Back To Basics: A Reintroduction To Properties, Part 1

by Dave Jewell

As part of our ongoing plans to improve *The Delphi Magazine* we intend to provide a few more introductory articles aimed at newcomers to both Delphi and Kylix. This article is aimed at those who are relatively new to Object Pascal, the object-oriented programming language used by Delphi and Kylix. Having said that, I reckon that even seasoned developers will benefit from brushing up on the basics from time to time, and I've done my best to incorporate a number of interesting little tips, techniques and dodges into these articles.

What Is A Property?

As the title suggests, this article is an introduction to the concept of properties. You don't have to use the IDE for more than a few minutes before you realise that all Delphi components have one or more associated properties that can be accessed via the Object Inspector. The same, of course, is true of any form objects derived from `TCustomForm`; `BorderStyle` and `Position` are examples of frequently used form properties. In actual fact, *any* Delphi object can have properties, regardless of whether it's a form, visual or non-visual component, or even some low-level utility class such as `TList`. To get the most from properties, you should try to remember they're not just things that appear in the Object Inspector: properties are much more generally applicable than that.

The best way of appreciating exactly what properties are is to think about what life would be like without them! Let's take a look at the familiar `TForm` class, and examine one of its properties. Suppose that you're writing a program with a main application window which you want to appear centred on the computer screen. The simplest way of doing this is to set the form's `Position` property to `poScreenCenter` at design-time (ie using the Object Inspector) but for the sake of argument, let's suppose that you do this at runtime instead.

In the code snippet in Listing 1, a button press triggers the event handler which causes the window to be centred on screen. From the perspective of someone who's never encountered properties, this assignment statement simply looks as if we're setting some public field, `Position`, to a certain value. In other words, it 'looks' (from a syntactic point of view) as if `MainForm` has some public field declared like this:

```
public
  Position: TPosition;
```

Naturally, this can't be the case. Why not? Because setting some arbitrary variable to a particular value isn't going to automatically change the position of a window, is it? To see what's *really* happening, we need to peek inside the class declaration of `TCustomForm` which you'll find inside the FORMS.PAS source file. Within that class declaration, you'll find a property definition that looks like this:

```
property Position: TPosition
  read FPosition write
  SetPosition stored IsForm
  default poDesigned;
```

The keyword `property` is always used to introduce a new property declaration, and is always followed by the name of the property itself. In this case, we're told (or rather, the compiler is told!) that the property is called `Position` and is of type `TPosition`. A property declaration can include an optional `read` and `write` clause, both of which are present in the above statement. For now, let's concentrate on the `read` and `write` clause: I'll defer a discussion of the `stored` and `default` keywords until later.

Firstly, look at the `read` clause. This tells the compiler that whenever an attempt is made to read this property, the compiler should replace it with a reference to `FPosition`, which is defined as a private field within the declaration of `TCustomForm`:

```
private
  FPosition: TPosition;
```

To put this another way, any time you write code which *reads* the value of the `Position` property, the compiler silently reads the value of `FPosition` instead. So this:

```
X := MyForm.Position;
```

is transmogrified into this:

```
X := MyForm.FPosition;
```

What's been achieved here? The key issue is encapsulation. Good object oriented design dictates that any object should behave like a little 'black box' as far as the rest of an application is concerned. An object is known to the outside world by means of its properties, methods and events. The way in which those properties, methods and events are implemented should be totally invisible to the client code which uses them. Thus, the code which accesses the `Position` property of a form or the `Count` property of a `TList` shouldn't need to know how those values are stored within the implementation of that object. In essence, the property mechanism allows us to

➤ *Listing 1*

```
procedure TForm1.Button1Click(
  Sender: TObject);
begin
  MainForm.Position :=
    poScreenCenter;
end;
```

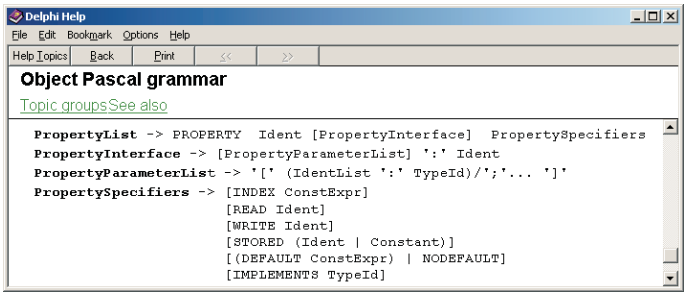
decouple the object's interface from the internal representation.

