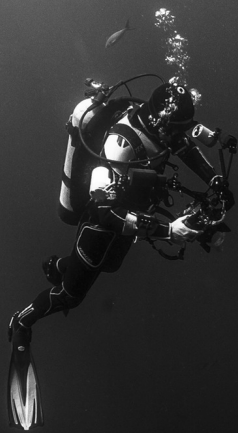


Alpha 版



# C# 本事

蔡煥麟 / 著

# C# 本事

Michael Tsai

This book is for sale at <http://leanpub.com/csharp-kungfu>

This version was published on 2015-08-25



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 Michael Tsai

# Contents

前言	i
關於本書	i
何處購買?	i
本書定位	i
閱讀建議	ii
書寫慣例	ii
需要準備的工具	ii
關於作者	iii
1. C# 基礎語法	1
1.1 型別	1
使用 var 宣告隱含型別	1
使用 dynamic 宣告動態型別	2
1.2 可為 null 的型別	3
宣告與初始化	3
賦值與取值	4
判斷是否有值	4
與非 Nullable 型別混用	5
1.3 物件初始設定式	6
1.4 自動實作屬性	6
1.5 匿名型別	7
投射初始設定式	8
注意事項	9
1.6 擴充方法	10
1.7 C# 6 新增的語法	12
nameof 表示式	12
字串插值	13
Null 條件運算子	14
自動屬性的初始設定式	15
以表示式為本體的成員	17
using static 陳述式	18
2. 泛型	20
2.1 為什麼要有泛型?	20

## CONTENTS

	手動轉型的問題 . . . . .	20
	土法煉鋼 . . . . .	21
	泛型之美 . . . . .	23
2.2	細說泛型 . . . . .	24
	泛型、型別參數、建構的型別 . . . . .	24
	泛型是型別的樣板 . . . . .	25
	建構式與解構式 . . . . .	28
	預設值的表示法 . . . . .	28
	型別參數的條件約束 . . . . .	29
	泛型介面與結構 . . . . .	32
2.3	泛型方法 . . . . .	34
	型別推斷 . . . . .	35
	擴充方法與泛型類別 . . . . .	36
2.4	泛型與 C++ 樣板的差異 . . . . .	37
2.5	泛型的型別相容問題 . . . . .	38
	核心概念：型別相容 . . . . .	39
	Covariance . . . . .	40
	Contravariance . . . . .	44
2.6	結語 . . . . .	45

# 前言



本書仍在 beta 階段，內容會不定期更新。

## 關於本書

本書將介紹 C# 語言當中幾個筆者認為屬於比較重要、進階的議題，或比較不容易理解的部分。

## 何處購買？

您可以透過下列電子書平台購買本書：

- Leanpub: <https://leanpub.com/u/michaeltsai>
- Pubu: <http://www.pubu.com.tw/store/huanlin>

由於各平台的功能有一些差異，使得同一本書在不同書店的定價和閱讀體驗不太一致，這點還請讀者明察。基本上，Leanpub 的功能比較豐富，而 Pubu 則提供了更多種付款方式，而且中文介面用起來更直觀。

## 本書定位

本書不適合從來沒寫過 C# 程式的人，因為本書不會介紹 C# 的一些基礎語法，例如：

- 如何宣告變數、呼叫函式、傳遞參數（傳值、傳參考、具名引數）。
- 如何撰寫迴圈、決策敘述。
- 如何定義類別、類別成員（欄位、方法、屬性、索引子），以及各種存取層級（private、protected、internal、public）的作用。
- 物件導向程式語言的相關概念，例如封裝、繼承、多型、抽象類別、改寫（override）等等。
- 如何處理執行時期的例外（exception）。

此外，雖然本書的範例是以 Visual Studio 撰寫，但書中不會提及工具的操作步驟。例如：建立新專案、加入組件參考、從主選單點某某項目、然後點這個、再點右鍵.... 諸如此類的。如有特別需要，則會以補充說明欄的方式加註操作指引。

## 閱讀建議

(TODO)

## 書寫慣例

對於某些不易翻譯成中文的術語，除了第一次出現時採取中英並呈，往後再碰到相同術語時，通常會直接使用英文——這並非絕對，主要還是以降低閱讀阻力為優先考量。

書中不時會穿插一些與正文有關的補充資料，依不同性質，有的是以單純加框的側邊欄 (sidebar) 圈住，有的則會佐以不同的圖案。底下是書中常用的幾個圖案：



此區塊會提示當前範例原始碼的所在位置。



注意事項。



相關資訊。



通常是筆者的碎碎念、個人觀點（不見得完全正確或放諸四海皆準），或額外的補充說明。

## 需要準備的工具

本書範例皆以 C# 寫成，使用的開發工具是 Visual Studio 2013。為了能夠一邊閱讀、一邊實作練習，您的 Windows 作業環境至少需要安裝下列軟體：

- .NET Framework 4.5
- Visual Studio Community 2013 或 Professional 以上的版本

上述軟體皆可免費取得。其中 Visual Studio Community 2013 是微軟於 2014 年 11 月發布的社群版，可[免費下載](http://www.visualstudio.com/products/visual-studio-community-vs)<sup>1</sup>使用（若於企業內部使用，請留意限制條款）。

---

<sup>1</sup><http://www.visualstudio.com/products/visual-studio-community-vs>

## 關於作者

.NET 程式設計師，現任 C# MVP（2007 年至今），有幸曾站在恆逸講台上體會「好為人師」的滋味，也翻譯過幾本書。

近期著作：

- [《.NET 本事－非同步程式設計》<sup>2</sup>](#)，2015。
- [《.NET 相依性注入》<sup>3</sup>](#)，2014。

陳年譯作：

- [軟體構築美學<sup>4</sup>](#)，2010（已絕版）。原著：Brownfield Application Development in .NET。
- [物件導向分析設計與應用<sup>5</sup>](#)，2009。原著：Object-Oriented Analysis and Design with Applications 3rd Edition。
- [ASP.NET AJAX 經典講座<sup>6</sup>](#)，2007。原著：Introducing Microsoft ASP.NET AJAX。
- [微軟解決方案框架精要<sup>7</sup>](#) 2007。原著：Microsoft Solution Framework Essentials。
- [軟體工程與 Microsoft Visual Studio Team System<sup>8</sup>](#)，2006。原著：Software Engineering and Mcicrosoft Visual Studio Team System。

您可以透過下列管道得知作者的最新動態：

- 部落格：<http://huan-lin.blogspot.com/>
- Facebook 專頁：<https://www.facebook.com/huanlin.notes>

---

<sup>2</sup><https://leanpub.com/dotnet-async>

<sup>3</sup><https://leanpub.com/dinet>

<sup>4</sup><http://www.books.com.tw/products/0010485217>

<sup>5</sup><http://www.books.com.tw/products/0010427868>

<sup>6</sup>[http://www.delightpress.com.tw/book.aspx?book\\_id=SKTP00007](http://www.delightpress.com.tw/book.aspx?book_id=SKTP00007)

<sup>7</sup><http://www.books.com.tw/products/0010357719>

<sup>8</sup><http://www.ithome.com.tw/node/40288>

# 1. C# 基礎語法

這裡的基礎,指的並不是如迴圈(例如 `for`、`foreach`)、決策(例如 `if...else`、`switch...case`)等基本元素,而是其他稍微進階一點、或更便捷的 C# 語法,包括:

- 使用 `var` 宣告隱含型別
- 可為 `null` 的型別 (nullable types)
- 物件初始設定式
- 自動實作屬性
- 匿名型別
- 擴充方法
- C# 6 新增的語法

若您已經很熟悉這些 C# 語法,便可略讀或直接跳到下一章。

## 1.1 型別

這一節要介紹一些與型別宣告有關的語法,包括 `var` 關鍵字、匿名型別、可為 `null` 的型別(nullable types)。



這裡假設讀者已經知道什麼是實質型別 (value types) 和參考型別 (reference types)。

### 使用 `var` 宣告隱含型別

C# 從 3.0 開始增加了 `var` 關鍵字,可用來宣告隱含型別,例如:

```
1      int i = 10;  
2      var j = 10;
```

這兩行的作用完全相同,連編譯出來的 IL 代碼也都一個模樣。這是因為宣告為 `var` 的變數其實在編譯時期就已經決定型別了。

那麼,編譯器如何決定型別呢? 透過你指定給變數的初始值來推測。因此,使用 `var` 變數時一定要給初始值,否則無法通過編譯。

再看一個 `var` 的範例:



```
1 var numbers = new[] { 10, 20, 30, 40 };
2
3 foreach (var x in numbers)
4 {
5     Console.WriteLine(x);
6 }
```

編譯器將會推測 `numbers` 的型別為整數陣列 (`Int32[]`)，迴圈計數值 `x` 則為 `Int32`。注意在建立此整數陣列時，使用的語法是 `new[]` 而不是一般的 `new[int]`。

此外，使用 `var` 宣告陣列時，陣列的所有元素都必須是同一型別，否則編譯器無法決定該用甚麼型別。例如下面這行程式碼將無法通過編譯：

```
var mixedAry = new[] { 10, "abc" }; // 錯誤！編譯器無法決定該用什麼型別。
```

除了必須給定初始值，`var` 還有下列限制：

- 只能用在區域變數。函式的參數也不能宣告為 `var`。
- 每一個 `var` 宣告只能有一個變數，所以你不能這樣寫：

```
var i = 10, j = 20; // 編譯失敗！
```



有一種情況一定得用 `var`，就是當你使用匿名型別 (anonymous type) 的時候。稍後會介紹匿名型別。

## 使用 `dynamic` 宣告動態型別

基本上，可以這麼說：

使用 `var` 宣告變數的意思就是：由編譯器來推測變數的實際型別。

使用 `dynamic` 來宣告變數則是：由 .NET CLR (Common Language Runtime) 來決定變數在執行時期的型別。

底下是一些範例，請從右邊的註解來了解每一行程式碼的用意：

```
1 var s1 = "hello";    // 編譯時期的型別為 string, 執行時期的型別也是 string
2 dynamic s2 = "hello"; // 編譯時期的型別為 dynamic, 執行時期的型別為 string
3 int i = s1;          // 編譯時就會出錯!
4 int j = s2;          // 執行時才會出錯!
```

此外，你可以把一個宣告為 `dynamic` 的變數指定給另一個宣告為 `var` 的變數。不過，混合使用 `dynamic` 和 `var` 變數時，會有一些需要特別注意的細節。以下是幾個混用的例子：

```
1 dynamic s1 = "hello";
2 var v = s1;    // 編譯時期的型別為 dynamic
3 int i = v;     // 執行時會出錯!
4
5 dynamic x = 2;
6 var y = x * 10; // 編譯時期的型別為 dynamic
7 var z = (int) x; // 編譯時期的型別為 int
```

## 1.2 可為 null 的型別

可為 null 的型別，指的就是泛型 `System.Nullable<T>`（泛型在本書第 2 章有介紹）。其中的型別參數 `T` 可以是任何實質型別，包括結構（例如 `int`、`bool`、或 `DateTime`），但不可以是參考型別。

顧名思義，`Nullable` 就是允許變數值為 `null` 的型別。顯然這不是用在參考型別（reference types），因為參考型別的變數值本來就可以是 `null`。換句話說，`Nullable<T>` 的用途就是讓實質型別的變數可以是「空值」。



