Chapter 6 Introducing type inheritance

Main concepts

– Substitution with programmer classes
– Early (static, compile time) and late (dynamic, run time) binding
  – Virtual and override methods
– Polymorphism = substitution + dynamic binding
  – Multiple associations through a single link, reduced coupling, evolvability
– Abstract methods
– The concept of a type
– The Polymorphism Pattern

Chapter contents

Introduction

The form of inheritance we concentrated on in the first four chapters is class inheritance, or subclassing. *Subclassing* is a very important part of OO programming since it provides a powerful mechanism for *reuse*: subclasses reuse their superclasses' data fields and methods through inheritance.

In chapter 5 we began using inheritance differently, for substitution. An important part of substitution is that the superclass establishes a particular *type*. Subclasses that implement this type fully, ie that can substitute for the superclass under all conditions, are called *subtypes*. This may not yet be particularly clear since the example in the previous chapter, which used VCL components, concentrated on substitution. So in this chapter we look briefly at substitution again, but in the context of programmer-generated classes. We then incorporate dynamic binding to extend substitution to the concept of *polymorphism*. This leads naturally to an exploration of abstract classes. The concepts of substitution, polymorphism and abstract classes are closely tied to the concept of subtyping and all three of these are important aspects of many OO patterns. For some people, polymorphism represents the heart of OO programming.

This chapter establishes the concept of polymorphism (subtyping) as distinct from reuse (subclassing). Following chapters illustrate some of the ways in which polymorphism is used and reveal why skilful use of polymorphism is one of the most important aspects of OO programming.

Example 6.1 A polymorphic program

We start this chapter by reviewing briefly the principles from the previous chapter (ie generalisation and substitution), but using programmer defined classes, and then extend these to a polymorphic program by introducing dynamic binding.

We use a simple ancestor class, TFurniture, and two subclasses, TChair and TTable. (TChair and TTable are also valid subtypes of the TFurniture type.) We will create objects of different types and then send a message to each by calling a method that each type responds too. To keep the principles clear, this is a very simple example, and so these various furniture objects know only what kind they are. In a full system, say for an antique shop, they would be more complex, with a variety of methods and carrying information about their value, what wood they are made of, when they were made, and so on. But our simple GetKind access method is sufficient to introduce polymorphism.

The driver program has the following interface (figures 1 & 2).

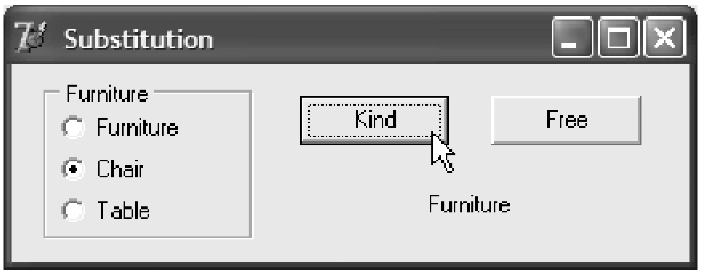


Figure 1 Interface for substitution example with static binding

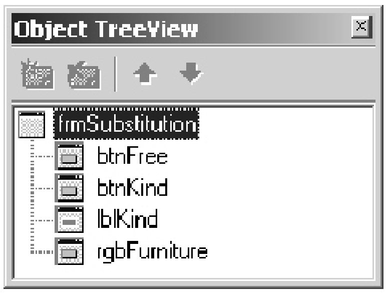


Figure 2 Components comprising the user interface

In this example we'll also introduce a convenient tool, called Class Completion (mentioned briefly in chapter 4), that speeds the creation of our own classes. So this example will cover a lot of interesting ground!

Ex 6.1 step 1 Creating the classes and types

Start a new application and add a second unit to it through the menu sequence File | New | Unit. Save this unit (called Unit2) as FurnitureU.pas and then enter the program as follows. Start with just the class declarations (lines 3–11 below). Then place the cursor on the declaration of function GetKind ... (line 6) and press <Shift+Ctrl+C> (or right-click and select Complete Class at Cursor) to invoke Class Completion. This will create a skeleton for the method. Now it is simply a matter of entering the program code (line 16).

```
 1 unit FurnitureU;

 2 interface

 3 type
 4   TFurniture = class (TObject)
 5   public
 6     function GetKind: string;
 7   end; // TFurniture = class (TObject)

 8   TChair = class (TFurniture)
 9   end; // TChair = class (TFurniture)

10   TTable = class (TFurniture)
11   end; // TTable = class (TFurniture)

12 implementation

13 { TFurniture }

14 function TFurniture.GetKind: string;
15 begin
16   Result := 'Furniture';
17 end; // function TFurniture.GetKind: string

18 end. // end Unit FurnitureU
```

This is a rather strange set of classes! We have three classes. TFurniture, derived from TObject (line 4), has one method, GetKind, that returns a string. The second and third classes, TChair and TTable, are derived from TFurniture and have neither data nor methods at this stage.

We'll expand these classes in a little while, but for now we need a driver to test these class definitions.

Ex 6.1 step 2 Testing the classes

We'll use the Form and Unit1 to write a driver program (figures 1 & 2). To add the RadioGroupBox's Items.Strings, select the RadioGroupBox and click on the three dots alongside the Items property in the Object Inspector. The property editor appears. Enter Furniture, Chair, Table, each on a new line.

Notice that we add a private data field of type TFurniture to our user interface class (ie to the form) in line 16 below and so we must add FurnitureU to the uses clause (line 5).

```
 1 unit SubstitutionU;

 2 interface

 3 uses
 4   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
 5   Dialogs, StdCtrls, ExtCtrls, FurnitureU;

 6 type
 7   TfrmSubstitution = class(TForm)
 8     btnKind: TButton;
 9     lblKind: TLabel;
10     btnFree: TButton;
11     rgbFurniture: TRadioGroup;
12     procedure btnKindClick(Sender: TObject);
13     procedure btnFreeClick(Sender: TObject);
14     procedure rgbFurnitureClick(Sender: TObject);
15   private
16     MyFurniture: TFurniture; // Declaring an application object
17   end; // TfrmPolymorphism = class(TForm)

18 var
19   frmSubstitution: TfrmSubstitution;

20 implementation

21 {$R *.DFM}

22 procedure TfrmSubstitution.rgbFurnitureClick(Sender: TObject);
23 begin
24   FreeAndNil (MyFurniture);                 // Clean up the references
25   case rgbFurniture.ItemIndex of
26     0: MyFurniture := TFurniture.Create;
27     1: MyFurniture := TChair.Create;                    // substitution
28     2: MyFurniture := TTable.Create;                    // substitution
29   end;
30   lblKind.Caption := '';
31 end; // end procedure TfrmSubstitution.rgbFurnitureClick
```

```
32 procedure TfrmSubstitution.btnKindClick(Sender: TObject);
33 begin
34   if MyFurniture = nil then
35     lblKind.Caption := 'Object not defined'
36   else
37     lblKind.Caption := MyFurniture.GetKind;
38 end; // procedure TfrmSubstitution.btnTypeClick

39 procedure TfrmSubstitution.btnFreeClick(Sender: TObject);
40 begin
41   if MyFurniture = nil then
42     lblKind.Caption := 'No object exists'
43   else
44   begin
45     FreeAndNil (MyFurniture);
46     lblKind.Caption := 'Object freed'
47   end;
48   rgbFurniture.ItemIndex := -1; // Clear indication
49 end; // end procedure TfrmSubstitution.btnFreeClick

50 end. // unit SubstitutionU
```

