SKILLTREE

OOP by C#

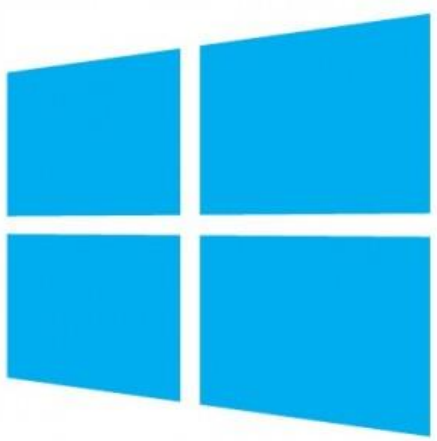
Bill Chung

關於我

- Bill Chung
- billchungiii@gmail.com
- 海角點部落
- Microsoft, Intel, 資策會 講師

我眼中的物件導向

概念性的議題



一段歷史

- 1967年大西洋公約組織
  - 為什麼軟體的發展跟不上硬體
  - 為什麼我們設計新程式的能力跟不上人們對新程式的需求？
  - 為什麼現今的程式設計是如此粗糙，以至於難以修改？

SKILLTREE

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
- .....

思考是甚麼 ？

SKILLTREE

進入物件導向的世界

物件是甚麼？

物件是
外界真實事物的抽象定義

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

程序和資料的結合

```
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();
}
```

類別與物件

- 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

抽象化

- 分類的基礎技巧就是抽象化
- 抽象化的定義
  - 找出關鍵性的特徵並加以描述
- 分類是必須的，但方式並非絕對的

抽象化的範例–地圖


Micasa Dolci
三六九
臺北君悅
大飯店
世貿中心
莊敬路
信義路五段
松壽路
NEO19
老乾杯
博愛國小
松仁路95巷
信義威
秀影城
台北展演
二館
松勇路
松勇路53巷
松智路
松仁路
信義路五段91巷
JOYCEEAST
餐廳
信義路五段
松勤街
光復南路
光復南路473巷
基隆路一段364巷
基隆路一段380巷
基隆路地下道
基隆路二段
文昌街
太平洋
商務會館
好,丘
goodchos
南村小吃店
大統領
經貿廣場
三興國小
臨江街
景平公園
松平路
松仁路153巷
信義國中
三梨公園
喜從天降
烘焙屋
莊敬路289巷
莊敬路325巷
莊敬路341巷
古早味
雞蛋糕
吳興街156巷
信安街9巷
私立仁康醫院
附設居家護理所
200 公尺
500 英呎
遠端桌面連線
A
B
H
D
F
P

