

Build School 課程 – C# 基礎課程 06

Bill Chung
Build School 講師
V2021.2 新竹尖兵班

請尊重講師的著作權及智慧財產權!

Build School 課程之教材、程式碼等、僅供課程中學習用、請不要任意自行散佈、重製、分享，謝謝

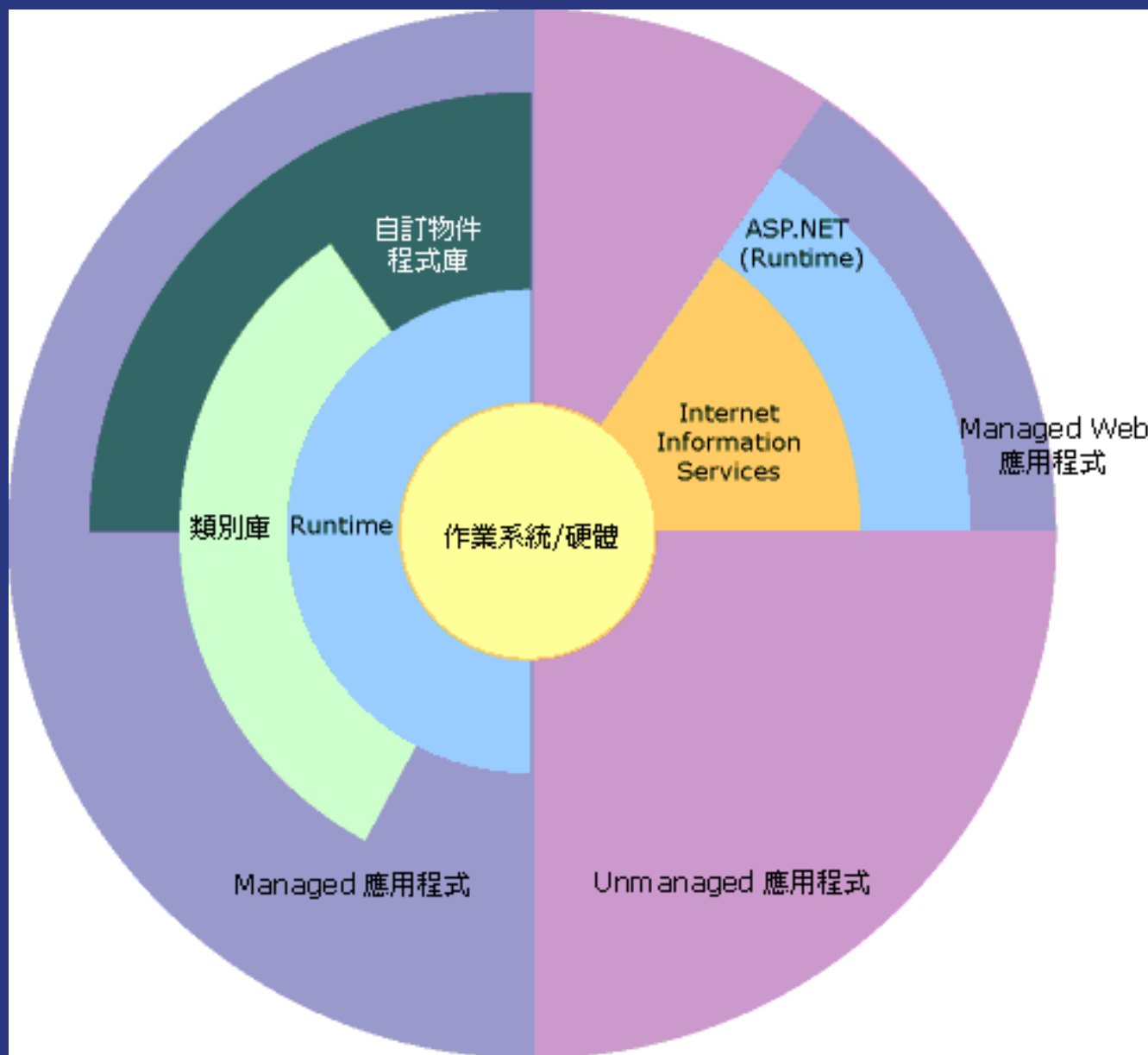
.Net Framework 概念

.Net Framework 的內容

- CLR – Common Language Runtime
- .Net Framework 類別庫

CLR

- Common Language Runtime
- 作用
 - 執行 Managed Code 中繼碼程式
 - 記憶體管理
 - 執行緒管理



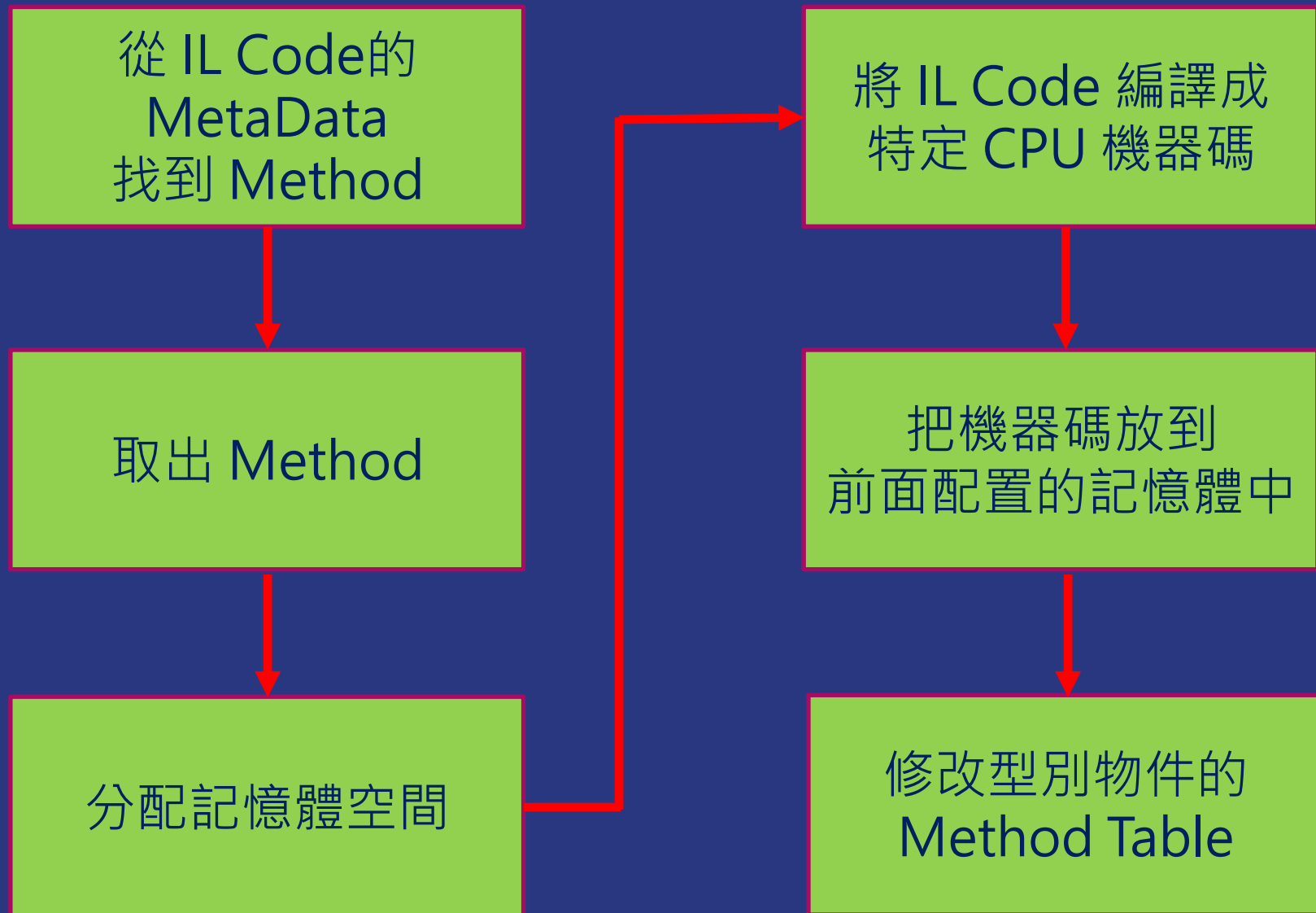
IL Code

- Microsoft Intermediate Language (MSIL)
- 與 CPU 無關的組合語言

JIT Compiler

- just-in-time compiler
- 在執行時期將 IL Code 編譯成 Native Code

JIT 如何運作



ILDASM

- ILDASM 是一個工具，可以將編譯後的 Managed Code (exe 或 dll) 反組譯成 IL Coded
- C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\ildasm.exe

EXE 與 DLL

EXE 檔

- Executable file
- EXE 是一種可以獨立執行的檔案
- 換言之，它具有程式的進入點讓 CLR 呼叫
- 常見的例子就是 Program Class 中的 static Main method

DLL 檔

- Dynamic Link Library file
- 動態鏈結函式庫檔案
- DLL 本身不具備進入點，必須由其他人呼叫

LAB

建立一個函式庫並使用它

新增一個方案及專案

- 方案名稱：DLLSamples
- 專案名稱：DLLSample001
- 範本：Console Application
- 建立好之後先放在一邊

在 DLLSamples 加入新專案

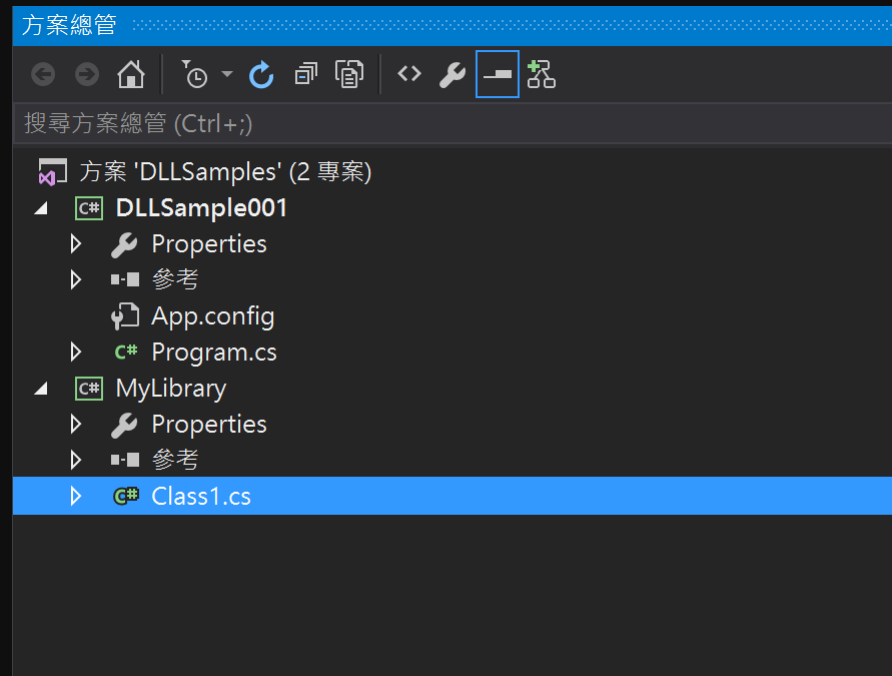
- 方案名稱： DLLSamples
- 專案名稱： MyLibrary
- 範本： Class Library
(類別庫 .NET Framework)
- 注意：請不要搞錯成 『可攜式類別庫』或 『.NET Core 類別庫』或 『.NET Standard』

Visual Studio 會很雞婆的幫你在 MyLibrary 專案 自動加入一個 Class1.cs 的類別檔案

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyLibrary
{
    public class Class1
    {

    }
}
```

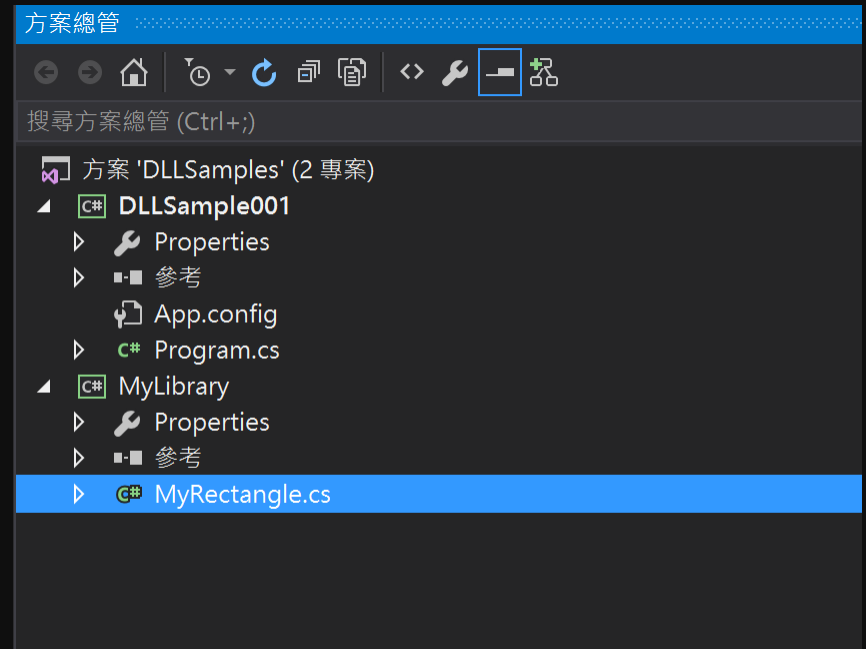


(1)修改 Class1 class 名稱並加入以下內容

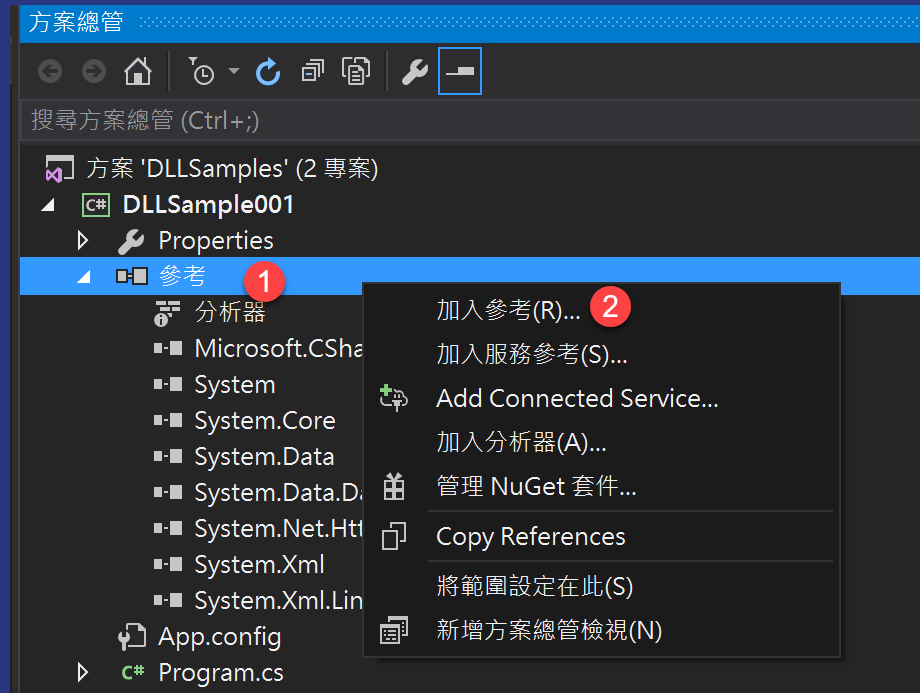
(2)然後在方案總管中把 Class1.cs 更名為 MyRectangle.cs