在關聯式資料庫中，通常每一種欄位都可以指定是否允許空值（NULL）。早期沒有 `Nullable<T>` 的時代（.NET 1.0），在處理可為空值的資料欄位時並不直觀——當時常見的一種作法是把特定數值來當成空值來判斷，例如 `int` 變數值若為 `-999999` 就表示沒有值，`DateTime` 變數則常以 `DateTime.MinValue` 來表示空值，另外還有 ADO.NET 的 `DBNull.Value` 也是用來判斷欄位值是否為 `null` 的方法。有了 `Nullable<T>` 之後，就不用再使用以前那些比較繞彎的方法了。

### 宣告與初始化

除了直接使用 `System.Nullable<T>` 來宣告可為 null 的變數，C# 還提供了更簡便的寫法：直接在實質型別後面加一個問號。參考以下範例：

```
1 Nullable<int> x;    // (1) 這種寫法太累，要打好多字。
2 int? y;            // (2) 作用同上，但簡潔、清楚多了。
```

簡單起見，往後都用比較簡潔的寫法。

## 賦值與取值

欲設定 `Nullable` 變數的值，可以透過 `Value` 屬性，亦可直接賦值。例如：

```
1 int? x;
2 x.Value = 20;
3 x = 20;           // 作用同上
```

同樣地，讀取變數值也是透過 `Value` 屬性。例如：

```
1 int? x = 10;
2
3 if (x.Value == 10) ...
4 if (x == 10) ....   // 作用同上
```

請注意，如果上述程式碼並未設定變數 `x` 的初始值，亦即令 `x` 為 `null` 的話，第 3 行程式碼將導致執行時期拋出異常 (`InvalidOperationException`)，但是第 4 行可以順利執行（其條件判斷式的結果為 `false`）。

## 判斷是否有值

欲判斷某個 `Nullable` 變數是否有值，可以用 `==` 或 `!=` 運算子，或者透過 `Nullable` 型別的 `HasValue` 屬性，例如：

```
1 int? x = null;
2
3 if (x != null) ....
4 if (x.HasValue) ....
```

順便提及，C# 有一種簡潔的邏輯判斷語法，是用兩個連續的問號，也就是 `??` 運算子。範例如下：

```
1 int? x = null;
2 int? y = x ?? 5;
```

第 2 行的作用等同於

```
int? y = (x != null) ? x : 5;
```

## 與非 Nullable 型別混用

一般實質型別的變數可以直接指定給 Nullable 變數。例如：

```
1 int i = 10;
2 int? j = i; // 隱含轉換
```

可是 Nullable 變數不能直接指定給一般實質型別的變數，例如：

```
1 int? x = 10;
2 int y = x; // 編譯錯誤！
```

第 2 行會無法通過編譯。但是如果用明確轉型：

```
int y = (int) x;
```

這樣就可以通過編譯。然而問題是，如果 `x` 是 `null`，儘管編譯成功，執行時還是會拋出異常，因為 `y` 是一般的實質型別，它無法接受 `null`。

另外，任何與 `null` 運算的結果，都會是 `null`，例如：

```
1 int? m = null;
2 int? n = 10;
3 int? o = m + n;
4 int? p = m * n;
5 Console.WriteLine("Nullable 變數運算: m + n = " + o);
6 Console.WriteLine("Nullable 變數運算: m * n = " + p);
```

執行結果為：

```
Nullable 變數運算: m + n =
Nullable 變數運算: m * n =
```

Nullable 與非 Nullable 型別可以相互運算：

```
1  int? m = 10;
2  int n = 5;
3  int? p = m * n;
4  Console.WriteLine("m * n = " p);
```

執行結果為：

m \* n = 50

但請注意運算的結果也是 `Nullable` 型別，因此本範例的變數 `p` 必須宣告為 `Nullable`。

## 1.3 物件初始設定式

(TODO)

## 1.4 自動實作屬性

```
1  public class Employee
2  {
3      public string Name // 這是一個自動實作屬性。
4      {
5          get;
6          set;
7      }
8
9      public int Age      // 這也是個自動實作屬性。
10     {
11         get;
12         set; // 若改為 private set，則為唯讀屬性，就算用物件初始設定式也無法設定此屬性。
13     }
14 }
15
16 class Program
17 {
18     static void Main(string[] args)
19     {
20         // 示範 object initializer 寫法
21         Employee emp = new Employee
22         {
23             Name = "Michael",
24             Age = 20
25         };
26     }
27 }
```

編譯器會替 `Employee` 的 `Name` 和 `Age` 屬性產生兩個私有成員，而這種由自動實作屬性所產生的成員變數有個專門的稱呼，叫做 *backing field*。此語法糖（*syntax sugar*）的方便之處即在於當我們碰到一些單純需要定義類別屬性來儲存資料的場合時，用這種語法就可以省掉自行宣告私有成員的步驟（就是少打一些字啦！）。

C# 6 對於自動實作屬性又提供了更方便的語法，這個部分請參閱稍後的〈C# 6 新語法〉一節的介紹。

## 1.5 匿名型別

有時候，例如在某個函式裡面，我們會臨時需要一個類別來儲存一些簡單資料，但又不想只為了這個臨時需要而去另外定義一個新的類別，此時便可使用 C# 的匿名型別（*anonymous type*）。

簡單地說，匿名型別讓我們可以臨時定義一個不知名的類別。參考以下範例：

```
1 private void Foo()  
2 {  
3     var emp = new { Name = "Michael", Birthday = new DateTime(1971, 1, 1) };  
4     Console.WriteLine("Employee name: " + emp.Name);  
5     Console.WriteLine("Birthday: " + emp.Birthday.ToString());  
6     Console.WriteLine(" 實際的型別是: " + emp.GetType());  
7 }
```

這個範例使用了隱含型別 `var` 來宣告區域變數 `emp`；這是必須的，因為我們使用了匿名型別。這表示從 `new` 運算子之後的大括弧包住的部分，會由編譯器產生一個類別。當然該類別的名稱也是由編譯器決定，因此我們在寫程式時便無法得知實際的類別名稱，自然也就得用 `var` 來宣告變數了。

執行結果如下：

```
Employee name: Michael  
Birthday: 1971/1/1 上午 12:00:00  
實際的型別是: <>f__AnonymousType2[System.String,System.DateTime]
```

範例程式的最後一行是把匿名型別的實際型別名稱秀出來。注意此型別是個泛型類別（*generic class*），它有兩個型別參數。

OK! 既然從程式的執行結果已經知道實際型別，那我們何不直接用這個型別名稱來宣告變數？

不行，你如果嘗試在程式中使用 `f__AnonymousType2<String, DateTime>` 來宣告變數，編譯器會看不懂。

那如果要宣告兩個相同匿名型別的變數怎麼辦？

例如，把範例程式碼改成這樣：

```
1 private void Foo()
2 {
3     var emp1 = new { Name = "Michael", Birthday = new DateTime(1971, 1, 1) };
4     var emp2 = new { Name = "John", Birthday = new DateTime(1981, 12, 31) };
5
6     Console.WriteLine("emp1 的實際型別是: " + emp1.GetType());
7     Console.WriteLine("emp2 的實際型別是: " + emp2.GetType());
8 }
```

你會發現 emp1 和 emp2 的實際型別都一樣。這是因為，基於效率考量，編譯器會重複使用具有相同參數個數與參數名稱的匿名型別，而不會每碰到一個匿名型別的宣告就產生一個新的泛型類別。

不過，如果你將 emp2 的宣告改成這樣：

```
var emp2 = new { Name = "John", Birth = new DateTime(1981, 12, 31) };
```

由於參數名稱不同，編譯器會產生另一個泛型類別。執行結果如下：

```
emp1 的實際型別是: <>f__AnonymousType0`2[System.String,System.DateTime]
emp2 的實際型別是: <>f__AnonymousType1`2[System.String,System.DateTime]
```

現在，你已經知道如何使用匿名型別和初始設定式了。接下來要介紹匿名型別的另一種定義方式：投射初始設定式（projection initializers）。

## 投射初始設定式

匿名型別還有一種投射初始設定式（projection initializers）的寫法。顧名思義，這當中會有「投射」的動作。投射什麼呢？投射將來要產生的類別的屬性。

除了像上一節範例那樣直接指定屬性名稱的寫法（var 變數名 = new { 屬性名 = 屬性值 }），你也可以透過投射既有物件屬性的方式來建立匿名型別。參考底下的範例：

```
1 public class EmpInfo
2 {
3     public string Name { get; set; }
4     public DateTime Birthday { get; set; }
5 }
6
7 class Program
8 {
9     static void Main(string[] args)
10    {
```

```
11     EmpInfo emp = new EmpInfo()  
12         { // 物件初始設定式 (前面介紹過)  
13             Name="Michael",  
14             Birthday = new DateTime(1971, 1, 1)  
15         };  
16  
17     var emp1 = new { emp.Name, emp.Birthday }; // (1) 使用物件的屬性來投射  
18  
19     string name = "John";  
20     int age = 20;  
21  
22     var emp2 = new { name, age }; // (2) 使用區域變數來投射  
23  
24     Console.WriteLine("emp1.Name = " + emp1.Name);  
25     Console.WriteLine("emp2.name = " + emp2.name);  
26  
27     Console.WriteLine("emp1 的實際型別: " + emp1.GetType());  
28     Console.WriteLine("emp2 的實際型別: " + emp2.GetType());  
29 }  
30 }
```

這裡示範了兩種投射方式。一種是利用事先定義的類別 `EmpInfo` 的物件屬性來投射，即程式註解標示 (1) 的那行；也就是說，編譯器在為匿名型別變數產生實際的類別時，會使用傳入物件的屬性名稱來定義新類別的屬性。另一種則是使用區域變數名稱，即範例中以英文小寫宣告的變數 `name` 和 `age`（註解標示 (2) 的地方）。

執行結果如下：

```
emp1.Name = Michael  
emp2.name = John  
emp1 的實際型別: <>f__AnonymousType0`2[System.String,System.DateTime]  
emp2 的實際型別: <>f__AnonymousType1`2[System.String,System.Int32]
```

## 注意事項

使用匿名型別時，請注意幾件事：

- 編譯器為匿名型別所產生的泛型類別並未實作 `IDisposable` 介面，因此，你應該避免在匿名型別中存放可拋棄的（disposable）物件。
- 一般而言，匿名型別的物件不應該儲存在 `List` 這類物件串列中。要是你這麼做的話，到時候取出物件時，你怎麼知道要將它轉成什麼型別呢？
- 承上，匿名型別的物件也不能用於呼叫方法時的參數傳遞。這是因為，如果硬要這麼做，參數型別勢必得宣告為 `object`，這當然可以通過編譯，可是在使用此參數時，還是會碰到不知該轉型成什麼型別的問題。



## 1.6 擴充方法

經常開發 .NET 應用程式的人，相信多少都會建立一些自己常用的工具類別，方便重複使用。就拿字串處理來說吧，我們可能會寫一個 `StringHelper` 類別，裡面提供一些常用的字串操作，例如字串反轉、把第一個英文字母轉成大寫等等。這個輔助工具類別通常會宣告成 `static`（即無法建立該類別的執行個體），像這樣：

```
1 public static class StringHelper
2 {
3     public static string Reverse(string s)
4     {
5         ...
6     }
7
8     public static string Capitalize(string s)
9     {
10        ...
11    }
12 }
```

那麼，如果想要把一個字串反轉之後再把第一個英文字母變成大寫，可以這麼寫：

```
1 string s1 = "abcdefg";
2 string s2 = StringHelper.Capitalize(StringHelper.Reverse(s1));
```

如果把這兩個方法改成擴充方法，剛才的程式碼可以這麼寫：

```
1 string s1 = "abcdefg";
2 string s2 = s1.Reverse().Capitalize();
```