Hiding The Implementation: Getters And Setters

If this isn't too clear yet, imagine that you're writing a class, `TOSInfo` which encapsulates real-time operating system information, such as the amount of free memory on your PC. One way of doing this would be to declare a property:

```
property FreeMemory: Integer
  read FFreeMemory;
```

In other words, every time the client code accesses the `FreeMemory` property, it actually gets the value of `FFreeMemory` which is assumed to be a private field within `TOSInfo`. Hmmm. There's a problem here. Since the amount of free memory is typically changing all the time, how are we going to keep this private field up to date? I guess we could create a timer within the `TOSInfo` class and update the field once every half a second or so. *Stop!* Don't even think about it! There's a much better way of doing things once you realise that you can specify the name of a function rather than a field in a property declaration's `read` clause:



➤ *Figure 1: Here's that part of the Object Pascal grammar which relates to properties. It looks straightforward, but there's a lot going on underneath the hood.*

```
property FreeMemory: Integer
  read GetFreeMemory;
```

This assumes that we've defined a private routine with this function prototype:

```
function GetFreeMemory:
  Integer;
```

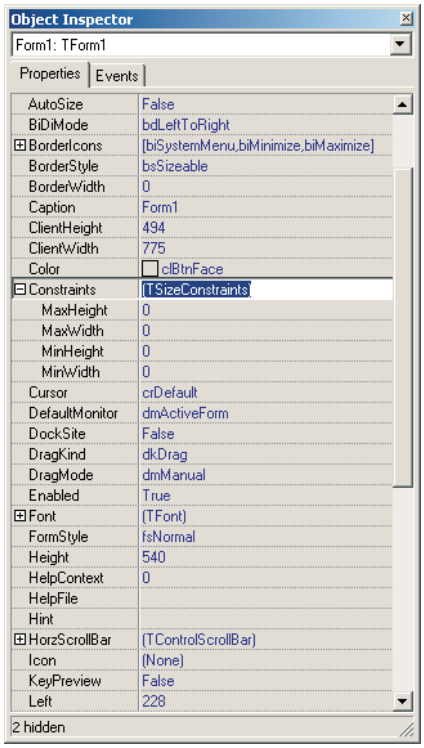
In effect, the `GetFreeMemory` routine will be called whenever the client code refers to the `FreeMemory` property of `TOSInfo`, and of course that's the point at which we should update the free memory information from the operating system itself. Thus, we've avoided the need to mess about with timers or self-updating fields.

What this really boils down to is that a property 'exported' from a Delphi class need not be stored in physical memory at all. It doesn't need to correspond to a memory location but can be some value which is calculated on demand as and when required. This makes properties particularly powerful, especially in the sort of scenario we're looking at here.

I refer to `GetFreeMemory` as a 'getter' function because it 'gets' the value of a property. Similarly, you can associate either a field or a routine name with the `write` clause of a property declaration. Again, I tend to refer to such a routine as a 'setter' procedure. In the above example, we've specified a getter function for `FreeMemory`, but not a setter procedure. This is logical because it's not possible to arbitrarily change the amount of free memory in Windows under program control! In other words, this particular property is naturally a read-only quantity, which is exactly what you'll get if you omit the `write` clause.

By convention, the getter and setter routines associated with a property are called `GetXXX` and `SetXXX` where `XXX` is the name of the property itself. Thus, you'd use `GetFluxDensity` and `SetFluxDensity` to read and write the value of a field called `FluxDensity`. This convention is not enforced by the compiler, although many Delphi programmers adhere to it since it helps to relate a particular getter or setter with a particular property.

➤ Figure 2: Next month, I'll be explaining how the Object Inspector uses RTTI to display property information, how to fool Object Inspector into displaying read-only properties and more...



You'll have hopefully realised from the above that a getter function must return a type which is consistent with the type of the associated property. If you define an `integer` property and try to associate it with a getter function that returns a `Boolean`, then the compiler will naturally complain. Similarly, a setter routine is defined as a procedure (not a function) which takes exactly one parameter corresponding to the required new value of a property. A complete example is shown in Listing 2.