Lines 27 and 28 are interesting. MyFurniture is declared as a TFurniture (line 16) and so it is no surprise that we can assign it to an instance of TFurniture in line 26. But in lines 27 and 28 we assign a reference of type TFurniture to instances of types TChair and TTable. We can do this because TChair and TTable are derived from TFurniture and so we can take advantage of substitution.

Run this program and test it. Clicking any of the RadioButtons and then the Kind button displays the message ‘Furniture’. We see the same message whether MyFurniture is a TFurniture, a TChair or a TTable since line 37 above invokes the GetKind method defined in TFurniture (step 1 lines 14–17) through the inheritance relationships between the classes.

But wouldn’t it be nice if the TChair instance could declare itself to be a chair and the TTable instance show that it is a table? We do this in the next step. But just before doing that, notice the care taken in this program about creating and freeing the object and testing for its existence. Since an object continues to exist after the event handler rgbFurnitureClick completes, the other two event handlers both start with a test for its existence.

Ex 6.1 step 3 Statically bound subclass methods

We want TChair and TTable instances to be able to respond individually about what kind of furniture they are, so let’s modify the class definitions to give them their own methods. (Remember to use Code Completion. First type in only the additional method declarations, lines 9–10, 13–14 below. Placing the cursor on these declarations in turn, press

<Shift+Ctrl+C>. Delphi inserts the relevant method skeletons, leaving the programmer to insert the required program statements (lines 25, 30).)

```
 1 unit FurnitureU;

 2 interface

 3 type
 4   TFurniture = class (TObject)
 5   public
 6     function GetKind: string;
 7   end; // TFurniture = class (TObject)

 8   TChair = class (TFurniture)
 9   public
10     function GetKind: string;
11   end; // TChair = class (TFurniture)

12   TTable = class (TFurniture)
13   public
14     function GetKind : string;
15   end; // TTable = class (TFurniture)

16 implementation

17 { TFurniture }

18 function TFurniture.GetKind: string;
19 begin
20   Result := 'Furniture';
21 end; // function TFurniture.GetKind: string

22 { TChair }

23 function TChair.GetKind: string;
24 begin
25   Result := 'A Chair';
26 end; // end function TChair.GetKind

27 { TTable }

28 function TTable.GetKind: string;
29 begin
30   Result := 'A Table';
31 end; // end function TTable.GetKind

32 end. // end Unit FurnitureU
```

Run and test this. The result is a bit disappointing! No matter whether we click Furniture, Chair or Table we always get the same Furniture message from TFurniture's GetKind method. So even if the subclasses have their own methods, *when they are assigned to the superclass's variable type, they invoke the superclass's method* and not their own.

Let's look at this again in terms of the program code. When the compiler reaches line 37 of step 2, the compiler has no way of knowing whether MyFurniture is of type TFurniture, TChair or TTable. The only clue it has is that in step 2 line 16 we declared the Furniture variable to be of type TFurniture. So when Delphi compiles step 2 line 37, it links the method calls that variable MyFurniture makes *to the class TFurniture*, irrespective of what class MyFurniture has been assigned to, because this is how the variable Furniture has been declared. This is called *early*, *static* or *compile-time* binding between the variable and the method.

However, this is not what we want. We want Delphi to associate the methods with the class of the particular object we are using at that moment in the execution. If we are using the superclass because the most recent execution of the Case statement caused the execution of step 2 line 26, we want Delphi to call the superclass's method when it gets to step 2 line 37 However, if we are using a subclass, (ie if step 2 lines 27 or 28 were most recently executed), we want Delphi to be smart enough to call the appropriate subclass's method in step 2 line 37. As the next step shows, we can achieve this by using *late*, *dynamic* or *run-time* binding. (All three terms refer to the same thing.)

Ex 6.1 step 4 Dynamic binding and polymorphism

We want the subclass methods to be able to *override* the superclass methods when we are dealing with the subclass. To do this, the superclass must have *virtual* methods (ie it must use dynamic binding). Virtual methods are overridden by subclass methods whenever a subclass is substituting for the superclass. Doing this is surprisingly easy. We need only change the method declarations to state that the superclass method is a virtual method (ie capable of being overridden) and that the subclass methods are override methods (lines 6, 10 & 14 below). We do not need to change the actual method implementations.

```
 3 type
 4   TFurniture = class (TObject)
 5   public
 6     function GetKind: string; virtual;
 7   end; // TFurniture = class (TObject)

 8   TChair = class (TFurniture)
 9   public
10     function GetKind: string; override;
11   end; // TChair = class (TFurniture)

12   TTable = class (TFurniture)
13   public
14     function GetKind : string; override;
15   end; // TTable = class (TFurniture)
```

Run the program and experiment with it to see what it does. Now if you click any RadioButton and then click the Kind button you get the response we were hoping for – the message displays the type of object that was created when you clicked on the RadioButton. At last, the subclass methods are being associated with the subclass instances as they are created at run-time.

So the *same* statement (step 2 line 37) results in *different* methods being called, and therefore in different behaviour, depending on the type of the object that is currently assigned to that variable. This is *polymorphism*, and step 2 line 37 is a polymorphic call.

Two factors are necessary for polymorphism. First, there must be (hierarchical) substitution so that an instance of a subclass can be assigned to a variable of the superclass type. Second, there must be dynamic binding so that the method that is called is determined by the type of the instance at run time. (As mentioned above, static binding is set during compilation. It calls the method determined by the type of the variable declaration unaffected by run time considerations.)

Substitution works only in one direction. If we declare the variable to be of a subclass type, we cannot then substitute a superclass instance. This makes sense: practically, the superclass cannot take on the role of the subclass because the subclass may have additional variables and/or methods not present in the superclass. These variables and/or methods would be left undefined if a superclass were assigned to the subclass. Semantically it also does not make sense for the superclass to substitute for the subclass. While TTable IsA TFurniture, TFurniture IsNotNecessarilyA TTable: TFurniture cannot act as a TChair because TChair is derived from TFurniture and not the other way around.

