

枚举

枚举是用户定义的整数类型。在声明一个枚举时,要指定该枚举的实例可以包含的一组可接受的值。

优点:

- 枚举可以使代码更易于维护,有助于确保给变量指定合法的、期望的值。
- 枚举使代码更清晰,允许用描述性的名称表示整数值,而不是用含义模糊、变化多端的数来表示。
- 枚举也使代码更易于键入。在给枚举类型的实例赋值时,Visual Studio .Net 会通过 IntelliSense 弹出一个包含可接受值的列表框,减少了按键次数,并能够让我们回忆起可选的值。

Example

```
namespace EnumTest
```

```
{  
    enum Week  
    {  
        Mon=1,  
        Tue=2,  
        Wed=3,  
        Thur,  
        Fri,  
        Sat,  
        Sun  
    }  
    [Flags]  
    enum ColorStyle  
    {  
        None = 0x00,  
        Red = 0x01,  
        Orange = 0x02,  
        Yellow = 0x04,  
        Green = 0x08,  
        Blue = 0x10,  
        Indigotic = 0x20,  
        Purple = 0x40,  
        All = Red | Orange | Yellow | Green | Blue | Indigotic | Purple  
    }  
    class enumtest  
    {  
        public static void Main(string[] args)  
        {  
            Week w = new Week();  
        }  
    }  
}
```

```

        Week day = (Week)6;
        day++;
        Console.WriteLine(day.ToString());
        foreach (Week item in Enum.GetValues(typeof(Week)))
        {
            Console.WriteLine("{0} is {1}", item.ToString("D"), item.ToString());
        }
        Console.WriteLine(Enum.GetUnderlyingType(typeof(Week)));
        ColorStyle mycs = ColorStyle.Red | ColorStyle.Yellow | ColorStyle.Blue;
        Console.WriteLine(mycs.ToString());
        Console.ReadLine();
    }
}
}

```

对象和类型

类和结构

类和结构实际上都是创建对象的模板,每个对象都包含数据,并提供了处理和访问数据的方法。

类定义了类的每个对象（称为实例）可以包含什么数据和功能。

结构与类的区别是它们在内存中的存储方式、访问方式（类是存储在堆(heap)上的引用类型，而结构是存储在栈(stack)上的值类型）和他们的一些特征（如结构不支持继承）。较小的数据类型使用结构可提高性能，但是在语法上，结构与类非常相似，主要的区别是使用关键字 **struct** 代替 **class** 来声明结构。

对于类和结构，都使用关键字 **new** 来声明实例：这个关键字创建对象并对其进行初始化，没有构造函数的类和结构的字段值都默认为 0。

类

类中的数据和函数成为类的成员。类可以包含嵌套的类型，成员的访问行可以是 **public**、**protected**、**internal** 或 **sealed**。

数据成员

数据成员是包含类的数据——字段、常量和事件的成员。数据成员可以是静态数据。类成员总是实例成员，除非用 **static** 进行显示的声明。

字段是与类相关的变量。

Example

namespace Test.ProfessnoI

```

{
    class Person
    {
        public int PersonId;
        public string FirstName;
        public string LastName;
    }
    class Demo
    {
        public static void Main(string[] args)
        {
            Person p = new Person();
            p.FirstName = "Will";
        }
    }
}

```

常量的关联方式同变量与类的关联方式。使用 `const` 关键字来声明常量。如果把他声明为 `public`，就可以在类的外部访问它。

Example

```

class Person
{
    public const string BirthDay= "2012/08/11";
    public int PersonId;
    public string FirstName;
    public string LastName;
}

```

事件是类的成员，在发生某些行为（如改变了字段或属性，或者进行了某种形式的用户交互操作）时，它可以让对象通知调用方。客户可以包含所谓“事件处理程序”的代码来响应该事件。

函数成员

函数成员提供了操作类中数据的某些功能，包括方法、属性、构造函数和终结器(`finalizer`)、运算符以及索引器。

- 方法是与某个类相关的函数，与数据成员一样，函数成员某认为实例成员，使用 `static` 修饰符可以把方法定义为静态方法。
- 属性是可以从客户端访问的函数组，其访问方式与访问类的公共字段类似。`C#`读写类中的属性提供了专用语法,所以不必使用那些名称中嵌有 `Get` 或 `Set` 的方法。因为属性的这种语法不同于一般函数的语法，在客户端代码中,虚拟的对象被当做实际的东西。
- 构造函数是在实例化对象时自动调用的特殊函数。它们必须与所属的类同名,且不能有返回类型。构造函数用于初始化字段的值。
- 终结器类似于构造函数,但是在 `CLR` 检测到不再需要某个对象时调用它。它们的名称与类相同,但前面有一个“~”符号。不可能预测什么时候调用终结器。
- 运算符执行的最简单的操作就是加法和减法。在两个整数相加时,严格地说,就是对整数

- 使用“+”运算符。C#允许指定把已有的运算符应用于自己的类(运算符重载)。
- 索引器允许对象以数组或集合的方式进行索引。

方法

在 C#术语中,“函数成员”不仅包含方法,而且也包含类和结构的一些非数据成员,如索引器、运算符、构造函数和析构函数等,甚至还有属性。这些都不是数据成员,字段、常量和事件才是数据成员。

给方法传递参数

参数可以通过引用或通过值传递给方法。在变量通过引用传递给方法时,被调用的方法得到的就是这个变量,所以在方法内部对变量进行的任何改变在方法退出后仍旧有效。而如果变量通过值传送给方法,被调用的方法得到的是变量的一个相同副本,也就是说,在方法退出后,对变量进行的修改会丢失。对于复杂的数据类型,按引用传递的效率更高,因为在按值传递时,必须复制大量的数据。

在 C#中,除非特别说明,所有的参数都通过值来传递。但是,在理解引用类型的含义时需要注

意。因为引用类型的变量只包含对象的引用,将要复制的正是这个引用,而不是对象本身,所以对底层对象的修改会保留下来。相反,值类型的对象包含的是实际数据,所以传递给方法的是数据本身的副本。例如,血通过值传递给方法,对应方法对该血的值所做的任何改变都没有改变原血对象的值。但如果把数组或其他引用类型传递给方法,对应的方法就会使用该引用改变数组中的值,而新值会反射在原始数组对象上。

Example

```
class Demo
{
    public static void Main(string[] args)
    {
        int[] ints = new int[] { 1, 2, 3, 5, 8 };
        int i = 1;
        //display the original values
        Console.WriteLine("i=" + i);
        Console.WriteLine("ints[0]=" + ints[0]);
        Console.WriteLine("Calling Some function");
        //After the methods returns,ints will change
        //i will not change
        SomeFun(ints, i);
        Console.WriteLine("i=" + i);
        Console.WriteLine("ints[0]=" + ints[0]);
        Console.ReadLine();
    }
    private static void SomeFun(int[] ints, int i)
    {
        ints[0] = 100;
```

```

        i = 100;
    }
}

```

Results

```

file:///D:/NSProject/Test/Tes
i=1
ints[0]=1
Calling Some function
i=1
ints[0]=100

```

C#的一个新特征是也可以给类编写无参数的静态构造函数。这种构造函数只执行一次,而构造函数是实例构造函数,只要创建类的对象,就会执行它。静态构造函数只能访问类的静态成员,不能访问类的实例成员。为在加载类时执行静态构造函数,而在创建实例时执行实例构造函数。

继承

实现继承和接口继承

在面向对象的编程中,有两种截然不同的继承类型:实现继承和接口继承

- 实现继承:表示一个类派生于一个基类型,它拥有该基类型的所有成员字段和函数。在实现继承中,派生类型采用基类型的每个函数的实现代码,除非在派生类型的定义中 **hiding** 重写某个函数的实现代码。在需要给现有的类型添加功能,或许多相关的类型共享一组重要的公共功能时,这种类型的继承非常有用。
- 接口继承:表示一个类型只继承了函数的签名,没有继承任何实现代码。在需要制定该类型具有某些可用的特性时,最好使用这种类型的继承。

结构和类

结构(值类型)、类(引用类型)。使用结构的一个限制是结构不支持继承,但每个结构都自动派生自 **System.ValueType**。不能编码实现类型层次的结构,但结构可以实现接口。也就是说,结构并不支持实现继承,但支持接口继承。事实上,定义结构和类可以总结为:

- 结构总是派生自 **System.ValueType**,它们还可以派生自任意多个接口。
- 类总是派生自用户选择的另一个类,它们还可以派生自任意多个接口。

如果有嵌套的类型,则内部的类型总是可以访问外部类型的所有成员。

接口

声明接口方法上与声明抽象类完全相同,但不允许重接口中任何成员的实现方式。一般情况下,接口只能包含方法、属性、索引器和事件的声明。

不能实例化接口,他只能包含其成员的签名。接口既不能有构造函数,也不能有字段。

接口定义也不允许包含运算符重载，尽管这不是因为声明它在原则上有什么问题，而是因为接口通常是公共协定，包含运算符重载会引起一些与其他.NET 语言不兼容的问题。

在接口定义中还不允许声明关于成员的修饰符。接口成员总是共有的，不能声明为虚拟或静态。

接口引用完全可以看做是类引用——但接口强大之处在于，它可以引用任何实现该接口的类。

Example

```
namespace Test.ProfessnoI
{
    interface IToString
    {
        void ToMyString(object obj);
    }
    class ITest:IToString
    {
        public void ToMyString(object obj)
        {
            Console.WriteLine(obj.ToString());
        }
    }
    class Demo
    {
        public static void Main(string[] args)
        {
            IToString ic = new ITest();
            ic.ToMyString("456");
            Console.ReadLine();
        }
    }
}
```

接口可以彼此继承，其方式与类的继承方式相同。

泛型

泛型是程序设计语言的一种特性。允许程序员在强类型程序设计语言中编写代码时定义一些可变部分，那些部分在使用前必须作出指明。各种程序设计语言和其编译器、运行环境对泛型的支持均不一样。将类型参数化以达到代码复用提高软件开发工作效率的一种数据类型。泛型类是引用类型，是堆对象，主要是引入了类型参数这个概念。

在程序编码中一些包含类型参数的类型，也就是说泛型的参数只可以代表类，不能代表个别对象。在程序编码中一些包含参数的类。其参数可以代表类或对象等等。

从值类型转换为引用类型成为装箱，拆箱即为将引用类型转换为值类型。容易操作，但是性能损失大。

System.Collections.Generic 名称空间中的 List<T>类不使用对象，而是在使用时定义类型。

在下面的例子中，List<T>类的泛型类型定义为 int，所以 int 类型在 JIT 编译器动态生成的类中使用，不再进行装箱和拆箱操作。(性能高)

Example

```
class Demo
{
    public static void Main(string[] args)
    {
        var list = new List<int>();
        list.Add(44); //no boxing--value types are stored in the List<int>
        int li = list[0]; //no unboxing,no cast needed
        foreach (int i in list)
        {
            Console.WriteLine(i);
        }
        Console.ReadLine();
    }
}
```

泛型的另一个特性是类型安全。使用 ArrayList 不能保证其所存储内容均为一个类型，在拆箱操作过程中不能保证类型转换一致，也就是所谓的类型安全。然而 List<T>，其中只能将固定类型的元素添加到其中。

二进制代码的重用，泛型类可以定义一次，并且可以用许多不同的类型实例化。

代码的扩展。因为泛型类的定义会放在程序集中，所以用特定类型实例化泛型类不会在 IL 代码中赋值这些类。但是，在 JIT 编译器把泛型类编译为本地代码时，会给每个值类型创建一个新类。引用类型共享同一个本地类的所有相同的实现代码。这是因为引用类型在实例化的泛型类中只需要 4 个字节的内存地址(32 位系统)，就可以引用一个引用类型。值类型包含在实例化的泛型类的内存中，同时因为每个值类型对内存的要求都不同，所以要为每个值类型实例化一个新类。

Example

```
using System;
using System.Collections.Generic;

namespace Test.Professnoi
{
    interface IDocument
    {
        string Title { get; set; }
        string Content { get; set; }
    }

    class Document : IDocument
    {
        public Document()
        {
        }
    }
}
```

```

    public string Title { get; set; }
    public string Content { get; set; }
    public Document(string title, string content)
    {
        this.Title = title;
        this.Content = content;
    }
}

```

`class DocumentManager<TDocument> where TDocument:IDocument` //where 要求所有 TDocument 对象都要实现 IDocement 接口

```

{
    private readonly Queue<TDocument> documentQueue = new
    Queue<TDocument>();

```

```

    public void AddDocument(TDocument doc)
    {
        lock (this)
        {
            documentQueue.Enqueue(doc);
        }
    }

```

```

    public bool IsDocumentAvailable
    {
        get { return documentQueue.Count > 0; }
    }
    public TDocument GetDocument()
    {

```

//不能把 null 赋予泛型类。原因是泛型类型也可以实例化为值类型，而 null 只能用于引用类型

//default 将 null 赋予引用类型，将赋予值类型 0

```

    TDocument doc = default(TDocument);
    lock (this)
    {
        doc = documentQueue.Dequeue();
    }
    return doc;
}

```

```

    public void DisplayAllDocument()
    {
        foreach (TDocument doc in documentQueue)
        {

```

//将 Tdocument 强制转换为 IDocement 接口，用来显示标题

`Console.WriteLine(((IDocument)doc).Title);`

//如果要求所有 DTDocument 对象都要派生自 Docement 类


```

        //Console.WriteLine(((Document)doc).Title);
    }
}

class Demo
{
    public static void Main(string[] args)
    {
        var dm = new DocumentManager<Document>();
        dm.AddDocument(new Document("Title A", "I am a good Student"));
        dm.AddDocument(new Document("Title B", "Will Yang Working"));
        dm.DisplayAllDocument();
        if (dm.IsDocumentAvailable)
        {
            Document doc = dm.GetDocument();
            Console.WriteLine(doc.Content);
        }
        Console.ReadLine();
    }
}

```

约束	说明
where T:struct	对于结构约束，类型 T 必须是值类型
where T:class	类约束制定类型 T 必须是引用类型
where T:IFoo	指定类型 T 必须实现接口 IFoo
where T:Foo	指定类型 T 必须派生自基类 Foo
where T:new()	这是一个构造函数约束，指定类型 T 必须有一个默认构造函数
where T1:T2	这个约束也可以指定，类型 T1 派生自泛型类型 T2。该约束也称为裸类型约束

泛型接口

使用泛型可以定义接口，在接口中定义的方法可以带泛型参数。

```

public interface IComparable<in T>
{
    int CompareTo(T other)
}

```

协变和抗变

.Net4 之前，泛型接口是不变的。.Net4 通过协变和抗变为泛型接口和泛型委托添加了一

个重要的扩展。协变和抗变指对参数和返回值的类型进行转换。

在 .Net 中，参数类型是协变的。假定有 Shape 和 Rectangle 类，Rectangle 派生自 Shape 基类。声明 Display() 方法是为了接受 Shape 类型对象作为其参数：

```
public void Display(Shape o) { }
```

现在可以传递派生自 Shape 基类的任意对象。因为 Rectangle 派生自 Shape，所以 Rectangle 满足 Shape 的所有要求，编译器接收这个方法调用：

```
Rectangle r=new Rectangle { Width=5, Height=2.5};  
Display(r);
```

方法的返回类型是抗变的。当方法返回一个 Shape 时，不能把它赋予 Rectangle，因为 Shape 不一定总是 Rectangle。反过来是可行的：如果一个方法像 GetRectangle() 方法那样返回一个 Rectangle，

```
public Rectangle GetRectangle();
```

就可以把结果赋予某个 Shape：

```
Shape s=GetRectangle();
```

在 .NET Framework 4 版本之前，这种行为方式不适用于泛型。在 C# 4 中，扩展后的语言支持泛型接口和泛型委托的协变和抗变。

```
class Shape  
{  
    public double Width  
    {  
        get;  
        set;  
    }  
    public double Height  
    {  
        get;  
        set;  
    }  
    public override string ToString()  
    {  
        return String.Format("Width {0}, Height {1}", Width, Height);  
    }  
}  
class Rectangle : Shape  
{  
}  
class CovariantExample  
{  
}
```

泛型接口的协变

如果泛型类用 out 关键字标注，泛型接口就是协变的。这也意味着返回类型只能是 T。

接口 `IIndex` 与类型 `T` 是协变的，并从一个只读索引器中返回这个类型：

```
public interface IIndex<out T>
{
    T this[index] { get; }
    Int Count { get; }
}
```

如果对接口 `IIndex` 使用了只读索引器，就把泛型类型 `T` 传递给方法，并从方法中检索这个类型，这不能通过协变来实现——泛型类型必须定义为不变的。不使用 `out` 和 `in` 标注，就可以把类型定义为不变的。

`IIndex<T>` 接口用 `RectangleCollection` 类来实现。`RectangleCollection` 类为泛型类型 `T` 定义了 `Rectangle`：

