Delphi to C# and Back Again

by Joanna Carter

Now that Delphi 2005 is nearly upon us, we will be able to use an IDE that allows us to program, not only in Delphi for Win32, but also in C# and Delphi for .NET. I thought it would be a good idea to look at some of the new language features that have been introduced into Delphi for .NET with Delphi 8 and that are being added to in Delphi 2005.

It certainly seems that Delphi has benefited from the invention of C# or at least the .NET framework so let's take a look at some of the new language features that have been added to Delphi to facilitate .NET development and compatibility with other .NET languages like C#.

Strict Visibility Specifiers

```
type
  TMyClass = class
  strict private
    fReallyPrivateField: string;
  strict protected
    fReallyProtectedField: string;
```

To start with, Delphi for .NET has two new visibility specifiers: **strict private** and **strict protected**. These act just like their peers, **private** and **protected**, with the exception that any members declared with such visibility are no longer visible to other code in the same unit. This brings the concept of truly private or protected visibility available in other languages to Delphi and, if used, stops developers using the 'get at protected members from other units' hack that allowed us to circumvent the protection afforded by the normal protected visibility specifier. Of course, we can still use private and protected specifiers but the new strict visibilities can be very useful when you want to avoid users of your classes doing dirty tricks and blaming your classes when their hacks go wrong.

Class Fields

Delphi for .NET not only allows us to have fields and properties scoped to an instance of a class, it also allows us to have fields and properties that are scoped to the class itself. Just as we could have class methods that allowed us to execute code without instantiating the class, so now we can have somewhere to keep data and properties to access it without creating an instance of the class.

In order to provide this capability, we now have the ability to declare fields in sections within a class declaration. The **var** keyword is now allowed to specify instance fields and a new **class var** declaration denotes that the following fields are class or static fields.

```
type
  TMyClass = class
  strict private
  //either
    class var
      _ClassField1: integer;
      _ClassField2: string;
    var
      fInstanceField1: Double;
      fInstanceField2: TDateTime;
    procedure AProcedure;
    class function: GetSomething: string;

  // or
    class var _ClassField1: integer;
    class var _ClassField2: string;
    var fInstanceField1: Double;
    var fInstanceField2: TDateTime;
    procedure AProcedure;
    class function: GetSomething: string;
```

There are three different ways of declaring both class fields and instance fields in the same class: you can either gather them together into sections as in the first example, you can prepend every field declaration with either **var** or **class var**, or you can mix and match the two styles. Note that I prepend an underscore for class fields to differentiate them from instance fields; this is purely a matter of personal preference.

If you place your instance fields before your class fields within one visibility section, then you do not need the var keyword; this is only necessary to identify when class fields end. Other ways of ending a class var section include declaring anything other than a field, e.g. method, property or type, or inserting another visibility specifier.

Class Properties

Class properties are now available in Delphi for .NET and, in Delphi 2005, are also available for Win32. The syntax is very similar to instance properties but you need to be aware of a new accessor declaration syntax.

```
type
  TMyClass = class
  strict private
    class var
      _ClassField: integer;
    class procedure SetClassProperty(Value: Integer); static;
  public
    class property ClassProperty: Integer
          read _ClassField
          write SetClassProperty;
```

The use of the **static** keyword at the end of the class method declaration is required for any method used as an accessor for a class property and indicates that the method does not contain the implicit self parameter normally associated with class methods, so don't try to call anything from within that method that explicitly requires the self parameter. Having said that, any class fields, methods or properties are all available providing you don't have to prepend the call with self.

Using Class Fields and Properties

When writing classes in Delphi for Win32, sometimes we needed to use a class as a factory. To this end we have to ensure that we only ever have one instance so, instead of writing code to ensure that only one instance of the class is created, we can use the fact that there is only one class, to enforce the Singleton pattern.

In Delphi for Win32, this would have looked something like this…

```
type
  TFormFactory = class
  public
    class procedure RegisterFormClass(FormClass: TFormClass);
    class function CreateForm(const FormClassName: string): TForm;
  end;
```

…Now because Delphi for Win32 did not support the idea of class or static variables or properties, we had to resort to some jiggery-pokery with unit variables to obtain data storage that only occurs once. So, in the case of our example, we ended up with…

```
implementation

var
  _FormClasses: TClassList;
```

…and, in order to create an instance of TClassList and to free it off, we had to resort to using the initialization and finalization sections of the unit like this…

```
initialization
  _FormClasses := TClassList.Create;

finalization
  _FormClasses.Free;
```