OOP 三大特性

三大特性

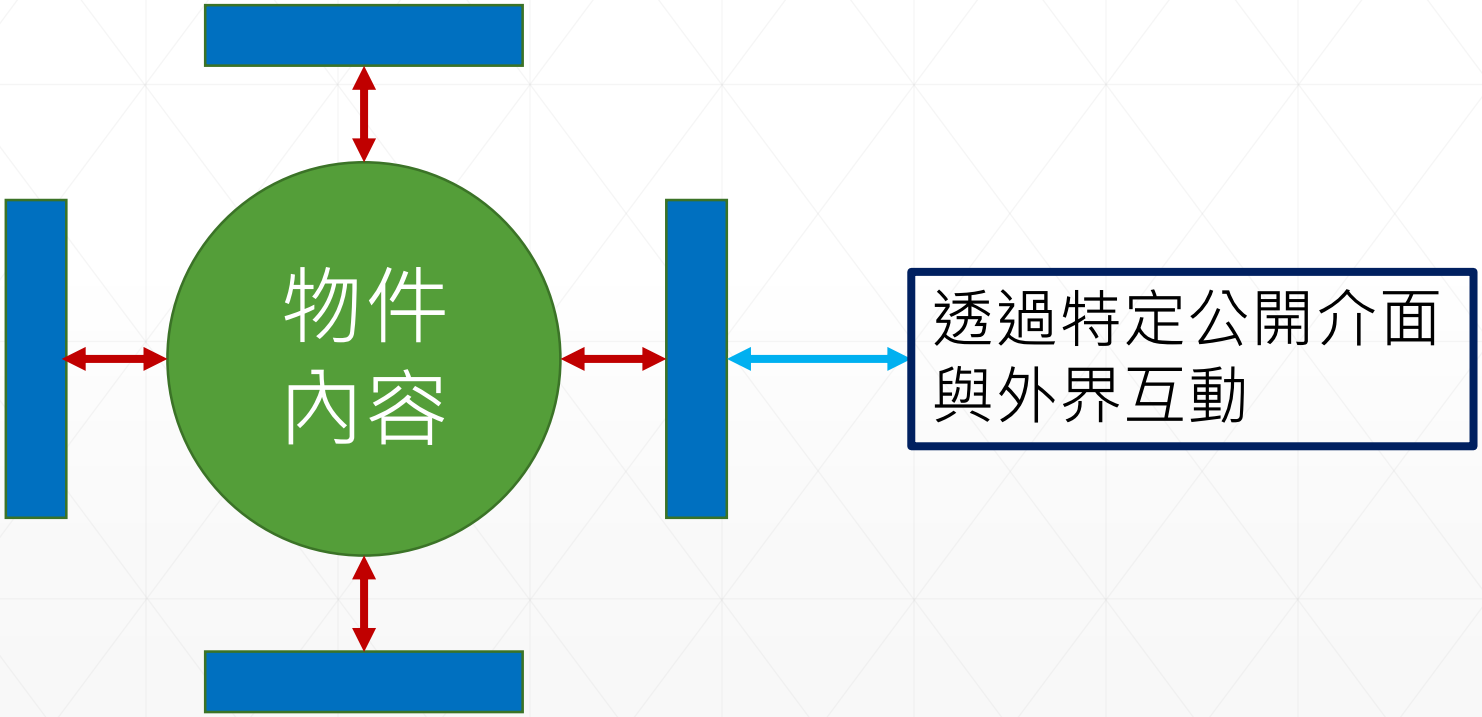
- 繼承
- 封裝
- 多型
- 絕大部分的設計就在應用這三大特性

繼承

- 繼承者會擁有被繼承者的型別特徵 (存取層級可見，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

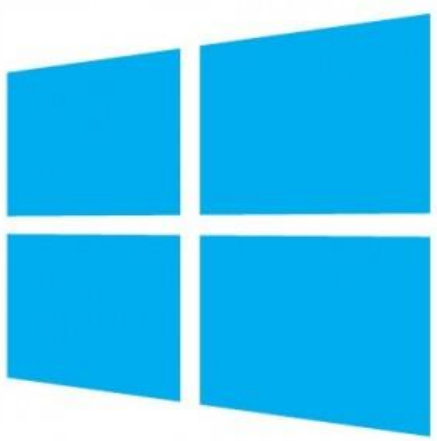
多載

- 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

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？

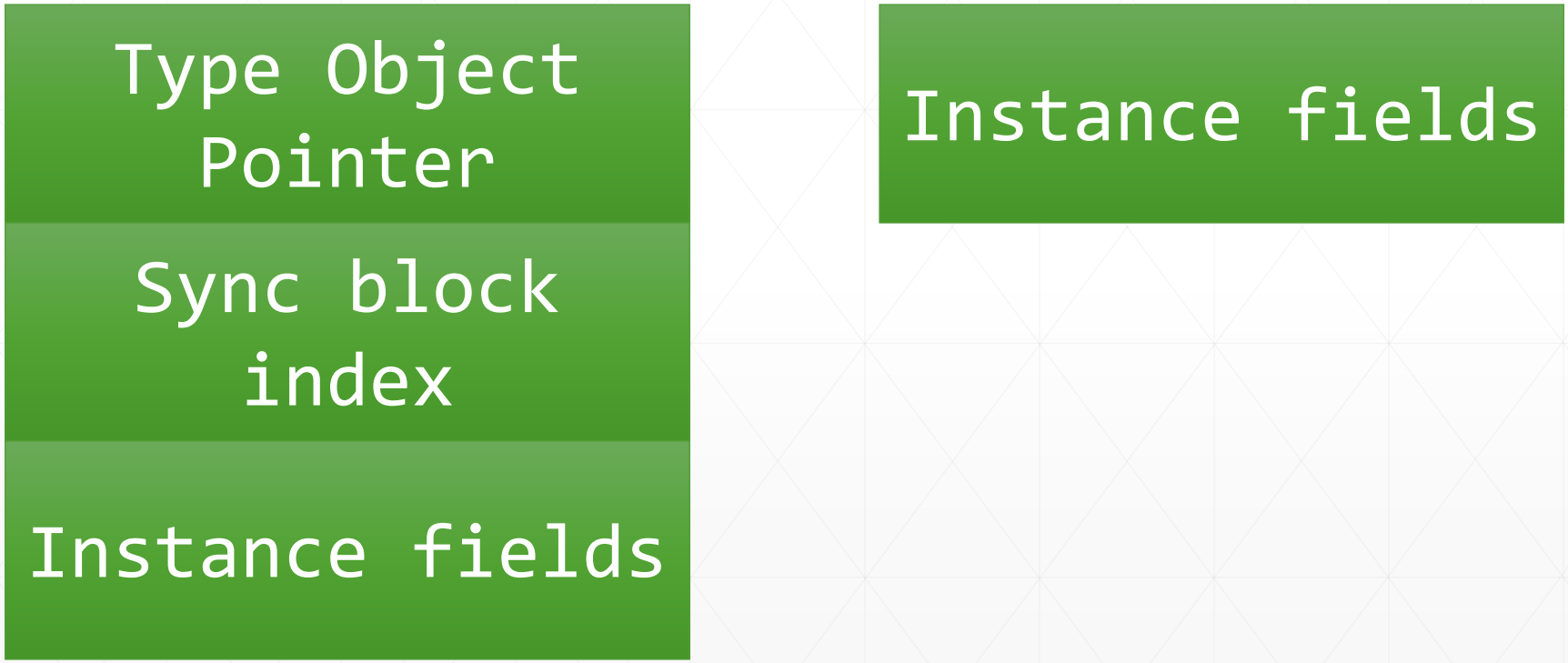
型別概論

- Primitive Type
- Reference Type
- Value Type

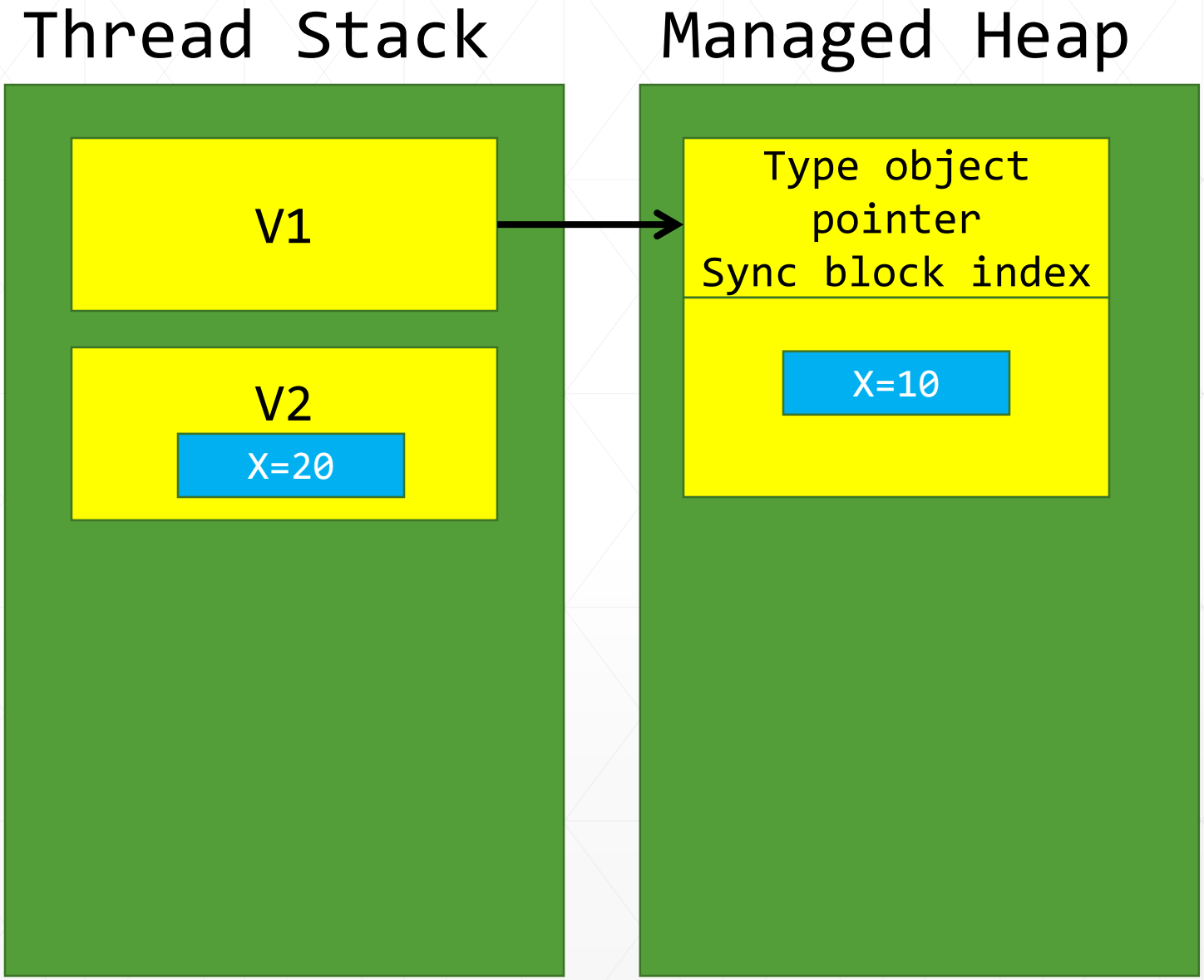
參考型別 vs 實值型別

- 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



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

   public struct MyValStruct
   { public int x; }

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


Thread Stack
Managed Heap
V1
V2
X=20
Type object pointer
Sync block index
X=10

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

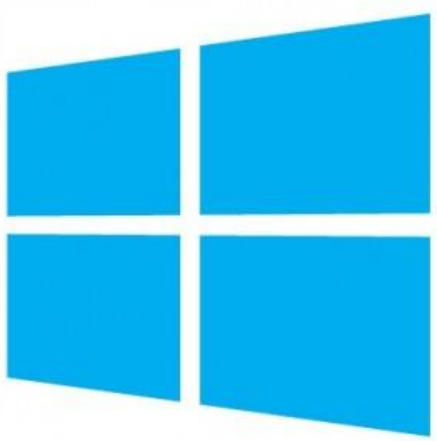
必須使用 var 的場合

```
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();
}
```

