OBJECT ORIENTED PROGRAMMING IN DELPHI

Pieter Blignaut

Department of Computer Science and Informatics

University of the Free State

pieterb.sci@ufs ac.za

January 2008

Table of contents

1. Basics and terminology

Object-oriented (OO) development is a way to develop software by building self-contained modules or objects that can easily be replaced, modified, and reused. It encourages a view of the world as a system of cooperative and collaborating objects. In an OO environment, software is a collection of discrete objects that encapsulate their data as well as their functionality to model real-world objects ("things").

The term object refers to a combination of data and logic that represents some real-world entity (thing).

Examples:
1. *An edit control placed on a form, named* ***`edtData`****, is a specific object; separate from other controls or even edit controls in a Delphi application.*
2. *A specific dog, Sam, is an object, a real-world entity, identifiable separate from its surroundings.*

Objects are grouped into classes. Classes are used to distinguish one type of object from another. A class is a set of objects that share common characteristics and behaviour. A single object is an instance of a class. A class is a specification of the structure (properties) and behaviour (methods) for the objects that belong to it. We have the convention of preceding class names with a capital **T**. This is a remnant from the older data-centred methodology where the word **`Type`** was used to define kinds of objects (or variables, as it was called then).

1. ***`edtData`*** *is an instance of the class* ***`TEdit`****, the template on which all edit controls in Delphi are based.*
2. *Sam is an instance of the class* ***`TDog`****. Sam has all the attributes that other dogs have and exhibit the same behaviour.*

The chief role of a class is to define the properties and methods of its instances. Each object of a class has **properties** (data describing its appearance) and **methods** (functions or procedures). The properties and methods of a class are collectively known as its **members**.

Properties describe the state of an object, i.e. how it looks like. Methods define an object's behaviour, i.e. what it can do.

1. *In Delphi, the class* ***`TEdit`*** *defines the property* ***`Height`****. For* ***`edtData`****, the value of this property could be 21. Other properties are* ***`Length`****,* ***`Top`****,* ***`Name`****,* ***`Text`****, etc. that characterises a specific instance of* ***`TEdit`*** *it and make it different from other edit controls.*
2. *The class* ***`TDog`*** *defines the property* ***colour****. For Sam, the value of this property is* ***brown****. Other properties that will distinguish Sam from other dogs and animals: could be colour, weight, name, owner, etc.*

Methods describe what an object is capable of doing or things that can be done to it. A method is defined as part of the class definition.

- Methods are implemented in functions or procedures.
- There are four kinds of methods: Accessors will only access the private members of a class and return their values while mutators can change the values of the private members. Constructors and destructors will be discussed later.

1. *The class **`TEdit`** contains a method, **`Clear`**, that will delete all text from the instance that will invoke it.*
2. *The class **`TDog`** contains the code that will instruct the computer's speakers to imitate a bark. Now all instances of **`TDog`**, including Sam, can bark.*

When an object is represented in a software program, the object is responsible for itself. In OO systems the focus is on the objects themselves and not on the system functionality.

Sam, the software object, must maintain its own attributes, e.g. name, colour, owner, etc. Instead of saying, "System, set the colour of Sam to brown", Sam is instructed "Sam, set your colour to brown". I.e. the functionality to assign Sam a colour is embedded inside the object and is not separate from the object.

Another consequence of the fact that an object is responsible for itself is that all validity checks must be done inside the methods. That means that the programmer who uses the class should not have to write complicated code or utilise extensive conditional checks through the use of case statements to ensure that an object's status is valid.

The instruction "Sam, set your colour to green", must be rejected by Sam since no dog can be green. It should not be necessary to code (algorithmically)

```
If sColour <>  'Green' then
   "Sam, set your colour to " + sColour
end if
```

Methods **encapsulate** (enclose) the behaviour of an object, provide interfaces to the object and hide any of the internal structures and states maintained by the object. Consequently, procedures and functions provide us the means to communicate with an object and access its properties. This is considered as good programming style since it limits the impact of any later changes to the representation of the properties. In Delphi, encapsulation, or **information hiding** as it is sometimes called, is done with **public** and **private** members. Public members may be accessed from any module where its class can be referenced. A private member is invisible outside of the unit or program where its class is declared. In other words, a private method cannot be called from another module, and a private field or property cannot be read or written to from another module.

*In **`TDog`**, the colour property should be private. If it was not the case, it would have been possible to set Sam's colour from anywhere in the system, overriding the built-in validation check to ensure that the colour may not be green. The method that contains this check and is used to access the colour should, however, be public of course.*

It is very important to note that a variable of a class type is actually a pointer. In other words, when we declare:

```
var Sam, Max : TDog;
```

Sam and Max are pointers, i.e. they hold only memory addresses and not the entire objects. Sam and Max point to two different places in memory where the actual objects with all the details of Sam and Max respectively, can be found. Both objects belong to the class (type) **`TDog`**.

2. <u>Implementation of a class in Delphi</u> (See Dog1 in the code pack)

Now it's time to put our knowledge into practice. Let's implement the class **`TDog`** in Delpi:

- Start with a new application
- Save your application and all associated modules in a separate folder, giving them appropriate names. I saved mine in **D:\Delphi projects\Dog1** with project name **Dog** and form **fDataEntry**.
- It is good programming practise to define every class in a separate unit. Add a new unit to your application (Click on **File/New...**, select **Unit** and click on **OK)**. Save the unit as **clsTDog**.

- Now look at the following code and ensure that your unit looks similar. You will get an error if you try to compile the unit at this time since we did not yet finish the **`implementation`** section.

```
unit clsTDog;
interface
type
  //Class definition
  TDog = class (TObject)
    private
      FName   : String;
      FColour : String;
      FWeight : Real;
    published
      property Name: String read FName write FName;
      property Colour: String read FColour write FColour;
      property Weight: Real read FWeight write FWeight;
    public
      procedure Bark;
    end;

implementation
end.
```

<u>Note the following</u>:

a. The class is defined under **`type`** in the **`interface`** section of the unit. Therefore, the class and all public members will be known to the rest of the system.
b. Our class, **`TDog`**, is a descendant of the general class, **`TObject`**, in Delphi. Therefore, we add **`TObject`** in brackets next to the key word **`class`** in the definition above. This means that **`TDog`** inherits all methods of **`TObject`**, including **`Create`** that will be used later to instantiate an instance of the class.
c. The private members are declared under a **`private`** section of the class. It is convention to prefix all properties with a capital '**`F`**'.
d. The methods are defined under **`public`**.
e. The properties are specified with the keyword **`property`** under the **`published`** section of the class. The properties are used to access the private members of the class.
   - *Published* members have the same visibility as public members. The difference is that an application can query the private members of an object through the published members.
   - A property whose declaration includes only a **`read`** specifier is a read-only property, and one whose declaration includes only a **`write`** specifier is a write-only property. It is an error to assign a value to a read-only property or use a write-only property in an expression.

- The implementation details of the methods are hidden from the rest of the system under **`implementation`**:

```
  uses MMSystem {for PlaySound};

  procedure TDog.Bark;
  begin
    PlaySound('DogBark.wav',0,snd_Async);
  end;
```

<u>Note the following</u>:

a. **`PlaySound`** is a function that plays the sound recorded in the file specified as first parameter. The second parameter is 0 except in special circumstances (read in **Help** about this if you are interested). The third parameter indicates whether the sound is played while program execution carries on (SND_ASYNC) or whether execution should wait for the sound to finish before it carries on (SND_SYNC).

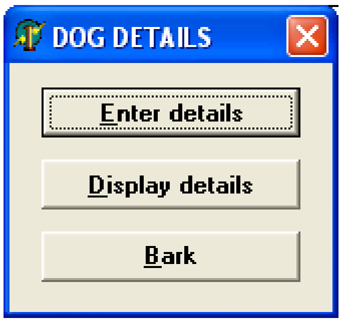
b. The unit **`MMSystem`** must be included in **`uses`** to enable **`PlaySound`**.

c. The file **DogBark.wav** must be saved in the same folder where the application is saved.

d. A common error that many programmers make is to omit the class name as qualifier for the methods in the **`implementation`** section:

```
     procedure Bark;   //Wrong!!
```

- Now, we can create a user interface to set and display the object's properties. You may add three buttons to the data entry form (see figure to the right) and add the following code to the interface and implementation sections of the form:



```
unit fDataEntry;
interface
  uses ... , clsTDog;

  type
    TfrmDataEntry = class(TForm)
    ...
    private
      { Private declarations }
      theDog : TDog;
    public
      { Public declarations }
    end;

implementation
{$R *.DFM}

  procedure TfrmDataEntry.FormCreate(Sender: TObject);
  begin
    theDog := TDog.Create;
  end;

  procedure TfrmDataEntry.btnEnterClick(Sender: TObject);
  begin
    theDog.Name   := InputBox('DATA ENTRY','Give the dog''s name:', theDog.Name);
    theDog.Colour := InputBox('DATA ENTRY','Give the dog''s colour:', theDog.Colour);
    theDog.Weight := StrToFloat(InputBox('DATA ENTRY','Give the dog''s weight:',
                                         FloatToStr(theDog.Weight)));
  end;
  procedure TfrmDataEntry.btnDisplayClick(Sender: TObject);
  var sMsg : String;
  begin
    sMsg := 'Name : ' + theDog.Name + chr(13);
    sMsg := sMsg + 'Colour : ' + theDog.Colour + chr(13);
    sMsg := sMsg + 'Weight : ' + FloatToStr(theDog.Weight);
    MessageBox(0,PChar(sMsg),'DOG DETAILS',MB_OK);
  end;

  procedure TfrmDataEntry.FormDestroy(Sender: TObject);
  begin
    theDog.Free;
  end;

  procedure TfrmDataEntry.btnBarkClick(Sender: TObject);
  begin
    theDog.Bark;
  end;

end.
```

Note the following:

a. The class definition of **TDog** is made available to this unit, **fDataEntry**, by including the unit in which it is defined, **clsTDog**, under **uses**.

b. A private object, **theDog**, of class **TDog** is declared in the **private** section of the form. Unlike a declaration at the start of a procedure of function, this declaration goes without the keyword **var** but it means the same thing: Reserve a space in memory, named **theDog**, where an object of type **TDog** will be saved. Note again that the variable name **theDog** contains a pointer and not the entire object.

c. Since the object is declared under **private**, it will be known to all procedures and functions of this form but not to anything outside the form.

d. The prefix **the** is used in front of object variable names. This indicates that we are dealing here with a specific instance of the class. In many references you will encounter the prefix **my** – in other words the object variable name will by **myDog**. We prefer **the**.

e. Since class variables are actually pointers that reference the memory space where the actual object data is saved, the objects have to be created (instantiated) before they can be used. This is done as the very first thing when the form is created.

   Hint: If you forget to instantiate an object you will get an awkward run-time error message that says "Access violation at address ...".

f. Note the way in which the class properties and methods are called. The object name is followed by a period and then the name of the method (procedure or function). We have used this notation from day 1 of your Delphi exposure when we referred to the properties of the visual components, e.g. **edtData.Text**.

g. Once we are done with an object, the memory that was assigned to it with **Create** must be released again. Therefore, the **Free** method of **TObject** and inherited by **TDog** was invoked in the **FormDestroy** event handler.