Ex 6.1 step 5 UML class and object diagrams

The declaration of MyFurniture: TFurniture; as part of TfrmSubstitution (step 2 line 16) establishes an association between these two classes. We can show this explicitly as a one-directional link in a UML class diagram (figure 3).

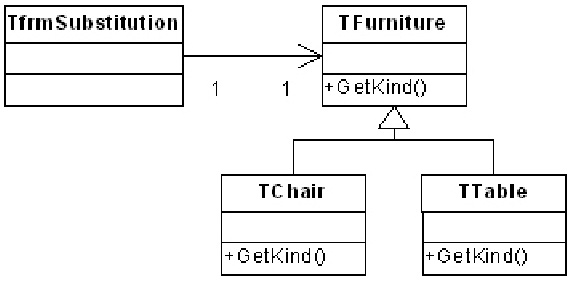


Figure 3 Association between TfrmSubstitution and TFurniture

However, because of substitution and dynamic binding, a TChair object or a TTable object can take on the role of a TFurniture object. So the possible associations are between frmSubstitution (the user interface object) and an instance of any of TFurniture, TChair or TTable. If, for example, we select Chair on the user interface (and so execute line 27 of step 2), we get the following object diagram (figure 4):

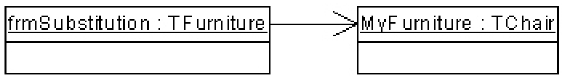


Figure 4 Association between instances

Summary of this example: Polymorphism is the combination of substitution and dynamic binding. By setting up a single association between two classes (ie between TfrmSubstitution and TFurniture), polymorphism has allowed us *three* different associations (ie between TfrmSubstitution and TFurniture and its two subclasses) *without the need to specify each separate association individually*. This reduces significantly the coupling between objects, an important attribute in any good program design.

Example 6.2 Alternatives to polymorphism

Newcomers to OO sometimes feel reluctant to exploit polymorphism; it seems too strange. How important is it? What are the alternatives? In this case, because this is such a simple example, the easiest alternative is not to subclass at all. Instead we can just declare an attribute in the TFurniture class to indicate Furniture, Chair, or Table. But in cases where the differences between subclasses are more extensive, combining everything into a single class leads to a class that is very big and unwieldy, and makes future change and maintenance very difficult. So can we retain the hierarchy of three separate domain classes from the previous example but find an alternative program that does not use polymorphism?

Polymorphism is the combination of substitution and dynamic binding. In step 1 of this example we'll look at what happens if we use neither substitution nor dynamic binding. In step 2 we'll use substitution with static binding.

Ex 6.2 step 1 No substitution

Without substitution we need to declare three reference variables in class TfrmSubstitution, to FreeAndNil all three references in the rgbFurnitureClick and btnFreeClick event handlers, and then to test for the existence of all three objects in the btnKindClick event handler. If you experiment with writing this program, you'll see that it is very messy as the class diagram suggests (figure 5, and compare with figure 3). The association links between classes become much more complex without substitution. This in turn leads to more complex, more tightly coupled programs that become a more difficult to maintain.

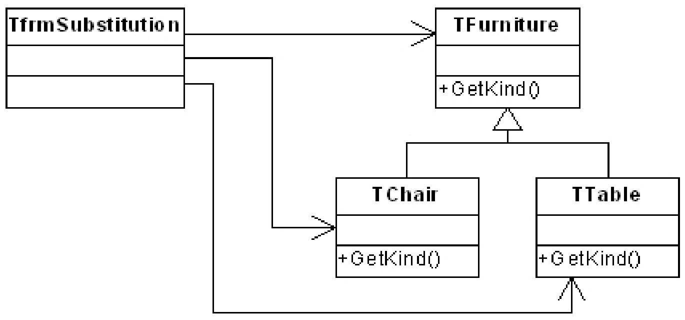


Figure 5 Association without substitution

In writing this program without substitution, the only benefit we gain from the inheritance hierarchy is that TFurniture defines a signature, ie the GetKind method, with which TChair and TTable comply.

Ex 6.2 step 2 Substitution without polymorphism

Considering that the previous step was not too successful, can we solve the problem using substitution but not dynamic binding? To do this, remove the dynamic binding in unit FurnitureU (ie revert from the code in example 6.1 step 4 to the code in step 3 by commenting out or removing the virtual and override keywords). Now replace the polymorphic message (example 6.1, step 2, line 37) with a complex If statement to test the class of the current MyFurniture instance and then call the required GetKind method (lines 34–40 below).

```
32 procedure TfrmSubstitution.btnKindClick(Sender: TObject);
33 begin
34   if MyFurniture = nil then
35     lblKind.Caption := 'Object not defined'
36   else if (MyFurniture is TChair) then
37     lblKind.Caption := (MyFurniture as TChair).GetKind
38   else if (MyFurniture is TTable) then
39     lblKind.Caption := (MyFurniture as TTable).GetKind
40   else
41     lblKind.Caption := MyFurniture.GetKind;
42 end; // procedure TfrmSubstitution.btnTypeClick(Sender: TObject)
```

What was accomplished in a single line with polymorphism (step 2 line 37) now requires a complex set of if statements with static binding. We must test explicitly for each possible instance type (using RTTI information through the is operator[1] as discussed in chapter 5) and then cast it specifically (using the as operator) to invoke the correct method.

Unlike the situation with example 6.1 step 3, these instances now bind to the required methods. At compile time, the compiler knows that MyFurniture in line 37 must be treated as a TChair because of the typecasting as operator. So in line 37, the MyFurniture object is *statically* bound to TChair's GetKind method. Similarly in line 39, the compiler has enough information at compile time to bind MyFurniture to TTable's GetKind method. Without any typecasting in line 41, MyFurniture is statically bound to TFurniture's GetKind method because MyFurniture is declared as TFurniture.

To test this, remove the typecasts in lines 37 and 39. You'll get the TFurniture's method even with TChair and TTable instances. Alternatively, if you make the wrong typecasts in lines 37 and 39 (and use the old Pascal-style typecasting which does no error checking), the static binding will bind to the wrong methods, giving the wrong message:

```
36   else if (MyFurniture is TChair) then
37     lblKind.Caption := TTable(MyFurniture).GetKind  // wrong typecast
38   else if (MyFurniture is TTable) then
39     lblKind.Caption := TChair(MyFurniture).GetKind  // wrong typecast
```

Because we still have substitution, the order of evaluation in the conditional statements is important. If we test first for the class highest in the hierarchy, in this case TFurniture, all the subclasses will also evaluate true because any subclass can substitute for its superclass.

Static binding is slightly faster than dynamic binding, and so it is sometimes suggested for speed critical operations. But all the If…ElseIf comparisons introduce their own performance penalty, counteracting the faster static binding, and introduce considerable room for error.

