



解決方案白皮書

---

# 使用全新的 Delphi 程式碼撰寫風格 與架構

Delphi 2009 語言特性評論

2008 年 12 月

---

公司總部  
100 California Street, 12th Floor  
San Francisco, California 94111

EMEA 總部  
York House  
18 York Road  
Maidenhead, Berkshire  
SL6 1SF, United Kingdom

亞太地區總部  
L7.313 La Trobe Street  
Melbourne VIC 3000  
Australia

## 簡介：DELPHI 語言

Delphi 語言（較常見的名稱是 Object Pascal）是一種現代化的強力類型檢查和物件導向語言，特色包括單一繼承和物件參照模型。最近幾年，這種語言擴增了 record 方法、記錄運算子負載、類別資料、巢狀類型、密封類別、final 方法，以及許多其他的相關功能。最令人驚奇的擴充則可能是類別輔助程式，這是用來在現有的類別中加入新方法或是取代部分現有方法的技術。

但是，在 Delphi 2009 中加入編譯器的全新功能，則更是意義重大。除了延伸字串類型以支援 Unicode 之外，最新版 Delphi 引進了通用資料類型、匿名方法，以及若干個其他較為「次要」但又非常有趣的功能。

## 泛型簡介

在通用類別的第一個例子中，我實作了鍵-值對組資料結構。以下的第一個程式碼片段（以傳統方式撰寫）顯示資料結構，含有一個用來存放值的物件：

```
type
  TKeyValue = class
  private
    FKey: string;
    FValue: TObject;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: TObject);
  public
    property Key: string read FKey write SetKey;
    property Value: TObject read FValue write SetValue;
  end;
```

若要使用這個類別，您可以建立物件、設定物件的鍵和值，並且使用物件，如下列程式碼片段所示：

```
// FormCreate
kv := TKeyValue.Create;

// Button1Click
kv.Key := 'mykey';
kv.Value := Sender;

// Button2Click
kv.Value := self; // the form

// Button3Click
ShowMessage ('[' + kv.Key + ', ' +
  kv.Value.ClassName + ']');
```

## 使用全新的 Delphi 程式碼撰寫風格與架構

泛型可讓您對值使用遠較為廣泛的定義，但是那並不是重點。全然不同的（我們將會加以瞭解）是在產生鍵-值通用類別之後，它就變成限制為給定資料類型的特定類別。這會讓您的程式碼類型更為安全，但是這個部分已經超出我要說明的範圍。讓我們先從用來定義通用類別的語法開始：

```
type
  TValue<T> = class
  private
    FKey: string;
    FValue: T;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: T);
  public
    property Key: string read FKey write SetKey;
    property Value: T read FValue write SetValue;
  end;
```

在這個類別定義中，有一個以替代符號 **T** 表示的未指定類型。通用 **TValue<T>** 類別使用這個未指定的類型作為屬性值欄位和 setter 方法參數。這些方法的定義方式如同平常一樣，不過，即使它們與通用類型相關，其定義仍然包含類別的完整名稱，其中包括通用類型：

```
procedure TValue<T>.SetKey(const Value: string);
begin
  FKey := Value;
end;

procedure TValue<T>.SetValue(const Value: T);
begin
  FValue := Value;
end;
```

若要改用類別，則您必須完整界定類別，提供通用類別的實際值。例如，現在您可以撰寫下列程式碼，將鍵-值物件主控按鈕宣告為值：

```
| kv: TValue<TButton>;
```

建立執行個體時也必須有完整名稱，因為這是實際的類型名稱（通用、未產生的類型名稱如同類型建構機制）。

## 使用全新的 Delphi 程式碼撰寫風格與架構

對鍵-值對組使用特定類型的值會使程式碼更加健全，因為現在您在鍵-值對組中只能加入 **TButton** (或衍生) 物件，並且可以使用所擷取物件的不同方法。以下是部分程式碼片段：

```
// FormCreate
kv := TKeyValuePair.Create;

// Button1Click
kv.Key := 'mykey';
kv.Value := Sender as TButton;

// Button2Click
kv.Value := Sender as TButton; // was "self"

// Button3Click
ShowMessage ('[' + kv.Key + ', ' + kv.Value.Name + ']');
```

在舊版程式碼中指定通用物件時，我們可以加入按鈕或表單。目前只能使用按鈕，這是編譯器所強制規定的規則。同樣地，我們可以使用元件 **Name** 或 **TButton** 的任何其他屬性，而不是輸出中的通用 **kv.Value.ClassName**。

當然，我們也可以模仿原始的程式，方法為將鍵-值對組宣告為：

```
| kv0: TKeyValuePair<TObject>;
```

在這個版本的通用鍵-值對組類別中，我們加入任何物件作為值。不過，我們對擷取的物件所能做的也很有限，除非是將它們轉換為更為特定的類型。要找到一個好的平衡點，您可能希望在特定的按鈕與任何物件之間設定某個項目，並要求值成為一個元件：

```
| kvc: TKeyValuePair<TComponent>;
```

最後，我們可以建立通用鍵-值對組類別的執行個體，此執行個體不儲存物件值而是儲存整數，如下所示：

```
| kvi: TKeyValuePair<Integer>;
```

### 泛型的類型規則

當您宣告泛型的執行個體時，在所有後續的作業中，編譯器會強制此類型取得特定的版本。因此，如果您有如下的通用類別：

```
type
  TSimpleGeneric<T> = class
    Value: T;
  end;
```