3. Input validation (See Dog2 in the code pack)

As said in the introduction, a class is responsible for its own well-being. It should, for example, not be necessary for a programmer that uses the class to ensure that user-input is valid. Instead of accessing a private member directly via a property, the **`write`** specifier of a property can be used to access the private member through a procedure in the **`private`** section of the class that will check that only valid data will be written into the private member.

- First, we have to change the class declaration a bit:

```
TDog = class (TObject)
  private
    FName   : String;
    FColour : String;
    FWeight : Real;
    procedure SetColour (Value : String);
    procedure SetWeight (Value : Real);
  published
    property Name: String read FName write FName;
    property Colour: String read FColour write SetColour;
    property Weight: Real read FWeight write SetWeight;
  public
    procedure Bark;
  end;
```

Note the following:
a. Two procedures (methods) are declared under **`private`**.
b. The **`write`** specifiers of the **`Colour`** and **`Weight`** properties now indicates that the two above-mentioned methods will be called when write access is requested.

- Next we implement the two mutator methods under **`implementation`**:

```
implementation

  uses MMSystem {for PlaySound}, SysUtils {for UpperCase}, Windows {for MessageBox};

  procedure TDog.Bark;
  begin
    PlaySound('dogBark.wav',0,snd_Async);
  end;

  procedure TDog.SetColour (Value : String);
  var sMsg : String;
  begin
    if (UpperCase(Value) = 'BLACK') or (UpperCase(Value) = 'BROWN')
        or (UpperCase(Value) = 'YELLOW') or (UpperCase(Value) = 'WHITE') then
      FColour := Value
    else
    begin
      sMsg := 'Choose colour from ''White'', ''Black'', ''Brown'' or ''Yellow''.';
      sMsg := sMsg + chr(13) + 'The colour was not changed.';
      MessageBox (0,PChar(sMsg),'INVALID INPUT', MB_ICONERROR + MB_OK);
    end;
  end;
```

```
procedure TDog.SetWeight (Value : Real);
  var sMsg : String;
  begin
    if Value > 0 then
      FWeight := Value
    else
    begin
      sMsg := 'Invalid weight.' + chr(13) + 'The weight was not changed.';
      MessageBox (0,PChar(sMsg),'INVALID INPUT', MB_ICONERROR + MB_OK);
    end;
  end;

end.
```

<u>Note the following</u>:

a. We had to include some of the units from Delphi's library to make sure that some of the pre-defined VCL procedures and functions of Delphi will work.

b. The private members, **`FColour`** and **`FWeight`**, are assigned values inside the mutator methods after the incoming value has been validated.

- You would have noticed above that no validation was done to ensure that the user enters a valid numeric value for the dog's weight. The private member, **`FWeight`**, is real and the **`SetWeight`** mutator takes a real argument. If the end-user enters an invalid numeric value, the program will return with a run-time error. Unforgiveable: it is not a stupid user, but an inconsiderate programmer. Instead of writing to a private member through a property, we can use a separate mutator.

  First, we have to edit our class definition a bit:

```
  TDog = class (TObject)
    private
      ...
      procedure SetWeight (Value : Real); overload;
    published
       ...
      property Weight: Real read FWeight write SetWeight;
    public
      ...
      procedure SetWeight (Value : String); overload;
    end;
```

  Then, we must define the new mutator under **`implementation`**:

```
  procedure TDog.SetWeight (Value : String);
  var iErrorCode : Integer;
      rWeight : Real;
      sMsg : String;
  begin
    Val(Value, rWeight, iErrorCode);
    if (iErrorCode = 0) and (rWeight > 0) then
      FWeight := rWeight
    else
    begin
      sMsg := 'Invalid weight.' + chr(13) + 'The weight was not changed.';
      MessageBox (0,PChar(sMsg),'INVALID INPUT', MB_ICONERROR + MB_OK);
    end;
  end;
```

Of course, the way in which the mutator is called will change:

```
procedure TfrmDataEntry.btnEnterClick(Sender: TObject);
var sWeight : String;
begin
  theDog.Name   := InputBox('DATA ENTRY', 'Give the dog''s name:', theDog.Name);
  theDog.Colour := InputBox('DATA ENTRY', 'Give the dog''s colour:', theDog.Colour);
//  theDog.Weight := StrToFloat(InputBox('DATA ENTRY', 'Give the dog''s weight:',
//                                       FloatToStr(theDog.Weight)));
  sWeight := InputBox('DATA ENTRY', 'Give the dog''s weight:', FloatToStr(theDog.Weight));
  theDog.SetWeight(sWeight);
end;
```

Note the following:

a. There are now two mutators with exactly the same name. This is called overloading and must be indicated as such through the **`overload`** keyword. The compiler will decide, based on the actual parameter list and the way in which the method is used, which implementation to use.

b. The new mutator takes a string argument and a conversion to **`Float`** (**`Real`**) is done before the assignment to the private member. This ensures that the user of the class would not need to ensure valid input by the end-user – it is done by the class.

c. Of course, it would have been possible to remove the previous implementation **`SetWeight`** altogether. In this case there is no need for the **`overload`** keyword and the property becomes read-only:

```
property Weight: Real read FWeight;
```

4. Constructors and destructors (See Dog2 in the code pack)

Besides accessors and mutators, we get two other kinds of methods, namely **constructors** and **destructors**.

A **constructor** is a class member function that is applied to a class to allocate memory for an instance of that class. If a constructor (**`Create`** method) is not explicitly declared in the class declaration, the **`Create`** method of **`TObject`** is inherited and may be used as a **default constructor**. The default constructor only creates the object – it does not initialise the object's data members. We should, however, always declare the constructor explicitly in a class declaration and initialise **all** data members. This is done as follows

```
unit clsTDog;
interface
type
  TDog = class (TObject)
    private
      ...
    public
      constructor Create;
      ...
    end;

implementation

  constructor TDog.Create;
  begin
    inherited Create;
    sName   := 'Not set';
    sColour := 'Not set';
    rWeight := 0;
  end;

 .. .. ..

end.
```

Note the following:

a. The keyword **`constructor`** is used where **`procedure`** or **`function`** is normally used.
b. The line **`inherited Create;`** in the implementation ensures that the full functionality of **`TObject.Create`** is included in **`TDog.Create`**.
c. You could check that the constructor is executing properly by clicking the **Display** button of our example before data entry.
d. The constructor must be **`public`**.
e. It is sometimes convenient to have more than one (possible overloaded) constructor in order to construct objects from different parameter sets.

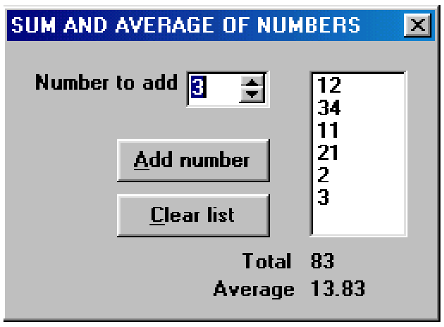
Just as a constructor call allocates memory for an object, a **destructor** call frees the memory. It is seldom necessary to write a custom destructor – the default **`TObject.Destroy`** can be used straight away. Instead of calling **`Destroy`** directly, however, a program should call **`Free`** which calls **`Destroy`** only if the object exists.

5. Use of predefined classes makes things easier

(See Average1 and Average2 in the code pack)

Let's leave Sam for the moment and focus our attention on a more abstract object. Consider the following problem scenario: The user wants to enter several numbers one at a time. The program must also display the arithmetic sum (total) as well as the average of these numbers. These values must be updated continuously as new numbers are entered.

Look at the form printed here. A spin edit is used to capture the numbers. A number is added to a list box each time when the user clicks on the **Add number** button.



Now, look at the following implementation of the **btnAddClick** event handler (see **Average1** in the code pack):

```
procedure TfrmSumAverage.btnAddClick(Sender:
TObject);
var i        : Integer;
    iSum     : Integer;
    rAverage : Real;
begin
  //Add number to list
  lstbxNumbers.Items.Add (IntToStr(spinedtNumber.value));
  //Traverse list and calculate sum of numbers
  iSum := 0;
  for i := 0 to lstbxNumbers.Items.Count-1 do
    iSum := iSum + StrToInt(lstbxNumbers.Items.Strings[i]);
  //Calculate average
  rAverage := iSum/lstbxNumbers.Items.Count;
  //Display results
  lblTotal.Caption := IntToStr(iSum);
  lblAverage.Caption := FloatToStrF(rAverage,ffFixed,6,2);
end;
```

Being an experienced programmer by now, you should have been able to think of a solution in more or less the same lines yourself. Consider, however, the following alternative:

- You are not that experienced and are not very sure how to use a **for**-loop to add numbers.
- You know that there is a class available, **TNumbers**, that does this for you and that it has the following methods:

```
procedure Add(Value: Integer);
procedure Clear;
function TotalAsString : String;
function AverageAsString : String;
```

The availability of this class simplifies the **btnAddClick** event handler considerably (see **Average2** in the code pack):

```
unit uAverage;
interface
uses ..., clsTNumbers;
type
  TfrmSumAverage = class(TForm)
    ...
  private
    { Private declarations }
    theNumbers : TNumbers;
  public
    { Public declarations }
  end;
implementation

procedure TfrmSumAverage.FormCreate(Sender: TObject);
begin
  //Create object
  theNumbers := TNumbers.Create;
end;

procedure TfrmSumAverage.btnAddClick(Sender: TObject);
begin
  //Add number to list
  lstbxNumbers.Items.Add (IntToStr(spinedtNumber.Value));
  //Add number to theNumbers
  theNumbers.Add(spinedtNumber.Value);
  //Display results
  lblTotal.Caption := theNumbers.TotalAsString;
  lblAverage.Caption := theNumbers.AverageAsString;
end;

procedure TfrmSumAverage.FormDestroy(Sender: TObject);
begin
  theNumbers.Free;
end;
```

- Note the following:
  a. The programmer must not forget to include the unit in which the class is defined (**`clsTNumbers.dcu`**) in his form and to declare and create the object.
  b. The details of the implementation are completely hidden form the class user, i.e. the in-experienced programmer. He does not need to know or understand how the calculations are done. He does not even have to convert the numeric values to string or format them.

By the way, since you are an experienced programmer, you might be curious about how the implementation was done:

```
unit clsTNumbers;

interface
  type
    TNumbers = class (TObject)
      private
        iTotal : Integer; iCount : Integer;
      public
        procedure Add(Value : Integer);
        procedure Clear;
        function TotalAsString : String;
        function AverageAsString : String;
    end;
```

```
implementation

  uses SysUtils {for FloatToStrF};

  procedure TNumbers.Add (Value : Integer);
  begin
    iTotal := iTotal + Value;
    iCount := iCount + 1;
  end;

  procedure TNumbers.Clear;
  begin
    iCount := 0; iTotal := 0;
  end;

  function TNumbers.TotalAsString : String;
  begin
    Result := IntToStr(iTotal);
  end;

  function TNumbers.AverageAsString : String;
  begin
    Result := FloatToStrF(iTotal/iCount,ffFixed,6,2);
  end;
end.
```

6. Class hierarchy and inheritance

6.1 Introduction

An object-oriented system organises classes into a **sub-class-super class hierarchy**. Different properties and behaviours are used as the basis for making distinctions between classes and subclasses. At the top of the class hierarchy are the most general classes and at the bottom are the most specific.