Example

```
using System;
namespace Test.Professno1
{
    class Shape
    {
        public double Width
        {
            get;
            set;
        }
        public double Height
        {
            get;
            set;
        }
        public override string ToString()
        {
            return String.Format("Width {0}, Height {1}", Width, Height);
        }
    }
    class Rectangle : Shape
    {
    }
    interface IIndex<out T>
    {
        T this[int index] { get; }
        int Count { get; }
    }
    class RectangleCollection : IIndex<Rectangle>
    {
        private Rectangle[] data = new Rectangle[3]
        {
```

```

        new Rectangle{Height=2,Width=5},
        new Rectangle{Height=3,Width=7},
        new Rectangle{Height=4.5,Width=2.9},
    };

    public static RectangleCollection GetRectangles()
    {
        return new RectangleCollection();
    }

    public Rectangle this[int index]
    {
        get
        {
            if (index < 0 || index > data.Length)
                throw new ArgumentOutOfRangeException("index");
            return data[index];
        }
    }

    public int Count
    {
        get { return data.Length; }
    }
}

class CovariantExample
{
    public static void Main(string[] args)
    {
        IIndex<Rectangle> rectangle = RectangleCollection.GetRectangles();
        IIndex<Shape> shapes = rectangle;
        for (int i = 0; i < shapes.Count; i++)
        {
            Console.WriteLine(shapes[i]);
        }
        Console.ReadLine();
    }
}

```

RectangleCollection.GetRectangle() 方法返回一个实现 IIndex<Rectangle> 接口的 RectangleCollection 类，所以可以把返回值赋予 IIndex<Rectangle> 类型的变量 rectangle。因为接口是协变的，所以也可以把返回值赋予 IIndex<Shape> 类型的变量。Shape 不需要 Rectangle 没有提供的内容。使用 shapes 变量，就可以在 for 循环中使用接口中的索引器和 Count 属性：

泛型接口的抗变

如果泛型类型用 `in` 关键字标注，泛型接口就是抗变的。这样，接口只能把泛型类型 `T` 用作其方法的输入：

```
interface IDisplay<in T>
{
    void Show(T item);
}
```

`ShapeDisplay` 类实现 `IDisplay<Shape>`，并使用 `Shape` 对象作为输入参数：

```
class ShapeDisplay : IDisplay<Shape>
{
    public void Show(Shape shape)
    {
        Console.WriteLine("{0} Width: {1}, Height: {2}", shape.GetType().Name,
shape.Width, shape.Height);
    }
}
```

创建 `ShapeDisplay` 的一个新实例，会返回 `IDisplay<Shape>`，并把它赋予 `shapedisplay` 变量。因为 `IDisplay<T>` 是抗变的，所以可以把结果赋予 `IDisplay<Rectangle>`，其中 `Rectangle` 派生自 `Shape`。

这次接口的方法只能把泛型类型定义为输入，而 `Rectangle` 满足 `Shape` 的所有要求：

```
public static void Main(string[] args)
{
    //.....
    //抗变
    IDisplay<Shape> shapedisplay = new ShapeDisplay();
    IDisplay<Rectangle> rectangledisplay = shapedisplay;
    rectangledisplay.Show(rectangle[0]);
    Console.ReadLine();
}
```

Example –协变与抗变

```
using System;

namespace Test.ProfessnoI
{
    class Shape
    {
        public double Width
        {
            get;
            set;
        }
        public double Height
        {
```

```

        get;
        set;
    }
    public override string ToString()
    {
        return String.Format("Width {0}, Height {1}", Width, Height);
    }
}
class Rectangle : Shape
{
}
#region 协变
interface IIndex<out T>
{
    T this[int index] { get; }
    int Count { get; }
}
class RectangleCollection : IIndex<Rectangle>
{
    private Rectangle[] data = new Rectangle[3]
    {
        new Rectangle{Height=2,Width=5},
        new Rectangle{Height=3,Width=7},
        new Rectangle{Height=4.5,Width=2.9},
    };

    public static RectangleCollection GetRectangles()
    {
        return new RectangleCollection();
    }

    public Rectangle this[int index]
    {
        get
        {
            if (index < 0 || index > data.Length)
                throw new ArgumentOutOfRangeException("index");
            return data[index];
        }
    }
    public int Count
    {
        get { return data.Length; }
    }
}

```

```

    }
    #endregion

    #region 抗变
    interface IDisplay<in T>
    {
        void Show(T item);
    }
    class ShapeDisplay : IDisplay<Shape>
    {
        public void Show(Shape shape)
        {
            Console.WriteLine("{0} Width: {1}, Height: {2}", shape.GetType().Name,
shape.Width, shape.Height);
        }
    }
    #endregion

    class CovariantExample
    {
        public static void Main(string[] args)
        {
            //协变
            Index<Rectangle> rectangle = RectangleCollection.GetRectangles();
            RectangleCollection rc = new RectangleCollection();
            Index<Shape> shapes = rectangle;
            for (int i = 0; i < shapes.Count; i++)
            {
                Console.WriteLine(shapes[i]);
            }
            //抗变
            IDisplay<Shape> shapedisplay = new ShapeDisplay();
            IDisplay<Rectangle> rectangledisplay = shapedisplay;
            rectangledisplay.Show(rectangle[0]);
            Console.ReadLine();
        }
    }
}

```

泛型结构

与类相似，结构也可以是泛型的。它们非常类似于泛型类，只是没有继承特性。

```
using System;
```

```

namespace Test.Professno1
{
    class Nullable<T> where T : struct
    {
        private bool hasValue;
        public bool HasValue
        {
            get { return hasValue; }
        }
        public Nullable(T value)
        {
            this.hasValue = true;
        }
        private T value;
        public T Value
        {
            get
            {
                if (!hasValue)
                {
                    throw new InvalidOperationException("no value!");
                }
                else
                {
                    return value;
                }
            }
        }
    }
    public static explicit operator T(Nullable<T> value)
    {
        return value.Value;
    }

    public static implicit operator Nullable<T>(T value)
    {
        return new Nullable<T>(value);
    }
    public override string ToString()
    {
        if (!HasValue)
        {
            return String.Empty;
        }
    }
}

```



```

        return this.value.ToString();
    }
}
class NullableExample
{
    public static void Main(string[] args)
    {
        Nullable<int> x;
        x = 4;
        int? y = null;
        if (y == null)
        {
            Console.WriteLine("Null");
        }
        if (x.HasValue)
        {
            int z = x.Value;
        }
        Console.WriteLine(x.ToString());
        Console.ReadLine();
    }
}
}

```

因为可控类型使用得非常频繁，定义可空类型变量时，可以不使用泛型结构的语法，而使用“?”运算符。可空类型可以与 null 和数字比较。

非可空类型可以转换为可空类型。从非可空类型转换为可空类型是，在不需要强制类型转换的地方可以进行隐式转换。

```

int y1=4;
int? x1=y1;

```

从可空类型转换为非可空类型可能会失败。如果可空类型的值是 null，并且把 null 值赋予非可空类型，就会抛出 `InvalidOperationException` 类型的异常。这就是需要类型强制转换运算符进行显示转换的原因：

```

int? x1=GetNullableType();
int y1=(int) x1;

```

如果不进行显示类型转换，还可以使用合并运算符(coalescing operator) 从可空类型转换为非可空类型。合并运算符的语法是“??”，为转换定义了一个默认值，以防可空类型的值是 null。这里，如果 x1 是 null，y1 的值就是 0。

```

int? x1=GetNullableType();
int y1=x1 ?? 0;

```

泛型方法

除了定义泛型类之外，还可以定义泛型方法。在泛型方法中，泛型类型用方法声明来定义。泛型方法可以在非泛型类中定义。

C#编译器会通过调用方法来获取参数的类型，所以不需要把泛型类型赋予方法调用。泛型方法可以像非泛型方法那样调用。

Example

```
using System;
```

```
using System.Collections.Generic;
```

```
namespace Test.ProfessnoI
```

```
{
```

```
    interface IAccount
```

```
    {
```

```
        decimal Balance { get; }
```

```
        string Name { get; }
```

```
    }
```

```
    class Account:IAccount
```

```
    {
```

```
        public string Name { get; private set; }
```

```
        public decimal Balance { get; private set; }
```

```
        public Account(string name ,Decimal balance)
```

```
        {
```

```
            this.Name = name;
```

```
            this.Balance = balance;
```

```
        }
```

```
    }
```

```
    static class Algorithm
```

```
    {
```

```
        public static decimal AccumulateSimple(IEnumerable<Account> source)
```

```
        {
```

```
            decimal sum = 0;
```

```
            foreach (Account a in source)
```

```
            {
```

```
                sum += a.Balance;
```

```
            }
```

```
            return sum;
```

```
        }
```

```
        public static decimal Accumulate<TAccount>(IEnumerable<TAccount> source)
```

where TAccount:IAccount //泛型函数，其中要求 TAccount 必须实现 IAccount 接口，原因是函数中使用 a.Balance 并不是所有 TAccount 都可以实现，也就是说原来的 TAccount 可以是任何类型，但不是任何类型都具有 TAccount.Balance 这个属性

```
        {
```

```
            decimal sum = 0;
```

```
            foreach (TAccount a in source)
```

```
            {
```

```
                sum += a.Balance;
```

```

        }
        return sum;
    }
}
class GenericExample
{
    public static void Main(string[] args)
    {
        var account = new List<Account>()
        {
            new Account("Will",2500),
            new Account("Yang",3650)
        };
        Console.WriteLine(Algorithm.AccumulateSimple(account));
        Console.WriteLine(Algorithm.Accumulate(account));
        Console.ReadLine();
    }
}
}

```

帶委托的泛型方法

泛型方法规范

泛型方法可以重载，为特定的类型定义规范。这也适用于带泛型参数的方法。所调用的方法是在编译期定义的，而不是运行期间。

Example

```

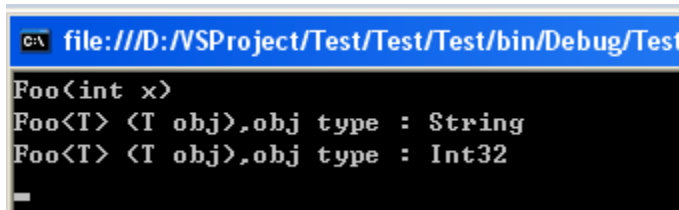
namespace Test.Professnoi
{
    class GenericFunctionExample
    {
        private static void Foo<T>(T obj)
        {
            Console.WriteLine("Foo<T> (T obj),obj type : {0} ", obj.GetType().Name);
        }
        private static void Foo(int x)
        {
            Console.WriteLine("Foo(int x)");
        }
        private static void Bar<T>(T obj)
        {
            Foo(obj);
        }
    }
}

```

```

        public static void Main(string[] args)
        {
            Foo(33);
            Foo("33");
            Bar(33);
            Console.ReadLine();
        }
    }
}

```



```

file:///D:/VSProject/Test/Test/Test/bin/Debug/Test
Foo<int x>
Foo<T> <T obj>,obj type : String
Foo<T> <T obj>,obj type : Int32

```

从输出可以看出，`Bar()`方法选择了泛型 `Foo()`方法，而不是 `int` 参数重载的 `Foo()`方法。原因是编译器在编译期间选择 `Bar()`方法调用的 `Foo()`方法。由于 `Bar()`方法定义了一个泛型参数，而且泛型 `Foo()`方法匹配这个类型，所以调用了 `Foo()`方法。在运行期间给 `Bar()`方法传递了一个 `int` 值不会改变这一点。

小结

通过泛型类可以创建独立于类型的类，泛型方法是独立于类型的方法。接口、结构和委托也可以用泛型的方式创建。泛型引入了一种新的编程方式，它们都是类型安全的，泛型委托可以去除集合中的算法。

数组

简单数组

声明：`int [] myArray;`

声明了数组后，就必须为数组分配内存，以保存数组的所有元素。数组是引用类型，所以必须给它分配堆上的内存。需要使用 `new` 运算符，制定数组中元素的类型和数量来初始化数组的变量。

```
int [] myArray=new int[4];
```



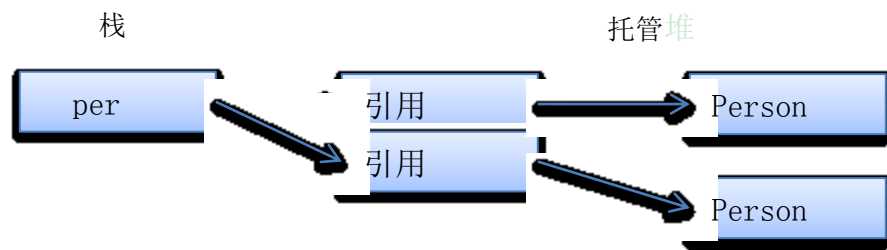
使用引用类型

除了能声明预定义类型的数组，还可以自定义类型的数组。

```
namespace Test.Professno1
{
    class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public override string ToString()
        {
            return String.Format("{0} {1}", FirstName, LastName);
        }
    }
    class ClassArrayExample
    {
        static void Main(string[] args)
        {
            Person[] per = new Person[2];
            per[0] = new Person { FirstName="Will", LastName="Yang"};
            per[1] = new Person { FirstName = "Wille", LastName = "Yang" };

            Console.WriteLine(per[0].ToString());
            Person[] p = //数组初始化器
            {
                new Person { FirstName = "Will", LastName = "Yang" },
                new Person { FirstName = "Wille", LastName = "Yang" }
            };
            Console.ReadLine();
        }
    }
}
```

内存空间分布:



多维数组

声明: `int [,] myArray=new int [3,3];`
 []中加两个 “,” 就可以声明一个三维数组

锯齿数组

声明: `int[][] myArray=new int[3][];`
`myArray[0]=new int[2] {1, 2};`
`myArray[1]=new int[3] {2,5,7};`
`myArray[2]=new int[5] {3,6,9,10,2};`

Example

```
static void Main(string[] args)
{
    int[][] array= new int[2][];
    array[0] = new int[3] { 1, 2, 3 };
    array[1] = new int[6] { 1, 2, 3, 6, 5, 4 };
    for (int i = 0; i < array.Length; i++)
        for (int j = 0; j < array[i].Length; j++)
            Console.WriteLine("row {0}, column {1}, element: {2}", i, j, array[i][j]);
    Console.ReadLine();
}
```

```
row 0, column 0, element: 1
row 0, column 1, element: 2
row 0, column 2, element: 3
row 1, column 0, element: 1
row 1, column 1, element: 2
row 1, column 2, element: 3
row 1, column 3, element: 6
row 1, column 4, element: 5
row 1, column 5, element: 4
```

Array 类

创建数组

Array 类是一个抽象类，所以不能用构造函数来创建数组。除了创建数组实例之外，还可以使用静态方法 `CreateInstance()` 创建数组。如果事先不知道元素的类型，该静态方法就非常有用，因为类型可以作为 `Type` 对象传递给 `CreateInstance()` 方法。其中可以用 `SetValue()` 方法设置对应元素的值，用 `GetValue()` 方法读取对应元素的值。

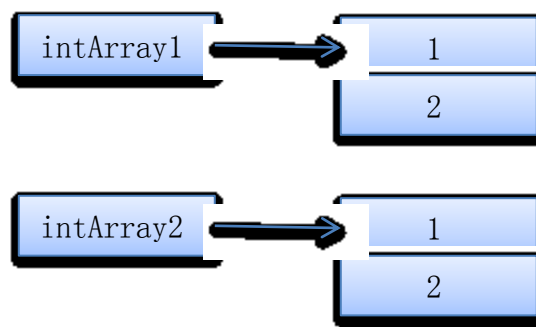
Example

```
namespace Test.Professnoi
{
    class ArrayListExample
    {
        static void Main(string[] args)
        {
            Array intArray = Array.CreateInstance(typeof(int), 5);
            for (int i = 0; i < intArray.Length; i++)
            {
                intArray.SetValue(i, i);
            }
            for (int i = 0; i < intArray.Length; i++)
            {
                Console.WriteLine(intArray.GetValue(i));
            }
            Console.ReadLine();
        }
    }
}
```

复制数组

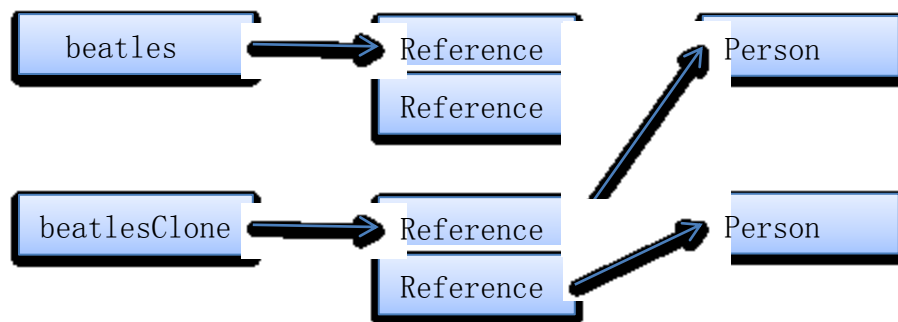
因为数组是引用类型，所以将一个数组变量赋予另一个数组变量，就会得到两个引用同一数组的变量。而复制数组，会使数组实现 `ICloneable` 接口。这个接口定义的 `Clone()` 方法会创建数组的浅表副本。

```
int[] intArray1 = { 1, 2 };
int[] intArray2 = (int[]) intArray1.Clone();
```



如果数组包含引用类型，则不赋值元素，而只复制引用。

```
Person[] beatles=
{
    new Person { FirstName="John", LastName="Lennon"},
    new Person { FirstName="Paul", LastName="McCartney"}
};
Person[] beatlesClone=( Person[] )beatles.Clone();
```



除了使用 `Clone()` 方法之外，还可以使用 `Array.Copy()` 方法创建浅表副本。但 `Clone()` 方法和 `Copy()` 方法有一个重要的区别：`Clone()` 方法会创建一个数组，而 `Copy()` 方法必须传递阶数相同且有足够元素的已有数组。

