01:用 Pascal 寫程式

這個章節會從一些可以用來建立一個 Object Pascal 應用程式的程式片段開始，會涵蓋到一些標準的程式寫法，程式註解、介紹關鍵字與完整程式的架構。我會開始寫一些簡單的程式，試著用透過說明這些程式碼來介紹並帶出接下來幾個章節的關鍵概念。

我們開始來看程式碼吧

這個章節涵蓋了 Object Pascal 語言的基礎，但也會花我們幾個章節的篇幅來帶領讀者們理解整個應用程式作業的一些細節。所以，我們先來快速的看兩個入門的程式吧(它們的架構會有所不同)，我們不會看的太仔細。目前我只想介紹一下我會用來建立範例程式的架構，接著我們才能介紹其他不同的部分。所以，我希望讀者能夠儘快開始把書裡提到的練習用相關資訊取回，從最開始的練習範例開始看，會是個好主意。

筆記 如果您是 Object Pascal 這個語言的初學者，需要逐步操作的指引來協助您使用本書的範例程式，或者讓您可以自己開始動手寫程式的話，請參閱本書的附錄 C.

Object Pascal 一開始就被設計成在透過 IDE 環境中可以即時上手，經由這個語言與 IDE 的堅強組合，Object Pascal 提供了對程式人員最友善的快速開發工具與語言。

在 IDE 裡面，您可以設計使用者介面、協助您撰寫程式碼，執行寫好的程式，還有更多的輔助功能。在本書中，就像我會介紹 Object Pascal 的程式語言，我也會跟您分享我使用 IDE 的方法。

第一個文字模式的應用程式

在一開始，我要透過一個文字模式應用程式，只簡單的顯示 Hello Word 這樣一個字串，來介紹 Pascal 程式語法的一些重要的部分。文字模式的應用程式，換句話說就是沒有視窗畫面的應用程式，執行時會以 DOS 視窗顯示文字，接受使用者透過鍵盤輸入，也只以 DOS 視窗顯示結果。文字模式的應用程式在現今的 PC 上面已經不太實用，但在行動平台上面，有時還是很

有用處的。

我暫時不會對以下這些程式碼做太多說明，這也是本書前幾章的用途，以下是 HelloConsole 這個程式專案的程式碼內容：

```
program HelloConsole;
{$APPTYPE CONSOLE}
var
    strMessage: string;
begin
    strMessage := 'Hello, World';
    writeln (strMessage);
    // 以下這個指令，是用來等待使用者輸入，直到使用者按下 Enter 鍵為止
    readln;
end.
```

筆記 在一開始的介紹中，我們已經介紹過，本書所有完整的程式碼都可以在 subversion 的 repository 裡面下載。這些範例的詳細介紹則會在本書裡面進行，在前文中，我已經提到專案名稱(在本範例裡面叫做 HelloConsole)，專案名稱也會用來當做資料夾的名稱，該資料夾裡面還有許多跟這個專案相關的檔案，由於我會把一個章節的範例放在同一個資料夾裡面，所以上面這個專案的資料夾名稱會是 01/HelloConsole.

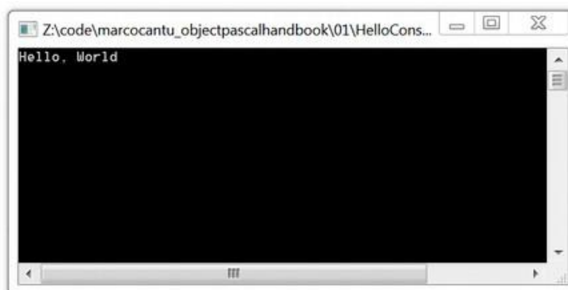
您可以在範例程式第一行看到程式的名字，程式名之後會包含一些指示詞(directives)：編譯程式的設定值(會以$這個符號開頭，並且用大括號整個包起來)、變數宣告的區塊（一個字串變數，命名為 strMessage），以及被 begin 跟 end 所包起來的三行程式碼跟一行說明用的註解。

這三行程式碼會把一段文字複製到一個字串變數裡面去，呼叫一個系統函式來把這個字串變數裡面的內容輸出在文字模式視窗裡面，並且呼叫另一個系統函式，等待使用者輸入（在這個範例中，只是用來等待使用者按下 Enter 鍵）。我們接下來就可以自行定義我們需要的函式，不過 Object Pascal 已經幫我們附上了數百個常用的函式了。

再強調一下，我們很快就會開始介紹這些程式內容，在一開始的章節，我們只是給您一個簡單的印象，讓您大概知道 Pascal 完整的程式大概長什麼樣子而已，當然您也可以直接打開、執行這個程式，程式執行的畫面，會像圖 1.1 的 DOS 視窗一樣，視窗的內容文字如下：

```
Hello, World
```

圖 1.1: HelloConsole 範例在 Windows 上面執行的結果：



第一個視覺化程式

現在的應用程式，都已經不像上面這個範例，長得像很傳統的文字模式視窗，通常會有許多視覺化的元素（在Object Pascal裡面我們稱之為控制元件）顯示在視窗畫面中。在本書中，我們大部分的範例都會是用FireMonkey元件庫來製作的視覺化程式（即使大多數案例我都會把它簡化到只顯示簡單的文字）。

筆記 在 Delphi 裡面，視覺元件庫已經區分成兩個不同的類別了：一個是 VCL (專屬 Windows 平台上面使用)，以及 FireMonkey（支援多種不同的平台、裝置，包含桌面應用程式跟行動裝置都支援）在 Appmthod 當中，則只提供了 FireMonkey，支援多種裝置的開發。但要把這些範例改為 VCL 版本也都非常容易。

要了解一個視覺化程式結構的細節，您必須把本書大多數的篇幅都讀過，例如一個form是一個特定類別的物件實體，它包含了許多方法、事件處理常式，以及屬性。剛剛這句話所提到的每個部分，在本書中都會介紹到。但要建立一個應用程式，您不需要先成為專家，您只需要透過選單上面的選項，就能輕鬆的建立一個新的桌面應用程式或行動裝置應用程式了。我在本書前面幾個章節要介紹的，都會是以FireMonkey的範例為基礎（兩種IDE都支援），簡單的介紹如何透過form的選單跟滑鼠點擊操作來完成這些動作。一開始，請您先建立任何一種form（桌面或行動應用程式均可，通常我會建立一個空白的行動應用程式專案，這個專案在Windows環境也可以執行的），然後放一個button元件在上頭，以及一個多行的文字元件(或者Memo也可以)來顯示輸出的結果。圖1.2就是讓您看一下在IDE裡面，預設情形下，這個行動應用程式的form會長什麼樣子。在附錄C裡面，您可以讀到建立這個範例程式的每個步驟。