1. *In Delphi,* **`TCustomEdit`** *is a super class of* **`TEdit`** *and* **`TMemo`**.
   (Hint: Search for help on **`TEdit`** *and then click on Hierarchy to examine its class hierarchy)*
2. **`TPet`** *would be a super class for* **`TDog`**, **`TParrot`**, *etc.*

A class may simultaneously be the subclass to some class and a super class to another class(es).

1. *In Delphi,* **`TCustomEdit`** *is a subclass of* **`TWinControl`** *while it is a super class of* **`TEdit`** *and* **`TMemo`**.
2. **`TPet`** *could be a subclass of* **`TAnimal`** *while it is a super class of* **`TDog`**.

Inheritance allows objects to be built from other objects. It is a relationship between classes where one class is the parent class (also called base-class or super-class) of another (derived or sub-) class. A subclass inherits all of the properties and methods defined in all of its super-classes. Subclasses generally add new methods and properties that are specific to that class. Subclasses may refine or constrain the state and behaviour inherited from its super class.

The real advantage of inheritance is the fact that we can build on and reuse what we already have.

1. *In Delphi,* **`TControl`** *defines the property* **`Name`** *that is applicable to all controls while* **`TCustomEdit`** *defines a property* **`BorderStyle`** *that is not applicable to, for instance,* **`TButton`** *or* **`TLabel`**.
2. *If a button is placed on a form, the button's* **`Font`** *property can be inherited from the form. (Hint: Put several edit controls on a form and then play around with their individual* **`ParentFont`** *properties while changing the form's font.)*
3. **`TPet`** *would define properties such as* **`Name`** *and* **`Colour`** *which are applicable to all of its subclasses.* **`TDog`** *and* **`TParrot`** *inherit the properties* **`Name`** *and* **`Colour`** *from* **`TPet`**. *These properties need not be defined again in each one of the sub-classes. On the other hand,* **`TParrot`** *could have a property such as* **`CanSpeak`** *that would not be applicable to* **`TDog`** *and* **`TDog`** *will have a method* **`Bark`** *that is not applicable to* **`TParrot`**.

6.2 Perspective of a programmer implementing a class hierarchy in Delphi (See Pets in the code pack)

- Study the following implementation of the class **TPet**:

```
unit clsTPet;

interface

uses Windows {for MessageBox}, SysUtils {for UpperCase};

type
  TPet = class (TObject)
    private
      FName   : String;
      procedure SetColour(Value : String); overload;
    protected
      FColour : String;
    published
      property Name: String read FName write FName;
      property Colour: String read FColour write SetColour;
    public
      constructor Create;
  end;

implementation

  constructor TPet.Create;
  begin
    inherited Create;
    FName := '';
    FColour := '';
  end;  //Create

  procedure TPet.SetColour(Value : String);
  begin
    FColour := Value;
  end; //SetColour

end.
```

Note the following:

a. The members, **FName** and **FColour**, which are common to all kinds of pets, are declared in the base class as well as their associated properties.

b. A third category of information hiding is used here, **protected**, that ensures that class members can be accessed from within the class itself as well as its subclasses. Since **TDog** has special validation rules for the **FColour** property, **TDog** will need to access the property as declared in the base class, **TPet**.

c. The method **SetColour** is declared with **overload** since another method will be defined in **TDog** with the same name that will be applicable when the pet is a dog.

- Now, have a look at the following implementation of the sub-class **TParrot**:

```
unit clsTParrot;

interface

uses SysUtils {for UpperCase}, clsTPet;

type
  TParrot = class (TPet)
    private
      FCanSpeak : Boolean;
    public
      constructor Create;
      procedure SetSpeak (Value:String);
      function CanSpeak : String;
  end;

implementation

  constructor TParrot.Create;
  begin
    inherited Create;
    FCanSpeak := false;
  end;

  procedure TParrot.SetSpeak (Value:String);
  begin
    FCanSpeak := UpperCase(Value) = 'YES';
  end;

  function TParrot.CanSpeak : String;
  begin
    if FCanSpeak then
      Result := 'Yes'
    else
      Result := 'No';
  end;

end.
```

Note the following:

a. **clsTPet** was included under **uses** in the **interface** section to enable inheritance from **TPet** (line 3).

b. **TParrot** is a descendant of **TPet** which is indicated in brackets in the class definition header (line 5). This means that **TParrot** inherits the properties **FName** and **FColour** from **TPet**. An extra property, **CanSpeak** which is specific to parrots, is added here.

c. The constructor includes the line **inherited Create;**. This causes **TPet.Create** to be executed as well so that the inherited properties are initialised also.

d. Although the property **FCanSpeak** is **Boolean**, the corresponding accessor and mutator methods accept and return the **String** values **Yes** or **No** which may be more readable for a user during run-time.

- Now, look at the following implementation of the sub-class **TDog**: The code that stayed the same from our previous implementation is indicated with … .

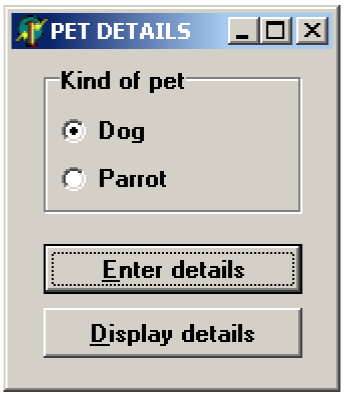
```
unit clsTDog;

interface

uses clsTPet;

type
  TDog = class (TPet)
    private
      FWeight : Real;
      procedure SetColour (Value : String); overload;
    published
      property Colour: String read FColour write SetColour;
      property Weight: Real read FWeight;
    public
      constructor Create;
      procedure Bark;
      procedure SetWeight (Value : String);
    end;

implementation

  uses ...

  constructor TDog.Create;
  begin
    inherited Create;
    FWeight := 0;
  end;

  procedure TDog.Bark;
  ...

  procedure TDog.SetColour (Value : String);
  ...

  procedure TDog.SetWeight (Value : String);
   ...
end.
```

Note the following:

a. The method, **procedure Colour**, is repeated here even though it is defined in **TPet**. This is because there is some extra functionality (validation checks in this case) that need to be done for **TDog** that is not the case with all pets. Therefore, when called from an instance of **TDog**, this more specific method will be executed, overriding the previous implementation in **TPet**. This principle is called **polymorphism** in an OO context. The keyword **overload** is again added to distinguish this method (mutator) from the accessor **function Colour**, which is still available through **TPet**.

6.3 Perspective of a programmer using a class hierarchy

The data entry form was implemented as follows to accommodate two kinds of pets, namely dogs and parrots. The `implementation` section of this form's unit is given here. Study this code and make sure that you understand everything.



```
unit fDataEntry;

interface

uses ... , clsTDog, clsTParrot;

type
  TfrmDataEntry = class(TForm)
    ...
  private
    { Private declarations }
    theDog    : TDog;
    theParrot : TParrot;
  public
    { Public declarations }
  end;

var
  frmDataEntry: TfrmDataEntry;

implementation

procedure TfrmDataEntry.FormCreate(Sender: TObject);
begin
  theDog := TDog.Create;
  theParrot := TParrot.Create;
end; //FormCreate

procedure TfrmDataEntry.btnEnterClick(Sender: TObject);
var sWeight   : String;
    sCanSpeak : String;
begin

  if radbtnDog.Checked then
  begin
    with theDog do
    begin
      Name    := InputBox('DATA ENTRY', 'Give the dog''s name:', Name);
      Colour  := InputBox('DATA ENTRY', 'The dog''s colour:', Colour);
      sWeight := InputBox('DATA ENTRY', 'The dog''s weight:', FloatToStr(Weight));
      SetWeight (sWeight);
    end;
  end;

  if radbtnParrot.Checked then
  begin
    with theParrot do
    begin
      Name := InputBox('DATA ENTRY', The parrot''s name:', Name);
      Colour := InputBox('DATA ENTRY', The parrot''s colour:', Colour);
      sCanSpeak := InputBox('DATA ENTRY', 'Can the parrot speak? (Yes/No) :', Speak);
      SetSpeak (sCanSpeak);
    end;
  end;

end; //btnEnterClick
```

```
procedure TfrmDataEntry.btnDisplayClick(Sender: TObject);
var sMsg : String;
begin

  if radbtnDog.Checked then
  begin
    sMsg := 'Name : ' + theDog.Name + chr(13);
    sMsg := sMsg + 'Colour : ' + theDog.Colour + chr(13);
    sMsg := sMsg + 'Weight : ' + FloatToStr(theDog.Weight);
    MessageBox(0,PChar(sMsg), 'DOG DETAILS', MB_OK);
  end;

  if radbtnParrot.Checked then
  begin
    sMsg := 'Name : ' + theParrot.Name + chr(13);
    sMsg := sMsg + 'Colour : ' + theParrot.Colour + chr(13);
    sMsg := sMsg + 'Can speak : ' + theParrot.Speak;
    MessageBox(0,PChar(sMsg), 'PARROT DETAILS', MB_OK);
  end;

end; //btnDisplayClick

procedure TfrmDataEntry.FormDestroy(Sender: TObject);
begin
  theDog.Free;
  theParrot.Free;
end; //FormDestroy

end.
```

Note the following:

a. The two objects, **theDog** and **theParrot**, is declared in the private section of the form class.

b. The two objects are instantiated in the forms **OnCreate** event handler and released in the form's **OnDestroy** event handler.

c. Two different sets of code are executed for data entry and displayed for the two different sub-classes of **TPet**.

d. The **with** structure enables the programmer to refer to an object's public members without having to repeat the object variable time and again. In other words,

   ```
   theDog.Colour (sColour)
   ```

   can be replaced with

   ```
   with theDog do Colour(sColour).
   ```

e. Run the program from the code pack and make sure that you understand that two different versions of the mutator **SetColour** are executed for **theDog** and **theParrot**.

7. Object persistence

Persistence refers to the ability of some objects to outlive the programs that created them. The most object-oriented languages do not support persistence, which is the process of writing or reading an object to and from a persistent storage medium, such as a disk file. Persistent object stores do not support query or interactive user-interface facilities, as found in fully supported object-oriented database management systems.

It is, however, not too difficult to store an object's data members in a text file. The following example picks up on **Dog2** (See **Dog3** on the code pack).

- We can modify the class definition of **TDog** as follows:

```
TDog = class (TObject)
  private
    .......
  public
    ....
    procedure Read (sFilename : String);
    procedure Save (sFilename : String);
  end;
```

Note the following:

a. Two methods were added: one to read the object's data from file and another to write it to file.
b. Both these methods expect a string parameter to indicate the file name where the data is/should be saved.

