

```

      01001
    0010010101101010
  1001010101001010101
01010101010101010101
001010101      101001010
10101010      10101010
01010101      10101010
01001010      10001001
01010011      11110111
00110101      10001001
01001011      00011010
10101001      01100101
01011011      11010101
01001010      01100101
01001010      10101010
10111101      11010101
00101010      10101001
010100011      01010101
001111010      101001010
1010110110101000101010
01010101010101010
100101010101

```

```

      10111
    0110110010001010
  0001010111010110101
01010111000101010101
001010101      101001010
10101010      10101010
00110101      10101010
01001010      01101100
11100011      00010111
10110101      11110001
01101011      00011010
00101001      01100101
01000111      11010101
01001010      01100101
01010010      10101010
10111101      11010101
00101010      10101001
01011101      01010101
001011010      001110101
010101101010001011001
0110000101010101010
010101010101

```

```

0100101001010110
101010100101010101001
0101010101010101010101
0010100101011010100101010
1010010      010101010
1010101      01010010
1011010      01001010
1010101      01010101
0101010      01010010
1011010      10100100
0010010      10101010
101010101010101001010010
1011010101001010101000
10101010101010101110
10101010001010
0101011
0101010
0101010
1010010
1010101
0101010
0010101

```

物件導向實作課程（使用C#）第十梯

2017-10-14 ~ 2017-10-28 共 21 H



Bill Chung V1.5.2

關於我

- Bill Chung
- 海角點部落

教材與範例

- 因課程教材眾多，為響應節能省碳本課程不提供紙本教材。
- 教材與範例皆放置於 Yammer 。
- 建議您使用 OneNote 做筆記
 - <http://demo.tc/post/829>



前言



物件導向的迷思

可能聽過..

- OOP 寫起來很花時間
- 程式會動就好
- 我把共用的方法都寫在一個類別裡
- 我用 C#，所以寫的就是 OOP
- 在 Visual Studio 上拉一拉畫面不就是 OOP 了嗎？
-歡迎提供更多奇怪的說法

不當設計的根源

- 對使用的 API 一知半解
- 對物件導向的基本概念不了解
- 急就章，隨便抄個程式碼
- ...

這堂課的重點是思考

思考是甚麼？

物件導向的基本觀念

物件的特性

- 物件必須有一個資料結構來存放資料
- 物件有狀態和行為
- 物件要能被識別
- 物件可以被創造及消滅
- 物件有生命週期
- 物件要能為自己負責

程序與資料結合的意義

```
public abstract class OODoor
{
    public Int32 Width
    { get; set; }

    public Int32 Height
    { get; set; }

    public bool Opened
    { get; protected set; }

    public abstract void Open();

    public abstract void Close();
}
```

CH0\OODoorSample

農夫渡河

- 農夫要帶著狼、羊、菜過河。
- 小船不夠大，因此農夫每次只能帶一樣東西過河。
- 當農夫在的時候，狼、羊、菜都不會有事情。
- 當農夫不在時，狼會吃羊，羊會吃菜。



<http://forums.indiegamer.com/threads/the-river-tests-pro-collection-of-river-crossing-puzzles-universal-app.34639/>

Form1

農夫
狼
羊
菜

--->

<---

```
public class Role
{
    public RoleName Name
    { get; private set; }

    public RoleName Food
    { get; private set; }

    public Role(RoleName name, RoleName food)
    {
        Name = name;
        Food = food;
    }

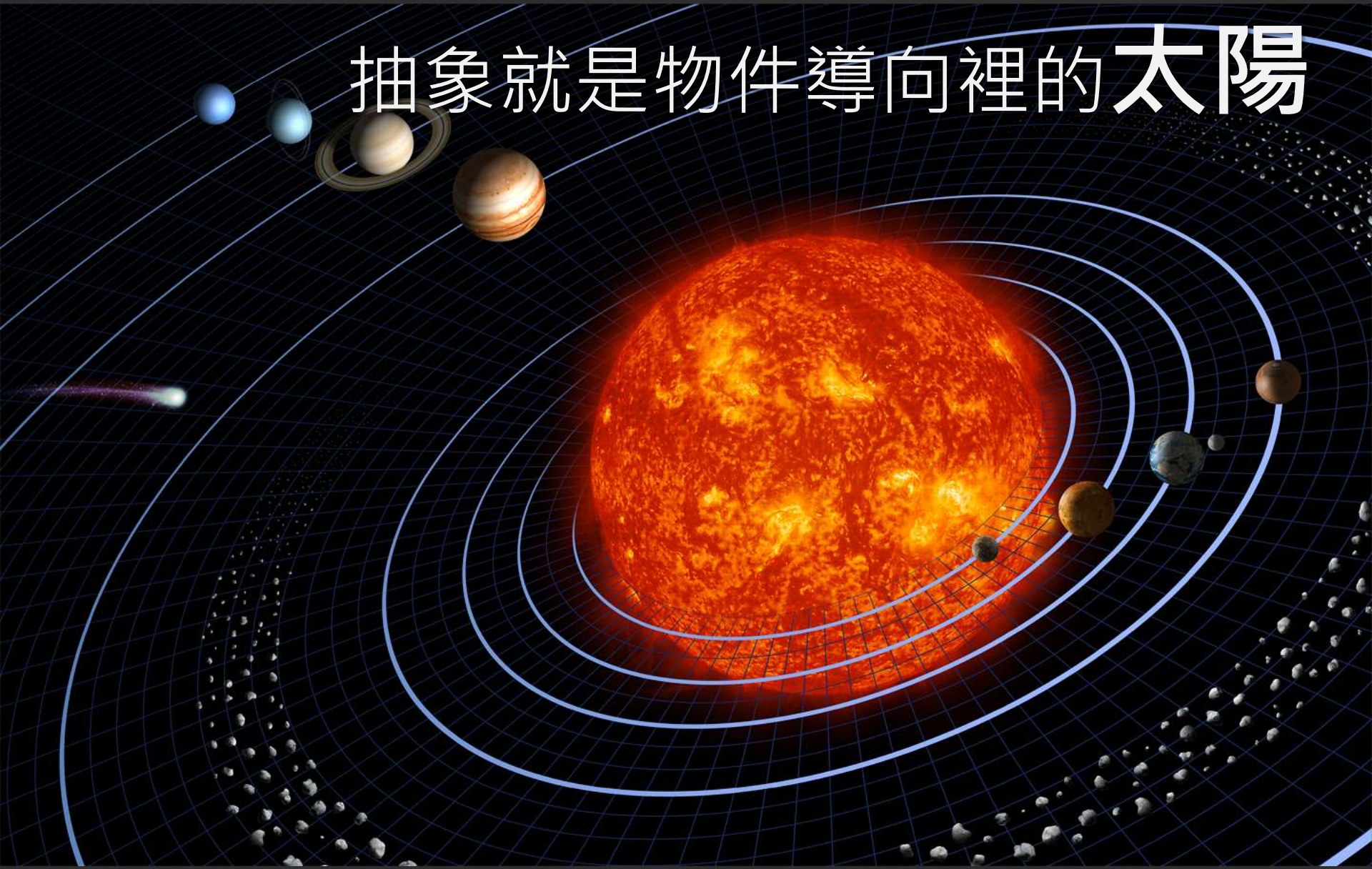
    public bool Eat(Role other)
    {
        return (Food != RoleName.None && Food == other.Name);
    }
}
```


類別與物件的關係

- 外界的真實事物都有其相似或相同性質之處
- 類別的形成即是透過分類的過程將一群類似的物件抽象化成一個概念

抽象

抽象就是物件導向裡的太陽



抽象是.....

- 找出關鍵性特徵並加以描述
- 簡化模型以協助思考與運用

抽象能力的重要性

- 分類的基礎技巧就是抽象化
- 分類是必須的，但方式並非絕對的

為什麼要依賴抽象？

幾個抽象化的例子

抽象不難理解，對吧



演算法的抽象

計算 1~100 的總和

- 迴圈
- 數學公式

小數基準的無條件進位

- 假如目前有個帶小數點的數值
- 如何能依照所給的基數值(即未滿此數值 則需無條件進位) 進位?
- 範例：數值 $A = 16.75$, A 的基數為變數
- 基數為 0.1 時 換算出來為 16.8
- 基數為 0.05 時 換算出來為 16.75
- 基數為 0.2 時 換算出來為 16.8
- 基數為 0.5 時 換算出來為 17.00

查表法？別鬧了

釐清需求

- 無條件進位
- 餘數不為零才需要進位
- 進位的單位是基數

所以...