```
public class MyRectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

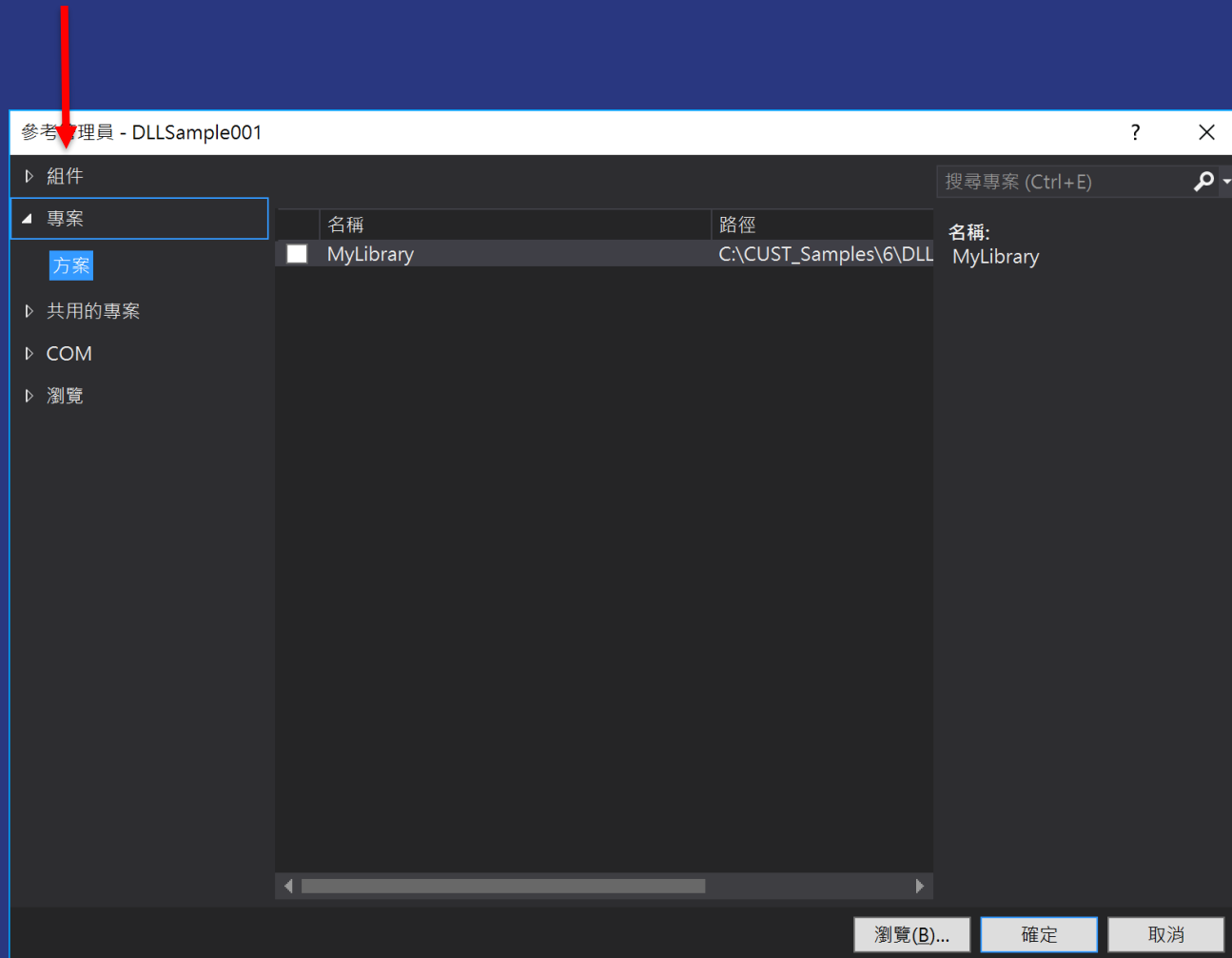
    public int GetArea()
    { return Width * Height; }
}
```



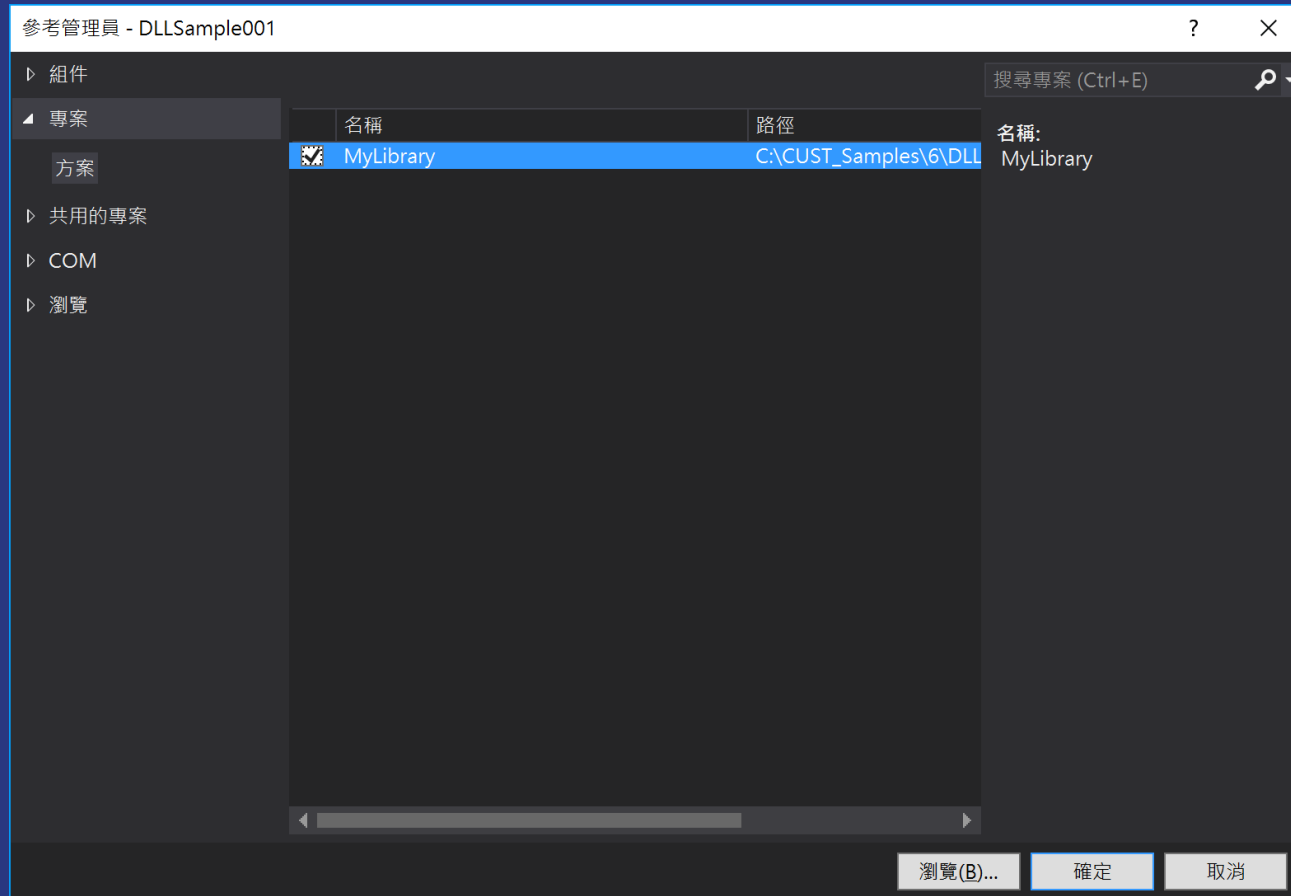
回到 DLLSample001 專案，開啟【參考】，
按滑鼠右鍵選擇【加入參考】



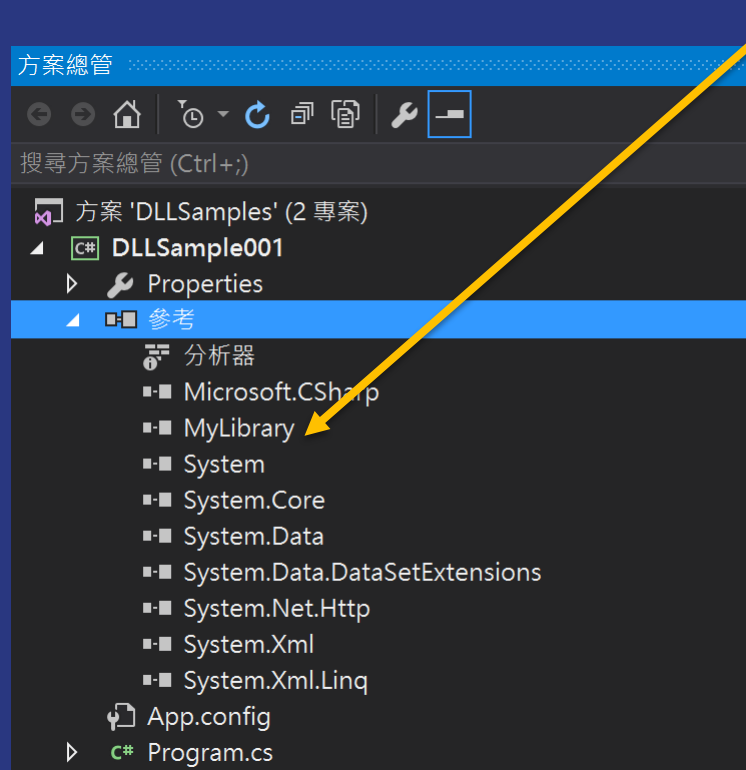
左邊這一塊是參考的來源



因為我們要參考同一個方案中的另一個專案
所以左邊選【專案】→【方案】
並且在中間窗格勾選該專案 【MyLibrary】



完成以後，你會發現在 DLLSample001專案中的
【參考】裡多了一項【MyLibrary】

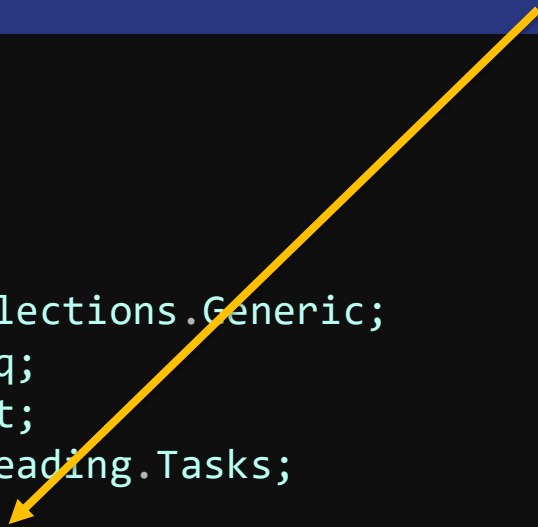


回到 DLLSample001 專案的 Program class

```
namespace DLLSample001
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

記得先引入命名空間，否則你就得用全名了

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using MyLibrary;
```



修改 Main method