觀念

- 參考型別物件(執行個體)與變數的關係
- 參考型別物件(執行個體)的型別和變數一定要相同嗎？

實質型別

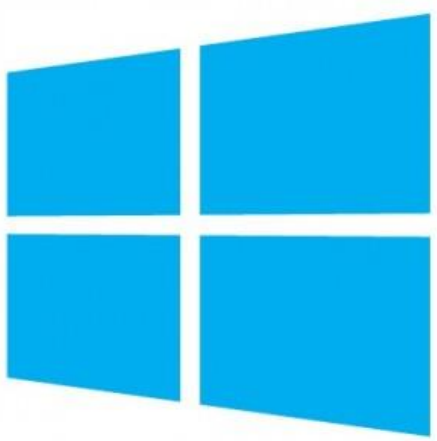


特徵

- 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

Boxing 無所不在

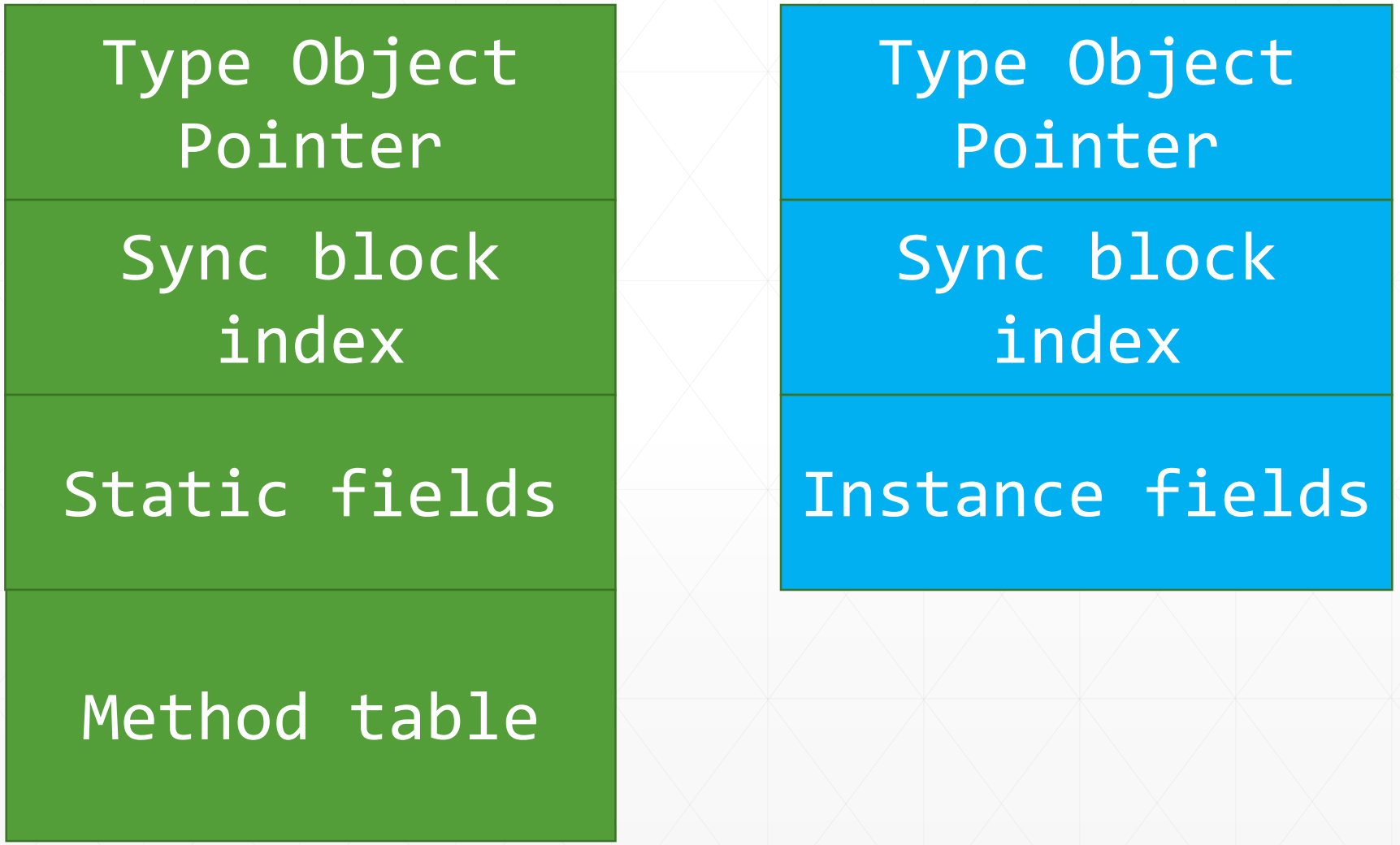
類別



類別成員

- 常數
- 欄位
- 屬性
- 方法
- 事件
- 建構式 ...

