Polymorphism

The classic explanation of polymorphism is to break the word down **into** two Greek words, 'poly' meaning 'many' and 'morph' meaning 'shape' giving us 'many shapes'. I have always felt that this explanation was particularly lacking in usefulness so we will leave this explanation behind and continue on a more lucid thread.

Polymorphism can be described as the ability to send the same message to objects of different types. For example, you can send the 'Next' message to a TTable object to instruct it to go to the next record. You can also send the same message i.e. 'Next' to a TClientDataSet to instruct it to go to the next record. This ability is called 'polymorphism' and it has a single purpose: to allow you to write more generic code and, therefore, more reusable code. Continuing with the TTable/TClientDataSet example you should be able to write a single routine which would work with both a TTable and a TClientDataSet where all of the code is generic.

Before we see an example of how it works in code we need to cover how it doesn't work. This may seem a strange place to begin but it is what you start with if you don't take any further action. Consider the following piece of code:

```
var
  ShadowedBox:      TShadowedBox;
  Box: TBox;
Begin
  ShadowedBox := TShadowedBox.Create(10,10,70,20);
  ShadowedBox.Show;   // TShadowedBox.Show
  ShadowedBox.Free;

  Box := TShadowedBox.Create(10,10,70,20);
  Box.Show;  // TBox.Show
  Box.Free;
End;
```

There are really two examples in one here. Firstly, there is a variable, ShadowedBox, of type TShadowedBox. The subsequent code creates a TShadowedBox object, shows it and frees it. The Show method which executes is the TShadowedBox.Show method. This part of the example is not in the slightest hit surprising. There is nothing new to learn in this part. It exists simply to prove what you already know and to prove it alongside another example which might make you think a little.

The other part of the code is apparently meaningless. For the moment suspend your criticism of what appears to be a nonsensical example and trust that there is a point. The second part of the code declares a variable, Box, of type TBox. It then creates a TShadowedBox and assigns it to the variable, shows the box and frees it. The Show method which executes is the TBox.Show method and this is sometimes surprising to Delphi programmers. The object in question is a TShadowedBox object but the method which was executed was the TBox.Show method and no the TShadowedBox.Show method. Why? Remember the variable declaration? The variable was declared as type TBox and not of type TShadowedBox.

There is a way to get the TShadowedBox.Show method to execute instead of TBox.Show. We will cover this in a moment but before then we need lo explain what a 'generic' class is. Consider this example:

```
var
  Box TBox;
  ShBox: TShadowedBox;
  InBox: TIntelligentBox
begin
  Box   := TBox.Create;
  ShBox := TShadowedBox.Create;
  InBox := TIntelligentBox.Create;

  ShowBox(Box);
  ShowBox(ShBox);
  ShowBox(InBox);

  Box.Free;
  ShBox.Free;
  InBox.Free;
end;

procedure ShowBox(Box: TBox);
begin
  Box.Show;
end;
```

This code creates three boxes of three different classes, TBox, TShadowedBox and TIntelligentBox. TShadowedBox and TIntelligentBox both inherit from TBox. Then, the three boxes are passed to a procedure called ShowBox in order to show the box. The procedure called ShowBox receives a single parameter, Box, of type TBox. In this context TBox is called a 'generic' class. It means that the parameter being passed in can be either TBox or a descendant of TBox. On the next line the box is sent the Show message. This line is polymorphic; it sends the same message (Show) to objects of different classes (TBox, TShadowedBox and TIntelligentBox). Unfortunately for us, it doesn't execute the correct Show method and that's the problem we are going lo solve next.

Incidentally, the definition of polymorphism is a little bit confused in the Delphi world. When Delphi 1 was released people believed that polymorphism was the ability to send the same message to objects which inherit from a common ancestor. Certainty this definition of polymorphism was true for Delphi 1 but the requirement lo inherit from a common ancestor is specific to Delphi 1 and 2 and not to other languages which implement polymorphism. Delphi 3 added support for Interfaces and this meant that Delphi could support the same definition of polymorphism as other languages because Interfaces do not require you to inherit from a common ancestor.

Method Directives

A method directive specifies how overridden methods should be handled and, therefore, whether polymorphism should work. Delphi supports the following three directives:

Static	Overridden methods are not seen by generic class types	Default Directive
Virtual	Overridden methods are always seen	Faster than dynamic
Dynamic	Overridden methods are always seen	Less memory than virtual

The default directive is Static and this is the cause of the symptoms which we have seen so far. The TBox.Show method was Static and it meant that any variable which was declared of type TBox would always get the TBox method and not the overridden method. This explains why the first example executed the TBox.Show method and not the TShadowedBox.Show method even though the object was constructed from the TShadowedBox class.

The other two directives, Virtual and Dynamic, essentially achieve the same result: they make polymorphism work. When they are used overridden methods are always seen regardless of how the variable was typed. So if the variable is a TShadowedBox then the TShadowedBox.Show method will always be executed regardless of its context if the parent's method is either Virtual or Dynamic.