```
class Program
{
    static void Main(string[] args)
    {
        MyRectangle rect = new MyRectangle();
        rect.Width = 10;
        rect.Height = 10;
        Console.WriteLine("面積為 :" + rect.GetArea());
        Console.ReadLine();
    }
}
```

執行

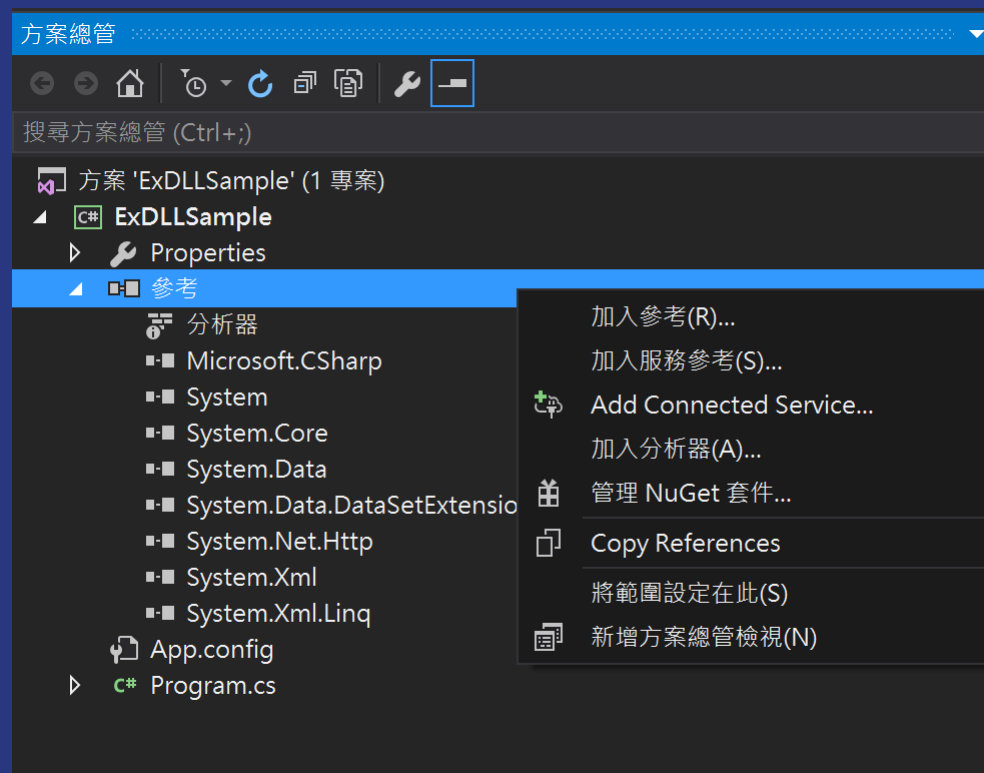
LAB

用檔案的方式加入參考

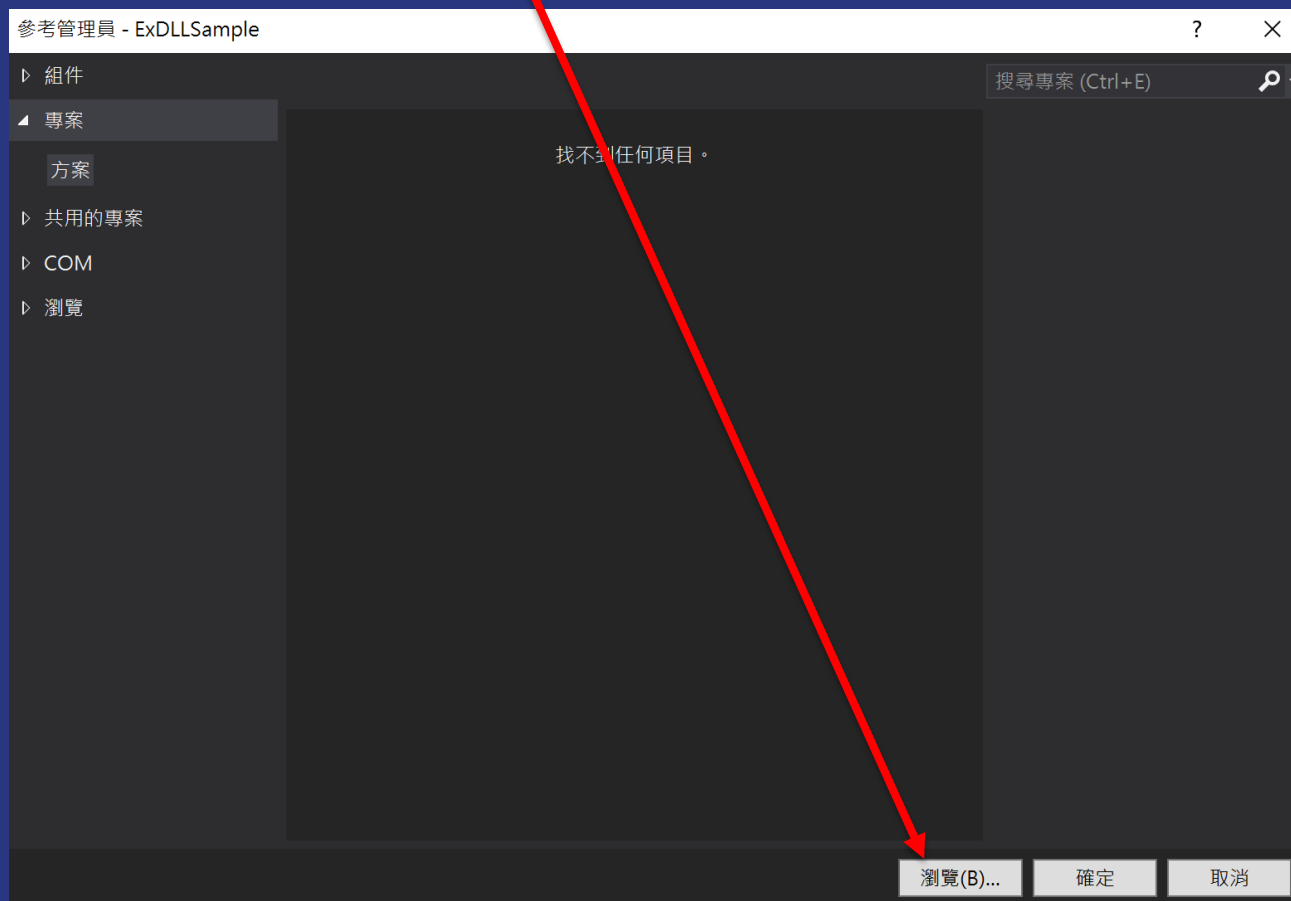
新增一個方案及專案

- 方案名稱：ExDLLSample
- 專案名稱：ExDLLSample
- 範本：Console Application

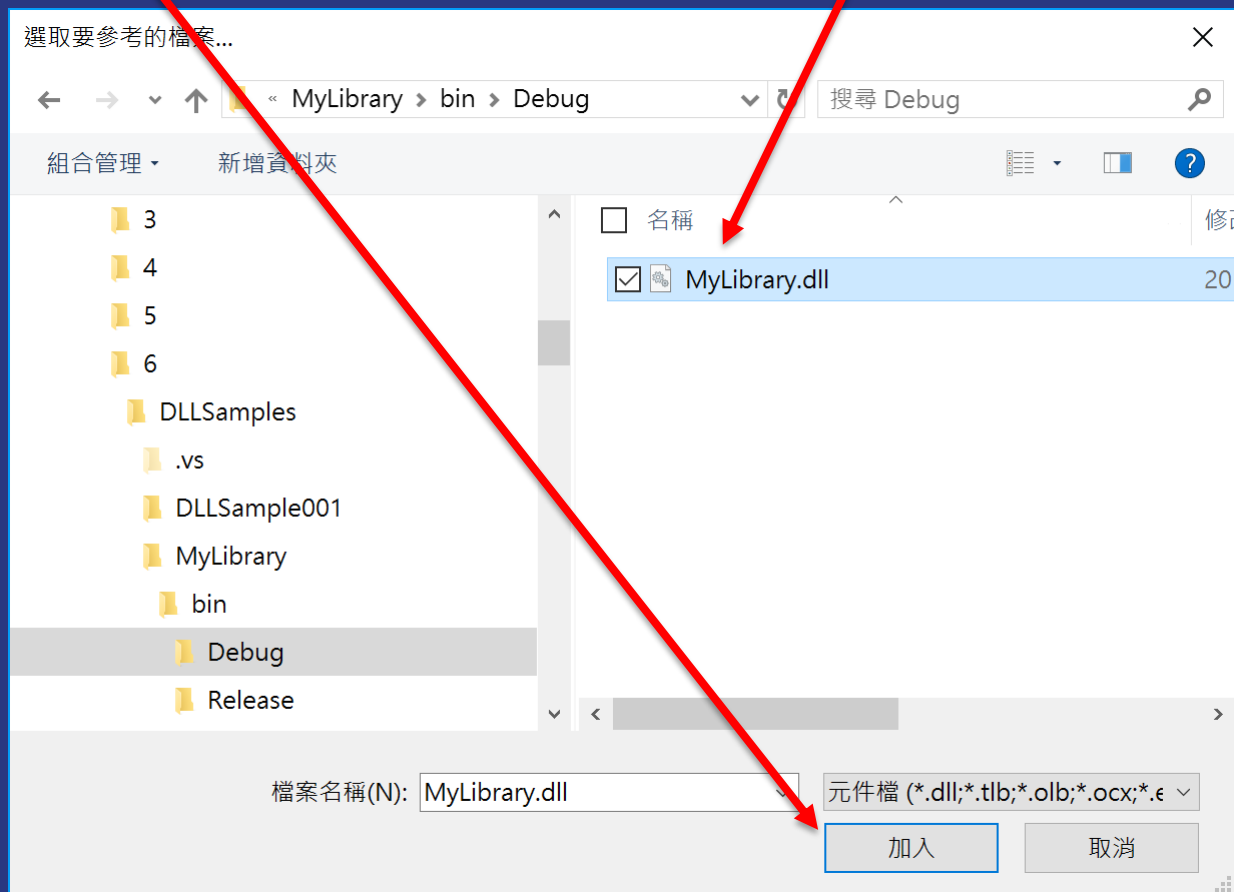
開啟【參考】→【加入參考】



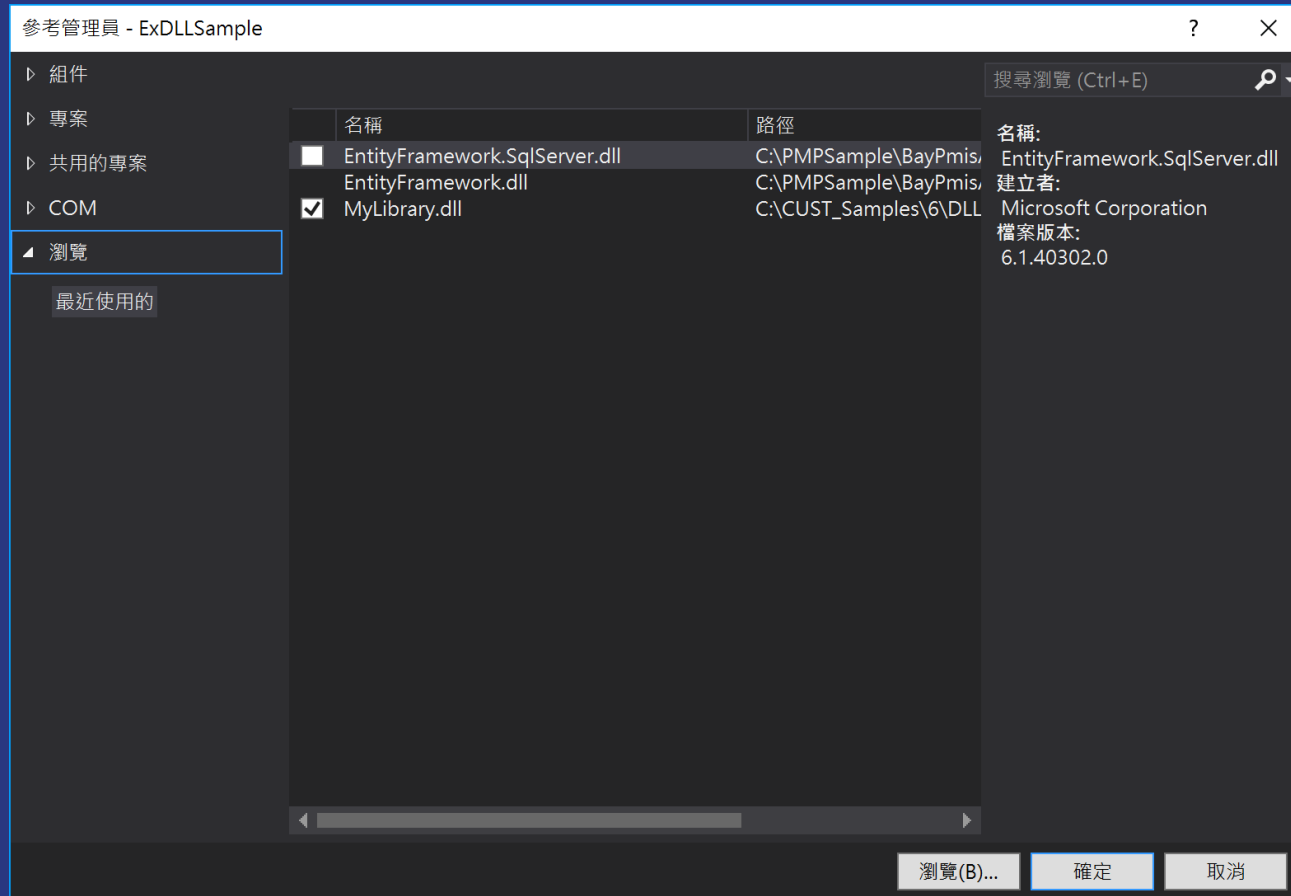
這次我們要選擇【瀏覽】



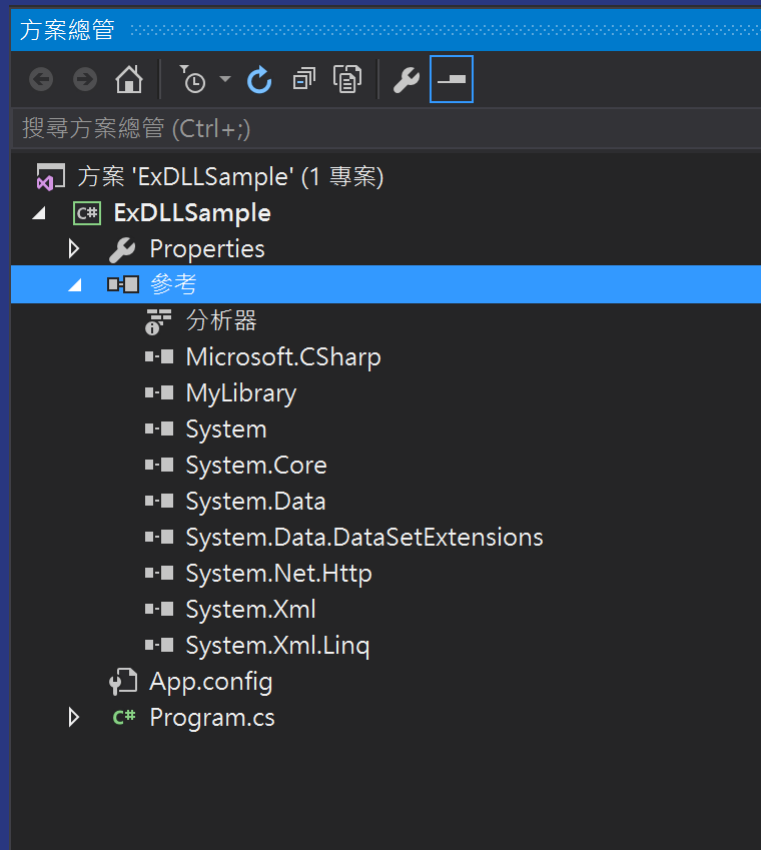
選擇剛剛 MyLibrary 專案所輸出的 MyLibrary.DLL
按下【加入】



你會看到類似這樣的畫面，
其中 **MyLibrary.dll** 是被勾選的，按下【確定】



接下來，就和前一個 lab 是一樣的了



認識組件 (Assembly)

- 組件是部署和管理 managed 程式碼程式的身分識別的單位。
- 雖然組件可以跨越一或多個檔案，通常對應一對一的 EXE / DLL 組件。
- 一個組件可能具有一個或以上的命名空間，但也可可能同一個命名空間會散布在不同的組件

一般慣例

- 通常我們不做一檔多組件，慣例上一個專案的輸出檔（一個 `exe` 或一個 `dll`）就做為一個組件。
- 我們也很少把一個命名空間分散在不同的組件，通常都是一個組件包含一個或以上的完整的命名空間
- 因此，當日後聽到課堂上講同一個組件的時候，就代表是位於同一個專案中。

.Net Framework 中的型別系統

型別概要

- Primitive Type
- Reference Type (參考型別)
 - 介面 Interface
 - 類別 Class
 - 委派 Delegate
- Value Type (實值型別)
 - 結構 Structure
 - 列舉 Enum

甚麼是物件 (object) ?

- 類別或結構定義就像是藍圖，會指定能夠使用的型別。物件基本上是記憶體區塊，是根據藍圖加以配置和設定。程式可能會建立很多相同類別的物件。物件也稱為執行個體 (**instance**)，可以儲存在已命名的變數、陣列或集合中。

變數和物件的關係

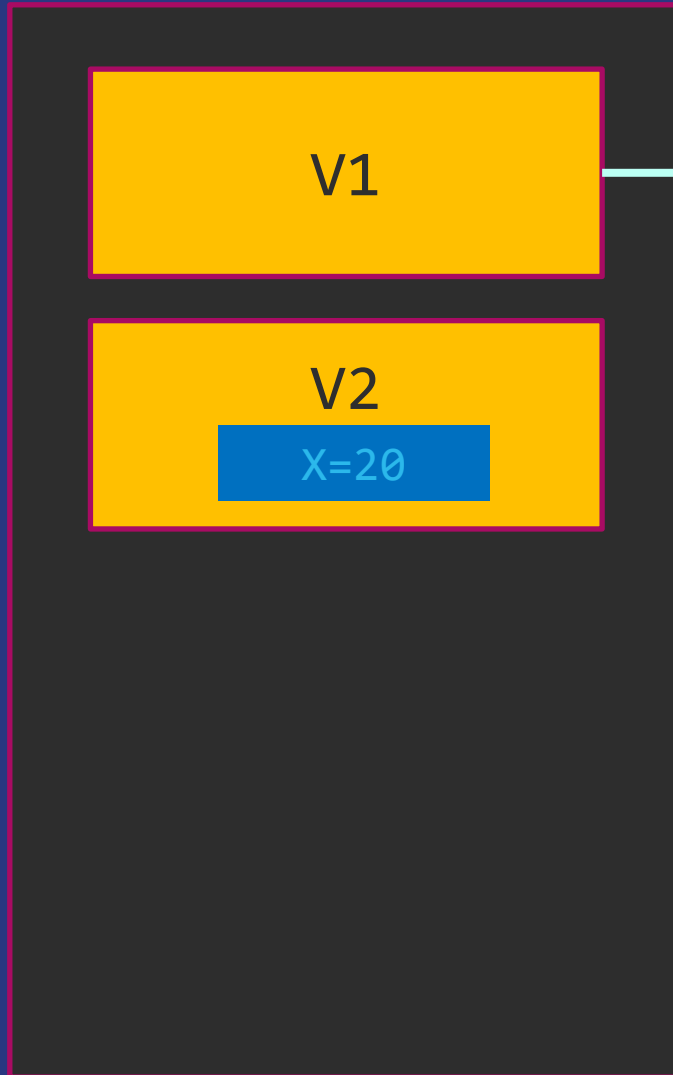
- 在實值型別的狀況
 - 物件包在變數裡面
 - 你也可以想像成變數就是物件
- 在參考型別的狀況
 - 變數的內容是一個指向物件的參考（也就是變數的內容是一個位址）
 - 變數不是物件

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

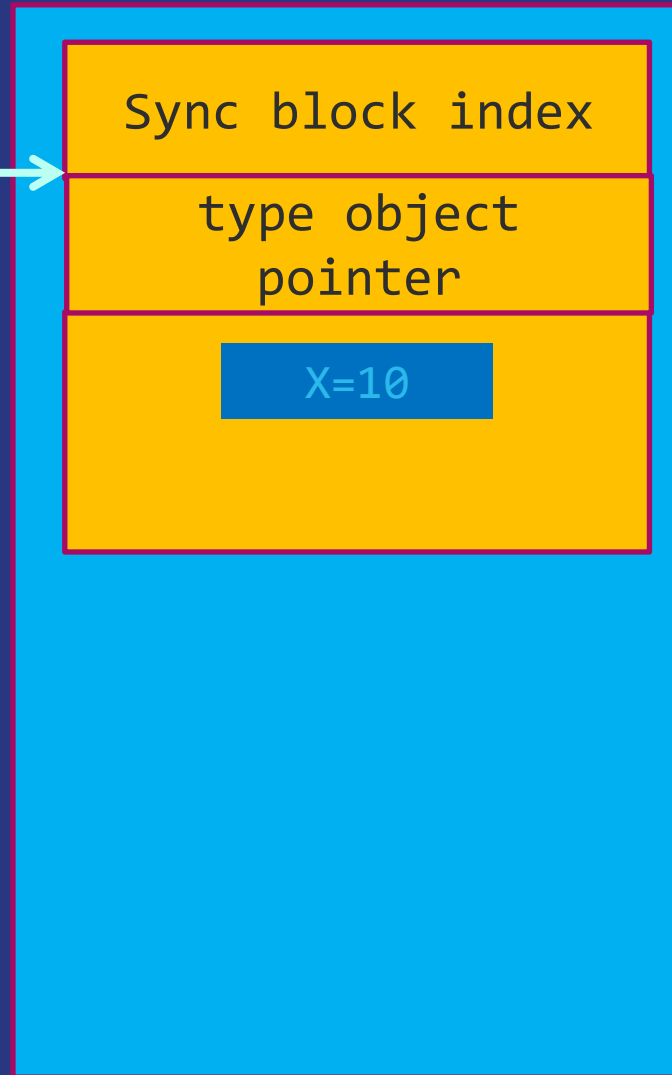
```
public struct MyValStruct  
{ public int x; }
```

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


Thread Stack



Managed Heap

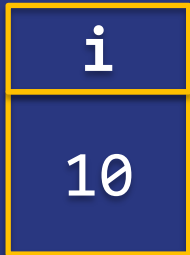


所以
指派運算倒底在幹嘛

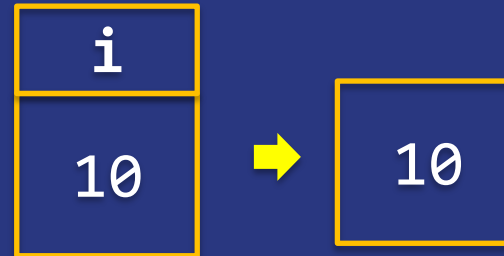
從以下的程式碼看實值型別的變化

```
int i = 10;  
int j = i;
```

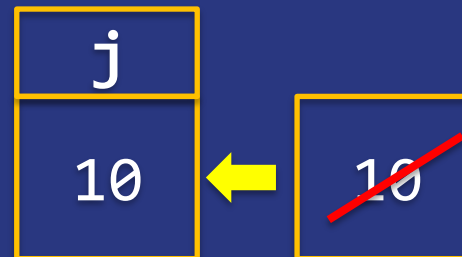
(1) 記憶體裡面有個變數，
名稱是 **i**，值為 **10**



(2) 電腦將**i**變數的值複製一份，
放在記憶體的另一個區塊



(3) 電腦將剛剛的**10**指派給
j變數，並將那個 **10** 移除



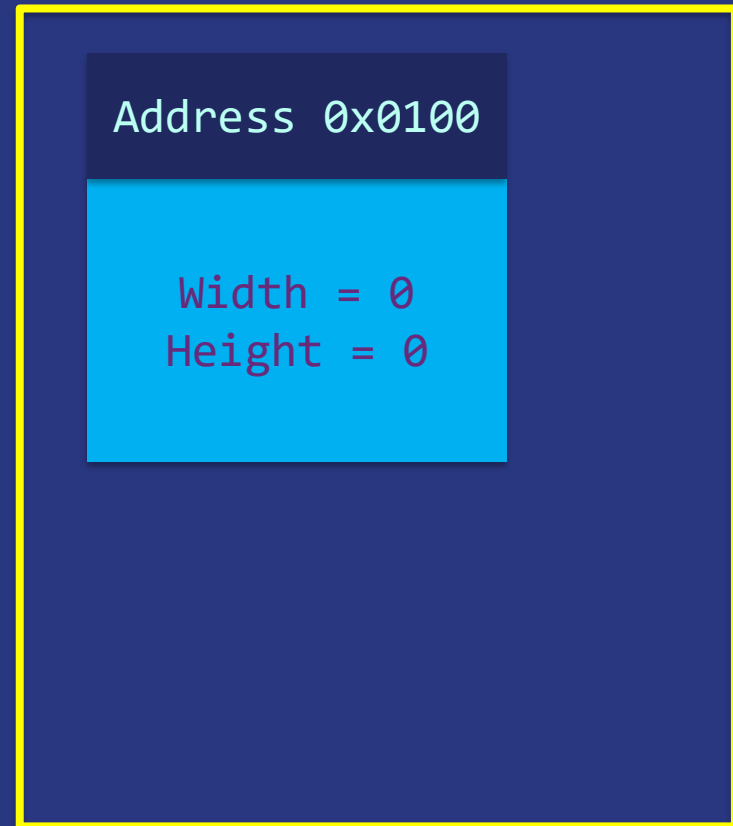
從以下的程式碼看參考型別的變化

```
MyRectangle r1 = new MyRectangle();  
MyRectangle r2 = r1;
```

```
public class MyRectangle  
{  
    public int Witdh { get; set; }  
    public int Height { get; set; }  
  