型別物件與執行個體物件比較



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

SKILLTREE

常數
欄位
屬性

常數

- 在編譯時期就會使用常數值取代
- 執行時期無法變更
- 使用常數注意事項

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

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
- 自動實做屬性

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

        public int GetX()
        { return _x; }

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

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

SKILLTREE

方法

方法

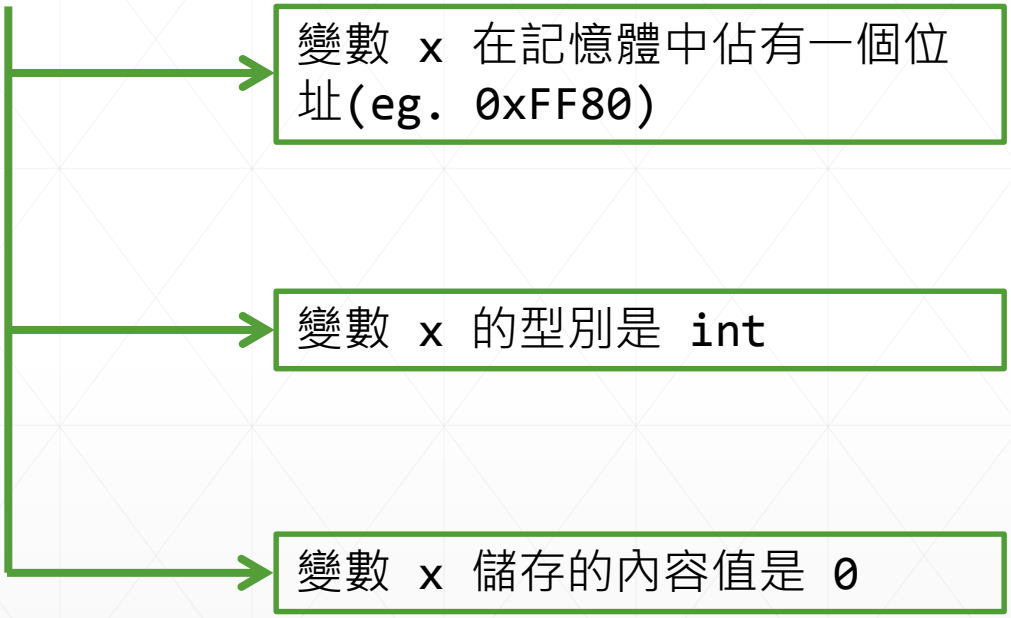
- 「方法」(Method) 是包含一系列陳述式 (Statement) 的程式碼區塊。 程式會「呼叫」(Calling) 方法並指定所有必要的方法引數，藉以執行陳述式
- 方法參數的重要關鍵字
  - ref
  - out
  - params

傳值與傳址

- 倒底傳值與傳址的主詞是誰？

實質型別的變數

```
int x = 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)
{
}
```

實值型別傳址

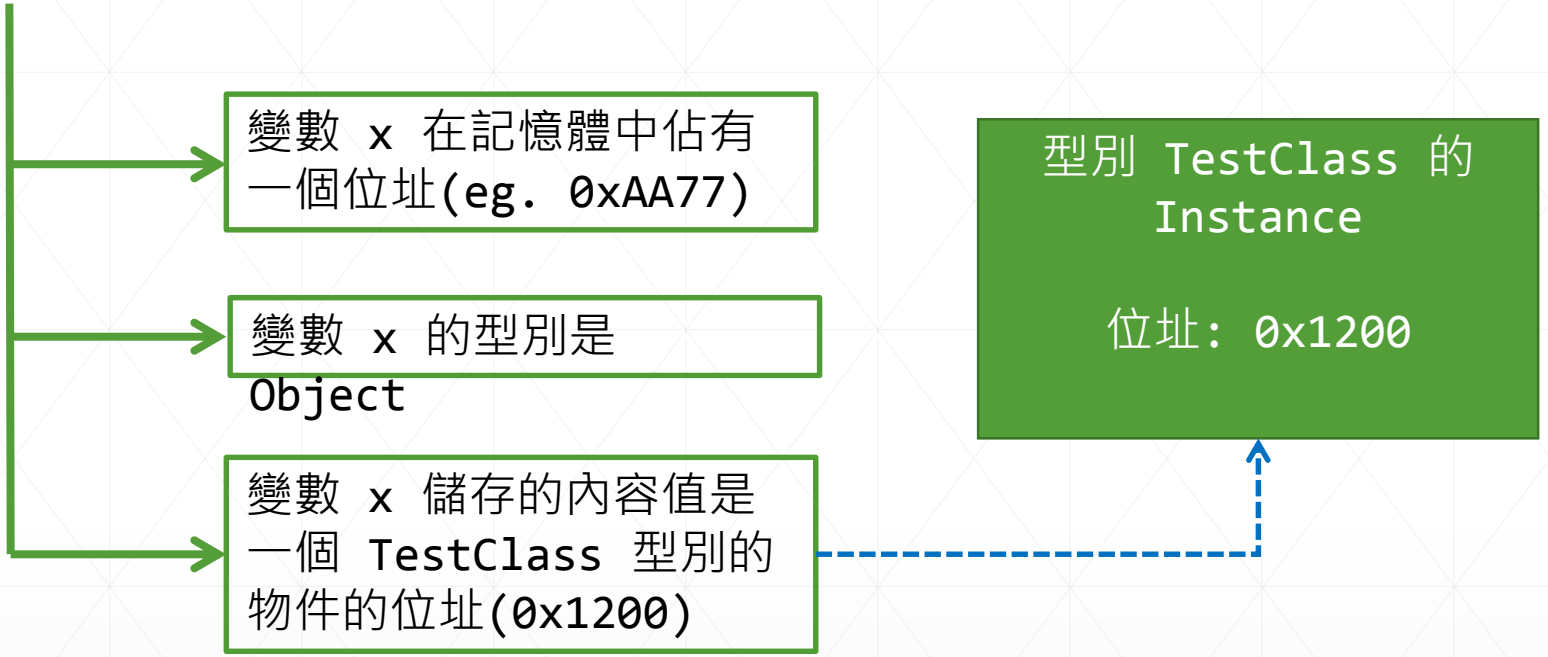
```
static void Main(string[] args)
{
    int x = 0;
    int y = ChangeX(ref x);
}
```