Now, Delphi for .NET and Delphi for Win32 in Delphi 2005 allow us to have class or static variables and properties, thus allowing us to tidy up such code, removing the need for initialization and finalization sections.

```
type
  TFormFactory = class
  private
    class var
      _FormClasses: TClassList;
    class constructor Create;
  public
```

```
    class procedure RegisterFormClass(FormClass: TFormClass);
    class function CreateForm(const FormClassName: string): TForm; static;
    class property NewForm[const Idx: string]
           read CreateForm;
  end;

implementation

class constructor Create;
begin
  _FormClasses := TClassList.Create;
end;

class procedure TFormFactory. RegisterFormClass(FormClass: TFormClass);
begin
  _FormClasses.Add(FormClass);
end;

class function TFormFactory. CreateForm(const FormClassName: string): TForm;
var
  i: Integer;
begin
  for i := 0 to _FormClasses.Count – 1 do
    if _FormClasses[i].ClassName = FormClassName then
    begin
      Result := _FormClasses[i];
      Break;
    end;
end;
```

The class constructor can be used to initialise any class fields but there is, as yet, no such concept as a class destructor. The only time that lack could become a problem is when you keep a reference to an unmanaged resource in a class field, but that is such an unlikely scenario that I will address it if and when it occurs.

Nested Types

For those of us who use OO techniques extensively, creating helper classes solely for use by one particular class has meant having either to declare a separate class in the interface section of a unit, or hide the helper class inside the implementation section of the unit that declares the outer class. The Delphi for .NET compiler now allows us to declare types nested one inside the other, thus allowing us lexically to link the two classes should we desire so to do.

```
type
  TEnumerator = class
  public
    function get_Current: TThing; virtual; abstract;
    property Current: TThing
             read get_Current;
    function MoveNext: Boolean; virtual; abstract;
    procedure Reset; virtual; abstract;
  end;

  TThingList = class
  strict private
    var fItems: IList;
  strict private
    type
      TThingEnumerator = class(TEnumerator)
      strict private
        fCurrentIdx: Integer;
        fList: TThingList;
      public
        constructor Create(const List: TThingList);
        function get_Current: TThing; override;
        function MoveNext: Boolean; override;
        procedure Reset; override;
      end;
  public
    constructor Create;
    procedure AddItem(const Item: IThing);
    function GetCount: Integer;
    function GetItem(Idx: Integer): IThing;
    function GetEnumerator: TEnumerator
  end;
```

Here, the TThingIterator class is embedded inside the TThingList class and marked as strict private so that it is not possible to access the enumerator class from outside of the TThingList class. The only way to get at the Enumerator is to call the GetEnumerator method on the list class and this returns the appropriate instance into an abstract TEnumerator reference. By using this syntax, it is possible to see at a glance that the List class has a related but hidden Enumerator class.

Of course, it is also possible to place the nested class in public visibility and use that inner class as part of the outer class, in which case the calling code would look something like:

```
procedure TestClass.ShowList(List: TThingList);
var
  ti: TThingList.TListEnumerator;
begin
  ti := List.GetEnumerator;
  while ti.MoveNext do
    ListBox1.Items.Add(ti.Current.GetName);
end;
```

Iterators/Enumerators

Because iteration or enumeration of lists is such a common requirement, both C# and Delphi for .NET encourage the use of iterators by providing a special syntax that simplifies the code required to loop through a list.

Instead of using...

```
//C#
void ShowList(ThingList list)
{
// either
  for (int i = 0; i < list.Count; i++)
    listBox1.Items.Add(list[i].GetName());
// or
  ThingIterator ti = list.GetIterator;
  while (ti.MoveNext())
    listBox1.Items.Add( ti.Current.GetName());
}

// Delphi for .NET
procedure TestClass.ShowList(List: TThingList);
var
// either
  i: Integer;
// or
  ti: TThingIterator;
begin
// either
  for i := 0 to List.Count - 1 do
    ListBox1.Items.Add(List[i].GetName);
// or
  ti := List.GetIterator;
  while ti.MoveNext do
    ListBox1.Items.Add(ti.Current.GetName);
end;
```

... we can now use a much cleaner and simpler syntax...

```
//C#
void ShowList(ThingList list)
{
  foreach (Thing t in list)
    listBox1.Items.Add(t.GetName());
}

// Delphi for .NET
procedure TestClass.ShowList(List: TThingList);
var
  t: TThing;
begin
  for t in List do
    ListBox1.Items.Add(t.GetName);
end;
```

As well as special type like string and array, if any list class contains a method called GetEnumerator and the enumerator class instance returned by that method contains a Boolean function MoveNext, a procedure Reset and a Current property, then the .NET compiler will use the enumerator / iterator that you have provided for use in the foreach / for…in construct.