如何？是不是更簡潔、更好理解呢？而且，程式碼看起來就像是 `String` 類別原本就有提供這兩個方法，而且用法就跟呼叫一般的物件方法（`instance method`）一樣。

那麼，要如何把靜態方法改成擴充方法呢？先來看看修改後的結果：

```
1 public static class StringHelper
2 {
3     public static string Reverse(this string s)
4     {
5         ...
6     }
7
8     public static string Capitalize(this string s)
9     {
10        ...
11    }
12 }
```

與先前的版本比較，唯一的差別只是在靜態方法的參數列中加上關鍵字 `this`，就讓它變成了擴充方法。

可是，編譯器怎麼知道我要擴充的是 `String` 類別呢？答案是：編譯器會將關鍵字 `this` 所修飾的型別視為該方法所欲擴充的類別。也就是說，當你這樣寫的時候：

```
1 public static string ToString(this DateTime aDate, char separator)
2 {
3     ...
4 }
```

意思就是要讓 `ToString` 成為 `DateTime` 類別的擴充方法，而參數列中的 `this` 即代表當前的物件。

如稍早提過的，擴充方法雖然也是宣告為 `static`，但寫法則與呼叫一般的物件方法（instance method）沒有兩樣。由於編譯器會把當前的物件傳遞給擴充方法的第一個參數（即以 `this` 修飾的參數），所以呼叫擴充方法時，實際傳入的參數會比宣告時的參數少一個。以剛才的 `ToString` 擴充方法為例，呼叫時只需傳入一個參數，例如：

```
1 DateTime.Now.ToString('/');
```

要注意的是，欲擴充的類別必須放在參數列的第一個位置。也就是說，關鍵字 `this` 只能用來修飾第一個參數。如果你將 `this` 加在第二個或之後的參數，程式將無法通過編譯。



使用擴充方法的程式碼不會增加執行時期的效能負擔，因為它是在編譯時期就已經被編譯器處理掉了。如果用反組譯工具查看編譯出來的 IL 代碼，你就會看到，呼叫擴充方法的程式碼其實還是編譯成靜態方法呼叫。



在 Visual Studio 2013 裡面，如果你知道擴充方法的名稱，寫程式時直接呼叫，卻沒有先匯入（using）該擴充方法所屬的命名空間，此時 Visual Studio 編輯器不會有任何輔助提示。到了 Visual Studio 2015 則會提示你可能需要引用哪個命名空間。

## 1.7 C# 6 新增的語法



撰寫本文時，Visual Studio 2015 仍為 Preview 版本，但 C# 語言的部分應該已經蠻穩定了，將來不會有太大變動。目前發現原本微軟打算加入的兩個新語法已確定拿掉：primary constructor 和 declaration expressions。

### nameof 表示式

C# 6 新增的 `nameof` 關鍵字可用來取得某變數的名稱。請看底下這個簡單範例，便能了解其作用：

```
1 string firstName = "Joey";
2 string varName = nameof(firstName); // 編譯結果: varName = "firstName"
```

程式碼右邊的註解已經透露，`nameof` 是作用於編譯時期，而非執行時期。

那麼，它可以用在哪裡呢？

比如說，為了防錯，我們經常在函式中預先檢查參數值是否合法，並將不合法的參數透過拋異常（exception）的方式通知呼叫端（或者寫入 log 檔案以供將來診斷問題）。像這樣：

```
1 void LoadProduct(string productId)
2 {
3     if (productId == null)
4     {
5         throw new ArgumentNullException("productId");
6     }
7     ....
8 }
```

如此一來，當程式出錯時，用戶端就能輕易知道是哪個參數不對。問題是，程式中的參數名稱是寫死的字串值（"productId"），將來萬一要修改參數名稱，稍不注意就會漏改這些字串。於是，即使煞費周章，仍有人選擇在執行時期動態取得參數名稱，只為了避免在程式中寫一堆固定的字串。其實，參數名稱在編譯時期就已經決定了，何不由編譯器代勞，既省事又不犧牲執行效率？

現在，C# 6 提供了 `nameof` 表示式，正好可以解決這個問題。於是剛才的範例可以改寫成這樣：

```
1 void LoadProduct(string productId)
2 {
3     if (productId == null)
4     {
5         throw new ArgumentNullException(nameof(productId));
6     }
7     ....
8 }
```

其中的 `nameof(productId)` 表示式在經過編譯之後，結果就等於手寫的固定字串 `"productId"`。

## 字串插值

.NET Framework 提供的字串格式化方法 `String.Format(...)` 是一種對號入座的寫法，相當好用。現在，C# 6 提供了另一種格式化字串的寫法，名曰「字串插值」（string interpolation）。字串插值的基本語法，是在雙引號包住的字串前面加上一個 `'$'` 字元，而在字串內容的部分使用一對大括弧 `{}` 來包住一個變數名稱。如下所示：

`$"{變數名稱: 格式規範}"`

其中的「格式規範」就跟 `String.Format()` 方法所使用的格式規範字元一樣。一個格式字串裡面可以有多對大括弧，用來帶入不同的變數值，而沒有被大括弧包住的部分則維持不變。

參考底下的範例：

```
1 string firstName = "Michael";
2 string lastName = "Tsai";
3 int salary = 22000;
4
5 string msg1 = String.Format("{0} {1} 的月薪是 {2:C0}", firstName, lastName, salary);
6 string msg2 = $"{firstName} {lastName} 的月薪是 {salary:C0}";
7 Console.WriteLine(msg1);
8 Console.WriteLine(msg2);
```

結果兩次輸出的字串值都相同，如下所示：

```
Michael Tsai 的月薪是 22,000
Michael Tsai 的月薪是 22,000
```

跟既有的 `String.Format()` 方法比較，我覺得字串插值的寫法更容易解讀，更容易想像最終格式化的結果。原因在於，解讀 `String.Format()` 時，我們必須把右邊的參數依序帶入左邊的格式字串；如果參數很多，有時還會對不上，導致順序錯置。新的字串插值語法則是直接在格式字串裡面填入變數名稱，不僅直觀，而且不會有弄錯順序的問題。

字串插值跟 `nameof` 表示式一樣都是語法糖，是編譯時期的魔法。明白地說，編譯器會把字串插值的語法編譯成 `String.Format()` 方法呼叫。底下是更多字串插值的範例，右邊註解則是編譯後的結果。

```
1 $" 姓名 = {myName}" // String.Format(" 姓名 = {0}", myName)
2 $" 小時 = {DateTime.Now:hh}" // String.Format(" 小時 = {0:hh}", DateTime.Now)
3 $"{{測試大括弧}}}" // String.Format("{{測試大括弧}}}")
4 $"{{秒 = {DateTime.Now :ss}}}" // String.Format("{{秒 = {0:ss}}}", DateTime.Now)
```

需特別注意的是，兩個連續的大括弧代表欲輸出一個大括弧字元。故第三個例子的執行結果為“{測試大括弧}”。

## Null 條件運算子

保險起見，在需要存取某物件的屬性之前，我們通常會先檢查該物件是否為 `null`，以免程式執行時拋出異常 (`NullReferenceException`)。一般常見的寫法如下：

```
1 static void NullPropagationDemo(string s)
2 {
3     if (s != null && s.Length == 4) // 只有當 s 不為空時才存取它的 Length 屬性。
4     {
5         // Do something.
6     }
7 }
```

C# 6 新增了 `null` 條件運算子語法，讓你用更簡短的語法達到同樣效果：先判斷物件是否為 `null`，不是 `null` 才會存取其成員。它的寫法是在變數後面跟著一個問號，然後是存取成員名稱的表示式。參考以下範例：

```
1 static void NullPropagationDemo(string s)
2 {
3     if (s?.Length == 4) // 只有當 s 不為空時才存取它的 Length 屬性。
4     {
5         // Do something.
6     }
7 }
```

更多範例：

```
1 int? length = productList?.Length; // 若 productList 是 null, 則 length 也是 null。
2 Customer obj = productList?[0];    // 若 productList 是 null, 則 obj 也是 null。
3 int length = productList?.Length ?? 0; // 若 productList 是 null, 則 length 是 0。
4 string name = productList?[0].FullName; // 若 productList 是 null, 則 name 是 null。
```

## 自動屬性的初始設定式

在 C# 6 之前，唯讀屬性的 `get` 方法必須有方法本體，所以通常會有一個與之對應的私有欄位（稱為屬性的 backing field）。範例如下：

```
1 class BeforCSharp6
2 {
3     private DateTime _startupTime = DateTime.Now;
4
5     public DateTime StartupTime
6     {
7         get { return _startupTime; }
8     }
9 }
```

C# 6 之後，可使用新的「自動屬性初始設定式」（auto-property initializers）來進一步簡化：

```
1 class NewInCSharp6
2 {
3     public DateTime StartupTime { get; } = DateTime.Now;
4 }
```

明顯的好處是，自動實作屬性有了初始設定式之後，編譯器不會再強迫你一定要替唯讀屬性定義一個對應的私有欄位。此新增語法的主要特色如下：

- 自動實作屬性的 `get` 方法可以不寫方法本體。
- 自動實作屬性之後可接著用 `=` 運算子來加上賦值敘述，以設定該屬性的初始值。此新增的賦值語法只能用於自動實作屬性；若用在一般的屬性，編譯時會報錯。



本書會交替使用「自動實作屬性」和「自動屬性」，兩者意思相同，指的都是 `automatically implemented property` 語法。

另一個可賦予唯讀屬性初始值的地方是建構函式。這個部分也稍有變化。

C# 6 之前：

```
1 class BeforCSharp6
2 {
3     private DateTime _startupTime;
4
5     public DateTime StartupTime
6     {
7         get { return _startupTime; }
8     }
9
10    public BeforCSharp6(DateTime time)
11    {
12        _startupTime = time; // 若寫成 this.StartupTime = time; 會無法通過編譯。
13    }
14 }
```

C# 6 之後:

```
1 class NewInCSharp6
2 {
3     public DateTime StartupTime { get; } // C# 6 OK。之前的版本會出現編譯錯誤。
4
5     public NewInCSharp6(DateTime time)
6     {
7         this.StartupTime = time; // C# 6 OK。之前的版本會出現編譯錯誤。
8     }
9 }
```

如果你在寫程式時經常覺得替一堆屬性定義對應的私有欄位很麻煩，希望能夠盡量使用自動實作屬性，那麼你應該會蠻喜歡這個新語法。

底下是更多自動實作屬性的寫法，有的是正確語法，有的則為錯誤示範（右方註解會註明是否能夠通過編譯）：

```
1 class NewInCSharp6
2 {
3     public DateTime StartupTime { get; } = DateTime.Now; // 編譯 OK。
4     public string FullName { get; set; } = "Michael Tsai"; // 編譯 OK。
5     public int Age { get; } = this.GetAge(); // 編譯錯誤：初始設定式不可以呼叫非靜態方法！
6     public Encoding encoding { get; } = Encoding.GetEncoding("BIG5"); // 編譯 OK。
7 }
```

此範例程式碼的第 5 行之所以無法通過編譯，理由已在註解中說明。第 6 行雖然也是利用方法呼叫來設定自動屬性的初值，但它是使用靜態方法，所以能夠通過編譯。

## 以表示式為本體的成員

「以表示式為本體的成員」好像太拗口了？英文是 `expression-bodied members`。意思是：以表示式來作為成員本體。我想，還是直接看程式碼比較清楚吧。

C# 6 之前：

```
1  class BeforeCSharp6
2  {
3      private DateTime startTime = DateTime.Now;
4
5      public int ElapsedSeconds
6      {
7          get
8          {
9              return (DateTime.Now - startTime).Seconds;
10         }
11     }
12 }
```