您如果想要製作一個類似的應用程式，只需要先建立一個空白的行動應用程式，然後在空白的form上面加入一個button元件即可。現在，我們來加入程式碼吧。這也是目前我們要接著介紹的，請用滑鼠左鍵雙擊form畫面上的按鈕，您就會看到以下的程式碼在畫面上顯示出來(也可能是很類似的其他程式碼)。

```
procedure TForm1.Button1Click (Sender: Tobject) begin
end;
```

即使您都還不知道類別的方法是什麼(就是上面這段程式的Button1Click啦)，你也已經可以在上面的程式碼的begin跟end之間加入一些程式碼了，這些程式碼就會在專案執行時，當我們用滑鼠左鍵點選按鈕的時候被執行到。

我們的第一個視覺化程式，會有一些程式碼跟第一個文字模式程式完全相同，只是在視覺化程式裡面，我們呼叫了不同函式庫裡面的不同函式，在這個範例程式中，我們呼叫的是ShowMessage。這個範例的程式碼，您可以在名為HelloVisual的資料夾裡面找到，您可以試著直接編譯它，就可以發現執行編譯的動作真的是非常的簡單：

```
procedure TForm1.Button1Click (Sender: Tobject)
var
    strMessage: string;
begin
    strMessage := 'Hello, World';
    ShowMessage (strMessage);
end;
```

圖 1.2: HelloVisual 範例在 IDE 環境中顯示的畫面：

請留意到 strMessage 這個字串變數的宣告，是寫在 begin 這個保留字之前，而真正執行的程式碼則是寫在它之後，再說一次，如果對於任何部分覺得不太清楚，別擔心，所有程式都會隨著您讀到越後面，而有更詳細的說明。

筆記 您可以在名為 01 的目錄中找到本章所有範例的原始碼，為了容易辨認，在這個範例的資料夾裡面有一個名稱跟專案檔很相似的檔案，我把這個檔案名的前面加上了"Form"，以利區分，這也是我在本書當中用來為檔案命名的標準規則，專案的結構將會在本章的後面篇幅介紹。

在圖1.3裡面，您可以看到這個簡單程式的Windows上面執行的結果，但您也可以把這個範例程式拿到Android或者iOS上面執行，結果也會相同的。

現在我們已經介紹怎麼撰寫、測試一個範例程式了，讓我們回頭仔細看一下細節，一如我們在本章開始的時候我提到的順序。我們要介紹的第一件事，就是如何閱讀程式，各個不同部分的程式碼要如何撰寫、以及我們剛建立這個專案是怎麼組成的。(這個專案會包含有PAS檔案跟DPR檔案)

圖 1.3: HelloVisual 範例只有一個簡單的按鈕執行的畫面：

語法和程式碼樣式

在我們開始介紹Object Pascal程式指令之前，我們要先來看一些Object Pascal程式碼的樣式，我在這裡想要點出的問題是：在程式語法之外(我們還沒開始介紹)，我們要怎麼來撰寫程式碼呢？這個問題並沒有固定的標準答案，每個人都有自己習慣的寫法，不同的習慣寫法就會讓程式碼看起來有不同的樣式。然而，還是有一些固定的規範必須先介紹給大家知道，例如註解、大小寫、空白字元，以及多年以前曾經被稱為*美觀列印*的排列樣式(這裡指的美觀當然是讓人來閱讀，跟電腦沒有關係)，而這個名詞現在也已經很少聽到人提起了。

通常程式碼樣式是為了讓人在閱讀程式碼的時候可以更簡潔、更快速的了解程式碼，這些樣式跟格式就是您可以決定的一些程式碼的排列方法，好讓程式看起來更整齊。而要讓程式碼看起來整齊，就必須要堅持相同的程式碼樣式，不管您選擇了哪一種樣式，記得要在整個專案的所有檔案當中都用同樣的程式碼樣式，不然反而會讓程式碼看起來更難懂。

筆記 IDE(Integrated Development Environment，整合開發環境，可以簡稱為開發環境)已經支援自動格式化程式碼的功能（可以選擇針對單一檔案或者整個專案），您可以按下快速鍵 Ctrl+D 要求 IDE 對目前的檔案進行程式碼樣式重新格式化，這個格式化的功能可以讓我們自己對 40 幾個程式碼樣式的細節做設定。(請從 Options 選單當中找到這個設定畫面)，您也可以把這些設定匯出，讓同一團隊的其他開發人員共享這些設定值，這樣可以讓整個團隊的程式樣式更為一致。

程式註解

雖然程式碼通常已經很容易被讀懂，但如果加上一些註解的話，其他人就更容易看的懂（如果過了一段時間之後，我們又回頭看一些自己也的程式的話，有註解也更容易看的懂）為什麼當時這段程式要這樣寫，以及當時寫這段程式的前提是什麼。

傳統的Pascal程式註解，是以兩個大括弧、或者小括弧帶星號來標註某段文字為註解的，而近期版本的Object Pascal程式語言則是把C++的註解語法，也就是用兩個斜線來標註其後的文字為註解，所以，以下的三種寫法，都是目前的Pascal語言可以辨識的註解寫法：

```
{ 這裡面的文字都是註解 }
(* 這是第二段註解 *)
// 從左邊出現了兩個斜線以後，到本行的末端都會被視為註解
```

第一種註解的寫法最為常見，第二種寫法則比較常出現在歐陸，因為許多歐洲國家的鍵盤上面沒有大括弧的符號，第三種註解的語法則是從C跟C++借來的，C跟C++裡面也會用/*註解*/這樣的語法來標註跨行的註解文字，這在C#, Objective-C, Java跟JavaScript裡面都可以看見。

單行註解的語法很有用，常用來寫短短的註解，或者把特定一行程式碼先暫時標註掉，這個語法也漸漸的成為在Object Pascal最常被使用的註解語法。

筆記 在IDE的編輯器裡面，您可以按下Ctrl+/這組快速鍵，來把單一一行程式碼，或者選擇多行程式碼進行註解或者解除註解，這組快速鍵在英文鍵盤裡面可以直接使用，但如果是其他語系的鍵盤或輸入法，就要先確定一下/這個符號的位置，實際的按鍵，可以從編輯器的功能選單(按滑鼠右鍵就會顯示了)來清楚的看到。

我們介紹了三種不同語法的註解文字，這些語法可以幫助我們把單行或者多行的程式碼先變成註解文字。如果您希望把程式碼或程式檔案裡面的部份文字變成註解，則您可以套用前述三種語法的不同排列來達成註解文字當中還有其他註解文字這種設定，但多行註解如果要包含其他註解的話，兩個註解文字的語法不能用同一種喔：

```
{ code...
        {comment, 這段註解底下的其他文字會不被當成註解喔}
code... }
```

