



# C# 物件導向程式開發

## 泛型 委派 Lambda Expressions

2017/02/16

網技二課

# 大綱

---

- 泛型
- 委派
- Func
- Lambda Expressions



# 泛型

設計類別的時候 先不寫死型別  
讓呼叫端決定後 填入型別

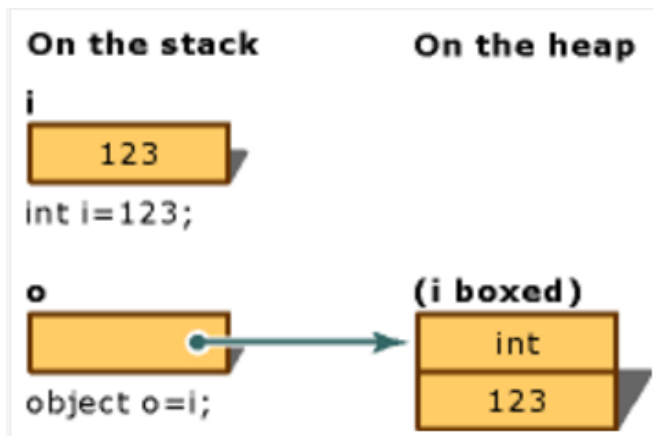
# 泛型概觀

.Net Framework 2.0 後才出現泛型

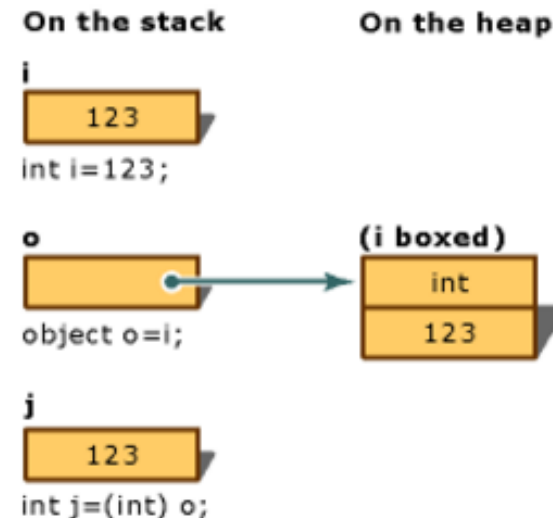
泛型是強型別的概念

避免容器操作的

## Boxing



## Unboxing



# 泛型

泛型類別 (C# 程式設計手冊)

```
class BaseNode { }  
class BaseNodeGeneric<T> { }
```

泛型介面 (C# 程式設計手冊)

```
interface IBaseInterface1<T> { }  
interface IBaseInterface2<T, U> { }
```

泛型方法 (C# 程式設計手冊)

```
void DoWork() { }  
void DoWork<T>() { }
```

泛型和陣列 (C# 程式設計手冊)

```
static void ProcessItems<T>(IList<T> coll)
```

泛型委派 (C# 程式設計手冊)

```
public delegate void Del<T>(T item);
```

**T** 是任意型別的意思 並**不是泛型**

**< >** 才是**泛型**

# 相似嗎

```
Product product;  
try  
{  
    product = JsonConvert.DeserializeObject<Product>(json);  
}  
catch (Exception)  
{  
    product = new Product();  
}
```

```
Order order;  
try  
{  
    order = JsonConvert.DeserializeObject<Order>(json);  
}  
catch (Exception)  
{  
    order = new Order();  
}
```

# 泛型方法

```
public static T JsonObject<T>(string json)
    where T:new()
{
    try
    {
        return JsonConvert.DeserializeObject<T>(json);
    }
    catch (Exception)
    {
        return new T();
    }
}
```