In summary, using static binding introduces a lot of extra coding complexity, makes the program more error prone, makes subsequent enhancement more brittle and generally does not introduce a significant speed advantage. So it is well worth the effort of getting to understand all the capabilities of polymorphism and to think out an application's inheritance structures carefully to maximise generalisation and the possibilities for substitution.

[1] RTTI is not part of the .NET specification, but Delphi 8 for .NET still makes the is, as and similar operators available.

Summary of early and late binding

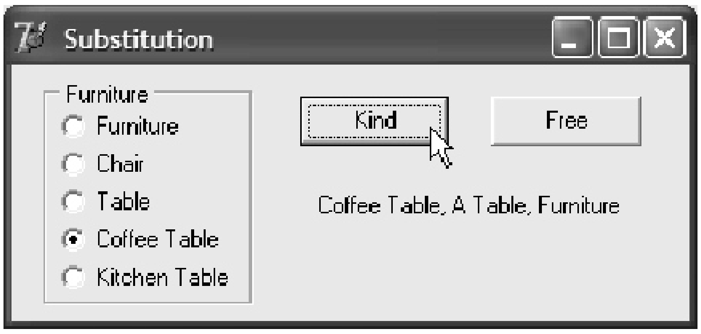
Because of substitution, the programmer can assign MyFurniture to any of three types, TFurniture, TChair or TTable. So for correct operation, Delphi must decide which of the three methods to use while the program is running, not before. This requires *late* binding (also called *run-time* binding or *dynamic* binding). (Binding refers to the link established between the object and the method). Delphi uses late binding when the virtual and override keywords are used in declaring the classes.

By default, Delphi uses *early* binding (also called *compile-time* or *static* binding). This establishes the link between an object and a method while the program is being compiled, before it runs. This means that Delphi cannot then invoke different methods depending on the type of object involved at run-time and so early binding is *non-polymorphic*.

Example 6.3 Evolution in a polymorphic program

To support the future evolution of a program, the coupling within the program should be kept low, particularly in those parts of a program that are subject to change. This is an important reason for using polymorphism, and reducing coupling through polymorphism is an important consideration in many patterns.

This example illustrates how the reduced coupling in a polymorphic program facilitates future evolutionary growth. It also looks at combining inheritance for reuse and for polymorphic behaviour. We'll suppose we need to introduce another level of specialisation into this program by subclassing TTable into TCoffeeTable and TKitchenTable (figure 6). What changes must we make?



Coffee Table, A Table, Furniture

Figure 6 Introducing two further subclasses

Ex 6.3 step 1 The additional subclasses

Our first step is to define the additional subclasses as in figure 7.

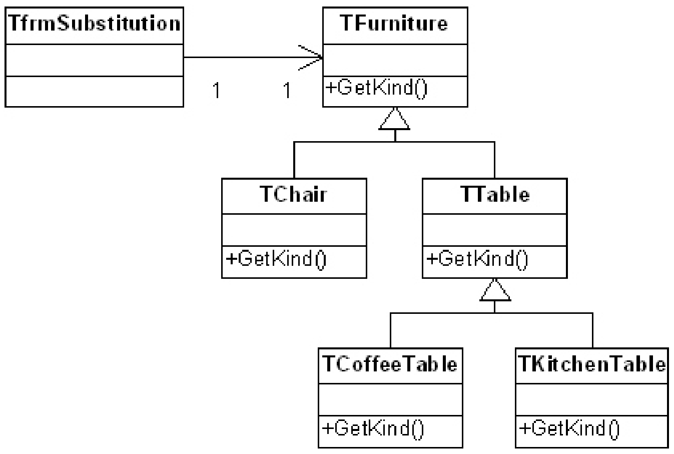


Figure 7 An additional (polymorphic) level

Start with the polymorphic version of the program (example 6.1, step 4) and add the two new subclasses (lines 16–23, 40–49 below, and remember about Class Completion <Ctrl+Shift+C>). To show the combination of inheritance for reuse (subclassing) with inheritance for polymorphism (subtyping), we have also changed the existing method definitions to invoke the immediate ancestor's method through the inherited keyword (lines 33, 38).

```
 1 unit FurnitureU;

 2 interface

 3 type
 4   {TFurniture and TChair definitions as before}

12   TTable = class (TFurniture)                             // unchanged
13   public
14     function GetKind: string; override;
15   end; // TTable = class (TFurniture)

16   TCoffeeTable = class (TTable)                                 // new
17   public
18     function GetKind : string; override;
19   end; // TCoffeeTable = class (TTable)

20   TKitchenTable = class (TTable)                                // new
21   public
22     function GetKind : string; override;
23   end; // TKitchenTable = class (TTable)

24 implementation
```

```
25 { TFurniture }

26 function TFurniture.GetKind: string;
27 begin
28   Result := 'Furniture';
29 end; // function TFurniture.GetKind: string

30 { TChair }

31 function TChair.GetKind: string;
32 begin
33   Result := 'A Chair, ' + inherited GetKind;
34 end; // end function TChair.GetKind

35 { TTable }

36 function TTable.GetKind: string;
37 begin
38   Result := 'A Table, ' + inherited GetKind;
39 end; // end function TTable.GetKind

40 { TCoffeeTable }

41 function TCoffeeTable.GetKind: string;
42 begin
43   Result := 'Coffee Table, ' + inherited GetKind;
44 end;  // end function TCoffeeTable.GetKind

45 { TKitchenTable }

46 function TKitchenTable.GetKind: string;
47 begin
48   Result := 'Kitchen Table, ' + inherited GetKind;
49 end; // end function TKitchenTable.GetKind

50 end. // end Unit FurnitureU
```

Several different things are happening here, so let's look at them one by one. First we derive two new subclasses, TCoffeeTable and TKitchenTable from TTable (lines 16–23), so adding another layer to the hierarchy (figure 7). These each have their own GetKind method declared as override, which therefore override the GetKind method they would otherwise inherit from TTable. These subclasses can use any of the methods they inherit from higher up in the hierarchy, and can override any of the inherited methods dynamically (ie using late binding to allow polymorphism) by using the override keyword provided that the overridden method is either a virtual method or is itself an override method (eg line 14). So GetKind in line 18 overrides its ancestor's method which in turn overrides it's ancestor's method (line 14).

We can override existing methods to whatever depth is needed. But what if we actually want to be able to use an ancestor's method even though it has been overridden? This turns out to be quite easy: Delphi has the inherited keyword and this allows us to access the ancestor's method. This is useful when we don't want to replace the ancestral method, but need to extend it for the descendant. In this case, we override the ancestor's method, inherit it to get its functionality and then add any code needed to extend it.