There are a couple more conventions used here worth mentioning: firstly, the private field associated with the `Color` property is named by simply adding an `f` to the front of the property name. Here again, the compiler doesn't insist that you do this, but it makes it easier for other developers to find their way around your code. Secondly, the obligatory parameter to the setter function is called `Value`. You can call it whatever you like, but `Value` is a favourite of seasoned component writers! If you're implementing a property of type `String`, then it's worthwhile using a `const` string argument to your property setter function. This causes Delphi to produce more efficient code:

```
procedure SetUserName(
  const Value: String);
```

In terms of encapsulation and implementation-hiding, setter routines are particularly important. If you protect your implementation behind a setter, then you've essentially given yourself the ability to police the possible values which can be assigned to the property. For example, if there's some reason why you want to disallow certain colour assignments, you can do it as in Listing 3.

In this example, any attempt to set the `Color` property to black will be politely ignored, something that it obviously wouldn't be possible to do if `Color` were a public field of the object. In a similar way, any property which is an enumerated type could potentially be cast to an invalid value through malicious casting, which again can be easily prevented through the use of a setter function. Something you'll also notice is that getter and setter routines allow some arbitrary action to be taken whenever a property is read or written. In the above case, any change in the `Color` property of the object triggers an immediate call to `Refresh`, which (in the case of a visual component) will cause it to be immediately redisplayed in its new livery. The bottom line is that properties are very useful things!

Property Visibility Issues

As you'll appreciate, Object Pascal offers a number of different ways of defining the scope of an item, based upon the use of the `private`, `protected`, `public` and `published` keywords in the class definition. In a nutshell, `private` scope means that an item can only be used within the source code unit where the item is defined. The `protected` keyword extends the visibility of the item to any derived class whereas `public` scope makes an item visible without restriction. The `published` keyword is a special case which we'll be looking at more closely in the next article.

What this all means is that it's perfectly legitimate to define a property which has, for example, only private visibility. Such a property will only be usable within the .PAS file where it is defined, but that's still useful if you have one or more classes defined within the same source file, and want to use the properties of one class from another. In fact, it's perfectly valid for a class to define private properties which are only used within the methods of that same class.

There is one big caveat however. Look back at the `TColoredWidget.SetColor` routine that I showed you earlier. Suppose you inadvertently replaced the assignment to `FColor` with an assignment to `Color`, like this:

```
Color := Value;
```

This would cause the compiler to replace the assignment with a call to the `SetColor` routine, and the result would be instant recursion with `SetColor` calling itself forever, or at least until a stack fault intervened! Because of this potential pitfall, my personal preference is to strongly avoid use of a class's properties from within the methods of that same class.

```
private
  FColor: TColor;
  procedure SetColor(
    Value: TColor);
published
  property Color: TColor
    read FColor
    write SetColor
    default clBtnFace;
```

➤ *Listing 2*

```
procedure TColoredWidget.SetColor(
  Value: TColor);
begin
  if Value <> clBlack then begin
    FColor := Value;
    Refresh;
  end;
end;
```

➤ *Listing 3*

While on the subject of visibility, you should appreciate that it's possible for a derived class to override property definitions in the ancestor class. As a real-world example of this, take another look at the declaration for `TCustomForm` in FORMS.PAS. Look through the protected part of the declaration and you'll see this:

```
property Ctl3D default True;
```

Hmmm... How come this property doesn't have an associated type? Is `Ctl3D` a byte, a Boolean or a banana? Whenever you see an apparent property declaration that doesn't include a type, you're actually looking at a property override. In other words, a redefinition of some property which is already defined in an ancestor class. `TCustomForm` is derived from `TScrollingWinControl` which, in turn, is derived from `TWinControl`. This class (defined inside CONTROLS.PAS) provides the initial definition of `Ctl3D`, which, as we all know, is a Boolean. The line of code above has just one job, to change the default value of `Ctl3D` from `False` to `True`. You can also use property overrides to increase the visibility of a previously defined property, but you can't use them to reduce visibility. For example, if an ancestor property was declared as `protected`, then you can 'promote' it to `public` or `published`, but you can't 'demote' it to `private`.