# 泛型約束

條件約束	描述
where T: struct	型別引數必須是實值型別。您可以指定 <code>Nullable</code> 以外的任何實值型別。如需詳細資訊，請參閱 <a href="#">用可為 Null 的類型</a> 。
where T: class	型別引數必須是參考型別，這是指任何類別、介面、委派或陣列型別。
where T: new()	型別引數必須擁有公用的無參數建構函式。將 <code>new()</code> 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。您可以同時指定多個介面條件約束。限制的介面也可以是泛型的。
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。

類型參數的條件約束

<https://msdn.microsoft.com/zh-tw/library/d5x73970.aspx>



# 設計一個類別相容下面 Json

```
{
  "Version": "V1",
  "Data": {
    "ID": 87,
    "Name": "產品"
  },
  "ErrorCode": 0
}

{
  "Version": "V1",
  "Data": "發生例外",
  "ErrorCode": 3.1
}
```

Data 跟 ErrorCode 物件跟型別不一樣  
如何克服



# 泛型介面

# 泛型類別

```
public interface IAPIData<T,U>
    where T:class
    where U:struct
{
    3 個參考| 0/1 通過
    string Version { get; set; }

    3 個參考| 0/1 通過
    T Data { get; set; }

    3 個參考| 0/1 通過
    U ErrorCode { get; set; }
}
```

```
public class APIData<T,U> : IAPIData<T,U>
    where T:class
    where U:struct
{
    3 個參考| 0/1 通過
    public string Version { get; set; }

    3 個參考| 0/1 通過
    public T Data { get; set; }

    3 個參考| 0/1 通過
    public U ErrorCode { get; set; }
}
```

# 設計一個類別相容下面 Json

```
{  
  "Version": "V1",  
  "Data": {  
    "ID": 87,  
    "Name": "產品"  
  },  
  "ErrorCode": 0  
}
```

```
public class APIData<T,U> : IAPIData<T,U>  
    where T:class  
    where U:struct  
{  
    3 個參考 | 0/1 通過  
    public string Version { get; set; }  
  
    3 個參考 | 0/1 通過  
    public T Data { get; set; }  
  
    3 個參考 | 0/1 通過  
    public U ErrorCode { get; set; }  
}
```

```
return Helper.JsonToObject<APIData<Product, int>>(json);
```

# 設計一個類別相容下面 Json

```
{  
  "Version": "V1",  
  "Data": "發生例外",  
  "ErrorCode": 3.1  
}
```

```
public class APIData<T,U> : IAPIData<T,U>  
    where T:class  
    where U:struct  
{  
    3 個參考 | 0/1 通過  
    public string Version { get; set; }  
  
    3 個參考 | 0/1 通過  
    public T Data { get; set; }  
  
    3 個參考 | 0/1 通過  
    public U ErrorCode { get; set; }  
}
```

```
return Helper.JsonToObject<APIData<string, float>>(json);
```

# 泛型類別 可以包泛型類別

```
return Helper.JsonToObject<APIData<string, float>>(json);
```

JsonToObject<T> 泛型方法

T 是 APIData<string, float>

APIData<T, U> 泛型類別

T 是 string

U 是 float

# 請實作泛型方法

```
int id = 9;  
var typeName1 = id.GetType().Name; //Int32  
  
Product p = new Product();  
var typeName2 = p.GetType().Name; //Product
```

Helper.GetTypeName(id)

Helper.GetTypeName(p)