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)
{
}
```

參考型別變數

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



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)
{

}
```

參考型別傳址

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

private static TestClass ChangeX(ref TestClass 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)

```
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;
}
```

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
- eg: xxx.TryParse

params

- 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

特別介紹
Tuple 類別

SKILLTREE

抽象方法
虚擬方法
覆寫方法
密封方法

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

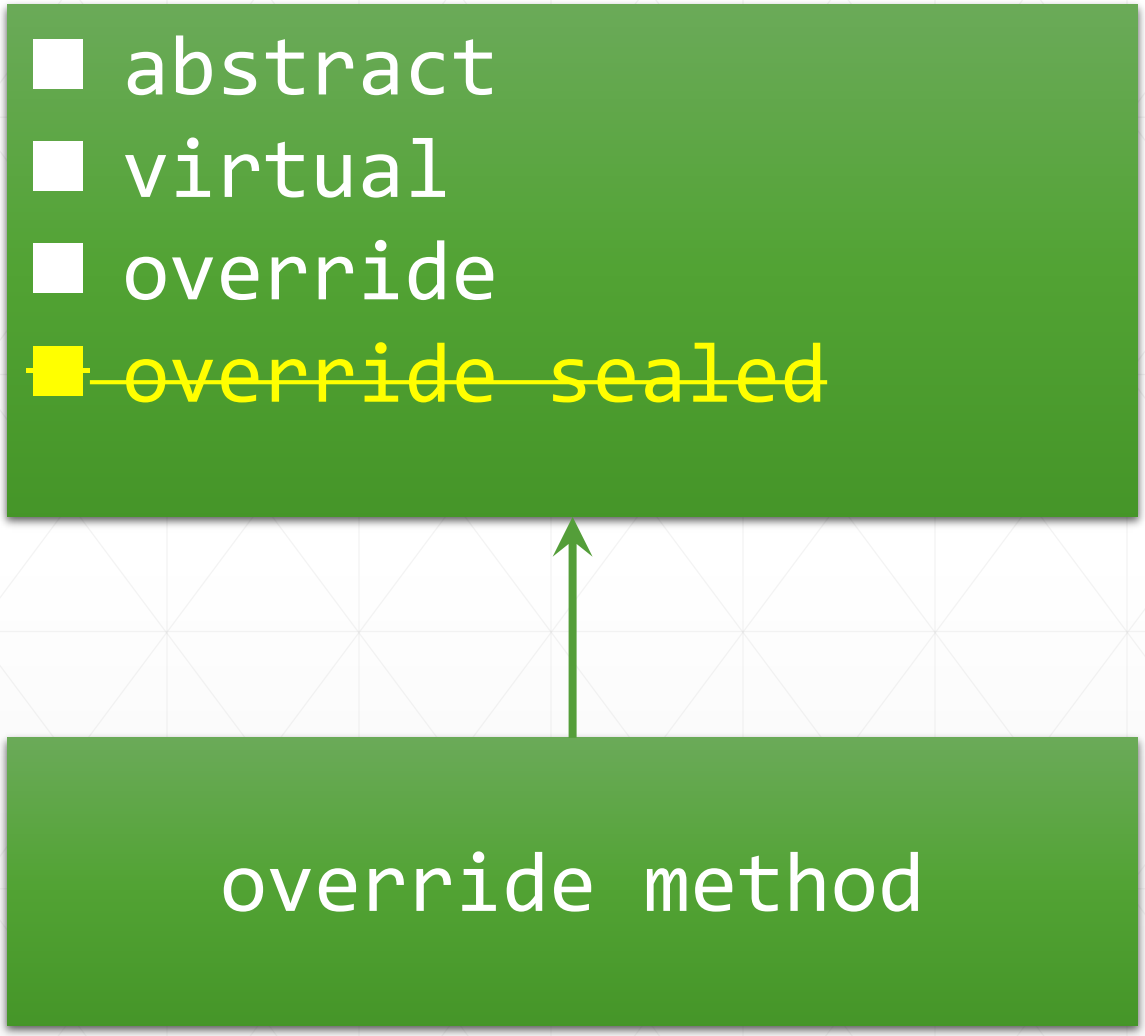
- virtual
  - 虛擬方法的實作則可由衍生類別所取代。取代繼承之虛擬方法實作的流程，稱為覆寫方法