- 當數值 A 除以 基數其餘數大於零時則為 (整數商+1)乘以基數
- 當數值 A 除以基數其餘數為零時則為 整數商乘以基數 (也就是數值 A) 本身

CH0\CellingSamples\CellingSample01

更進一步...

- 就是剛好等於 (數值A 除以 基數) 無條件進位再乘以基數

CH0\CellingSamples\CellingSample02

順便學個簡單的單元測試

OOP 三大特性

三大特性

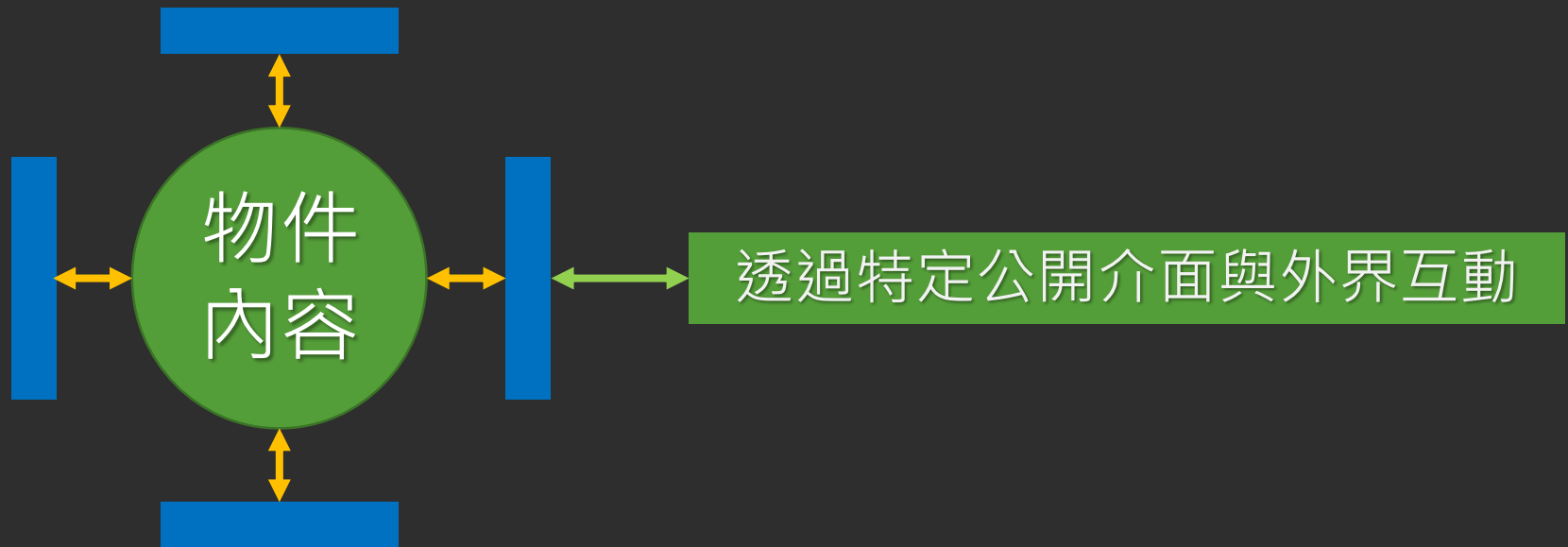
- 繼承
- 封裝
- 多型
- 絕大部分的設計就在應用這三大特性

繼承

- 繼承者會擁有被繼承者的型別特徵（非靜態）
- C# 中的繼承
 - 繼承一個上層類別（只能一個）
 - 實做介面（可以多個）

封裝

- 隱藏不必要為外界所知的資訊
- 隱藏行為的變化



多型

- 廣義多型 (universal polymorphism)
 - 繼承式多型 (inclusion)
 - 參數式多型 (parametric)
- 特設多型 (ad hoc polymorphism)
 - 多載 (overloading)
 - 強制同型 (coercions)

來源: [On Understanding Types, Data Abstraction, and Polymorphism](#)
1.3. Kinds of Polymorphism

繼承式多型

- 一般用語中的多型多半指的是繼承式多型
- 這表示繼承者會擁有被繼承者的型別特徵
- C# 中表現繼承式多型的方式
 - 繼承一個上層類別
 - 實做介面

參數式多型

- 以參數型式，讓類別可以達到動態變化的方法
- C# 中的泛型就是參數式多型的實踐

多載

- 程序多載
 - 表示使用同一個名稱但不同的參數清單，定義多個版本的程序
- 運算子多載

運算子多載範例

```
public static implicit operator Custom<T>(T value)
{
    return new Custom<T>(value);
}
```

```
public static explicit operator T(Custom<T> value)
{
    return value._value;
}
```

強制同型

```
int i = 1;  
double j = 5.66;  
double k = i + j;
```

型別與變數

**int 和 System.Int32
在定義上有甚麼不同？**

型別概論

- Primitive Type
- Reference Type
 - 介面 Interface
 - 類別 Class
 - 委派 Delegate
- Value Type
 - 結構 Structure
 - 列舉 Enum

參考型別 vs 實值型別

- 從變數內容的觀點
 - 實值型別變數內容就是物件本身
 - 參考型別的變數內容則是儲存指向物件的參考(指標)
- 從記憶體分配的觀點 (區域變數)
 - 實值型別的物件存在於 Stack
 - 參考型別的物件存在 Heap

參考與實值型別物件比較

參考型別

Type Object
Pointer

Sync block
index

Instance fields

實值型別

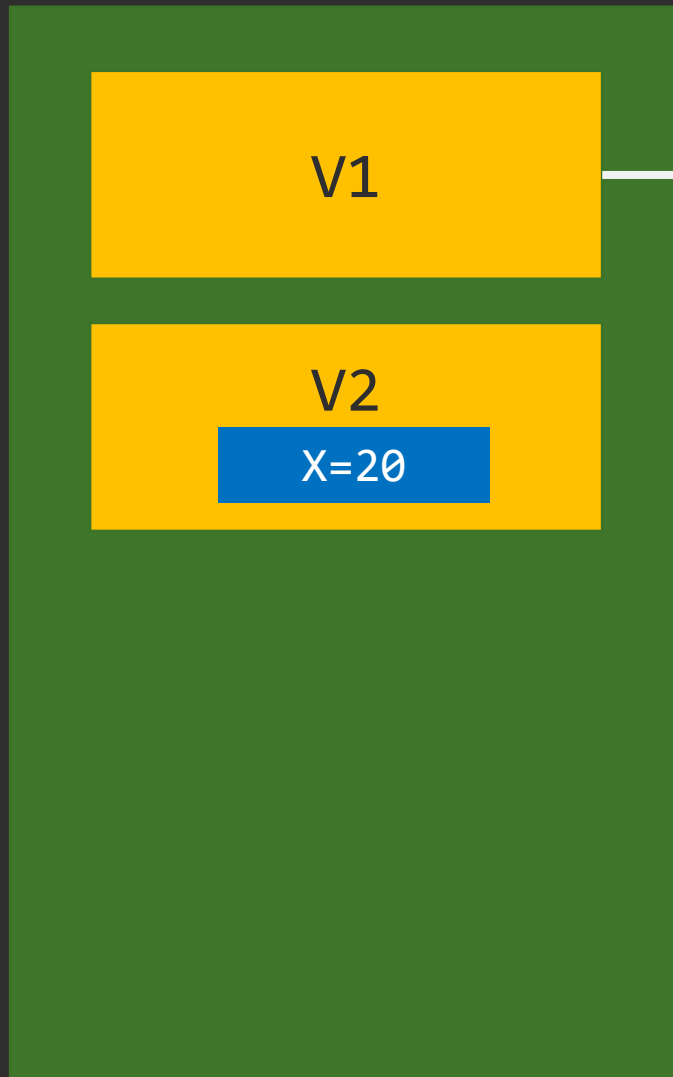
Instance fields

```
public class MyRefClass  
{ public int x; }
```