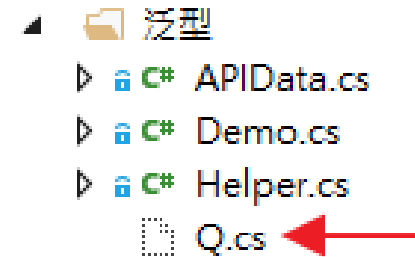
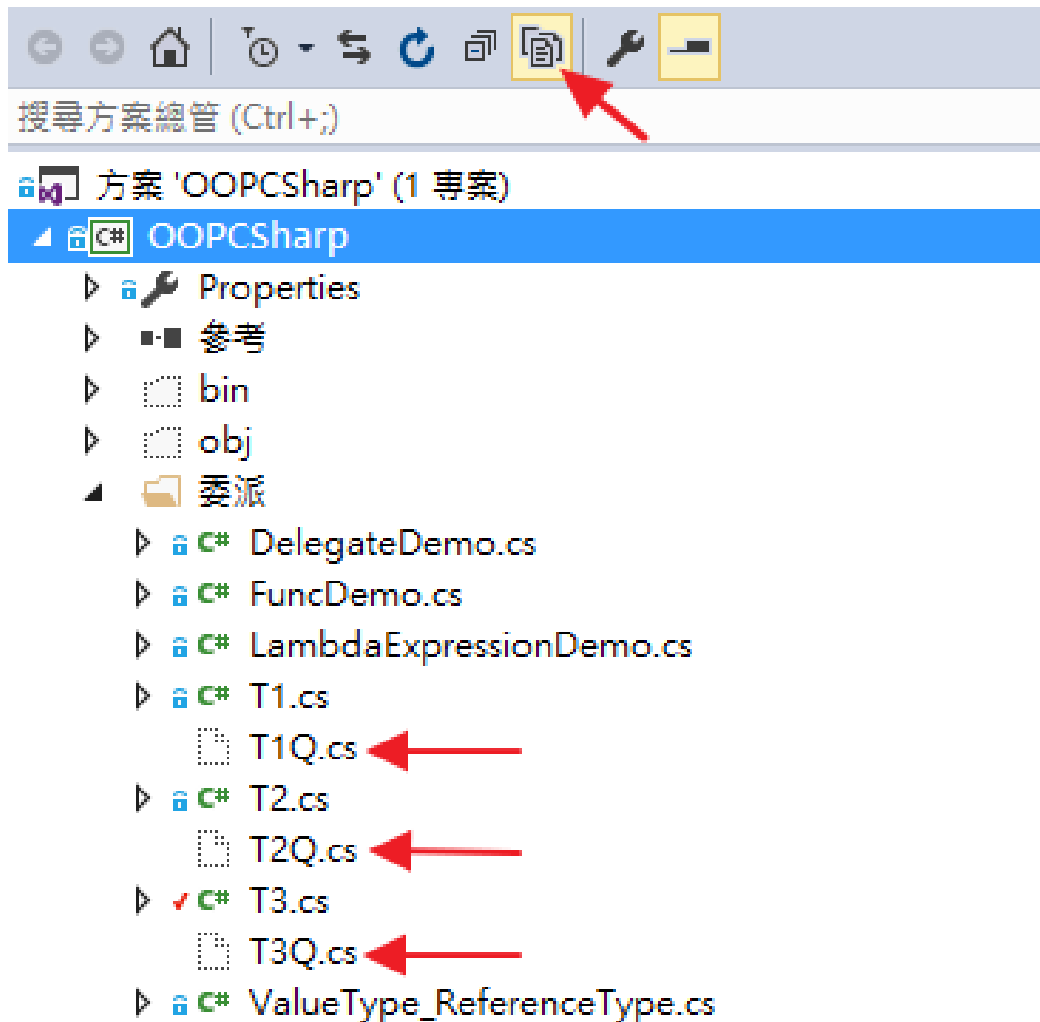
# 請實作泛型方法

```
int id = 9;  
var typeName1 = id.GetType().Name; //Int32  
  
Product p = new Product();  
var typeName2 = p.GetType().Name; //Product
```

Helper.GetTypeName(id)

Helper.GetTypeName(p)

# 實作的答案





# 請實作泛型約束

---

讓 `GetTypeName` 只能傳參考型別

`Helper.GetTypeName(id)` - Error

`Helper.GetTypeName(p)` - OK

1 where T: struct

---

2 where T : class

# 實務運用 - 發現重複程式碼

## Set

```
// Mapping to SearchParameterDto
Mapper.CreateMap<HFIDsParameter, HFIDsParameterDto>();
HFIDsParameterDto parameterModel = Mapper.Map<HFIDsParameter, HFIDsParameterDto>(parameter);

var parameterJson = JsonConvert.SerializeObject(parameterModel);
var request = actionContext.Request;
request.Properties.Add("HFIDsParameterDto", parameterJson);
```

