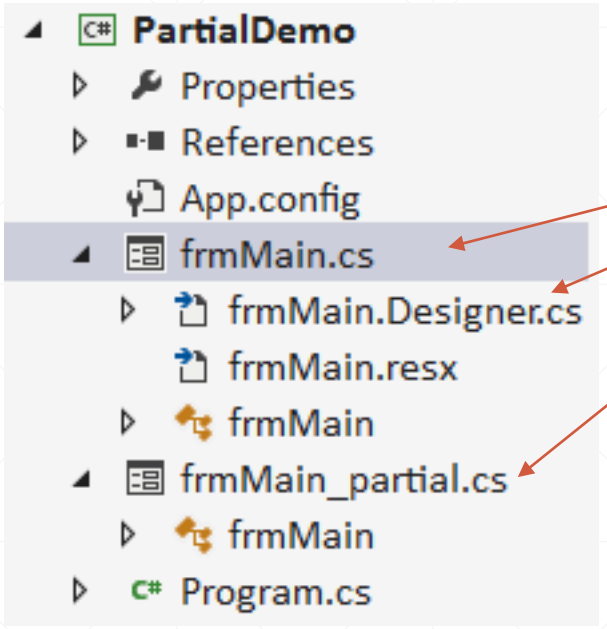
- 把同一个类的代码分散到多个文件中

分部方法（partial method）

- 在一个文件中声明方法，在另一个文件中实现或调用方法

不把鸡蛋放到一个篮子里

示例： PartialDemo



frmMain窗体类的代码，分布于三个文件之中

frmMain.cs

```
public partial class frmMain : Form
{
    1 reference
    public frmMain()...
    //定义一个分部方法
    2 references
    partial void MyPartialMethod();

    1 reference
    private void btnClick_Click(object sender, EventArgs e)
    {
        //调用分部方法
        MyPartialMethod();
    }
}
```

frmMain.Designer.cs

```
partial class frmMain
{
    /// <summary> ...
    private System.ComponentModel.IContainer components = null;

    /// <summary> ...
    1 reference
    protected override void Dispose(bool disposing)...

    Windows Form Designer generated code

    private System.Windows.Forms.Button btnClick;
    private System.Windows.Forms.Label lblInfo;
}
```

frmMain_partial.cs

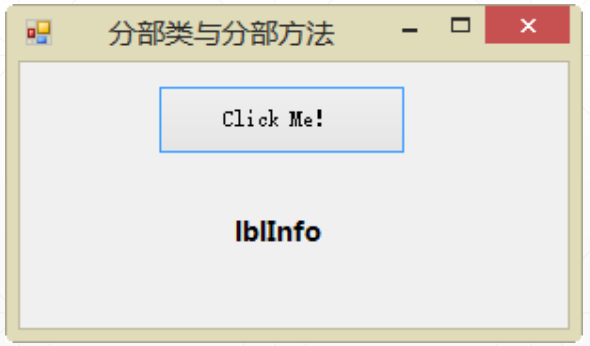
```
//一个分部类，放置分部方法的具体实现代码
//当这里面的分部方法代码被注释掉时，程序仍然能够编译运行
4 references
partial class frmMain
{
    private int count = 0;
    //分部方法的实现
    2 references
    partial void MyPartialMethod()
    {
        count++;
        lblInfo.Text = String.Format("单击了{0}次", count);
    }
}
```

当给分部方法提供实现代码时，这些代码将会被调用

```
partial class frmMain
{
    private int count = 0;
    //分部方法的实现
    2 references
    partial void MyPartialMethod()
    {
        count++;
        lblInfo.Text = String.Format("单击了{0}次", count);
    }

}
```


```
partial class frmMain
{
    private int count = 0;
    //分部方法的实现
    //partial void MyPartialMethod()
    //{
    //    count++;
    //    lblInfo.Text = String.Format("单击了{0}次", count);
    //}
}
```



当没有找到方法的实现代码时，C#编译器会移除所有对此分部方法的调用代码。

分部方法与分部类的应用实例

在.NET中，Windows Forms、WPF、ASP.NET Web Forms，Entity Framework等都使用了分部类将“自动生成”的代码与程序员“手工编写”的代码分隔开来。