當您將某特定物件宣告為給定的類型時，就不能將不同的類型指定給 `value` 欄位。在給定下列兩個物件的情況下，以下的部分指定是不正確的：

```
var
  sg1: TSimpleGeneric<string>;
  sg2: TSimpleGeneric<Integer>;
begin
  sg1 := TSimpleGeneric<string>.Create;
  sg2 := TSimpleGeneric<Integer>.Create;

  sg1.Value := 'foo';
  sg1.Value := 10; // Error
  // E2010 Incompatible types: 'string' and 'Integer'

  sg2.Value := 'foo'; // Error
  // E2010 Incompatible types: 'Integer' and 'string'
  sg2.Value := 10;
```

在為特定類型定義泛型宣告之後，編譯器會強制執行此定義，也就是按照 Object Pascal 的強型別語言的預期方式。整體而言，類型檢查也適合通用類型。在為物件指定通用參數時，您無法指定類似的泛型（依據不同的和不相容的類型執行個體）給它。如果這很令人困惑，以下範例應該有助於釐清：

```
sg1 := TSimpleGeneric<Integer>.Create; // Error
// E2010 Incompatible types:
// 'TSimpleGeneric<System.string>'
// and 'TSimpleGeneric<System.Integer>'
```

類型相容性規則是按結構而不是按類型名稱，但是您無法將泛型執行個體指定給不同的和不相容的類型。

## DELPHI 中的泛型

在前例中，我們看到了您可以如何定義和使用通用類別，這種類別是 Object Pascal 語言自 Delphi 3 引進介面後其中一個最具意義的擴充。我決定先用範例介紹此功能，然後再探究技術細節，這些細節相當繁複也關係重大。在從語言的觀點介紹泛型之後，我們將會回來說明更多範例，其中包括通用容器類別的使用和定義，這是在語言中加入這項技術的其中一個主因。我們已看到，現在當您在 Delphi 2009 中定義類別時，可以在角括弧內加入額外的「參數」來表示稍後提供類型的位置：

```
type
  TMyClass <T> = class
  ..
end;
```

泛型可用來作為欄位的類型（如同我在上例中所做的）、作為屬性的類型、作為參數的類型或函式的傳回值等。

## 使用全新的 Delphi 程式碼撰寫風格與架構

請注意，並不是一定要對區域欄位 (或陣列) 使用此類型，因為有些情況是泛型僅用來作為結果、參數，或者也不是用於類別的宣告，而僅僅用於其部分方法的定義。

這種形式的延伸或通用類型宣告適用於類別，也適用於記錄 (提醒您，在最新版的 Delphi 中，記錄也可以有方法和負載運算子)。有別於 C++，您無法宣告通用全域函式，但是您可以使用單一類別方法宣告通用類別，這幾乎並沒有兩樣。

就像其他的靜態語言一樣，在 Delphi 中實作泛型並不是根據執行時期架構，而是由編譯器和連結器來處理，且幾乎沒有任何事項要留待執行時期機制處理。有別於虛擬函式呼叫在執行時期繫結，系統會為您產生的每種範本類型產生範本方法一次，並且是在編譯時期產生！我們將會看到此方法可能會有的缺點，但在正面的一方，意味著通用類別與一般類別一樣有效率，甚至超過降低執行時期 costs 所需的效率。

## 泛型函式

到目前為止，我們已知泛型定義的最大問題是，您對泛型的物件所能做的非常有限。有兩種方法可用來克服這項限制。第一種方法是利用執行時期程式庫的幾個特殊函式，這些函式專門支援泛型。第二種方法 (這種方法遠遠較為強大) 是將通用類別定義為具有可以使用之類型上的限制。

我將在本節中著重於第一個部分，而在下一節著重於限制。如我所提及的，有一個全新的函式以及兩個傳統的函式，專為適用於泛型定義的參數類型 (**T**) 而修改的函式：

- **Default(T)** 是一個全新的函式，會針對目前類型傳回空白、「零值」(zero value) 或空值；這可為零、空字串、nil 等。
- **TypeInfo (T)** 會傳回指標，指向現行版泛型的執行時期資訊；
- **sizeof (T)** 會傳回類型的記憶體大小 (以位元組為單位)。

下列範例中有一個通用類別，顯示三個泛型函式執行的情況：

```
type
  TSampleClass <T> = class
  private
    data: T;
  public
    procedure Zero;
    function GetDataSize: Integer;
    function GetDataName: string;
  end;

function TSampleClass<T>.GetDataSize: Integer;
begin
  Result := SizeOf (T);
```

使用全新的 Delphi 程式碼撰寫風格與架構

```
end;  
  
function TSampleClass<T>.GetDataName: string;  
begin  
    Result := GetTypeName (TypeInfo (T));  
end;  
  
procedure TSampleClass<T>.Zero;  
begin  
    data := Default (T);  
end;
```

在 `GetDataName` 方法中，我使用了 `GetTypeName` 函式 (或 `TypeInfo` 單元) 而非直接存取資料結構，因為它從存放類型名稱的已編碼 `ShortStringValue` 執行適當的 UTF-8 轉換。