## Get

```
// 取得驗證後的查詢參數
var parameterJson =
    ControllerContext.Request.Properties["HFIDsParameterDto"].ToString();

var parameterDto =
    JsonConvert.DeserializeObject<HFIDsParameterDto>(parameterJson);
```

# 實務運用 - 泛型方法 Set

## 原程式碼

```
var parameterJson = JsonConvert.SerializeObject(parameterModel);  
var request = actionContext.Request;  
request.Properties.Add("HFIDsParameterDto", parameterJson);
```

## 改泛型方法

```
protected void SetParameterDto<T>(T parameterModel, HttpRequestMessage request)  
    where T : IParameterDTO  
{  
    var parameterJson = JsonConvert.SerializeObject(parameterModel);  
    request.Properties.Add(parameterModel.GetType().Name, parameterJson);  
}
```

## 使用泛型方法

```
var parameterModel = Mapper.Map<RecommendParameterDto>(parameter);  
SetParameterDto(parameterModel, actionContext.Request);
```

# 實務運用 - 泛型方法 Get

## 原程式碼

```
// 取得驗證後的查詢參數
var parameterJson =
    ControllerContext.Request.Properties["HFIDsParameterDto"].ToString();

var parameterDto =
    JsonConvert.DeserializeObject<HFIDsParameterDto>(parameterJson);
```

## 改泛型方法

```
protected T GetParameterDto<T>() where T : IParameterDTO
{
    var parameterJson = ControllerContext.Request.Properties[typeof(T).Name].ToString();
    var parameterDto = JsonConvert.DeserializeObject<T>(parameterJson);
    return parameterDto;
}
```

## 使用泛型方法

```
var recommendParameterDto = GetParameterDto<RecommendParameterDto>();
recommendParameterDto.MobileID = requestParameterDto.MobileID;
```

# 委派

方法可以當參數傳

# 參考型別 vs 實質型別

```
public void Demo1()
{
    var i = 5;
    var o = new Order()
    {
        ID = 9,
        Name = 'a'
    };

    Method(o, i);
    Console.WriteLine(i);
    Console.WriteLine(o.ID);
    Console.WriteLine(o.Name);
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
private void Method(Order no, int ni)
{
    ni = 6;
    no.ID = 10;
    no.Name = 'b';
}
```



```
public class Order
{
    3 個參考 | 0 項變更 | 0 位作者, 0 項變更
    public int ID { get; set; }

    3 個參考 | 0 項變更 | 0 位作者, 0 項變更
    public char Name { get; set; }
}
```

# 參考型別 vs 實質型別


```
public void Demo1()
{
    var i = 5;
    var o = new Order()
    {
        ID = 9,
        Name = 'a'
    };

    Method(o, i);
    Console.WriteLine(i);
    Console.WriteLine(o.ID);
    Console.WriteLine(o.Name);
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
private void Method(Order no,
{
    ni = 6;
    no.ID = 10;
    no.Name = 'b';
}
```

Stack			Heap	
i	06ffe8dc	5	02EDF258	ID = 10 Name = b
o	079eebc8	02EDF258		
ni	079eebb8	6		
no	079eeb70	02EDF258		



[C#] 基礎 - Value Type , Reference Type 用看記憶體內容 來測試

[https://dotblogs.com.tw/initials/2017/01/28/a00\\_basis](https://dotblogs.com.tw/initials/2017/01/28/a00_basis)