此範例中的唯讀屬性 `ElapsedSeconds` 的值是動態計算出來的，代表從 `startTime` 開始之後到目前為止經過了幾秒鐘。

C# 6 可以這麼寫：

```
1  class NewInCsharp6
2  {
3      private DateTime startTime = DateTime.Now;
4
5      public int ElapsedSeconds => (DateTime.Now - startTime).Seconds;
6  }
```

這裡使用了 C# 6 新增的語法：以表示式作為成員本體。如你所見，程式碼變得更簡短了。原本的唯讀屬性 `ElapsedSeconds` 的 `get` 方法本體不見了，取而代之的是屬性名稱後面跟著一個以 `=>` 符號開始的 `lambda` 表示式——其實骨子裡，編譯器會把它編譯成一個 `get` 方法，就跟先前的 `BeforeCSharp6` 類別裡面的寫法一樣。

要注意的是，這個表示式只能有一行敘述，而不能包含多行程式碼的區塊。因此，你不能這麼寫：



```
1     public int ElapsedSeconds =>
2     {
3         return (DateTime.Now - startTime).Seconds; // 編譯失敗！
4     }
```

此外，「以表示式為本體的成員」可以是屬性或方法，而且有沒有傳回值都行。底下是更多範例：

```
1     public string FullName => this.FirstName + " " + this.LastName;
2     public Color GetColor(int r, int g, int b) => Color.FromArgb(r, g, b);
3     public void Log(string msg) => Console.WriteLine(msg);
4     public static string HowCold(int temperature) =>
5         temperature > 17 ? " 不冷" : " 挺冷";
6     public Employee this[int id] => this.FindEmployee(id); // 索引子也可以用此語法。
```

顯然地，對於那些只包含一行程式碼的屬性或方法，若使用「以表示式為本體的成員」語法來撰寫，程式碼會再稍微簡短一些。

## using static 陳述式

在 C# 6 之前，只要有匯入（using）某類別所屬的命名空間，便可使用「類別名稱. 成員名稱」的語法來存取該類別的靜態成員。例如：

```
1 using System;           // 匯入 System 命名空間（的所有型別）
2 using System.IO;        // 匯入 System.IO 命名空間（的所有型別）
3
4 class BeforeCSharp6
5 {
6     void Demo()
7     {
8         Console.WriteLine("Hello!");
9         File.WriteAllText(@"C:\test.txt", "Hello!");
10    }
11 }
```

到了 C# 6，新增的 using static 陳述式可以讓存取靜態成員的寫法更簡潔。它的語法是：

```
using static 某型別名稱;
```

關鍵字 using static 本身已經揭示，其目的是要讓你更方便地「使用靜態成員」。說得更明白些，也就是匯入某型別的所有靜態成員，以便在目前的程式檔案中直接使用那些靜態成員，而無須指涉型別名稱。

於是，剛才的範例可以改寫成這樣：

```
1 using static System.Console; // 匯入 System.Console 類別的所有靜態成員
2 using static System.IO.File; // 匯入 System.IO.File 類別的所有靜態成員
3
4 class NewInCSharp6
5 {
6     void Demo()
7     {
8         WriteLine("Hello!"); // 省略類別名稱 "Console"
9         WriteAllText(@"C:\test.txt", "Hello!"); // 省略類別名稱 "File"
10    }
11 }
```

如你所見，當你需要在程式中大量存取某類別的靜態成員時，`using static` 可以讓你少打一些字。

不過，這個新口味的語法糖恐怕不見得人人都愛。怎麼說呢？

譬如 `WriteLine` 這類常見的方法，由於不只一個類別提供此方法，使用時可能會產生名稱衝突而導致編譯失敗。另一個可能的問題是，如果大量使用 `using static` 來匯入許多類別的靜態成員，雖然寫程式的時候可以少打一些字，但將來閱讀程式碼的時候可能會經常有疑問：這個靜態方法（或屬性）到底是屬於哪個型別？

然而在某些場合，省略類別名稱的寫法卻是同時兼具簡潔與易讀的優點。例如 `Math` 類別。試比照底下兩行程式碼的寫法：

```
1 var value1 = Math.Sqrt(Math.Sin(90) + Math.Cos(45)); // using System
2 var value2 = Sqrt(Sin(90) + Cos(45)); // using static System.Math
```

第一行程式碼是一般的靜態方法呼叫。第二行程式碼則是使用了 `using static` 的寫法。就我的感覺，程式碼不僅更簡潔，而且不影響程式碼的理解，甚至還更容易閱讀了。



感覺是主觀的，但我想只要試寫個幾次，應該就能大概抓到一個原則：當靜態成員的名稱非常明確，一看就知道它的功用與所屬型別，而且不會有任何猶疑。有這種感覺，八成是用對地方了。

順便提及，由於 `using static` 作用的對象是型別（而不是命名空間），因此它還可以用來限定只匯入某類別的擴充方法（`extension methods`），而不是像一般的 `using` 陳述式那樣匯入某命名空間的全部擴充方法。當你在 C# 程式檔案中匯入了許多命名空間，而其中有三個命名空間碰巧包含了相同名稱的擴充方法，此時便可藉由 `using static` 匯入指定類別的方式來嘗試解決名稱衝突的問題。

（未完待續）

## 2. 泛型

布魯斯：「羅賓，為什麼你的程式裡面有那麼多自訂的串列類別？像是 `EmployeeList`、`CustomerList`、`VendorList` 這些，它們有什麼特殊用途嗎？」

羅賓：「喔，沒有啦，因為我需要處理大量的員工、客戶、和廠商資料，如果用 `ArrayList` 來存放這些物件……你也知道，這樣就得寫一大堆型別轉換的程式碼。為了避免手動轉型，我就分別為員工、廠商、和客戶，個別定義其專屬的串列類別。雖然會多出三個自訂類別，剛開始要花點時間寫，可是一旦這些串列類別寫好了，在使用這些類別的時候就有編譯時期型別安全的好處，而且不用手動轉型，程式碼也更簡潔了。」

布魯斯：「嗯，我想你說的沒錯。但…你有考慮過『泛型』嗎？」

### 2.1 為什麼要有泛型？

在解釋何謂泛型（generic types）之前，且讓我們先了解為什麼要有泛型，以及如果沒有泛型，對我們平日的程式開發工作有何不便之處。

假設我們有一堆整數要處理，但由於無法事先知道究竟有多少個整數，或者程式執行過程中可能會動態增加或減少其數量，因此我們不能用整數陣列（陣列大小是在宣告陣列時就固定下來的），而得用集合來處理。在沒有泛型的情況下，我們可能會選擇使用 `ArrayList`。請看底下這個程式範例：

```
1 ArrayList<int> intList = new ArrayList<>();
2 intList.Add(100);
3 intList.Add(200);
4
5 int i1 = intList[1];           // 無法通過編譯，需要明確轉型。
6 int i2 = (int)intList[2];     // 編譯 OK!
```

從這個範例當中，不難看出 `ArrayList` 的問題：每當你從 `ArrayList` 集合物件中取出元素時，都必須手動轉型（或者說，明確轉型）。這是因為當你將某個物件存入 `ArrayList` 集合時，它的型別就已經隱含轉換成 `Object` 了。不只 `ArrayList`，還有 `Hashtable`、`SortedList`、`Stack` 等等，只要是 `System.Collections` 命名空間裡的集合類別都是如此。

#### 手動轉型的問題

「手動轉型有什麼關係呢？只是多打一點字而已，不是嗎？」也許有人會這麼質疑。手動轉型的確不是什麼嚴重的問題；有時候，我們會覺得它是必要之惡——至少在還沒有泛型可

用的時候是如此。在沒有其他選擇的情況下，我們願意多打一些字，使用明確轉型的語法。但這只是其中一個問題。

另一個小問題，是對效能的影響。在處理集合的時候，常常都會需要一個迴圈來取出或存入集合元素。如果這個迴圈的數量很大，那些手動轉型的動作就會反覆執行很多次。當然啦，每一次轉型所需要花費的時間並不多，使用者根本感覺不出來。只有當次數很頻繁時，對程式的執行速度才會產生些微的影響。

不過，手動轉型還有一個比較大的問題：執行時期的型別安全轉換。延續前面的例子，如果取出元素的時候這樣寫：

```
DateTime dt = (DateTime)intList[1]; // 編譯 OK! 執行時會出錯!
```

雖然可以通過編譯，執行時卻會出錯，因為 `int` 無法轉型為 `DateTime` —— 這兩種型別是不相容的。如果是取出元素時轉型為 `long` 就不會出錯，因為 `int` 與 `long` 都是整數，可互相轉換。也就是說，當你使用手動轉型時，萬一不小心把欲轉換的目標型別寫錯了，寫成與來源物件不相容的型別，這種錯誤，編譯器是檢查不出來的。型別之間的轉換能否順利執行，責任完全在你（開發人員）身上。

好，現在我們已經知道，手動轉型的寫法有三個問題：你得多打一些字、影響執行速度（雖然很輕微）、以及——最重要的——缺乏編譯時期的型別安全檢查。那麼，如何改善呢？

## 土法煉鋼

在沒有泛型的情況下，解決前述轉型問題的一個方法，就是如本章開頭引言中，羅賓的作法：為每一種需要儲存的元素類型設計專屬的集合類別。舉例來說，若需要存放一堆整數的整數串列，我們可能會寫個 `IntList` 類別，像這樣：

```
1 public class IntList
2 {
3     private ArrayList _numbers = new ArrayList();
4
5     public void Add(int num)
6     {
7         _numbers.Add(num);
8     }
9
10    public int this[int index]    // 索引子 (indexer)
11    {
12        get
13        {
14            return (int) _numbers[index];
15        }
16        set
```

```
17     {
18         _numbers[index] = value;
19     }
20 }
21 }
```

然後，為了處理字串串列，又要寫一個字串串列類別。這時候，聰明的程式設計師當然會選擇最省力的作法：將已經寫好的 `IntList` 類別的程式碼複製成另一份，然後把類別名稱改為 `StringList`，並將串列所儲存之元素的型別從 `int` 改為 `string`。結果如下（以黃色背景標示的部分即為替換掉的元素型別）：

```
1  public class StringList
2  {
3      private ArrayList _strings = new ArrayList();
4
5      public void Add(string s)
6      {
7          _strings.Add(s);
8      }
9
10     public string this[int index]
11     {
12         get
13         {
14             return (string) _strings[index];
15         }
16         set
17         {
18             _strings[index] = value;
19         }
20     }
21 }
```

最後，我們可能會寫了一堆大同小異的類別，例如：`IntList`、`StringList`、`StudentList`、`EmployeeList`...等等。這一切的努力，只是為了存取集合元素時，不用手動轉型、不會有執行時期型別轉換失敗的意外。使用這些自訂的集合類別時，程式碼寫起來的確比較乾淨，像這樣：

```
1    IntList intList = new IntList();
2    intList.Add(100);
3    intList.Add(200);
4
5    int num = intList[1];
6
7    StringList strList = new StringList();
8    strList.Add(" 王曉明");
9    strList.Add(" 李大同");
10
11   string s = strList[1];
```

但我們不禁要問，為了避免手動轉型而額外撰寫一堆類別，值得嗎？這種複製一份再修改的方式，勢必會產生許多重複的程式碼，以至於將來萬一需要為串列增加新的方法或屬性時，必須逐一修改這些類別。你會發現，以「複製再修改」的方式來產生新類別的時候很容易，可是將來維護的時候卻很麻煩，而且也會增加測試成本。如此說來，似乎就只好選擇手動轉型，並承受執行時期轉型失敗的風險了。

難道沒有兩全其美的辦法嗎？也就是說，既不用自己寫一堆自訂的集合類別（我們希望只寫一次就好），同時又享有編譯時期型別安全的檢查。還好，我們有泛型。



C# 是從 2.0 版開始支援泛型。

## 泛型之美

延續上一節的 `IntList` 和 `StringList` 的例子。如果用泛型來解，會省力、簡潔許多。我們可以選擇 .NET Framework 提供的泛型集合：`List`

暫且別管泛型語法，以及參數化型別到底是什麼。先來看看，把前面的範例用泛型集合改寫之後，會是什麼樣子：

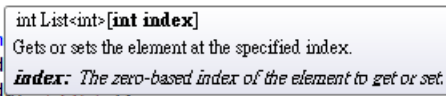
```
1    List<int> intList = new List<int>();
2    intList.Add(100);
3    intList.Add(200);
4
5    int num = intList[1];
6
7    List<string> strList = new List<string>();
8    strList.Add(" 王曉明");
9    strList.Add(" 李大同");
10
11   string s = strList[1];
```

你可以看到，我們不再需要手動轉型，也不用再另外寫兩個 `IntList` 和 `StringList` 類別。而且，由於編譯時期就已經知道集合元素的型別，Visual Studio 的 IntelliSense 功能也能發揮作用，在存取集合元素的時候會提示我們實際的元素型別，如下圖所示。

```
List<int> intList = new List<int>();
intList.Add(100);
intList.Add(200);

int num = intList[h];

List<string> strList = new List<string>();
strList.Add("100");
strList.Add("200");
```



The tooltip shows the signature `int List<int>[int index]` and the description: "Gets or sets the element at the specified index. **index**: The zero-based index of the element to get or set."