在給定上述宣告的情況下，您可以編譯下列測試程式碼，此程式碼在三個不同的泛型執行個體上重複其本身三次。我省略了重複的程式碼，但是保留了用來存取 `data` 欄位的陳述式，因為它們會依實際的類型而改變：

```
var  
    t1: TSampleClass<Integer>;  
    t2: TSampleClass<string>;  
    t3: TSampleClass<double>;  
begin  
    t1 := TSampleClass<Integer>.Create;  
    t1.Zero;  
    Log ('TSampleClass<Integer>');  
    Log ('data: ' + IntToStr (t1.data));  
    Log ('type: ' + t1.GetDataName);  
    Log ('size: ' + IntToStr (t1.GetDataSize));  
  
    t2 := TSampleClass<string>.Create;  
    ...  
    Log ('data: ' + t2.data);  
  
    t3 := TSampleClass<double>.Create;  
    ...  
    Log ('data: ' + FloatToStr (t3.data));
```

執行此程式碼會產生下列輸出：

```
TSampleClass<Integer>  
data: 0  
type: Integer  
size: 4  
TSampleClass<string>  
data:  
type: string  
size: 4  
TSampleClass<double>
```

```
data: 0  
type: Double  
size: 8
```

要特別注意的是，您可以在通用類別的競爭之外的特定類型上使用泛型函式。例如，您可以撰寫：

```
var  
  I: Integer;  
  s: string;  
begin  
  I := Default (Integer);  
  Log ('Default Integer': + IntToStr (I));  
  
  s := Default (string);  
  Log ('Default String': + s);  
  
  Log ('TypeInfo String': +  
    GetTypeName (TypeInfo (string));
```

雖然呼叫 `Default` 在 Delphi 2009 中是全新的作法 (儘管在範本之外並非那麼管用)，但是在結尾處的 `TypeInfo` 已經可以在舊版 Delphi 中使用。以下是普遍的輸出：

```
Default Integer: 0  
Default String:  
TypeInfo String: string
```

## 泛型限制

如前面所看到的，通用類別的方法對於泛型值所能做的相當有限。您可以傳遞它 (也就是說，指定它)，並且執行我剛提過的泛型函式所容許的有限作業。

爲了能夠對類別的泛型執行一些實際的作業，您通常應對其設定限制。例如，當您將泛型限制爲必須是類別時，編譯器將可讓您在其上呼叫所有的 `TObject` 方法。您也可以進一步限制類別必須是給定階層的一部分或是實行特定的介面。

### 類別限制

您可以採用的最簡單限制是類別限制。若要使用它，您應將泛型宣告爲：

```
type  
  TSampleClass <T: class> = class
```

指定類別限制之後，表示您只能使用物件類型作爲泛型。

## 使用全新的 Delphi 程式碼撰寫風格與架構

使用下列宣告：

```
type
  TSampleClass <T: class> = class
  private
    data: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (t: T);
  end;
```

您可以建立前兩個執行個體，但是不能建立第三個：

```
sample1: TSampleClass<TButton>;
sample2: TSampleClass<TStrings>;
sample3: TSampleClass<Integer>; // Error
```

這最後一個宣告所導致的編譯器錯誤會是：

```
E2511 Type parameter 'T' must be a class type
```

指出此限制的優點是什麼？現在，在通用類別方法中，您可以呼叫任何的 `TObject` 方法，其中甚至包括虛擬的方法！這是 `TSampleClass` 通用類別的 `One` 方法：

```
procedure TSampleClass<T>.One;
begin
  if Assigned (data) then
  begin
    Form30.Log('ClassName: ' + data.ClassName);
    Form30.Log('Size: ' + IntToStr (data.InstanceSize));
    Form30.Log('ToString: ' + data.ToString);
  end;
end;
```

您可以嘗試此程式，瞭解它定義和使用若干個泛型執行個體時的實際效果，如下列程式碼片段所示：

```
var
  sample1: TSampleClass<TButton>;
begin
  sample1 := TSampleClass<TButton>.Create;
  try
    sample1.SetT (Sender as TButton);
    sample1.One;
  finally
    sample1.Free;
  end;
```

## 使用全新的 Delphi 程式碼撰寫風格與架構

請注意，藉由使用自訂的 `ToString` 方法宣告類別，當資料物件為特定類型時，無論提供給泛型的實際類型為何，都會呼叫此自訂版本。換句話說，如果您有 `TButton` 子代，例如：

```
type
  TMyButton = class (TButton)
  public
    function ToString: string; override;
  end;
```

您可以按值 `TSampleClass<TButton>` 傳遞此物件或是定義泛型的特定執行個體，而在這兩種情況下，呼叫 `One` 最終會執行特定版本的 `ToString`：

```
var
  sample1: TSampleClass<TButton>;
  sample2: TSampleClass<TMyButton>;
  mb: TMyButton;
begin
  ...
  sample1.SetT (mb);
  sample1.One;
  sample2.SetT (mb);
  sample2.One;
```

類似類別限制，您可以有宣告為如下的記錄限制：

```
type
  TSampleRec <T: record> = class
```

不過，不同記錄的共同點極少（沒有共同始組），因此這項宣告有點受制。

### 特定類別限制

如果您的通用類別必須使用類別的某個特定子集（某個特定的階層），則您應重新排序以根據給定的基本類別來指定限制。例如，如果您宣告：

```
type
  TCompClass <T: TComponent> = class
```

此通用類別的執行個體只能套用到元件類別，也就是任何 `TComponent` 子代類別。這可讓您擁有極特定的泛型（沒錯，這聽起來是很特別，但是實際上就是如此），同時編譯器會讓您使用 `TComponent` 類別的所有方法來處理泛型。

如果這看來似乎過於強大，請再想一想。如果您考量使用繼承和類型相容規則可以達到的結果，則您可能能夠使用

傳統的物件導向技術處理相同的問題，而無需使用通用類別。我並不是說特定的類別限制一點用處都沒有，而是它肯定不會像較高層類別限制或 (我發現相當有意思的部分) 介面型限制那麼強大。