- The implementation of these added methods may look as follows:

```
procedure TDog.Save (sFilename : String);
var f : TextFile;
begin
  AssignFile (f, sFilename);
  Rewrite (f);
  Writeln (f, FName);
  Writeln (f, FColour);
  Writeln (f, FWeight:8:2);
  CloseFile (f);
end;

procedure TDog.Read (sFilename : String);
var f : TextFile;
begin
  AssignFile (f, sFileName);
  {$I-} Reset (f); {$I+}
  If IOResult = 0 then
  begin
    Readln (f, FName);
    Readln (f, FColour);
    Readln (f, FWeight);
    CloseFile (f);
  end;
end;
```

Note the following:

a. The normal text file operations are applicable.
b. Checking for the existence of the file is hidden inside the object method.

- These methods may be called as follows from the data entry form:

```
procedure TfrmDataEntry.FormCreate(Sender: TObject);
var sPath : String;
begin
  theDog := TDog.Create;
  sPath  := ExtractFiledir(Application.ExeName);
  theDog.Read(sPath + '\Dog.txt');
end;

procedure TfrmDataEntry.FormDestroy(Sender: TObject);
var sPath : String;
begin
  sPath  := ExtractFiledir(Application.ExeName);
  theDog.Save(sPath + '\Dog.txt');
  theDog.Free;
end;
```

Note the following:

a. The code to read an object's data members automatically at start-up was added to the existing **`FormCreate`** event handler. Instead of having it here, this code could also have been entered into the **`OnClick`** event handler of a button or menu item, forcing the user to explicitly instruct the object to read its detail from the file.
b. The code to save the object's data members is entered into the form's **`FormDestroy`** event handler just before the object is released from memory. This will ensure that the user do not have to remember to save the data. Of course, it would be possible to enter this code into the **`OnClick`** event handler of a button or menu item again.
c. The code could also be adapted to expect the user to enter the path and filename instead of accepting the default path to be the same as that of the application and hard coding the file name.
d. Run the example in the code pack (**Dog3**) and inspect the contents of **`Dog.txt`** with a text editor such as Notepad.

8. Advantages of OO

Object oriented development enables us to create sets of objects that work together synergistically to produce software that are easier to adapt to changing requirements, easier to maintain, more robust and promotes greater design and code reuse. Object-oriented development allows us to create modules of functionality. Once objects are defined, it can be taken for granted that they will perform their desired functions and you can seal them off in your mind like black boxes, i.e. you may focus on *what* they are doing and not worry about *how* they are doing it.

An object orientation yields important benefits to the practice of software construction:

- OO supports a high level of abstraction, i.e. the system is simplified because it is broken down into sub-parts. Objects encapsulate (isolate) both data (attributes) and functions/procedures (methods) and therefore development can proceed at the object level and ignore the rest of the system as long as necessary. Several programmers can work on the same system simultaneously, each one focusing on his/her own object only.
- There is a seamless transition from one phase of system development to the next. The OO approach uses the same "language" to talk about analysis, design, programming and database design. This reduces the level of complexity and redundancy and makes for *clearer, more robust system development.*
- The OO approach encourages good programming techniques. A class in an OO system carefully delineates between its interface (specifications of *what* an object of the class can do) and the implementation of that interface (*how* the class does what it does). The routines and attributes within a class are held together tightly: Changing one class has no impact on other classes.
- The OO approach promotes reusability. Objects are reusable because they are modelled directly out of a real-world problem domain. Each object stands by itself and does not concern itself with the rest of the system or how it is going to be used within a particular system. This means that classes are designed generically, with reuse as a constant background goal.
- A direct result of reusability is that OO adds inheritance, which is a powerful technique that allows classes to be built from each other. Therefore, only differences and enhancements between the classes need to be designed and coded. All the previous functionality remains and can be reused without change.
- The class hierarchy is dynamic. The function of a developer in an OO environment is to foster the growth of the class hierarchy by defining new, more specialized classes to perform the tasks that the application require.

9. <u>Object oriented system development</u>

OO systems development consists of several phases, the most generic of which are the following:

- OO analysis
- OO modelling
- OO design
- Implementation
- Testing

9.1 Object-oriented analysis

The first activity of OO analysis is to identify the classes that will compose the system. This is also one of the hardest activities in analysis. There is no such thing as the perfect class structure or the right set of objects. Nevertheless, the most classes fall in one of the following three categories and these can be used as guidelines identify classes in the given problem domain:

- **Persons**. What role does a person play in the system? For example, you might identify customers, employees, etc.
- **Places**. These are physical locations, buildings, stores, sites or offices about which the system should keep information.
- **Things** or **events**. These are events, i.e. points in time that must be recorded. For example, the system might need to remember when a customer makes an order; therefore, an order is an object.

The second activity of OO analysis is to identify the hierarchical relation between super classes and subclasses.

Next, we have to identify the attributes (properties) of objects, such as colour, cost, and manufacturer.

Fourthly, objects' behaviours must be identified. The main question to ask is: "What services must the class provide?

9.2 Object-oriented modelling

A model is an abstract representation of a system, constructed to understand the system prior to building or modifying it. A model can also be described as a simplified representation of reality. Good models are essential for communication among project teams. As the complexity of systems increase, so does the importance of good modelling techniques.

The unified modelling language (UML) is becoming the world's universal language for modelling object-oriented systems. UML is a language for specifying, constructing, visualising, and documenting a software system and its components. UML is a graphical language with sets of rules and semantics. A full discussion of UML is beyond the scope of this book.

9.3 Object-oriented design

During this phase, the classes that were identified during the analysis phase are designed. This can best be done by writing the class definitions as it is done under the **`interface`** section in a Delphi class unit. Remember to define each class in a separate unit.

9.4 Implementation

In this phase, the various classes' methods are implemented in detail in the `implementation` sections of the respective units.

9.5 Testing

If a developer waits until after development to test an application, he/she could be wasting thousands, if not millions, of rands and hours of time. Testing should be done integrally with development and should follow each one of the phases of development. Testing should also be done incrementally, i.e. each component, albeit how small, should be tested before the next component is added to a system.

Within our frame of reference, this means that you should press F9 the first time just after you have added, named and saved the first form to your project. Thereafter you should preferably press F9 time and again each time after you added a component or written a single line of code. You may use program stubs (empty frameworks for methods that have not yet been implemented) very effectively to run you program long before it is completed. NEVER develop a complete project and then try to run it for the first time!

10. Assignments and projects

Shorter assignments

10.1 Define a class for the following scenario. A learner wants to keep track of the marks he/she obtains for his/her subjects during a specific examination. Make provision for the subject name, total mark and marks obtained. Provide methods to update and return the values. Also define a method to return the percentage mark for the specific subject.

10.2 Define a class for a student class that can be used by a teacher. The properties should include the student number and marks for each one of two term tests. The class should include accessor methods to set and retrieve the values of the student number and the marks for the two tests. The class should also include a method called **`Average`** that returns the student's average marks for the two semester tests.

10.3 A learner wants to computerise his diary. Define a class for a single entry. Make provision for the date and details about the appointment. Define appropriate methods.

10.4 Define a class to represent information on an inventory item that includes an item number (integer), description (string), quantity (integer), and price (real). Include methods to set and retrieve the values of each one of these properties. Also include a method to return the current inventory value of an item (price times quantity).

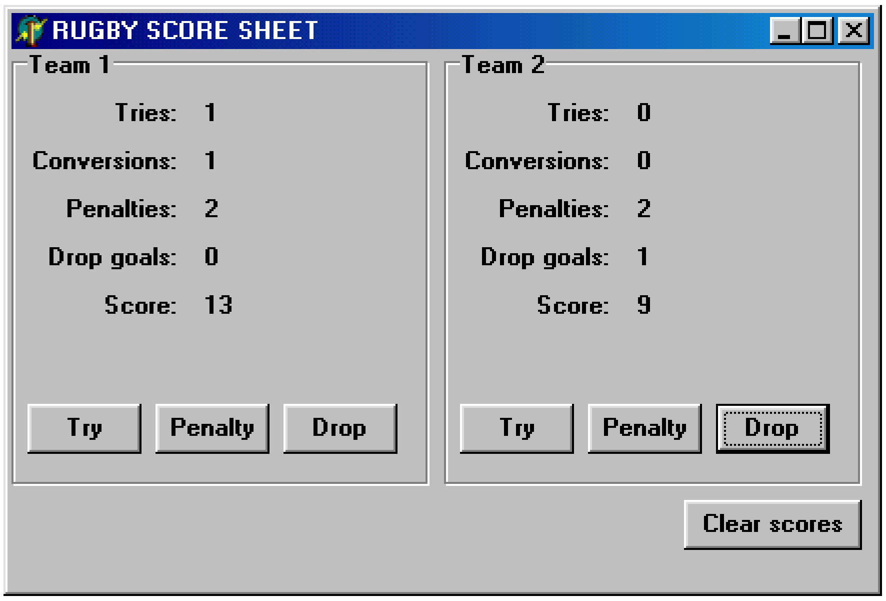
10.5 Define a class for the following scenario. Consider the bank account of a single client. Make provision for the account number, current balance and interest rate. Make provision for a constructor that will initialise the above attributes. Make provision for mutators and accessors to read and modify the attributes.

Programming Projects

10.6 Implement the class definition of 10.1 above in a full-fledged application. Create a separate object for each subject.

10.7 Consider the following screen print. This program can be used be a statistician watching a rugby match to record the points as they are scored.

10.7.1 Develop a class, **`TScore`**, which can be used to keep track of the score of one team. Include methods to update and return the number of tries, conversions, penalties and drop goals for every team. Also include a method that will calculate the current score as well as a method that will clear all scores. *(Try = 5, Conversion = 2, Penalty = 3, Drop goal = 3)*
10.7.2 Develop the user interface for this program.
10.7.3 Develop the functionality for this program. The event handlers of each of the buttons must refer to the appropriate method that was defined in 10.7.1 above.
10.7.4 You may, of course, extend the program. For example, you may capture and display the teams' names, the percentage of possession and territorial advantage, the moment in time when each point was scored, the names of the players who scored the points, etc. etc.



10.8 Develop a similar program than the one above, but with reference to a cricket match. The score sheet should resemble a typical cricket scoreboard, showing the runs of each batsman as well as the team total, etc. Make provision for 11 batsmen per team and two teams.