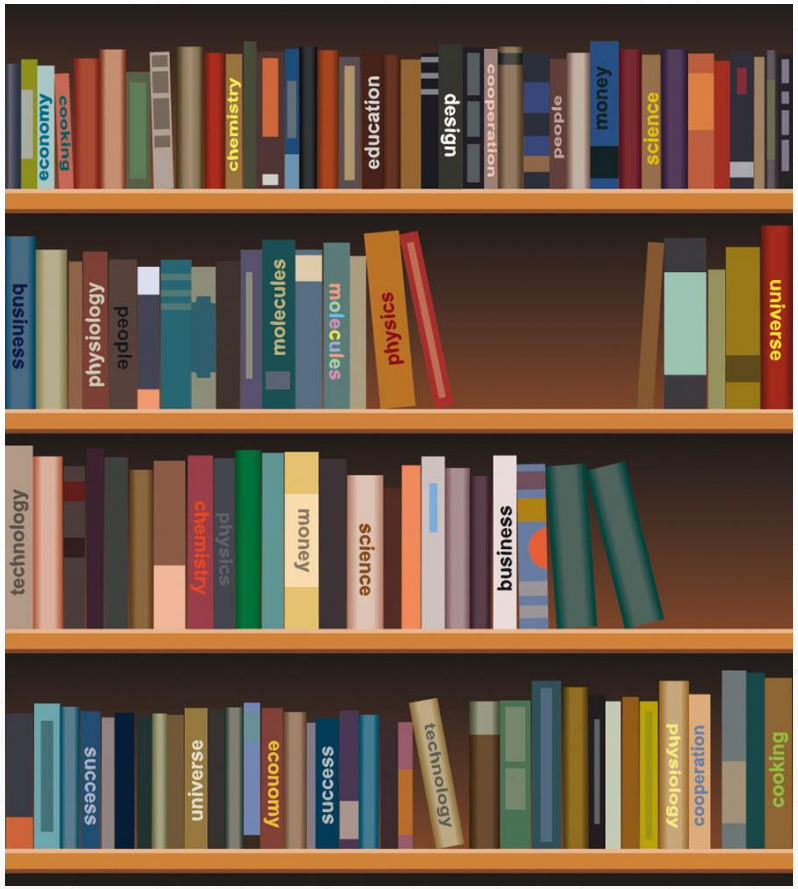
在实际开发中，我们可以使用分部类将不同程序员对同一个类的修改“**隔离**”开来，提升代码的可维护性。

利用分部方法，我们可以预先定义好一些“**扩展点**”（有点类似于在自习室用书本占座），然后在需要这些扩展点发挥作用时，提供其实现代码（人到自习室了），从而在源代码级别让系统易于扩展（无需修改原有代码，即可实现新的特性）。

车多是件麻烦事.....

命名空间

理解“命名空间（namespace）”