## 介面限制

一般而言，僅接受那些將給定的介面作為類型參數來實作的類別會較有彈性，而不是將通用類別限制為給定的類別。這樣就能夠在泛型的執行個體上呼叫介面。

上述針對泛型使用介面限制在 .NET 架構中也非常普遍。我們就先開始舉個例子。首先，我們必須宣告介面：

```
type
  IGetValue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    property value: Integer
      read GetValue write SetValue;
  end;
```

接下來，我們可以定義實作它的類別：

```
type
  TGetValue = class (TSingletonImplementation, IGetValue)
  private
    fValue: Integer;
  public
    constructor Create (Value: Integer = 0);
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
  end;
```

當您將通用類別定義為限於實作給中介面的類型時，事情會開始變得有趣：

```
type
  TInftClass <T: IGetValue> = class
  private
    val1, val2: T; // or IGetValue
  public
    procedure Set1 (val: T);
    procedure Set2 (val: T);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
  end;
```

使用全新的 Delphi 程式碼撰寫風格與架構

Notice that in the code for the generic methods of this class we can write:

```
function TInftClass<T>.GetMin: Integer;
begin
    Result := min (val1.GetValue, val2.GetValue);
end;

procedure TInftClass<T>.IncreaseByTen;
begin
    val1.SetValue (val1.GetValue + 10);
    val2.Value := val2.Value + 10;
end;
```

有了以上這些定義後，這時我們可以如下使用通用類別：

```
procedure
    TFormIntfConstraint.btnValueClick( Sender:
        TObject);
var
    iClass: TInftClass<TGetValue>;
begin
    iClass := TInftClass<TGetValue>.Create;
    iClass.Set1 (TGetValue.Create (5));
    iClass.Set2 (TGetValue.Create (25));
    Log ('Average: ' + IntToStr (iClass.GetAverage));
    iClass.IncreaseByTen;
    Log ('Min: ' + IntToStr (iClass.GetMin));
end;
```

爲了展示此通用類別的彈性，我爲介面建立了另一個全然不同的實作：

```
TButtonValue = class (TButton, IGetValue)
public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
        Parent: TWinControl): TButtonValue;
end;

{ TButtonValue }

function TButtonValue.GetValue: Integer;
begin
    Result := Left;
end;

procedure TButtonValue.SetValue(Value: Integer);
begin
    Left := Value;
end;
```

## 使用全新的 Delphi 程式碼撰寫風格與架構

類別函式會在 `Parent` 控制項內的隨機位置上建立一個按鈕，並用於下列示範程式碼：

```
procedure
  TFormIntfConstraint.btnValueButtonClick( Sender:
    TObject);
var
  iClass: TInftClass<TButtonValue>;
begin
  iClass := TInftClass<TButtonValue>.Create;
  iClass.Set1 (TButtonValue.MakeTButtonValue (
    self, ScrollBox1));
  iClass.Set2 (TButtonValue.MakeTButtonValue (
    self, ScrollBox1));
  Log ('Average: ' + IntToStr (iClass.GetAverage));
  Log ('Min: ' + IntToStr (iClass.GetMin));
  iClass.IncreaseByTen;
  Log ('New Average: ' + IntToStr (iClass.GetAverage));
end;
```

### 介面參照與泛型介面限制

在上例中，我定義了一個通用類別，此類別能與任何實作給中介面的物件搭配使用。依據介面參照建立標準(非通用)類別也可能能夠獲得類似的效果。事實上，我可能定義如下的類別：

```
type
  TPlainInftClass = class
  private
    val1, val2: IGetValue;
  public
    procedure Set1 (val: IGetValue);
    procedure Set2 (val: IGetValue);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
  end;
```

這兩種方法之間有何差異？其中一項差異在於，在上述類別中，您可以將兩個不同類型的物件傳遞到 setter 方法，前提是它們的類別都實作給定的介面。然而，在泛型版本中，您只能傳遞(傳遞給通用類別的任何給定的執行個體)給定類型的物件。因此，泛型版本較為保守，就類型檢查方面來說則相當嚴謹。

我的看法是，關鍵的差異在於使用介面型版本意味著使用 Delphi 參照計數機制，而在使用泛型版本時，類別是處理給定類型的一般物件，並不涉及參照計數。此外，泛型版本可能有多項限制(例如建構函式限制)，並讓您使用各種通用函式(例如要求泛型的實際類型)。這是使用介面時的一些限制。(事實上，使用介面時，您無法存取基本 `TObject` 方法)。

換句話說，帶有介面限制的通用類別可以讓您享有介面的優點，而沒有其繁瑣的事項。還有，要注意這兩種方法在大部分的情況下是相等的相當重要，而介面型解決方案比起其他解決方案則較具彈性。

## 使用預先定義的通用容器

自早期 C++ 語言中的範本之後，其中一個最明顯使用通用類別的就是通用容器、清單或容器的定義。事實上，當您定義物件清單時，例如 Delphi 自己的 `TObjectList`，您就擁有了有可能存放任何類別物件的清單。無論是使用繼承還是合成，您的確是可以定義特定類型的自訂容器，但是這種方法卻是令人厭煩 (而且也有發生錯誤的傾向)。

Delphi 2009 定義了一小組通用容器類別，可以在新的 `Generics.Collections` 單元中找到。四個核心的容器類別全都以獨立的方式實作 (它們並不彼此繼承)、全部以類似的方式實作 (使用動態陣列)，且全部都對映到 `Containers` 單元相對應的非通用容器類別：

```
type
  TList<T> = class
  TQueue<T> = class
  TStack<T> = class
  TDictionary<TKey,TValue> = class
```