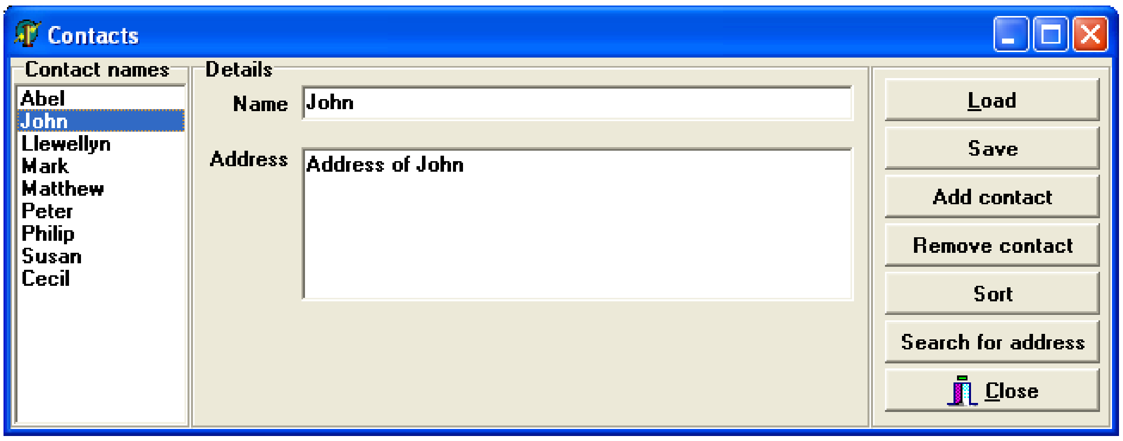
11. <u>Arrays of objects</u> (See Contacts in the code pack)

We will now discuss the implementation of an array of objects. That means that we will be able to declare more than one object of the same (or different) classes in an array in internal memory. Although in most cases it will be easier to develop a database program to keep record of multiple objects, there are cases where the much faster data access time in internal memory will be advantageous. For example, let us say there are 10 objects of a specific class but each property of each object is used about 100 times for some or other complicated process. Instead of fetching the same data time and again from external memory, which is very slow, we can read that data into the internal memory. The complicated process can then access the data from internal memory, which is much much faster.

11.1 Example scenario

In order to explain the principles, we will use the following scenario: You want to keep track of the contact details of your friends. You can make provision for their names, address, landline phone number, cell number, e-mail address, etc. To keep the examples as short as possible, however, we will only define members for the names and addresses of your friends.

The interface of our application might look like this. To the left we have a normal list box containing the names of the contacts. This will allow easy look-ups of a specific contact. In the middle we have a panel with the details of the selected contact and to the right we have a set of buttons to interact with the array of objects. Ensure that you <u>don't</u> set the **Sorted** property of the list box to true.



Consider the following simple class definition. Each object in the array will be a separate instance of **clsTContact**.

```
unit clsTContact;

interface
  type
    TContact = class (TObject)
      private
        FName       : String;
        FAddress    : String;
      published
        property Name    : String read FName write FName;
        property Address : String read FAddress write FAddress;
      end;
```

11.2 Using a TList to hold the array of objects

- Let us say the form was named **`frmContacts`** and saved as **`fContacts`**. The interface section should be adapted as follows:

```
interface

uses
  ..., clsTContact;

type
  TfrmContacts = class(TForm)
    ...
  private
    { Private declarations }
    lstContacts : TList;
  public
    { Public declarations }
  end;
```

  Note the following:

  a. The unit with the class definition of **`TContact`**, **`clsTContact`**, should be added to the **`uses`** clause.

  b. We will use the built-in **`TList`** class from the Delphi VCL to hold the array of objects. An instance of **`TList`**, we call it **`lstContacts`**, is declared in the private section of the form's class definition.

  In other words we will not declare our own array, but rather use an off-the-shelf class with built-in methods for adding, deleting, sorting, etc. objects in the array. The following is an excerpt from Delphi help:

  > TList, which stores an array of pointers, is often used to maintain lists of objects. TList introduces properties and methods to
  > - Add or delete the objects in the list.
  > - Rearrange the objects in the list.
  > - Locate and access objects in the list.
  > - Sort the objects in the list.

  Any other way of declaring an array of objects is like re-inventing the wheel. Why do something the difficult way if an existing class from the VCL that is specifically tailored to hold an array of objects, makes it so much easier?

  It is important to note that the **`TList`** object does not actually contain the objects themselves, but rather a list of pointers (memory addresses) to where the actual objects are stored.

- Since the **`TList`** object, **`lstContacts`**, is an object itself, it must be instantiated and released in the same way as for other objects. We do this in the form's **`Create`** and **`Destroy`** event handlers respectively.

```
procedure TfrmContacts.FormCreate(Sender: TObject);
begin
  lstContacts := TList.Create;
end; //FormCreate

procedure TfrmContacts.FormDestroy(Sender: TObject);
var i : Integer;
begin
  for i := 0 to lstContacts.Count-1 do
    TContact(lstContacts[i]).Free;
  lstContacts.Free;
end; //FormDestroy
```

Note the following:

a. In the **`Destroy`** event handler, each object in the list is released first and then the list itself is released. Later we will replace this code with a call to a procedure that will do the same.
b. As we have become used to now for all the built-in lists in the VCL, The first item in the list has index 0, the second item has index 1, etc. The last item will have an index that is 1 less than the number of items in the list.
c. To release the individual objects in the list, we had to step through the elements in the list in the same way that we would have done for a list box, starting from index 0 to index (number of items-1), releasing the objects one by one at a time.
d. Since the **`TList`** object holds only pointers, it can hold the pointers to objects of any class. In other words, when we work with the objects in the list, we have to tell the compiler to what class the object belongs. We do this by means of the typecast:
   `TContact(lstContacts[i])`
   In simple English this means: The i'th element of **`lstContacts`** points to a memory cell where a **`TContact`** is saved. It is very important that you understand this principle because we will use it regularly from now on.

11.3 Keeping track of the objects in the array

We need a way to ensure that each time a user clicks on a name in the list box, the relevant details from the corresponding object (address, etc.) will be displayed in the middle panel of the form. We can write all the code in the **`OnClick`** event handler of the list box, but it is actually better to write a separate procedure and then call the procedure from the **`OnClick`** event handler of the list box. By doing it this way, we would not have to repeat the code next time when we need to display the details of the objects.

```
type
  TfrmContacts = class(TForm)
    ...
  private
    { Private declarations }
    lstContacts : TList;
    procedure DisplayDetails;
    ...
  public
    { Public declarations }
  end;
```

```
procedure TfrmContacts.DisplayDetails;
var i : Integer;
begin
  i              := lstbxContactNames.ItemIndex;
  edtName.Text    := TContact(lstContacts[i]).Name;
  mmoAddress.Text := TContact(lstContacts[i]).Address;
end; //DisplayDetails
procedure TfrmContacts.lstbxContactNamesClick(Sender: TObject);
begin
  DisplayDetails;
end; //lstbxContactNamesClick
```

Note the following:

a. It is important that you add the header of the new procedure to the private section of the form's class declaration
b. The two lists, the list of objects and the list of names are parallel arrays. (This is, of course only true if the **Sorted** property of the list box is **false**.) To locate the relevant object in the list of objects, we can therefore refer to it by means of the corresponding index in the list of names.
c. In order to access the properties of the objects, we need to cast the list elements to **TContact** first.
d. The **Text** properties of the form controls, **edtName** and **mmoAddress**, is assigned values from the particular object in the array of objects.

11.4 Adding objects to the array

The **Add** button's **OnClick** event handler is used to add objects one by one to the array:

```
procedure TfrmContacts.btbtnAddContactClick(Sender: TObject);
var Contact : TContact;
begin
  //Create new object, enter a value for the key field and add to the list of objects
  Contact := TContact.Create;
  Contact.Name := InputBox('Contacts', 'Contact name: ', '');
  lstContacts.Add(Contact);
  //Add name of new object to list box with names and highlight it
  lstbxContactNames.Items.Add(Contact.Name);
  lstbxContactNames.ItemIndex := lstbxContactNames.Items.Count-1;
  DisplayDetails;
end;  //btbtnAddContactClick
```

Note the following:

a. A local variable (pointer) of class **TContact** is created that will be added to the list of contacts.
b. Only the key field of the contact, his name, is entered here. The address and other fields will be entered once the new object is created.
c. Similar to additions to a list box that you are used to, the built-in method, **Add**, is used to add a new item to the list and assign the pointer of **Contact** to it.
d. The local variable is not released from memory. Remember that these variables are only pointers. In the code above the pointer is assigned to the new list item (which is also a pointer). If we release the local variable, it will mean that we will also release the object to which the new list item refers. You will quickly get the dreaded "Access violation …" error.
e. The name of the new object should be added to the list box on the left hand-side of the form as well. Note that this is not part of the addition to the list of objects.
f. The newly added name is highlighted and its details displayed.

11.5 Updating the properties of an existing object

Once an object is in the array, we may update the other fields. We can use the **OnChange** event handler for each one the form controls to update the object properties while the user types:

```
procedure TfrmContacts.edtNameChange(Sender: TObject);
var i : Integer;
begin
  i := lstbxContactNames.ItemIndex;
  TContact(lstContacts[i]).Name := edtName.Text;
  lstbxContactNames.Items[i]    := edtName.Text;
end; //edtNameChange

procedure TfrmContacts.mmoAddressChange(Sender: TObject);
var i : Integer;
begin
  i := lstbxContactNames.ItemIndex;
  TContact(lstContacts[i]).Address := mmoAddress.Text;
end; //mmoAddressChange
```

Note the following:

a. Because the two lists are running in parallel, we can locate the relevant object in the list of objects by means of the index of the corresponding element in the list of names.
b. Note again that we need to cast the list items to **TContact** before we can access the properties.
c. In some cases it might be better to rather use the **OnExit** event handler for the update. The **OnExit** event handler is executed when a control loses focus; in other words when the user moves the cursor with the mouse or **Tab** to another control.

11.6 Removing an object from the array

Look at the following implementation of the **Remove** button's **OnClick** event handler:

```
procedure TfrmContacts.btbtnRemoveContactClick(Sender: TObject);
var i : Integer;
begin
  i := lstbxContactNames.ItemIndex;
  if i >= 0 then
  begin
    if Application.MessageBox
      (PChar('Sure you want to remove ' + lstbxContactNames.Items[i] + '?'),
             'Contacts', MB_ICONWARNING + MB_YESNO) = IDYES then
    begin
      lstContacts.Delete(i);
      lstbxContactNames.Items.Delete(i);
      lstbxContactNames.ItemIndex := 0;
      DisplayDetails;
    end;
  end
  else
    Application.MessageBox
      ('No contact selected.', 'Contacts', MB_ICONINFORMATION + MB_OK);
end; //btbtnRemoveContactClick
```

Note the following:

a. Once again, we locate the relevant object in the list of objects by means of the index of the corresponding element in the list of names.

b. Note that it is always good practice to ask the user to confirm that he wants to remove an object from the array.
c. The actual removal of the object is done by means of the **`Delete`** method of the **`TList`** class, which takes the index of the element to be removed as argument.
d. Of course, the name must be removed from the list box as well. After this, the first name in the list box is highlighted and its details displayed.

11.7 Searching for a specific object in the array

Although we do have a list of names to the left of the form, it might happen that the list becomes so long that it will be difficult to find a specific name. The user might even want to locate an object with a specific address, i.e. a field that is not in the list box of names. We can implement a search facility that will search through the array of objects, using our familiar linear search algorithm:

```
procedure TfrmContacts.btbtnSearchAddressClick(Sender: TObject);
var sSearch : String;
    i       : Integer;
    isFound : Boolean;
begin
  sSearch := InputBox ('Contacts', 'Address (or part thereof): ', '');
  i := 0;
  isFound := false;

  while (i <= lstContacts.Count-1) and (not isFound) do
  begin
    if Pos(UpperCase(sSearch), Uppercase(TContact(lstContacts[i]).Address)) > 0 then
      isFound := true
    else
      Inc(i);
  end; //while (i <= lstContacts.Count-1) and (not isFound) do

  if isFound then
  begin
    lstbxContactNames.ItemIndex := i;
    DisplayDetails;
  end
  else
    Application.MessageBox
         (PChar(sSearch + ' not found.'), 'Contacts', MB_ICONINFORMATION + MB_OK);
end; //btbtnSearchClick
```

<u>Note the following</u>:

a. Note the use of the **`UpperCase`** and **`Pos`** functions to enable a search on a part of the full address only and that is not case sensitive.