To use Virtual or Dynamic you must remember that there are two halves to the process. First, we need to instruct the parent's method that child classes can override the method and, second, we need lo instruct the child's method that it is actually overriding the parent's method. Let's see an example. Here are the TBox and TShadowedBox classes re-implemented using the virtual and override keywords:

```
TBox = class
  Left    : integer;
  Top     : integer;
  Bottom  : integer;
  Right   : integer;
  Constructor Create(crLeft, crTop, crRight, crBottom : integer);
  procedure Show; virtual;
  procedure Hide;
end;

TShadowedBox = class(TBox)
  procedure Show; override;
end;
```

Armed with these changes the previous two examples will always call the overridden method no matter what the context.

So when should you use Virtual and Dynamic? Unless you have a good reason not to use these directives then you should always use them for every method which is not Private. The good reason not to is usually speed. Static methods do not go through a lookup process at runtime so they execute slightly faster.

Virtual vs. Dynamic

We have two method directives, virtual and dynamic, which achieve the same result. There is a general rule of thumb for which one to use which goes like this:

Virtual is faster than dynamic. Dynamic uses less memory than virtual.

This would be a helpful guide to remember if it weren't for the fact that it is not always true. I will digress into a short discussion on the technical differences between virtual and dynamic but you do

not need to know or understand the discussion. If you wish to skip to the next section you can do so and remember a simpler maxim:

Always use virtual (unless memory is critical and you are prepared to prove that dynamic really does use less memory than virtual in your class hierarchy).

Now for the discussion on the technical difference. A virtual method's lookup information is stored in a table called a Virtual Method Table (VMT). Every class has a VMT. When a message is sent to the object the object attempts to resolve the name by looking it up in the VMT. Each VNT contains a list of all of the virtual methods that the class introduces **plus** all of the virtual methods of its parent. Because its parent's VMT also contains all of the virtual methods of its parent you can see that each class's VMT is a complete record of all of the virtual methods that could be executed for an object for this class. The benefit is that any attempt to execute a virtual method requires just a single lookup into the VMT regardless of which ancestor actually implemented the method. This is the reason why, earlier, I said that the 'tree was flattened' and that Delphi doesn't suffer the performance penalty that applies to some other languages. The downside to this is that because each class's VMT contains the complete record of all of the class's virtual methods it requires an equivalent amount of memory to hold this VMT.

This is where dynamic methods come in. Dynamic methods are stored in a Dynamic Method Table (DMT) which is similar to a VMT. However, the entries in a DMT are just the methods which are implemented in the class alone and not all of the parent's entries as well. When a method name needs to be resolved the class's DMT is looked up for the method name and if it is not found then the class's parent's DMT is looked up. The lookup continues back up through all of the ancestors until it is found or there are no more ancestors. There are two consequences of this. Firstly, the lookup process can be slower than for a VMT because there may be as many lookups as there are ancestors. Secondly, the DMT only contains the entries it needs to contain and therefore is usually smaller than the equivalent VMT.

All of this information leads us to believe that the original rule of thumb (i.e. Virtual is faster than dynamic. Dynamic uses less memory than virtual) is correct. The reason why it is misleading is that it doesn't take into account the size of the entries in the VMT and the DMT. The entry in the DMT is bigger. The entry in the VMT is just a method pointer but the entry in the DMT is a method pointer and also a method number. So it is problematical to tell whether the dynamic method uses less memory or not. You would need to perform a calculation yourself in order to determine the total memory requirement of a VMT as opposed to the total memory requirement of a DMT. All of this is affected by how many ancestors a class has and in which ancestors the methods are implemented and overridden so you can see how difficult it is to be sure that dynamic really does use less memory than virtual.

Abstract Methods

Delphi allows any virtual or dynamic method to be an abstract method. An abstract method is one which is not implemented by the class in which it is declared. Here's a TBox class which has an abstract Execute method:

```
TBox = class
  protected
    function Execute: boolean; virtual; abstract;
end;
```

At first sight it seems like a strange concept to declare a method but to provide no implementation for it. In practice this is a very useful concept. It allows you to build classes where you know what the class will look like when it is finished but it is up to the subclasses to finish it off. Such classes are often called abstract classes. TDataSet is a classic example of an abstract class. It contains many methods which are marked as abstract. TDataSet knows the syntax of these methods but because TDataSet is generic it cannot implement them. Instead it leaves them to subclasses like TTable and TClientDataSet to provide the missing pieces. The abstract class is useless by itself the benefit is that people can use TDataSet as a blueprint for what the class should look like when it is finished by a subclass.

Another good example of an abstract class is TThread. TThread knows all about running a separate process. It simply lacks the all important method which contains the code which should execute in the separate process. As a result TThread.Execute is abstract.

Of course if someone were foolish enough to try to use it like this:

```
var
  Box: TBox;
begin
  Box := TBox.Create;
  Box.Execute;
  Box.Free;
end;
```

then the compiler would generate the following warning:

```
Warning: Constructing instance of 'TBox' containing abstract methods
```

If you were foolish enough to execute the code then it would fail on the Box.Execute line with an 'Abstract error'.