從這些類別的名稱就能明顯看出它們之間的邏輯差異。一個測試它們的好方法，是算出您必須在使用非通用容器類別的現有程式碼上執行多少變更。舉例而言，我取用了 *Mastering Delphi 2005* 這本書的實際範例程式，並將它轉換成使用泛型。

### 使用 `TList<T>`

範例程式有一個定義 `TDate` 類別的單元以及用來參照日期 `TList` 的主表單。我在 `Generics.Collections` 中加入了 `uses` 子句參照作為起點，然後將主表單欄位的宣告改成：

```
private
  ListDate: TList <TDate>;
```

當然，建立清單的主表單 `OnCreate` 事件處理常式也需要更新，因此變成：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListDate := TList<TDate>.Create;
end;
```

這時，我們可以嘗試按其餘程式碼的現狀加以編譯。程式出現一個必須注意的錯誤，這個錯誤嘗試在清單中加入物件。這時，用來編譯的對應程式碼失敗：

使用全新的 Delphi 程式碼撰寫風格與架構

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
    // add a button to the list
    ListDate.Add (Sender); // Error:
    // E2010 Incompatible types: 'TDate' and 'TObject'
end;
```

就類型檢查方面而言，新的日期清單要比原始的通用清單指標更為健全。移除該行後，程式就能編譯並運作。儘管如此，其中仍有改善空間。

下列原始程式碼用來顯示 ListBox 控制項中清單的所有日期：

```
var
    I: Integer;
begin
    ListBox1.Clear;
    for I := 0 to ListDate.Count - 1 do
        Listbox1.Items.Add (
            (TObject(ListDate [I]) as TDate).Text);
```

要注意的是轉換有點繁雜，原因是程式使用的是指標清單 (TList)，而不是物件清單 (TObjectList)。原因也可能是原始示範的日期早於 TObjectList 類別！撰寫下列程式碼可以輕鬆改善程式：

```
for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add (ListDate [I].Text);
```

使用列舉 (預先定義的通用清單完整支援的項目) 而不是一般的 for 迴路也可以改善此程式碼片段：

```
var
    aDate: TDate;
begin
    for aDate in ListDate do
        begin
            Listbox1.Items.Add (aDate.Text);
        end;
```

最後，使用擁有 TDate 物件的通用 TObjectList 也可以改善程式，不過那是下一節的主題。

先前我曾提到，TList<T> 通用類別擁有高度的相容性。其中有所有的傳統方法，例如 Add、Insert、Remove 和 IndexOf。其中也有 Capacity 和 Count 屬性。特別的是，Items 變成了 Item，但是因為是預設屬性，因此您也很少會明確參照它。

## 排序 TLIST<T>

瞭解排序如何運作其實非常有趣 (本節的目標是要為上一個例子添加排序支援)。Sort 方法的定義如下所示：

```
procedure Sort; overload;  
procedure Sort(const AComparer: IComparer<T>); overload;
```

其中 **IComparer<T>** 介面是在 Generics.Defaultsunit 中宣告。如果您呼叫程式的第一個版本，它會使用由 **TList<T>** 的預設建構函式所起始的預設比較函式。在此例中，這會毫無用處。

反而我們必須要做的是定義 **IComparer<T>** 介面的適當實作。為使類型相容，我們必須定義在特定的 **TDate** 類別上運作的實作。完成此目的的方式有多種，其中包括使用匿名方法 (在下節中說明)。它同時也是一種有趣的方法，因為它讓我有機會展示泛型的數種用法模式，並利用屬於單元 Generics.Defaults 的 *structural* 類別，稱為 **Tcomparer**。此類別定義為抽象類別以及介面的通用實作，如下所示：

```
type  
  TComparer<T> = class(TInterfacedObject, IComparer<T>)  
  public  
    class function Default: IComparer<T>;  
    class function Construct(  
      const Comparison: TComparison<T>): IComparer<T>;  
    function Compare(  
      const Left, Right: T): Integer; virtual; abstract;  
  end;
```

我們該做的是針對特定的資料類型 (在本例中為 **TDate**) 產生此通用類別，並且繼承實體類別，此類別實作特定類型的 **Compare** 方法。兩項作業可以使用要花點時間消化的程式碼撰寫用語一次完成：

```
type  
  TDateComparer = class (TComparer<TDate>)  
    function Compare(  
      const Left, Right: TDate): Integer; override;  
  end;
```

不止是您，很多人都會覺得這段程式碼極不尋常。新類別是繼承自通用類別的特定執行個體，這可以用兩個分開的步驟來表示，如下所示：

```
type  
  TAnyDateComparer = TComparer<TDate>;  
  TMyDateComparer = class (TAnyDateComparer)  
    function Compare(  
      const Left, Right: TDate): Integer; override;  
  end;
```

您可以在原始程式碼中找到 **Compare** 函式的實際實作，不過這並不是我在這裡所要強調的重點。但是請記住，即使您排序清單，其 **IndexOf** 方法也不會利用到它 (有別於 **TStringList** 類別)。

## 使用匿名方法排序

上一節呈現的排序程式碼看來相當複雜，實際情況也的確是如此。直接將排序函式傳遞至 **Sort** 方法會較為容易簡潔。在過去，這通常是藉由傳遞函式指標來達成。到了 Delphi 2009，則可以用傳遞匿名方法來取得此目的。