11.8 Sorting an array of objects

Sorting of objects in a **`TList`** object can be either short but tricky or longwinded but easy.

- Let's tackle the short one first. The entire sorting process is done in only two lines of code!

```
procedure TfrmContacts.Sort1;

  function CompareNames (Contact1, Contact2 : Pointer) : Integer;
  begin
    Result := CompareText(TContact(Contact1).Name, TContact(Contact2).Name);
  end; //CompareNames

begin //Sort1
  lstContacts.Sort(@CompareNames);
end; //Sort1

procedure TfrmContacts.btbtnSortClick(Sender: TObject);
begin
  if lstContacts.Count >= 2 then
  begin
    Sort1; //Or Sort2 - see later
    lstbxContactNames.Sorted := true;
    lstbxContactNames.Sorted := false;
  end;
end; //btbtnSortClick
```

Note the following:

a. The **Sort** button's **`OnClick`** event handler does two things, it calls a separate sort procedure that sorts the objects in the list of contacts and it sorts the list box with names. The **`Sorted`** property of the list box must be returned to false after sorting to maintain the principle of parallel arrays.

b. It is important that you add the header of the sort procedure to the private section of the form's class declaration.

c. The **`TList`** class contains a built-in method that does the sorting in one shot!

d. This **`TList.Sort`** method takes a function as parameter (indicated with @) which is maybe unfamiliar to you. This function must be written by you, the programmer, to indicate how the sorting must be done. In our example we want to sort the objects alphabetically according to the contact names.

e. The compare function must return an integer and the result must be either -1, 0 or 1. If the result is 1, two consecutive items in the array will be swopped, otherwise not.

f. The **`CompareText`** built-in function provides a short way for the following code:

```
if TContact(Contact1).Name < TContact(Contact2).Name then Result := -1;
if TContact(Contact1).Name > TContact(Contact2).Name then Result := 1;
if TContact(Contact1).Name = TContact(Contact2).Name then Result := 0;
```

g. Note once again that we need to cast the list elements to **`TContact`** before we can compare the **`Name`** properties.

h. In the code above the compare function is nested inside the sort procedure. This is, of course, not necessary, but provides a neat way to keep things that belong together, together.

- If you don’t understand the short way of sorting objects in a **`TList`**, you can always revert to the trusted bubble sort algorithm:

```
procedure TfrmContacts.Sort2;
var j, iEnd, iLast : Integer;
    isSwopped   : Boolean;
    tmpContact  : Pointer;
begin
  iEnd       := lstContacts.Count-2;
  isSwopped := true;
  while isSwopped do
  begin
    isSwopped := false;
    for j := 0 to iEnd do
    begin
      if TContact(lstContacts[j]).Name > TContact(lstContacts[j+1]).Name then
      begin
        tmpContact       := lstContacts[j];
        lstContacts[j]   := lstContacts[j+1];
        lstContacts[j+1] := tmpContact;
        isSwopped := true;
        iLast     := j;
      end;
      iEnd := iLast-1;
    end; //for j
  end; //while isSwopped
end; //Sort 2
```

Note the following:

a. Again we had to cast the list elements to **`TContact`** before we can compare the **`Name`** properties.

b. Note that the **`for`** loop runs from 0 and not 1 as you are maybe used to. This is because the first element in the array is indexed 0. Also, the second last element is **`lstContacts.Count`**`-2`.

c. Note that we don’t actually swop the entire objects in memory space. We could have done that, but it would be both cumbersome to program and inefficient during run-time. It is much better to just reconnect the pointers.

d. Since each element in **`lstContacts`** is a pointer (memory address), we need to declare our temporary swop variable as a pointer.

12. Persistence of an array of objects (See Contacts in the code pack)

We referred to this topic in section 7 already but there we explained how to save the private members of a single object to a text file or how to read from a text file. Doing the same with an array of objects is not much different. We only have to step through the list and handle each individual object separately.

12.1 Saving an object array to text file

The procedure to save the array of objects can look like this:

```
procedure TfrmContacts.SaveToTextFile;
var sPath : String;
    f     : TextFile;
    i     : Integer;
begin
  Screen.Cursor := crHourglass;
  sPath  := ExtractFiledir(Application.ExeName);
  AssignFile (f, sPath + '\Objects.txt');
  Rewrite (f);
  for i := 0 to lstContacts.Count-1 do
    Writeln (f,
             TContact(lstContacts[i]).Name, Chr(9),
             TContact(lstContacts[i]).Address);
  CloseFile (f);
  Screen.Cursor := crDefault;
end; //SaveToTextFile

procedure TfrmContacts.btbtnSaveClick(Sender: TObject);
begin
  //SaveToDB;
  SaveToTextFile;
end; //btbtnSaveClick
```

Note the following:

a. It is important that you add the header of the save procedure to the private section of the form's class declaration.

b. The private elements of a single object are saved to the text file as a single, tab-delimited, line.

c. Since access to secondary memory is time consuming, it is important to give the user visible feedback that a process is taking place and that he should be patient. In this example, the mouse pointer is changed to an hourglass during the saving process.

12.2 Read data from a text file into an object array

The procedure to read the records from the text file into an array of objects can look like this:

```
procedure TfrmContacts.FreeAll;
var i : Integer;
begin
  for i := 0 to lstContacts.Count-1 do
    TContact(lstContacts[i]).Free;
  lstContacts.Free;
  lstbxContactNames.Clear;
end; //FreeAll
```

```
procedure TfrmContacts.LoadFromTextFile;
var sPath, sLine : String;
    f       : TextFile;
    Contact : TContact;
begin
  Screen.Cursor := crHourglass;
  sPath  := ExtractFiledir(Application.ExeName);
  AssignFile (f, sPath + '\Objects.txt');
  {$i-} Reset(f); {$i+}
  if IOResult = 0 then
  begin
    //Prepare
    FreeAll; //Free all previous instances of clsContact as well as the TList
    lstContacts := TList.Create; //Recreate TList
    while not eof(f) do
    begin
      Readln(f, sLine); sLine := sLine + Chr(9);
      //For each record, create an instance of TContact and populate with data
      Contact := TContact.Create;
      Contact.Name := Trim(Copy(sLine, 1, Pos(Chr(9), sLine)-1));
      Delete(sLine, 1, Pos(Chr(9), sLine));
      Contact.Address  := Trim(Copy(sLine, 1, Pos(Chr(9), sLine)-1));
      //Add new instance of clsContact to TList
      lstContacts.Add(Contact);
      lstbxContactNames.Items.Add(Contact.Name);
    end; //while not tblContacts.Eof do
    lstbxContactNames.ItemIndex := 0;
    DisplayDetails;
  end; //if IOResult = 0
  Screen.Cursor := crDefault;
end; //LoadFromTextFile

procedure TfrmContacts.btbtnLoadClick(Sender: TObject);
begin
  LoadFromTextFile;
end; //btbtnLoad
```

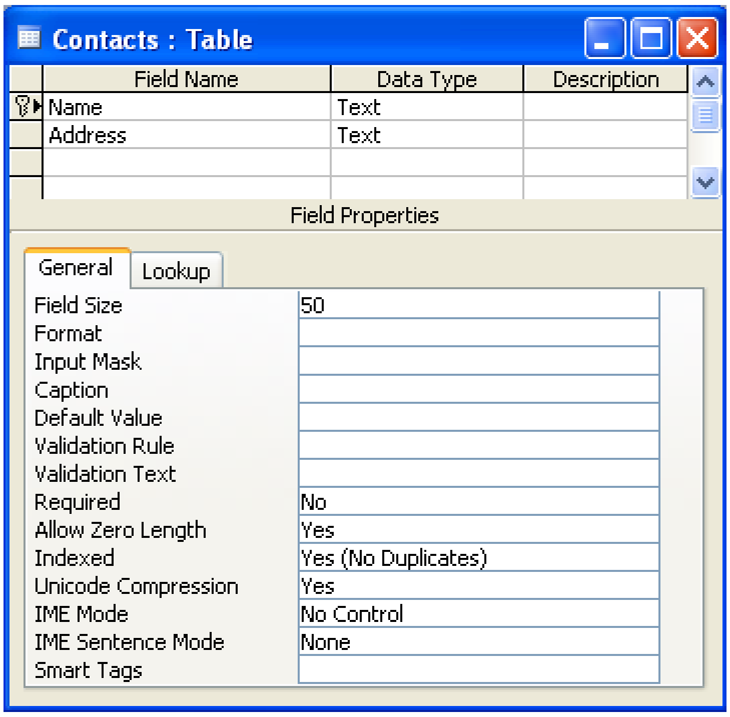
Note the following:

a. Before we can load the records into the objects, we must first ensure that all existing objects are released from memory. This has previously also been done in the **`FormDestroy`** event handler. Repeating the same code in several places in a program is bad programming practice since it creates a maintenance nightmare. Therefore, a procedure, **`FreeAll`**, has been created which should be called whenever the code is needed.
b. It is important that you add the header of the **`FreeAll`** procedure as well as that of the load procedure to the private section of the form's class declaration.
c. It is important to check that the data file exists before reading from it.
d. After loading the records into the array of objects, the first element in the list box is highlighted and its details displayed.

12.3 Saving an object array to a database

It is much more professional to save the data into a database rather than into a text file. In this example Microsoft Access was used as database environment.