了解泛型的基本用法和優點之後，現在我們可以比較正式的來介紹泛型語法了。

## 2.2 細說泛型

若從「泛型」這個中文詞彙來解釋，可以說是「可廣泛應用的型別」。為什麼說廣泛應用呢？因為你只需要定義一個型別，就可以套用在各種類似的情境或用途。簡言之：寫一次，重複套用。

拿前面的例子來說，當我們的應用程式需要各種串列，例如整數串列、字串串列、員工串列、客戶串列……等等，既然它們都是串列，且都具有串列的共同屬性和行為，那麼最理想的做法，通常是寫一個通用的串列類別，然後套用到任何需要使用串列的場合。當然，它還得具備編譯時期的型別安全檢查才行。

剛才舉例的那些串列類別，彼此共同的、不變的特徵就是串列，而唯一的差異則在於串列裡面所儲存的那些元素的型別，有的是字串，有的是整數、員工、客戶…等等。泛型可以讓你將這些變動的部分抽離出來，定義成參數。如此一來，固定不變的部分就只需要寫一次，其餘變動的部分，則以帶入參數的方式套進去。很符合經濟效益，不是嗎？

接著就來看看，如何以泛型語法來定義我們自己的泛型類別。

### 泛型、型別參數、建構的型別

泛型可用於類別、介面、委派、和方法。這裡要先介紹的是宣告泛型類別的基本語法：

```
class 類別名稱 <T1, T2, ..., Tn>
{
    ...
}
```

其中以角括弧 `< >` 符號包住的 `T1, T2, ..., Tn` 即是所謂的「型別參數」(type parameters)，也就是先前提過的「變動的部分」。型別參數可有一至多個，端看實際的需要而定。慣例

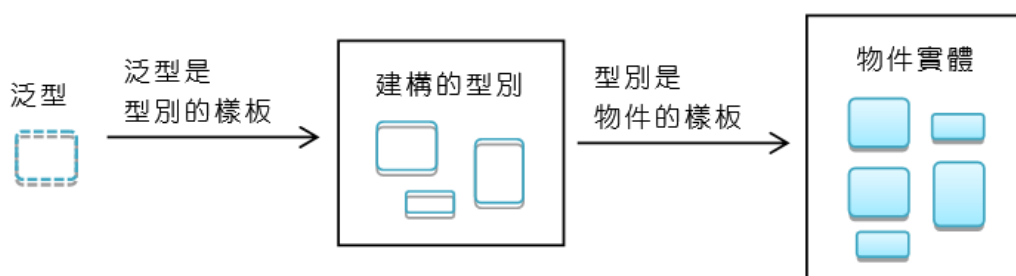


上，型別參數的名稱通常以一個大寫英文字母 ‘T’ 開頭，如果只有一個型別參數，通常就單純命名為 “T”，例如：List<T>。若有多個型別參數，則建議使用望文生義的名稱，以便識別。像 .NET 的泛型集合 Dictionary<TKey, TValue> 就很清楚，一看就知道這個泛型集合所要儲存的每一筆資料都包含兩種元素：key（索引鍵）與 value（資料值），而這兩種元素的真正型別，則是等到實際要使用集合物件的時候才以參數的方式帶入。

請注意「類別名稱

## 泛型是型別的樣板

在學習物件導向程式語言的時候，有一個很常見的譬喻：類別就像是一張藍圖，而物件則是根據該藍圖所建立出來的實體（instance）；只要先把藍圖設計好，就可以在程式中利用該藍圖來建立多個物件實體。同樣的，泛型就是類別的藍圖（樣板），根據此藍圖，便能夠產生多種建構的型別；然後，再利用這些建構的型別來建立物件實體。下圖描繪了泛型、建構的型別、與物件實體這三者之間的關係。



泛型、建構的型別、與物件實體的關係

雖然您已經知道 .NET Framework 已經有提供現成的泛型串列，現在假設您因為某些特殊原因需要設計自己的泛型串列，並將它命名 MyGenericList。程式碼如下：

```
1 public class MyGenericList<T>
2 {
3     private ArrayList _elements = new ArrayList();
4
5     public void Add(T item)
6     {
7         _elements.Add(item);
8     }
9
10
11     public T this[int index]    // 索引子 (indexer)
12     {
13         get
14         {
```



```

15         return (T) _elements[index];
16     }
17     set
18     {
19         _elements[index] = value;
20     }
21 }
22 }

```

其中的 `T` 即為型別參數。您不妨與前面的 `IntList` 和 `StringList` 類別的程式碼對照一下兩邊的差異。你會看到，只要是需要用到型別參數的部分，都是以型別參數名稱 `T` 來表示。等到真正需要使用「建構的型別」(constructed type) 時，編譯器便會以外部指定的型別帶入參數 `T`，以產生真正的類別。完整起見，底下再列出實際建立物件時的程式範例：

```

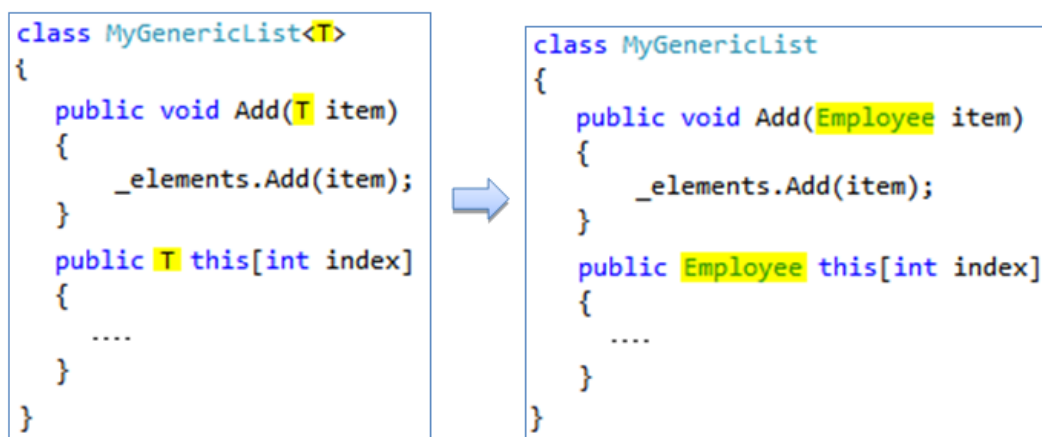
1  MyGenericList<int> intList = new MyGenericList<int>();
2  intList.Add(100);
3  int num = intList[0];
4
5  MyGenericList<Employee> empList = new MyGenericList<Employee>();
6  empList.Add(new Employee("A001", " 王曉明"));
7  empList.Add(new Employee("A002", " 李大同"));
8
9  Employee anEmp = empList[1];
10 ....

```

此例中，根據泛型 `MyGenericList<T>` 所建構的型別共有兩個，分別是 `MyGenericList<int>` 和 `MyGenericList<Employee>`，而程式便是利用這兩個建構的型別來各自建立不同的物件實體。下圖所示即為編譯器根據我們定義的泛型來產生建構型別的示意圖。圖的左方為泛型類別的宣告，右方為編譯器所產生的建構型別。當編譯器看到 `MyGenericList<Employee>` 時，就會將型別 `Employee` 帶入泛型的型別參數 `T`，從而產生一個新的型別（即建構的型別），如圖中右邊的區塊所示。



這裡的「編譯器」指的是 .NET 即時編譯器 (just-in-time compiler)，而非 C# 編譯器；這個部分會在稍後的〈泛型與 C++ 樣板的差異〉一節中進一步說明。



編譯器根據泛型的宣告（左邊）來產生建構的型別（右邊）

在了解宣告泛型的基本語法，以及如何使用泛型來建立物件之後，讓我們來看一個稍微複雜一點的例子：

```

1 public class MyGenericList<T>
2 {
3     ....
4 }
5
6 public class MyStackList<T>
7 {
8     private MyGenericList< Stack<T> > stkList;
9 }

```

在這個例子當中，`MyStackList<T>` 的成員 `stkList` 使用了層層套疊的泛型宣告，也就是利用既有的泛型來產生另一個開放型別。我們可以這麼解讀：`stkList` 是一個串列，此串列中的每一個元素的型別都是 `Stack<T>`，而這個型別 `T` 則是由外界傳入。也就是說，`stkList` 是一個堆疊串列（串列中的每個元素都是堆疊），而這些堆疊裡面存放的是什麼東西呢？由外界透過型別參數 `T` 來決定。

#### 泛型類別中的靜態欄位

現在你已經知道，泛型實際上會被展開成建構的型別，然後程式是用那些建構的型別來建立物件實體。那麼，你應該可以想像得到，如果泛型類別裡面包含大量的靜態欄位，程式在執行時可能會占用較多記憶體，因為每一個建構型別就會有一份自己專屬的靜態欄位。而且，由於 .NET 組件的型別資訊一旦載入就會一直存在記憶體中，直到程式結束（app domain 卸載）才會釋放，故那些靜態欄位也會一直存在記憶體中。

## 建構式與解構式

我們知道，C# 類別的建構式（constructor）和解構式（finalizer）的名稱必須跟類別名稱相同。不過，泛型的建構式和解構式的名稱並不需要加上型別參數。因此，上一節的 `MyGenericList<T>` 的建構式和解構式看起來會像這樣：

```
1 public class MyGenericList<T>
2 {
3     private ArrayList _elements;
4
5     // 建構式
6     public MyGenericList()
7     {
8         _elements = new ArrayList();
9     }
10
11    // 解構式
12    ~MyGenericList()
13    {
14    }
15 }
```