上面這段程式碼會被編譯程式當成錯誤語法，因為第一個大括弧在第二行遇到了結束的大括弧，因此第三行已經不會再被視為註解文字，所以編譯程式就會判定語法有錯，我們可以把它改寫如下：

```
{ code...
        // this comment is OK
code... }
```

這樣的寫法，由於兩種註解語法不互相衝突，所以我們如果要把整段文字或程式碼的註解狀態取消，就只要把前後兩個大括弧拿掉就行了，第二行還是可以保持註解文字的狀態。

筆記 在大括弧之後如果出現一個錢字號$，就不再代表是註解，而是編譯器的設定代碼，例如在我們介紹的第一個範例程式當中，程式碼裡面有一行寫著{$APPTYPE CONSOLE}。編譯器設定代碼會讓編譯器去執行特定的一些動作，我們在本章後段加以介紹。

其實，編譯器設定代碼說到底也算是註解，例如{$X+ 這是一個註解}這樣寫也不會造成語法錯誤，這種寫法同時扮演兩個角色，不過大多數的程式人員都還是會把編譯器跟註解給分開來寫就是了。

識別符號(Symbolic Identifiers)

一個程式是由許多個不同的識別符號所構成的,我們會用這些識別符號來為不同的程式區段命名(例如資料型別、變數、函式、物件、類別等等)即使我們可以幾乎可以使用我們想用的任何一個識別符號,仍然需要遵守一些規則:

- 識別符號不能包含空白字元(空白字元會用以區分不同的識別符號)
- 識別符號可以包含英文字母與數字,包含字元與所有Unicode的文字,所以我們已經完全可以用任何一種語言的文字(當然用中文也行)來做為識別符號的名稱了。
- 在傳統的ASCII符號之外,識別符號只能包含底線(_),其他的ASCII符號則不可以用在識別符號上,不可以使用的符號,包含有+, -, *, /, = 以及所有標點符號跟括弧、特殊字元(像是@ # $ % ^ & \ |),我們只能用Unicode的字元,例如 ☼ 或 ∞ 都可以。
- 識別符號必須用底線或字元來開頭,不可以用數字開頭(所以數字當然可以當成識別符號的一部分,只是不能當成第一個字)我們在這裡提到的數字是指0-9的阿拉伯數字,至於Unicode裡面的其他語言數字,例如國字的一或壹則沒有問題。

以下是很常見的一些識別符號命名方法,我把他們列在名為IdentifiersTest的範例程式中:

```
MyValue
Value1
My_Value
_Value
Val123
_123
```

以下則是一些合法的Unicode識別符號:

```
Cantù (拉丁文)
结 (簡體中文)
画像 (日文漢字)
☼ (Unicode 裡面的太陽符號)
```

接下來我們也介紹一些不合法的識別符號：

```
123 (數字開頭)
1Value (數字開頭)
My Value (內含空白字元)
My-Value (內含特殊符號)
My%Value (內含特殊符號)
```

提示 如果您在程式執行狀態中想要檢查識別符號是否合法(需要這樣做的情境很少發生，除非您撰寫的是幫助其他開發者的工具)，在執行階段函式庫(Runtime Library)當中有提供這樣的一個函式，名為 IsValidIdent。

使用大寫字母

跟許多其他的程式語言不同，許多以C語言為基礎的程式語言(像C++, Java, C#, 跟 JavaScript)都是把英文字母大小寫視為相異，英文稱為Case-Sensitive，Object Pascal是把英文字母大小寫視為相同的，英文稱為Case-Insensitive。

因此，對Object Pascal的編譯器來說，Myname, MyName, myname, myName, 以及MYNAME這五個大小寫各有不同的字串，編譯器會把他們全部當做完全一樣的字串。在我的觀點中，把英文字母大小寫視為相同，絕對是正面的功能，因為這樣一來，因為拼字錯誤或者打字一時打錯而造成語法錯誤的機會，會比其他語言發生的機會來的低。

如果我們把Unicode當成識別符號的情況也採計進來，事情就會變得更為複雜了，當我們把大小寫視為相同時，只有一個字母大小寫相異的關鍵字就會被視為完全相同，因此也就可以避免在語意上完全相同而只有大小寫拼法不同的情形發生了。

```
cantu: Integer;
Cantu: Integer; // 錯誤: 因為與前一個名稱重複
cantù: Integer; // 正確: 這是完全不同的名稱了
```

筆記 在 Object Pascal 的大小寫視為相同的規則中，只有一個情形例外，就是元件函式庫套件(component package)的 Register 函式，這個函式一定必須寫成第一個字母大寫的 Register, 因為必須相容於 C++。

當然，當我們在引入由其他語言製作的函式庫時，也必須隨著該函式庫製

作時所使用的字元大小寫，才能正確的呼叫、使用這些函式庫。

在字母大小寫的規則當中還是有一些問題的，所以我們首先要意識到不同大小寫的識別符號實際上是完全相同的，所以一定要避免在不同的地方使用相同的識別符號，其次，我們也要適當的使用大寫字母，讓我們的程式碼更容易被閱讀。

編譯器並沒有強制我們要在專案當中使用持續的規則，但使用持續的規則真的是個好習慣。通常大家會很習慣把第一個字母大寫，當我們要用連續幾個英文字作為識別符號的命名時，通常會讓每個有意義的英文字的第一個字母大寫，例如：

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

這樣的習慣通常被稱為Pascal-casing, 相對於Java以及其他以C語言語法為基礎的程式語言所使用的Camel-casing：第一個英文字的首個字母小寫，其他的英文字首個字母均大寫：

```
myLongIdentifier
```

實際上，目前也越來越常在Object Pascal的程式裡面看到Camel-casing的規則了，目前僅剩Class宣告、參數宣告以及其他全域變數的範疇內還會使用Pascal-casing，在本書中，還是會盡量在所有的識別符號中都使用Pascal-casing的規則的。

使用空白字元

空白字元、tab跟換行符號在程式碼裡面，幾乎是完全被編譯器所忽略掉的。這三個字元對於編譯器來說都被當成空白字元。空白字元只用來讓程式碼在閱讀上更為舒服，在編譯作業上完全沒有作用。

跟傳統的BASIC不同，Object Pascal允許我們把一個程式敘述句（statement）用很多行的程式碼來表達。允許用多行程式碼來表達一個程式敘述句的缺點，是我們自己必須記得在敘述句的最後加上一個分號，讓編譯器知道這是一個程式敘述句的完結點。Object Pascal的多行敘述句的唯一限制，是不能把字串用多行來表示。