事實上，可以呼叫 **TComparer<T>** 的 **Construct** 方法來使用 **TList<T>** 類別 **Sort** 方法的 **IComparer<T>** 參數，將匿名方法作為參數傳遞，定義如下所示：

```
type
  TComparison<T> = reference to function(
    const Left, Right: T): Integer;
```

在實務上，您可以撰寫類型相容函式，並將它作為參數傳遞：

```
function DoCompare (const Left, Right: TDate): Integer;
var
  lDate, rDate: TDateTime;
begin
  lDate := EncodeDate(Left.Year, Left.Month, Left.Day);
  rDate := EncodeDate(Right.Year, Right.Month, Right.Day);
  if lDate = rDate then
    Result := 0
  else if lDate < rDate then
    Result := -1
  else
    Result := 1;
end;

procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct (DoCompare));
end;
```

如果這看來過於傳統，請考慮您也可以避免宣告各別的函式，並將它 (其原始程式碼) 作為參數傳遞至 **Construct** 方法，如下所示：

```
procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct
    ( function (const Left, Right: TDate):
      Integer var
        lDate, rDate: TDateTime;
      begin
```

```
lDate := EncodeDate(Left.Year,
    Left.Month, Left.Day);
rDate := EncodeDate(Right.Year,
    Right.Month, Right.Day);
if lDate = rDate then
    Result := 0
else if lDate < rDate then
    Result := -1
else
    Result := 1;
end));
end;
```

這個例子應該能夠激發您進一步瞭解匿名方法！毫無疑問的，撰寫這個最後版本會遠比上一節中原先撰寫的更為簡單。只是，對於許多 Delphi 開發人員而言，採用衍生類別可能會看來較為簡潔並且較易於瞭解。

## 匿名方法 (或 CLOSURE)

Delphi 語言擁有程序化類型（宣告指向程序和函式之指標的類型）和方法指標（宣告指向方法之指標的類型）已經有一段相當久的時間。儘管您並不常直接使用它們，這些仍然是每一位開發人員都會使用的 Delphi 重要功能。事實上，方法指標類型還是 VCL 中的事件處理常式的基礎：每次當您宣告事件處理常式時，即使是純粹的 **Button1Click**，您也是在宣告會使用方法指標連接到事件 (在此例中為 **OnClick** 事件) 的方法。

匿名方法延伸了這項功能，可讓您將方法的實際程式碼作為參數傳遞，而不是在別處定義的方法名稱。不過，這並不是唯一的差異。匿名方法管理區域變數生命週期的方式讓它們與其他方法迥然不同。

匿名方法是 Delphi 的全新功能，但是在其他的程式設計語言 (特別是動態語言) 中，它們早已以不同的形式和不同的名稱行之有年。我在 JavaScript 中的 closure 方面擁有豐富的經驗，特別是有關 jQuery ([www.jquery.org](http://www.jquery.org)) 程式庫和 AJAX 呼叫。C# 中的對應功能為匿名委派。

但是我不準備比較不同程式設計語言中的 closure 和相關方法，而會在 Delphi 2009 中詳細說明它們的運作方式。

## 匿名方法的語法與語意

Delphi 中的匿名方法是一項 *在運算式環境定義中建立方法值* 的機制。一個較隱密但突顯與方法指標重大差異的定義：*運算式環境定義*。在探討這個定義之前，讓我先以一個非常簡單的程式碼從頭開始。

使用全新的 Delphi 程式碼撰寫風格與架構

這是匿名方法類型的宣告，是 Delphi 強型別語言仍為強型別語言時就有需要：

```
type  
  TIntProc = reference to procedure (n: Integer);
```

這與參照方法唯一的差別只有宣告中所用的關鍵字：

```
type  
  TIntMethod = procedure (n: Integer) of object;
```

## 匿名方法變數

有了匿名方法類型之後，您可以針對此類型宣告變數、指定類型相容匿名方法，以及透過變數呼叫方法：

```
procedure  
  TFormAnonymFirst.btnSimpleVarClick( Sender:  
  TObject);  
var  
  anIntProc: TIntProc;  
begin  
  anIntProc :=  
    procedure (n: Integer)  
    begin  
      Memo1.Lines.Add (IntToStr (n));  
    end;  
  anIntProc (22);  
end;
```

請注意用來將含有即時程式碼的實際程序指定給變數的語法，這過去在 Pascal 中是前所未見的。

## 匿名方法參數

在更有趣的例子中 (含有更令人驚奇的語法)，我們可以將匿名方法作為參數傳遞給函式。假設您有一個含有匿名方法參數的函式：

```
procedure CallTwice (value: Integer;  
  anIntProc: TIntProc);  
begin  
  anIntProc (value);  
  Inc (value);  
  anIntProc (value);  
end;
```

此函式以兩個連續整數值 (一個作為參數傳遞的值以及接續的另一個值) 呼叫作為參數傳遞的方法兩次。您藉由傳遞實際的匿名方法給函式來呼叫它，其中含有令人出乎意料的直接即時程式碼：

```
procedure
  TFormAnonymFirst.btnProcParamClick( Sender:
  TObject);
begin
  CallTwice (48,
    procedure (n: Integer)
    begin
      Memo1.Lines.Add (IntToHex (n, 4));
    end);
  CallTwice (100,
    procedure (n: Integer)
    begin
      Memo1.Lines.Add (FloatToStr(Sqrt(n)));
    end);
end;
```