如果將建構式寫成 `MyGenericList<T>() { ...}`，反而會無法通過編譯。

## 預設值的表示法

由於型別參數 `T` 代表外界可能傳入的任何型別，因此，當我們在撰寫泛型類別的時候，如果要將某個泛型參數的物件變數設定為該型別的初始值，就不能固定寫成特定數值，例如 `0`、`null`、或 `false` 等等。此時應該要用 `default` 運算子來取得該未知型別的預設值。參考以下程式片段：

```
1 public class MyGenericList<T> : IList<T>
2 {
3     private ArrayList _elements;
4
5     public void GenerateItems(int numItems)
6     {
7         T[] buf = new T[numItems]; // 建立 numItems 個 T 物件的陣列
8         for (int i = 0; i < numItems; i++)
9         {
10             buf[i] = default(T); // 設定每個元素的初始值
11         }
12     }
13 }
```



設計泛型時，由於還不知道元素的實際型別，當然也就不知道初始值要設定成什麼。此時 `default` 運算子便可派上用場。

## 型別參數的條件約束

到目前為止所舉的例子，都是一些用來儲存特定型別資料的泛型類別。如果你在設計泛型的時候，也只是單純的需要將程式碼當中的型別參數替換成外界指定的型別，那麼目前所了解的語法就已經夠用了。不過，有時候我們會需要在我們的泛型類別裡面直接存取那些參數型別的方法或屬性，這時候就會有問題了。舉例來說，假設我們要設計一個泛型串列，此類別要提供一個方法，用來比較兩個物件的內容。若物件 1 小於物件 2，便傳回負數，大於則傳回正數，相等則傳回 0。那麼，如何比較兩個物件呢？我們預期這些物件都有 `Compare` 方法，作為比較之用。此泛型串列的程式碼如下：

```
1 public class MyList<T>
2 {
3     public int Compare(T obj1, T obj2)
4     {
5         return obj1.Compare(obj2); // 編譯失敗：型別 T 沒有 Compare 方法!
6     }
7 }
```

這段程式碼在編譯時會出錯：

‘T’ does not contain a definition for ‘Compare’ and no extension method ‘Compare’ accepting a first argument of type ‘T’ could be found (are you missing a using directive or an assembly reference?)

你應該很容易就能看出來，為什麼這段程式碼會無法通過編譯：型別 `T` 根本就沒有 `Compare` 方法。就算你在使用此泛型時所傳入的型別真的有 `Compare` 方法，編譯器也照樣報錯。這是因為在沒有提供型別參數 `T` 的額外資訊的情況下，編譯器必須假設這個型別 `T` 可能是任何型別。在 .NET 裡面，任何東西都是物件，亦即任何型別一定都是繼承自 `System.Object`，因此對編譯器來說，這個 `T` 就等同於 `Object`。也就是說，當你要在泛型類別裡面存取 `T` 的物件實體的成員時，就只有 `Object` 的成員可以用，例如 `Equals()`、`GetType()`、`ToString()`。

這會有個問題，亦即先前提到的，有時我們還是會需要在泛型類別裡面直接存取那些參數型別的特定方法或屬性。這個時候，請注意不要用底下這種明確轉型的方式來解決：

```
1 public class MyList<T>
2 {
3     public int Compare(T obj1, T obj2)
4     {
5         return (obj1 as IComparable<T>).Compare(obj2);
6     }
7 }
```

這樣雖然可以繞過編譯器的型別安全檢查，卻可能導致執行時期發生轉型失敗的錯誤。

那麼，正確的解法是什麼呢？我們該如何告訴編譯器，型別參數 `T` 一定會有特定成員呢？

答案是泛型語法的 `where` 子句。

以剛才的例子來說，我們可以先定義一個泛型介面：`IComparable<T>`，並且在此介面中定義一個 `Compare` 方法。程式碼如下：

```
1 public interface IComparable<T>
2 {
3     int Compare(T obj);
4 }
```

接著在定義泛型類別的時候，用 `where` 子句來限定可帶入的型別：

```
1 public class MyList<T> where T: IComparable<T>
2 {
3     public int Compare(T obj1, T obj2)
4     {
5         return obj1.Compare(obj2);
6     }
7 }
```

關鍵之處在第一行後面增加的 `where` 子句，其作用是告訴編譯器：在使用 `MyList<T>` 時，外界所提供的型別 `T` 必須要實作 `IComparable<T>` 介面，否則便無法通過編譯。如此一來，編譯器就能夠確定這個 `T` 一定有實作該介面所定義的成員，自然也就允許我們在此泛型類別中使用那些成員了。

由於此 `where` 子句可限制外界所能夠傳入泛型的型別參數，因此這項語言機制有個比較正式的稱呼，叫做「泛型約束」(**generic constraints**)。剛才的範例所展示的條件約束語法是屬於比較單純的情況，你也可以限定型別參數必須實作多個介面，或者繼承自某個類別。請注意如果要限定類別繼承，就只能寫一個類別，因為 `C#` 並不支援多重繼承。底下是個稍微複雜的例子：

```
1 public class MyKey
2 {
3     ...
4 }
5
6 public class MyPair<TKey, TValue>
7     where TKey: MyKey, IComparable<TKey>, IEquatable<TKey>
8     where TValue: IComparable<TKey>
9 {
10     ...
11 }
```

上面這段程式碼可以這麼解讀：泛型 `MyPair<TKey, TValue>` 的型別參數 `TKey` 必須繼承自 `MyKey` 類別，且實作 `IComparable<TKey>` 與 `IEquatable<TKey>` 介面；型別參數 `TValue` 只要實作 `IComparable<TKey>` 介面即可。從這個範例當中也可以看到，`where` 子句是可以寫在獨立一行的，不必非得緊跟在類別名稱後面。

除了明白指定特定的類別或介面，泛型約束語法當中還可以使用兩個比較特殊的關鍵字：`class` 和 `struct`。當你在 `where` 子句中使用這兩個關鍵字時，就表示型別參數必須是個類別（即參考型別 [reference type]）或結構（即實質型別 [value type]），但不限定哪一個類別或哪一種結構。而且，`class` 和 `struct` 這兩個關鍵字一定要寫在條件約束子句的第一個位置。參考以下範例：

```
1 public class MyPair<TKey, TValue>
2     where TKey: class, IComparable<TKey>, IEquatable<TKey>
3     where TValue: struct, IEquatable<TKey>
4 {
5     ...
6 }
```

最後一個要介紹的泛型約束語法是 `new()`。它的用途是限定外界傳入的型別必須提供預設建構式（**default constructor**）。如果你在泛型類別裡面需要用 `new` 運算子來建立型別參數的物件實體，就必須在 `where` 子句中加入關鍵字 `new()`。像這樣：

```
1 public class MyPair<TKey, TValue>
2     where TKey: IComparable<TKey>, new()
3     where TValue: struct, new()    // 這行編譯不會過！
4 {
5     ...
6 }
```

注意第三行 `TValue` 的條件約束 `new()` 不能與 `struct` 關鍵字同時出現，否則編譯會出錯。由於 `struct` 是實質型別（value type），而實質型別會有系統自動產生的預設建構式，故編譯器不允許此二者同時出現。

## 泛型介面與結構

除了泛型類別，我們還可以利用泛型語法來定義泛型介面與泛型結構。比如說，假設我們想要把 `MyGenericList<T>` 類別中有關串列的基本操作和屬性抽離出來，定義成一個串列介面，這個串列介面的程式碼可能會像這樣：

```
1 public interface IList<T>
2 {
3     int Count { get; }
4     T this[int index]
5     {
6         get;
7         set;
8     }
9 }
```

此介面定義了兩個屬性：一個唯讀屬性 `count`，用來傳回串列的元素數量；以及一個索引子（`indexer`），用來取得或設定某個元素的內容。因此，若要讓 `MyGenericList<T>` 類別實作此介面，自然也就必須提供這兩個屬性的實作程式碼。參考底下的程式片段，其中的索引子是原本就已經寫好的，這次只是增加了唯讀屬性 `Count`。

```
1 public class MyGenericList<T> : IList<T>
2 {
3     private ArrayList _elements;
4
5     public int Count
6     {
7         get { return _elements.Count; }
8     }
9
10    public T this[int index]
11    {
12        get
13        {
14            return (T) _elements[index];
15        }
16        set
17        {
18            _elements[index] = value;
19        }
20    }
21 }
```

如你所見，泛型類別與一般類別實作介面的規則是完全一樣的：如果類別在宣告時有指定要實作某個（或某些）介面，則該類別就必須提供那個（或那些）介面所規定的所有成員，包括方法、屬性、事件，否則將無法通過編譯。

再來看一個泛型類別實作泛型介面的例子：

```
1  // 宣告泛型介面
2  interface ISayHello<T>
3  {
4      string GetWords(T obj);
5  }
6
7  // 實作泛型介面時，型別參數由外界指定。
8  class Hello1<T> : ISayHello<T>
9  {
10     public string GetWords(T obj)
11     {
12         return "Hello, " + obj.ToString();
13     }
14 }
15
16 // 實作泛型介面時，型別參數明確指定特定型別。
17 class Hello2<T> : ISayHello<int>
18 {
19     public string GetWords(int i)
20     {
21         return "Hello, " + i.ToString();
22     }
23 }
24
25 class Program
26 {
27     static void Main(string[] args)
28     {
29         var hello1 = new Hello1<string>();
30         Console.WriteLine(hello1.GetWords("michael"));
31
32         var hello2 = new Hello2<string>();
33         Console.WriteLine(hello2.GetWords(10));
34     }
35 }
```

在此範例中，Hello1 與 Hello2 這兩個泛型類別都實作了泛型介面 ISayHello，但它們的實作方式不太一樣。Hello1 在宣告實作介面時，泛型介面的型別參數仍舊維持 T，亦即該型別同樣是由外界指定。Hello2 則是在宣告時明白指定了欲實作的泛型介面是 ISayHello<int>。



你可以比較這兩個類別的 `GetWords` 方法的實作，便可看出它們的差異。同時也請注意，在 `Main` 方法中，建立 `Hello1` 與 `Hello2` 物件時所傳入的型別引數都是 `string`，可是在呼叫 `GetWords` 方法時，`Hello2` 物件就一定要傳入 `int`。這兩種寫法都可以編譯、執行，該用哪一種，端看實際的需要而定。一般來說，`Hello2` 的寫法會用在特別需要限定型別參數的場合。

不過，底下這種寫法就無法通過編譯了：

```
1 class Foo3<T> : ISayHello<int>, ISayHello<T>
2 {
3     ....
4 }
```

這是因為一個類別不能實作重複的介面。`ISayHello<T>` 的型別參數 `T` 就已經涵蓋 `int` 了，因此編譯器會將它們視為重複的介面。

泛型介面的部分就介紹到這裡。至於泛型結構，其宣告語法與前述之泛型類別與泛型介面相同，此處便不再贅述。僅舉一例：`.NET Framework` 內建的 `KeyValuePair<TKey, TValue>` 就是一個泛型結構，用來儲存一組索引鍵與資料值。其宣告如下：

```
public struct KeyValuePair<TKey, TValue>
```

讀者若有興趣了解其用法，可至 MSDN 網站參考線上文件的說明。

## 2.3 泛型方法

泛型方法（**generic method**；或譯為「泛用方法」）與其他泛型不同：它不是型別，而是個函式（方法）。泛型方法可宣告在一般的型別裡（包括類別、介面、結構），也可以宣告在泛型裡。泛型方法的宣告語法如下：

```
回傳型別 方法名稱 <T1, ..., Tn>(T1 param) [where 子句]
{
    ...
}
```

說明：

- `<T1, ..., Tn>` 是型別參數。
- `(T1 param)` 是形式參數。
- `[where 子句]` 是型別條件約束。

注意泛型方法的宣告包含兩種參數列：型別參數列和形式參數列。型別參數列緊跟在方法名稱之後、形式參數列之前。最後的 `where` 子句可有可無，其用法已在前面提過，這裡不再贅述。底下是一個簡單的範例：

```

1  class GenericMethodDemo
2  {
3      public void Print<T>(T obj)
4      {
5          Console.WriteLine("Hello, " + obj.ToString());
6      }
7
8      public R Print<T, R>(T obj) where R : new()
9      {
10         Console.WriteLine("Hello, " + obj.ToString());
11         R result = new R();
12         return result;
13     }
14 }