- First of all, we need to design the database structure to correspond with that of the class. The table **Contacts** can be designed as follows:



Field Name	Data Type	Description
Name	Text	
Address	Text	

Field Size	50
Format	
Input Mask	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	Yes
Indexed	Yes (No Duplicates)
Unicode Compression	Yes
IME Mode	No Control
IME Sentence Mode	None
Smart Tags	

An ADO connection is placed on the form with the following set of properties:

ConnectionString	: 'Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Code Pack\Contacts\Contacts.mdb;Persist Security Info=False'
LoginPrompt	: false
Name	: cnContacts

<u>Note the following</u>:

a. Of course you don't have to remember the connection string. Use the browse facility offered by Delphi (click on the ··· next to the **ConnectionString** property) to complete the connection string for you.

b. If you intend to distribute your application to others who will not have the database in the same path as on your computer, you can use the following code in the **BeforeConnect** event handler of the connection object:

```
procedure TfrmContacts.cnContactsBeforeConnect(Sender: TObject);
var sPath : String;
begin
  sPath := ExtractFileDir(Application.ExeName);
  cnContacts.ConnectionString := 'Provider=Microsoft.Jet.OLEDB.4.0;Data Source='
                                 + sPath + '\Contacts.mdb;Persist Security Info=False';
end;
```

Next, you should place an ADO table component on the form with the following properties;

Connection	: cnContacts
Name	: tblContacts
TableName	: Contacts

- Now we are ready to write the procedure that saves the objects to the database:

```
procedure TfrmContacts.SaveToDB;
var i   : Integer;
    sql : String;
begin
  Screen.Cursor := crHourglass;
  //Delete all previous records in DB
  sql := 'DELETE * FROM Contacts';
  cnContacts.Execute(sql);
  //Refresh local copy of table in database
  tblContacts.Close; tblContacts.Open;
  //Step through TList and write all objects to the DB
  for i := 0 to lstContacts.Count-1 do
  begin
    tblContacts.Insert;
    tblContacts.FieldByName('Name').AsString
      := TContact(lstContacts[i]).Name;
    tblContacts.FieldByName('Address').AsString
      := TContact(lstContacts[i]).Address;
    tblContacts.Post;
  end; //for i := 0 to lstContacts.Count-1 do
  Screen.Cursor := crDefault;
end; //SaveToDB

procedure TfrmContacts.btbtnSaveClick(Sender: TObject);
begin
  SaveToDB; //Replaced SaveToTextFile
end; //btbtnSaveClick
```

Note the following:

a. It is again important to inform the user that a lengthy process is taking place by changing the mouse cursor temporarily to an hourglass.
b. A SQL instruction is used to delete all existing records in the database prior to saving the objects from internal memory. Of course this could have been done in another way.
c. After the records in the database were deleted, it is necessary to refresh the contents of the local copy of the table control by closing it and reopening it again.
d. Similar to saving the objects to a text file, we step through the array one object at a time and enter the private members into the table.
e. The **Save** button's **`OnClick`** event handler is changed to call the **`SaveToDB`** procedure instead of the **`SaveToTextFile`** procedure.

12.4 Read data from a database into an object array

The procedure to load the records from the database into an array of objects can look like this:

```
procedure TfrmContacts.LoadFromDB;
var Contact : TContact;
begin
  Screen.Cursor := crHourglass;

  FreeAll; //Free all previous instances of clsContact as well as the TList
  lstContacts := TList.Create;          //Recreate TList
  tblContacts.Close; tblContacts.Open; //Refresh local copy of table

  //Step through records in DB
  while not tblContacts.Eof do
  begin
    //For each record, create an instance of TContact and fill with data
    Contact := TContact.Create;
    Contact.Name    := tblContacts.FieldByname('Name').AsString;
    Contact.Address := tblContacts.FieldByname('Address').AsString;
    //Add new instance of clsContact to TList
    lstContacts.Add(Contact);
    lstbxContactNames.Items.Add(Contact.Name);
    //Next record in DB
    tblContacts.Next;
  end; //while not tblContacts.Eof do
  lstbxContactNames.ItemIndex := 0;
  DisplayDetails;

  Screen.Cursor := crDefault;
end; //LoadFromDB
```

Note the following:

a. Once again we have to release all previous objects and clear the list before populating it with the new data.
b. We step through the records in the table one by one, each time creating a new instance of **`TContact`**.
c. After updating the new objects properties from the database record, the object is added to the list of objects.
d. After loading the records into the array of objects, the first element in the list box is highlighted and its details displayed.

13. Inheritance and polymorphism in an array of objects (See Contacts2 in the code pack)

The principles of inheritance and polymorphism were discussed in section 6 above. In this section we will focus on the implementation of these principles when referring to an array of objects. We will continue with the same scenario as in the previous section, i.e. an array of objects containing contact details. This time however, we will distinguish between contacts who are friends and those who are formal or business contacts. For all contacts we will still have the private members of **`FName`** and **`FAddress`**, but for friends we will add the date of birth and for formal contacts we will add fields for the manager's name and the manager's telephone number.

13.1 Parent class and sub-class definitions of example scenario

- To refresh your mind, the class definition for contacts look like this:

```
type
  TContact = class (TObject)
    private
      FName    : String;
      FAddress : String;
    published
      property Name    : String read FName write FName;
      property Address : String read FAddress write FAddress;
    public
      constructor Create;
    end;
```

- The class definition for friends will look as follows:

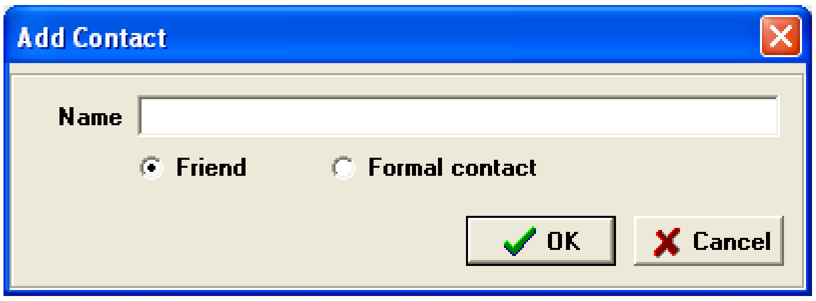
```
type
  TFriend = class (TContact)
    private
      FDateOfBirth : String;
      procedure SetDOB (Value : String);
    published
      property DOB : String read FDateOfBirth write SetDOB;
    public
      constructor Create;
    end;
```

- The class definition for formal contacts will look as follows:

```
type
  TFormal = class (TContact)
    private
      FManagerName : String;
      FManagerTel  : String;
    published
      property ManagerName : String read FManagerName write FManagerName;
      property ManagerTel  : String read FManagerTel write FManagerTel;
    public
      constructor Create;
    end;
```

13.2 Using a dialog box to add a new object

You will remember that we used an input box in the previous version of our program to add a new contact. Now we also have to indicate what the class of a new contact is and it will be better to develop our own dialog box to capture the details:



This dialog box is a normal form saved as **dAddContact** and named **dlgAddContact**. We have to add an `Execute` function to the public section of its class definition:

```
type
  TdlgAddContact = class(TForm)
    ...
  private
    { Private declarations }
  public
    { Public declarations }
    function Execute (var lstContacts : TList) : Boolean;
  end;
```

Note the following:

a. The **`Execute`** function takes a reference (**`var`**) parameter of the array of objects, **`lstContacts`**. This means that we created a separate name as formal parameter but it refers to the same variable as the actual parameter. Any changes made to **`lstContacts`** in this function will, therefore, also be available to the original **`lstContacts`**.

b. The function returns a **Boolean** value that indicates to the calling statement whether the user pressed **OK** or **Cancel** when he/she closed the form.

- The **`Execute`** function is implemented as follows:

```
implementation

function TdlgAddContact.Execute (var lstContacts : TList) : Boolean;
var Friend : TFriend;
    Formal : TFormal;
begin
  //Prepare form before showing it
  edtName.Clear;
  radbtnFriend.Checked := true;
  //Show form modally, i.e. the user must press either OK or Cancel to close
  ShowModal;
```

```
    //Processing of user entries after the form is closed
    if ModalResult = mrOk then //User pressed OK
    begin
      if radbtnFriend.Checked then //New contact is a friend
      begin
        //Create friend object and add to the list of contacts
        Friend := TFriend.Create;
        Friend.Name := edtName.Text;
        lstContacts.Add(Friend);
      end
      else  //New contact is formal
      begin
        //Create formal object and add to the list of conatcts
        Formal := TFormal.Create;
        Formal.Name := edtName.Text;
        lstContacts.Add(Formal);
      end;
      Result := true;
    end
    else  //User pressed Cancel
      Result := false;
end; //Execute
```

Note the following:

a. Note that there are basically three phases in the function: Preparation before the form is displayed, displaying the form modally, and processing after the user closed the form.

b. The **Kind** property of the **OK** button is set to **bkOK** and that of the **Cancel** button to **bkCancel**. This has the effect that the **ModalResult** property of the form gets either the value **mrOk** or **mrCancel** when it is closed.

c. Two local variables, **`Friend`** and **`Contact`**, were declared to refer to the two kinds of contacts respectively. Depending on which radio button the user selected, a new instance of the applicable kind of contact is created and then added to the array of objects.

d. It is very important to note that we save all contacts, irrespective of class, to the same array. We can do that since the **`TList`** object takes pointers and not the objects themselves. In fact, it is possible to mix objects of any class in the same array, for example pictures, buttons, pets, etc.

- The **`OnClick`** event handler of the **Add** button has to be adapted as follows:

```
procedure TfrmContacts.btbtnAddContactClick(Sender: TObject);
begin
  //Call dialog box where details will be inserted
  if dlgAddContact.Execute (lstContacts) then //if user pressed OK in the dialog box
  begin
    //Add name of new object to list box with names and highlight it
    lstbxContactNames.Items.Add(TContact(lstContacts[lstContacts.Count-1]).Name);
    lstbxContactNames.ItemIndex := lstContacts.Count-1;
    DisplayDetails;
  end;
end;  //btbtnAddContactClick
```

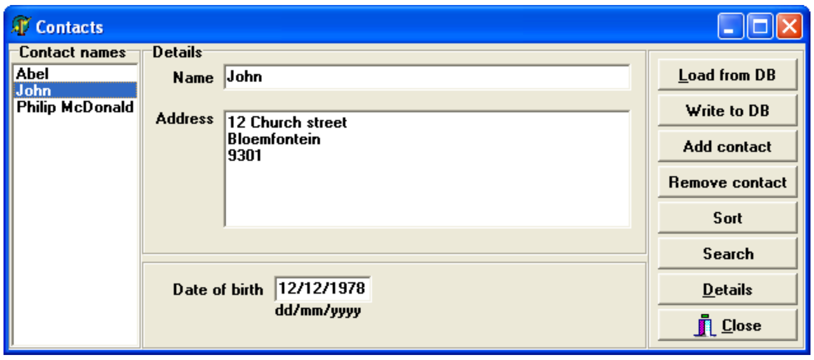
Note the following:

a. The new contact was added to the end of the array. Therefore, its index is **`lstContacts.Count-1`**.

b. We don't have to know whether the new contact is a friend or a formal contact to obtain his/her name. Since the **`Name`** property is shared in the parent class, we can simply cast the object to **`TContact`** to access the **`Name`** property.

13.3 Displaying the properties of sub-classes

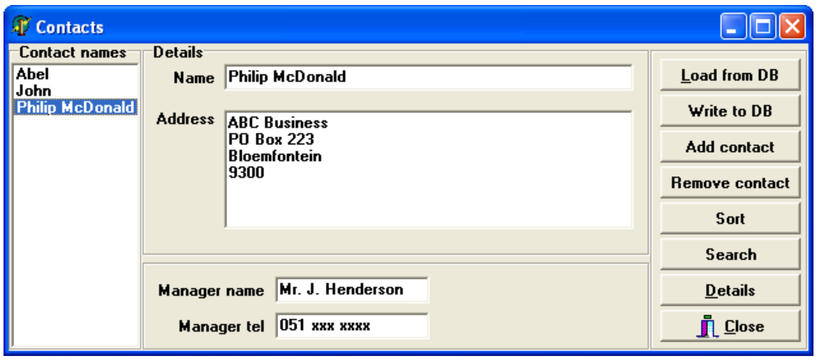
- The main form, **`frmContacts`**, has to be changed somewhat to make provision for the extra properties of the sub-classes. At the bottom of the middle panel we have added two group boxes, one for each of the sub-classes: **`grpbxFriends`** and **`grpbxFormal`**. Only one of the group boxes will be visible at a time, depending on the class of the selected object.