如果需要包含引用类型的数组的深层副本，就必须迭代数组并创建新对象。

排序

`Array` 类使用 `QuickSort` 算法对数组中元素进行排序。`Sort()` 方法需要对数组中的元素实现 `IComparable` 接口。因为简单类型(如 `System.String` 和 `System.Int32`)实现 `IComparable` 接口，所以可以对包含这些类型的元素排序。

如果对数组使用自定义类，就必须实现 `IComparable` 接口。这个接口只定义了一个方法 `CompareTo()`，如果要比较的对象相等，该方法就返回 0。如果该实例应排在参数对象的前面，该方法就返回小于 0 的值。如果该实例应排在参数对象的后面，该方法就返回大于 0 的值。

Example

```
namespace Test.Professnoi
{
    class Person : IComparable<Person>
```



```

{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public override string ToString()
    {
        return String.Format("{0} {1}", FirstName, LastName);
    }

    public int CompareTo(Person other)
    {
        if (other == null)
        {
            throw new ArgumentNullException("other");
        }
        int result = this.LastName.CompareTo(other.LastName);
        if (result == 0)
        {
            result = this.FirstName.CompareTo(other.FirstName);
        }
        return result;
    }
}

class IComparableExample
{
    static void Main(string[] args)
    {
        Person[] per = new Person[2];
        per[0] = new Person { FirstName = "Will", LastName = "Yang" };
        per[1] = new Person { FirstName = "Aille", LastName = "Yang" };
        Person[] p =
        {
            new Person { FirstName = "Will", LastName = "Yang" },
            new Person { FirstName = "Wille", LastName = "Yang" }
        };
        Array.Sort(per);
        foreach (var pe in per)
        {
            Console.WriteLine(pe);
        }
        Console.ReadLine();
    }
}
}
Example

```

```

namespace Test.Professno1
{
    enum PersonCompareType
    {
        FirstName,
        LastName
    }
    class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public override string ToString()
        {
            return String.Format("{0} {1}", FirstName, LastName);
        }
    }
    class PersonComparer : IComparer<Person>
    {
        private PersonCompareType compareType;

        public PersonComparer(PersonCompareType compareType)
        {
            this.compareType = compareType;
        }
        public int Compare(Person x, Person y)
        {
            if (x == null)
                throw new ArgumentNullException("x");
            if (y == null)
                throw new ArgumentNullException("y");
            switch (compareType)
            {
                case PersonCompareType.FirstName:
                    return x.FirstName.CompareTo(y.FirstName);
                case PersonCompareType.LastName:
                    return x.LastName.CompareTo(y.LastName);
                default:
                    throw new ArgumentException("Unexpected compare type" );
            }
        }
    }
    class EnumICompareExample
    {
        static void Main(string[] args)
    }
}

```

```

    {
        Person[] per = new Person[2];
        per[0] = new Person { FirstName = "Will", LastName = "Yang" };
        per[1] = new Person { FirstName = "Aille", LastName = "Yang" };
        Array.Sort(per, new PersonComparer(PersonCompareType.FirstName));
        foreach (var pe in per)
        {
            Console.WriteLine(pe);
        }
        Console.ReadLine();
    }
}

```

类 `PersonComparer` 实现 `IComparer<Person>` 接口，可以按照 `FirstName` 或 `LastName` 对 `Person` 对象排序。枚举 `PersonCompareType` 定义了可用于 `PersonComparer` 的排序选项：`FirstName` 和 `LastName`。排序方式由 `PersonComparer` 类的构造函数定义，在构造函数中设置了一个 `PersonCompareType` 值。实现 `Compare()` 方法时使用一个 `switch` 语句指定是按 `FirstName` 还是 `LastName` 排序。

`Array.Sort(per, new PersonComparer(PersonCompareType.FirstName));` 将 一 个 `PersonComparer` 对象传递给 `Array.Sort()` 方法的第二个参数。

数组作为参数

数组可以作为参数传递给方法，也可以从方法中返回。要返回一个数组，只需把数组声明为返回类型。

```

static Person[] GetPerson()
{
    return new Person[] {
        new Person { FirstName="Damon", LastName="Hill" },
        new Person { FirstName="Niki", LastName="Luda" }
    };
}

```

要把数组传递给方法，应把数组声明为参数，如：

```

static void DisplayPerson( Person[] persons)
{
    //.....
}

```

数组协变

数组支持协变。这表示数组可以声明为基类，其派生类型的元素可以赋予数组元素。例

如，可以声明一个 `object[]` 类型的参数，给他传递一个 `Person[]`：

```
static void DisplayArray(object[] data)
{
    //.....
}
```

Tips:

- 数组协变只能用于引用类型，不能用于值类型。
- 数组协变有一个问题，它只能通过运行时异常来解决。如果把 `Person` 数组赋予给 `object` 数组，`object` 数组就可以使用派生自 `object` 的任何元素。例如，编译器允许把字符串传递给数组元素。但因为 `object` 数组引用 `Person` 数组，所以会出现一个运行时异常。

ArraySegment<T>

结构 `ArraySegment<T>` 表示数组的一段。如果某方法应返回数组的一部分，或者给某方法传递数组的一部分，就可以使用数组段。通过 `ArraySegment<T>` 可以传递一个参数，而不是用 3 个参数传递数组、偏移量和元素个数。在这个结构体中，关于数组段的信息(偏移量和元素个数)包含在结构的成员中。

`SumOfSegments()` 方法提取遗嘱 `ArraySegment<int>` 元素，计算该数组段定义的所有整数之和，并返回整数和：

```
using System;
```

```
namespace Test.Professional
```

```
{
    class ArraySegmentExample
    {
        static int SumOfSegments(ArraySegment<int>[] segments)
        {
            int sum = 0;
            foreach (var segment in segments)
            {
                for (int i = segment.Offset; i < segment.Offset + segment.Count; i++)
                {
                    sum += segment.Array[i];
                }
            }
            return sum;
        }
        public static void Main(string[] args)
        {
            int[] ar1 = { 1, 4, 5, 11, 13, 18 };
            int[] ar2 = { 3, 4, 5, 18, 21, 27, 33 };
            var segments = new ArraySegment<int>[2]
            {
                //传递数组段，第一个元素从 ar1 的第一个元素开始，引用了 3 个元素；第
```

二个数组从元素从 ar2 的第 4 个元素开始，引用了 3 个元素。

```
        new ArraySegment<int>(ar1,0,3),
        new ArraySegment<int>(ar2,3,3)
    };
    int sum = SumOfSegments(segments);
    Console.WriteLine(sum);
    Console.ReadLine();
}
}
}
```

Tips:

- 数组段不复制原数组的元素，但原数组可以通过 `ArraySegment<T>` 访问。如果数组段中的元素改变了，这些变化就会反映到原数组中。

Example:

`using System;`

```
namespace Test.Professional
{
    class ArraySegmentExample
    {
        static int SumOfSegments(ArraySegment<int>[] segments)
        {
            int sum = 0;
            foreach (var segment in segments)
            {
                for (int i = segment.Offset; i < segment.Offset + segment.Count; i++)
                {
                    sum += segment.Array[i];
                }
            }
            return sum;
        }
        static void SetSegments(ArraySegment<int>[] segments)
        {
            foreach (var segment in segments)
            {
                for (int i = segment.Offset; i < segment.Offset + segment.Count; i++)
                {
                    segment.Array[i] = i;
                }
            }
        }
        public static void Main(string[] args)
```

```

{
    int[] ar1 = { 1, 4, 5, 11, 13, 18 };
    int[] ar2 = { 3, 4, 5, 18, 21, 27, 33 };
    var segments = new ArraySegment<int>[2]
    {
        new ArraySegment<int>(ar1,0,3),
        new ArraySegment<int>(ar2,3,3)
    };
    int sum = SumOfSegments(segments);
    Console.WriteLine(sum);
    Console.WriteLine("Begin to Change!");
    SetSegments(segments);
    Console.WriteLine("After Change!");
    foreach (var segment in segments)
    {
        for (int i = segment.Offset; i < segment.Offset + segment.Count; i++)
        {
            Console.Write(segment.Array[i]+"\t");
        }
        Console.WriteLine();
    }
    Console.ReadLine();
}
}
}

```

运行结果:

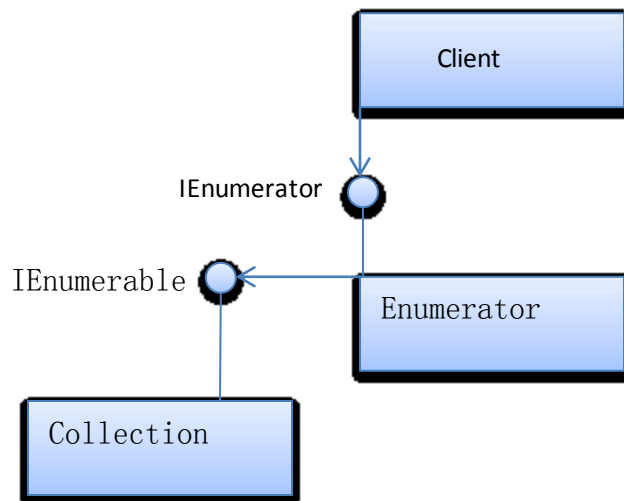
```

c:\ file:///D:/VSPProject/Test/Tes
76
Begin to Change!
After Change!
0      1      2
3      4      5

```

枚举

在 `foreach` 语句中使用枚举，可以迭代集合中的元素，且无需知道集合中元素个数。`foreach` 语句中使用了一个枚举器。下图显示了调用 `foreach` 方法的客户端和集合之间的关系。数组或集合实现带 `GetEnumerator()` 方法的 `IEnumerable` 接口。`GetEnumerator()` 方法返回一个实现 `IEnumerator` 接口的枚举。接着，`foreach` 语句就可以使用 `IEnumerator` 接口迭代集合了。



Tips:

- GetEnumerator()方法用 IEnumerable 接口定义。foreach 语句并不真的需要在集合类中实现这个接口，有一个名为 GetEnumerator()的方法，它返回实现了 IEnumerator 接口的对象就足够了。

IEnumerator 接口

foreach 语句使用 IEnumerator 接口的方法和属性，迭代集合中的所有元素。为此，IEnumerator 定义了 Current 属性，来返回光标所在的元素，该接口的 MoveNext()方法移动到集合的下一个元素上，如果有这个元素，该方法就返回 true。如果集合不再有更多的元素，该方法就返回 false。

这个接口的泛型版本 IEnumerator<T>派生自接口 IDisposable，因此定义了 Dispose()方法，来清理枚举器占用的资源。

Tips:

- IEnumerator 接口还定义了 Reset()方法，以与 COM 交互操作。许多.NET 枚举器通过抛出 NotSupportedException 类型的异常，来实现这个方法。

foreach 语句

C#的 foreach 语句不会解析为 IL 代码中的 foreach 语句。C#编译器会把 foreach 语句转换为 IEnumerable 接口的方法和属性。下面是一条简单的 foreach 语句，它迭代 persons 数组中的所有元素，并逐个显示它们：

Example:

```
using System;
```

```
namespace Test.Professional
```

```
{
```

```
    class Person
```

```
    {
```

```
        public string FirstName { get; set; }
```

```

        public string LastName { get; set; }
        public override string ToString()
        {
            return String.Format("{0} {1}", FirstName, LastName);
        }
    }
}
class ForeachExample
{
    public static void Main(string[] args)
    {
        Person[] per = new Person[3];
        per[0] = new Person { FirstName = "Will", LastName = "Yang" };
        per[1] = new Person { FirstName = "Aille", LastName = "Yang" };
        per[2] = new Person { FirstName = "Bob", LastName = "Long" };
        foreach (Person p in per)
        {
            Console.WriteLine(p.ToString());
        }
        Console.ReadLine();
    }
}

```

上述代码中的 `foreach` 语句会解析为下面的代码段。首先，调用 `GetEnumerator()` 方法，获得数组的一个枚举器。在 `while` 循环中——只要 `MoveNext()` 返回 `true`——就用 `Current` 属性访问数组中的元素：

```

IEnumerator<Person> enumerator=perons.GetEnumerator();
while(enumerator.MoveNext())
{
    Person p=enumerator.Current;
    Console.WriteLine(p);
}

```

yield 语句

自 C# 的第一个版本以来，使用 `foreach` 语句可以轻松地迭代集合。在 C#1.0 中，创建枚举器仍需要做大量的工作。C#2.0 添加了 `yield` 语句，以便于创建枚举器。`yield return` 语句返回集合的一个元素，并移动到下一个元素上。`yield break` 可停止迭代。

下面一个例子是用 `yield return` 语句实现一个简单集合的代码。`HelloCollection` 类包含 `GetEnumerator()` 方法。该方法的实现代码包含两条 `yield return` 语句，它们分别返回字符串 `Hello` 和 `World`。

Example:

```

using System;
using System.Collections;

```



```

namespace Test.Professional
{
    class HelloCollection
    {
        public IEnumerator GetEnumerator()
        {
            yield return "Hello";
            yield return "World";
        }
    }
    class YieldExample
    {
        public static void Main(string[] args)
        {
            var helloCollection = new HelloCollection();
            foreach (var s in helloCollection)
            {
                Console.WriteLine(s);
            }
            Console.ReadLine();
        }
    }
}

```

Tips:

- 包含 `yield` 语句的方法或属性也称为迭代块。迭代块代码必须声明为返回 `IEnumerator` 或 `IEnumerable` 接口，或者这些接口的泛型版本。这个块可以包含多条 `yield return` 语句或 `yield break` 语句，但不能包含 `return` 语句。
- `yield` 语句会产生一个枚举器，而不仅仅生成一个包含的项的列表。这个枚举器通过 `foreach` 语句调用。从 `foreach` 中依次访问每一项时，就会访问枚举其。这样就可以迭代大量数据，而无需一次把所有的数据都读入内存。

使用迭代块，编译器会生成一个 `yield` 类型，其中包含一个状态机，如下面代码所示。

`yield` 类型实现 `IEnumerator` 和 `IDisposable` 接口的属性和方法。在下面的例子中，可以把 `yield` 类型看作内部类 `Enumerator`。外部内的 `GetEnumerator()` 方法实例化并返回一个新的 `yield` 类型，在 `yield` 类型中，变量 `state` 定义了迭代的当前位置，每次调用 `MoveNext()` 时，当前位置都会改变。`MoveNext()` 封装了迭代块的代码，并设置了 `current` 变量的值，从而是 `Current` 属性根据位置返回一个对象。

Example

```

using System;
using System.Collections;
using System.Collections.Generic;

namespace Test.Professional
{
    class HelloCollection

```

```

{
    public IEnumerator<string> GetEnumerator()
    {
        return new Enumerator(0);
    }
}

public class Enumerator : IEnumerator<string>, IEnumerator, IDisposable
{
    private int state;
    private string current;

    public Enumerator(int state)
    {
        this.state = state;
    }

    bool System.Collections.IEnumerator.MoveNext()
    {
        switch (state)
        {
            case 0:
                current = "Hello";
                state = 1;
                return true;
            case 1:
                current = "World";
                state = 2;
                return true;
            case 2:
                break;
            default:
                break;
        }
        return false;
    }

    void System.Collections.IEnumerator.Reset()
    {
        throw new NotSupportedException();
    }

    string System.Collections.Generic.IEnumerator<string>.Current
    {
        get { return current; }
    }

    object System.Collections.IEnumerator.Current
    {

```

```

        get { return current; }
    }
    void IDisposable.Dispose()
    {
    }
}
}
class YieldExample
{
    public static void Main(string[] args)
    {
        var helloCollection = new HelloCollection();
        foreach (var s in helloCollection)
        {
            //s为string类型
            Console.WriteLine(s);
            Console.WriteLine(s.GetType());
        }
        Console.ReadLine();
    }
}
}

```

迭代集合的不同方式

下面的示例可以使用 `yield return` 语句，以不同方式迭代集合的类。类 `MusicTitles` 可以使用默认方式通过 `GetEnumerator()` 方法迭代标题，用 `Reverse()` 方法逆序迭代标题，用 `Subset()` 方法迭代子集：

Example:

```

using System;
using System.Collections.Generic;

namespace Test.Professional
{
    class MusicTitles
    {
        string[] names = { "Tubular Bells", "Hergset Ridge", "Ommadawn", "Platinum" };
        public IEnumerator<string> GetEnumerator()
        {
            for (int i = 0; i < 4; i++)
            {
                yield return names[i];
            }
        }
    }
}

```

```

public IEnumerable<string> Reverse()
{
    for (int i = 3; i >= 0; i--)
    {
        yield return names[i];
    }
}

public IEnumerable<string> Subset(int index, int length)
{
    for (int i = index; i < index + length; i++)
    {
        yield return names[i];
    }
}

}

class Demo
{
    public static void Main(string[] args)
    {
        var titles = new MusicTitles();
        foreach (var title in titles)
        {
            Console.WriteLine(title);
        }
        Console.WriteLine();
        Console.WriteLine("Reverse");
        foreach (var title in titles.Reverse())
        {
            Console.WriteLine(title);
        }
        Console.WriteLine();
        Console.WriteLine("Subset");
        foreach (var title in titles.Subset(2, 2))
        {
            Console.WriteLine(title);
        }
        Console.ReadLine();
    }
}