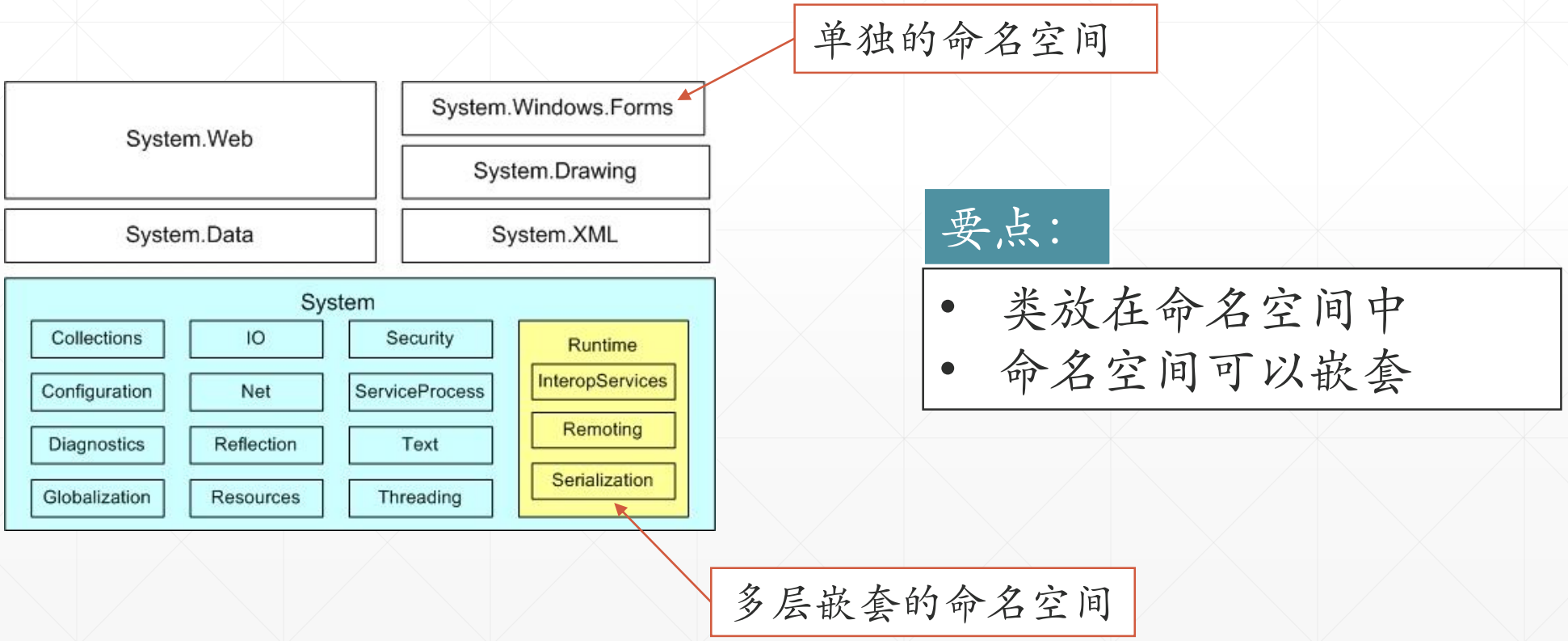


书 <-- --> 类

书架<-- --> 命名空间

面向对象编程语言中的“命名空间”……

命名空间（namespace）：可以看成是**类的仓库**，.NET中所有的功能都由类提供，这些类被分门别类地存放在特定的命名空间中。



要点：

- 类放在命名空间中
- 命名空间可以嵌套

在C#中创建和使用命名空间

C#中使用**namespace**关键字来定义命名空间

```
/// <summary>
/// 自定义命名空间
/// </summary>
namespace MyNameSpace
{
    4 个引用
    public class MyClass
    {
    }
    /// <summary>
    /// 子命名空间
    /// </summary>
    namespace InnerNamespace
    {
        0 个引用
        public class MyInnerClass
        {

        }
    }
}
```

使用**using**语句来引用命名空间的类

```
//引入MyClass类所在的命名空间
using MyNameSpace;
//引入子命名空间
using MyNameSpace.InnerNamespace;
```

```
static void Main(string[] args)
{
    //创建MyNameSpace命名空间中类的实例
    MyClass obj = new MyClass();

    //也可以直接指定完整的命名空间路径字串，从而不需要使用using语句
    MyNameSpace.MyClass obj2 = new MyNameSpace.MyClass();

    //创建子命名空间中的类的实例
    MyInnerClass inner = new MyInnerClass();
}
```