In summary, we replace an ancestral method by redeclaring the method in the subclass with exactly the same name and parameter list as in the superclass. If we want this to be polymorphic substitution using late binding, the root declaration of the method must be virtual with an override declaration each time it is subsequently redeclared lower down the hierarchy. Where we want to extend the ancestral method, we invoke it in the subclass method through the inherited keyword.

This is the theory, at any rate. We haven't seen it in action yet. So let's modify the driver program to incorporate these additional classes.

Ex 6.3 step 2 Calling additional subclasses

Extend the user interface as shown in figure 6 by adding the items CoffeeTable and KitchenTable to rgbFurniture. Extend the rgbFurnitureClick event handler as shown:

```
22 procedure TfrmSubstitution.rgbFurnitureClick(Sender: TObject);
23 begin
24   FreeAndNil (MyFurniture);               // Clean up the references
25   case rgbFurniture.ItemIndex of
26     0: MyFurniture := TFurniture.Create;
27     1: MyFurniture := TChair.Create;               // 4 substitutions
28     2: MyFurniture := TTable.Create;
29     3: MyFurniture := TCoffeeTable.Create;
30     4: MyFurniture := TKitchenTable.Create;
31   end;
32   lblKind.Caption := '';
33 end; // end procedure TfrmSubstitution.rgbFurnitureClick

34 procedure TfrmPolymorphism.btnKindClick(Sender: TObject);
35 begin
36   if MyFurniture = nil then
37     lblKind.Caption := 'Object not defined'
38   else
39     lblKind.Caption := MyFurniture.GetKind;        // runtime binding
40 end; // procedure TfrmPolymorphism.btnTypeClick(Sender: TObject)
```

The *only* programming change we have to make in the client class is to instantiate and assign these two new subclasses (lines 29–30). Because of the late (runtime) binding, *we make no change in the statement that uses these instances* (lines 34–40 are unaltered from lines 32–38 in example 6.1 step 2). This is the power of polymorphism. Each subclass knows how to do whatever it needs to do as a result of its own and its inherited methods. If these methods allow runtime binding, the program knows which method to call depending on the identity of the object. This means that if we add additional subclasses to an existing program we do not have to go through the program adding calls to these new subclasses. Because of polymorphism the correct methods will be called automatically.

In essence, the program merely needs to tell an object to do XYZ. Each different class in the hierarchy knows how it must do XYZ and so goes ahead and does it. The program using the object is not concerned with how the object performs the required operation, just that it does whatever is appropriate.

To see this in action, run the program. Select a RadioButton and then click on the Kind Button. A message identifying the selected object and its ancestors appears. Even though there is only one instruction causing the display (line 40 above), each object knows who it is and what its ancestry is and so creates a different display despite receiving the identical message.

Polymorphism simplifies the process of modifying and evolving a program and minimises the impact on the client of adding or removing classes. (The polymorphic call in line 43 above is completely unaffected by the addition or subtraction of TFurniture descendants since it relies only on the GetKind method signature.) Substitution also reduces coupling between objects since an association link is valid for the designated class and all its descendants. (The private field in TfrmSubstitution that holds a reference to TFurniture (example 6.1, step 2, line 16) enables a TfrmSubstitution instance to be associated with an instance of any of the five classes in the TFurniture hierarchy.)

Example 6.4 Abstract methods

So, this is all very nice and slick, but haven’t we chosen our example rather carefully? Yes we have, in order to concentrate on the essential principles. But there are different circumstances that we should also be able to accommodate. For example, the overridden methods were all similar in the previous example. What happens if they are very different? Let’s say our base class is called MyShape. As subclasses we have MyEllipse and MyRectangle. We want to create a Draw method. MyShape.Draw does not make any sense – what does it draw, a rectangle or an ellipse? So although every MyShape subclass that we create needs to have a Draw method, we can’t define Draw for the base class in the usual way.

In a situation like this we can use an *abstract method*. This is a method that we declare as abstract in the superclass. Being abstract means that it does not have an implementation and so does not exist in the superclass. Each subclass then provides its own implementation.

Maybe it's easier to understand if we work through an example. We'll create a TMyShape class with an abstract Draw method and two subtypes, TMyEllipse and TMyRectangle. The subtypes will have concrete Draw methods to draw either an ellipse or a rectangle (figure 8) and so override the abstract declaration in the superclass. Overriding implies dynamic binding, and so the abstract declaration must also be made virtual.