以下這幾段程式雖然看起來奇怪，但他們都描述了同一件事情：

```
a := b + 10;
a := b
```

```
+ 10;
a :=
// 在程式碼當中夾一行註解, 也是可以的
 b + 10;
```

再次強調，在程式碼裡面使用空白字元跟換行符號並沒有一定的規則，但有幾個常用的寫法：

- 編輯器畫面通常在每行達到80個字元之後，就會切行，如果你的程式碼超過了這個長度，就會被推到下一行去，這是為了讓你的程式碼看起來更容易閱讀，因為這樣就不用橫向捲動，在比較小的螢幕上也能夠很快的把程式碼閱讀完畢。原本每行80個字元的用意，只是讓程式碼列印出來的時候比較好看，而近年來也很少看到程式碼列印的需求了。

- 當函式或程序有多個複雜的參數時，我們通常會讓每個參數放在單獨一行裡面，這個習慣是從C語言而來的。

- 我們也可在註解前後留下一行空白行，這樣可以讓我們的程式讀起來更清楚易懂。

- 記得在呼叫函式的時候，在每個參數之間加入一個空白，甚至是在不需要參數時，也在括號當中留一個空白字元，我也會在運算式裡面，讓每個運算符號前後多放一個空白，看起來好讀多了。

程式碼內縮

關於空白字元使用上的最後一個建議，是跟典型的Pascal語言格式相關的議題，原本是為了讓列印出來的程式碼比較美觀，但目前已經都統整為程式碼內縮來呈現了。

這個規則非常簡單：每當我們需要寫一段複合的程式敘述句，就把整段程式碼內縮兩個字元（不是tab字元，tab字元是C語言的開發人員常用的），如果這段程式碼當中還有其他複合的程式敘述，則該段程式碼再多內縮兩個字元，依此類推：

```
if ... then
   statement;
 if ... then
 begin
statement1;
   statement2;
 end;
```

```
if ... then
begin
  if ... then
    statement1;
  statement2;
end;
```

重申一次，不同的程式人員在這個常用的規則上會有自己慣用的作法，有些程式人員會把begin到end之間的程式碼內縮，而有些程式人員則會把begin這個關鍵字放在前一段程式碼的最後（C語言大多都是這樣做，譯者也是習慣如此），這是因為個人習慣而有些許不同而已。

相似的規則也常用在變數清單跟資料型別的定義上：

```
type
  Letters = ('A', 'B', 'C');
  AnotherType = ...
var
    Name: string;
    I: Integer;
```

在以往的作法，常常會在type宣告新的型別名稱時，讓所有的等號都放在同一個位置對齊，以及讓變數宣告時的冒號都對齊起來，但現在已經很少見了，上述的程式碼如果用以往的規則來編排，就會變成：

```
type
  Letters     = ('A', 'B', 'C');
  AnotherType = ...
var
    Name : string;
    I    : Integer;
```

內縮的排列也常用在跨行的程式敘述句上面，通常第二行以後的程式碼，就會被內縮，而函式的參數如果長過一行，也會被用內縮的方式來顯示：

```
MessageDlg ('This is a message',
    mtInformation, [mbOk], 0);
```

強化語法標示

為了讓Object Pascal的程式更容易被閱讀與編寫，IDE的編輯器具備了一個名為強化語法標示的功能。根據我們所繕打的程式碼語法跟關鍵字，這些程式碼會被以不同的顏色跟字體加以標註。預設設定中，關鍵字會以粗體顯示、字串跟註解會以不同的顏色顯示（而且通常會是斜體）等等規則。

保留字、註解跟字串這三個類型的程式元素絕對是這個功能中對程式人員幫助最大的，透過這個功能，我們可以一眼看出程式碼是不是拼錯字了？字串是不是少打了一個引號？以及跨行的註解有沒有把前後註解的標示符號正確的標示上去。

您可以很容易的透過Editor Colors這個設定來設定您自己喜歡的語法標示設定，如果您的開發工作只有自己需要看這些程式碼，則您可以直接設定，如果您需要跟其他程式開發人員一起工作，則請您還是先使用標準的顏色佈景吧，我自己也發現一旦習慣了特定的佈景顏色以後，一下看到其他顏色跟樣式的畫面的確很容易楞住一下，不知所措。

錯誤檢知和程式碼檢知

IDE編輯器有許多功能可以協助我們寫出正確的程式碼，最直覺的應該就是錯誤檢知這個功能了，當我們輸入了錯誤的語法時，編輯器畫面上就會立刻在第一個發生錯誤那一行標上一串紅色的書名號，讓我們知道編譯器無法辨識該段程式碼，目前已經連文字編輯器跟文書軟體也都有相同的功能了。

筆記 在您第一次試著撰寫 Object Pascal 程式的時候，請務必記得也要先把適當的 unit 引入，這樣可以避免您的程式檔案最上方被標註許多的錯誤，
正確的引入其他的 unit, 可以解決掉不少錯誤標示。

其他功能，例如程式碼自動完成，會協助您顯示當時我們輸入的可能程式描述句，它會列出許多符合我們輸入的函式名稱或者屬性名稱，我們只要用下拉選單選擇我們適用的fucntion即可。又或者是一個函式的參數，也會被以下拉式選單的方式呈現。又或者我們可以按著Ctrl按鍵，用滑鼠左鍵點選某一個程式碼裡面的文字，就可以直接跳到該變數/型別被宣告的地方了。接下來我們就不再贅述關於IDE編輯器的功能了。我們還是把主要篇幅用來介紹Object Pascal語言本身吧。

程式語言的關鍵字

這裡所指的關鍵字，就是由程式語言特別保留下來的識別符號。這些符號都是在程式語言裡面已經預先賦予了功能，所以我們在整個程式的任何部分都不能拿來當成我們自己寫的程式的符號（例如變數名、Class名等）。而基本上，指示詞（directives）跟關鍵字是有所不同的:關鍵字是不能拿來當做變數、類別名稱等識別字的，但指示詞只要不被放在{$}這個符號組合中，就

沒有影響，所以指示詞可以有其他的用途，但實務上，建議還是不要拿任何關鍵字（包含指示詞）來做為識別字比較好。

如果您寫了以下這樣的程式碼（property這個字是關鍵字喔）：

```
var
    property: string
```

編譯器就會給您這樣的錯誤訊息：

```
E2029 Identifier expected but 'PROPERTY' found
```

警告您使用了關鍵字作為識別字，這是不被允許的。通常當您誤用到關鍵字的時候，您會在編輯器或者編譯程式的時候得到錯誤訊息，當然用到不同的關鍵字會有不同的錯誤訊息出現。當編譯程式發現到關鍵字出現了，而覺得這個關鍵字出現的位置不對，就會依照錯誤關鍵字出現的地點回報錯誤。