    public int GetArea()  
    { return Witdh * Height; }  
}
```

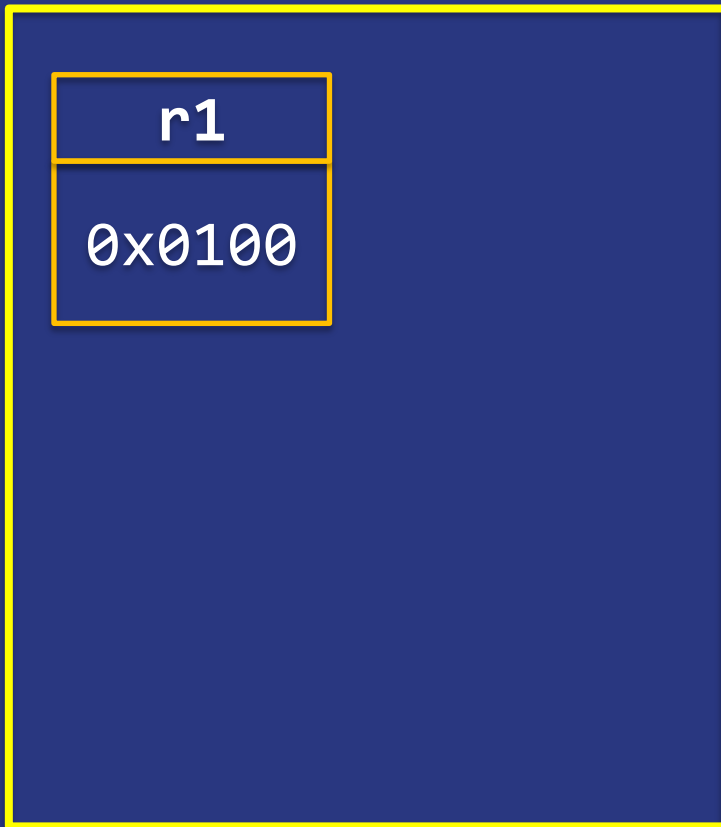
- (1) 在Heap中依據 MyRectangle class 產生一個物件 (也就是 `new MyRectangle();` 執行的作業)
- (2) 假設這個物件在 Heap 中的位址是 0x0100 (實際上位址應該是 4或8 Bytes)

Heap

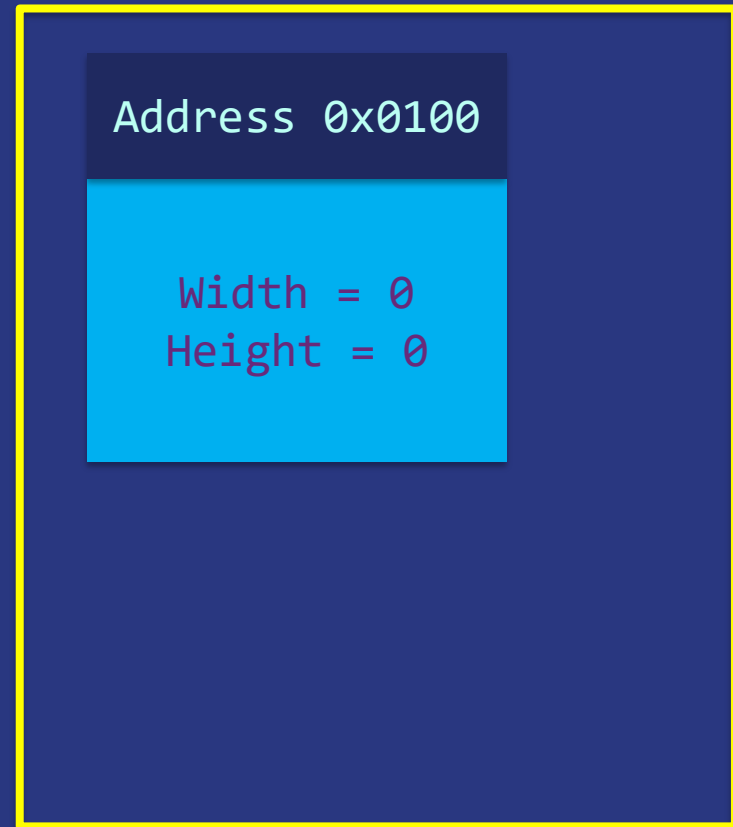


(3) 接著指派運算 (也就是 =) 會將 Heap 中物件的位址複製一份傳給 Stack 裡的 r1 變數當成 r1 變數的內容

Stack

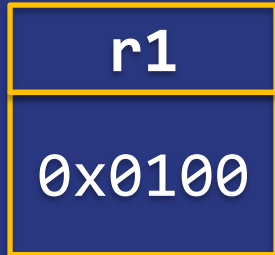


Heap

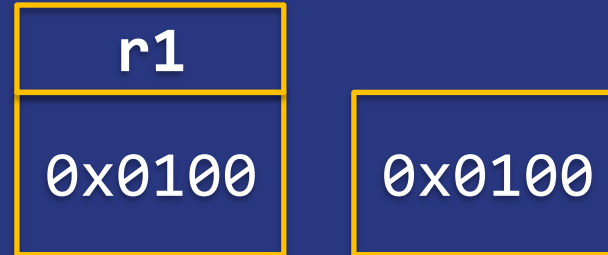


(4) 接著就是Stack 裡的複製作業

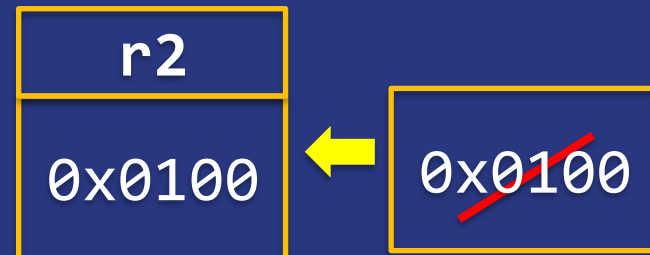
(1) 記憶體裡面有個變數，
名稱是 r1，值為 0x0100



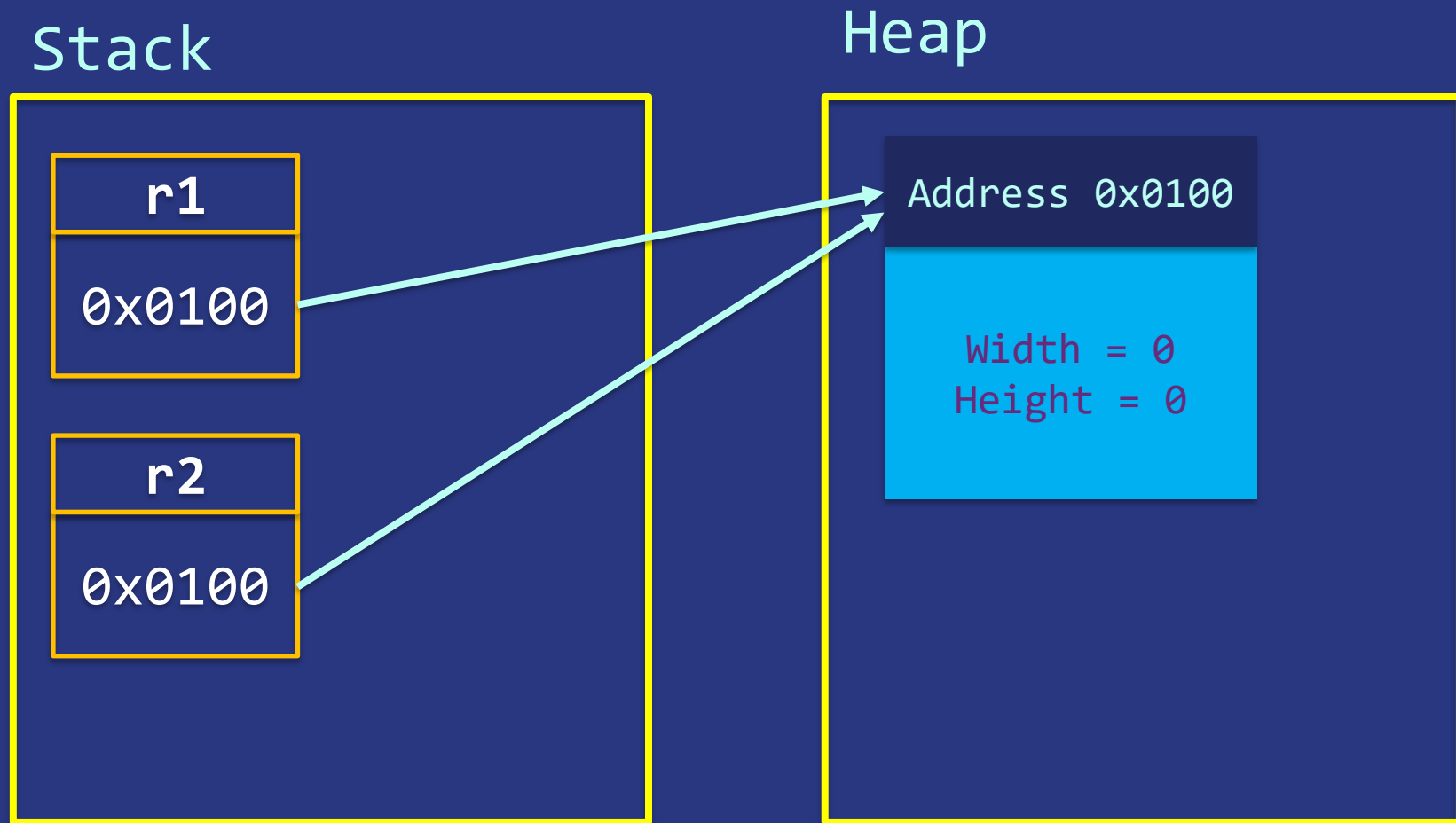
(2) 電腦將r1變數的值複製一份，
放在記憶體的另一個區塊



(3) 電腦將剛剛的0x0100指派給
r2變數，並將那個 0x0100 移除



(5) 結果就是這樣



關於比較運算子 == 與 !=

- 在沒有多載 (overloading) 的情形下
- == 和 != 基本上比較的就是『變數的內容』
- **String** 型別就是一個有多載 == 和 != 的例子，所以 **String** 是一個很特別的类型別
- 使用者定義結構型別預設不支援 == 和 != 運算子

萬物之母 Object Class

Object Class 重點歸納

- 隱含繼承
- 重要的公開成員方法
 - `Object.GetType` 方法
 - `Object.Equals` 方法 (virtual)
 - `Object.ToString` 方法 (virtual)
- 重要的公開靜態方法
 - `Object.Equals` 方法
 - `Object.ReferenceEquals` 方法

隱含繼承

當你這樣宣告一個類別時

```
public class MyRectangle
```

其實就是代表這樣的意思

```
public class MyRectangle : object
```

Object.Equals (Object, Object)

- Object 中有兩個 Equals 方法
- 這個有兩個參數的是靜態方法 (static)
- 另一個只有一個參數的則是執行個體方法

```
public static bool Equals(object objA, object objB)
{
    return ((objA == objB) || (((objA != null) && (objB != null)) && objA.Equals(objB)));
}
```

LAB

使用 `object.Equals (object,object)`

新增一個方案及專案

- 方案名稱：EqualitySamples
- 專案名稱：EqualitySample001
- 範本：Console Application

加入 MyRectangle Class，並建立其內容

```
class MyRectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
}
```

在 Main method 加入以下內容

```
static void Main(string[] args)
{
    int i = 10;
    int j = 10;
    Console.WriteLine($"object.Equals(i,j) is { object.Equals(i, j) }");

    MyRectangle r1 = new MyRectangle { Width = 5, Height = 5 };
    MyRectangle r2 = new MyRectangle { Width = 5, Height = 5 };
    MyRectangle r3 = r2;
    Console.WriteLine($"object.Equals(r1,r2) is { object.Equals(r1, r2) }");
    Console.WriteLine($"object.Equals(r2,r3) is { object.Equals(r2, r3) }");

    Console.ReadLine();
}
```

執行

Object.ReferenceEquals (Object, Object)

- `ReferenceEquals` 專門用來比較兩個物件的參考 (位址) 是否相等
- 只用於比較參考型別的物件
- 實值型別的比較永遠都是 `false`

LAB

使用 `object.ReferenceEquals (Object, Object)`

在 EqualitySamples 加入新專案

- 方案名稱： EqualitySamples
- 專案名稱： EqualitySample002
- 範本： Console Application

加入 MyRectangle Class，並建立其內容

```
class MyRectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
}
```

在 Main method 加入以下內容

```
static void Main(string[] args)
{
    int i = 10;
    int j = 10;
    Console.WriteLine($"ReferenceEquals(i,i) is {object.ReferenceEquals(i, i)}");
    Console.WriteLine($"ReferenceEquals(i,j) is {object.ReferenceEquals(i, j)}");

    MyRectangle r1 = new MyRectangle { Width = 5, Height = 5 };
    MyRectangle r2 = new MyRectangle { Width = 5, Height = 5 };
    MyRectangle r3 = r2;
    Console.WriteLine($"ReferenceEquals(r1,r2) is {object.ReferenceEquals(r1, r2)}");
    Console.WriteLine($"ReferenceEquals(r2,r3) is {object.ReferenceEquals(r2, r3)}");

    Console.ReadLine();
}
```


執行

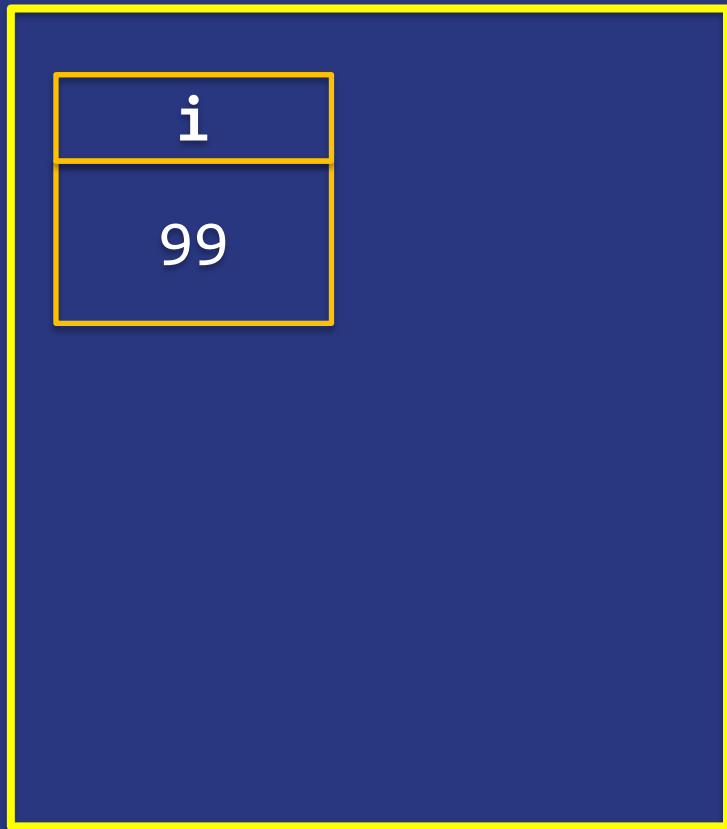
Boxing 與 Unboxing

- 實值型別與參考型別之間的轉換
- 造成效能耗損

從以下的程式碼看Boxing 如何發生