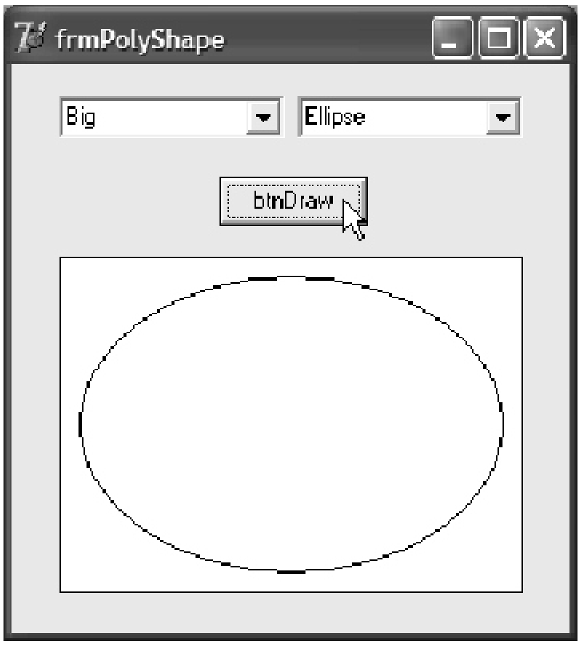


Figure 8 Illustrating a virtual, abstract method

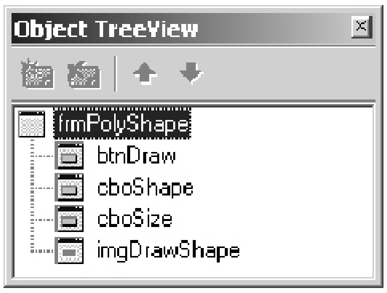


Figure 9 Objects on the user interface

Ex 6.4 step 1 Creating an abstract method

Start a new application. Add a second unit for the class definitions shown below.

```
unit MyShapesU;

interface
```

```
 3 uses ExtCtrls;

 4 type
 5   TMyShape = class (TObject)
 6   public
 7     procedure Draw (AnImage: TImage; ABorder: integer); virtual;
 8                                                         abstract;
 9   end; // end TMyShape = class (TObject)

10   TMyEllipse = class (TMyShape)
11   public
12     procedure Draw (AnImage: TImage; AnBorder: integer); override;
13   end; // end TMyEllipse = class (TMyShape)

14   TMyRectangle = class (TMyShape)
15   public
16     procedure Draw (AnImage: TImage; ABorder: integer); override;
17   end; // end TMyRectangle = class (TMyShape)

18 implementation

19 { TMyEllipse }

20 procedure TMyEllipse.Draw(AnImage: TImage; ABorder: integer);
21 begin
22   AnImage.Canvas.Rectangle(0, 0, AnImage.Width, AnImage.Height);
23   AnImage.Canvas.Ellipse (ABorder, ABorder,
24           AnImage.Width - ABorder, AnImage.Height - ABorder);
25 end; // end procedure TMyEllipse.Draw

26 { TMyRectangle }

27 procedure TMyRectangle.Draw(AnImage: TImage; ABorder: integer);
28 begin
29   AnImage.Canvas.Rectangle(0, 0, AnImage.Width, AnImage.Height);
30   AnImage.Canvas.Rectangle (ABorder, ABorder,
31           AnImage.Width - ABorder, AnImage.Height - ABorder);
32 end; // end procedure TMyRectangle.Draw

33 end. // end MyShapesU
```

As indicated in figure 10, we have a base class, TMyShape (lines 5–9), derived from TObject and two subclasses, TMyEllipse and TMyRectangle, derived from TMyShape (lines 10–17). Each of these declares a method Draw which is a procedure.

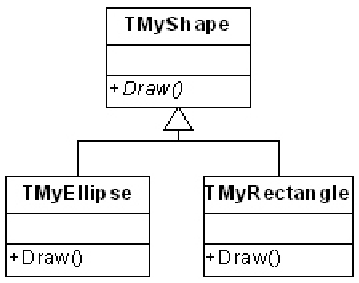


Figure 10 An abstract method in the base class

For TMyShape, method Draw is declared as *virtual* (lines 7–8), so that it can be overridden dynamically by a subclass, and as *abstract*, which means that it does not have an implementation. To show that it is abstract, TMyShape's Draw method is italicised in figure 10. Looking at its method signature given by the declaration, we see that the Draw method has two parameters, a TImage where the object must be drawn, and an integer value ABorder to specify how far from the edge of the TImage the object must be drawn. TImage is part of Delphi's VCL and displays a graphical image on a form. For more information, consult Delphi's online Help.

An implementation for Draw must be provided in every branch derived from TMyShape. Thus, in the implementation section (lines 18–33), there is an implementation for TMyEllipse.Draw (lines 20–25) and for TMyRectangle.Draw (lines 27–32). Since TMyShape.Draw is abstract it has no implementation. Methods like TMyEllipse.Draw or TMyRectangle.Draw, which implement an abstract method, are often called *concrete* methods.

Here, TMyShape establishes a *type* (only the Draw method in this example) which is implemented in the *subtypes* TMyEllipse and TMyRectangle.

Ex 6.4 step 2 Testing the MyShape classes

On the user interface (figures 8&9) place two ComboBoxes, a Button and an Image (from the Additional tab).

```
 1 unit PolyShapeU;

 2 interface

 3 uses
 4   Windows, Messages, SysUtils, Variants, Classes, Graphics,
 5   Controls, Forms, Dialogs, StdCtrls, ExtCtrls, MyShapesU;

 6 type
 7   TfrmPolyShape = class(TForm)
 8     { standard RAD declarations }

15   private
16     MyEllipse: TMyEllipse;
17     MyRectangle: TMyRectangle;
18   end;  // TfrmPolyShape = class(TForm)

19 var
20   frmPolyShape: TfrmPolyShape;

21 implementation

22 {$R *.dfm}
```

```
23 procedure TfrmPolyShape.FormCreate(Sender: TObject);
24 begin
25   MyEllipse := TMyEllipse.Create;
26   MyRectangle := TMyRectangle.Create;
27 end; // end procedure TfrmPolyShape.FormCreate

28 procedure TfrmPolyShape.btnDrawClick(Sender: TObject);
29 var
30   Border: integer;
31   MyShape: TMyShape;                        // polymorphic reference
32 begin
33   case cboSize.ItemIndex of
34     0: Border := 10;
35   else Border := 50;
36   end; // end case cboSize.ItemIndex
37   case cboShape.ItemIndex of
38     0: MyShape := MyEllipse;                          // substitution
39   else MyShape := MyRectangle;                        // substitution
40   end; // end case cboShape.ItemIndex
41   MyShape.Draw (imgDrawShape, Border);         // polymorphic message
42 end; // end procedure TfrmPolyShape.btnDrawClick

43 procedure TfrmPolyShape.FormDestroy(Sender: TObject);
44 begin
45   MyEllipse.Free;
46   MyRectangle.Free;
47   inherited;
48 end; // end procedure TfrmPolyShape.FormDestroy

49 end. // end unit PolyShapeU
```

Here we change our previous approach slightly to emphasise slightly different aspects. We declare two private data fields for frmPolyShape, one for each concrete type that we define in MyShapeU (lines 15–17) and instantiate them when we create the form (lines 26–27). Then, in btnDrawClick, we assign one or other of these to MyShape (lines 39–40), a locally declared TMyShape reference. Because of substitution, we can assign an instance of a child class to a variable of the parent class. So, depending on whether line 38 or 39 is executed, MyShape refers to either MyEllipse or MyRectangle. In line 41, we make a polymorphic call using the MyShape reference and so draw either an ellipse or a rectangle based on the assignment in the Case statement (lines 38–41). This is because the GetKind methods are declared as virtual and override methods and so use late binding. The statement MyShape.Draw would call TMyShape's GetKind method only if MyShape is instantiated as a TMyShape. Trying to do this would lead to an error because TMyShape's GetKind method is abstract (example 6.4 step 1, lines 7–8) and does not have a concrete implementation.

The MyEllipse and MyRectangle objects persist for the life of the form since we create them in the form's OnCreate event handler and free them in its OnDestroy event handler. However, we never use these directly. Instead, we assign one or other to MyShape and use that. This means that we do not have to create and free instances as we did in the previous examples in this chapter. We never instantiate MyShape directly, but use it to hold a reference to one of the objects we have created.

Strictly speaking, we don't need to free the objects in the form's OnDestroy event handler since Delphi automatically releases all memory it holds when the program terminates. However it does no harm to free these objects. Here it emphasises the create-use-free sequence in using objects that we discussed in previous chapters and the concept that if an object creates any other objects it generally has the responsibility of freeing them before it itself is destroyed.

Ex 6.4 step 3 A helper method in the superclass

Going back to the definition of TMyShape and its descendants we see that both GetKind methods have the same statement to clear the Image component and to place a border around it (example 6.4 step 1 lines 22 & 29). Although this repetition is not crucial in a small example like this, in principle we try to avoid duplication by placing repeated code into a separate method, sometimes called a *helper method* because it is for use only within the class. If we declare this method as part of the base class, as we do here in lines 9–10 below, each subclass can access it through inheritance rather than defining it separately – a nice bit of reuse.