在這裡，我不打算把完整的關鍵字列表列出來，僅列出我們在寫程式的時候比較常用到的關鍵字，並把他們依照功能分組列出，即使如此，要完整的涵蓋到這些關鍵字仍舊需要用掉好幾個章節來說明。

筆記 請注意，部分關鍵字會在不同的地方出現，在這裡我謹列出最常見的部分，有些關鍵字可能會被列到兩次。
原因之一，是經過了這麼多年以後，編譯程式的團隊希望不要再導入新的關鍵字，以免舊有的程式反而失效了，所以他們把其中一些關鍵字再度回鍋了。

那麼，我們就開始關鍵字的旅程吧，這些關鍵字有一些可能在您過去寫程式時或者在前面的章節當中有見過，他們就是構成整個應用程式專案的骨幹：

program	標明應用程式專案的名稱
library	標明函式庫專案的名稱
package	標明套件函式庫專案的名稱
unit	標明單元檔的名稱，單元檔也就是程式碼的原始檔
uses	指示當前這個單元檔會參考到哪些單元檔案
interface	單元檔的區段，用來進行宣告
implementation	單元檔的區段，用來放置實作的程式碼
initialization	當程式啟動時，要先被執行的程式碼區段
finalization	當程式結束前，最後要被執行的程式碼區段
begin	宣告一個程式碼區塊的開始
end	宣告一個程式碼區塊的結束

另一組關鍵字則是跟一些基礎資料型別的宣告與變數相關的，茲列出如下:

type	標明開始進入資料型別宣告區段
var	標明開始進入變數宣告區段
const	標明開始進入常數宣告區段
set	定義一個集合變數
string	定義一個字串變數，或者自定的字串型別
array	定義一個陣列型別
record	定義一個複合資料型別
integer	定義一個整數變數
real	定義一個浮點數型態的變數
file	定義一個檔案變數
record	定義一個複合資料型別

筆記 在後面的章節中，我還會介紹更多 Object Pascal 的資料型別。

第三組關鍵字則是介紹 Object Pascal 程式語言的基礎敘述句，例如條件判斷式跟迴圈，也包含了函式(function)跟程序(Procedure):

if	標明一個條件判斷式
then	將條件判斷式與符合條件時執行的程式碼分隔的符號
else	標明條件判斷式中，不符條件時要執行的程式碼
case	標明一個多重選項的條件判斷式
of	把多重選項判斷式的條件與各個選項分隔的符號
for	標明一個固定次數的迴圈開始
to	標明 for 迴圈將變數遞增計算時的最終數值
downto	標明 for 迴圈將變數遞減計算時的最終數值
in	標明在列舉迴圈當中，用來表示要被列舉的組合變數
while	標明一個條件化的迴圈開始
do	把 while 迴圈的條件式與要執行的程式碼做分隔的符號
repeat	標明一個具終止條件的迴圈開始
until	標明repeat迴圈的終止條件
with	標明要針對特定的資料結構進行處理
function	標明一個會回傳執行結果的副程式（名為函式）
procedure	標明一個不會回傳執行結果的副程式（名為程序）
inline	要求編譯程式對函式或程序進行優化
overload	允許同名的函式或程序被重複使用（稱為多載）

以下則是跟類別、物件相關的關鍵字：

class	標明一個新的類別型別
object	用來標明一個就的類別型別（目前已不再使用）
abstract	標明一個抽象類別，表示該類別還沒有完全被定義
sealed	標明一個已封鎖類別，該類別不能再被繼承
interface	標明一個介面型別（這個關鍵字也在第一組當中出現過）
constructor	一個類別或物件的初始方法
destructor	一個類別或物件的清除方法
virtual	一個虛擬方法，在衍生類別中需要被實作出來
override	在衍生類別中，實作虛擬方法的關鍵字
inherited	直接呼叫、引用父類別的方法
private	宣告類別中不能被外界存取的屬性、事件或方法
protected	宣告類別中有條件供外界存取的屬性、事件或方法
public	宣告類別中可以完全被外界存取的屬性、事件或方法
published	宣告類別中特別為了使用者建立的屬性、事件或方法
strict	比private跟protected限制更為嚴格的類別區段
property	被對應到變數或方法的一個符號，稱之為類別的屬性
read	屬性的資料來源
write	屬性的變更方法
nil	表示空物件，在許多有指標類型的語言當中也都有鄉對應的特別符號，在C裡面稱為NULL

還有一小群跟例外處理(我們在第11章裡面會介紹)有關的關鍵字：

try	標明例外處理區塊開始
finally	表示不管例外發生與否，都要被執行的區塊
except	表示當例外發生時，要被執行的程式碼區塊
raise	用來觸發一個例外事件

另外還有一小群關鍵字是用來作為運算用的，我們會在本章稍後的篇幅『算式與運算子』的部份介紹到（有一些進階的運算子則會在後面的章節介紹）：

as	and	div
is	in	mod

```
not           or                shl
shr           xor
```

最後，我們列出一些比較不常用的關鍵字，包含一些不建議使用的舊的關鍵字，在本書的附錄，或者在IDE的協助文件中都可以找到，如果您對這些關鍵字有興趣的話：

```
default       dynamic           export
exports       external          file
forward       goto              index
label         message           name
nodefault     on                out
packed        reintroduce       requires
```

請注意，近幾年來Object Pascal的關鍵字已經很少有新增的了，因為任何新增的關鍵字都有可能會使得已存在的程式碼在使用新版的編譯程式進行編譯時，導致舊有的程式發生編譯錯誤，因為誰也不敢保證程式人員一定不會用到什麼英文字。Object Pascal最近新增的功能都不需要透過關鍵字來達成，例如泛型（generics）與匿名方法（anonymous methods）。

程式結構

您可能曾經把所有的程式碼寫在同一個檔案裡面，就像本章的第一個簡單的文字模式應用程式一樣。而當我們越常開發視覺化程式，就越有機會在專案檔之外使用到第二個原始碼檔案。這『第二個檔案』就被稱為*單元檔*，通常它的副檔名會是PAS（Pascal 原始檔的意思），專案檔的副檔名則會用DPR(Delphi專案檔的意思)，這兩種檔案都會內含有Object Pascal的原始碼。

Object Pascal透過了單元檔或者程式模組的使用提供了延伸性。事實上，單元檔就提供了模組化以及資料封裝的功能，即使沒有使用到物件也一樣。Object Pascal的應用程式通常都是由好幾個單元檔所建立的，包含用來儲存畫面與資料模組的單元檔。事實上，當我們加入一個視覺化的畫面表單到專案裡面，IDE就會幫我們加入一個單元檔，這個單元檔正是對應所加入的視覺化畫面的程式碼。