```

Tips:

- 类支持的默认迭代是定义为返回 `IEnumerator` 的 `GetEnumerator()` 方法。命名的迭代返回 `IEnumerable`。

用 yield return 返回枚举器

使用 yield 语句还可以完成更复杂的任务，例如，从 yield return 中返回枚举器。

在 TicTacToe 游戏中有 9 个域，玩家轮流在这些域中防止一个“十”字或一个圆。这些移动操作由 GameMoves 类模拟。方法 Cross() 和 Circle() 是创建迭代类型的迭代块。变量 cross 和 circle 在 GameMoves 类的构造函数中设置为 Cross() 和 Circle() 方法。这些字段不设置为调用的方法，而是设置为用迭代块定义的迭代类型。在 Cross() 迭代块中，将移动操作的信息写到控制台上，并递增移动次数。如果移动次数大于 8，就用 yield break 停止迭代；否则，就在每次迭代中返回 yield 类型的 circle 的枚举对象。Circle() 迭代块非常类似于 Cross() 迭代块，只是它在每次迭代中返回 cross 迭代器类型。

Example

```
using System;
using System.Collections;
using System.Collections.Generic;

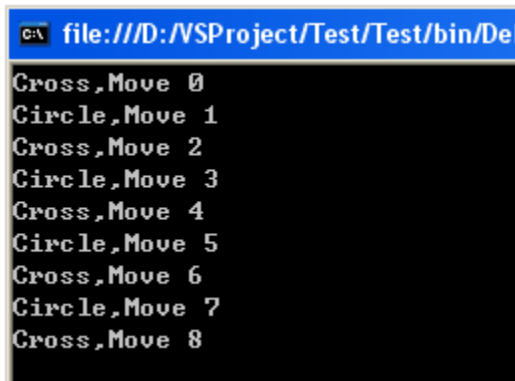
namespace Test.Professional
{
    class GameMoves
    {
        private IEnumerator cross;
        private IEnumerator circle;
        public GameMoves()
        {
            cross = Cross();
            circle = Circle();
        }
        private int move = 0;
        private int maxMoves = 9;
        public IEnumerator Cross()
        {
            while (true)
            {
                Console.WriteLine("Cross, Move {0}", move);
                if (++move >= maxMoves)
                {
                    yield break;
                }
                yield return circle;
            }
        }
        public IEnumerator Circle()
        {
            while (true)
            {
```

```

        Console.WriteLine("Circle,Move {0}", move);
        if (++move > maxMoves)
        {
            yield break;
        }
        yield return cross;
    }
}
}
class Demo
{
    public static void Main(string[] args)
    {
        var game = new GameMoves();
        IEnumerator enumerator = game.Cross();
        while (enumerator.MoveNext())
        {
            enumerator = enumerator.Current as IEnumerator;
        }
        Console.ReadLine();
    }
}

```

运行结果:



```

c:\ file:///D:/VSPProject/Test/Test/bin/De
Cross,Move 0
Circle,Move 1
Cross,Move 2
Circle,Move 3
Cross,Move 4
Circle,Move 5
Cross,Move 6
Circle,Move 7
Cross,Move 8

```

元组

数组合并了相同类型的对象，而元组(Tuple)合并了不同类型的对象。元组起源于函数编程语言(如 F#)。

.NET3 定义了 8 个泛型 Tuple 类和一个静态 Tuple 类，他们用作元组的工厂。这里的不同泛型 Tuple 类支持不同数量的元素。例如，Tuple<T1>包含一个元素，Tuple<T1,T2>包含两个元素，以此类推。

Example: Tuple<>仅受.NET4.0 支持。

```
using System;
```

```

namespace Test.Professional
{
    class TupleExample
    {
        public static Tuple<int, int> Divide(int divided, int divisor)
        {
            int result = divided / divisor;
            int reminder = divided % divisor;
            return Tuple.Create<int, int>(result, reminder);
        }
        public static void Main(string[] args)
        {
            var result = Divide(5, 2); //使用 var 推断结果
            Tuple<int, int> mytuple = Divide(6, 2);
            Console.WriteLine("Result of division: {0}, reminder: {1}", mytuple.Item1,
mytuple.Item2);
            Console.ReadLine();
        }
    }
}

```

方法 `Divide()` 返回包含两个成员的元组 `Tuple<int,int>`。泛型类的参数定义了成员的类型，它们都是整数。元组用静态 `Tuple` 类的静态 `Create()` 方法创建。`Create()` 方法的泛型参数定义要实例化的元组类型。新建的元组用 `result` 和 `reminder` 变量初始化，返回这两个变量相除的结果。

结构比较

数组和元组都实现接口 `IStructuralEquatable` 和 `IStructuralComparable`。这两个接口都是 .NET4.0 新增的，不仅可以比较引用，还可以比较内容。这些接口都是显示实现的，所以在使用时需要把数组和元组强制转换为这个接口。`IStructuralEquatable` 接口用于比较两个元组或数组是否有相同的内容，`IStructuralComparable` 接口用于给元组或数组排序。

Example

```

using System;
using System.Collections.Generic;

namespace Test.Professional
{
    class Person : IEquatable<Person>
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}

```

```

public override string ToString()
{
    return String.Format("{0} , {1} {2}", Id, FirstName, LastName);
}
public override bool Equals(object obj)
{
    if (obj == null)
    {
        throw new ArgumentNullException("obj");
    }
    return Equals(obj as Person);
}
public override int GetHashCode()
{
    return Id.GetHashCode();
}
public bool Equals(Person other)
{
    if (other == null)
    {
        throw new ArgumentNullException("other");
    }
    else
    {
        return this.Id == other.Id && this.FirstName == other.FirstName &&
this.LastName == other.LastName;
    }
}
}
class IStructuralEquatableExample
{
    public static void Main(string[] args)
    {
        var janet = new Person { FirstName = "Janet", LastName = "Jackson" };
        Person[] person1 ={
            new Person
            {
                FirstName="Michael",
                LastName="Jackson"
            },
            janet
        };
        Person[] person2 ={
            new Person

```



```

        {
            FirstName="Michael",
            LastName="Jackson"
        },
        janet
    };

    if (person1 != person2)
    {
        Console.WriteLine("Not the same references!");
    }
    Console.ReadLine();
}
}

```

对于上例说明 `IStructuralEquatable` 接口的示例，使用实现 `IEquatable` 接口的 `Person` 类。`IEquatable` 接口定义了一个强类型转化的 `Equals()` 方法，以比较 `FirstName` 和 `LastName` 属性的值。

先建立了两个包含 `Person` 项的数组。这两个数组通过变量名 `janet` 包含相同的 `Person` 对象和两个内容不同 `Person` 对象。比较运算符“`!=`”返回 `true`，因为这其实是两个变量 `person1` 和 `person2` 引用的两个不同数组。因为 `Array` 类没有重写一个带参数的 `Equals()` 方法，所以用“`==`”运算符比较引用也会得到相同的结果，级这两个变量不相同。

对于 `IStructuralEquatable` 接口定义的 `Equals()` 方法，它的第一个参数是 `object` 类型，第二个参数是 `IEqualityComparer` 类型。调用这个方法时，通过传递一个实现了 `IEqualityComparer<T>` 的对象，就可以定义如何进行比较。通过 `EqualityComparer<T>` 类完成 `IEqualityComparer` 的一个默认实现。这个实现检查该类型是否实现了 `IEquatable` 接口，并调用 `IEquatable.Equals()` 方法。如果该类型没有实现 `IEquatable`，就调用 `Object` 基类中的 `Equals()` 方法进行比较。

Example:

```

using System;
using System.Collections;
using System.Collections.Generic;

namespace Test.Professional
{
    class Person : IEquatable<Person>
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public override string ToString()
        {
            return String.Format("{0} , {1} {2}", Id, FirstName, LastName);
        }
    }
}

```

```

public override bool Equals(object obj)
{
    if (obj == null)
    {
        throw new ArgumentNullException("obj");
    }
    return Equals(obj as Person);
}
public override int GetHashCode()
{
    return Id.GetHashCode();
}
public bool Equals(Person other)
{
    if (other == null)
    {
        throw new ArgumentNullException("other");
    }
    else
    {
        return this.Id == other.Id && this.FirstName == other.FirstName &&
this.LastName == other.LastName;
    }
}
}
class IStructuralEquatableExample
{
    public static void Main(string[] args)
    {
        var janet = new Person { FirstName = "Janet", LastName = "Jackson" };
        Person[] person1 ={
            new Person
            {
                FirstName="Michael",
                LastName="Jackson"
            },
            janet
        };
        Person[] person2 ={
            new Person
            {
                FirstName="Michael",
                LastName="Jackson"
            },

```

```

                                janet
                                };
                                if (person1 != person2)
                                {
                                    Console.WriteLine("Not the same references!");
                                }
                                if ((person1 as IStructuralEquatable).Equals(person2,
EqualityComparer<Person>.Default))
                                {
                                    Console.WriteLine("the same content!");
                                }
                                Console.ReadLine();
                            }
                        }
                    }
                }
            }
        }
    }
}

```

Person 实现 IEquatable<Person>, 在此过程中比较对象的内容, 而数组的确包含相同的内容。

```

if ((person1 as IStructuralEquatable).Equals(person2, EqualityComparer<Person>.Default))
{
    Console.WriteLine("the same content!");
}

```

下面看看如何对元组执行相同的操作。这里创建了两个内容相同的元组实例。当然, 因为引用 t1 和 t2 引用了两个不同的对象, 所以比较运算符 “!=” 返回 true:

```

var t1 = Tuple.Create<int, string>(1, "Stephanie");
var t2 = Tuple.Create<int, string>(1, "Stephanie");
if (t1 != t2)
{
    Console.WriteLine("Not the same reference to the Tuple");
}

```

Tuple<>类提供了两个 Equals()方法: 一个重写了 Object 基类中的 Equals()方法, 并把 object 作为参数, 第二个由 IStructuralEqualityComparer 接口定义, 并把 object 和 IEqualityComparer 作为参数。可以给第一个方法传送给另一个元组, 如下所示。这个方法使用 EqualityComparer<object>.Default 获取一个 ObjectEqualityComparer<Object>, 以进行比较。这样, 就会调用 Object.Equals()方法比较元组的每一项。如果每一项都返回 true, Equals()方法的最终结果就是 true, 这里因为 int 和 string 值都相同, 所以返回 true。

```

if (t1.Equals(t2))
{
    Console.WriteLine("the same content!");
}

```

还可以使用类 TupleComparer 创建一个自定义的 IEqualityComparer, 如下所示。这个类实现可 IEqualityComparer 接口的两个方法 Equals()和 GetHashCode():

```

Example:
using System;

```

```

using System.Collections;
using System.Collections.Generic;

namespace Test.Professional
{
    class TupleComparer : IEqualityComparer
    {
        public new bool Equals(object x, object y)
        {
            return x.Equals(y);
        }
        public int GetHashCode(object obj)
        {
            return obj.GetHashCode();
        }
    }
    class TupleComparerExample
    {
        public static void Main(string[] args)
        {
            Tuple<int, string> t1 = new Tuple<int, string>(1, "Will");
            Tuple<int, string> t2 = new Tuple<int, string>(1, "Will");
            TupleComparer comparer = new TupleComparer();
            if (comparer.Equals(t1,t2))
            {
                Console.WriteLine("The same content!");
            }
            Console.ReadLine();
        }
    }
}

```

使用 TupleComparer，给 Tuple<T1,T2>类的 Equals()方法传递一个新的实例。Tuple 类的 Equals()方法为要比较的每一项调用 TupleComparer 的 Equals()方法。所以，对于 Tuple<T1,T2>类，要调用两次 TupleComparer，以检查所有项是否相等：(代码有误)

```

if (t1.Equals(t2,new TupleComparer()))
{
    Console.WriteLine("The same content!");
}

```

运算符和类型强制转换

运算符

类别	运算符
算术运算符	+ - * / %
逻辑运算符	& ^ ~ && !
字符串连接运算符	+
增量和减量运算符	++ --
移位运算符	<< >>
比较运算符	== != <> <= >=
赋值运算符	= += -= *= /= %= &= = ^= <<= >>=
成员访问运算符(用于对象和结构)	.
索引运算符(用于数组和索引器)	[]
类型转换运算符	()
条件运算符(三元运算符)	?:
委托连接和删除运算符	+ -
对象创建运算符	new
类型信息运算符	sizeof is typeof as
溢出异常控制运算符	checked unchecked
间接寻址运算符	[]
名称空间别名限定符	::
空合并运算符	??

可空类型和运算符

对于布尔类型,可以给它指定 `true` 或 `false` 值。但是,要把该类型的值定义为 `undefined`,该怎么办? 此时使用可空类型可以给应用程序提供一个独特的值。如果在程序中使用可空类型,就必须考虑 `null` 值在与各种运算符一起使用时的影响。通常可空类型与一元或二元运算符一起使用时,如果其中一个操作数或两个操作数都是 `null`,其结果就是 `null`。例如:

```
int? a=null;
int? b=a+4; //b=null
int? c=a*5; //c=null
```

但是在比较可空类型时,只要有一个操作数是 `null`,比较的结果就是 `false`。即不能因为一个条件是 `false`,就认为该条件的对立面是 `true`,这在使用非可空类型的程序中很常见。例如:

```
int? a = null;
int? b = -5;
if (a >= b)
    Console.WriteLine("a>=b");
else
```

```
Console.WriteLine("a<b"); //a<b
```

Tips:

- null 值的可能性表示，不能随意合并表达式中的可空类型和非可空类型。

合并空运算符

空合并运算符(??)提供了一种快捷方式，可以在处理可空类型和引用类型时表示 null 可能的值。这个运算符放在两个操作数之间，第一个操作数必须是一个可空类型或引用类型；第二个操作数必须与第一个操作数的类型相同，或者可隐含的转换为第一个操作数的类型。空合并运算符的计算如下：

- 如果第一个操作数不是 null，整个表达式就等于第一个操作数的值。
- 如果第一个操作数是 null，整个表达式就等于第二个操作数的值。

例如：

```
int? a = null;
int b;
b = a ?? 10; //b has the value 10
Console.WriteLine(b);
a = 3;
b = a ?? 10; //b has the value 3
Console.WriteLine(b);
```

如果第二个操作数不能隐含地转换为第一个操作数的类型，就生成一个编译错误。

运算符的优先级

下表显示了 C# 运算符的优先级，顶部的运算符有最高的优先级(即在包含多个运算符的表达式中，最先计算该运算符)：

组	运算符
初级运算符	() . x++ x-- new typeof sizeof checked unchecked
一元运算符	+ - ! ~ ++x --x 数据类型强制转换
乘/除运算符	* / %
加/减运算符	+ -
移位运算符	<< >>
关系运算符	< > <= >= is as
比较运算符	== !=
按位 AND 运算符	&
按位 XOR 运算符	^
按位 OR 运算符	
布尔 AND 运算符	&&
布尔 OR 运算符	
条件运算符	?:
赋值运算符	= += -= *= /= %= &= = ^= <<= >>= >>>=

Tips:

- 在复杂的表达式中，应避免利用运算符优先级来生成正确的结果。使用圆括号指定运算

符的执行顺序，可以代码更整洁，避免出现潜在的冲突。

类型的安全性

中间语言 IL 可以对其代码强制实现强类型安全性。强类型化支持 .NET 提供的许多服务，包括安全性和语言的交互性。因为 C# 这种语言会编译为 IL，所以 C# 也是强类型。此外，这说明数据类型并不总是无缝地可互换的。

Tips:

- C# 也支持不同引用类型之间的互换，在与其它类型相互转换时还允许定义所创建的数据类型的行为方式。
- 泛型可以避免对一些常见的情形进行类型转换。

类型转换

Example:

```
byte value1 = 10;
byte value2 = 25;
byte total = value1 + value2;
Console.WriteLine(total);
```

在试图编译这些代码时，会得到一条错误消息：

Cannot implicitly convert type 'int' to 'byte'

上述问题是，我们把两个 byte 类型数据加在一起时，应返回 int 类型结果，而不是另一个 byte。这是因为 byte 包含的数据只能为 8 位，所以把两个 byte 类型数据加在一起，很容易得到不能储存在单个字节中的值。如果要把结果储存在一个 byte 变量中，就必须把它转回一个 byte。C# 支持两种转换方式：隐式转换和显示转换。

隐式转换

只要能保证值不会发生任何变化，类型转换就可以自动(隐式)进行。这就是前面代码失败的原因：试图从 int 转换为 byte，而可能丢失了 3 个字节的数据。编译器不会告诉我们该怎么做，除非我们明确告诉它这就是我们希望的！如果在 long 类型变量中而不是 byte 类型变量中储存结果，就不会有问题了：

```
byte value1 = 10;
byte value2 = 25;
long total = value1 + value2;
Console.WriteLine(total);
```

这是因为 long 类型变量包含的数据字节比 byte 类型多，所以没有丢失数据的危险，在这些情况下，编译器会很顺利地转换，我们也不需要显示提出要求。

下表列出了 C# 支持的隐式类型转换。

源类型	目的类型
sbyte	short int long float double decimal BigInteger
byte	short ushort int uint long ulong float double decimal BigInteger
short	int long float double decimal BigInteger

ushort	int uint long ulong float double decimal BigInteger
int	long float double decimal BigInteger
uint	long ulong float double decimal BigInteger
long、ulong	float double decimal BigInteger
float	double BigInteger
char	ushort int uint long ulong float double decimal BigInteger