It is for this reason that the example I have given uses a hidden Enumerator class because, as long as you use the foreach / for..in syntax, client code will never need to know the exact type of the enumerator created.

The End of Global Data and Routines?

Delphi has always allowed, and will still allow, us to declare global variables and routines but, when such things are compiled for .NET, behind the scenes, things are far from being the same. Because everything in either Delphi for .NET or C# has to be a class, your global variables and routines will be compiled into a class. For the sake of simplicity, we can say that anything declared in the interface section of a unit is converted into a public static class member and everything declared in the implementation section is converted into a protected static class member; not strictly accurate, but you get the gist.

With that in mind, let's look at what it takes to port some procedural type code in a Delphi unit into a C# class.

```
unit PersonU;

interface

type
  TPerson = record
    Name: string;
    DatOfBirth: TDateTime;
  end;

procedure Speak(Person: TPerson);

function GetAge(Person: TPerson): Integer;

implementation

uses
  Dialogs;

procedure Speak(Person: TPerson);
begin
  ShowMessage('My Name is ' + Person.Name);
end;

function YearsBetween(Date1, Date2: TDateTime): Integer;
var
  Day1, Month1, Year1: Word;
  Day2, Month2, Year2: Word;
begin
  DecodeDate(Date1, Year1, Month1, Day1);
  DecodeDate(Date2, Year2, Month2, Day2);
  Result := Abs(Year1 – Year2);
end;

function GetAge(Person: TPerson): Integer;
begin
  Result := YearsBetween(Person.DateOfBirth, Now);
end;

end.
```

Now the Delphi for .NET compiler creates an artificial class that encapsulates the unit, in this case also adding the record as a struct just for good measure. Altogether this is very messy but it does allow Delphi programmers to retain a pure procedural style if they so wish. If you want to see what goes on behind a Delphi for .NET unit, I suggest you use Lutz Roeder's Reflector to see just what the Delphi for .NET compiler does.

The unit just described can better be translated into C# as…

```
  public class Person
  {
    private string name;

    private DateTime dateOfBirth;
```

```
    private int YearsBetween(DateTime Date1, DateTime Date2)
    {
      return Math.Abs(Date1.Year - Date2.Year);
    }

    public string Name
    {
      get {return name;}
      set {this.name = value;}
    }

    public int Age
    {
      get
      {
        return YearsBetween(dateOfBirth, DateTime.Now);
      }
    }
  }
```

...but if we want to stay in Delphi, then we can write the same class like this...

```
type
  TPerson = class
  private
    fName string;
    fDateOfBirth: System.DateTime;
    function YearsBetween(Date1, Date2: System.DateTime): Integer;
    function GetAge: Integer;
  public
    property Name: string
             read fName
             write fName;
    property Age: Integer
             read GetAge;

implementation

function TPerson.YearsBetween(Date1, Date2: System.DateTime): Integer;
begin
  Result := Math.Abs(Date1.Year - Date2.Year);
end;

function TPerson.GetAge: Integer;
begin
  Result := YearsBetween(fDateOfBirth, DateTime.Now);
end;
```

Now, there is no reason why we could not have used the same code for YearsBetween and GetAge that we used in the procedural unit, but I wanted to demonstrate that Delphi also has access to the same .NET library classes that C# has.

The point of this exercise is to demonstrate an approximation of what the Delphi for .NET compiler does to procedural code in Delphi units and to suggest that, as your code will be converted to a class, maybe you should write it as a class to start with.

In general, any Delphi unit that contains global variables and routines can easily be converted into a class of class or static methods: something that better fits the .NET paradigm.

Conclusion

Borland have done an amazing job of trying to keep the majority of the Delphi language the same from Win32 to .NET. In the process, they have also had to add in several new language features required for interoperability with the .NET framework. This article is intended to be an introduction to some of those new features as well as some ideas of how working in .NET can improve the OO-ability of your code.

Some of the features included here are only available in the new Delphi 2005 compilers and therefore, as this article was written before the release of Delphi 2005, the syntax of some features must be regarded as changeable although several reference sites are also demonstrating the same syntax.

Technical Editor for this article was Brian Long - thanks for all your valuable comments and assistance, Brian.

[The Two Joannas]