```
int i = 99;  
object o1 = i;  
object o2 = i;
```

(1) 首先，在 Stack 中產生 `int i = 99;`

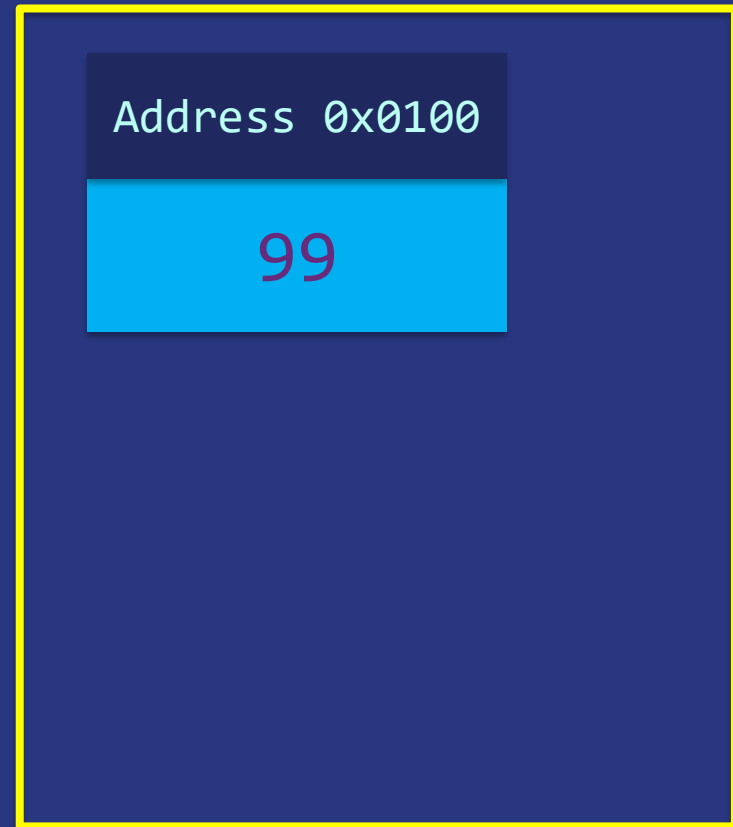
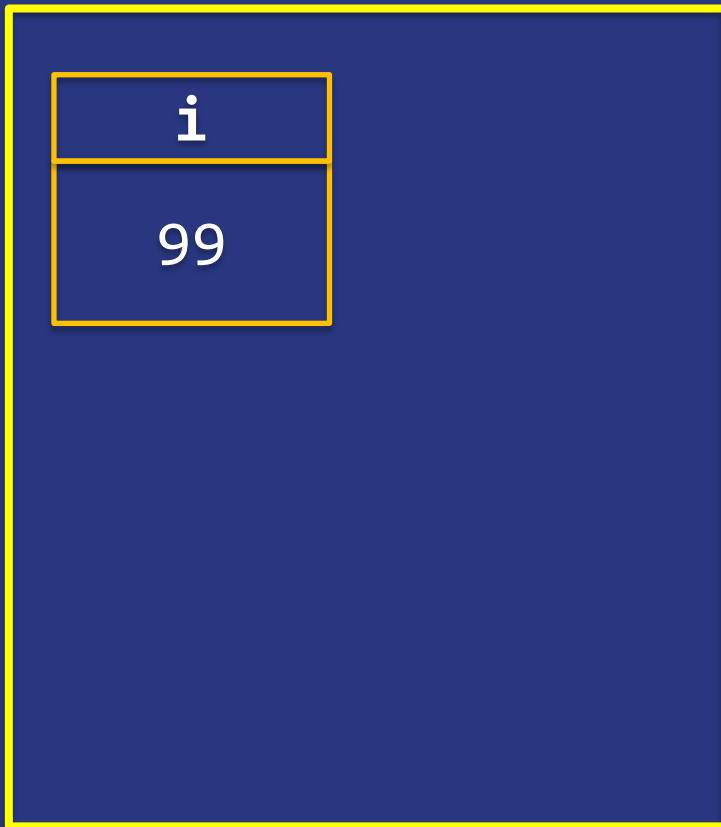


Heap

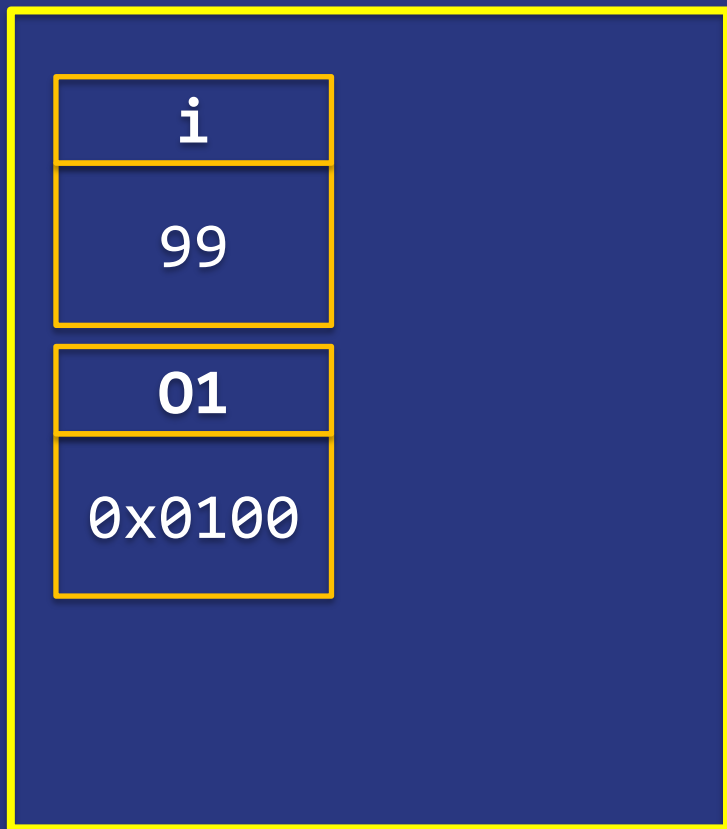


(2) 接著遇到 `object o1 = i`; 由於 `o1` 是參考型別的變數，因此它的變數內容會是指向 Heap 的某個物件的位址。
所以 Heap 會先產生一個物件 (這個玩意被稱為 `Boxed-Int32`)，然後將 `i` 變數的內容放在這個 `Boxed-Int32` 的物件中

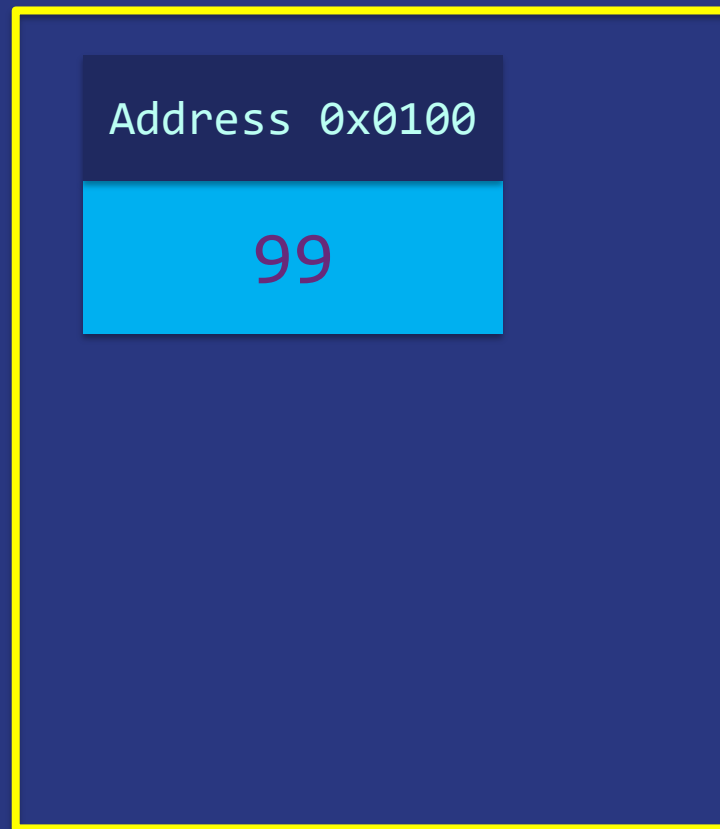
Heap



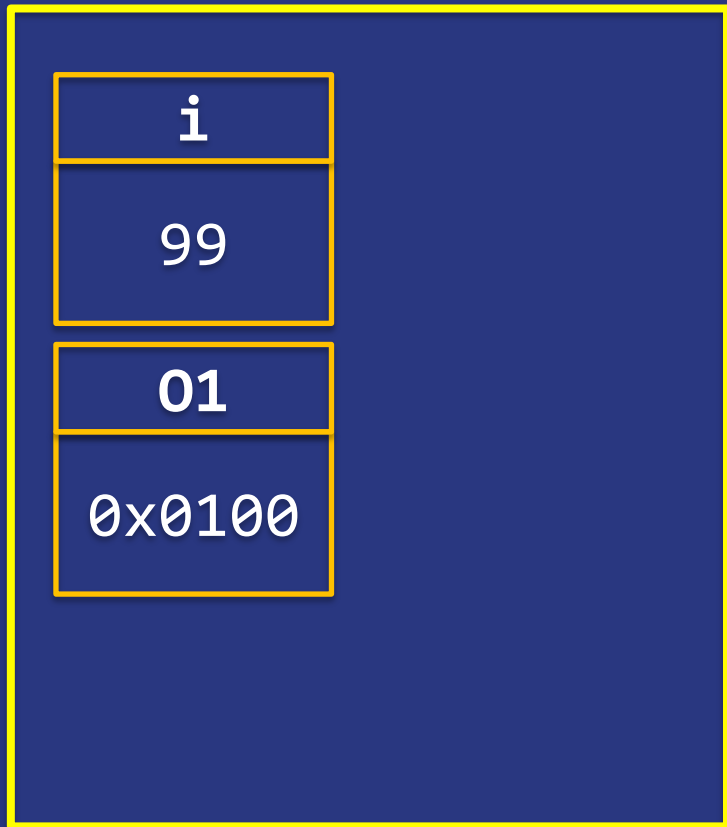
(3) 然後將變數 o1 指向這個 Boxed-Int32 的物件



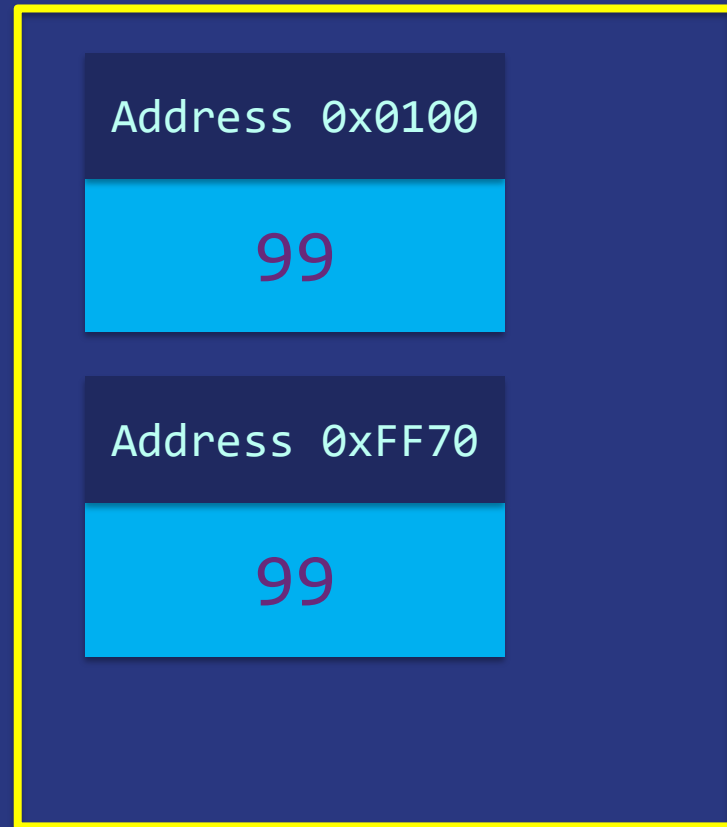
Heap



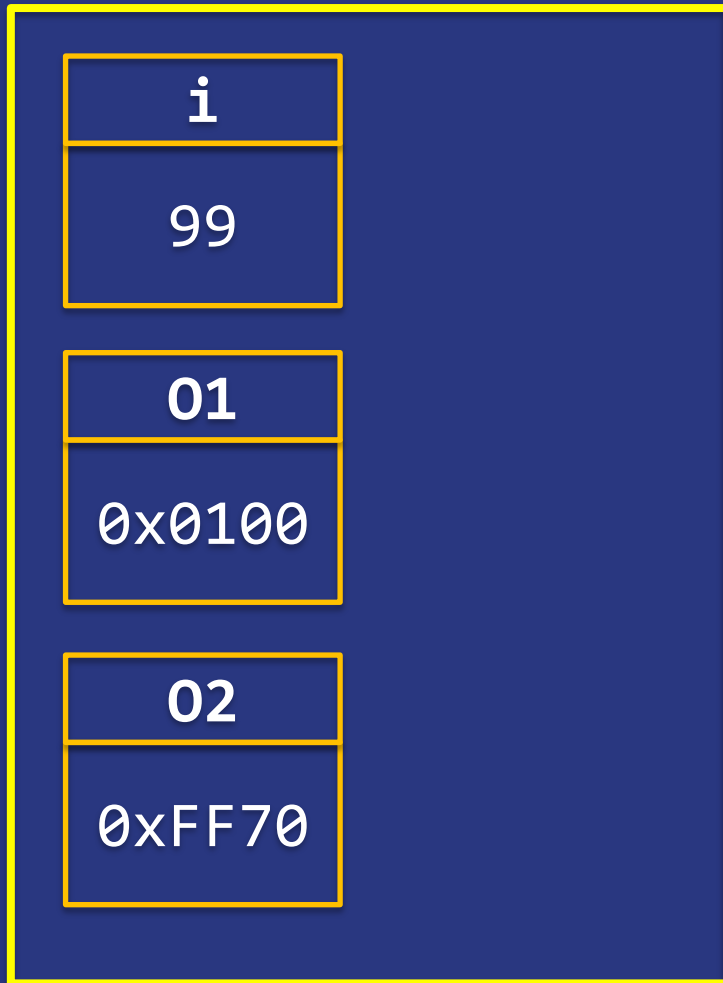
(4) 接著遇到 `object o2 = i;` 和前述(2) 的狀況相同，他也會在 Heap 產生一個新的 `Boxed-Int32` 物件



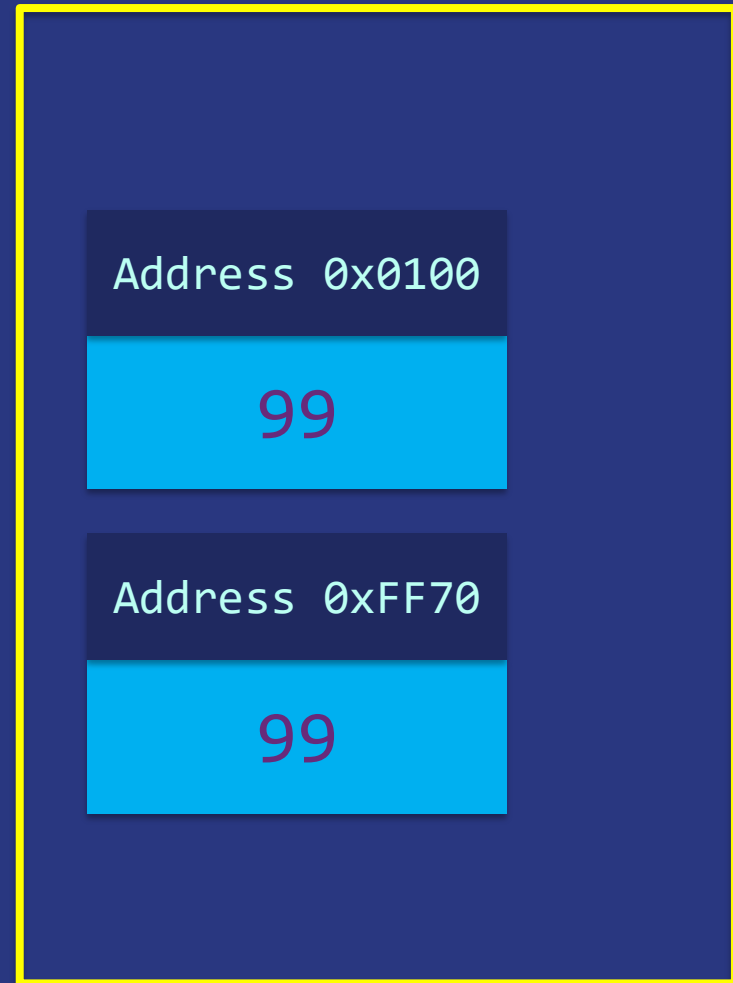
Heap



(5) 然後將變數 o2 指向這個 Boxing-Int32 的物件



Heap



討論

- 根據剛剛 Boxing 的流程，討論一下為什麼 `object.ReferenceEquals` 遇到實值型別就一定是 `false`？



UnBoxing 則是倒過來

```
object o1 = 76;  
int j = (int)o1;
```

Object.GetType()

- 用來取得執行個體的类型
- 回傳值的类型為 `Type Class`

Object.Equals (Object)

- 這個 Equals 是執行個體方法，表示自己與傳入的物件比較
- 同時它也是個虛擬方法 (virtual method)

LAB

使用 `object.Equals (object)`

在 EqualitySamples 加入新專案

- 方案名稱： EqualitySamples
- 專案名稱： EqualitySample003
- 範本： Console Application

加入 MyRectangle Class，並建立其內容 (自動產生 GetHashCode)

```
class MyRectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

    public bool Equals(MyRectangle other)
    {
        return (this.Height == other.Height) &&
            (this.Width == other.Width);
    }

    public override bool Equals(object obj)
    {
        return this.Equals((MyRectangle)obj);
    }

    public override int GetHashCode()
    {
        var hashCode = 859600377;
        hashCode = hashCode * -1521134295 + Width.GetHashCode();
        hashCode = hashCode * -1521134295 + Height.GetHashCode();
        return hashCode;
    }
}
```

在 Main method 加入以下內容

```
class Program
{
    static void Main(string[] args)
    {
        int i = 10;
        int j = 10;
        Console.WriteLine("i.Equals(j) is " + i.Equals(j));
        MyRectangle r1 = new MyRectangle { Width = 5, Height = 5 };
        MyRectangle r2 = new MyRectangle { Width = 5, Height = 5 };
        MyRectangle r3 = r2;
        Console.WriteLine("r1.Equals(r2) is " + r1.Equals(r2));
        Console.WriteLine("r2.Equals(r3) is " + r2.Equals(r3));

        Console.ReadLine();
    }
}
```


至於 EqualitySample004

- 各位可以在下課後研究一下這個 Sample
- 看看和 Sample003 為何會有不同的結果

Object.ToString ()

- 傳回代表目前物件的字串。
- 虛擬方法 (virtual method)
- 如果沒有覆寫 (override) 就會傳回這個執行個體型別的字串
- 有些型別 (如：Int32 等) 會覆寫或多載 ToString() 方法。

以 Int32 為例說明 ToString 的多載與覆寫

- System.Int32 具有以下的 ToString 方法
- 覆寫 object.ToString()
 - ToString()
- 多載
 - ToString(IFormatProvider)
 - ToString(String)
 - ToString(String, IFormatProvider)
- 字串格式化請參閱
 - .NET Framework 中的格式化類型
 - [https://msdn.microsoft.com/zh-tw/library/26etazsy\(v=vs.110\).aspx](https://msdn.microsoft.com/zh-tw/library/26etazsy(v=vs.110).aspx)

LAB

Int32.ToString

開啟專案

- 請開啟範例中的 `IntToStringSample`
- 執行並觀察其結果
- 閱讀我所寫的註解
- 對照 `Microsoft Docs` 文件的說明
- 請互相討論
- 務必熟悉字串格式化的用法，你會常常用到的

null 代表甚麼意義

- null 代表甚麼都沒有
- 所有的變數在沒有指派任何內容給它以前，無論它的記憶體長度為何，在記憶體的每個 bit 都是 0
- 因此，對於參考型別的變數來說，所有 bit 都是 0 就代表它沒有指向 Heap 中的任何東西，我們稱此為 null

LAB

初步學習繼承

新增一個方案及專案

- 方案名稱：FirstInheritSamples
- 專案名稱：FirstInheritSample001
- 範本：Console Application

加入 Class1 Class，保持其內容空白

```
class Class1  
{  
}
```

在 Program class 中的 Main method 加入程式碼
你會發現即使 Class1 類別甚麼都沒寫
它還是擁有 Equals 方法
此時我們稱這個 Equals 方法繼承自 Object 類別

```
static void Main(string[] args)
{
    Class1 o1 = new Class1();
    Class1 o2 = new Class1();
    bool result = o1.Equals(o2);
    Console.WriteLine("o1.Equals (o2) is " + result);

    Console.ReadLine();
}
```

執行

討論

- 了解 object 類別的基本用法以及簡單的繼承了嗎？



奇妙的 String Class

關於 string

- `String` 是參考型別，但是因為多載了 `==` 和 `!=` 運算子，常被誤以為是實值型別。
- `String pool`
- 在 C# 中 `String` 具有不可變性 (`immutable`)，除非你使用 `unsafe` 或著其他暗黑的方式操作字串。

String Class 的重要成員

- 屬性
 - Length : 取得目前 `string` 中的字元數量
- 靜態方法
 - Format 方法 : 以指定之物件的字串表示，取代指定之字串中的一或多個格式項目，此方法有許多個多載
- 靜態欄位
 - Empty : 唯讀欄位，表示空字串

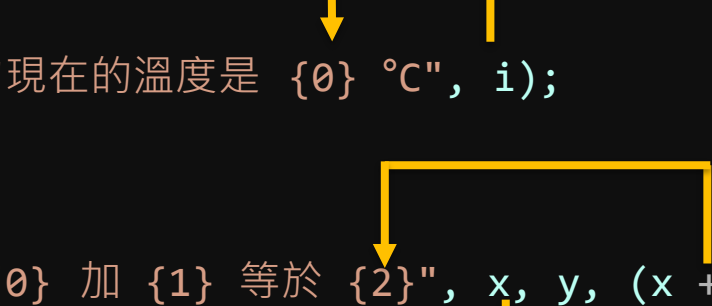
String Class 的重要成員

- 執行個體方法
 - PadLeft：在字串左方填補字元
 - PadRight：在字串右方填補字元
 - Replace：取代某個字串
 - Split：分割字串
 - Trim：移除開頭和結尾的所有空白字元
 - TrimStart：移除開頭的所有空白字元
 - TrimEnd：移除結尾的所有空白字元

String.Format

- 不具文化特性的使用方式

```
int i = 32;  
string r1 = string.Format("現在的溫度是 {0} °C", i);  
  
int x = 1;  
int y = 8;  
string r2 = string.Format("{0} 加 {1} 等於 {2}", x, y, (x + y));
```



超過三個怎麼辦？

```
object[] array = new object[] { 11, 12, 13, 14, 15 };  
string r3 = string.Format("{0},{1},{2},{3},{4}", array);
```

請注意陣列的型別是 `object[]`

C# 6.0 後可用語法糖

```
string r1 = string.Format("現在的溫度是 {0} °C", i);  
string r1x = $"現在的溫度是 {i} °C";  
  
int x = 1;  
int y = 8;  
string r2 = string.Format("{0} 加 {1} 等於 {2}", x, y, (x + y));  
string r2x = $"{x} 加 {y} 等於 {x + y}";
```

String pool (字串池)

- 被宣告為固定字串的字串，會被放入字串池。如果有另一個變數指派到同樣內容的字串，這兩個變數會指向字串池中的同一個字串。
- 字串池中的字串生命週期會隨著應用程式定義域 (Application Domain)。

```
static void Main(string[] args)
{
    string s1 = "ABC";
    string s2 = "ABC";
    // "ABC" 為固定字串，因此 s1, s2 會指向字串池中的同一個 "ABC"
    Console.WriteLine(object.ReferenceEquals(s1, s2));
    int x = 1;
    string s3 = x.ToString();
    string s4 = x.ToString();
    // s3, s4 所指向的字串為程式產生，所以這兩個 "1"並非相同的物件
    Console.WriteLine (object.ReferenceEquals(s3, s4));
    Console.ReadLine();
}
```

註：這檔事在 .NET Core 3.0 後有點改變

字串的不可變性

- 除非你使用 `unsafe` 的方式操作字串，否則一般情形下字串是不可變的。
- 因為這個不可變性，因此大量的字串連接會造成效能損耗。
- 若要在大量迴圈內處理字串的連接或變更，請使用 `StringBuilder` 類別。

以下程式碼將會在 Heap 中產生三個物件