注意，只能从较小的整数类型隐式地转化为较大的整数类型，不能从加大的整数类型隐式地转换为较小的整数类型。也可以在整数和浮点数之间转换，然而，其规则略有不同。尽管可以在相同大小的类型之间转换，如 `int/uint` 转换为 `float`，`long/ulong` 转换为 `double`，但是也可以从 `long/ulong` 转换回 `float`。这样做可能会丢失 4 个字节的数据，但这仅表示得到 `float` 值比使用 `double` 得到的值精度低，编译器认为这是一种可以接受的错误，而其值的大小不会受到影响。无符号的变量可以转换为有符号的变量，只要无符号的变量值的大小在有符号的变量的范围之内即可。

在隐式地转换为值类型时，可空类型需要考虑其它因素：

- 可空类型隐式地转换为其它可空类型，应遵循上表的非可空类型的转换规则。即 `int?` 隐式地转换为 `long?`、`float?`、`double?` 和 `decimal?`。
- 非可空类型隐式地转换为可空类型也遵循上表的转换规则，即 `int` 隐式地转化为 `long?`、`float?`、`double?` 和 `decimal?`。
- 可空类型不能隐式地转换为非可空类型，此时必须进行显示转换。这是因为可空类型的值可以是 `null`，但非可空类型不能表示这个值。

显示转换

有许多场合不能隐式地转换类型，否则编译器会报告错误。下面是不能进行隐式转换的一些场合：

- `int` 转换为 `short`——会丢失数据。
- `int` 转换为 `uint`——会丢失数据。
- `uint` 转换为 `int`——会丢失数据。
- `float` 转换为 `int`——会丢失小数点后面的所有数据。
- 任何数字类型转换为 `char`——会丢失数据。
- `decimal` 转换为任何数字类型——因为 `decimal` 类型的内部结构不同于整数和浮点数。
- `int?` 转换为 `int`——可空类型的值可以是 `null`。

但是，可以使用 `cast` 显示地执行这些转换。在把一种类型强制转换为另一种类型时，要有意地迫使编译器进行转换。类型强制转换的一般语法如下：

```
long val = 3000;
int i = (int)val;
```

这表示，把强制转换的目标类型名放在要转换的值之前的圆括号中。

这种强制转换是一种比较危险的操作，即使在从 `long` 转换为 `int` 这样简单的类型强制转换过程中，如果原来 `long` 的值比 `int` 的最大值还打，就会出问题：

```
long val = 3000000000;
int i = (int)val; //An invalid cast. The maximum int is 2147483647
```

最好假定显示转换强制转换不会给出希望的结果，**C#** 提供了一个 `checked` 运算符。使用它可以测试操作是否会导致算术溢出。使用 `checked` 运算符可以检查类型强制转换是否安全，如果不安全，就会迫使运行库抛出一个溢出异常：


```
long val = 3000000000;
int i = checked((int)val);
```

小心地使用显式的类型强制转换,就可以把简单值类型的任何实例转换为几乎任何其他类型。但在进行显式的类型转换时有一些限制,就值类型来说,只能在数字、char 类型和 Enum 型之间转换。不能直接把布尔型强制转换为其他类型,也不能把其他类型转换为布尔型。如果需要在数字和字符串之间转换,就可以使用.NET类库中提供的一些方法。Object类实现了一个ToString()方法,该方法在所有的.NET预定义类型中都进行了重写,并返回对象的字符串表示:

```
int i = 10;
string s = i.ToString();
```

同样,如果需要分析一个字符串,以检索yield数字或布尔值,就可以使用所有预定义值类型都支持的Parse()方法:

```
string s = "100";
int i = int.Parse(s);
Console.WriteLine(i + 50); //Add 50 to prove it is really an int
```

注意,不过不能转换字符串(例如,要把字符串 Hello 转换为一个整数),Parse()方法就会通过抛出一个异常注册一个错误。

装箱和拆箱

C#数据类型可以分为在栈上分配内存的值类型和在堆上分配内存的引用类型。

装箱(boxing)和拆箱(unboxing)可以把值类型转换为引用类型,并把引用类型转换回值类型。即把值强制转换为 object 类型。装箱用于描述把一个值类型转换为引用类型。运行库会为堆上的对象创建一个临时的引用类型“箱子”。

该转换可以隐式地进行,也还可以显示地进行转换:

```
string s = 10.ToString(); //隐式进行装箱
int myIntNum = 10;
object myObject = myIntNum; //Box the int
int myUboxNum = (int)myObject; //Unbox it back into an int
```

只能对以前进行装箱的变量进行拆箱。当 myObject 不是装箱后的 int 类型时,如果执行最后一行代码,就会在运行期间抛出一个异常。

在拆箱时,必须非常小心,确保得到的值变量有足够的空间存储拆箱的值中所有的字节。例如,C#的 int 类型只有 32 位,所以把 long 类型值(64 位)拆箱为 int 时,会导致一个 InvalidCastException 异常:

```
long myIntNum = 10;
object myObject = myIntNum; //Box the int
int myUboxNum = (int)myObject; //Unbox it back into an int An Error
```

比较对象的相等性

对象相等的机制有所不同,这取决于比较的是引用类型(类的实例)的比较还是值类型(基元数据类型,结构或枚举的实例)。

比较引用类型的相等性

`System.Object` 定义了 3 个不同的方法，来比较对象的相等性：`ReferenceEqual()`和两个版本的 `Equals()`。再加上比较运算符(`==`)，实际上有 4 中进行比较相等性的方式。

`ReferenceEquals()`方法：

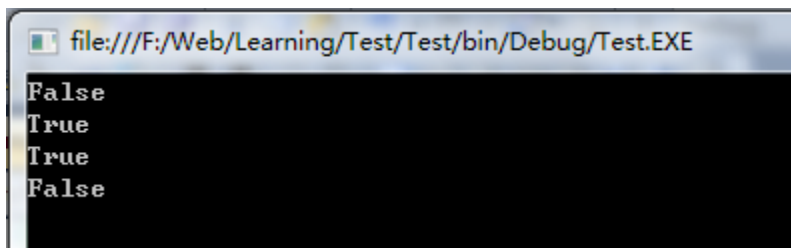
`ReferenceEquals()`是一个静态方法，测试两个引用是否引用类的同一个实例，特别是两个引用是否包含内存中的相同地址。作为静态方法，它不能重写，所以 `System.Object` 的实现代码保持不变。如果提供的两个引用引用同一个对象实例，则 `ReferenceEquals()`总是返回 `true`；否则就返回 `false`。但是它认为 `null` 就等于 `null`。

Example:

```
using System;
```

```
namespace Test.Professno1
{
    class Things
    {
        string name;
    }
    class ReferenceEqualsExample
    {
        public static void Main(string[] args)
        {
            Things x, y, z;
            x = new Things ();
            y = new Things ();
            z = x;
            Console.WriteLine (System.Object.ReferenceEquals(x, y));
            Console.WriteLine (ReferenceEquals(x, z));
            Console.WriteLine (ReferenceEquals(null, null));
            Console.WriteLine (ReferenceEquals(x, null));
            Console.In.ReadLine();
        }
    }
}
```

运行结果：



虚拟版本的 `Equals()`方法

`Equals()`虚拟版本的 `System.Object` 实现代码也可以比较引用。但因为这个方法是虚拟的，所以可以在自己的类中重写它，从而按值来比较对象。特别是如果希望类的实例用作字典中

的键，就需要重写这个方法，以比较相关值。否则，根据重写 `Object.GetHashCode()` 的方式，包含对象的字典类要么不工作，要么工作的效率非常低。在重写 `Equals()` 方法时要注意，重写的代码不会抛出一场。同理，这是因为如果抛出异常，字典类就会出问题，一些在内部调用这个方法的 .NET 基类也可能出问题。

静态的 `Equals()` 方法

`Equals()` 的静态版本与其虚拟实例版本的作用相同，其区别是静态版本带有两个参数，并对它们进行相等性比较。这个方法可以处理两个对象中有一个是 `null` 的情况，因此，如果一个对象可能是 `null`，这个方法就可以抛出一场，提供额外的保护。静态重载版本首先要检查它传递的引用是否为 `null`。如果他们都是 `null`，就返回 `true` (因为 `null` 与 `null` 相等)。如果只有引用是 `null`，它就返回 `false`。如果两个引用实际上引用了某个对象，它就调用 `Equals()` 的虚拟实例版本。这表示在重写 `Equals()` 的实例版本时，其效果相当于也重写了静态版本。

比较运算符 `==`

最好将比较运算符看作严格的值比较和严格的引用比较之间的中间选项。在大多数情况下，下面的代码表示正在比较引用：

```
bool b = (x==y); //x,y object references
```

但是，如果把一些类看作值，其含义就会比较只管，这是可以接受的。在这些情况下，最好重写比较运算符，以执行值的比较。`System.String` 类，`Microsoft` 重写了 `==` 运算符，以比较字符串的内容，而不比较它们的引用。

比较值类型的相等性

在比较值类型的相等性时，采用与引用类型相同的规则：`ReferenceEquals()` 用于比较引用类型，`Equals()` 用于比较值，比较运算符可以看作一个中间项。但最大的区别是值类型需要装箱，才能把它们转换为引用，进而才能对它们执行方法。另外，`Microsoft` 已经在 `System.ValueType` 类中重载了实例方法 `Equals()`，以便对值类型进行合适的相等性测试。如果调用 `sA.Equals(sB)`，其中 `sA` 和 `sB` 是某个结构的实例，则根据 `sA` 和 `sB` 是否在其所有字段中包含相同的值，而返回 `true` 或 `false`。另一方面，在默认情况下，不能对自己的结构重载“`==`”运算符。在表达式中使用 `(sA=sB)` 会导致一个编译错误，除非在有问题的代码中为结构提供了“`==`”的重载版本。

尽管 `System.ValueType` 提供的 `Equals()` 的默认重写版本肯定足以应付绝大多数自定义的结构，但仍可以针对自己的结构再次重写它，以提高性能。另外，如果值类型包含作为字段的引用类型，就需要重写 `Equals()`，以便为这些字段提供合适的语义，因为 `Equals()` 的默认重写版本仅比较它们的地址。

运算符重载

重载不紧紧限于算术运算符，还需要考虑比较运算符 `==`、`<`、`>`、`!=`、`>=`、`<=`。例如，语句 `if(a==b)`。对于类，这个语句在默认状态下会比较引用 `a` 和 `b`。检测这两个引用是否只想内存中的同一个地址，而不是检测两个实例实际上是否包含相同的数据。对于 `string` 类，这种操作就会重写，于是比较字符串实际上就是比较每个字符串的内容。

运算符的工作方式

```
int myInteger=3;
uint myUnsignedInt=2;
long myLong=myInteger + myUnsignedInt;
会发生什么情况:
long myLong=myInteger + myUnsignedInt;
```

编译器知道它需要把两个证书加起来，并把结果赋予一个 `long` 类型变量。调用一恶搞方法把数字加在一起时，表达式 `myInteger + myUnsignedInt` 是一种非常直观和方便的语法。该方法接受两个参数 `myInteger` 和 `myUnsignedInt`，并返回它们的和。所以编译器完成的任务与任何方法调用一样——它会根据参数类型查找最匹配的“+”运算符，这里是两个整数参数的“+”运算符重载。与一般的重载方法一样，预定义的返回类型不会因为编译器所调用方法的哪个版本而影响编译器的选择。在本例中调用的重载方法接受两个 `int` 参数，返回一个 `int`，这个返回值随后会转换为一个 `long`。

运算符重载的示例：Vector(矢量)结构

制作一个结构 `Vector`，来说明运算符重载，这个 `Vector` 结构表示一个三维矢量。

Example:

```
using System;

namespace Test.Professno1
{
    struct Vector
    {
        public double x, y, z;
        public Vector(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
        public Vector(Vector rhs) //复制构造函数
        {
            this.x = rhs.x;
            this.y = rhs.y;
            this.z = rhs.z;
        }
        public override string ToString()
        {
            return "(" + x + ", " + y + ", " + z + ")";
        }
    }
}
```

```

public static Vector operator +(Vector lhs, Vector rhs)
{
    Vector result = new Vector(lhs);
    result.x += rhs.x;
    result.y += rhs.y;
    result.z += rhs.z;
    return result;
}
public static bool operator ==(Vector lhs, Vector rhs)
{
    // return lhs.x.Equals(rhs.x) && lhs.y.Equals(rhs.y) && lhs.z.Equals(rhs.z); //
    可能会引起null.Equals(obj) 异常
    return lhs.x == rhs.x && lhs.y == rhs.y && lhs.z == rhs.z;
}
public static bool operator !=(Vector lhs, Vector rhs)
{
    return !(lhs == rhs);
}
public static int GetHashCode(object vec)
{
    return vec.GetHashCode();
}

}

class OperateOverLoadExample
{
    public static void Main(string[] args)
    {
        Vector vect1, vect2, vect3;
        vect1 = new Vector(1.0, 3.0, 5.0);
        vect2 = new Vector(2.0, 3.0, 6.0);
        vect3 = vect1 + vect2;
        vect1 += vect2;
        Console.WriteLine(vect3);
        Console.WriteLine(vect1);
        Console.WriteLine(vect1 == vect3);
        Console.In.ReadLine();
    }
}

```

运行结果：

```

file:///F:/Web/Learning/Test/Test/bin/Debug/Test.EXE
<3,6,11>
<3,6,11>
True

```

Tips:

- 虽然“+=”一般用作单个运算符，但实际上它对应的操作分为两步：相加和赋值。与C++语言不同，C#不允许重载“=”运算符，但如果重载“+”运算符，编译器就会自动使用“+”运算符的重载来执行“+=”运算符的操作。-=、&=、*=和/=等所有赋值运算符也遵循此规则。
- 比较运算符重载分为3对==和!=、>和<、>=和<=
- C#语言要求成对重载比较运算符。即，如果重载了“==”，也就必须重载“!=”；否则会编译错误。
- 在重载“==”和“!=”时，还必须重载从System.Object中集成的Equals()和GetHashCode()方法，否则会产生一个编译警告。原因是Equals()方法应实现与“==”运算符相同类型的相等逻辑。
- 浅度比较是比较对象是否指向内存中的同一个位置，而深度比较是比较对象的值和属性是否相等。
- 不要通过调用从System.Object中继承的Equals()方法的实例版本来重载比较运算符。如果这么做，在obj是null时判断(objA==objB)，就会产生一个异常，因为.NET运行库会试图判断null.Equals(objB)。采用其他方法(重写Equals()方法以调用比较运算符)比较安全。

可以重载的运算符

并不是所有的运算符都可以重载。可以重载的运算符见下表：

类别	运算符	限制
算数二元运算符	+ * / - %	无
算术一元运算符	+ - ++ --	无
按位二元运算符	& ^ << >>	无
按位一元运算符	! ~ true false	true 和 false 运算符必须成对重载
比较运算符	== != >= < <= >	必须成对重载
赋值运算符	+= -= *= /= >>= <<=	不能显示地重载这些运算符，在重写单个运算符(如+, -, %等)时，它们会被隐式地重写
索引运算符	[]	不能直接重载索引运算符。索引器成员类型允许在类和结构上支持索引运算符
数据类型转换运算符	()	不能直接重载类型强制转换运算符。用户定义的类型强制转换允许定义定制的类型强制转换行为

用户定义的类型强制转换

C#允许进行两种不同数据类型的强制转换：隐式强制转换和显示强制转换。

```
int i=3;
long l=i;           //implicit(隐式强制转换)
short s=(short) l;  //explicit(显示强制转换)
```

C#允许定义自己的数据类型(结构和类，这意味着需要某些工具支持在自定义的数据类型之间进行类型强制转换。方法是把类型强制转换运算符定义为相关类的一个成员运算符，类型强制转换运算符必须标记为隐式或显式，以说明希望如何使用它。我们应遵循与预定义的类型强制转换相同的规则，如果知道无论在源变量中存储什么值，类型强制转换总是安全的，就可以把它定义为隐式强制转换。然而，如果某些数值可能会出错，如丢失数据或抛出异常，就应把数据类型转换定义为显式强制转换。

Tips:

- 如果元数据值会使类型强制转换失败，或者可能会抛出异常，就应把任何自定义类型强制转换定义为显式强制转换。

定义类型强制转换的语法类似于重载运算符。类型强制转换在某种情况下可以看作是一种运算符，其作用是从源类型转换为目标类型。

```
public static implicit operator float(Currency value)
{
    //processing
}
```

运算符的返回类型定义了强制转换操作的目标类型，它有一个参数，即要转换的源对象。这里定义的类型强制转换可以隐式地把 **Currency** 类型的值转换为 **float** 类型。注意，如果数据类型转换声明为隐式，编译器就可以隐式或显示地使用这个转换。如果数据类型转换声明为显式，编译器就只能显式地使用它。与其它运算符重载一样，类型强制转换必须同时声明为 **public** 和 **static**。