```

此範例在非泛型類別中宣告兩個多載的泛型方法：Print<>，其中一個沒有傳回值，另一個有，而且使用了 where 子句來限定傳回值型別 R 必須有提供預設建構式。下圖的 Main 函式則示範了如何呼叫這兩個泛型方法，其中的虛線方框是即時編譯器（JIT compiler）將型別參數帶入之後所展開的程式碼。

```

static void Main(string[] args)
{
    GenericMethodDemo gmd = new GenericMethodDemo();

    gmd.Print<string>("Jacky");
    gmd.Print<int>(100);
    // 呼叫另一個版本.
    DateTime dt = gmd.Print<string, DateTime>("Current time");
}

```

```

public void Print(string obj)
{
    ....
}

public void Print(int obj)
{
    ....
}

```

## 型別推斷

如果在呼叫泛型方法時，編譯器有辦法推斷傳入參數的型別，那麼型別參數就可以省略不寫。所以剛才的範例也可以改寫成這樣：

```

1  static void Main(string[] args)
2  {
3      GenericMethodDemo gmd = new GenericMethodDemo();
4
5      string s = "Jacky";
6      gmd.Print(s);    // 編譯器知道傳入參數的型別，故可省略型別參數
7
8      int i = 100;
9      gmd.Print(i);    // 編譯器知道傳入參數的型別，故可省略型別參數
10
11     DateTime dt2 = gmd.Print<string, DateTime>("Current time");
12 }

```

最後一個呼叫敘述維持原樣，是因為傳回值的型別必須明確宣告，所以還是得傳入完整的型別參數。

### 擴充方法與泛型類別

擴充方法也可以用來擴充泛型類別，其規則與擴充一般的非泛型類別相同：

- 必須寫在靜態類別裡（提供擴充方法的類別必須宣告為 `static`）。
- 必須是靜態方法。
- 第一個參數必須使用關鍵字 `this`，後面跟著欲擴充的泛型類別名稱，然後是參數名稱。

參考底下的範例程式：

```

1  public static class ListExtender
2  {
3      public static void Print<T>(this List<T> aList)
4      {
5          for (int i = 0; i < aList.Count; i++)
6          {
7              Console.WriteLine("{0} : {1}", i, aList[i].ToString());
8          }
9      }
10 }
11
12 static void Main(string[] args)
13 {
14     List<string> strList = new List<string>();
15     strList.Add("Bruce");
16     strList.Add("Robin");
17     strList.Print();    // 也可以寫成 strList.Print<string>();
18 }

```

此範例的執行結果如下：

```
0 : Bruce
1 : Robin
```

## 2.4 泛型與 C++ 樣板的差異

如果你沒寫過 C++，本節內容對你來說或許就不是那麼重要。不過，進一步了解 .NET Framework 泛型背後的運作機制倒也沒什麼壞處。

泛型與 C++ 樣板（`template`）之間的主要差別在於泛型的型別展開是動態的（發生於執行時期），而 C++ 樣板的型別展開是靜態的（發生於編譯時期）。由於 C++ 樣板必須在編譯時期就展開成實際型別，因此，C++ 樣板並無法單獨包在獨立的 DLL 裡面，供其他程式重複使用。這也是為什麼 C++ 樣板程式庫（例如 STL）的程式碼通常都寫在標頭（`header`）檔案裡的緣故。

相較之下，.NET Framework 的泛型就可以包成單獨的 DLL 組件（`assembly`），供其他程式重複使用。因為 .NET 泛型是等到程式執行時才由 JIT（`just-in-time`）編譯器動態產生「建構的型別」（`constructed type`）。

在本節結束前，讓我們來看一段挺有趣的程式碼：

```
1  // 一個有趣的泛型方法呼叫範例
2  public class TestGenericCall
3  {
4      public static void Run()
5      {
6          Foo.DoIt<string>("Mike"); // 等同於 Foo.DoIt("Mike");
7      }
8  }
9
10 public class Foo
11 {
12     public static void DoIt<T>(T t)
13     {
14         ReallyDoIt(t); // 會呼叫哪一個版本的 ReallyDoIt 方法?
15     }
16
17     public static void ReallyDoIt(string s)
18     {
19         Console.WriteLine("Natural version: " + s);
20     }
21
22     public static void ReallyDoIt<T>(T t)
```

```
23     {  
24         Console.WriteLine("Generic version: " + t);  
25     }  
26 }
```

此範例中，類別 `TestGenericCall` 的 `Run` 方法呼叫了 `Foo` 類別的 `DoIt<T>` 方法，此方法又會呼叫 `ReallyDoIt`，而這個 `ReallyDoIt` 方法有兩個版本。那麼，問題就是：究竟會呼叫哪一個版本的 `ReallyDoIt` 呢？

執行結果會輸出 `"Generic version: Mike"`。也就是說，呼叫的是泛型的版本。原因在於，當編譯器碰到多載方法（overloaded methods）時，會根據呼叫時所傳遞的參數型別來決定哪一個方法才是最合適的版本。在這個例子當中，由於呼叫 `ReallyDoIt` 時傳入的參數型別是泛型 `T`，於是編譯器會優先尋找同樣需要傳入泛型參數的版本。如果將程式碼改為直接呼叫 `Foo.ReallyDoIt("Mike")`，那麼執行的結果就會是 `"Natural version: Mike"`。

進一步來說，此行為也和前面所說的 .NET 泛型運作機制有關：泛型是在執行時期展開，而非編譯時期。在編譯時期，C# 編譯器並不會因為你在呼叫泛型方法時分別傳入了 `string` 和 `int` 兩種參數而編譯出兩種版本的泛型方法的 IL 代碼（intermediate language code）。既然泛型方法不會在編譯時期展開，因此在這個範例中，`Foo` 類別的 `DoIt<T>` 方法並不會在編譯時期把呼叫端傳入的 `string` 型別參數帶入並展開。也就是說，編譯出來的 IL code 就只有一個 `DoIt<T>` 方法。

對編譯器來說，泛型就只是…泛型。因此，當編譯器在尋找 `ReallyDoIt(t)` 的最佳版本時，自然會選擇與參數 `t` 最相符的版本，也就是泛型的版本。

## 2.5 泛型的型別相容問題

布魯斯：「請告訴我，字串是不是一種物件？」

羅賓想都沒想，立刻回答：「當然是囉。」（這問題也太簡單了吧！）

「那麼，字串串列是不是一種物件串列？」

羅賓稍微想了一下，答案當然也是肯定的。

「很好，」布魯斯再問：「那如果有一段程式碼，將一個 `List<String>` 型別變數指派給另一個 `List<Object>` 型別的變數，這樣寫 OK 嗎？我是說，編譯會出錯嗎？還是編譯會過，但執行時會出錯？」

羅賓：「呃……」

欲解答上述問題，必須了解泛型的型別相容規則，而其中的關鍵概念，則在於 .NET Framework 4 進一步支援的泛型 **covariance** 與 **contravariance** 機制。這兩個術語有點抽象，不太容易說明白。這裡嘗試用一些簡單的例子來說明相關的概念。

### 專有名詞的翻譯

對於 **covariance** 和 **contravariance**，常見的中文翻譯分別是「共變數」與「反變數」。這兩個術語，筆者是傾向直接使用英文，所以本節會出現較多中英文夾雜的情形，還請讀者見諒。若您覺得不用中文就會有不踏實、難以前進的感覺，這裡提供您另一個選項：「共變性」和「逆變性」。此外，還有另一個相關的名詞：**invariance**，或可譯為「不變性」。這只是一種選擇，並未指涉孰優孰劣。

## 核心概念：型別相容

就如本節標題所揭示的，其實無論 **covariance**、**contravariance**、還是 **invariance**，這些「-variance」所牽涉到的，都是有關型別安全的議題，尤其是靜態的（編譯時期的）型別安全檢查。進一步說，設計這些語言功能的主要用意，是讓你在寫程式時，能夠把某個型別的物件當成另一種（相容）型別的物件來處理。

這些「-variance」概念並不是到了 .NET Framework 4 才有，其實 C# 1.0 的陣列語法就已經支援 **covariance** 了。那麼，什麼是 **covariance** 呢？

先想一下這個簡單的問題：「字串陣列是不是一種物件陣列？」或者，「一串香蕉是不是一串水果？」答案應該很明顯吧！請看以下範例程式碼。此範例展示了 C# 陣列原本就有支援的相容型別轉換機制。

```
1 string[] strings = new string[3];
2 object[] objects = strings;    // 將字串陣列 assign 給物件陣列（隱含轉型）
3
4 objects[0] = DateTime.Now;    // 執行時會發生 ArrayTypeMismatchException!
5 string s = (string)objects[0];
```

我們知道，字串也是一種物件，那麼從型別相容的原則來看，將字串陣列指派（assign）給物件陣列就沒什麼好奇怪的了。這裡的型別相容原則，指的是比較「大」的型別可以兼容同一家族後代的「小」型別。如果你覺得太囉嗦、不好記，可以試試看這個口訣：大的可以吃小的。這裡的「大」、「小」，指的是繼承階層中的位階高低。換言之，比較「小」的子代物件可以指派給同一族系的父代型別的變數。有點像繞口令？底下是比較正式的说法：

若型別 A 繼承自型別 B（A 比 B 小），則型別 A 可以隱含（自動）轉換為型別 B，且陣列 A[] 可以指派給陣列 B[]。

這是 C# 原本就支援的陣列 **covariance** 型別相容規則。

剛才的範例程式碼雖然在 C# 2/3/4 都可以通過編譯，但是這樣的 **covariance** 寫法並不是頂安全。比如說，它允許我們將一個 **DateTime** 物件丟進物件陣列，而那個物件陣列裡面的每

個元素的型別其實是 `string`。把 `DateTime` 物件指派給 `string` 型別的變數，執行時當然會出錯了。簡言之，陣列 `covariance` 的缺點就是無法在編譯時期抓出型別不相容的錯誤，而必須等到執行時期才會出現（希望不是由使用者發現）。相對的，泛型則提供了編譯時期的型別相容檢查。這正是我們喜歡泛型集合的主要原因之一。

### 一小段歷史

既然這種陣列 `covariance` 有型別安全的問題，為什麼 C# 要支援這種語法呢？根據微軟的一位主管 Jonathan Allen 的說法，.NET 甫推出時，為了提高 C# 與 Java 的相容性，便決定向 Java 看齊，讓 C# 和 Visual Basic 都支援此語法。儘管有些人覺得這是錯誤的決定，但每一項設計決策都是取捨，亦有其歷史背景，這項既成的事實應該是不會改變了。

（資料來源：<http://www.infoq.com/news/2008/08/GenericVariance>）

## Covariance

儘管 C# 陣列的 `covariance` 語法不太「安全」，但根據型別相容的規則，編譯器還是得允許這麼寫。不過，同樣的概念如果套用到 C# 2.0 開始提供的泛型，就不是這麼回事了。考慮以下程式片段：

```
List<string> stringList = new List<string>();  
List<object> objectList = stringList; // 即使 C# 4 也不能這樣寫！
```