單元檔無需定義畫面表單，兩者之間會自動被關聯起來，兩者之間的類別、屬性、方法、事件處理常式，都已經被自動連結好，無需我們額外做什麼處

理了。如果您要加入一個新的空白單元檔到專案裡面，這個空白單元檔只需要幾個簡單的關鍵字來宣告幾個必要的區段即可，如下所示：

```
unit Unit1;
interface
implementation
end.
```

單元檔的結構極其簡單，就像上面的範例一樣：

- 首先，單元檔要有一個整個專案不能重複的名字為之命名，同時也當作主檔名（所以上面這個例子存檔時，檔名就會是*Unit1.pas*）
- 其次，單元檔一定要有一個interface區段，用來宣告讓其他單元檔可以使用、存取的資料。
- 第三，單元檔要有implemention區段，用來實作這個單元檔裡面真正的程式碼，這裡的程式碼也可以比interface區段所宣告得來的更多，只是在interface區段沒有宣告的，就只有同一個單元檔的其他程式碼可以使用，不管是方法或屬性都一樣。

單元與程式名稱

如同我提過的，單元的名稱必須跟單元檔的檔名一致，程式名也一樣，要為單元重新命名的話，我們可以用IDE裡面的Save As（另存新檔）來處理，另存的新檔名跟單元名稱也會自動同步變更。當然您也可以直接從檔案總管把檔案直接改名，但是如果改名後，您沒有到單元檔裡面把第一行的單元名稱也一起做修改的話，在編譯程式的時候，就會看到一個錯誤發生（或者只在載入專案的時候，IDE就會告訴你有錯誤了），以下的訊息就是當我們只改了檔名，卻沒有同步修改單元名稱時會發生的錯誤：

```
[DCC Error] E1038 Unit identifier 'Unit3' does not match file name
```

這表示單元名稱也必須符合Pascal識別符號規則，以及檔案系統的命名規則，例如像我們前面提到過的，不能包含空白字元、不能有特殊符號(除了底線)。只要我們使用了符合規範的名稱來為單元命名，就會自動被儲存為合法的檔案名，所以這一點我們不用太過擔心。當然凡事都有例外，就是Unicode的符號，有些符號並不是檔案系統允許我們拿來作為檔名的，就別故意挑戰檔案系統了。

單元名的規則還有一個延伸的規則，就是單元名可以包含. 所以以下列出的單元名稱也都是合法的：

```
unit1
myproject.unit1
mycompany.myproject.unit1
```

這個延伸規則的由來，是因為單元名稱必須是唯一的，而隨著Embarcadero跟第三方開發商所提供的單元檔越來越多，單元檔的名稱就變複雜了，所以目前隨著Delphi開發工具所預載提供的RTL單元以及各種不同功能的單元檔，都可以看到有許多用.來構成單元名稱的情形，例如：

System	代表核心RTL的單元
Data	代表資料庫存取與相關的單元
FMX	代表FireMonkey平台與跨裝置元件的單元
VCL	代表Windows平台視覺元件庫的單元

筆記 您經常都會在單元檔的全名裡面使用到名稱裡面有.的單元，或者函式庫的單元檔。不過也可以只在程式的參考單元當中使用到整個單元名稱的最後部分（這也是為了讓舊版的程式能夠相容新版的編譯程式），您只需要設定專案選項中的對應項目即可，這個設定選項的名稱是"Unit scope names"，它是一個以分號做項目區隔的清單。不過請注意，使用這個功能相對的會讓編譯的速度比使用完整單元名稱的時候變慢許多。

更多關於單元檔的結構

除了 interface 跟 implementation 這兩個區段之外，每個單元還可以有 initialization 跟 finalization 這兩個非必要的區段。Initialization 是用來處理該單元被執行時最開始的程式碼，而 finalization 則是用來處理該單元在程式結束時要處理的程式碼。

筆記 您也可以在類別的建構方法(constructor)當中加入 initialization 程式碼，Object Pascal 的許多最新功能在第 12 章會介紹到，使用類別的建構方法可以幫助連結程式移除非必要的程式碼，這也是為什麼建議大家使用類別的建構方法跟解構方法(destructor)，而比較不建議使用 initialization 跟 finalization 區段的原因。在過去的歷史中，initialization 區段還是需要依靠 begin 這個關鍵字來進行宣告的，begin 的類似用法仍然是專案程式碼的標準。

換句話說，單元的結構，包含了所有可能的區段與一些簡單的元素，應該長得像下面這個範例程式碼：

```
unit unitName;
interface
// 其他我們在本單元會引入的單元名稱，在 interface 這個區段中宣告
uses
    unitA, unitB, unitC;
// 要公告周知的型別定義
type
    newType = TypeDefinition;
// 要公告周知的常數
const
   Zero = 0;
// 全域變數
var
   Total: Integer;
// 要公告周知的函式與程序
procedure MyProc;
implementation
// 其他在 implementation 區段我們會引入的單元名稱
uses
   unitD, unitE;
// 不對其它單元檔告知的全域變數
var
   PartialTotal: Integer;
// 所有被公告的函式都必須在此被實作
procedure MyProc;  begin
// ... MyProc 這個程序的程式碼
end;
 initialization
// 非必要的 initialization 區段的程式碼
 finalization
// 非必要的 finalization 區段程式碼
end.
```

一個單元的 interface 區段存在的意義，是為了告知其它單元，這個單元包含什麼，能為其它單元或專案提供什麼。而 implementation 區段則包

含了所有其它單元都無法得知的程式碼。這也是Object Pascal之所以可以提供資訊封裝，而不需要靠類別或物件就能達成該功能的原因。

我們可以發現，單元的 interface 區段可以宣告不少型別各異的元素，包含程序、函式、全域變數，以及資料型別。資料型別當然是最常出現在這個程式區段中的。IDE環境會自動放一段新的類別宣告，當我們建立一個新的視覺化畫面表單，然而，宣告表單定義並不是Object Pascal當中包含單元這個功能的唯一理由。我們也可以在單元當中只有程式碼，只有函式與程序（就像傳統 Pascal 程式的作法），甚至在單元裡面有新的類別，但不用參考其它的表單的或者視覺化元件。

Uses 條文

Uses條文位於interface區段的開頭部分，是用來標示我們在該單元中需要參考的其它單元名稱。是標明我們在這個單元當中，為了定義資料型別時，需要參考的其它單元，而參考的程式也限於那些單元的資料型別，例如我們在畫面表單當中定義的元件。