就語法的觀點而言，要注意程序是作為加上括弧的參數傳遞，而且結尾沒有分號。此程式碼的實際效果是呼叫帶 48 和 49 的 **IntToHex** 以及帶平方根 100 和 101 的 **FloatToStr**，產生下列輸出：

```
0030
0031
10
10.0498756211209
```

## 使用區域變數

即使是用不同的和「較不簡便」的語法，我們也可能使用方法指標達到同樣的效果。會讓匿名方法明顯不同的是，它們參照呼叫方法之區域變數的可行方式。考慮以下的程式碼：

```
procedure
  TFormAnonymFirst.btnLocalValClick( Sender:
  TObject);
var
  aNumber: Integer;
begin
  aNumber := 0;
  CallTwice (10,
    procedure (n: Integer)
    begin
      Inc (aNumber, n);
    end);
  Memo1.Lines.Add (IntToStr (aNumber));
end;
```

這裡的方法（仍然傳遞給 **CallTwice** 程序）使用區域參數 **n**，但是也在呼叫環境定義中使用區域變數 **aNumber**。這會產生什麼效果？兩個匿名方法呼叫會修改區域變數、在其中加入參數，第一次是 10，第二次是 11。最後 **aNumber** 的值會是 21。

## 延伸區域變數的生命週期

前一個例子展示了相當有趣的效果，但是在一序列的巢狀函式呼叫下，可以使用區域變數的事實就不是那麼樣的令人驚喜了。不過，匿名方法的效能在於它們可以使用區域變數並可視需要延伸其生命週期。範例會比冗長的說明更容易證明這一點。

我在 **TFormAnonymFirst** 表單類別中加入了 (使用類別完成) 匿名方法指標類型 (事實上是我在專案的所有程式碼中使用的同一種匿名方法指標類型) 的屬性：

```
private
  FAnonMeth: TIntProc;
  procedure SetAnonMeth(const Value: TIntProc);
public
  property AnonMeth: TIntProc
    read FAnonMeth write SetAnonMeth;
```

然後，我在表單中另外加入了兩個按鈕。第一個按鈕將匿名方法儲存在使用區域變數的屬性中 (多少與在先前的 `btnLocalValClick` 方法中一樣)：

```
procedure
  TFormAnonymFirst.btnStoreClick( Sender:
  TObject);
var
  aNumber: Integer;
begin
  aNumber := 3;
  AnonMeth :=
    procedure (n: Integer)
    begin
      Inc (aNumber, n);
      Memo1.Lines.Add (IntToStr (aNumber));
    end;
end;
```

此方法執行時，不會執行匿名方法，而只會加以儲存。初始化為零的區域變數 `aNumber` 未經修改，超出了區域範圍 (在方法終止時) 並已遭到取代。至少那是您預期會從標準 Delphi 程式碼得到的結果。

我針對此特定步驟在表單中加入的第二個按鈕呼叫了匿名方法，並儲存在 `AnonMeth` 屬性中：

```
procedure TFormAnonymFirst.btnCallClick(Sender: TObject);
begin
  if Assigned (AnonMeth) then
  begin
    CallTwice (2, AnonMeth);
  end;
end;
```

## 使用全新的 Delphi 程式碼撰寫風格與架構

當此程式碼執行時，它會在匿名方法上呼叫，此方法使用已不在堆疊上的方法之區域變數 **aNumber**。不過，由於匿名方法會攫取其執行環境定義，因此變數仍然在該處，而且只要匿名方法的該給定執行個體 (也就是方法的參照) 在周圍，就都可以使用。

若要進一步證明，請執行下列動作：按下「Store」按鈕一次，並按下「Call」按鈕兩次，您就會看到正在使用相同的 *captured* 變數：

```
5  
8  
10  
13
```

現在，請按下「Store」一次，然後再次按下「Call」。區域變數的值為何會重設？指定新的匿名方法執行個體後，會刪除舊的執行個體 (連同其自己的執行環境定義) 並且會攫取新的執行環境定義，其中包括區域變數的新執行個體。完整的序列 *Store - Call - Call - Store - Call* 產生：

```
5  
8  
10  
13  
5  
8
```

此運作方式的意涵 (類似其他某些語言的運作方式) 是使匿名方法成為極強大的語言功能，可用來實作過去簡直不可能進行的事項。

## 其他全新語言功能

Object Pascal 語言中有這麼多全新的相關功能，很容易就會遺漏一些次要的功能。

### 加上註解的 DEPRECATED 指示詞

為了達到相容性，**deprecated** 指示詞 (用來表示符號) 仍然可供使用，但是現在後面可以接上一個將顯示為編譯器警告一部分的字串。如果您定義程序，並在如下的程式碼片段中呼叫它：

```
procedure DoNothing;  
  deprecated 'use DoSomething instead';  
begin  
end;  
  
procedure  
  TFormMinorLang.btnDepracatedClick( Sender:  
  TObject);  
begin  
  DoNothing;  
end;
```

## 使用全新的 Delphi 程式碼撰寫風格與架構

在呼叫位置 (在 `btnDeprecatedClick` 方法中) 上會出現下列警告：

```
w1000 Symbol 'DoNothing' is deprecated: 'use DoSomething instead'
```

這遠勝於先前在已棄用符號的宣告上加入註解的做法：按一下錯誤訊息可進入其中使用此符號的原始程式碼行，並跳至宣告位置，然後尋找註解。毫無疑問，Delphi 2007 無法編譯上述程式碼，其中會出現錯誤：

```
E2029 Declaration expected but string constant found
```