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

SKILLTREE

覆寫與遮蔽

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

SKILLTREE

方法多載

多載

- 同樣的方法名稱, 不同的參數清單
- 覆寫+多載

SKILLTREE

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

- delegate

宣告一個 SomeAction 委派型別

```
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);
}
```

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);
}
```

傳遞方法

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

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

```
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();
    }

}
```

SKILLTREE

Action

Func

SKILLTREE

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();
            }
        }
    }
}
```

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 delegate void CustomEventHandler(Object sender, CustomEventArgs e);

    public class Class1
    {
        public event CustomEventHandler XChanged;
        private void OnXChanged(int oldvalue, int newvalue)
        {
            if (XChanged != null)
            { XChanged(this, new CustomEventArgs() { OldValue = oldvalue,
NewValue = newvalue }); }
        }


    }
```

EventHandler<T>

```
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 }); }
        }

}
```

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 )
```

事件原型

```
private List<CustomEventHandler> handlers = new
List<CustomEventHandler>();

public event CustomEventHandler XChanged
{
    add
    {
        lock(handlers)
        {
            handlers.Add(value);
        }
    }
    remove
    {
        lock (handlers)
        {
            handlers.Remove(value);
        }
    }
}
```

SKILLTREE

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
- 抽象類別的建構式會被編譯成 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()
    { _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());
    }
}
```

事實上是這樣

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

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

這樣就會出錯

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

public class Fighter : Airplane
{
    public Fighter(): base("噴射引擎")
    {
        Console.WriteLine (_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

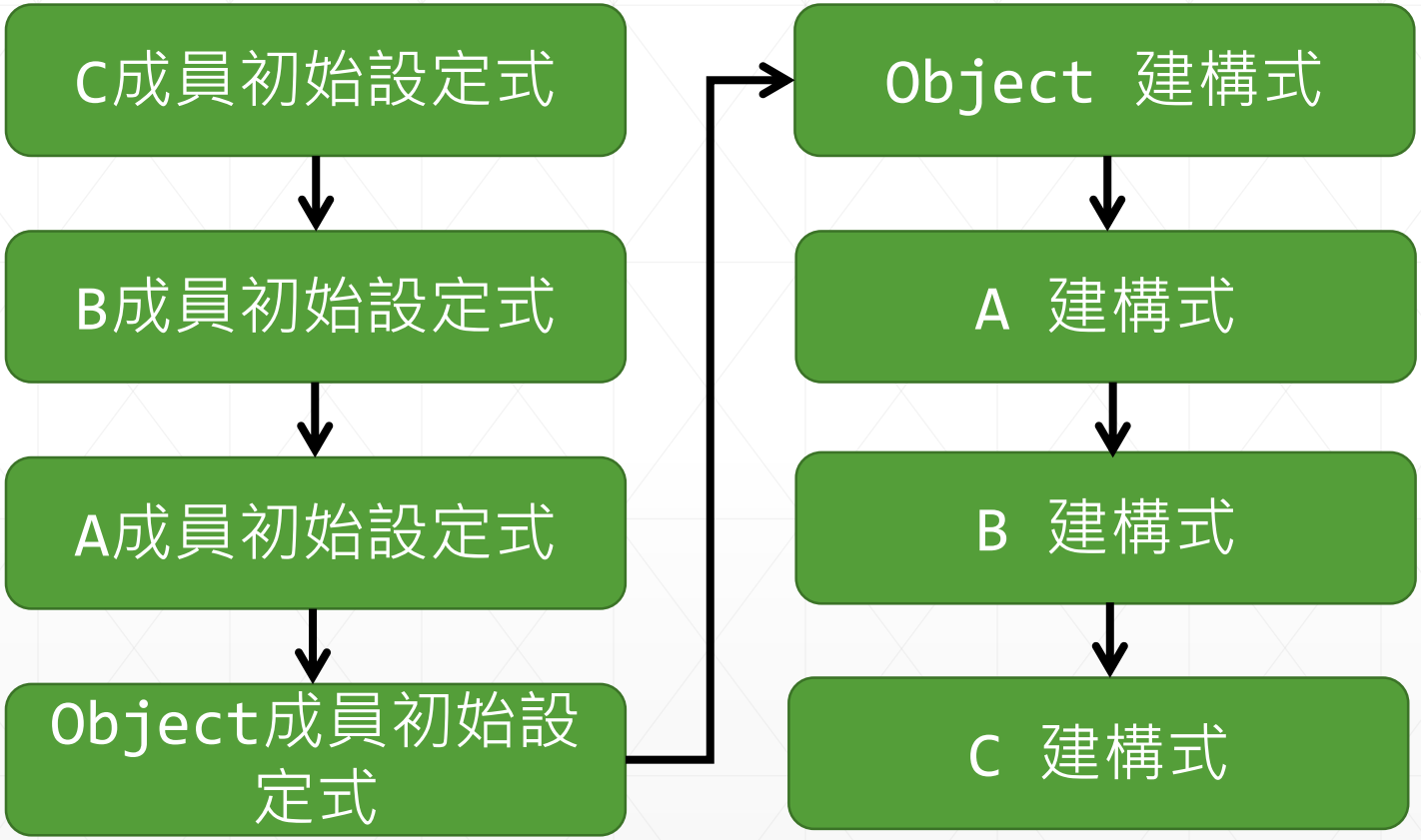
類別內部建構式呼叫