第二個uses條文是出現在implementation區段的開頭，是標明我們只在程式碼實作階段需要參考的單元檔。當您只需要參考其它單元的程式碼，例如副程式、方法，我們就得在implememtation區段當中的uses來標明這些單元的名稱。在uses條文中被標明要參考的單元檔，都必須位於專案目錄當中，或者IDE環境的搜尋路徑當中，這樣編譯才不會出問題。

提示 您可以在 Project Options 的選單當中設定搜尋路徑，系統也會先搜尋已經位於 Library Path 裡面的檔案，這個設定算是 IDE 的全域設定值。

C++的程式人員們請注意，uses敘述跟C++的include敘述句並不對等，uses敘述句的效應只會把預先編譯的單元的interface部分引入。Implementation區段的程式碼會在該單元實際被編譯器處理的時候才被考慮，被引入的單元檔，可以以原始碼的格式(PAS檔)或者編譯過的二進位檔(DCU檔)的形式存在前述的目錄或路徑中。

雖然在Object Pascal當中並不常用到，但Object Pascal的編譯器設定當中也有一個類似C/C++ include的編譯器設定，名為$INCLUDE，這些特別的引入檔會被部分需要共用編譯器設定的函式庫或者在許多單元檔要共用其它設定的時候才被用到的，而且通常會使用INC這個特別的副檔名，這個編譯器設定會在本章的最後介紹。

筆記 請注意，Object Pascal 的二進位編譯檔(DCU)只會在使用相同版本的編譯器與系統函式庫時相容。用舊版的編譯程式編譯的二進位檔案，通常無法與後來版本的編譯器相容，這一點不可不察。

單元與界限

在Object Pascal當中，單元正是資料封裝與程式碼界限的關鍵，在這個規範下，單元能提供的程式碼界限甚至比類別中private或public關鍵字能提供的還更重要。一個識別碼(例如變數、程序、函式或資料型別)的界限，是表示這個識別碼能夠被其它程式碼存取的範圍,所以也被稱為該識別碼的可視範圍。基本規則就是只有在該識別碼的界限中，它才是有意義的，因此只有在這個單元當中的程序、函式才能使用這個識別碼，我們無法在識別碼的界限之外使用它。

筆記 請注意，Object Pascal 跟 C、C++都不一樣，Object Pascal 在一般程式碼區塊裡面不允許變數、常數的宣告。當我們已經進入了 begin-end 的程式碼區塊範圍之後，在這裡面就不能夠再宣告任何變數了。

通常一個識別碼只有在它被定義之後才能使用。但在Object Pascal當中，也有方法在一個識別碼被完整定義之前先進行宣告,但我們在完整考慮了定義與宣告的規則後，應該可以發現其實他還是遵循著Object Pascal的基本規範的。

假設把整個程式的程式碼寫在單一一個檔案裡面是有意義的，那麼，這個規則會怎麼修正好讓我們在使用多個單元檔的時候能夠遵循呢？簡單的說，當我們透過uses條文把其它單元引入的時候，在被引入的單元中，interface區塊所宣告的所有識別碼也在新的單元檔裡面變成可以被存取的了。

反向理解一下，如果您在interface區段宣告了一個識別碼（可能是型別、函式、類別、變數等等），所有引用現在建立的這個單元的其它所有程式碼也都可以看到這個剛宣告的識別碼了。但如果您是在implementation區段中宣告這個識別碼的話,則這個識別碼就只能在自己這個單元檔中被看見,其餘引用這個單元的程式碼都無法看見了,我們可以把它理解成區域識別碼，就像區域變數那樣。

把單元檔當成命名空間來使用

我們已經看過了uses敘述句，它是讓單元檔能夠看見其它引入的單元檔相關識別碼的標準技術。這時您可以存取這個單元中的所有定義，但有時在兩個單元檔裡面可能宣告了相同的識別碼，例如您可能有兩個類別，或者兩個副程式使用了完全相同的名字。

在這種情形下，我們可以簡單的把單元名稱前置在這個被重複使用的識別碼之前。舉個例子來說，您可以引入在Calc這個單元當中名為ComputeTotal的程序，這時我們可以把它寫成Calc.ComputeTotal，IDE不常要求我們一定要這麼寫，但當我們在兩個不同單元中有相同的識別碼時，這樣寫可以避免重複，以及讓編譯程式不會誤解。

然而如果您曾經深入看過系統或第三方元件的程式，您應該會發現許多函式跟類別的名稱重複，最常見的例子就是在不同平台的視覺化元件中，常常有相同命名的元件，當您深入看到TForm或TControl的程式碼時，裡面有很多類別或函式會依據您所引入的單元來決定要執行哪一個函式。

如果使用了相同名稱的兩個單元，正好都被您的單元檔引入了，最後被引入的那個單元會搶到該名稱的使用權，編譯程式也就會把該名稱直接對應到最後被引入的單元檔的識別碼去，如果您無法避免這種情形的話，請一定在重複識別碼名稱的前面加入該單元檔的完整名字，這樣編譯程式就不會弄錯了。

筆記

在 Delphi XE5 之後，FireMonkey 平台提供了一個名為 TPath 的類別，讓我們可以處理跨平台的常用路徑，因此我們可以透過 TPath.GetDocumentPath 來取得各平台上面的文件路徑。

但 TPath 就是一個重複名稱的識別碼，我們要使用的這個功能，是由 System.IOUtils 這個單元當中的 TPath 所提供的，因此譯者最近在用到 TPath 的時候，都會自己寫成 System.IOUtils.TPath，這樣編譯程式就不會混淆了。

程式檔案

我們在前面的篇幅已經看過了，Delphi的應用程式檔案會包含兩種程式碼檔案：一個是會出現多次的單元檔，而另外一個是只會出現一次的程式檔，或者我們也可以叫它專案檔。單元檔可以看成是第二層的檔案，所有單元檔都會被扮演主要階層角色的專案檔所引入。

理論上這是對的，在實務上，專案檔通常是自動產生出來的檔案，而且有其被侷限的角色。專案檔只用來啟動這個應用程式、通常會建立、執行主要的畫面表單（如果應用程式是視覺化應用程式時）。專案檔的內容也可以手動編輯，但是當我們修改專案設定的時候，這個檔案的內容就會被自動修改（就跟應用程式當中的其它物件跟畫面表單一樣）

專案檔的結構通常比其他單元檔案來的簡單，以下就是一個簡單的專案檔，這個內容會是由IDE幫我們自動產生：

```
program Project1;
uses
    FMX.Forms,
    Unit1 in 'Unit1.PAS' {Form1};
begin
    Application.Initialize;
    Application.CreateForm (TForm1, Form1);
    Application.Run;
end.
```