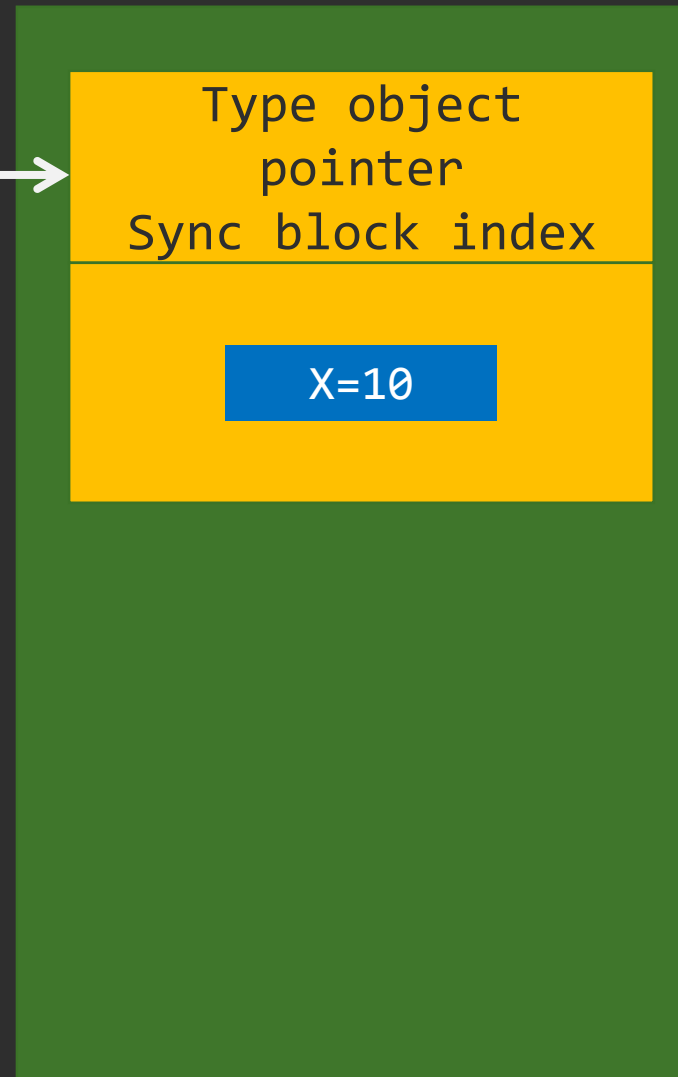
```
public struct MyValStruct  
{ public int x; }
```

```
private void CreateInstance()  
{  
    MyRefClass v1 = new MyRefClass();  
    MyValStruct v2 = new MyValStruct();  
    v1.x = 10;  
    v2.x = 20;  
}
```


Thread Stack



Managed Heap



變數的複製行為

var 宣告

- 強型別
- 隱含型別
- 或稱右(後)決議型別
- 只能做為宣告區域變數使用

必須使用 var 的情境

```
static void Main(string[] args)
{
    string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };
    var newwords = words.Select((w) =>
        new { Upper = w.ToUpper(), Lower = w.ToLower() });
    foreach (var x in newwords)
    {
        Console.WriteLine(x.Upper + " : " + x.Lower);
    }
    Console.ReadLine();
}
```

CH0\VarSample

實值型別 Value Type

特徵

- 一定會繼承 `System.ValueType`
- 以結構或列舉的形式存在
- 變數與物件是一對一的關係
- 實質型別物件沒有
 - `Type object pointer`
 - `Sync block index`
- 自訂宣告
 - `struct`
 - `enum`

Enum (列舉)

- Enum 的用途
- Enum 與 FlagsAttribute

Enum 宣告方式 (一般)

```
public enum Gender  
{ Man, Woman }
```

```
public enum Days : byte  
{ Sat = 1, Sun, Mon, Tue, Wed, Thu, Fri }
```

```
public enum PowerStatus  
{  
    On= 4,  
    Off =8  
}
```

CH0\EnumSamples\EnumSample

Enum with FlagsAttribute

```
[Flags]
public enum Authority
{
    Read = 1,
    Write = 2,
    Create = 4,
    Delete = 8
}
```

CH0\EnumSamples\FlagsEnumSample

使用 FlagsAttribute

```
static void Main(string[] args)
{
    Authority authority = Authority.Read | Authority.Write;
    Console.WriteLine(authority.ToString());
    Console.WriteLine(Convert.ToInt32(authority));

    if ((authority & Authority.Read) == Authority.Read)
    {
        Console.WriteLine("有讀取權限");
    }
    else
    {
        Console.WriteLine("無讀取權限");
    }

    Console.ReadLine();
}
```

Flags 是如何計算的？

Structure (結構)

- Structure 的弔詭之處
- Nullable<T>

Tricky sample 1

```
class Program
{
    static void Main(string[] args)
    {
        MyStruct a = new MyStruct() { X = 1, Y = 1 };
        var b = a;
        b.X = 2;
        Console.WriteLine(string.Format("a.X = {0}", a.X));
        Console.WriteLine(string.Format("b.X = {0}", b.X));
        Console.ReadLine();
    }
}

public struct MyStruct
{
    public int X { get; set; }
    public int Y { get; set; }
}
```

CH0\StructureTrickySamples\StructureTrickySample01

Tricky sample 2

```
static void Main(string[] args)
{
    List<MyStruct> list = new List<MyStruct>();
    MyStruct o = new MyStruct() { X = 1, Y = 1 };
    list.Add(o);
    o.X = 99;
    Console.WriteLine(list[0].X);
    // list[0].X = 100;
    var a = list[0];
    a.X = 777;
    list[0] = a;
    Console.WriteLine(list[0].X);
    Console.ReadLine();
}
```

CH0\StructureTrickySamples\StructureTrickySample02

Tricky Sample 3

```
static void Main(string[] args)
{
    MyStruct[] array = new MyStruct[2];
    array[0].X = 100;
    var o = array[0];
    Console.WriteLine(o.X);
    o.X = 888;
    array[0].X = 999;
    Console.WriteLine(array[0].X);
    Console.ReadLine();
}
```

CH0\StructureTrickySamples\StructureTrickySample03

Nullable<T>

- 它是個結構
- Properties
 - Value
 - HasValue
- Method
 - GetValueOrDefault

使用 Nullable<T>

```
static void Main(string[] args)
{
    Nullable<int> i;
    //int? i;
    //System.Int32? i;
    i = 100;
    Console.WriteLine(i.HasValue.ToString());
    Console.WriteLine(i.Value.ToString());
    Console.WriteLine(i.GetValueOrDefault().ToString());
    i = null;
    Console.WriteLine(i.GetValueOrDefault().ToString());
    Console.ReadLine();
}
```

CH0\NullableSample\NullableSample

Nullable<T> 的比較運算

```
static void Main(string[] args)
{
    int? x = null;
    int y = 0;
    Console.WriteLine(x == y);
    Console.WriteLine(x != null);
    Console.WriteLine(x.HasValue);
    Console.ReadLine();
}
```

CH0\NullableSample\NullableCompareSample

Boxing 與 UnBoxing

- 實質型別與參考型別之間的轉換
- 造成效能耗損

CH0\BoxingSample

類別

類別成員

- 可包含巢狀型別

 常數 `constant`

 欄位 `field`

 屬性 `property`

 方法 `method`

 事件 `event`

- 建構式 `constructor`

存取修飾詞

- 在命名空間中宣告的類別可為 `public` 或 `internal` (預設)
- 類別內的成員可以宣告為
 - `private` (預設)
 - `internal`
 - `protected`
 - `protected internal`
 - `public`

其他修飾詞

▪ `abstract`

- 類別：表示抽象類別，不具有公開建構式
- 成員：表示為抽象成員，此成員實做不完整，在其衍生類別中必須實做其內容

▪ `sealed`

- 類別：表示密封類別，此類別無法再被繼承
- 成員：當套用至成員時，`sealed` 修飾詞必須一律和 `override` 搭配使用，其衍生類別將無法再覆寫此成員