```
string s1 = "ABC";  
string s2 = "DEF";  
string s3 = s1 + s2;
```

實值型別

實值型別 (Value Type)

- 實值型別的繼承鏈中，必然包含有 `ValueType` `Class`
- 兩種實值型別
 - `Enum`
 - `Structure`

實值型別的特性

- 物件本身儲存在變數裡
- 在運算堆疊中，直接拿變數中的值運算

結構 Structure

結構

- 通常可用來封裝一小組相關變數
- 結構可以實作介面，但無法繼承自其他結構。 因此，結構成員無法宣告為 `protected`
- 結構還包含建構函式、常數、欄位、方法、屬性、索引子、運算子、事件和巢狀類型，不過如果需要數個這類成員，應該考慮以類別來取代結構

結構概念

- 結構宣告內不能初始化欄位，除非將其宣告為 `const` 或 `static`
- 結構不可宣告預設建構函式（沒有參數的建構函式）或解構函式
- 結構可以宣告含有參數的建構函式
- 結構是在指派時複製的。當指派結構給新變數時，就會複製所有資料，而新變數所做的任何修改都不會變更原始複本的資料

- 與類別不同的是，結構並不需要使用 `new` 運算子就能具現化
- 結構無法從另一個結構或類別繼承而來，且它不能成為類別的基底。
- 結構可實作介面
- 它是型別，所以可以宣告在命名空間區段。

結構的宣告

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

關於結構的注意事項

- 結構的欄位和屬性成員不要使用太複雜的型別。
- 基本上結構有效能上的優勢，但這個優勢會建立在對於結構的完整了解，因此建議你們現階段多用自訂類別，少用自訂結構。

Nullable<T> 可為 null 的結構

關於 Nullable<T>

- 泛型（也就是 T 的部分）只能塞實值型別。
- Nullable<T> 本身是個結構，也就是實值型別。
- 由於實值型別本身不會有 null，但在資料庫的設計中，不管甚麼型別都可能會有 null 值，而 Nullable<T> 就是為了解決這個問題所使用的。
- Nullable<T> 雖稱為可為 null 的實值型別，但是它只是偽裝自己有 null 的表示方式。

Nullable<T> 的重要成員

- 屬性

- Value : 表示所代表的值
- HasValue : 表示是否有值，若為 false，則代表 null

- 方法

- GetValueOrDefault : 當 HasValue 為 true，則取得目前的T型別的值 (也就是 Value 屬性)；當 HasValue 為 false，則取得T型別的預設值。

Nullable<T> 的宣告

```
Nullable<int> i = 10;  
Nullable<int> j = null;  
double? d = 9.8;
```

LAB

使用 Nullable<T>

新增一個方案及專案

- 方案名稱：NullableSamples
- 專案名稱：NullableSample001
- 範本：Console Application

修改 Main method

```
static void Main(string[] args)
{
    int? i = 10;

    if (i.HasValue)
    { Console.WriteLine(i); }

    i = null;
    if (!i.HasValue)
    { Console.WriteLine(i.GetValueOrDefault()); }

    if (i == null)
    { Console.WriteLine("i is null"); }

    Console.ReadLine();
}
```

執行

列舉 Enum

Enum 概觀

- 使用 `enum` 宣告
- 它是型別，所以可以宣告在命名空間區段。
- 由一組稱為列舉值清單的具名常數所構成的獨特型別
- 預設基礎型別是 `int`
- 核准型別為 `byte`、`sbyte`、`short`、`ushort`、`int`、`uint`、`long` 或 `ulong`
- 如同任何常數，在編譯時期，`enum` 之個別值的所有參考都會轉換成數值常值

列舉的基本宣告方式

```
/// <summary>
/// 自動從 0 開始
/// </summary>
public enum MyWeekDays
{
    Sun, Mon, Tue, Wed, Thu, Fri, Sat
}
```

```
/// <summary>
/// 設定從 1 開始
/// </summary>
public enum BrowserTypes
{
    IE = 1, Edge, FireFox, Chrom
}
```

```
/// <summary>
/// 全部值都手動設定
/// </summary>
public enum SwitchTypes
{
    On = 0, Off = 1
}
```

LAB

使用 Enum

新增一個方案及專案

- 方案名稱：EnumSamples
- 專案名稱：EnumSample001
- 範本：Console Application

先加入一個類別

```
namespace EnumSample001
{
    class Class1
    {
    }
}
```

把 class Class1 區段移除

```
namespace EnumSample001  
{  
  
}
```

在 namespace 中加入以下程式碼

```
/// <summary>  
/// 自動從 0 開始  
/// </summary>  
public enum MyWeekDays  
{  
    Sun, Mon, Tue, Wed, Thu, Fri, Sat  
}
```

```
/// <summary>  
/// 設定從 1 開始  
/// </summary>  
public enum BrowserTypes  
{  
    IE = 1, Edge, FireFox, Chrom  
}
```

```
/// <summary>  
/// 全部值都手動設定  
/// </summary>  
public enum SwitchTypes  
{  
    On = 0, Off = 1  
}
```


修改 Program Class 的 Main method

```
static void Main(string[] args)
{
    MyWeekDays day = MyWeekDays.Sun;
    Console.WriteLine($"Today is {day}");
    if (day == MyWeekDays.Mon)
    {
        Console.WriteLine("Today is Monday");
    }
    else
    {
        Console.WriteLine("Today is not Monday");
    }

    //轉換回 int
    int i = (int)day;
    Console.WriteLine($"The value of {day} is {i}");

    Console.ReadLine();
}
```

執行

討論

- Enum 是一個非常重要的型別，請互相討論是否了解基本的用法了。



列舉的另一個課題

- 旗標列舉型別
- 表示各列舉值是可以疊加狀態的
- 參閱：MSDN 文件列舉類型 (C# 程式設計手冊)
- <https://msdn.microsoft.com/zh-tw/library/sbdt4032.aspx>

參考型別

參考型別

- 類別 (Class)
- 介面 (Interface)
- 委派 (Delegate)

參考型別的特性

- 參考型別的變數儲存的是物件的參考
- 參考型別的物件是存在於 **Heap**

再度回到類別

先來談對於型別的存取修飾詞

- 在命名空間中宣告的型別別可為 `public` 或 `internal`（預設，你沒有寫的時候叫預設）

類別內的成員

- 建構式 – Constructor
- 常數 – Constant
- 欄位 – Field
- 屬性 – Property
- 方法 – Method
- 事件 – Event
- 索引子 – Indexer
- 類別內也可以定義其他的型別，我們稱為巢狀型別

類別內部成員的存取修飾詞

- 類別內的成員可以宣告為
 - `private` (預設)
 - `internal`
 - `protected`
 - `protected internal`
 - `public`

類別特性的修飾詞

- **abstract**

- 類別宣告為 **abstract** 表示抽象類別，不具有公開建構式（預設建構式為 **protected**，而且無法使用 **public** 或 **internal**，宣告會過，但一遇到 **new** 就掛點了）。這表示你不能直接以 **new** 關鍵字建立其執行個體。
- 抽象類別通常是用來當作父類別使用。
- 抽象類別可以擁有實做不完整的抽象成員。

- **sealed**

- 類別宣告為 **sealed** 表示為密封類別，也就是這個類別無法作為其他類別的父類別；換句話說，沒有任何一個類別可以繼承自被宣告為 **sealed** 的類別。

類別內部成員的特性修飾詞

- **abstract**

- 成員以 **abstract** 宣告表示為抽象成員，亦即這個成員不具備完整實作，而繼承此成員所在抽象類別的子類別必須實作此成員（除非子類別也被宣告為抽象類別，並且宣告此成員為抽象成員）。

- **virtual**

- 成員以 **virtual** 宣告表示為虛擬成員，而繼承此成員所在類別的子類別可以覆寫(override) 此成員

類別內部成員的特性修飾詞

- **override**

- 表示覆寫父類別成員，當此成員所在類別的父類別中是以 **abstract**，**virtual** 或 **override** 宣告的時候，可以宣告 **override** 覆寫此成員

- **sealed**

- 表示密封此成員，一定和 **override** 同時使用，表示從此以後這個成員就無法再被覆寫了

- **new**

- 表示遮蔽或隱藏成員，**99.9%** 不會用它，它是個危險的東西。
- 參考：MSDN **new** 修飾詞 (C# 參考)
- <https://msdn.microsoft.com/zh-tw/library/435f1dw2.aspx>

LAB

抽象類別與繼承的實作

新增一個方案及專案

- 方案名稱：AllClassSamples
- 專案名稱：AllClassSample001
- 範本：Console Application

加入 MyShape Class，並建立其內容

```
/// <summary>
/// 宣告一個抽象類別
/// </summary>
public abstract class MyShape
{
    /// <summary>
    /// 宣告一個抽象方法
    /// </summary>
    /// <returns></returns>
    public abstract double GetArea();
}
```

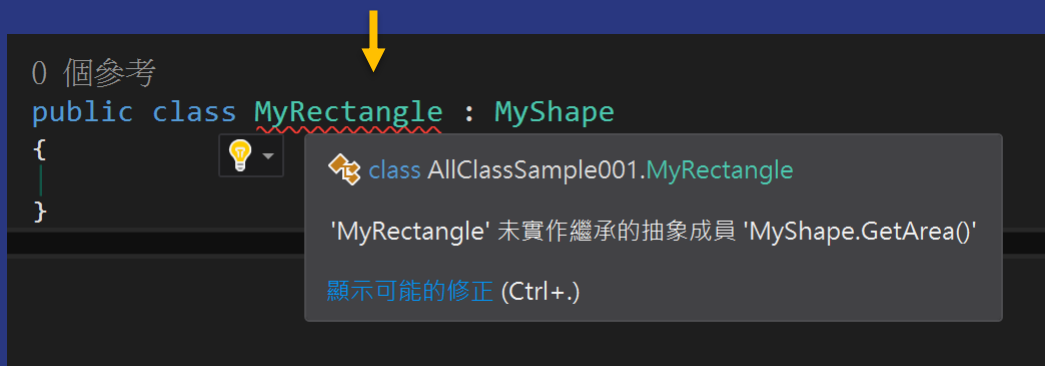
為什麼 MyShape 是抽象的

- **MyShape** 代表一個形狀的抽象，而我們目前不知道這個形狀會是方形、三角形還是圓形。所以他只有一個抽象的定義。
- 所謂形狀，就是一種封閉的平面區域，封閉平面區域必然具有面積。
- 因此，這個 **MyShape** 定義了一個抽象的方法叫 **GetArea**，子類別繼承時，會因為形狀的不同而實作不同方式取得面積的程式碼。

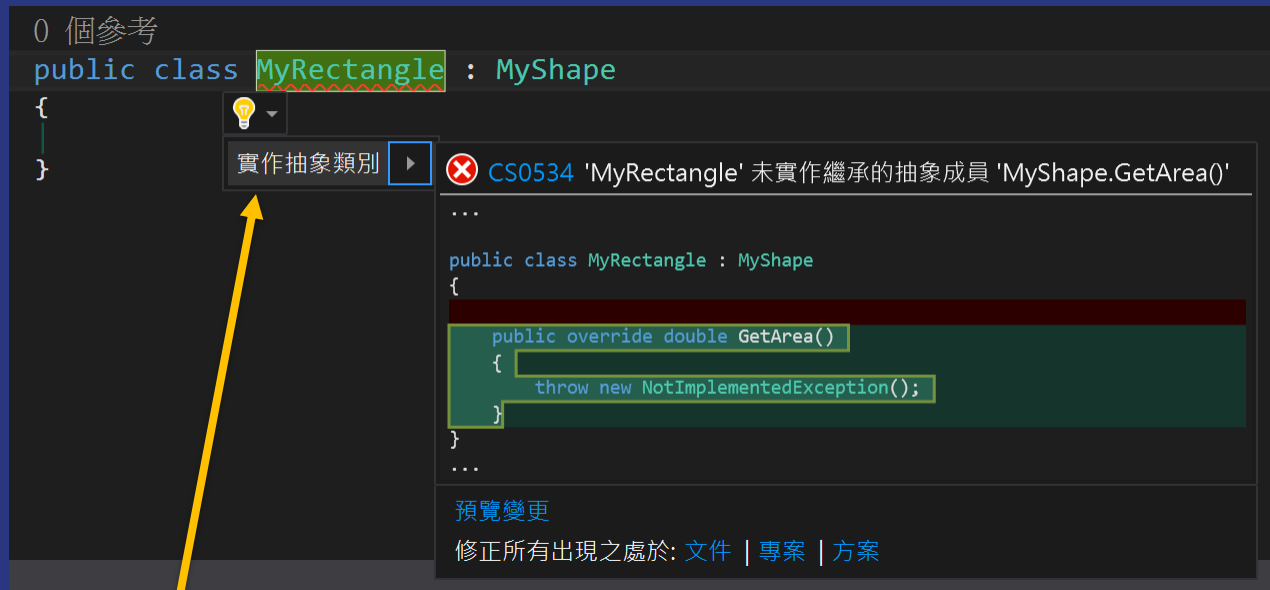
繼承 MyShape 加入 MyRectangle Class 先在此暫停一下

```
public class MyRectangle : MyShape  
{  
  
}
```

你會看到 MyRectangle 上有個紅色的波浪警示



前述的紅色波浪警示表示 `MyRectangle` 必須要實作 `MyShape` 的抽象方法（也就是 `GetArea`）但你沒有實作。
此時，請按下燈泡



Visual Studio 提示你應該要【實作抽象類別】
請按下它

Visual Studio 會幫你完成以下的程式碼
(有看到 `override` 嗎？表示 `MyRectangle` 覆寫了 `MyShape` 的 `GetArea` 方法)

```
public class MyRectangle : MyShape
{
    public override double GetArea()
    {
        throw new NotImplementedException();
    }
}
```

注意：這個 `throw new NotImplementedException()` 表示程式執行到此會拋出一個 `NotImplemented`（代表沒有實作完成）的例外，所以要把這行拿掉。

```
public class MyRectangle : MyShape
{
    public override double GetArea()
    {

    }
}
```

完成整個 MyRectangle 的內容

```
public class MyRectangle : MyShape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double GetArea()
    {
        return Width * Height;
    }
}
```

然後依樣畫葫蘆來做的圓形的類別
請不要完全用打字的，善用 Visual Studio 的功能

```
public class MyCircle : MyShape
{
    public double Radius { get; set; }
    public override double GetArea()
    {
        return Math.PI * Math.Pow(Radius, 2);
    }
}
```

這裡用到了 **Math Class**，這個類別裡面有許多專門用來
做數學計算的方法和欄位

(1) **Math.PI** 是一個唯讀欄位，代表 π

(2) **Math.Pow** 是一個靜態方法，代表乘幂運算（就是
x 的 **y** 次方這種）

回到 Program class，修改 Main method

```
static void Main(string[] args)
{
    MyShape rect = new MyRectangle() { Width = 10, Height = 10 };
    Console.WriteLine($"方形的面積是 {rect.GetArea()}");
    MyShape circle = new MyCircle() { Radius = 3 };
    Console.WriteLine($"圓形的面積是 {circle.GetArea()}");
    Console.ReadLine();
}
```

各位是否有注意到，雖然變數的型別是 **MyShape**，但是每個執行個體很盡責的呼叫了自己的覆寫後的 **GetArea**。這是一件非常重要的事情。

執行

注意

- 雖然抽象類別可以擁有抽象的成員，但未必每個成員都需要宣告成抽象的。它也可以擁有一般的成員。
- 欄位沒有抽象這回事。
- 你可以試著寫 `MyShape shape = new MyShape();` 看看會發生甚麼事。

討論

- 抽象類別在設計上佔有很重要的地位，請互相討論你們對於剛剛的範例程式的內容。
- `new MyShape();` 為何行不通？



virtual member

- 虛擬成員和抽象成員不同的是虛擬成員是具備有完整實作的內容，也就是具備完整的 { } 區段，即使內容為空都視為有實作。
- 虛擬成員和抽象成員不同的第二點是繼承者可以選擇覆寫或不覆寫這個成員，不像抽象成員一定要被覆寫。

LAB

虛擬成員與繼承的覆寫

在 AllClassSamples 加入新專案

- 方案名稱：AllClassSamples
- 專案名稱：AllClassSample002
- 範本：Console Application

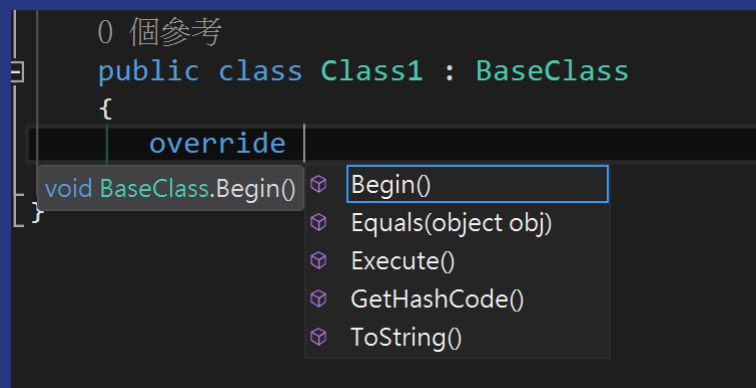
先建立 BaseClass class
兩個方法都宣告為 virtual (虛擬)

```
public class BaseClass
{
    public virtual void Execute()
    {
        Console.WriteLine("BaseClass Execute Method");
    }

    public virtual void Begin()
    {
        Console.WriteLine("BaseClass Begin Method");
    }
}
```

覆寫成員的方式

- (1) 輸入 `override`
- (2) 然後按下空白鍵
- (3) 就會出現可以覆寫的成員選單



建立 Class1 class (繼承 BaseClass)
並 override (覆寫) Execute()

```
public class Class1 : BaseClass
{
    public override void Execute()
    {
        Console.WriteLine("Class1 Execute Method");
    }
}
```

回到 Program class，修改 Main method

```
static void Main(string[] args)
{
    BaseClass o1 = new Class1();
    o1.Execute();
    o1.Begin();

    Console.ReadLine();
}
```

執行時請觀察 `o1.Execute()` 和 `o1.Begin()` 的結果

執行

- (1) 接著同一個專案繼續，建立 Class2 class(繼承 Class1)
- (2) override (覆寫) Execute() 且宣告為 sealed (密封)
- (3) Override Begin()

```
public class Class2 : Class1
{
    public override sealed void Execute()
    {
        Console.WriteLine("Class2 Execute Method");
    }

    public override void Begin()
    {
        Console.WriteLine("Class2 Execute Method");
    }
}
```

- (1) 這表示父類別(Class1)的方法(Execute())若為 override，子類別(Class2) 一樣可以覆寫它
- (2) 若祖父類別(BaseClass)的方法(Begin())若為 virtual/override，而父類別(Class1)雖然沒有覆寫，但子類別(Class2)一樣可以覆寫

回到 Program class , 修改 Main method

```
static void Main(string[] args)
{
    BaseClass o1 = new Class1();
    o1.Execute();
    o1.Begin();

    BaseClass o2 = new Class2();
    o2.Execute();
    o2.Begin();