示例： UseNamespace

程序集

什么是“程序集”？

程序集<---->砖

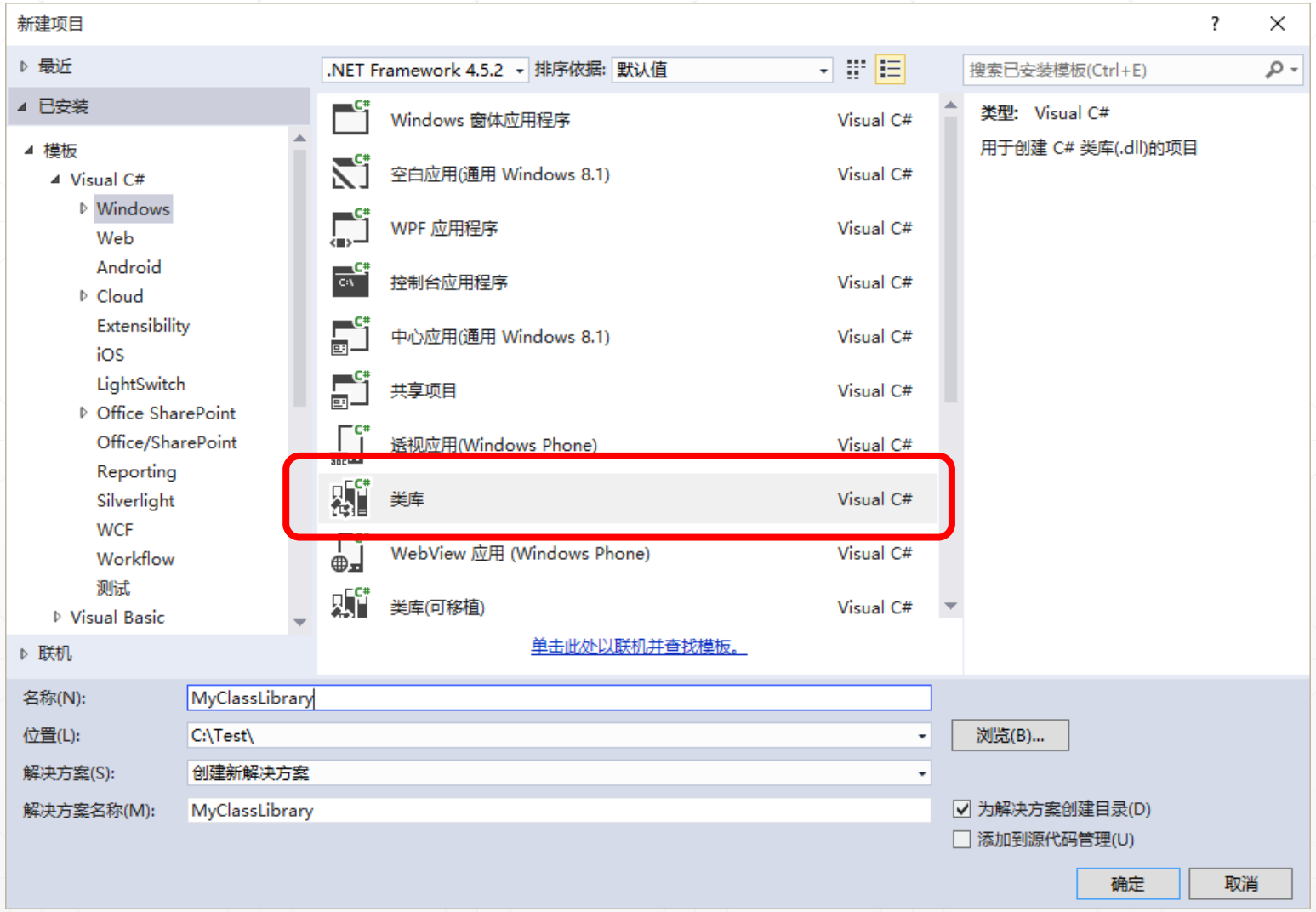
应用程序<---->建筑物

关于“程序集”应该知道的……

1. .NET程序的基本构造块是“程序集（Assembly）”。
2. 程序集是一个扩展名为.dll或.exe的文件。
3. .NET Framework中的各个类，存放在相应的程序集文件中。