參考型別物件與執行個體物件比較

Type Object
Pointer

Sync block
index

Static fields

Method table

Type Object
Pointer

Sync block
index

Instance fields

成員修飾詞

- **abstract**

- 表示為抽象成員，此成員實做不完整，在其衍生類別中必須實做其內容

- **sealed**

- 當套用至成員時，**sealed** 修飾詞必須一律和 **override** 搭配使用
- 其衍生類別將無法再覆寫此成員

- **virtual**

- 允許在衍生類別中覆寫此成員

- **new**

- 明確隱藏繼承自基底類別的成員，或稱為遮蔽

- **override**

- 覆寫基底類別的虛擬(**virtual**) 成員

常數

常數

- 在編譯時期就會使用常數值取代
- 以 `const` 宣告常數
- 宣告同時必須初始化
- 具不可修改之特性
- 常數的運作方式
- 使用常數的注意事項

欄位

欄位

- 在 .Net 中，我們將定義於類別層級的變數稱為欄位
- 欄位 (Field) 是一個任意型別的變數
- 一般情境下，欄位的存取層級很少是 `public`
- `readonly field`

屬性

屬性

- 屬性 (Property) 就是提供讀取、寫入或計算私用 (Private) 欄位值之彈性機制的成員
- 方法的變形
- 使用屬性取代欄位成為公開介面
- C# 的自動實做屬性

屬性是方法的變形

```
public class Class2
{
    private int _x = 0;
    public int X
    {
        get { return _x; }
        set { _x = value; }
    }
}
```

```
public class Class1
{
    private int _x = 0;

    public int GetX()
    { return _x; }

    public void SetX(int value)
    { _x = value; }
}
```


屬性是方法的變形

```
public class Class2
{
    private int _x = 0;
    public int X
    {
        get { return _x; }
        set { _x = value; }
    }
}
```

屬性是方法的變形

```
public class Class1
{
    private int _x = 0;

    public int GetX()
    { return _x; }

    public void SetX(int value)
    { _x = value; }
}
```

屬性是方法的變形

```
public class Class2
{
    private int _x = 0;
    public int X
    {
        get { return _x; }
        set { _x = value; }
    }
}
```

```
public class Class1
{
    private int _x = 0;

    public int GetX()
    { return _x; }

    public void SetX(int value)
    { _x = value; }
}
```

唯讀 / 唯寫

- 只宣告 `get` / `set` 存取子其中之一
- 將要隱藏的存取子的存取層級降低

屬性宣告的方式

```
private int _x;  
public int X  
{  
    get { return _x; }  
    set { _x = value; }  
}
```

```
private int _y;  
public int y  
{  
    get { return _y; }  
}
```

```
private int _z;  
public int Z  
{  
    get { return _z; }  
    private set { _z = value; }  
}
```

自動實作屬性

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



```
public string T  
{ get; }
```

(2013與2015情況不同)



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



方法

方法

- 「方法」(Method) 是包含一系列陳述式 (Statement) 的程式碼區塊。 程式會「呼叫」 (Calling) 方法並指定所有必要的方法引數，藉以執行陳述式
- 方法參數的重要關鍵字
 - ref
 - out
 - params

方法宣告

```
publ i c   vi rtual   stri ng   GetStri ng(i nt x)  
存取修飾詞  其他修飾詞  回傳型別  方法名稱  傳入參數  
{  
  
  
  
}
```

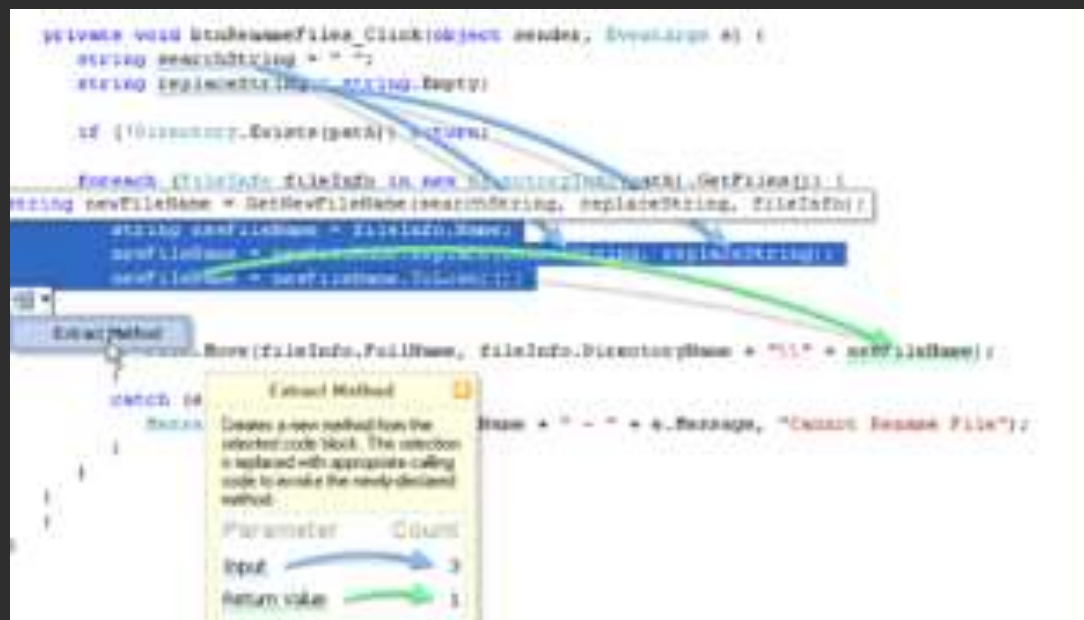
參數傳遞

一段不精確的闡釋

- 在.NET裡，除了像是int, string, decimal....等諸如此類的最基本的原生資料型態是以傳值方式在做之外，所有物件都是傳參考

傳值與傳址

- By Value
- By Reference
- 到底傳值與傳址的主詞是誰？



**正確描述 by value,
by reference ?**

實質型別的變數

```
int x = 0;
```

變數 `x` 在記憶體中佔有一個位址(eg. `0xFF80`)

變數 `x` 的型別是 `int`

變數 `x` 儲存的内容值是 `0`

實值型別傳值

```
static void Main(string[] args)
{
    int x = 0;
    int y = ChangeX(x);
}
```

取出 Main 方法中 x 的值
複製一份到 ChangeX 方法中的 x
(兩個 x 的變數位址不同)

```
private static int ChangeX(int x)
{
}
```

CH0\ParameterSamples\ParameterSample01

實值型別傳址

```
static void Main(string[] args)
{
    int x = 0;
    int y = ChangeX(ref x);
}
```

取出 Main 方法中 x 的位址
傳遞給 ChangeX 方法中的 x
(兩個 x 的變數位址相同)

```
private static int ChangeX(ref int x)
{
}
```

CH0\ParameterSamples\ParameterSample02

參考型別變數

```
TestClass x = new TestClass();
```

變數 x 在記憶體中佔有一個位址 (eg. 0xAA77)

變數 x 的型別是 `TestClass`

變數 x 儲存的内容值是一個 `TestClass` 型別的物件的位址 (0x1200)

型別 `TestClass` 的 Instance

位址: 0x1200

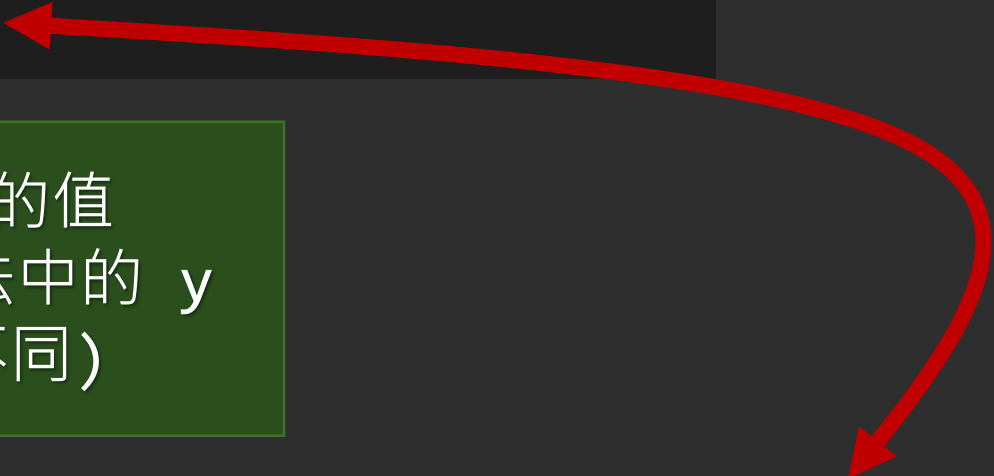
記憶體中有兩個東西

(1) 變數 x

(2) `TestClass` 所產生的實體

參考型別傳值

```
static void Main(string[] args)
{
    TestClass y = new TestClass();
    ChangeX(y);
}
```