`deprecated` 的新功能在 Delphi 2009 RTL 和 VCL 中使用得相當頻繁，而我希望協力廠商供應商能夠避免使用它，因為與舊版編譯器並不相容。

## 帶值結束

習慣上，Pascal 函式過去一向都是使用函式名稱來指定結果，如下所示：

```
function ComputeValue: Integer;  
begin  
    ...  
    ComputeValue := 10;  
end;
```

長久以來，Delphi 一直提供替代的程式碼編寫，使用 `Result` 識別碼將傳回值指定給函式：

```
function ComputeValue: Integer;  
begin  
    ...  
    Result := 10;  
end;
```

兩種方法完全相同，也不會改變程式碼的流程。如果您需要指定函式結果並停止目前的執行，您可以使用兩個分開陳述式、指定結果，然後呼叫 `Exit`。下列程式碼片段 (在字串清單中尋找含有給定數字的字串) 顯示這種方法的典型範例：

```
function FindExit (s1: TStringList; n: Integer): string;  
var  
    I: Integer;  
begin  
    for I := 0 to s1.Count do  
        if Pos (IntToStr (n), s1[I]) > 0 then  
            begin  
                Result := s1[I];  
                Exit;  
            end;  
    end;  
end;
```

## 使用全新的 Delphi 程式碼撰寫風格與架構

在 Delphi 2009 中，您可以全新的 **Exit** 特別呼叫取代兩個陳述式，以類似 C 語言的 **return** 陳述式的方式，將函式的傳回值傳遞給它。因此，您可以用更為精簡的版本撰寫上述的程式碼（此外，由於是單一陳述式，所以可以避免 **begin/end**）：

```
function FindExitValue (
    sl: TStringList; n: Integer): string;
var
    I: Integer;
begin
    for I := 0 to sl.Count do
        if Pos (IntToStr (n), sl[I]) > 0 then
            Exit (sl[I]);
end;
```

## 新的及別名的整數類型

雖然這不完全是編譯器的改變，而應該說是在 **System** 單元中添加的功能，總之您現在可以使用較易於記住的別名代表帶正負號和不帶正負號的整數資料類型。以下是編譯器中帶正負號和不帶正負號的預先定義類型：

<b>ShortInt</b>	<b>Byte</b>
<b>SmallInt</b>	<b>Word</b>
<b>Integer</b>	<b>Cardinal</b>
<b>NativeInt</b>	<b>NativeUInt</b>
<b>Int64</b>	<b>UInt64</b>

原先在 Delphi 2007 和舊版中就已經有這些類型，但是 64 位元編譯器日期僅支援少數幾個版本的編譯器。在 Delphi 2007 中原先就已經有 **NativeInt** 和 **NativeUInt** 類型，這些類型應視編譯器的版本 (32 位元和將來的 64 位元) 而定，但是並未記載它們。

如果您需要將符合 CPU 負整數大小的資料類型，則這些都是得使用的類型。事實上，整數類型在從 32 位元轉移至 64 位元編譯器時應保持不變。

## 使用全新的 Delphi 程式碼撰寫風格與架構

System 單元所新增的下列預先定義別名集在 Delphi 2009 中為全新的別名：

```
type
  Int8    = ShortInt;
  Int16   = SmallInt;
  Int32   = Integer;
  UInt8   = Byte;
  UInt16  = Word;
  UInt32  = Cardinal;
```

雖然它們並沒有添加什麼新的功能，但是可能會較易於使用，因為如果 ShortInt 小於 SmallInt，通常會難以記住，而記住 Int16 或 Int8 的實際實作卻非常容易。

## 結論

我概述了幾個加入到 Delphi 語言的有趣事件，但是這個編譯器版本的重大不同在於支援泛型、匿名方法以及兩者的組合。這些功能不只是擴充 Delphi 語言，更是在傳統的物件導向程式設計之外，獲取全新的程式設計範例以及 Delphi 向來特有的事件驅動式程式設計。讓類別在一個或多個資料類型上參數化以及將常式作為參數傳遞的功能，開啓了建構 Delphi 應用程式的全新程式碼編寫風格和全新方式。Delphi 2009 現在就擁有語言的強大功能，但是程式庫和元件開始充分利用這些功能還需要一段時間。儘管如此，有了 Delphi 2009，您馬上就能開始發展出全新的程式碼編寫技術。

## 關於作者

本白皮書是由暢銷系列 *Mastering Delphi* 的作者 Marco Cantù 為 Embarcadero Technologies 進行撰寫，其中內容摘錄自該作者最新的 *Delphi 2009 Handbook* 一書，詳見 <http://www.marcocantu.com/dh2009>。您可以在 Marco 的部落格 (<http://blog.marcocantu.com>) 閱讀關於他的相關訊息，並且寄送電子郵件到 [marco.cantu@gmail.com](mailto:marco.cantu@gmail.com) 與他交流。



Embarcadero Technologies Inc. 能夠讓應用程式開發者與資料庫專業人員在所選擇的環境中，藉由工具設計、建立與執行軟體應用程式。全球超過三百萬使用者的社群以及「財富」雜誌一百大公司中的九十家公司都是仰賴 Embarcadero 的工具增加生產力、公開協力合作與自由創新。Embarcadero 成立於 1993 年，總部位於加州舊金山，全球各地都設有辦公室。Embarcadero 的網址為 [www.embarcadero.com](http://www.embarcadero.com)。公司的旗艦版 CodeGear 工具包括：Delphi®、C++Builder® 和 JBuilder®。