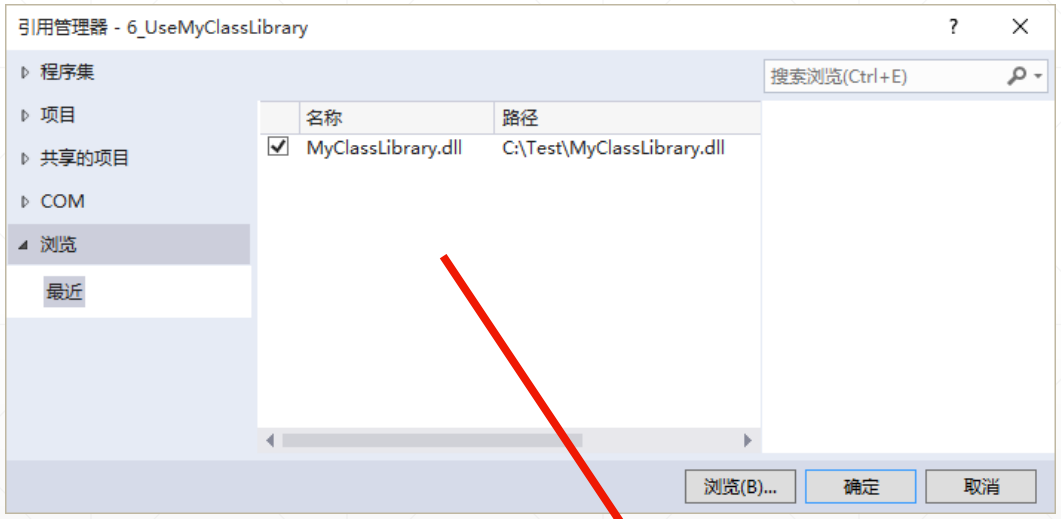
如何创建一个程序集？

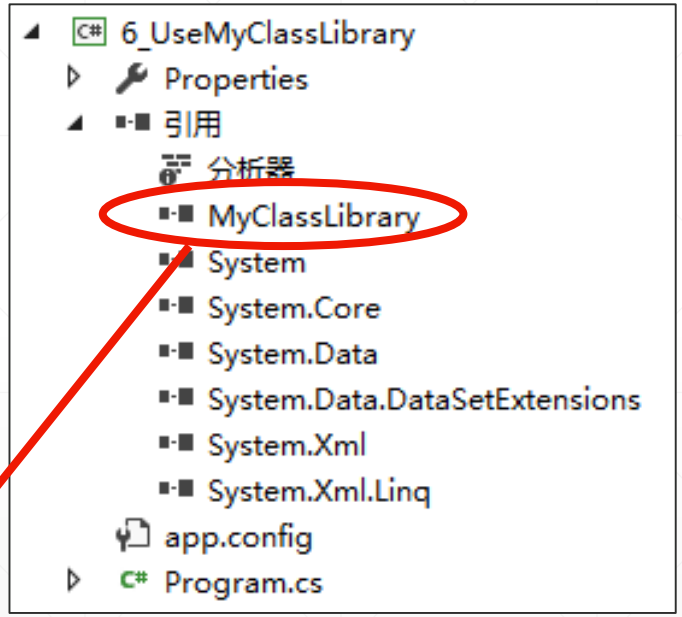
“类库（class library）”项目模板可以用于创建一个DLL程序集。



使用程序集

在一个新项目中添加对于特定程序集的“**引用（Reference）**”，即可使用此程序集中的类：

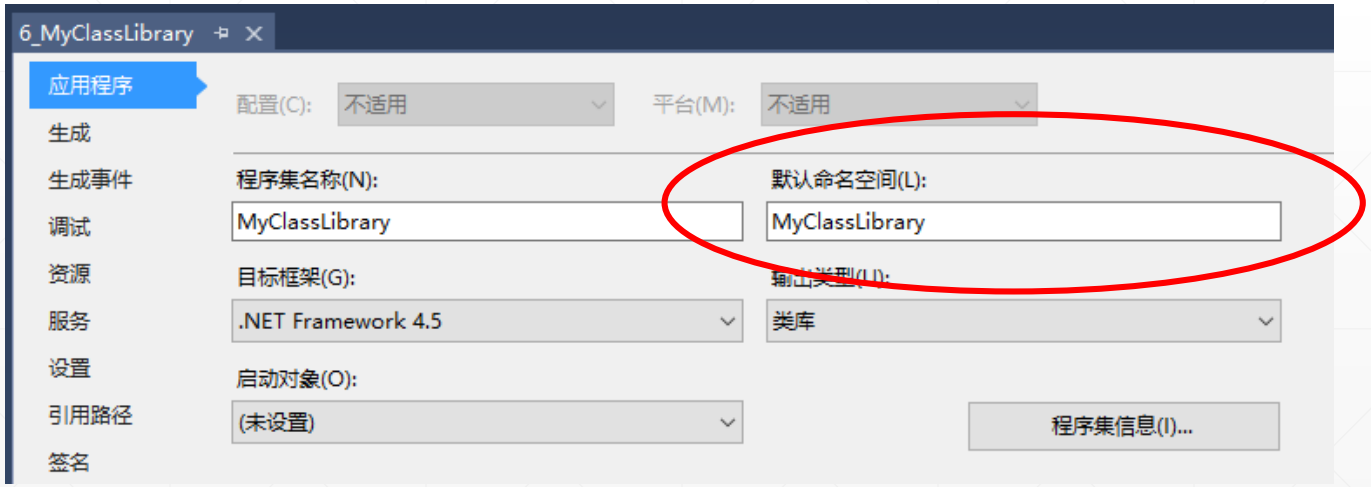




从“项目”菜单中选择“添加引用”，给当前项目加上对某个程序集文件的引用

程序集与命名空间的关系

1. 程序集的**物理**载体是“**实实在在可以看得到**”的.dll或.exe文件。
2. 命名空间是类的一种组织方式，它是一个**逻辑**上的概念。一个命名空间中的类可以分布在多个程序集文件中。
3. 一个程序集至少包含一个命名空间。可以在项目的“属性”面板中直接指定其生成的程序集默认的命名空间（如下图所示）。



基于程序集开发

- 通过将需要复用的代码放到类库项目中，生成二进制的.dll程序集文件，然后在新项目中直接引用此.dll文件，即可以使用其中的类。
- 使用程序集构造软件不需要有类的源代码，有.exe和.dll文件即可。
- 象程序集这种可以复用的软件系统构造单元，被称之为“软件组件”。

积木式的软件开发方式

基于程序集，可以方便地在.NET平台上实现组件化开发，其具体过程为：

1. 重用已有的组件
2. 开发部分新的组件
3. 将新老组件合在一起

“搭积木”。

用“软件积木”构造“软件大厦”

从现在开始，当开发正式的项目时，都应该采用基于程序集的开发方式！

“组件化开发”实践

1. 开发一个程序集，其中有一个MathOpt类，可以完成加减乘除四种运算。

2. 设计一个应用程序（Windows Form或控制台均可），指定或由用户输入两个数，程序调用程序集中的MathOpt类完成加减乘除运算并输出结果