实现用户定义类型强制转换

Example:

```
using System;

namespace Test.Professno1
{
    struct Currency
    {
        public uint Dollars;
        public ushort Cents;
        public Currency(uint dollars, ushort cents)
        {
            this.Dollars = dollars;
            this.Cents = cents;
        }
    }
}
```

```

public override string ToString()
{
    return String.Format("$ {0}. {1,-2:00}", Dollars, Cents);
}

public static implicit operator float(Currency value)
{
    return value.Dollars + (value.Cents / 100.0f);
}

public static explicit operator Currency(float value)
{
    uint dollars = (uint)value;
    ushort cents = (ushort)((value - dollars) * 100);
    return new Currency(dollars, cents);
}
}

class SelfConvertExample
{
    public static void Main(string[] args)
    {
        try
        {
            Currency balance = new Currency(50, 35);
            Console.WriteLine(balance);
            Console.WriteLine("balance is " + balance);
            Console.WriteLine("balance is (using ToString) " + balance.ToString());

            float balance2 = balance;
            Console.WriteLine("After convert to float, = " + balance2);

            balance = (Currency)balance2;
            Console.WriteLine("After convert back to Currency, = " + balance);

            Console.WriteLine("Now attempt to convert out of range values of -$50.50
to a Currency: ");
            checked
            {
                balance = (Currency)(-50.50);
                Console.WriteLine("Result is " + balance.ToString());
            }
            Console.ReadLine();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}

```

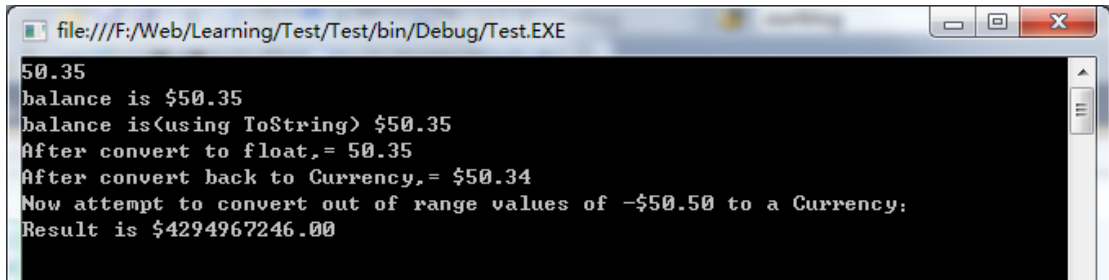


```

        Console.ReadLine();
    }
}
}
}

```

运行结果:



```

file:///F:/Web/Learning/Test/Test/bin/Debug/Test.EXE
50.35
balance is $50.35
balance is(using ToString) $50.35
After convert to float,= 50.35
After convert back to Currency,= $50.34
Now attempt to convert out of range values of -$50.50 to a Currency:
Result is $4294967246.00

```

这个结果表示代码并没有像我们希望的那样工作。首先，从 `float` 转换回 `Currency` 得到一个错误的结果\$50.34，而不是\$50.35。其次，在试图转换明显超出范围的值时，并没有生成异常。

第一个问题是由舍入错误引起的。如果类型强制转换用于把 `float` 转换为 `uint`，计算机就会截去多余的数字，而不是四舍五入它。计算机以二进制方式储存数字，而不是十进制，小数部分 0.35 不能用二进制小数来精确表示(像 1/3 这样的分数不能精确地表示为十进制小数，它应等于循环小数 0.3333)。所以，计算机最后存储了一个略小于 0.35 的值，它可以用二进制格式精确地表示。把该数字乘以 100，就会得到一个小于 35 的数字，它截去了 34 美分。显然在上述结果中，这种由截去引起的错误是很严重的，避免该错误的方式是确保在数字转换过程中执行智能的四舍五入操作。幸运的是，Microsoft 编写了一个类 `System.Convert` 来完成该任务。`System.Convert` 对象包含大量的静态方法来完成各种数字转换，我们需要使用的是 `Convert.ToInt16()`。注意，在使用 `System.Convert` 类的方法时会造成额外的性能损失，所以只应在需要时才使用它们。

下面看看为什么没有抛出期望的溢出异常，此处的问题是溢出异常实际发生的位置根本不在 `Main()` 进程汇中——它实在强制转换运算符的代码中发生的，该代码在 `Main()` 方法中调用，而且没有标记为 `checked`。

其解决办法是确保类型强制转换本身也在 `checked` 环境下进行。修改代码如下：

```

public static explicit operator Currency(float value)
{
    checked
    {
        uint dollars = (uint)value;
        ushort cents = Convert.ToUInt16((value - dollars) * 100);
        return new Currency(dollars, cents);
    }
}

```

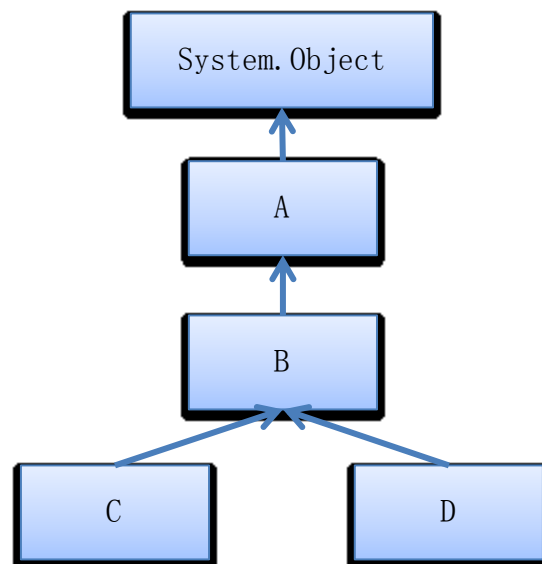
最后运行结果:

```
file:///F:/Web/Learning/Test/Test/bin/Debug/Test.EXE
50.35
balance is $50.35
balance is<using ToString> $50.35
After convert to float,= 50.35
After convert back to Currency,= $50.35
Now attempt to convert out of range values of -$50.50 to a Currency:
算术运算导致溢出。
```

类之间的类型强制转换

类型转换不一定会涉及任何简单的数据类型。定义不同结构或类的实例之间的类型强制转换是完全合法的，但有两个限制：

- 如果某个类派生自另一个类，就不能定义这两个类之间的类型强制转换(这些类型的类型转换已经存在)。
- 类型强制转换必须在源数据类型或目标数据类型的内部定义。



假如有上图所示的类层次结构。在这种情况下，在 A、B、C 或 D 之间唯一合法的自定义类型强制转换就是类 C 和 D 之间的转换，因为这些类并没有相互派生。代码如下(假定希望类型强制转换是显示的，这是在用户定义的类之间定义类型强制转换的通常情况)：

```
public static explicit operator D(C value)
{
    //and so on
}
public static explicit operator C(D value)
{
    //and so on
}
```

对于这些类型强制转换，可以选择放置定义的地方——在 C 的类定义内部，或者在 D 的类定义内部，但不能在其它地方定义。C# 要求把类型强制转换的定义放在源类(或结构)或目标类(或结构)的内部。它的副作用是不能定义两个类之间的类型强制转换，除非至少可以

编辑其中一个类的源代码。这是因为，这样可以防止第三方把类型强制转换引入类中。

一旦在一个类的内部定义类类型强制转换，就不能在另一个类中定义相同的类型强制转换。显然，对于每一种转换只能有一种类型强制转换，否则编译器就不知道该选择哪个类型强制转换了。

基类和派生类之间的强制转换

首先看源和目标的数据类型都是引用类型的情况。考虑两个类 `MyBase` 和 `MyDerived`，其中 `MyDerived` 直接或间接派生自 `MyBase`。

首先是从 `MyDerived` 到 `MyBase` 的转换，代码如下(假定可以使用构造函数)：

```
MyDerived derivedObject = new MyDerived();
```

```
MyBase baseCopy = derivedObject;
```

在本例中，是从 `MyDerived` 隐式地强制转换为 `MyBase`。这是可以行，因为对类型 `MyBase` 的任何引用都可以引用 `MyBase` 类的对象或派生自 `MyBase` 的对象。在 OO 编程中，派生类的实例实际上是基类的实例，但是加上了一些额外的信息。在基类上定义的所有函数和字段也都在派生类上定义了。

下面看另一种方式，编写下面的代码：

```
MyBase derivedObject = new MyDerived();
```

```
MyBase baseObject = new MyBase();
```

```
MyDerived derivedCopy1 = (MyDerived) derivedObject;    //OK
```

```
MyDerived derivedCopy2 = (MyDerived) baseObject;        //Throws exception
```

上面的代码都是合法的 C# 代码(从句法的角度来看，它是合法的)，它说明了把基类强制转换为派生类。但是，在执行最后一条语句会抛出一个异常。在进行类型强制转换时，会检查被引用的对象。因为基类引用原则上可以引用一个派生类的实例，所以这个对象可能是要强制转换的派生类的一个实例。如果是这样，强制转换就会成功，派生的引用被设置为引用这个对象。但如果有问题的对象不是派生类(或者派生于这个类的其它类)的一个实例，强制转换就会失败，并抛出一个异常。

注意，编译器已经提供了基类和派生类之间的强制转换，这种转换实际上并没有对有问题对象进行任何数据转换。如果要进行的转换是合法的，它们也仅是把新引用设置为对对象的引用。这些强制转换在本质上与用户定义的强制转换不同。例如，在 `Currency` 示例中，我们定义了 `Currency` 结构和 `float` 数之间的强制转换。在 `float` 到 `Currency` 的强制转换中，实际上实例化了一个新的 `Currency` 结构，并用要求的值初始化它。在基类和派生类之间的预定义强制转换则不是这样。如果实际上要把 `MyBase` 实例转换为真是的 `MyDerived` 对象，该对象的值根据 `MyBase` 实例的内容来确定，就不能使用类型强制转换语法。最合适的选项通常是定义一个派生类的构造函数，它以基类的实例作为参数，让这个构造函数完成相关的初始化：

```
class DerivedClass : BaseClass
{
    public DerivedClass(BaseClass rhs)
    {
        //initialize object from the base instance
    }
    //etc.
}
```

装箱和拆箱数据类型强制转换

基类和派生类都是引用类型，其数据类型强制转换的规则也适用用强制转换值类型，尽管在转换值类型时，不可能仅仅复制引用，还必须复制一些数据。

当然，不能从结构或基元值类型中派生。所以基本结构和派生结构之间的强制转换总是基元类型或结构与 `System.Object` 之间的转换(理论上可以在结构和 `System.ValueType` 之间进行强制转换，但一般很少这么做)。

从结构(或基本类型)到 `object` 的强制转换总是一种隐式的强制转换，因为这种的强制转换是从派生类型到基本类型的转换，及装箱过程。如 `Currency` 结构：

```
Currency balance = new Currency(40 , 0);
object baseCopy = balance;
```

在执行上述隐式的强制转换时，`balance` 的内容被复制到堆上，放在一个装箱的对象上，`baseCopy` 对象引用被设置为该对象。实际上在后台发生的情况是：在最初定义 `Currency` 结构时，`.NETFramework` 隐式地提供另一个(隐藏的)类，即装箱的 `Currency` 类，它包含与 `Currency` 结构相同的所有字段，但它是一个引用类型，存储在堆上。无论定义的这个值类型是一个结构，还是一个枚举，定义它时都存在类型的装箱引用类型，对应于所有的基元值类型，如 `int`、`double` 和 `uint` 等。不能也不必在源代码中直接通过编程访问某些装箱类，但在把一个值类型强制转换为 `object` 时，它们是在后台工作的对象。在隐式地把 `Currency` 转换为 `object` 时，会实例化一个装箱的 `Currency` 实例，并用 `Currency` 结构中的所有数据进行初始化。在上面的代码中，`baseCopy` 对象引用的就是这个已装箱的 `Currency` 实例。通过这种方式，就可以实现从派生类型到基本类型的强制转换，并且，值类型的语法与引用类型的语法一样。

强制转换的另一种方式称为拆箱。与在基本引用类型和派生引用类型之间的强制转换一样，这是一种显式的强制转换,因为如果要强制转换的对象不是正确的类型，就会抛出一个异常：

```
object derivedObject = new Currency (40 , 0);
object baseObject = new object();
Currency derivedCopy1 = (Currency)derivedObject;    //OK
Currency derivedCopy2 = (Currency)baseObject;        // Exception thrown
```

上述代码的工作方式与前面的引用类型中的代码一样。把 `derivedObject` 强制转换为 `Currency` 会成功进行，因为 `derivedObject` 实际上引用的是装箱 `Currency` 实例——强制转换的过程是把已装箱的 `Currency` 对象的字段复制到一个新的 `Currency` 结构中。第二种强制转换会失败，因为 `baseObject` 没有引用已装箱的 `Currency` 对象。

在使用装箱与拆箱时，这两个过程都把数据复制到新装箱或拆箱的对象上。

多重类型强制转换

在定义类型强制转换时必须考虑的一个问题是，如果在进行要求的数据类型转换时，`C#` 编译器没有可用的直接强制转换方式，`C#` 编译器就会寻找一种转换方式，把几种强制转换合并起来。例如，在 `Currency` 结构中，假定编译器遇到下面几行代码：

```
Currency balance = new Currency(10 , 50);
long amount = (long) balance;
double amountD = balance;
```

首先初始化一个 `Currency` 实例，再把它转换为一个 `long` 数。问题是不能定义这样的强

制转换。但是这段代码仍可以编译成功。因为编译器知道我们已经定义一个从 `Currency` 到 `float` 的隐式强制转换，而且它知道如何显示地从 `float` 转换为 `long`。所以它会把这行代码便意味中间语言(IL)代码，IL 代码首先把 `balance` 转换为 `float`，再把结果转换为 `long`。把 `balance` 转换为 `double` 类型时，在上述代码的最后一行中也是这样。因为从 `Currency` 到 `float` 的强制转换和从 `float` 到 `double` 的预定义强制转换都是隐式的，所以可以在编写代码时把这种转换当作一种隐式转换。如果要显示地指定强制转换过程，则可以编写如下代码：

```
Currency balance = new Currency(10 , 50);
```

```
long amount = (long)(float) balance;
```

```
double amountD = (double)(float) balance;
```

但是，在大多数情况下，这会使代码变得比较复杂，因此是不必要的。相比之下，下面的代码会产生一个编译错误：

```
Currency balance = new Currency(10 , 50);
```

```
long amount = balance;
```

原因是编译器可以找到最佳匹配的转换仍是首先转换为 `float`，再转换为 `long`。但从 `float` 到 `long` 的转换需要显示地指定。

并非所有这些转换都会带来太多的麻烦。毕竟 转换的规则非常直观，主要是为了防止在开发人员不知情的情况下丢失数据。但是，在定义类型强制转换时如果不小心，编译器就有可能指定一条导致不期望的结果的路径。例如，假定编写 `Currency` 结构的其他小组成员要把一个 `uint` 转换为 `Currency`，其中该 `uint` 中包含了美分的总数(美分不是美元，因为我们不希望丢掉美元的小数部分)。为此应编写如下代码来实现强制转换：

```
public static implicit operator Currency (uint value)
```

```
{
```

```
    return new Currency(value/100u , (uint)(value%100) );
```

```
} //Do not do this!
```

注意，在这段代码中，第一个 100 后面的 `u` 可以确保把 `value/100u` 解释为一个 `uint` 数。如果写成 `value/100`，编译器就会把它解释为一个 `int` 类型的值，而不是 `uint` 类型的值。

在这段代码中清楚地标记了“Do not do this!”。下面说明其原因。看看下面的代码段，它把包含 350 的一个 `uint` 转换为一个 `Currency`，再转换为 `uint`。那么在执行完这段代码后，`bal2` 中又将包含什么？

```
uint bal = 350;
```

```
Currency balance = bal;
```

```
uint bal2 = (uint) balance;
```

答案不是 350，而是 3！而且这是符合逻辑的。我们把 350 隐式地转换为 `Currency`，得到的结果是 `balance.Dollars=3`，`balance.Cents=50`。然后编译器进行通常的操作，为转换会 `uint` 指定最佳路径。`balance` 最终会被隐式地转换为 `float` 类型(其值为 3.5)，然后显示地转换为 `uint` 类型，其值为 3。

当然，在其它示例中，转换为另一种数据类型后，再转换回来有时会丢失数据。例如，把包含 5.8 的 `float` 数转换为 `int` 数，再转换回 `float` 数，会丢失数字的小数部分，得到 5，但原则上丢失数字的小数部分和一个证书被大于 100 的数整除的情况略有区别。`Currency` 现在成了一种相当危险的类，它会对整数进行一些奇怪的操作。

问题是，在转换过程中如何解释整数存在冲突。从 `Currency` 到 `float` 的强制转换为把整数 1 解释为 1 美元，但从 `uint` 到 `Currency` 的强制转换会把这个整数解释为 1 美分，这是很糟糕的一个示例。如果希望类易于使用，就应该确保所有的强制转换都按一种互相兼容的方式执行，即这些转换直观上应得到相同的结果。在本例中，显然要从新编写从 `uint` 到 `Currency`

的强制转换，把整数 1 解释为 1 美元：

```
public static implicit operator Currency (uint value)
{
    return new Currency(value , 0);
}
```

偶尔也会觉得这种新的转换方式可能根本不需要。但实际上这种转换方式可能非常有用。没有这种强制转换，编译器在执行从 `uint` 到 `Currency` 的转换时，就只能通过 `float` 来进行。此时直接转换的效率要高得多，所以进行这种额外的转换强制会提高性能，但需要确保它的结果与通过 `float` 进行转换得到的结果相同。在其他情况下，也可以为不同的预定义数据类型分别定义强制转换，让更多的转换隐式地执行，而不是显式地执行，但本例不是这样。