取出 Main 方法中 y 的值
複製一份到 ChangeX 方法中的 y
(兩個 y 的變數位址不同)

```
private static TestClass ChangeX(TestClass y)
{
}
}
```

CH0\ParameterSamples\ParameterSample03

參考型別傳址

```
static void Main(string[] args)
{
    TestClass y = new TestClass();
    ChangeX(ref y);
}
```

取出 Main 方法中 y 的位址
傳遞給 ChangeX 方法中的 y
(兩個 y 的變數位址相同)

```
private static TestClass ChangeX(ref TestClass y)
{
}
```

CH0\ParameterSamples\ParameterSample04

可以直接看出結果嗎？

```
static void Main(string[] args)
{
    TestClass y = new TestClass();
    TestClass r1 = ChangeByVal(y);
    Console.WriteLine("r1 和 y 指向同實體 : " + (r1 == y).ToString());
    TestClass r2 = ChangeByRef(ref y);
    Console.WriteLine("r2 和 y 指向同實體 : " + (r2 == y).ToString());
    Console.ReadLine();
}

private static TestClass ChangeByVal(TestClass y)
{
    y = new TestClass();
    return y;
}

private static TestClass ChangeByRef(ref TestClass y)
{
    y = new TestClass();
    return y;
}
```

CH0\ParameterSamples\ParameterSample05

Out 宣告

- 參數宣告為 `out` 會強迫該方法實作內部一定要產生物件
- 例如: `xxx.TryParse` 方法

params 宣告

- `params` 關鍵字可讓您指定 方法參數，這種參數可以採用可變數目的引數
- 一個方法宣告中的 `params` 關鍵字之後不可再有其他參數，且一個方法宣告中只能有一個 `params` 關鍵字

CH0\ParameterSamples\ParameterSample06

抽象、虛擬、覆寫與密封

抽象方法

- `abstract`
 - 只能用在抽象類別
 - 方法不提供實作，非抽象衍生類別必須覆寫此方法
 - 隱含 `virtual`

虛擬方法

- `virtual`

- 虛擬方法的實作則可由衍生類別所取代。取代繼承之虛擬方法實作的流程，稱為覆寫方法

覆寫方法

▪ override

- 被覆寫的基底方法必須是虛擬、抽象或覆寫的執行個體方法。換言之，覆寫基底方法不能為靜態或非虛擬。
- 被覆寫的基底方法不能為密封方法。
- 覆寫宣告和覆寫基底方法有相同的傳回型別。
- 覆寫宣告和覆寫基底方法擁有相同的宣告存取層級。換言之，覆寫宣告不能更改虛擬方法的存取層級。

密封方法

- `sealed`
 - 防止衍生類別覆寫該方法
 - 如果執行個體方法宣告包含 `sealed` 修飾詞，它同時也必須包含 `override` 修飾詞。

- abstract
- virtual
- override
- ~~override sealed~~

override method



CH0\MethodModifierSamples\MethodModifierSample01

覆寫與遮蔽

- 使用 `new` 宣告遮蔽方法
- 遮蔽方法與覆寫方法的不同
- 使用情境

方法多載

- 同樣的方法名稱，不同的參數清單
- 覆寫+多載

委派

委派

- 委派是一種方法簽章的型別
- C# 中委派的觀念類似於 C++ 的函式指標
- 可以透過委派叫用 (Invoke) 或呼叫方法
- 委派可以用來將方法當做引數傳遞給其他方法
- C# 中的委派是多重的 (鏈式委派)

委派宣告

宣告一個 `SomeAction` 委派型別

▪ delegate

```
public delegate void SomeAction(string message);
static void Main(string[] args)
{
    SomeAction action = ShowMessage;
    action.Invoke("Test");
    Console.ReadLine();
}

public static void ShowMessage(string message)
{
    Console.WriteLine(message);
}
```

CH0\DelegateSamples\DelegateSample01

MulticastDelegate 類別

- 表示多重傳送的委派 (Delegate)；也就是說，委派可以在它的引動過程清單中包含一個以上的項目
- MulticastDelegate 為特殊類別。編譯器 (Compiler) 和其他工具可以衍生自這個類別，但是您無法明確衍生自這個類別
- MulticastDelegate 具有由一個或多個項目組成的委派連結串列 (Linked List)，稱為引動過程清單。當叫用 (Invoke) 多點傳送委派時，依照顯示的順序同步呼叫引動過程清單中的委派

多重委派

```
public delegate void SomeAction(string message);
static void Main(string[] args)
{
    SomeAction action = ShowMessage;
    action += ShowText;
    action.Invoke("Test");
    Console.ReadLine();
}

public static void ShowMessage(string message)
{
    Console.WriteLine("ShowMessage :" + message);
}

public static void ShowText(string text)
{
    Console.WriteLine("ShowText :" + text);
}
```

CH0\DelegateSamples\DelegateSample01

利用委派傳遞方法

```
public delegate void SomeAction(string message);

public class Class1
{
    public void DoAction(SomeAction action, string message)
    {
        action.Invoke(message);
    }
}
```

CH0\DelegateSamples\DelegateSample02

```
class Program
```

```
{
```

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

```
    {
```

```
        Class1 obj = new Class1();
```

```
        SomeAction a = Show;
```

```
        obj.DoAction(a, "pass delegate");
```

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

```
    }
```

```
    public static void Show(string text)
```

```
    {
```

```
        Console.WriteLine("Show " + text);
```

```
    }
```

```
}
```

GetInvocationList

```
class Program
{
    private delegate int SomeDelegate();

    static void Main(string[] args)
    {
        SomeDelegate method = Method01;
        method += Method02;
        method += Method03;
        int value = method.Invoke();
        Console.WriteLine("Result : " + value.ToString());
        Console.ReadLine();

        foreach (var d in method.GetInvocationList())
        { Console.WriteLine(d.DynamicInvoke()); }
        Console.ReadLine();
    }
}
```

CH0\DelegateSamples\DelegateSample03

Action 與 Func

Action 委派

- Action 委派是一系列無回傳值宣告的委派
- 傳入參數的數量從 0 到 16
- 傳入的泛型參數支援逆變性

Func 委派

- Func 委派是一系列具有泛型回傳值宣告的委派
- 傳入參數的數量從 0 到 16
- 傳入的泛型參數支援逆變性
- 傳出的泛型回傳值支援共變性

事件

事件

- 事件可讓類別或物件在某些相關的事情發生時，告知其他類別或物件
- 傳送（或「引發」(Raise)）事件的類別稱為「發行者」(Publisher)，而接收（或「處理」(Handle)）事件的類別則稱為「訂閱者」(Subscriber)
- 事件與事件委派函式

基本宣告

```
public class Class1
{
    public event EventHandler XChanged;
    private void OnXChanged()
    {
        if (XChanged != null)
        { XChanged(this, new EventArgs()); }
    }
    private int _x;
    public int X
    {
        get { return _x; }
        set
        {
            if (_x != value)
            {
                _x = value;
                OnXChanged();
            }
        }
    }
}
```

CH0\EventSamples\EventSample01

帶有資料的宣告

- 自訂委派
- 使用 `EventHandler<T>`

自訂 EventArgs

```
public class CustomEventArgs : EventArgs
{
    public int OldValue
    { get; set; }
    public int NewValue
    { get; set; }
}
```