既然字串是一種物件，字串串列自然也是一種物件串列囉，這應該是很直觀的寫法。可是，即使到了 C# 4，也不允許直接把泛型字串串列指派給物件串列，編譯器會告訴你：

Cannot implicitly convert type 'System.Collections.Generic.List

為什麼會這樣？這是因為泛型本身具有不變性（`invariance`）的緣故。為了方便理解，你可以把 `invariance` 想成呼叫函式時所傳遞的 `ref` 參數，亦即傳參考（`pass-by-reference`）的參數，因為它們的效果是一樣的：當某個參數是以傳參考的方式傳遞時，傳遞給該參數的變數型別必須跟宣告時的參數型別完全一樣，否則編譯器將視為不合法的陳述式。

可是，字串串列明明就是一種物件串列，寫程式時卻不能將字串串列指派給物件串列，在某些應用場合總是不太方便。於是，.NET Framework 4 針對泛型的部分進一步支援了 `covariance` 和 `contravariance`，而 C# 4 和 Visual Basic 10 也提供了對應的語法。在 C# 4，先前的範例程式碼可以改成這樣：



```
List<string> stringList = new List<string>();  
IEnumerable<object> objects = stringList; // C# 2/3 不允許, C# 4 OK!
```

如此一來，雖然你還是不能直接把字串串列指派給物件串列（基於型別安全的理由），但相容型別之間的串列指派操作已經可以透過泛型介面 `IEnumerable<T>` 獲得解決。這是 `covariance` 的好處——你可以將比較「小」的型別當作比較「大」的型別來操作。

可是，為什麼同樣的寫法，C# 2/3 編譯會失敗，到了 C# 4 卻能編譯成功？答案就在 .NET Framework 4 的 `IEnumerable` 原型宣告：

```
1 public interface IEnumerable<out T> : IEnumerable  
2 {  
3     IEnumerable<T> GetEnumerator();  
4 }
```

注意泛型參數 `T` 的前面多了一個修飾詞：`out`。這是 C# 4 新增的 `covariance` 語法，其作用是告訴編譯器：`IEnumerable<T>` 這個泛型介面的型別參數 `T` 在該介面中只能用於輸出的場合。換個方式說，型別 `T` 在這個泛型介面中只能用於方法的傳回值，而不能用來當作方法的傳入參數。當你在參數前面加上關鍵字 `out`，編譯器就會強制檢查這項規則。因此，下面這段程式碼會無法通過編譯：

```
1 public interface IFoo<out T>  
2 {  
3     string Convert(T obj); // 編譯失敗：型別 T 在這裡只能用於方法的傳回值，不可當作參數。  
4     T GetInstance();      // OK!  
5 }
```

如果你對「`covariant` 型別只能用於輸出的場合」這句話沒有太大的感覺，也不要緊。只要記住一個重點，也就是此規則對於我們在寫程式時所提供的好處：任何參考型別（`reference type`）的泛型串列，你都可以將它當作 `IEnumerable<object>` 的物件來操作。舉例來說，如果你有一個用來處理集合物件的演算法，而且想要將它寫成共用函式，讓它能夠處理各種泛型串列，例如員工串列、學生串列，你就可以為這個共用函式加入一個 `IEnumerable<object>` 型別的參數。如此一來，只要是參考型別的泛型串列物件，便都可以傳入這個函式了。參考底下的程式範例。



```
1 public static void Main()
2 {
3     List<string> strings = new List<string>();
4     Print(strings);
5
6     List<int> numbers = new List<int>();
7     Print(numbers); // 無法通過編譯: int 不是參考型別。
8
9
10 }
11
12 static void Print(IEnumerable<object> list)
13 {
14     ....
15 }
```

也許你會質疑:「這樣寫固然方便,可是用 `IEnumerable<T>` 把一串物件包起來,難道不會跟前面的陣列 `covariance` 範例一樣有執行時期型別安全的問題嗎?」答案是不會。因為 `IEnumerable<T>` 的操作都是唯讀的,你無法透過它去替換或增刪串列中的元素。

在看更多範例之前,讓我們稍微整理一下前面提過的兩個重點:

- `Covariance` 指的是可以將比較「小」的型別當作比較「大」的型別來處理。
- 無論你對 `covariance` 的理解為何,編譯器在乎的是:有加上 `out` 修飾詞的泛型引數只能用於輸出的場合。

### 泛型介面的 **Covariance** 範例

經過前面的說明,如果您已經瞭解 `covariance` 的意義和作用,大可直接跳過這個小節。如果還是覺得有點模糊,不妨再看一個例子吧。希望透過這個範例的解說,能讓你有豁然開朗的感覺。

假設我們設計了三個類別: `Fruit` (水果)、`Apple` (蘋果)、和 `Peach` (水蜜桃),其中 `Apple` 和 `Peach` 皆繼承自 `Fruit` (蘋果和水蜜桃都是一種水果)。這些類別的屬性和方法並不重要,故不列出。現在我們打算建立兩個串列,分別用來儲存蘋果和水蜜桃,程式碼如下:

```
1 // 蘋果串列
2 List<Apple> apples = new List<Apple>();
3 apples.Add(new Apple());
4 apples.Add(new Apple());
5
6 // 水蜜桃串列
7 List<Peach> peaches = new List<Peach>();
8 peaches.Add(new Peach());
9 peaches.Add(new Peach());
```

接下來，我們可能需建立一個綜合水果串列，把剛才的 `apples` 和 `peaches` 這兩個串列裡面的元素都加到綜合水果串列裡。以下程式碼示範了 C# 4 支援的兩種作法。一種是建立一個 `List<Fruit>` 泛型串列，另一種則是使用 `IEnumerable<T>`。

```
1 // 建立綜合水果串列。
2 List<Fruit> fruits = new List<Fruit>();
3 fruits.AddRange(apples); // C# 4 OK! C# 2/3 編譯失敗。
4 fruits.AddRange(peaches); // C# 4 OK! C# 2/3 編譯失敗。
5
6 // 或者也可以這樣寫：
7 IEnumerable<Fruit> fruits2 = apples; // C# 4 OK! C# 2/3 編譯失敗。
8 fruits2 = fruits.Concat(peaches);
```

是否有點驚訝？原來「把一堆蘋果和水蜜桃包裝成一個綜合水果禮盒」的概念在 C# 3.0 竟然無法用這麼直觀的方式實作。在這個範例裡面，我們先示範以泛型串列的 `AddRange` 方法來加入蘋果和水蜜桃串列。`AddRange` 方法的原型宣告是：

```
public void AddRange(IEnumerable<T> collection)
```

請注意傳入參數的型別是 `IEnumerable<T>`，其中的 `T` 就是程式在宣告 `fruits` 變數時指定的參數型別，所以這裡的 `AddRange` 的傳入參數型別就是 `IEnumerable<Fruit>`。當我們將 `apples` 和 `peaches` 當作引數傳遞給 `AddRange` 方法時，就等於把比較「小」的 `List<Apple>` 和 `List<Peach>` 指派給比較「大」的 `IEnumerable<Fruit>`。編譯器知道 `IEnumerable<out T>` 的型別 `T` 符合 `covariance`，故基於型別相容原則，編譯器會允許這個指派操作，並自動轉換型別（注意只是型別的轉換，其中並未建立任何新物件）。

第二種寫法，是先把 `apples` 指定給 `IEnumerable<Fruit>` 的串列（這裡也同樣有發生 `covariance` 的隱含轉型動作），然後再利用 `System.Linq.Enumerable` 類別提供的擴充方法 `Concat` 來附加另一個 `peaches` 串列。

你會發現，不管哪一種寫法，主角都是泛型介面 `IEnumerable<T>`。事實上，.NET Framework 4 只有泛型介面和泛型委派有支援 `covariance` 和 `contravariance`，而且這類型別的數量並不多。所以在解釋各種 `-variance` 概念時，大部分都是拿 `IEnumerable<T>` 來舉例說明。這多少

也意味著 `IEnumerable<T>` 是我們的好朋友，寫程式的時候可別忘了還有這個傳遞集合物件的好用型別。



本章沒有介紹泛型委派，是因為其中牽涉到委派的語法。這部分會另闢專章討論，請參考〈第 3 章：委派與 Lambda 表示式〉。

Covariance 討論得夠多了，接下來要看的是 contravariance。

## Contravariance

理解 covariance 之後，contravariance 就不會太困難了。基本上，二者只是「方向相反」而已：

- Covariance 限定用於輸出的場合，contravariance 則是指泛型參數型別只能用於輸入的場合（方法的傳入參數）。
- Covariance 讓你可以將比較「小」的型別指派給比較「大」的型別；contravariance 則是允許將比較「大」的型別指派給比較「小」的型別。

沿用前面的水果例子，假設 `Fruit` 類別有一個公開屬性 `SweetDegree`，型別是 `int`，代表水果的甜度。現在我們想要對一個蘋果串列依甜度排序，程式碼如下所示。

```
1 public class ContravarianceDemo
2 {
3     public void Run()
4     {
5         // 建立蘋果串列.
6         List<Apple> apples = new List<Apple>();
7         apples.Add(new Apple() { SweetDegree = 54 });
8         apples.Add(new Apple() { SweetDegree = 60 });
9         apples.Add(new Apple() { SweetDegree = 35 });
10
11        // 依甜度排序.
12        IComparer<Fruit> cmp = new FruitComparer();
13        apples.Sort(cmp);
14
15        foreach (Fruit fruit in apples)
16        {
17            Console.WriteLine(fruit.SweetDegree);
18        }
19    }
20 }
21
```

```
22 public class FruitComparer : IComparer<Fruit>
23 {
24     public int Compare(Fruit x, Fruit y)
25     {
26         return x.SweetDegree - y.SweetDegree;
27     }
28 }
```

注意此範例中的 `apples.Sort(cmp)` 方法呼叫。在撰寫這行敘述時，Visual Studio 的 IntelliSense 功能會提示這裡的 `apples.Sort` 方法需要傳入的參數型別是 `IComparer<Apple>`，但我們傳入的參數卻是 `IComparer<Fruit>`。也就是說，這裡會發生「把較大型別指派給較小型別」的動作（跟 `covariance` 正好相反）。之所以能夠這樣寫，是因為 `ICompare<T>` 的泛型參數 `T` 在 .NET Framework 4 已經加入了 `contravariance` 的宣告，也就是修飾詞「`in`」。底下是 `ICompare<T>` 的原型宣告：

```
1 interface IComparer<in T>
2 {
3     int Compare(T x, T y);
4 }
```

透過前面幾個範例程式碼的解說，相信讀者應該已經能夠大致掌握 `covariance` 與 `contravariance` 的精神了。

## 2.6 結語

本章先從型別安全的角度解釋為什麼要有泛型，以及使用泛型的好處。然後，我們看到了泛型的語法，包括型別參數、條件約束（`where` 子句）、建構式、預設值的表示法等等。除了泛型類別之外，泛型語法也可以用於介面、結構、委派、以及方法。



泛型委派的部分請參考〈第 3 章：委派與 Lambda 表示式〉。

在說明泛型語法的同時，我們也同時討論了 .NET 泛型與 C++ 樣板的差異。基本上，C++ 樣板是在編譯時期就由編譯器將型別參數帶入，並展開成實際的型別。而 .NET 泛型則是等到程式執行時，才由 JIT 編譯器產生建構的型別（`constructed type`）。換言之，對 C# 編譯器來說，泛型就真的只是泛型，在編譯成 IL code 時，並不會有展開建構型別的動作。

本章最後提到了泛型的型別相容問題，主要在解釋 `covariance` 與 `contravariance` 的概念，以及它們對我們實際撰寫程式的時候會有哪些影響。這一節的議題可能略嫌艱澀，讀者若能將書中的範例程式碼實際練習一遍，並稍加修改變化，相信一定能有更深的體會。