测试这种强制转换是否兼容，应确定无论使用什么转换路径，它是否都能得到相同的结果(而不是像在从 `float` 到 `int` 的转换过程中那样丢失数据)。`Currency` 类就是一个很好的示例。看看下面的代码：

```
Currency balance = new Currency(50 , 35);
ulong bal = (ulong) balance;
```

目前，编译器只能采用一种方式来完成这个转换：把 `Currency` 隐式地转换为 `float`，再显式地转换为 `ulong`。从 `float` 到 `ulong` 的转换需要显式转换，本例就显式指定了这个转换，所以编译是成功的。

但假定要添加另一种强制转换，从 `Currency` 隐式地转换为 `uint`，就需要修改 `Currency` 结构，添加从 `uint` 到 `Currency` 的强制转换和从 `Currency` 到 `uint` 的强制转换。这段代码可以作为 `Currency2` 示例：

```
public static implicit operator Currency (uint value)
{
    return new Currency(value , 0);
}
public static implicit operator uint (Currency value)
{
    return value.Dollars;
}
```

现在，编译器从 `Currency` 转换到 `ulong` 可以使用另一条路径：先从 `Currency` 隐式地转换为 `uint`，再隐式地转换为 `ulong`。该采用哪条路径？`C#` 有一些严格的规则(参见 `MSDN` 文档)，告诉编译器如何确定哪条是最佳路径。但最好自己设计类型强制转换，让所有的转换路径都得到相同的结果(但没有精确度的损失)，此时编译器选择哪条路径就不重要了(在本例中，编译器会选择 `Currency→uint→ulong` 路径，而不是 `Currency→float→ulong` 路径)。

结论是：如果方法调用带有多个重载方法，并要给该方法传送参数，而该参数的数据类型不匹配任何重载方法，就可以迫使编译器确定使用哪些强制转换方式进行数据转换，从而决定使用哪个重载方法(并进行相应的数据转换)。当然，编译器总是按逻辑和严格的规则来工作，但结果可能并不是我们所期望的。如果存在任何疑问，最好指定显式地使用哪种强制转换。

委托、Lambda 表达式和事件

委托是寻址方法的 .NET 本。在 `C++` 中，函数指针只不过是一个指向内存位置的指针，它

不是类型安全的。我们无法判断这个指针实际指向什么，像参数和返回类型等项就更无从知晓了。而.NET 委托完全不同，委托是类型安全的类，它定义了返回类型和参数的类型。委托类不仅包含对方法的引用，也可以包含对多个方法的引用。

Lambda 表达式与委托直接相关。当参数是委托类型时，就可以使用 Lambda 表达式实现委托引用的方法。

委托

启动线程和任务——在 C#中，可以告诉计算机并行运行某些新的执行序列同时运行当前的任务。这种序列就称为线程，在其中一个基类 `System.Threading.Thread` 的一个实例上使用方法 `Start()`，就可以启动一个线程。如果要告诉计算机启动一个新的执行序列，就必须说明要在哪里启动该序列。必须为计算机提供开始启动的方法的细节，即 `Thread` 类的构造函数必须带有一个参数，该参数定义了线程调用的方法。

通用库类——许多库包含执行各种标准任务的代码。这些库通常可以自我包含。这样在编写库时，就会知道任务该如何执行。但是有时在任务中还包含子任务，只有使用该库的客户端代码才知道如何执行这些子任务。例如，编写一个类，它带有一个对象数组，并把它们按升序排列。但是，排序的部分过程会涉及重复使用数组中的两个对象，比较它们，看看哪一个应放在前面。如果要编写的类必须能对任何对象数组排序，就无法提前告诉计算机应如何比较对象。处理类中对象数组的客户端代码也必须告诉类如何比较要排序的特定对象。换言之，客户端代码必须给类传递某个可以调用并且进行这种比较的合适方法的细节。

事件——一般是通知代码发生了什么事情。GUI 编程主要处理事件。在引发事件时，运行库需要知道应执行哪个方法。这就需要把处理事件的方法作为一个参数传递给委托的。

在 C 和 C++ 中，只能提取函数的地址，并作为一个参数传递它。C 没有类型安全性。可以把任何函数传递给需要函数指针的方法。但是，这种直接方法不仅会导致一些关于类型安全性的问题，而且没有意识到：在进行面向对象编程时，几乎没有方法是孤立存在的，而是在调用方法前通常需要与类实例相关联。所以.NET Framework 在语法上不允许使用这种直接方法。如果要传递方法，就必须把方法的细节封装在一种新类型的对象中，即委托。委托只是一种特殊类型的对象，其特殊之处在于，我们以前定义的所有对象都包含数据，而委托包含的只是一个或多个方法的地址。

声明委托

在 C#中使用一个类时，分两个阶段。首先，需要定义这个类，即告诉编译器这个类由什么字段和方法组成。然后(除非只使用静态方法)实例化类的一个对象。使用委托时，也需要经过这两个步骤。首先必须定义要使用的委托，对于委托，定义它就是告诉编译器这种类型的委托表示哪种类型的方法。然后，必须创建该委托的一个或多个实例。编译器在后台将创建表示该委托的一个类。

定义语法如下：

```
delegate void IntMethodInvoker (int x);
```

在这个示例中，定义了一个委托 `IntMethodInvoker`，并指定该委托的每个实例都可以包含一个方法的引用，该方法带有一个 `int` 参数，并返回 `void`。理解委托的一个要点是它们的类型安全性非常高。在定义委托时，必须给出它所表示的方法签名和返回类型等全部细节。

Tips:

- 理解委托的一种好方式是把委托当作这样一件事情，它给方法的签名和返回类型指定名称。

假定要定义一个委托 `TwoLongsOp`，该委托表示的方法有两个 `long` 类型参数，返回类型为 `double`。可以编写如下代码：

```
delegate double TwoLongsOp(long first, long second);
```

或者要定义一个委托，它表示的方法不带参数，返回一个 `string` 类型的值，可以编写如下代码：

```
delegate string GetAString();
```

其语法类似与方法的定义，但没有方法体，定义的前面要加上关键字 `delegate`。因为定义委托基本上是定义一个新类，所以可以在定义类的任何相同地方定义委托，也就是说，可以在另一个类的内部定义，也可以在任何类的外部定义，还可以在名称空间中把委托定义为顶层对象。根据定义的可见性和委托的作用域，可以在委托的定义上应用任何常见的访问修饰符：`public`、`private`、`protected` 等：

```
public delegate string GetAString();
```

Tips:

- 实际上，“定义一个委托”是指“定义一个新类”。委托实现为派生自基类 `System.MulticastDelegate` 的类，`System.MulticastDelegate` 又派生自基类 `System.Delegate`。`C#`编译器能识别这个类，会使用其委托语法，因此我们不需要了解这个类的具体执行情况。

定义好委托后，就可以创建它的一个实例，从而用它存储特定方法的细节。

Tips:

- 但是，在术语方面有一个问题。类有两个不同的术语：“类”表示较广义的定义，“对象”表示类的实例。但委托只有一个术语。在创建委托的实例时，唆创建的委托的实例仍称为委托。必须从上下文中确定委托的确切含义。

使用委托

下面的代码说明了如何使用委托。这是在 `int` 上调用 `ToString()` 方法的一种相当冗长的方式：

Example:

```
using System;
```

```
namespace Test.Professnol.Delegates
```

```
{
```

```
    delegate string GetAString();
```

```
    class DelegateExample
```

```
    {
```

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

```
        {
```

```
            int x=40;
```

```
            GetAString firstStringMethod = new GetAString(x.ToString);
```

```
            GetAString SecondStringMehod = x.ToString;
```

```
            Console.WriteLine("String is {0}", firstStringMethod);
```

```
            Console.WriteLine("Second Stirng is {0}", SecondStringMehod);
```



```

        Console.In.ReadLine();
    }
}
}

```

在这段代码中，实例化了类型为 `GetAString` 的一个委托，并对它进行初始化，使它引用整型变量 `x` 的 `ToString()` 方法。在 `C#` 中，委托在语法上总是接受一个参数的构造函数，这个参数就是委托引用的方法。这个方法必须匹配最初定义委托时的签名。所以在这个示例中，如果用不带参数并返回一个字符串的方法来初始化 `firstStringMehod` 变量，就会产生一个编译错误。注意，因为 `int.ToString()` 是一个实例方法(不是静态方法)，所以需要指定实例(`x`)和方法名来正确地初始化委托。

下一行代码使用这个委托来显示字符串。在任何代码中，都应提供委托实例的名称，后面的圆括号中应包含调用该委托中的方法时使用的任何等效参数。所以在上面的代码中，`Console.WriteLine()` 语句完全等价于注释语句中的代码行。

实际上，给委托实例提供圆括号与调用委托类的 `Invoke()` 方法完全相同。因为 `firstStringMeghod` 是委托类型的一个变量，所以 `C#` 编译器会用 `firstStringInvoke()` 代替 `firstStringMethod()`。

```

firstStringMethod();
firstStringMethod.Invoke();

```

为了减少输入量，只要需要委托实例，就可以只传送地址的名称。这称为委托推断。只要编译器可以把委托实例解析为特定的类型，这个咧特就是有效的。下面的示例用 `GetAString` 委托的一个新实例初始化 `GetAString` 类型的 `firstStringMethod` 变量：

```

GetAString firstStringMethod = new GetAString( x.ToString );

```

只要用变量 `x` 把方法名传送给变量 `firstStringMethod`，就可以编写出作用相同的代码：

```

GetAString firstStringMethod = x.ToString;

```

`C#` 编译器创建的代码是一样的。由于编译器会用 `firstStringMethod` 检测需要的委托类型，因此它创建 `GetAString` 委托类型的一个实例，用对象 `x` 把方法的地址传送给构造函数。

Tips:

- 调用上述方法名时输入形式不能为 `x.ToString()`(不要输入圆括号)，也不能把它传送给委托变量。输入圆括号调用一个方法。调用 `x.ToString()` 方法会返回一个不能赋予委托变量的字符串对象。只能把方法的地址赋予委托变量。

委托推断可以在需要委托实例的任何地方使用。委托推断也可以用于事件，因为事件基于委托。

委托的一个特征是它们的类型是安全的，可以确保被调用的方法的签名是正确的。但有趣的是，它们不关心在什么类型的对象上调用该方法，甚至不考虑该方法是静态方法，还是实例方法。

Tips:

- 给定委托的实例可以引用任何类型的任何对象上的实例方法或静态方法——只要方法的签名匹配于委托的签名即可。

Example:

```

using System;

namespace Test.Professnol.Delegates
{
    delegate string GetAString();
}

```

```

struct Currency
{
    public uint Dollars;
    public ushort Cents;

    public Currency(uint dollars, ushort cents)
    {
        this.Dollars = dollars;
        this.Cents = cents;
    }
    public override string ToString()
    {
        return String.Format("$ {0}. {1,2:00}", Dollars, Cents);
    }
    public static string GetCurrentUnit()
    {
        return "Dollars";
    }
    public static explicit operator Currency(double value)
    {
        checked
        {
            uint dollars = (uint)value;
            ushort cents = (ushort)((value - dollars) * 100);
            return new Currency(dollars, cents);
        }
    }
    public static implicit operator float (Currency value)
    {
        return value.Dollars+(value.Cents/100f);
    }
    public static implicit operator Currency(uint value)
    {
        return new Currency(value, 0);
    }
    public static implicit operator uint (Currency value)
    {
        return value.Dollars;
    }
}

```

```

class DelegateExample

```

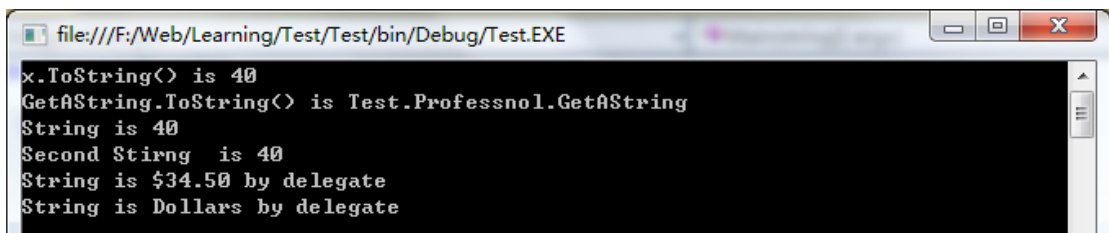
```

{
    public static void Main(string[] args)
    {
        int x=40;
        GetAString firstStringMethod = new GetAString(x.ToString);
        GetAString SecondStringMehod = x.ToString;
        Console.WriteLine("x.ToString() is {0}", x.ToString());
        Console.WriteLine("GetAString.ToString() is {0} ",
firstStringMethod.ToString());
        Console.WriteLine("String is {0}", firstStringMethod());
        Console.WriteLine("Second Stirng is {0}", SecondStringMehod());

        Currency balance = new Currency(34, 50);
        firstStringMethod = balance.ToString;
        Console.WriteLine("String is {0} by delegate", firstStringMethod());
        firstStringMethod = Currency.GetCurrentUint;
        Console.WriteLine("String is {0} by delegate", firstStringMethod());
        Console.In.ReadLine();
    }
}
}

```

运行结果:



```

file:///F:/Web/Learning/Test/Test/bin/Debug/Test.EXE
x.ToString() is 40
GetAString.ToString() is Test.Professnol.GetAString
String is 40
Second Stirng is 40
String is $34.50 by delegate
String is Dollars by delegate

```

这段代码说明了如何通过委托来调用方法，然后重新给委托指定在类的不同实例上引用的不同方法，甚至可以指静态方法，或者在类的不同类型的实例上引用的方法，只要每个方法的签名匹配委托定义即可。

简单的委托示例

Example:

```

using System;

namespace Test.Professnol.Delegates
{
    class MathOperations
    {
        public static double MultiplyByTwo(double value)
        {
            return value * 2;
        }
    }
}

```

```

    }

    public static double Square(double value)
    {
        return value * value;
    }
}

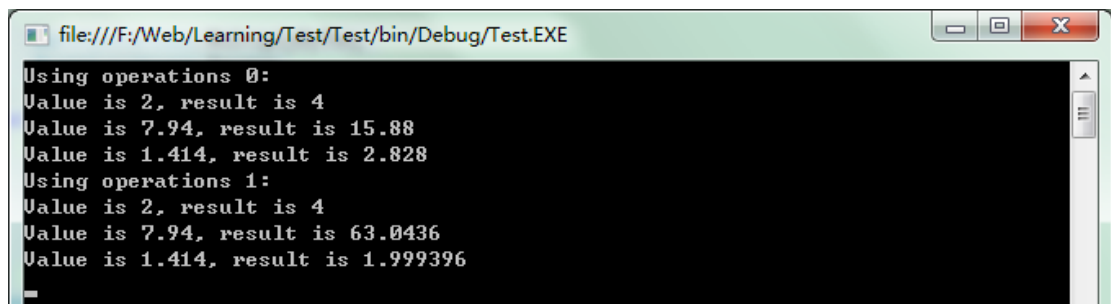
delegate double DoubleOp(double x);

class SimpleDelegateExample
{
    public static void Main(string[] args)
    {
        DoubleOp [] operations =
        {
            MathOperations.MultiplyByTwo,
            MathOperations.Square
        };
        for (int i = 0; i < operations.Length; i++)
        {
            Console.WriteLine("Using operations {0}:", i);
            ProcessAndDisplayNumber(operations[i], 2.0);
            ProcessAndDisplayNumber(operations[i], 7.94);
            ProcessAndDisplayNumber(operations[i], 1.414);
        }
        Console.In.ReadLine();
    }

    public static void ProcessAndDisplayNumber(DoubleOp action, double value)
    {
        double result = action(value);
        Console.WriteLine("Value is {0}, result is {1}", value, result);
    }
}
}

```

运行结果:



```

file:///F:/Web/Learning/Test/Test/bin/Debug/Test.EXE
Using operations 0:
Value is 2, result is 4
Value is 7.94, result is 15.88
Value is 1.414, result is 2.828
Using operations 1:
Value is 2, result is 4
Value is 7.94, result is 63.0436
Value is 1.414, result is 1.999396

```

上述代码中实例化了一个委托数组 `DoubleOp`(记住，一旦定义了委托类，基本上就可以实例化它的实例，就像处理一般的类那样——所以把一些委托的实例放在数组中是可以的)。该数组的每个元素都初始化为有 `MathOperations` 类实现的不同操作。然后遍历这个数组，把每个操作应用到 3 个不同的值上。这说明了使用委托的一种方式——把方法组合到一个数组中来使用，这样就可以在循环中调用不同的方法了。

上述代码的关键一行是把每个委托传递给 `ProcessAndDisplayNumber()` 方法，例如：

```
ProcessAndDisplayNumber(operations[i] , 2.0);
```

其中传递了委托名，但不带任何参数。假定 `operations[i]` 是一个委托，其语法是：

`operations[i]` 表示“这个委托”。换言之，就是委托的表示方法。

`operations[i](2.0)` 表示“实际上调用这个方法，参数放在圆括号中”。

`ProcessAndDisplayNumber()` 方法定义为把一个委托作为其第一个参数：

```
public static void ProcessAndDisplayNumber( DoubleOp action , double value)
```