# 委派

---

委派是一種方法簽章的型別

委派可以用來將方法當做參數傳

C# 中的委派是重疊的（鏈式委派）



# 委派

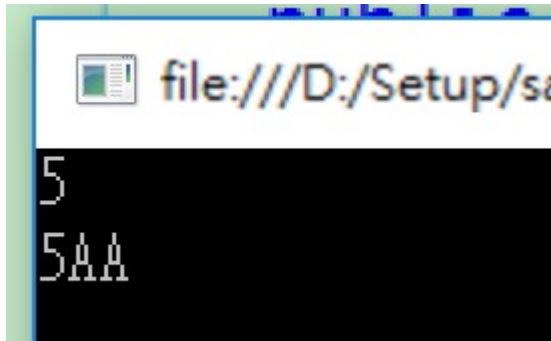


```
public class DelegateDemo
{
    //定義 委派型別
    public delegate string MyDelegate(int x);

    0 個參考 | 0 項變更 | 0 位作者, 0 項變更
    public void Demo1()
    {
        MyDelegate doo = new MyDelegate(Method);
        string result = doo.Invoke(5);
    }

    1 個參考 | 0 項變更 | 0 位作者, 0 項變更
    public string Method(int x)
    {
        var temp = x.ToString();
        return temp;
    }
}
```

# 鏈式委派



```
//定義 委派型別  
public delegate string MyDelegate(int x);
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public void Demo1()  
{  
    //加入第一個方法  
    MyDelegate doo = new MyDelegate(Method);  
  
    //加入第二個方法  
    doo += Method2;  
    string result = doo.Invoke(5);  
}
```

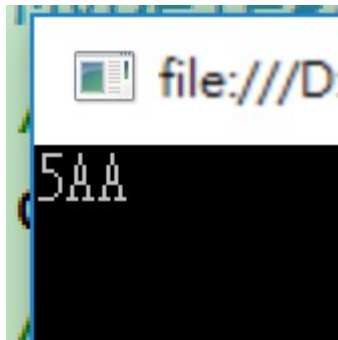
1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    Console.WriteLine(temp);  
    return temp;  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method2(int x)  
{  
    var temp = x.ToString() + "AA";  
    Console.WriteLine(temp);  
    return temp;  
}
```

# 鏈式委派



//定義 委派型別

```
public delegate string MyDelegate(int x);
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public void Demo1()
```

```
{
```

```
    MyDelegate doo = new MyDelegate(Method);
```

```
    doo += Method2;
```

```
    //移除第一個方法
```

```
    doo -= Method;
```

```
    string result = doo.Invoke(5);
```

```
}
```

2 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method(int x)
```

```
{
```

```
    var temp = x.ToString();
```

```
    Console.WriteLine(temp);
```

```
    return temp;
```

```
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method2(int x)
```

```
{
```

```
    var temp = x.ToString() + "AA";
```

```
    Console.WriteLine(temp);
```

```
    return temp;
```

```
}
```



# 委派 - 簡化寫法

//完整寫法

```
MyDelegate doo = new MyDelegate(Method);  
string result = doo.Invoke(5);
```

//可簡寫 (通常都用簡寫)

```
MyDelegate doo2 = Method;  
string result2 = doo(5);
```

# 委派 - 方法當做參數傳

//定義 委派型別

```
public delegate string MyDelegate(int x);
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public void Demo1()
```

```
{
```

```
    MyDelegate doo2 = Method;
```

```
    Demo2(doo2);
```

```
}
```

//方法當做參數傳

//MyDelegate 表示 我要傳入一個int 跟回傳string的方法

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public void Demo2(MyDelegate d)
```

```
{
```

```
    string result2 = d(10);
```

```
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method(int x)
```

```
{
```

```
    var temp = x.ToString();
```

```
    Console.WriteLine(temp);
```

```
    return temp;
```

```
}
```

# 參數可以傳

```
public void Demo3(int a ,Order o,MyDelegate d)...
```

int 實質型別

Order 參考型別

MyDelegate 委派型別

**傳方法** ( 傳入 int 回傳 string)