自訂委派

```
public event CustomEventHandler XChanged;

private void OnXChanged(int oldValue, int newValue)
{
    if (XChanged != null)
    {
        XChanged(this, new CustomEventArgs()
        {
            OldValue = oldValue, NewValue = newValue
        });
    }
}
```

CH0\EventSamples\EventSample02

使用 EventHandler<T> 替代自訂委派

```
public class Class1
{
    public event EventHandler<CustomEventArgs> XChanged;
    private void OnXChanged(int oldValue, int newValue)
    {
        if (XChanged != null)
        {
            XChanged(this, new CustomEventArgs()
            {
                OldValue = oldValue, NewValue = newValue
            });
        }
    }
}
```

CH0\EventSamples\EventSample03

Framework 版本的差異

- 2.0~4.0

[SerializableAttribute]

public delegate void EventHandler<TEventArgs>

(Object sender, TEventArgs e)

where TEventArgs : EventArgs

- 4.5

[SerializableAttribute]

public delegate void EventHandler<TEventArgs>

(Object sender, TEventArgs e)

建構式

建構式

- 類別 或 結構 建立時，它的建構函式呼叫。 建構函式的名稱與類別或結構相同，因此，它們通常用來初始化新物件的資料成員
- 不使用任何參數的建構函式稱為「預設建構函式」(Default Constructor)。 每當使用 `new` 運算子來具現化物件，而且未提供引數給 `new` 時，便會叫用預設建構函式
- 建構式不會繼承
- 抽象類別的建構式通常為 `protected`

```
public class Car
{
    protected int _wheels;
    public Car()
    { _wheels = 4; }
}

public class Coupe : Car
{
    public Coupe()
    { Console.WriteLine("Coupe" + _wheels.ToString()); }
}

public class Truck : Car
{
    public Truck(int wheels)
    {
        _wheels = wheels;
        Console.WriteLine("Truck: " + _wheels.ToString());
    }
}
```

```
public class Car
{
    protected int _wheels;
    public Car() : base()
    { _wheels = 4; }
}

public class Coupe : Car
{
    public Coupe() : base()
    { Console.WriteLine("Coupe" + _wheels.ToString()); }
}

public class Truck : Car
{
    public Truck(int wheels) : base()
    {
        _wheels = wheels;
        Console.WriteLine("Truck: " + _wheels.ToString());
    }
}
```

```
public class Airplane
{
    protected string _engine;
    public Airplane(string engine)
    {
        _engine = engine;
    }
}
```

```
public class Fighter : Airplane
{
    public Fighter()
    {
        _engine = "噴射引擎";
    }
}
```

```
public class Airplane
{
    protected string _engine;
    public Airplane(string engine)
    {
        _engine = engine;
    }
}
```

```
public class Fighter : Airplane
{
    public Fighter() : base("噴射引擎")
    { }
}
```

```
public class Bomber : Airplane
{
    public Bomber(string engine) : base(engine)
    { }
}
```

基底類別沒有無參數建構式，衍生類別必須要明確呼叫基底類別建構式

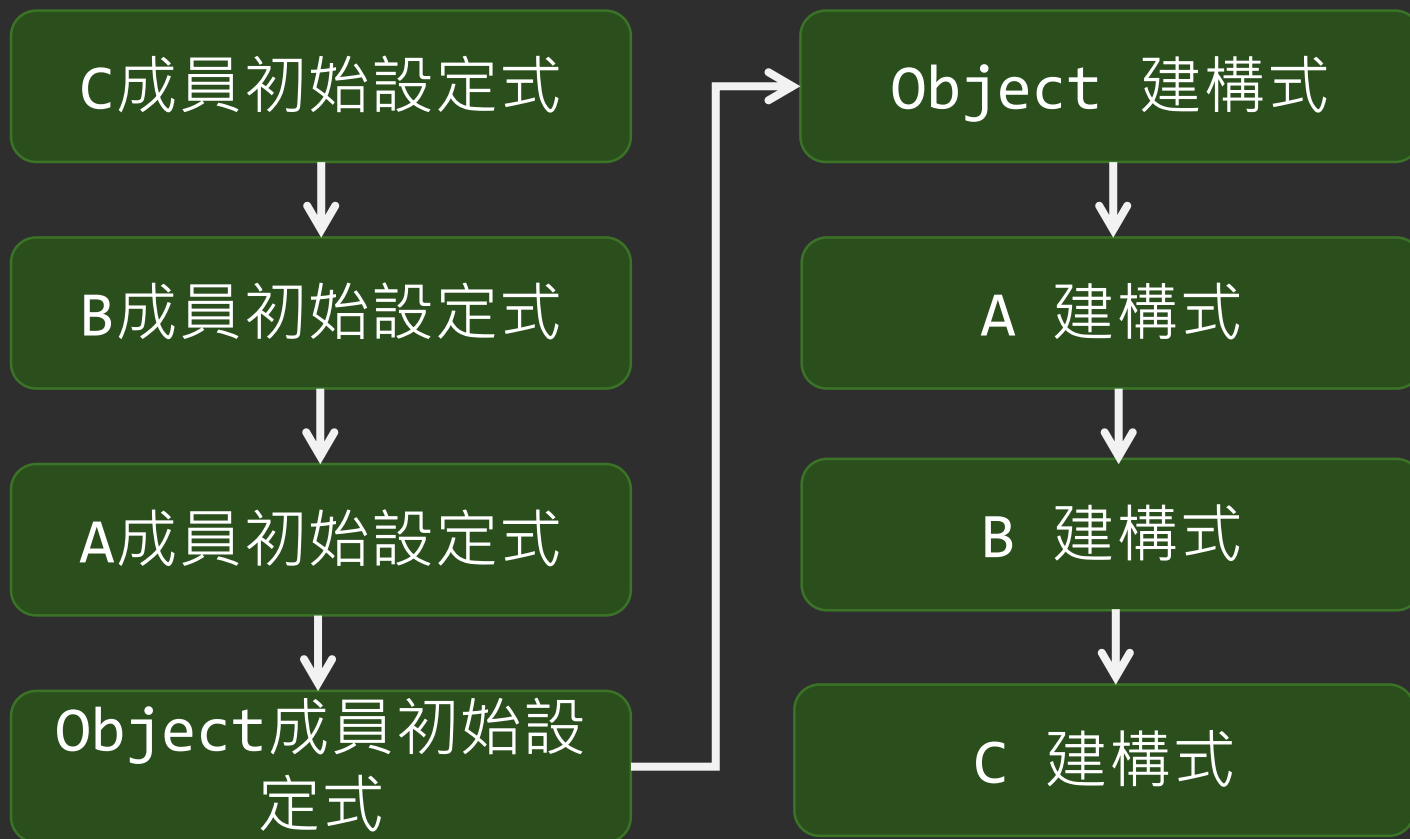
類別內部建構式呼叫

```
protected int _wheels;  
protected int _displacement;  
  
public Truck()  
    : this(8, 3500)  
{ }  
  
public Truck(int wheels)  
    : this(wheels, 3500)  
{ }  
  
public Truck(int wheels, int displacement)  
{  
    _wheels = wheels;  
    _displacement = displacement;  
}
```

CH0\ConstructorSamples\ConstructorSample03

繼承鏈上的建構式呼叫順序

繼承鍊：System.Object -- A -- B -- C



注意

避免建構式呼叫虛擬方法

```
public class Wheel
{
    public int Wheels
    { get; set; }
}

public class Car
{
    private Wheel _wheelsA;
    public Car()
    { _wheelsA = new Wheel(); _wheelsA.Wheels = 4; Initial(); }
    protected virtual void Initial()
    { Console.WriteLine("Car :" + _wheelsA.Wheels.ToString()); }
}

public class Truck : Car
{
    private Wheel _wheelsB;
    public Truck()
    { _wheelsB = new Wheel(); _wheelsB.Wheels = 10; }
    protected override void Initial()
    { Console.WriteLine("Truck :" + _wheelsB.Wheels.ToString()); }
}
```