Array Properties

A particularly nice feature of Object Pascal is the ability to create array properties. That is, properties that behave as if they're arrays. Here's an example of such a property declaration: notice that the property declaration has been declared just as if it were an array.

```
property FileName[Index:
  Integer]: String
  read GetFileName
  write SetFileName;
```

I mentioned earlier that a getter function takes no parameters and

that a setter procedure takes a single `Value` parameter. When it comes to indexed properties, this is no longer the case because it's also necessary to supply an index parameter to distinguish between different 'elements' of the array. In the above case, the getter and setter routines might look like this:

```
function
  TSomeObject.GetFileName(
  Index: Integer): String;
procedure
  TSomeObject.SetFileName(
  Index: Integer;
  const Value: String);
```

Notice that the `Index` value always comes first in the argument list: this isn't a matter of choice. Because there needs to be some way of passing the array index to the getter or setter method, the Delphi documentation states that you can't use fields in the `read/write` clause of indexed property declarations: you must use a method. Personally, I don't see why it wasn't possible to use a compatible array as a field and map the property access directly onto this field, but then I didn't write the compiler!

Surprisingly, Delphi doesn't limit you to single-dimensional arrays. Look at the declaration for `TCanvas` in Graphics.Pas and you'll see this:

```
property Pixels[X, Y: Integer]:
  TColor read GetPixel
  write SetPixel;
```

Here, `Pixels` is a two-dimensional array property. As before, the two index values, `X`, `Y` must precede the `Value` parameter in the property setter routine. In fact, Delphi will even allow you to create arrays with non-integer indexes such as:

```
property Password[const
  UserName: String]: string
  read GetPassword
  write SetPassword;
```

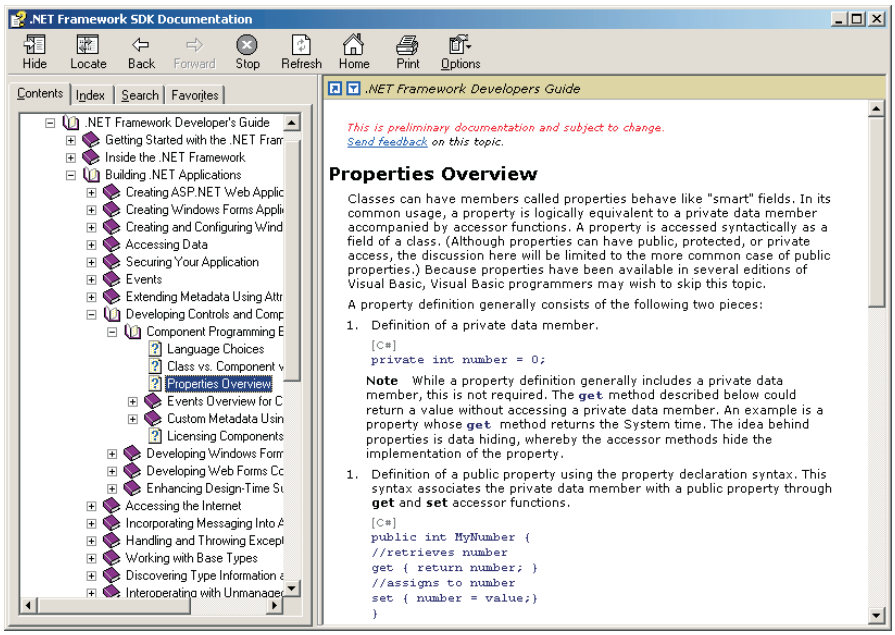
In this (rather scary!) example, a property has been declared which takes a user name as the index and returns the corresponding password for that user. You would define the `GetPassword` routine like this:

```
function
  TPasswordManager.GetPassword(
  const UserName: String):
  String;
```

With such a property, client code could retrieve a password as easy as this:

```
TPOP3.MessageText := 'Shhh: '+
  'Dave''s Password is '+
  PWManager.Password[
  'Dave Jewell'];
```

So far, we've just skimmed over the surface of property support under Delphi and Kylix. In the second part of this article, we'll delve considerably deeper. Amongst other things, I'll take a look at Object Inspector compatibility, the format of RTTI, default properties, indexed properties, persistence, implementing read-only properties that show up in the Object Inspector (such as the ubiquitous `About` property), and the new support for property categories in Delphi 5. As an aside, I'll also show you how properties are implemented in Microsoft .NET: it's almost as good as Delphi!



➤ *Figure 3: As you might expect, there are a lot of similarities between the Object Pascal concept of properties, and the way in which Microsoft .NET does it. Here again, I'll discuss this in more detail in Part 2.*

Dave Jewell is the Technical Editor of *The Delphi Magazine*; email him at TechEditor@itecuk.com