```
//定義 委派型別  
public delegate string MyDelegate(int x);
```

# 請實作委派

---

1. 請定義一個委派型別為  
輸入 float 回傳 double 的方法  
( 方法內容可直接回傳傳入值 )  
此委派型別命名一律取名叫做 Func

請寫在 T1.cs

# Func

簡化 delegate 的寫法





# 科技來自人性（懶）

```
//public delegate string MyDelegate(int x);  
public delegate string Func(int x);
```

0 個參考 | 0 項變更 | 0 位作者，0 項變更

```
public void Demo()  
{  
    Func doo = Method;  
    string result = doo(5);  
}
```

1 個參考 | 0 項變更 | 0 位作者，0 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    return temp;  
}
```



# 科技來自人性（懶）

```
//public delegate string MyDelegate(int x);  
//public delegate string Func(int x);  
public Func<int, string> doo;
```

0 個參考 | 0 項變更 | 0 位作者，0 項變更

```
public void Demo()  
{  
    //Func doo = Method;  
    doo = Method;  
    string result = doo(5);  
}
```

1 個參考 | 0 項變更 | 0 位作者，0 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    return temp;  
}
```



# 科技來自人性（懶）

最後可以直接在方法內使用

```
public void Demo()  
{  
    //Func doo = Method;  
    Func<int, string> doo = Method;  
    string result = doo(5);  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    return temp;  
}
```

# Func

---

## 有回傳的委派是 Func

Func(TResult) 委派

Func(T, TResult) 委派

Func(T1, T2, TResult) 委派

Func(T1, T2, T3, TResult) 委派

Func(T1, T2, T3, T4, TResult) 委派

Func(T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult) 委派

# 比一比

```
public delegate string MyDelegate(int x);
```

1 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public void Demo1()  
{  
    MyDelegate doo = Method;  
    string result2 = doo(5);  
    Demo2(doo);  
}
```

1 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public void Demo2(MyDelegate d)  
{  
    string result2 = d(10);  
}
```

2 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    Console.WriteLine(temp);  
    return temp;  
}
```

```
public void Demo1()
```

```
{  
    Func<int, string> doo = Method;  
    string result2 = doo(5);  
    Demo2(doo);  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public void Demo2(Func<int, string> d)  
{  
    string result2 = d(10);  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    Console.WriteLine(temp);  
    return temp;  
}
```

# 熟練轉換

```
public void Demo()  
{  
    Func<string> doo1 = Method1;  
    Func<int, int> doo2 = Method2;  
    Func<int, DateTime, string> doo3 = Method3;  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method1() { return "a"; }
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public int Method2(int x) { return 1; }
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method3(int x, DateTime y) { return "a"; }
```

# Func

---

## 有回傳的委派是 Func

Func(TResult) 委派

Func(T, TResult) 委派

Func(T1, T2, TResult) 委派

Func(T1, T2, T3, TResult) 委派

Func(T1, T2, T3, T4, TResult) 委派

Func(T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult) 委派

# 請實作 Func

---

T1.cs 的程式碼改寫成 Func  
請寫在 T2.cs



# Action

---

## 沒有回傳的委派是 Action

Action(T) 委派

Action(T1, T2) 委派

Action(T1, T2, T3) 委派

Action(T1, T2, T3, T4) 委派

Action(T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16) 委派

# Lambda Expression

簡化 方法 的寫法

# 簡化目標

```
public delegate string MyDelegate(int x);
```

1 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public void Demo1()  
{  
    MyDelegate doo = Method;  
    string result2 = doo(5);  
    Demo2(doo);  
}
```

1 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public void Demo2(MyDelegate d)  
{  
    string result2 = d(10);  
}
```

2 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    Console.WriteLine(temp);  
    return temp;  
}
```

```
public void Demo1()  
{
```

```
    Func<int, string> doo = Method;  
    string result2 = doo(5);  
    Demo2(doo);  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public void Demo2(Func<int, string> d)  
{  
    string result2 = d(10);  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    Console.WriteLine(temp);  
    return temp;  
}
```

# 匿名方法

把 public string Method 換成 delegate