What kind of visibility should we assign to a helper method like this? We don't want to make it public since other units would then also be able to use it, compromising encapsulation. Here we define TMyShape, TMyEllipse and TMyRectangle in the same unit. Delphi's visibility specifiers apply to a unit[2], and so we can specify the parent's helper methods as private: the subclasses can still access them. If we code the subclasses in separate units, we would have to make the helper method *protected*. This makes it accessible to the parent's class descendants, even if they are in a different unit, but not to anyone else.

Here we declare the helper method as protected in the superclass (lines 9–10) and invoke it from the subclasses (lines 24, 31).

```
1 unit MyShapesU;
2 interface
3 uses ExtCtrls;
```

[2] Delphi 8 introduces additional 'strictly private' and 'strictly protected' specifiers which refer to the class structure only without concern for the units where they are declared.

```
type
  TMyShape = class (TObject)
  public
    procedure Draw (AnImage: TImage; ABorder: integer); virtual;
                                                         abstract;
  protected
    procedure Clear (AnImage: TImage); // concrete: subclasses use it
  end; // end TMyShape = class (TObject)

  TMyEllipse = class (TMyShape)
  public
    procedure Draw (AnImage: TImage; ABorder: integer); override;
  end; // end TMyEllipse = class (TMyShape)

  TMyRectangle = class (TMyShape)
  public
    procedure Draw (AnImage: TImage; ABorder: integer); override;
  end; // end TMyRectangle = class (TMyShape)

implementation

{ TMyEllipse }

procedure TMyEllipse.Draw(AnImage: TImage; ABorder: integer);
begin
  Clear (AnImage);
  AnImage.Canvas.Ellipse (ABorder, ABorder,
      AnImage.Width - ABorder, AnImage.Height - ABorder);
end; // end procedure TMyEllipse.Draw

{ TMyRectangle }

procedure TMyRectangle.Draw(AnImage: TImage; ABorder: integer);
begin
  Clear (AnImage);
  AnImage.Canvas.Rectangle (ABorder, ABorder,
      AnImage.Width - ABorder, AnImage.Height - ABorder);
end; // end procedure TMyRectangle.Draw

{ TMyShape }

procedure TMyShape.Clear(AnImage: TImage);
begin
  AnImage.Canvas.Rectangle (0, 0, AnImage.Width, AnImage.Height);
end; // end procedure TMyShape.Clear

end. // end MyShapesU
```

Assigning references

In example 6.4 step 2 we declare an object reference (MyShape, line 32) but then don't ever instantiate it. Instead, we assign it to one or other of the two objects we do create (line 39 or

40). It's worth looking at this briefly. First we look at the memory layout before the Case statement runs (figure 11).

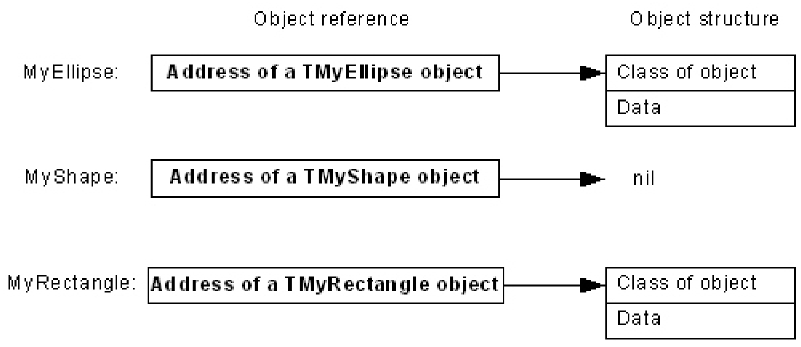


Figure 11 References after the form's OnCreate event handler has run

After the Case statement, and assuming that the user selected the Ellipse RadioButton, MyShape refers to the same instance as MyEllipse (figure 12).

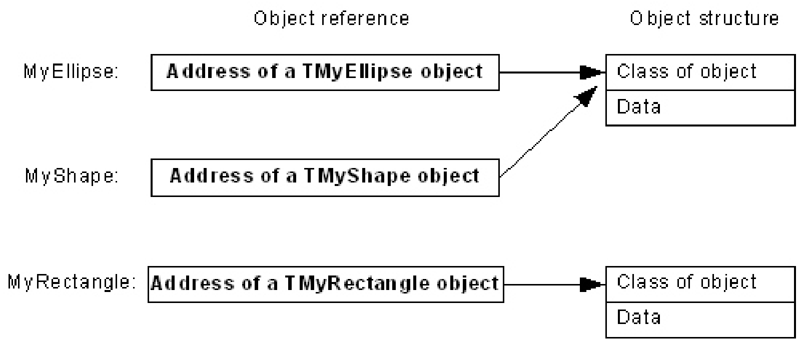


Figure 12 After assigning a MyShape reference to a MyEllipse instance

Notice two points. First, we can assign more than one reference to the same object. In figure 12, both MyEllipse and MyShape refer to the same object. We declared references to TMyEllipse, TMyRectangle and TMyShape, but only created a TMyEllipse and a TMyRectangle. In lines 39 or 40 we then assign an additional reference, MyShape, to one of these objects. It is often important to be able to assign more than one reference to an object, and this is one reason that Delphi (and Java) use *reference semantics*.

The second point to note is the principle of substitution: A variable that can hold a reference to an object of class A can also hold a reference to an object of any subclass of A.

Pattern 6.1 Polymorphism

At times, alternative behaviour is needed on the basis of an object's type. One approach to this is to use a series of If statements that test the object's type and then invoke the required behaviour. However there are several problems associated with sets of If statements. They can rapidly become complex, making it difficult to test them and to modify them, particularly if similar If evaluations are needed at different stages of a program.

Therefore,

when the required behaviour is determined by the type of an object, consider using a polymorphic call instead of using an If statement. The programmer then does not need to encode the logic for determining the type since the resulting behaviour is a product of both the method call and the receiving object. Future changes, such as additional classes of the same subtype, are accommodated by the polymorphic methods of the new classes rather than by extending If statements. Thus the types that vary have the responsibility for the correct behaviour. This reduces the possibility of introducing errors when If statements have been changed incorrectly or have been overlooked. Polymorphism therefore can make code simpler to implement and more reliable. Because polymorphism relies on substitution, and so on the generalisation hierarchies in the program, designing for polymorphism can improve the class inheritance structure and reduce coupling. (A single association link can be used to navigate between all descendants of the two associated classes.)

The drawbacks of polymorphism are that it can make code more difficult to understand and can introduce additional classes into the program structure.