    Console.ReadLine();
}
```

執行時請觀察 `o2.Execute()` 和 `o2.Begin()` 的結果
並與前面的結果比較

執行

- (1) 接著同一個專案繼續，建立 Class3 class(繼承 Class2)
- (2) 試圖硬要覆寫被 sealed 的 Execute
- (3) 會出現如下圖的錯誤

```
0 個參考
public class Class3 : Class2
{
    5 個參考
    public override void Execute()
    {
        Console.WriteLine("Class3")
    }
}
```

void Class3.Execute()

'Class3.Execute()': 無法覆寫繼承的成員 'Class2.Execute()'，因為其已密封。

**這些修飾詞
對於物件導向的設計
非常重要**

常數

- 以 `const` 宣告常數
- 宣告同時必須初始化
- 具不可修改之特性
- 常數一定是靜態的
- 常數的運作方式
- 常數的命名慣例通常是全大寫
- 建議使用 `readonly` 欄位取代 `const` 宣告

常數的宣告與使用

```
public class Class1
{
    public const int MAX = 100;
    public const int MIN = 0;
    public const int COUNT_OF_BOOKS = 999;
}
```

```
static void Main(string[] args)
{
    int i= 10;
    if ( (i >= Class1.MIN ) && (i <= Class1.MAX ))
    {
        Console.WriteLine("符合條件");
    }
    Console.ReadLine();
}
```

回顧一下欄位

- 「欄位」(**Field**) 是一個任意型別的變數，直接在類別或結構中宣告
- 類別 (**Class**) 或結構 (**Struct**) 可能會有執行個體 (**Instance**) 欄位或靜態 (**Static**) 欄位，或者兩個都有
- 執行個體成員和靜態成員的存取路徑是不一樣的。
- 沒有 **static** 宣告的成員就是執行個體成員，但常數除外，常數永遠都是靜態的。
- 欄位的存取修飾詞通常都不會是 **public**，但使用 **readonly** 宣告為唯讀的欄位除外。

又要談到方法了

前面我們學過甚麼？

- 方法的基本宣告
- 多載方法 (overloading method)
- 抽象方法 (abstract method)
- 虛擬方法 (virtual method)
- 覆寫方法 (override method)
- 密封方法 (override sealed method)

接下來

- 我們來談談關於方法間參數傳遞的問題
- 首先記得方法的參數傳遞只有兩種方式
 - By Value (傳值，道理和變數的複製是一樣的)
 - By Reference (傳參考或稱為傳址)
- 不論是 by value 或 by reference，對象指的是變數，請不要和 heap 中的物件混淆了。
- 所以
 - by value 是把變數的內容值複製後傳遞給被呼叫的方法
 - by reference 則是把變數的位址傳遞給被呼叫的方法，也就是說在 by reference 的情況下，呼叫端和被呼叫端會操作同一個變數。

實值型別的 by value

請開啟 ParameterSample001

```
static void Main(string[] args)
{
    int x = 0;
    Console.WriteLine($"x 的初始值為 {x}");
    int y = ChangeX(x);
    Console.WriteLine($"退出 ChangeX 方法回到 Main 方法後, x 的值為 {x}");
    Console.ReadLine();
}

private static int ChangeX(int x)
{
    Console.WriteLine($"進入 ChangeX 方法的時候, x 的值為 {x}");
    x = 10;
    Console.WriteLine($"在 ChangeX 方法重新指派後, x 的值為 {x}");
    return x;
}
```

ParameterSamples/ParameterSample001

實值型別的變數

```
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)
{
}
}
```

實值型別的 by reference

請開啟 ParameterSample002

```
static void Main(string[] args)
{
    int x = 0;
    Console.WriteLine($"x 的初始值為 {x}");
    int y = ChangeX(ref x);
    Console.WriteLine($"退出 ChangeX 方法回到 Main 方法後，x 的值為 {x}");
    Console.ReadLine();
}

private static int ChangeX(ref int x)
{
    Console.WriteLine($"進入 ChangeX 方法的時候，x 的值為 {x}");
    x = 10;
    Console.WriteLine($"在 ChangeX 方法重新指派後，x 的值為 {x}");
    return x;
}
```

ParameterSamples/ParameterSample002

by reference 時，參數和引數端都要加上 ref 關鍵字

實值型別的變數

```
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
(操作同一個變數內容)

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

參考型別的 by value

請開啟 ParameterSample003

```
class Program
{
    static void Main(string[] args)
    {
        TestClass y = new TestClass();
        Console.WriteLine($"y 實體中的 x 的初始值為 {y.x}");
        ChangeX(y);
        Console.WriteLine($"退出 ChangeX 方法回到 Main 方法後,y 實體中的 x 的值為 {y.x}");
        Console.ReadLine();
    }

    private static TestClass ChangeX(TestClass y)
    {
        Console.WriteLine($"進入 ChangeX 方法的時候, y 實體中的 x 的值為 {y.x}");
        y.x = 10;
        Console.WriteLine($"在 ChangeX 方法重新指派後,y 實體中的 x 的值為 {y.x}");
        y = new TestClass();
        Console.WriteLine
            ($"在 ChangeX 方法重新產生 TestClass 的實體後,y 實體中的 x 的值為 {y.x}");
        return y;
    }
}

public class TestClass
{
    public int x = 0;
}
```


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

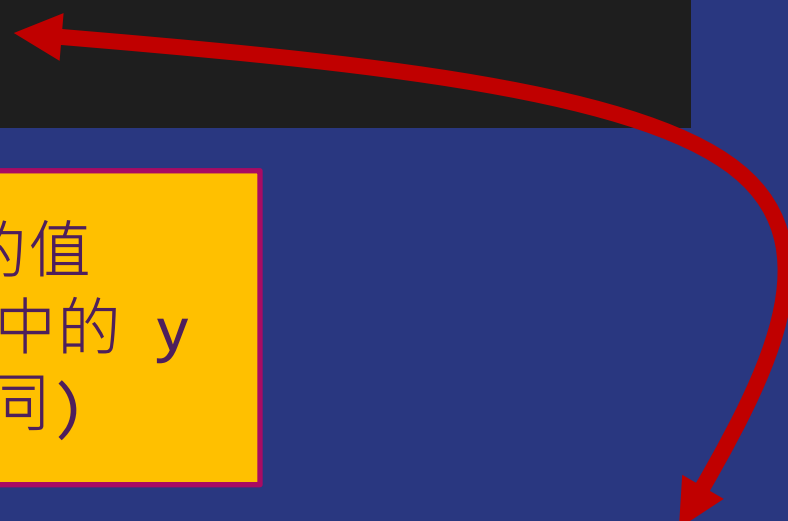
位址: `0x1200`

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)
{
}
}
```

參考型別的 by reference

請開啟 ParameterSample004

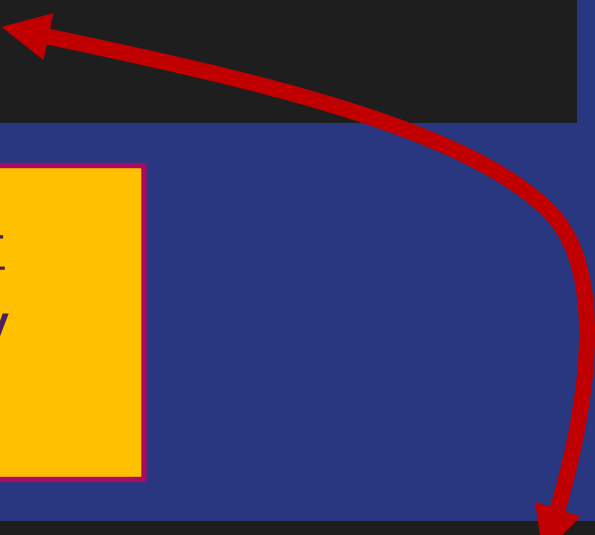
```
class Program
{
    static void Main(string[] args)
    {
        TestClass y = new TestClass();
        Console.WriteLine($"y 實體中的 x 的初始值為 {y.x}");
        ChangeX(ref y);
        Console.WriteLine($"退出 ChangeX 方法回到 Main 方法後,y 實體中的 x 的值為 {y.x}");
        Console.ReadLine();
    }

    private static TestClass ChangeX(ref TestClass y)
    {
        Console.WriteLine($"進入 ChangeX 方法的時候, y 實體中的 x 的值為 {y.x}");
        y.x = 10;
        Console.WriteLine($"在 ChangeX 方法重新指派後,y 實體中的 x 的值為 {y.x}");
        y = new TestClass();
        Console.WriteLine
            ($"在 ChangeX 方法重新產生 TestClass 的實體後,y 實體中的 x 的值為 {y.x}");
        return y;
    }
}

public class TestClass
{
    public int x = 0;
}
```

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
(操作同一個變數內容)

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

討論

- 開啟
ParameterSample005，
你可以直接看出結果嗎？



out 宣告

- out 是 by reference
- 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

LAB

使用 Int32.TryParse 方法

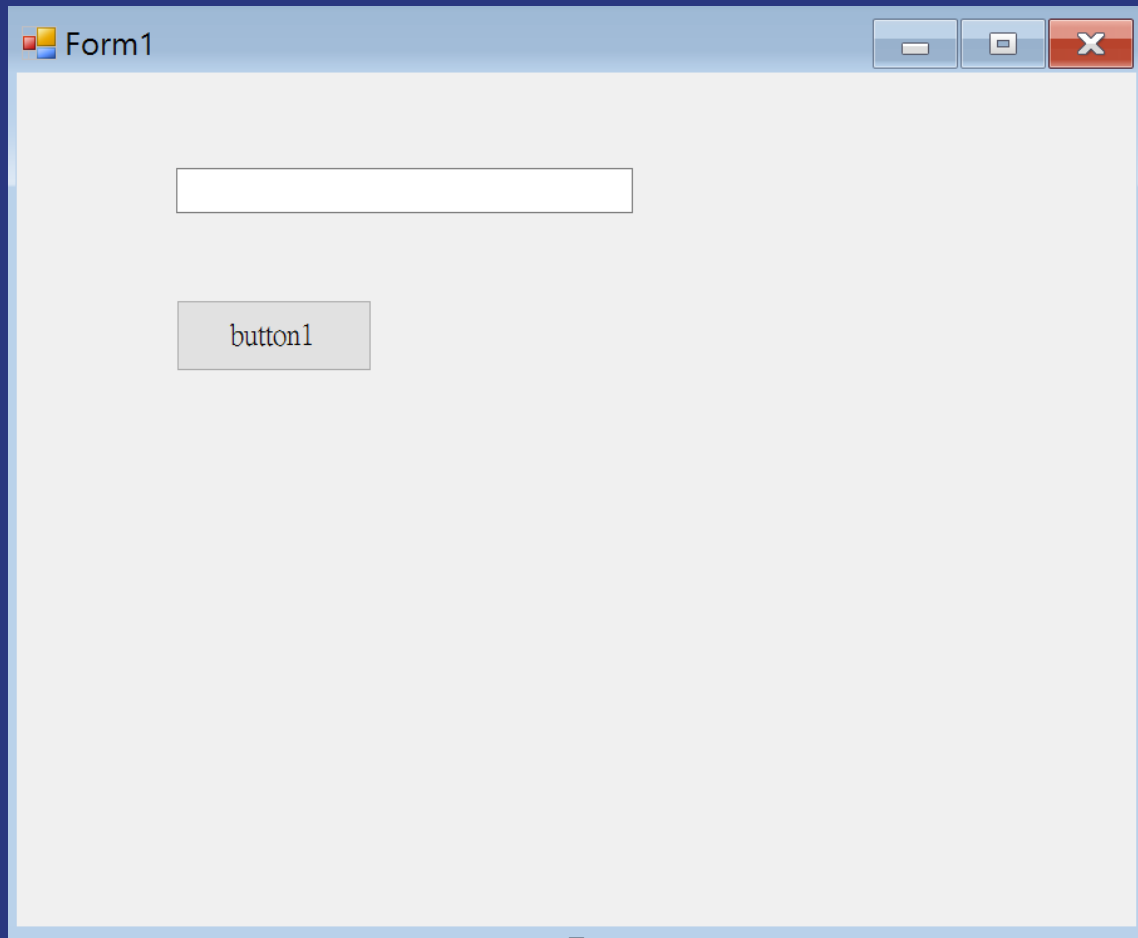
先了解以下事項

- 之前教各位使用 `Int32.Parse` 並不是我們最常用的方式，事實上比較常使用 `Int32.TryParse`
- `Int32.Parse` 在遇到輸入字串無法解析成 `Int32` 的時候會拋出例外，但 `Int32.TryParse` 會使用另一種方式處理
- 由於例外處理會耗損額外的效能，這也就是為什麼會比較建議使用 `TryParse` 的原因了
- 現在，我們要來使用 `TryParse` 。

新增一個方案及專案

- 方案名稱：OutSamples
- 專案名稱：OutSample001
- 範本：Windows Forms Application

畫面配置



一個 TextBox
一個 Button

為 Button 的 Click 事件掛上委派方法

```
private void button1_Click(object sender, EventArgs e)
{
    string source = textBox1.Text;
    int result;
    bool isParsed = int.TryParse(source, out result);
    string message;
    if (isParsed )
    {
        message = $"正確解析字串 {source} 為整數 {result}";
    }
    else
    {
        message = $"無法解析字串 {source}，只好傳回預設值 {result}";
    }

    MessageBox.Show(message);
}
```

執行

委派型別 Delegate Type

委派型別的基本觀念

- 委派型別是一種方法簽章的型別
- 在此簽章代表回傳值型別及參數型別與數量
- 可以透過委派執行個體叫用（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）
- 既然委派型別是型別的一種，表示你可以用委派型別來宣告變數或方法的參數。
- 可以宣告在 namespace 區段

委派型別的宣告

delegate 關鍵字表示這是一個委派型別的宣告

public delegate void SomeAction(string message);

回傳值是 void

表示有一個string型別的參數

這個委派型別的名稱叫 SomeAction

基於前述的宣告，以下兩個方法

(1) ShowText 符合 SomeAction 委派型別的宣告，故可以加入到由 SomeAction 產生的執行個體中

(2) ShowNumber 不符合 SomeAction 委派型別的宣告，所以無法加入到由 SomeAction 產生的執行個體中

```
static void ShowText(string msg)
{
    Console.WriteLine($"ShowText {msg}");
}

static void ShowNumber(int i)
{
    Console.WriteLine($"ShowText {i}");
}
```

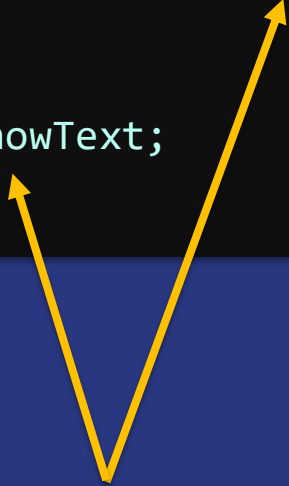
使用委派型別宣告變數並加入方法

// 標準寫法

```
SomeAction action1 = new SomeAction(ShowText);
```

// 也可以這樣寫

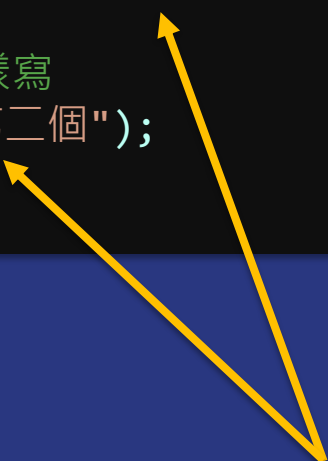
```
SomeAction action2 = ShowText;
```



要加入到委派型別所宣告的執行個體內的方法名稱

執行委派的内容

```
// 標準寫法  
action1.Invoke("第一個");  
  
//也可以這樣寫  
action2("第二個");
```



執行委派執行個體內的方法時所必須傳入的參數
還記得 **SomeAction** 型別有宣告一個 **string** 型別的參數嗎？

加入多個方法到委派執行個體

```
static void ShowText(string msg)
{
    Console.WriteLine($"ShowText {msg}");
}

static void ShowMessage(string str)
{
    Console.WriteLine($"ShowMessage {str}");
}
```

將上面兩個 method 加入到同一個委派執行個體

```
SomeAction action1 = new SomeAction(ShowText);
action1 += ShowMessage;
```

加入的順序會影響執行順序，
而且同一個方法也可以多次加入

LAB

委派的基本使用

新增一個方案及專案

- 方案名稱：DelegateSamples
- 專案名稱：DelegateSample001
- 範本：Console Application

在 namespace 區段宣告一個委派型別
(當然，你也可以開一個新檔案做這件事)

```
namespace DelegateSample001
{
    public delegate void SomeAction(string message);
}
```

在 Program class 中建立以下兩個 method

```
static void ShowText(string msg)
{
    Console.WriteLine($"ShowText {msg}");
}

static void ShowMessage(string str)
{
    Console.WriteLine($"ShowMessage {str}");
}
```


在 Program class 修改 Main method

```
static void Main(string[] args)
{
    // 標準寫法
    SomeAction action1 = new SomeAction(ShowText);
    action1 += ShowMessage;

    // 也可以這樣寫
    SomeAction action2 = ShowText;

    // 標準寫法
    action1.Invoke("第一個");

    // 也可以這樣寫
    action2("第二個");

    Console.ReadLine();
}
```

執行

LAB

委派的傳遞


委派的傳遞

- 委派既然可以宣告變數，就表示它可以和其他型別一樣把這些變數藉由參數的設定在方法間傳遞。
- 這個 lab 我們要練習模擬 `linq` 中的 `Where` 方法，請注意，這只是個模擬，並非 `Where` 真正的寫法。

在 DelegateSamples 加入新專案

- 方案名稱：DelegateSamples
- 專案名稱：DelegateSample002
- 範本：Console Application

- (1)先加入一個類別 MyCalss，先不管其內容
- (2)在這個檔案的 namespace 區段加入一個 委派



```
namespace DelegateSample002
{
    public delegate bool MyPredicate(string value);