然后在这个方法中调用：

```
double result = action (value);
```

这实际上是调用 `action` 委托实例封装的方法，其返回结果存储在 `result` 中。

Action<T>和 Func<T>委托

除了为每个参数和返回类型定义一个新委托类型之外，还可以使用 `Action<T>` 和 `Func<T>` 委托。泛型 `Action<T>` 委托表示引用一个 `void` 返回类型的方法。因为这个委托类存在不同的变体，所以可以传递至多 16 中不同的参数类型。没有泛型参数的 `Action` 类可调用没有参数的方法。`Action<in T>` 调用带一个参数的方法，`Action<in T1 , in T2>` 调用带两个参数的方法，`Action<in T1 , in T2 , in T3 , in T4 , in T5 , in T6 , in T7 , in T8>` 调用带 8 个参数的方法。

`Func<T>` 委托可以以类似的方式使用，`Func<T>` 允许调用带返回类型的方法。与 `Action<T>` 类似，`Func<T>` 也定义了不同的变体，至多也可以传递 16 个参数类型和一个返回类型。`Func<out TResult>` 委托类型可以调用带返回类型且无参数的方法，`Func<in T , out TResult>` 调用带一个参数的方法。`Func<in T1 , in T2 , in T3 , in T4 , out TResult>` 调用带 4 个参数的方法。

上一个示例声明了一个委托，其参数是 `double` 类型，返回类型是 `double`：

```
delegate double DoubleOp (double x);
```

除了声明自定义委托 `DoubleOp` 之外，还可以使用 `Func<in T , out TResult>` 委托。可以声明一个该委托类型的变量，或者该委托类型的数组，如下所示：

```
Func <double , double>[] operations=
```

```
{  
    MathOperations.MultiplyByTwo,  
    MathOperations.Square  
};
```

使用它，并将 `ProcessAndDisplayNumber()` 方法作为参数：

```
public static void ProcessAndDisplayNumber(Func<double , double>action , double value)  
{  
    double result = action(value);  
    Console.WriteLine("Value is {0} , result of operation is {1}" , value , result);  
}
```

BubbleSorter 示例

Example:

```
using System;
using System.Collections.Generic;

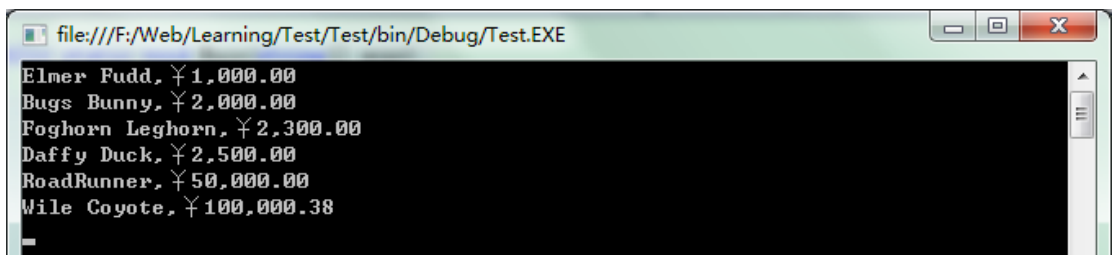
namespace Test.Professno1.Delegates
{
    class Employee
    {
        public string Name {get;private set;}
        public decimal Salary {get;private set;}
        public Employee(string name, decimal salary)
        {
            this.Name = name;
            this.Salary = salary;
        }
        public override string ToString()
        {
            return String.Format("{0}, {1:C}", Name, Salary);
        }
        public static bool CompareSalary(Employee e1, Employee e2)
        {
            return e1.Salary < e2.Salary;
        }
    }
    class BubbleSort
    {
        public static void Sort<T>(IList<T> sortArray, Func<T, T, bool> comparsion)
        {
            bool swapped = true;
            do
            {
                swapped = false;
                for (int i = 0; i < sortArray.Count - 1; i++)
                {
                    if (comparsion(sortArray[i + 1], sortArray[i]))
                    {
                        T temp = sortArray[i];
                        sortArray[i] = sortArray[i + 1];
                        sortArray[i + 1] = temp;
                        swapped = true;
                    }
                }
            }
        }
    }
}
```

```

    }
    while (swapped);
}
}
class CommonDelegateExample
{
    public static void Main(string[] args)
    {
        Employee[] employees =
        {
            new Employee("Bugs Bunny", 2000),
            new Employee("Elmer Fudd", 1000),
            new Employee("Daffy Duck", 2500),
            new Employee("Wile Coyote", 100000.38m),
            new Employee("Foghorn Leghorn", 2300),
            new Employee("RoadRunner", 50000)
        };
        BubbleSort.Sort(employees, Employee.CompareSalary);
        foreach (var employee in employees)
        {
            Console.WriteLine(employee);
        }
        Console.In.ReadLine();
    }
}
}

```

运行结果：



```

file:///F:/Web/Learning/Test/Test/bin/Debug/Test.EXE
Elmer Fudd, ¥1,000.00
Bugs Bunny, ¥2,000.00
Foghorn Leghorn, ¥2,300.00
Daffy Duck, ¥2,500.00
RoadRunner, ¥50,000.00
Wile Coyote, ¥100,000.38

```

上述代码实现冒泡排序的泛型版本。然而对于接受类型 `T` 的泛型方法 `Sort<T>()` 需要一个比较方法，其两个参数的类型是 `T`，if 比较的返回类型是布尔类型。这个方法可以从 `Func<T1, T2, TResult>` 委托中引用，其中 `T1` 和 `T2` 的类型相同：`Func<T, T, bool>`。

给 `Sort<T>` 方法指定下述签名：

```
public static void Sort<T>(IList<T> sortArray, Func<T, T, bool> comparsion)
```

这个方法的文档说明，`comparsion` 必须引用一个方法，该方法带有两个参数，如果第一个参数的值“小于”第二个参数，就返回 `true`。

在 `Employee` 类中，为了匹配 `Func<T, T, bool>` 委托的签名，在这个类中必须定义 `CompareSalary`，它的参数是两个 `Employee` 引用，并返回一个布尔值。

多播委托

委托可以包含多个方法，这种委托成为多播委托。如果调用多播委托，就可以按顺序连续调用多个方法。为此，委托的签名就必须返回 **void**；否则，就只能得到委托调用的最后一个方法的结果。

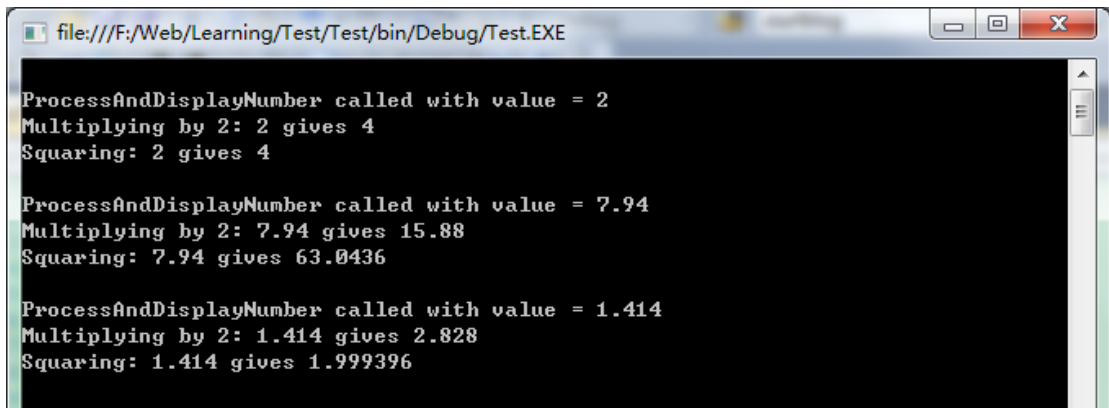
Example:

```
using System;
```

```
namespace Test.Professno1.Delegates
```

```
{  
    class MathOperations  
    {  
        public static void MultiplyByTwo(double value)  
        {  
            double result = value * 2;  
            Console.WriteLine("Multiplying by 2: {0} gives {1}", value, result);  
        }  
        public static void Square(double value)  
        {  
            double result = value * value;  
            Console.WriteLine("Squaring: {0} gives {1}", value, result);  
        }  
    }  
    class MultiplyDelegateExample  
    {  
        public static void ProcessAndDisplayNumber(Action<double> action, double value)  
        {  
            Console.WriteLine();  
            Console.WriteLine("ProcessAndDisplayNumber called with value = {0}", value);  
            action(value);  
        }  
        public static void Main(string[] args)  
        {  
            Action<double> operations = MathOperations.MultiplyByTwo;  
            operations += MathOperations.Square;  
            ProcessAndDisplayNumber(operations, 2.0);  
            ProcessAndDisplayNumber(operations, 7.94);  
            ProcessAndDisplayNumber(operations, 1.414);  
            Console.In.ReadLine();  
        }  
    }  
}
```

运行结果：



```
file:///F:/Web/Learning/Test/Test/bin/Debug/Test.EXE

ProcessAndDisplayNumber called with value = 2
Multiplying by 2: 2 gives 4
Squaring: 2 gives 4

ProcessAndDisplayNumber called with value = 7.94
Multiplying by 2: 7.94 gives 15.88
Squaring: 7.94 gives 63.0436

ProcessAndDisplayNumber called with value = 1.414
Multiplying by 2: 1.414 gives 2.828
Squaring: 1.414 gives 1.999396
```

上述代码中使用了返回类型为 `void` 的 `Action<double>` 委托，并且在同一个多播委托中添加两个操作。多播委托可以识别运算符 “+” 和 “+=”。另外，还可以扩展上述代码中主函数中关于多播委托的代码：

```
Action<double> operation1 = MathOperations.MultiplyBuTwo;
```

```
Action<double> operation2 = MathOperations.Square;
```

```
Action<double> operations = operation1 + operation2;
```

多播委托还识别运算符 “-” 和 “-=”，以从委托中删除方法调用。

Tips:

- 多播委托实际上是一个派生自 `System.MulticastDelegate` 的类，`System.MulticastDelegate` 又派生自基类 `System.Delegate`。`System.MulticastDelegate` 的其他成员允许把多个方法调用链接为一个列表。

如果正在使用多播委托，就应知道对同一个委托调用方法链的顺序并未正式定义。因此应避免编写依赖于以特定顺序调用方法的代码。

通过一个委托调用多个方法还可能导致一个大问题。多播委托包含一个逐个调用的委托集合。如果通过委托调用的其中一个方法抛出一个异常，整个迭代就会停止。

Example:

```
using System;
```

```
namespace Test.Professnol.Delegates
```

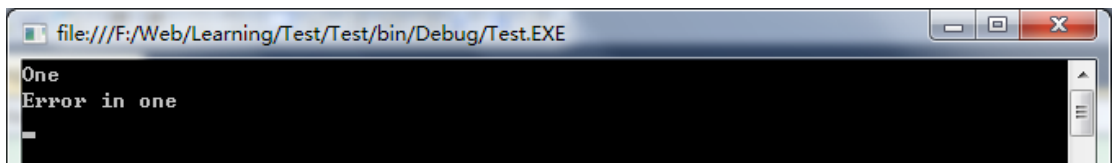
```
{
    class BubbleDelegate
    {
        static void One ()
        {
            Console.WriteLine("One");
            throw new Exception("Error in one");
        }
        static void Two ()
        {
            Console.WriteLine("Two");
        }
        static void Main(string[] args)
        {
            Action d1 = One;
```

```

        dl += Two;
        try
        {
            dl();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
}

```

运行结果:



根据结果显示委托只调用了第一个方法。因为第一个方法抛出了一个异常，所以委托的迭代会停止，不再调用 **Two** 方法。没有指定当调用方法的顺序时，结果会有所不同。

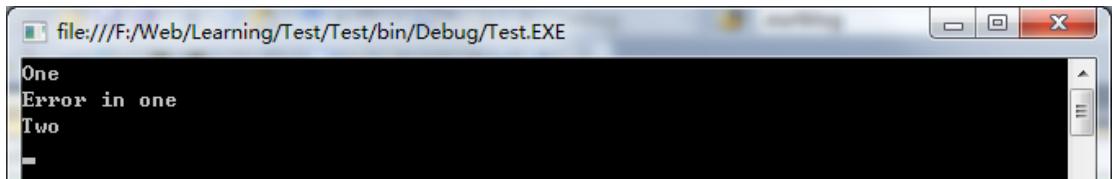
在这种情况下，为了避免这个问题，应自己迭代方法列表。**Delegate** 类定义 **GetInvocationList()** 方法，它返回一个 **Delegate** 对象数组。现在可以使用这个委托调用与委托直接相关的方法，捕获异常，并继续下一次迭代。

```

static void Main(string[] args)
{
    Action dl = One;
    dl += Two;
    Delegate[] delegates = dl.GetInvocationList();
    foreach (Action action in delegates)
    {
        try
        {
            action();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
    Console.In.ReadLine();
}

```

运行结果:



匿名方法

要想使委托工作,方法必须已经存在(即委托是用它将调用的方法的相同签名定义的)。但还有另外一种使用委托的方式:即通过匿名方法。匿名方法是用作委托的参数的一段代码。

用匿名方法定义委托的语法与前面的定义并没有区别。但在实例化委托时,就有区别了。下面是一个非常简单的控制台应用程序,它说明了如何使用匿名方法:

Example:

```
using System;
```

```
namespace Test.Professno1.Delegates
```

```
{
```

```
    class AnonymousDelegates
```

```
    {
```

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

```
        {
```

```
            string mid = ", middle part,";
```

```
            Func<string, string> anonDel = delegate(string param) //the anonymous delegate
```

```
            {
```

```
                //the anonymous delegate method
```

```
                param += param;
```

```
                param += " and this was added to the string.";
```

```
                return param;
```

```
            };
```

```
            Func<string, string> Del=ReturnString; //another method
```

```
            Console.WriteLine (anonDel (mid));
```

```
            Console.WriteLine (Del("using another method"));
```

```
            Console.In.ReadLine();
```

```
        }
```

```
        public static string ReturnString (string param)
```

```
        {
```

```
            param += param;
```

```
            param += " and this was added to the string.";
```

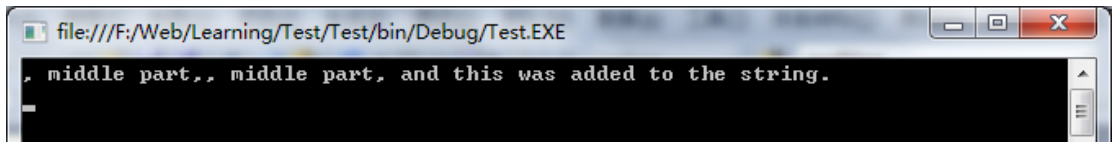
```
            return param;
```

```
        }
```

```
    }
```

```
}
```

运行结果:



`Func<string,string>`委托接受一个字符串参数，返回一个字符串。`anonDel` 是这种委托类型的变量。不是把方法名赋予这个变量，而是使用一段简单的代码：它前面是关键字 `delegate`，后面是一个字符串参数：

可以看出，该代码块使用方法级的字符串变量 `mid`，该变量是在匿名方法的外部定义的，并把它添加到要传递的参数中。接着代码返回该字符串值。在调用委托时，把一个字符串作为参数传递，将返回的字符串输出到控制台上。

匿名方法的优点是减少了要编写的代码。不必定义仅由委托使用的方法。在为事件定义委托时，这是非常显然的。这有助于降低代码的复杂性，尤其是定义了好几个事件时，代码会显得比较简单。使用匿名方法时，代码执行得不太快。编译器仍定义了一个方法，该方法只有一个自动指定的名称，我们不需要知道这个名称。

在使用匿名方法时，必须遵循两条规则。在匿名方法中不能使用跳转语句(`break`、`goto` 或 `continue`)跳到该匿名方法的外部，反之亦然：匿名方法外部的跳转语句不能跳到该匿名方法的内部。

在匿名方法内部不能访问不安全的代码。另外，也不能访问在匿名方法外部使用的 `ref` 和 `out` 参数。但可以使用在匿名方法外部定义的其他变量。

如果需要用匿名方法多次编写同一个功能，就不要使用匿名方法。在上面示例中，除了复制代码，编写一个指定的方法比较好，因为该方法只需编写一次，以后可通过名称引用它。

从 C#3.0 开始，可以使用 `Lambda` 表达式替代匿名方法。

Lambda 表达式

自 C#3.0 开始，就可以使用一种新语法把实现代码赋予委托：`Lambda` 表达式。只要有委托参数类型的地方，就可以使用 `Lambda` 表达式。前面使用匿名方法的例子可以改为使用 `Lambda` 表达式：

Tips:

- `Lambda` 表达式的语法比匿名方法简单。如果所调用的方法有参数，且不需要参数，匿名方法的语法就比较简单，因为这样不需要提供参数。

Example:

```
using System;
```

```
namespace Test.Professnol.Delegates.LambdaExample
{
```

```
    class SimpleLambda
```

```
    {
```

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

```
        {
```

```
            string mid = ", middle part,";
```

```
            Func<string, string> anonDel = param=> //the delegate using by Lambda
```

```
            {
```

```
                //the method of Lambda
```

```
        param += param;
        param += " and this was added to the string.";
        return param;
    };
    Console.WriteLine(anonDel("Start of a string"));
    Console.In.ReadLine();
}
}
```

Lambda 运算符“=>”的左边列出了需要的参数。Lambda 运算符的右边定义了赋予 lambda 变量的方法的实现代码。

参数