從上面的程式碼我們可以看到，它只有一個簡單的uses區段，以及透過begin-end標明的應用程式主要程式碼，這個程式的uses敘述句非常重要，因為它們會被用來進行編譯、連結成應用程式執行檔。

筆記 在專案檔裡面的單元列表對應了在 IDE 當中專案管理員畫面的單元列表，當我們在 IDE 裡面加入一個單元到這個專案裡面的時候，這個單元的名稱也會自動被加入到這個專案檔的 uses 區段裡面。當然，如果我們從 IDE 裡面刪除了一個單元檔，專案檔的 uses 區段也會立即有反應。
反之如果我們直接編輯專案檔，從裡面直接刪除某個 uses 裡面的單元檔，在 IDE 裡面的設定畫面也會同時立即有反應。

編譯程式設定

程式結構裡面另一個特殊的部分（跟其他實際的程式碼相比），就是編譯程式設定了，我們稍早曾經提到過，這些特殊的指令是給編譯程式用的，會以以下的格式撰寫：

```
{$X+}
```

有些編譯程式設定只是一個簡單的字元，就像上面這個例子，用加號或減號來表示該設定是有效或者無效，大多數的設定會有一個長一點或者能夠被判讀的寫法，使用ON與OFF來表示有效或者無效，而部分設定值則只有較長的寫法，沒有像上例這種簡寫。

編譯程式設定通常不是直接把程式碼編譯成二進位，而是告訴編譯程式，這個設定值出現之後，要把編譯程式當中的部分設定進行調整以後再行編譯之後的程式碼，大多數的時候，我們可以透過修改IDE當諸的專案設定值來調整這些設定，即使有些情況下，我們只需要對一個單元或一部分的程式碼進行編譯程式設定值的調整。

在後續的篇幅中，只要提到相關的程式語言功能，我也會介紹一下相關的編譯程式設定，在這個章節中，我只稍微介紹一些跟程式流程相關的編譯程式設定，例如條件化定義(Conditional Defines)與引入(includes)

條件化定義(Conditional Defines)

條件化定義會透過$IFDEF讓我們可以告訴編譯程式使用哪一部分的程式碼，或者忽略它。通常我們會在定義識別碼或者引入單元的時候用到這個功能。它們可能會基於部分已經被定義過的識別碼或者常數。這些被定義的識別碼可能是系統預先定義的（例如各平台的變數），可能是一個特殊的專案設定值，也可能是我們用另一個編譯程式設定句：$DEFINE來定義的：

```
{$DEFINE TEST} ...
{$IFDEF TEST}
// 這部份的程式碼會被編譯
{$ENDIF}
{$IFNDEF TEST}
// 這部份的程式碼不會被編譯
{$ENDIF}
```

我們可以加上$ELSE把一個條件的兩種情形做出區隔，比較彈性的作法是使用$IF這個句子，記得結束時要加上$IFEND，這樣可以把編譯程式設定的敘述句寫的比較像一般Object Pascal程式碼。所以我們也常定義一個常數，然後用一個參考該常數的判斷式來搭配編譯程式設定的判斷句，以下就是以編譯程式版本作為範例的寫法，讓不同版本的編譯程式使用一些通用的系統定義。

編譯程式版本(Compiler Versions)

每個版本的Delphi編譯程式都有一個特別的定義值，我們可以依此進行判斷，檢查我們的程式是否能使用特定版本的編譯程式。這有賴我們使用了後面介紹的一些功能，但希望在編譯時先檢查一下編譯程式是否能夠處理這些功能的程式碼。

如果我們需要最近幾版的Delphi來處理特定的程式碼，我們可以在$IFDEF後面判斷以下的版本代號：

Delphi 2007	VER180
Delphi XE	VER220
Delphi XE2	VER230
Delphi XE4	VER250
Delphi XE5	VER260
Appmethod Spring 2014	VER260
Delphi XE6	VER270
Appmethod June 2014	VER270
Delphi XE7	VER280
Appmethod September 2014	VER280

後面的數字是該版的編譯器版本代號（例如26就是Delphi XE5），這個號碼並不會區分Appmethod或Delphi，但回推到第一版的Pascal編譯器則是由Borland所推出的。

我們也可以在$IF判斷式裡面使用這些內部的代號常數，這樣我們就可以直接用>=來判斷編譯程式是否符合特定版本的需求，版本的常數名稱是CompilerVersion，在Delphi XE5裡面，這個常數是一個浮點數，數值是26.0，所以範例如下：

```
{$IF CompilerVersion >= 26}
  // 需要 Delphi XE5 或更新的版本編譯器才能編譯的程式碼
{$IFEND}
```

舉一反三，我們也可以使用一些系統常數，例如用來判斷是哪個作業系統平台，萬一我們需要使用到該平台特定的程式功能：

Windows系統（32或64位元都一樣） MSWINDOWS

Mac OS X	MACOS
iOS	IOS
Android	Android

以下是簡單的程式片段，使用了上述的作業系統定義，它們是HelloPlatform範例的部份程式：

```
{$IFDEF IOS}
  ShowMessage ('Running on iOS');
{$ENDIF}
{$IFDEF ANDROID}
  ShowMessage ('Running on Android');
{$ENDIF}
```

引入檔(Include Files)

我在此想介紹的另一個編譯程式設定指令，是$INCLUDE這個指令，我們在前面介紹uses的時候已經提到過了，這個指令讓我們可以參照、引入特定程式檔案中的一部分程式碼，通常這個用法會被用來在不同的檔案中引入相同的程式碼，例如某段程式碼定義了一些編譯程式設定，而我們在使用一個單元時，只需要引入一部分的程式碼，當我們引入一個檔案時，該檔案所引入的所有單元都會一起被編譯（這就是為什麼我們應該避免在引入檔裡面加入新的識別符號的原因）

換句話說，我們應該不要在引入檔裡面加入任何程式相關的元素與定義（這跟C語言的例子正好相反），相關的程式元素與定義都應該在單元檔裡面來處理，所以我們到底該如何使用引入檔呢？好的例子是在引入檔裡面寫入一些我們希望在大多數的單元檔中都要用到的編譯程式設定，或者特殊的額外定義。

大型函式庫通常會使用引入檔來達成前述的目的，例如FireDAC函式庫，這是已經成為系統預設函式庫之一的用來處理資料庫相關的函式庫，另一個例子則是系統的執行階段函式庫(RunTime Library, 又簡寫成RTL)也在各個作業系統中使用了獨立的引入檔，而在編譯程式中會隨著我們所選擇的作業平台單獨套用該平台的設定。