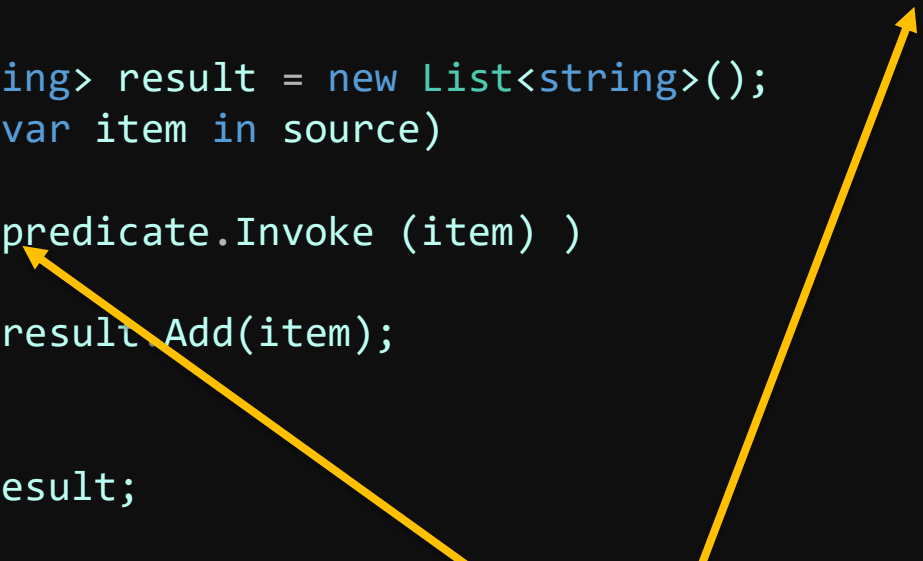
    public class MyCalss
    {
    }
}
```

接著完成 MyCalss class 的內容

```
public class MyCalss
{
    public List<string> DoWhere(List<string> source, MyPredicate predicate)
    {
        List<string> result = new List<string>();
        foreach(var item in source)
        {
            if (predicate.Invoke (item) )
            {
                result.Add(item);
            }
        }
        return result;
    }
}
```

觀察剛剛寫的 DoWhere 方法

```
public List<string> DoWhere(List<string> source, MyPredicate predicate)
{
    List<string> result = new List<string>();
    foreach(var item in source)
    {
        if (predicate.Invoke (item) )
        {
            result.Add(item);
        }
    }
    return result;
}
```



DoWhere 本身不知道該怎麼判斷，
完全靠 predicate 告訴它該怎麼判斷
所以我們會從外部傳進判斷式

在 Program class 中加入
一個符合 MyPredicate 宣告的方法

```
static bool SearchDavid(string value)
{
    return (value == "David");
}
```

如果是 “David” 則回傳 true
(是不是有點像我們寫 linq 的 Where ?)

在 Program class 修改 Main method

```
static void Main(string[] args)
{
    // 建立來源資料
    List<string> source
        = new List<string> { "Bill", "John", "David", "Tom", "David" };

    MyCalss obj = new MyCalss();
    MyPredicate predicate = SearchDavid;
    var result = obj.DoWhere(source, predicate);
    foreach (var item in result)
    {
        Console.WriteLine(item);
    }
    Console.ReadLine();
}
```

執行

LAB

匿名委派

關於匿名委派

- 剛剛那樣寫有點麻煩，還得要額外建立一個 `method`
- 於是後來演變出一種匿名委派的寫法

在 DelegateSamples 加入新專案

- 方案名稱：DelegateSamples
- 專案名稱：DelegateSample003
- 範本：Console Application

與 DelegateSample002 相同的步驟

一樣的 MyPredicate 委派宣告

一樣的 MyClass 內容

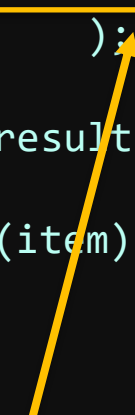
```
public delegate bool MyPredicate(string value);
public class MyClass
{
    public List<string> DoWhere(List<string> source, MyPredicate predicate)
    {
        List<string> result = new List<string>();
        foreach (var item in source)
        {
            if (predicate.Invoke(item))
            {
                result.Add(item);
            }
        }
        return result;
    }
}
```

在 Program class 修改 Main method

```
static void Main(string[] args)
{
    // 建立來源資料
    List<string> source
        = new List<string> { "Bill", "John", "David", "Tom", "David" };

    MyCalss obj = new MyCalss();
    var result = obj.DoWhere(source,
        delegate (string x) { return x == "David"; }
    );

    foreach (var item in result)
    {
        Console.WriteLine(item);
    }
    Console.ReadLine();
}
```



這就是匿名委派，回傳值型別依據程式碼推斷為 bool
(因為 == 運算子的結果一定是 bool)

執行

LAB

Lambda

關於 Lambda

- 在匿名委派以後，微軟進一步簡化成為 Lambda 表示式
- 所以整個演變就是
 1. 具名委派
 2. 匿名委派
 3. Lambda 表示式

在 DelegateSamples 加入新專案

- 方案名稱：DelegateSamples
- 專案名稱：DelegateSample004
- 範本：Console Application

與 DelegateSample002 相同的步驟

一樣的 MyPredicate 委派宣告

一樣的 MyClass 內容

```
public delegate bool MyPredicate(string value);
public class MyClass
{
    public List<string> DoWhere(List<string> source, MyPredicate predicate)
    {
        List<string> result = new List<string>();
        foreach (var item in source)
        {
            if (predicate.Invoke(item))
            {
                result.Add(item);
            }
        }
        return result;
    }
}
```

在 Program class 修改 Main method

```
static void Main(string[] args)
{
    List<string> source
        = new List<string> { "Bill", "John", "David", "Tom", "David" };

    MyCalss obj = new MyCalss();
    var result = obj.DoWhere(source,
                             (x) => { return x == "David"; }
    );

    foreach (var item in result)
    {
        Console.WriteLine(item);
    }
    Console.ReadLine();
}
```

這是標準的 Lambda 表示式

```
(x) => { return x == "David"; }
```

因為剛好區段內的程式碼只有一行，所以又可以寫成以下這樣
(只有在一行敘述的狀況才可以這樣寫)

```
(x) => x == "David"
```

我們之前 `linq` 就是這樣寫的對吧？

執行

討論

- 經過這一整輪的練習，你們對委派應該有進一步的了解，請和你的同學們相互討論，是否能明瞭委派的用法。



.Net Framework 中既有的委派型別

- **Action**：無回傳值的委派，有 17 個，分別代表參數由 0~16 個。
- **Func**：具有回傳值的委派，有 17 個，分別代表參數由 0~16 個。
- **Predicate<T>** 委派：這是一個具有一個參數和 **bool** 型別回傳值的委派。
- **EventHandler** 與 **EventHandler<T>**：這是用來宣告事件使用的特殊委派，其他還有許多 **xxxHandler** 的委派。

LAB

使用 Func 取代剛剛的範例

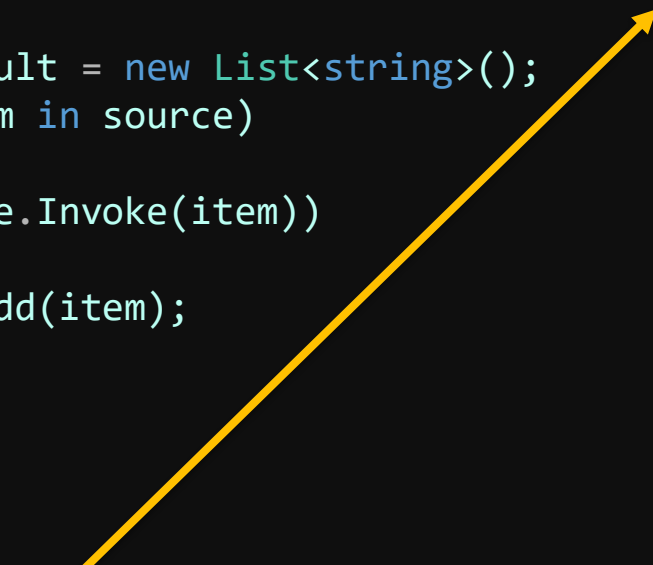
在 DelegateSamples 加入新專案

- 方案名稱：DelegateSamples
- 專案名稱：DelegateSample005
- 範本：Console Application

這次我們只建立 MyClass

使用 Func 來取代剛剛的自訂委派

```
public class MyClass
{
    public List<string> DoWhere(List<string> source, Func<string,bool> predicate)
    {
        List<string> result = new List<string>();
        foreach (var item in source)
        {
            if (predicate.Invoke(item))
            {
                result.Add(item);
            }
        }
        return result;
    }
}
```



代表這個委派是

- (1) 具有一個傳入參數，型別為 string
- (2) 回傳值的型別為 bool

在 Program class 修改 Main method

```
static void Main(string[] args)
{
    List<string> source
        = new List<string> { "Bill", "John", "David", "Tom", "David" };

    MyCalss obj = new MyCalss();
    var result = obj.DoWhere(source, (x) => x == "David");

    foreach (var item in result)
    {
        Console.WriteLine(item);
    }
    Console.ReadLine();
}
```

執行

事件 Event

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)
- 事件要宣告在型別的區段中
- 事件與事件委派函式務必要分清楚
- 事件和委派有很大的關係，請務必深入了解委派。

事件的宣告

event 關鍵字表示這是事件的宣告

public event EventHandler XChanged;


事件的名稱

這個事件所使用的委派型別
凡是要掛上這個事件的事件委派方法
都必須符合(不是相同喔)這個委派的簽章宣告

設計一個用來執行被掛給事件の委派方法的方法

```
public class Class1
{
    public event EventHandler XChanged;


    private void OnXchanged()
    {
        if (XChanged != null)
        {
            XChanged.Invoke(this, new EventArgs());
        }
    }
}
```



先確認有無委派方法掛到這個事件上

```
public class Class1
{
    public event EventHandler XChanged;

    private void OnXchanged()
    {
        if (XChanged != null)
        {
            XChanged.Invoke(this, EventArgs.Empty);
        }
    }
}
```



執行掛進來的委派方法

(1) 第一個參數就是把自己傳出去

(2) 第二個參數基本上目前是廢物，但非傳不可

所以你知道為什麼寫 **Button.Click** 事件委派方法時，
那個 **sender** 會是被按下去的那個 **Button** 了吧

為什麼這個參數會長這樣？

```
XChanged.Invoke(this, EventArgs.Empty);
```

因為 `EventHandler` 委派的宣告是長這樣

```
public delegate void EventHandler(object sender, EventArgs e);
```

小提醒：你可以將游標移到程式碼的 `EventHandler` 上，然後選【移至定義】就會看到這個宣告了。

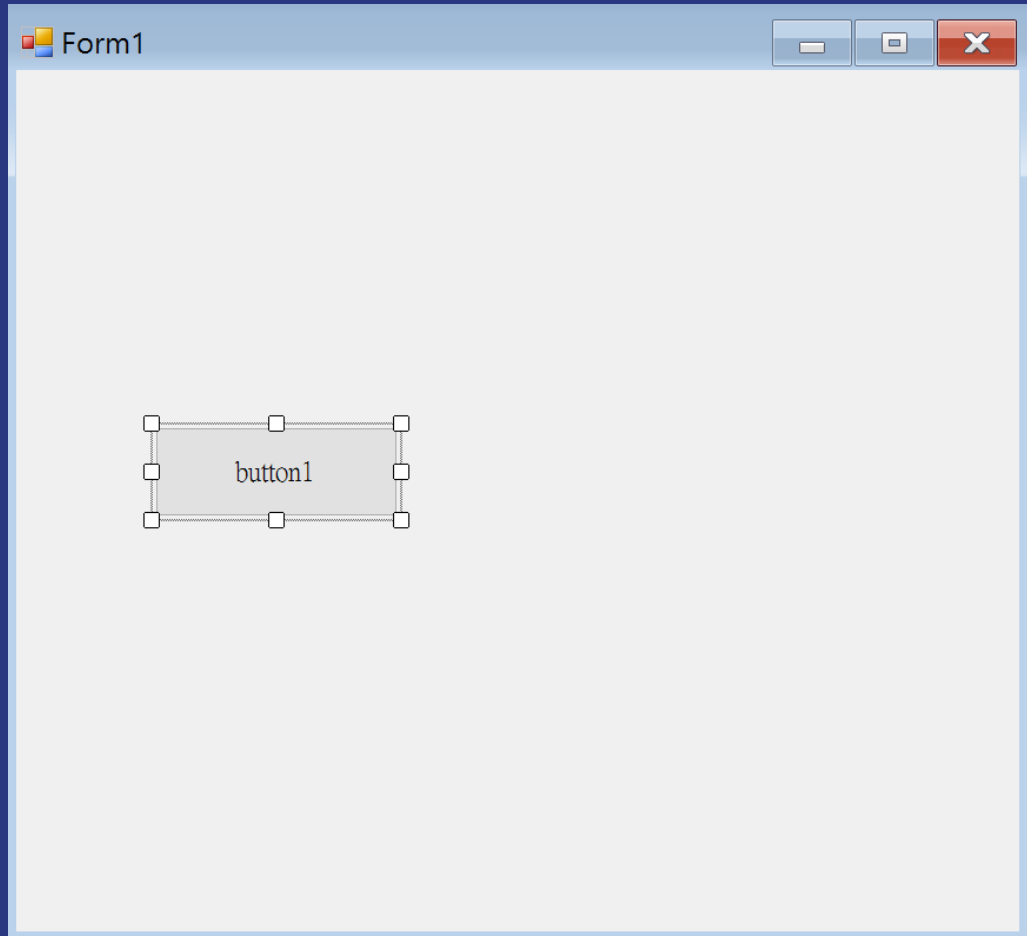
LAB

事件的基本練習

新增一個方案及專案

- 方案名稱：EventSamples
- 專案名稱：EventSample001
- 範本：Windows Forms Application

畫面配置



一個 Button

加入 MyClass 類別，並完成以下程式碼

```
public class MyClass
{
    public event EventHandler XChanged;

    private void OnXChanged()
    {
        if (XChanged != null)
        {
            XChanged.Invoke(this, EventArgs.Empty);
        }
    }

    private int _x;
    public int X
    {
        get { return _x; }
        set
        {
            if (_x != value)
            {
                _x = value;
                OnXChanged();
            }
        }
    }
}
```

若 `_x` 的值改變了，則執行 `OnXChanged` 方法，
讓 `XChanged` 事件去執行委派給他的方法們

完成 Form1 class 的必要程式碼 (記得要將 button1_Click 方法掛給 button1.Click 事件)

```
public partial class Form1 : Form
{
    private MyCalss obj;
    public Form1()
    {
        InitializeComponent();
        obj = new MyCalss();
        obj.XChanged += Obj_XChanged;
    }

    private void Obj_XChanged(object sender, EventArgs e)
    {
        MessageBox.Show("X 的值改變了");
    }

    private void button1_Click(object sender, EventArgs e)
    {
        obj.X += 1;
    }
}
```

執行

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
- 建構式也可以多載
- 建構式不會繼承
- 抽象類別的建構式通常為 `protected`

預設建構式

```
public class MyClass
{
    private int x = 0;
    private int y = 0;
}
```

當你沒有寫任何建構式的時候，編譯器在編譯的時候會幫你做一個出來，所以上面的程式碼就會變成


```
public class MyClass
{
    private int x;
    private int y;

    public MyClass() : base()
    {
        x = 0;
        y = 0;
    }
}
```

這個表示呼叫父類別的無參數建構式

```
public class MyClass
{
    private int x;
    private int y;

    public MyClass() : base()
    {
        x = 0;
        y = 0;
    }
}
```



呼叫父類別的無參數建構式可採隱含呼叫寫法(就是不寫啦)

```
public MyClass()
{
    x = 0;
    y = 0;
}
```

LAB

建構式多載

新增一個方案及專案

- 方案名稱：ConstructorSamples
- 專案名稱：ConstructorSample001
- 範本：Console Application

加入 MyBase class

```
public class BaseClass
{
    public int X { get; private set; }

    public BaseClass ()
    {
        X = 0;
    }

    public BaseClass (int y)
    {
        X = y;
    }
}
```

修改 Program class

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass o1 = new BaseClass();
        Display("o1", o1.X);

        BaseClass o2 = new BaseClass(99);
        Display("o2", o2.X);

        Console.ReadLine();
    }

    static void Display(string name, int value)
    {
        Console.WriteLine($"{name} 的 X 是 {value}");
    }
}
```

執行

建構式多載時可以呼叫另一個建構式

- 剛剛的寫法其實沒有很好
- 其實我們可以讓 `MyBaseClass()` 去呼叫 `MyBaseClass(int y)`。
- 接著我們來修改這一段

LAB

建構式多載的內部呼叫

在 ConstructorSamples 加入新專案

- 方案名稱：ConstructorSamples
- 專案名稱：ConstructorSample002
- 範本：Console Application

加入 MyBase class

```
public class BaseClass
{
    public int X { get; private set; }

    public BaseClass() : this(0)
    { }

    public BaseClass(int y)
    {
        X = y;
    }
}
```



直接呼叫 BaseClass(int y) 並將 0 傳給它

在建構式中看到：**this** 表示呼叫自己的另一個建構式

修改 Program class (和前面的範例一樣)

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass o1 = new BaseClass();
        Display("o1", o1.X);

        BaseClass o2 = new BaseClass(99);
        Display("o2", o2.X);

        Console.ReadLine();
    }

    static void Display(string name, int value)
    {
        Console.WriteLine($"{name} 的 X 是 {value}");
    }
}
```

LAB

呼叫父類別的建構式

接續前面的 Sample 加入Class1 class (繼承 BaseClass)

```
public class BaseClass
{
    public int X { get; private set; }

    public BaseClass() : this(0)
    { }

    public BaseClass(int y)
    {
        X = y;
    }
}
```

```
public class Class1 : BaseClass
{
    public int K { get; set; }
    public Class1 (int x, int y) : base(y)
    {
        K = x;
    }
}
```

呼叫父類別 BaseClass(int y) 並將 y 傳給它



修改 Program class

```
static void Main(string[] args)
{
    BaseClass o1 = new BaseClass();
    Display("o1", o1.X);

    BaseClass o2 = new BaseClass(99);
    Display("o2", o2.X);

    Class1 o3 = new Class1(55, 77);
    Display("o3", o3.X, o3.K);

    Console.ReadLine();
}

static void Display(string name, int value)
{
    Console.WriteLine($"{name} 的 X 是 {value}");
}

static void Display(string name, int v1, int v2 )
{
    Console.WriteLine($"{name} 的 X 是 {v1} , K 是 {v2}");
}
```

執行

在這裏你將學到

Learn How to Learn

- 學新東西、新技術的能力
- 尋找解答的能力
- 隨時吸取新知識的能力

跟著你一輩子的能力 ...

對學生的好處

不只是就業銜接，培養自我解決問題的能力及信心!



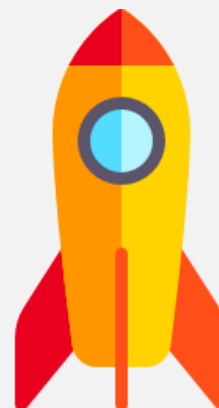
解決問題能力 (Problem-Solving)

- Learn how to learn
- Peer Learning
- Project-based



就業銜接 (Professional skills ready)

- Technical + Soft Skill



擴散及影響 (Make impacts)

- Mentor Networks