```
public void Demo3()
{
    //Func<int, string> doo = Method;
    Func<int, string> doo1 = delegate (int x)
    {
        var temp = x.ToString();
        return temp;
    };
    Func<int, string> doo2 = delegate (int x) { return x.ToString(); };
    string result = doo1(5);
}

0 個參考 | 0 項變更 | 0 位作者, 0 項變更
public string Method(int x)
{
    var temp = x.ToString();
    return temp;
}
```

# Lambda Expression


最大的差異 就是不用打 `delegate` 改打 `=>` (念作 goes to)  
位置換到輸入參數的 右邊去了

有大括號可以多行程式碼的寫法叫做**陳述式 Lambda**

```
// 匿名方法
Func<int, string> doo1 = delegate (int x)
{
    var temp = x.ToString();
    return temp;
};

//Lambda Expression
//陳述式 Lambda (有大括號 可多行)
Func<int, string> doo3 = (int x) =>
{
    var temp = x.ToString();
    return temp;
};

Func<int, string> doo2 = delegate (int x) { return x.ToString(); };
Func<int, string> doo4 = (int x) => { return x.ToString(); };
```



# Lambda Expression 更簡化

還有更簡化的寫法就是想辦法簡化為一行 code

就可以去除大括號 跟 return 就叫做**運算式 Lambda**

```
//陳述式 Lambda
```

```
Func<int, string> doo4 = (int x) => { return x.ToString(); };
```

```
//運算式 Lambda
```

```
Func<int, string> doo5 = (int x) => x.ToString();
```

# 輸入更簡化

其實 輸入參數可以由 Func 就知道型別 所以 int x 的 **int** 可不打  
再加上只有一個參數時連**括號**都**不用打**

```
Func<int, string> doo5 = (int x) => x.ToString();  
Func<int, string> doo6 = x => x.ToString();
```

不過當有**兩個參數**時就**不能省**略**括號**的

```
Func<int, float, string> b3 = (int x, float y) => (x + y).ToString();  
Func<int, float, string> b4 = (x, y) => (x + y).ToString();
```

# Lambda Expression 練習

---

T1.cs 的程式碼改寫成 Func  
請寫在 T2.cs



# 比一比

```
public delegate string MyDelegate(int x);
```

1 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public void Demo1()  
{  
    MyDelegate doo = Method;  
    string result2 = doo(5);  
    Demo2(doo);  
}
```

1 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public void Demo2(MyDelegate d)  
{  
    string result2 = d(10);  
}
```

2 個參考 | sam, 44 分鐘前 | 1 位作者, 1 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    Console.WriteLine(temp);  
    return temp;  
}
```

# A

```
public void Demo1()  
{  
    Func<int, string> doo = Method;  
    string result2 = doo(5);  
    Demo2(doo);  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public void Demo2(Func<int, string> d)  
{  
    string result2 = d(10);  
}
```

1 個參考 | 0 項變更 | 0 位作者, 0 項變更

```
public string Method(int x)  
{  
    var temp = x.ToString();  
    Console.WriteLine(temp);  
    return temp;  
}
```

# B

```
public void Demo1()  
{  
    Demo2(x=> x.ToString());  
}
```

1 個參考 | sam, 1 天前 | 1 位作者, 1 項變更

```
public void Demo2(Func<int, string> d)  
{  
    string result2 = d(10);  
}
```

# C

# 可能是你天天都用 但是卻不知道的功能

---

Linq 的 where 就是 Delegate (Func)

```
st.Where(x => x.Price < 250).ToList());
```

(擴充功能) `IEnumerable<Book> IEnumerable<Book>.Where<Book>(Func<Book, bool> predicate)`

根據述詞來篩選值序列。

*predicate:* 用來測試每個項目是否符合條件的函式。

謝謝大家