```
public class Truck
{
    protected int _wheels;
    protected int _displacement;

    public Truck()
    {
        _wheels = 8;
        _displacement = 3500;
    }

    public Truck(int wheels)
        : this()
    { _wheels = wheels; }

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

SKILLTREE

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 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()); }
}
```

SKILLTREE

存取層級

存取層級

- private
  - 存取只限於包含類別
- protected
  - 存取只限於包含的類別或衍生自包含類別的型別
- internal
  - 存取只限於目前的組件
- protected internal
  - 存取只限於目前的組件或衍生自包含類別的型別
- public
  - 存取沒有限制

規則

- 存取修飾詞不能用於命名空間, 因此命名空間沒有存取限制
- 命名空間中的最上層型別只能有 internal 或 public 存取範圍, 這些型別的預設存取範圍是 internal

善用存取層級實現封裝

SKILLTREE

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

SKILLTREE

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。

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 );
    }
}
```

呼叫擴充方法

```
String str = "ABC" + Environment.NewLine + " " +
Environment.NewLine + "CDE";

            var result = str.Splitline();
            foreach (var s in result)
            {
                MessageBox.Show(s);
            }
```

情境與注意事項

SKILLTREE

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

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

SKILLTREE

泛型

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

SKILLTREE

不變性 Invariant

共變性 Covariant

逆變性 Contravariant

共變性

- 用基底類別取代衍生類別

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

public class Gen1 : Gen0
    { }
```

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        //Gen1 obj = OutMethodx();
        Gen0 obj = OutMethodx();
    }

    private Gen1 OutMethodx()
    { return new Gen1(); }
}
```

逆變性

- 用衍生類別取代基底類別

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

public class Gen1 : Gen0
    { }
```

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        //InMethod(new Gen0());
        InMethod(new Gen1());
    }

    private 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
- 介面實作型別安全

SKILLTREE

泛型介面變異性

宣告

- Out (共變性)
- In (逆變性)

具有共變性的既有泛型介面

- IEnumerable<T>
- IEnumerator<T>
- IQueryable<T>
- IGrouping<TKey, TElement>

具有逆變性的既有泛型介面

- IComparer<T>
- IComparable<T>
- IEqualityComparer<T>

泛型與多載的選擇

IComparable<T>

- IComparable<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 )
```

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");
    }
}
```

SKILLTREE

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

SKILLTREE

類別設計

概觀

- 流程
  - 需求 → 職責 →抽象
- 目標
  - 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

SKILLTREE

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> 中的值符不符合 1,4,8,9,77

```
35          bool result = false;
36          if (list[0] == "Dog")
37          {
38              if (list[1] == "Cat")
39              {
40                  if (list[2] == "Apple")
41                  {
42                      if (list[3] == "House")
43                      {
44                          if (list[4] == "Car")
45                          {
46                              if (list[5] == "Taxi")
47                              {
48                                  if (intlist[0] == 1)
49                                  {
50                                      if (intlist[1] == 4)
51                                      {
52                                          if (intlist[2] == 8)
53                                          {
54                                              if (intlist[3] == 90)
55                                              {
56                                                  if (intlist[4] == 77)
57                                                  {
58                                                      result = true;
59                                                  }
60                                                  else
61                                                  {
62                                                      result = false;
63                                                  }
64                                              }
65                                          }
66                                      }
67                                  }
68                              }
69                          }
70                      }
71                  }
72              }
73          }
74          MessageBox.Show(" 結果是 :" + result.ToString());
```

簡化判斷

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

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

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

SKILLTREE

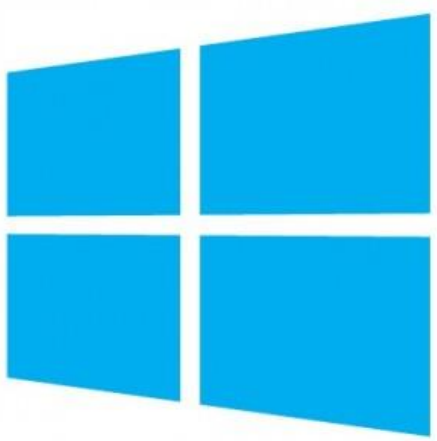
實作 BMI

BMI 計算

- 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”

你會怎麼設計 ?

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

SKILLTREE

謝謝各位

http://skilltree.my