Polymorphism is a crucial difference between object-oriented programming and other programming styles. Used carefully, polymorphism produces systems that are simpler and also more flexible, and so can be changed more easily to meet changing requirements. It plays an important role in many patterns, and we'll be meeting it regularly in the remaining chapters.

Difference between subclassing and subtyping

This chapter refers to the difference between subclassing and subtyping. Although this distinction is crucial to using polymorphism effectively, not all authors maintain it. These terms also sometimes mean different things to different people. We follow the Gang of Four (GoF) definitions (Gamma *et al,* 1995, p17):

'Class inheritance defines an object's implementation in terms of another object's implementation.' Thus class inheritance (or subclassing) allows for the re-use of data fields and methods, and is a mechanism for code and representation sharing. It is therefore largely a consideration that comes up during implementation and is the form of inheritance we used in the first few chapters of these notes.

‘In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another.’ Subtyping is concerned about being able to substitute a child for a parent, and consequently about polymorphism. It is largely a design consideration though its implementation needs dynamic binding.

Subtyping allows a particular type or interface to be defined in a parent class. The various child classes then implement this interface in such a way that any child may take on the role nominally assigned to the parent. Client programs declare variables of the type of the parent only so that they can remain unaware of the specific child class which is dynamically bound at any particular moment to the parent type,

An important distinction here is the difference between static and dynamic binding. With static binding, the message initiator determines how the message will be carried out through addressing a specific class (the class declared) irrespective of the actual class of the receiver. With dynamic binding, the message initiator addresses a specific type at run time and the receiver performs its (own) implementation of the message.

What makes the difference between subclassing and subtyping a little difficult to grasp at first is that both depend on inheritance. So it’s difficult to see from just a quick glance at a set of class definitions whether subclassing, subtyping or both are being used. A useful clue is given by the virtual, abstract and override keywords, which usually indicate that subtyping is being used in some way or other. As we work further through these notes, the distinction will become clearer.

Chapter 6 summary

Main points:

1. Substitution with programmer classes
2. Early (static, compile time) and late (dynamic, run time) binding
    a. Virtual and override methods
3. Polymorphism = substitution + dynamic binding
    a. Multiple associations through a single link, reduced coupling, evolvability
4. Abstract methods
5. The concept of a type
6. The Polymorphism Pattern

Objects as derived entities: Dynamic binding (virtual and override methods); abstract methods
Objects as interacting entities: Polymorphic messages and behaviour
Patterns: Polymorphism

References

Larman (2001) lists Polymorphism as one of his GRASP patterns (pp 326–329). Grand (1999) discusses the GRASP patterns, including Polymorphism (pp 69–72). Gamma *et al* (1995) emphasise that the distinction between class and type is very important in OO programming and list several patterns that depend heavily on it (p17). They discuss polymorphic iteration in particular as part of the Iterator pattern (pp 258–259).

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of reusable object-oriented software.* Addison-Wesley, Reading, MA.

Grand, M. 1999. *Patterns in Java, vol 2.* Wiley: New York.

Larman, C. 2001. *Applying UML and patterns: An introduction to object-oriented analysis and design and the Unified Process,* 2nd ed. Prentice Hall: New Jersey.

Problems

Problem 6.1 Study Chapter 6

Identify the appropriate example(s) or section(s) of the chapter to illustrate each comment made in the summary at the end of chapter 6.

Problem 6.2 No substitution

Recode example 6.1 to work without any substitution.

Problem 6.3 Order of class testing with static binding

With static binding does the order of class testing make any difference? For instance, In example 6.2 would it make any difference if the event handler were changed to the following? Explain your answer.

```
20 procedure TfrmPolymorphism.btnKindClick(Sender: TObject);
21 begin
22   if MyFurniture = nil then
```

```
    lblKind.Caption := 'Object not defined'
  else if (MyFurniture is TFurniture) then
    lblKind.Caption := (MyFurniture as TFurniture).GetKind
  else if (MyFurniture is TChair) then
    lblKind.Caption := (MyFurniture as TChair).GetKind
  else if (MyFurniture is TTable) then
    lblKind.Caption := (MyFurniture as TTable).GetKind;
end; // procedure TfrmPolymorphism.btnTypeClick(Sender: TObject)
```

Problem 6.4 Compare static and dynamic binding

Convert example 6.3 from dynamic (late) binding to static (early) binding. Comment on the differences between these two approaches.

Problem 6.5 Need for an abstract method

In example 6.4 we create an abstract method in TMyShape. An abstract method has no concrete implementation of its own and must be implemented separately in each subclass. Why declare it at all, then? Why not just remove the abstract virtual GetKind declaration that is part of TMyShape?

If you're not sure of this, try commenting out lines 7 & 8 of unit MyShapesU and see what happens.

Problem 6.6 Remuneration for Employees

Create a polymorphic program with three subclasses of Employees: fixed, weekly and contract. All fixed rate Employees receive R2,500-00 per week. Hourly Employees receive R50-00 per hour for the first forty hours worked in a week and R60-00 per hour for any extra hours. Type 1 fixed contracts are worth R1,000-00 and type 2 are worth R2,500-00.

The program’s main screen must provide some way for the user to select the Employee type (as in figure 13).

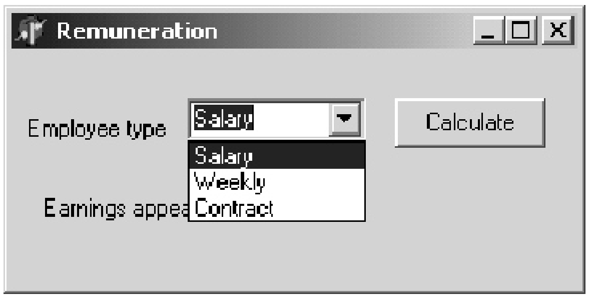


Figure 13 Selecting the type of Employee

For the fixed rate Employees the program merely displays the answer. However, for the other two the user must provide further information (eg figure 14). The program code for these input dialogues must appear in the methods that calculate the remuneration.

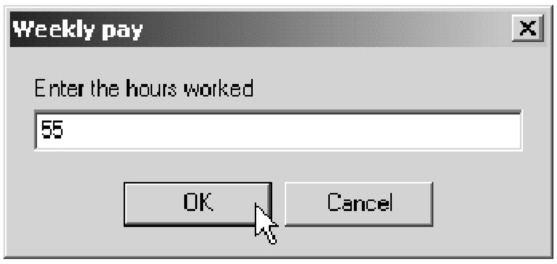


Figure 14 Entering the hours worked for the weekly paid employees

Problem 6.7 Additional examples

Imagine that you are mentoring several new programmers. They have worked through all the examples given above but are still feel unsure about several of the concepts. Work out additional examples for them to illustrate:

a) virtual / override methods, and

b) abstract methods.