CH0\ConstructorSamples\ConstructorSample04

靜態類別

靜態成員

靜態類別

- 只包含靜態成員
- 無法產生實體
- 一定是密封的，無法被繼承
- 基底類別只能是 `Object` Type
- 沒有執行個體建構函式

靜態建構函式

- 靜態建構函式可以用來初始化任何靜態資料，或執行只需執行一次的特定動作。在建立第一個執行個體或參考任何靜態成員之前，會自動呼叫靜態建構函式。
- 靜態建構函式並不使用存取修飾詞，也沒有參數
- 在建立第一個執行個體或參考任何靜態成員之前，就會自動呼叫靜態建構函式以初始化類別

- 不能直接呼叫靜態建構函式
- 使用者無法控制程式中靜態建構函式執行的時間
- 靜態建構函式通常用在當類別使用記錄檔，而建構函式被用來將項目寫入該檔案
- 如果靜態建構函式擲回例外狀況，執行階段將不會再一次叫用它，且在您的程式執行的應用程式定義域存留期中，型別都將保持未初始化狀態

靜態成員

- 除非透過執行個體，否則靜態成員是無法直接存取執行個體成員
- 靜態方法能多載但不能覆寫
- C# 不支援靜態區域變數
- 靜態方法與執行個體方法的選擇

擴充方法

擴充方法

- 擴充方法讓您能將方法「加入」至現有類型，而不需要建立新的衍生類型 (Derived Type)、重新編譯，或是修改原始類型。擴充方法是一種特殊的靜態方法，但是需將它們當成擴充類型上的執行個體方法 (Instance Method) 來呼叫。

CH0\ExtensionSamples\NotExtensionSample

實做擴充方法

```
public static class ExtensionClass  
{  
    public static string[] Splitline(this string str)  
    {  
        return str.Split(new string[] { Environment.NewLine },  
            StringSplitOptions.None);  
    }  
}
```

CH0\ExtensionSamples\ExtensionSample01

情境與注意事項

CH0\ExtensionSamples\ExtensionSample02

介面

概觀

- 單一繼承 + 多介面實作
- 介面是一系列方法、屬性、事件與索引的簽章
- 介面不能定義執行個體欄位與建構式
- C# 不允許在介面中定義靜態成員
- 介面可視為一種契約
- 介面的設計要儘量簡單

CH0\InterfaceSamples\InterfaceSample01

明確實作介面成員

- 出現相同簽章成員時
- 明確實作介面的成員必須在變數型別為此介面型別時才能呼叫
- 明確實作的成員不需要寫存取修飾詞

CH0\InterfaceSamples\InterfaceSample02~03

為現有類別建立介面

泛型

(Generic Type)

概觀

- .Net Framework 2.0後才出現泛型
- 泛型將型別參數的概念引進 .NET Framework 中，使得類別和方法在設計時，可以先行擱置一個或多個型別規格，直到用戶端程式碼對類別或方法進行宣告或執行個體化時再行處理
- 泛型是強型別的概念
- 避免容器操作的 Boxing 與 Unboxing

應用面

- 泛型介面
 - `interface Itest<T>`
- 泛型類別
 - `class Test<T>`
- 泛型方法
 - `void Test<T> (T value)`
 - `T Test<T>()`
- 泛型委派
 - `delegate void Del<T>(T item)`

default 關鍵字

- 參考型別的 `null`
- 實質型別的 `0`
- 當沒有條件約束時泛型如何正確回傳

泛型條件約束

- 定義泛型類別時，可限制用戶端程式碼在執行個體化類別時的型別
- 使用 `where` 內容關鍵字指定條件約束
- `where T: struct`
 - 型別引數必須是實值型別
- `where T : class`
 - 型別引數必須是參考型別
- `where T : new()`
 - 型別引數必須擁有公用的無參數建構函式
- 將 `new()` 條件約束與其他條件約束一起使用時，一定要將其指定為最後一個

- `where T : <base class name>`
 - 型別引數必須本身是指定的基底類別，或衍生自該類別
- `where T : <interface name>`
 - 型別引數必須本身是指定的介面，或實作該介面
- `where T : U`
 - 提供給 `T` 的型別引數必須是（或衍生自）提供給 `U` 的引數

不變性 Invariant

共變性 Covariant

逆變性 Contravariant

什麼是共變與逆變

- 共變性
 - 用基底類別取代衍生類別
- 逆變性
 - 用衍生類別取代基底類別

```
public class Gen0
{ public int x; }

public class Gen1 : Gen0
{ }
```

```
static void Main(string[] args)
{
    // 共變
    //Gen1 obj = OutMethod();
    Gen0 obj = OutMethod();

    //逆變
    //InMethod(new Gen0());
    InMethod(new Gen1());
}

private static Gen1 OutMethod()
{ return new Gen1(); }

private static void InMethod(Gen0 obj)
{ }
```

變異性與型別安全

- 何謂型別安全
- 變異性對型別安全的影響
- 泛型中的變異性

CH0\VarianceSamples\VarianceSample02

繼承多型、泛型 與多載的選擇

Comparable<T>

- Comparable<T>.CompareTo 方法
 - 小於零 : 這個物件小於 other 參數
 - 零 : 這個物件等於 other
 - 大於零 : 這個物件大於 other

int CompareTo(T other)

CH0\IComprarableSamples\IComprarableSample01~02

變異性與委派

- 委派的回傳型別支援共變性
- 委派的參數型別支援逆變性
 - 代表：事件的委派函式可以宣告為委派所規定型別的衍生型別

```
public partial class Form1 : Form
{

    public Form1()
    {
        InitializeComponent();
        button1.MouseClick += button1_Click;
    }

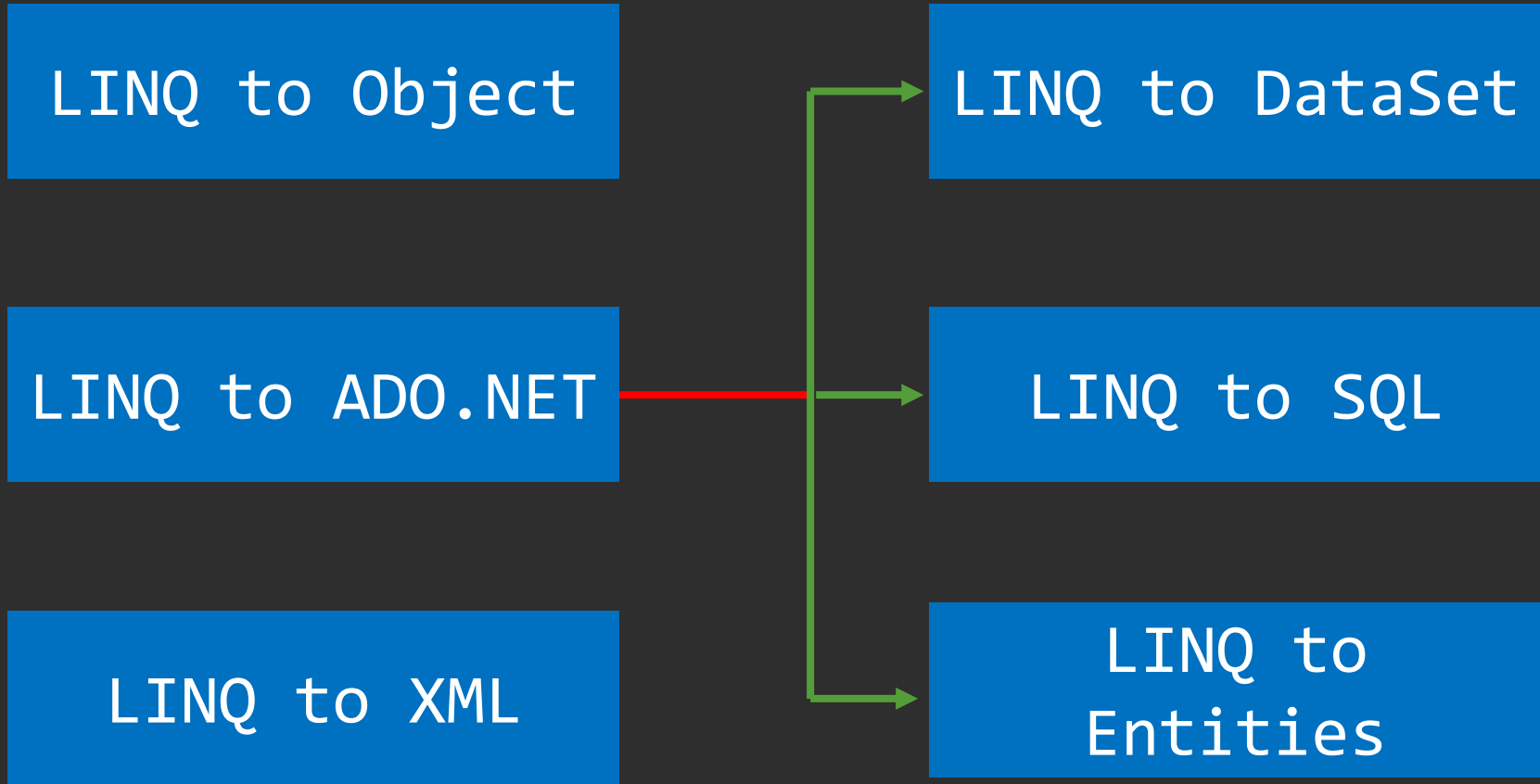
    void button1_MouseClick(object sender, MouseEventArgs e)
    {
        MessageBox.Show("Mouse Click");
    }

    void button1_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Click");
    }

}
```


Lambda 簡介與原理

LINQ Framework



從 Enumerable.Where 談起

```
public static IEnumerable<TSource>  
    Where<TSource>  
( this IEnumerable<TSource> source, Func<TSource, bool> predicate )
```

```
Li st<string> data = new Li st<string>()  
{  
    "bi ll",  
    "davi d",  
    "j ohn"  
};  
var resul t = data.Where((x) => x == "davi d");
```

沒有語法糖的傳統做法

```
private bool TestMethod(string value)
{
    return (value == "david");
}

private void button2_Click(object sender, EventArgs e)
{
    // 最傳統的寫法 Boolean Func<T,Boolean>(T value)
    Func<string, bool> TestDelegate;
    TestDelegate = TestMethod;
    var result = data.Where(TestDelegate);
}
```

CH0\LambdaSamples\LambdaSample01

從匿名委派到Lambda

```
Where(delegate (string x) { return (x == "david"); });
```

參數

方法內容區段

```
Where((x) => { return x == "david"; });
```

參數

較為簡單的方法區段
可省略 return

```
Where((x) => x == "david");
```

迭代器 (Iterator)

yield

- 在陳述式中使用 `yield` 關鍵字時，表示關鍵字所在的方法、運算子或 `get` 存取子是迭代器。
- `yield return <expression>`
- `yield break`
- `yield return` 陳述式到達時，目前在程式碼的位置會被記住。下一次呼叫此 `Iterator` 時，便會從這個位置重新開始執行。

- 宣告需求
 - 傳回類型必須是 IEnumerable、IEnumerable<T>、IEnumerator 或 IEnumerator<T>
 - 宣告不可包含任何 ref 或 out 參數

迭代器實作

CH0\IteratorSamples\IteratorSample01
CH0\IteratorSamples\IteratorInPropertySample

foreach-in

- 如非必要，優先採用 foreach 替代 for
- CLR 對 foreach 的處理優化
- 不可在 foreach 區塊內修改來源集合

自己寫個 Where ?

CH0\LambdaSamples\CustomWhereSample

索引子

索引子

- 使用與陣列相同的方式來索引類別（**Class**）或結構（**Struct**）的執行個體。
- this 關鍵字的用途為定義索引子。
- 索引子不需要以整數值來索引；您可以決定如何定義特定的查詢機制。
- 索引子可以多載。
- 索引子可以具有一個以上的型式參數，例如，在存取二維陣列時便是如此。
- 介面中也可以宣告索引子

CH0\IndexerSamples\IndexerSample01

Read Only Collection

CH0\IndexerSamples\ReadOnlyCollectionSample

類別設計

流程

需求

職責

抽象



高內聚

低耦合

技巧

- 用圖畫分析你的概念
- 說的一嘴好程式
- 想像力是你的超能力
- 進化的重構

類別與介面的選擇

- 抽象類別的重點在於重用性設計
- 介面設計著重的則是抽象程度
- 重用與彈性 / 血統與能力

示範 巢狀重構

巢狀判斷式

- 巢狀判斷式易讀性不佳
- 巢狀判斷式的彈性不足
- 如果，你的程式中需要依序判斷許多的條件？

假設一個巢狀判斷，你需要判斷 `List<string>` 中的
第一個字串是不是 `Dog`，如果 `True` 則繼續，`False` 則跳出
第二個字串是不是 `Cat`，如果 `True` 則繼續，`False` 則跳出
第三個字串是不是 `Apple`，如果 `True` 則繼續，`False` 則跳出
第四個字串是不是 `House`，如果 `True` 則繼續，`False` 則跳出
第五個字串是不是 `Car`，如果 `True` 則繼續，`False` 則跳出
第六個字串是不是 `Taxi`，如果 `True` 則繼續，`False` 則跳出

接著依序判斷 `List<int>` 中的值不符合 `1,4,8,9,77`

CH0\NestedConditionRefactorSamples\NestedConditionRefactorSample01

```

function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}

```



簡化判斷

```
public interface ICheckData
{
    bool GetResult();
}

public class CheckData<T> : ICheckData where T : IComparable<T>
{
    private T _source;
    private T _target;

    public CheckData(T source, T target)
    {
        _source = source;
        _target = target;
    }
    public bool GetResult()
    {
        return (_source.CompareTo(_target) == 0);
    }
}
```

CH0\NestedConditionRefactorSamples\NestedConditionRefactorSample02

實作 BMI

BMI 計算

你會怎麼設計 ?

- 公式 : kg/m^2
- Man
 - BMI < 20 -> 結果字串為 “太瘦”
 - BMI > 25 -> 結果字串為 “太胖”
 - 中間值結果字串為 “適中”
- Woman
 - BMI < 18 -> 結果字串為 “太瘦”
 - BMI > 22 -> 結果字串為 “太胖”
 - 中間值結果字串為 “適中”

類別庫

善用命名空間

- 使用命名空間組織其多種類別
- 宣告自己的命名空間，將有助於在較大型的程式設計專案中控制類別和方法名稱的範圍

Blog 是記錄知識的最佳平台



<https://dotblogs.com.tw>

OzCode

Your Road to Magical Debugging



```
float CalculateCost( Customer customer, string restaurant)
{
    float courseCost = GetCourseCost(restaurant);
    bool shouldTip = waiter.IsNice && courseCost > COSTLY_MEAL;
```

Exceptions Trail:



ReservationException HotelException **IndexOutOfRangeException**

Exception:

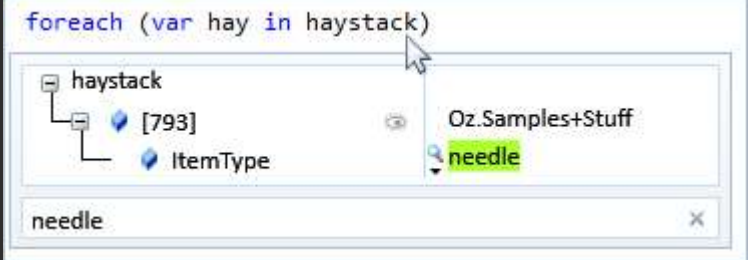
System.IndexOutOfRangeException

Message:

"Invalid customer index"

[Go to where exception was thrown](#) [Go to where exception was handled](#)

<http://www.oz-code.com/>



學員可使用 Yammer 取得優惠價

謝謝各位

<https://skilltree.my>

-
- 本投影片所包含的商標與文字皆屬原作者所有，僅供教學之用。
 - 本投影片的內容包括標誌、設計、文字、圖像、影片、聲音...等著作財產權均屬電魔小鋪有限公司所有，受到中華民國著作權法及國際著作權法律的保障。對本投影內容進行任何形式的引用、轉載、重製前，請務必取得電魔小鋪有限公司的"書面授權"，否則請勿使用，以免侵權。