Contact names
Abel
John
Philip McDonald

Details
Name: John
Address: 12 Church street
Bloemfontein
9301

Date of birth: 12/12/1978
dd/mm/yyyy



Contact names
Abel
John
Philip McDonald

Details
Name: Philip McDonald
Address: ABC Business
PO Box 223
Bloemfontein
9300

Manager name: Mr. J. Henderson
Manager tel: 051 xxx xxxx

- You will remember that we had a procedure in the previous example to display the details of a specific contact when his/her name is selected in the list box with names on the left-hand side. This procedure must now be adapted somewhat to display the details of the appropriate sub-class as well:

```
procedure TfrmContacts.DisplayDetails;
var i : Integer;
begin
  i                := lstbxContactNames.ItemIndex;
  edtName.Text     := TContact(lstContacts[i]).Name;
  mmoAddress.Text := TContact(lstContacts[i]).Address;

  grpbxFriends.Visible := false;
  grpbxFormal.Visible  := false;
  if (TContact(lstContacts[i]) is TFriend) then //Contact is a friend
  begin
    grpbxFriends.Visible := true;
    edtDOB.Text          := TFriend(lstContacts[i]).DOB;
  end
  else //Contact is formal
  begin
    grpbxFormal.Visible := true;
    edtManagerName.Text := TFormal(lstContacts[i]).ManagerName;
    edtManagerTel.Text  := TFormal(lstContacts[i]).ManagerTel;
  end;
end; //DisplayDetails
```

Note the following:

a. The form controls **edtName** and **mmoAddress** are updated as previous.
b. Since all objects in the array are contacts, we can cast the selected object to **TContact** and then we make use of the class operator **is** to determine if this contact is a friend or formal contact.
c. Depending on the class of the contact, the relevant group box is made visible and the form controls updated.

13.4 Updating the properties of an object in a sub-class

In the previous example we used the **OnChange** event handlers of the respective form controls to update the object properties as the user types. We will have to do the same for the properties of the sub-classes as well. Since there is some data validation done for the DOB field, it cannot be done in the **OnChange** event handler. The program will respond with an error message after every digit that is typed. It will be better to do the update in the **OnExit** event handler of **edtDOB**. This event handler will be executed when **edtDOB** loses focus when the user clicks elsewhere with the mouse or presses the **Tab** key.

```
procedure TfrmContacts.edtDOBExit(Sender: TObject);
var i : Integer;
begin
  i := lstbxContactNames.ItemIndex;
  TFriend(lstContacts[i]).DOB := edtDOB.Text;
end; //edtDOBExit
```

13.5 Saving objects of a class hierarchy to text file

- Saving the objects to a text file is done by writing the two sub-classes to separate files:

```
procedure TfrmContacts.SaveToTextFile;
var sPath : String;
    fFriends, fFormals : TextFile;
    i      : Integer;
begin
  Screen.Cursor := crHourglass;
  sPath   := ExtractFiledir(Application.ExeName);
  AssignFile (fFriends, sPath + '\Friends.txt'); Rewrite (fFriends);
  AssignFile (fFormals, sPath + '\Formals.txt'); Rewrite (fFormals);
  for i := 0 to lstContacts.Count-1 do
  begin
    if TContact(lstContacts[i]) is TFriend then //Object is a friend
      Writeln (fFriends, TContact(lstContacts[i]).Name, Chr(9),
                         TContact(lstContacts[i]).Address, Chr(9),
                         TFriend(lstContacts[i]).DOB)
    else //Object is a formal contact
      Writeln (fFormals, TContact(lstContacts[i]).Name, Chr(9),
                         TContact(lstContacts[i]).Address, Chr(9),
                         TFormal(lstContacts[i]).ManagerName, Chr(9),
                         TFormal(lstContacts[i]).ManagerTel);
  end;
  CloseFile (fFriends); CloseFile (fFormals);
  Screen.Cursor := crDefault;
end; //SaveToTextFile
```

Note the following:

a. The only difference from the previous procedure where we had only one class, is the **if** statement where the **is** operator is used to determine the appropriate sub-class.

13.6 Reading objects of a class hierarchy from text file into an object array

When we read the objects back into the array, we have to deal with the two text files separately:

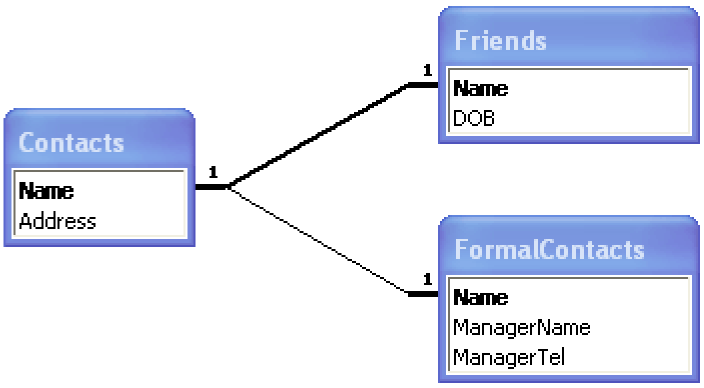
```
procedure TfrmContacts.LoadFromTextFile;
var sPath, sLine : String;
    f            : TextFile;
    Friend       : TFriend;
    Formal       : TFormal;
begin
  Screen.Cursor := crHourglass;
  FreeAll; //Free all previous instances of clsContact as well as the TList
  lstContacts := TList.Create; //Recreate TList

  //Read friends from text file
  sPath  := ExtractFiledir(Application.ExeName);
  AssignFile (f, sPath + '\Friends.txt'); {$i-} Reset(f); {$i+}
  if IOResult = 0 then
  begin
    while not eof(f) do
    begin
      Readln(f, sLine); sLine := sLine + Chr(9);
      //For each record, create an instance of TContact and populate with data
      Friend := TFriend.Create;
      Friend.Name := Trim(Copy(sLine, 1, Pos(Chr(9),sLine)-1));
      Delete(sLine, 1, Pos(Chr(9),sLine));
      Friend.Address := Trim(Copy(sLine, 1, Pos(Chr(9),sLine)-1));
      Delete(sLine, 1, Pos(Chr(9),sLine));
      Friend.DOB := Trim(Copy(sLine, 1, Pos(Chr(9),sLine)-1));
      //Add new instance of clsTFriend to TList
      lstContacts.Add(Friend);
      lstbxContactNames.Items.Add(Friend.Name);
    end; //while not tblContacts.Eof do
    CloseFile(f);
  end; //if IOResult = 0

  //Read formal contacts from text file
  sPath  := ExtractFiledir(Application.ExeName);
  AssignFile (f, sPath + '\Formals.txt'); {$i-} Reset(f); {$i+}
  if IOResult = 0 then
  begin
    while not eof(f) do
    begin
      Readln(f, sLine); sLine := sLine + Chr(9);
      //For each record, create an instance of TContact and populate with data
      Formal := TFormal.Create;
      Formal.Name := Trim(Copy(sLine, 1, Pos(Chr(9),sLine)-1));
      Delete(sLine, 1, Pos(Chr(9),sLine));
      Formal.Address := Trim(Copy(sLine, 1, Pos(Chr(9),sLine)-1));
      Delete(sLine, 1, Pos(Chr(9),sLine));
      Formal.ManagerName := Trim(Copy(sLine, 1, Pos(Chr(9),sLine)-1));
      Delete(sLine, 1, Pos(Chr(9),sLine));
      Formal.ManagerTel := Trim(Copy(sLine, 1, Pos(Chr(9),sLine)-1));
      //Add new instance of clsTFormal to TList
      lstContacts.Add(Formal);
      lstbxContactNames.Items.Add(Formal.Name);
    end; //while not tblContacts.Eof do
    CloseFile(f);
  end; //if IOResult = 0

  //Highlight first name in listbox and display details of the corresponding object
  lstbxContactNames.ItemIndex := 0;
  DisplayDetails;
  Screen.Cursor := crDefault;
end;  //LoadFromTextFile
```

Note the following:

a. Since we first read all the friends and then all the formal contacts, the order of elements in the lists will not necessarily be the same as it was originally.

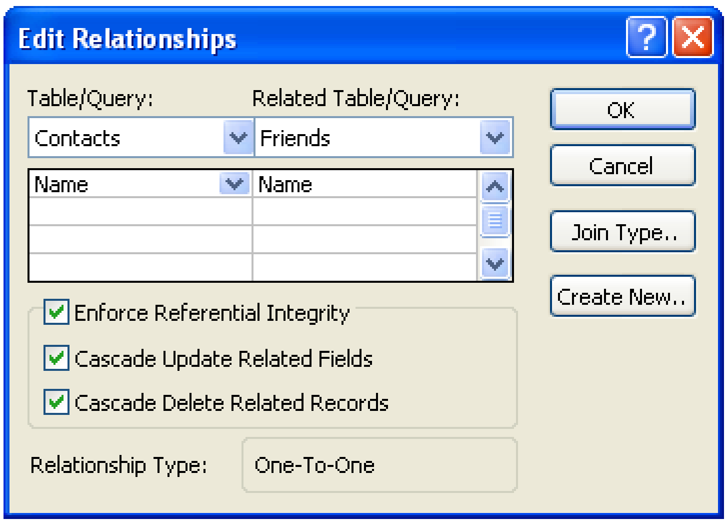
13.7 Saving objects of a class hierarchy to a database

- Saving the objects to a database entails a change of the database design to correspond to the class definitions. A possible design could look like this:



Note the following:

a. The entity relationship diagram (ERD) of Microsoft Access should be used to create relationships between the parent class and sub-classes. It is important to ensure that both cascade updates and cascade deletes are checked.



Table/Query: Contacts

Related Table/Query: Friends

Table/Query	Related Table/Query
Name	Name

[✓] Enforce Referential Integrity

[✓] Cascade Update Related Fields

[✓] Cascade Delete Related Records

Relationship Type: One-To-One

- The **SaveToDB** procedure can be implemented like this:

```
procedure TfrmContacts.SaveToDB;
var i   : Integer;
    sql : String;
begin
  Screen.Cursor := crHourglass;
  //Delete all previous records in DB
  sql := 'DELETE * FROM Contacts'; //Implicitly deleting all friends and formal
                                   //  contacts due to cascading deletes
  cnContacts.Execute(sql);
  //Refresh local copy of tables in database
  tblContacts.Close; tblContacts.Open; tblFriends.Close; tblFriends.Open;
  tblFormal.Close; tblFormal.Open;
  //Step through TList and write all objects to the DB
  for i := 0 to lstContacts.Count-1 do
  begin
    tblContacts.Insert;
    tblContacts.FieldByName('Name').AsString    := TContact(lstContacts[i]).Name;
    tblContacts.FieldByName('Address').AsString := TContact(lstContacts[i]).Address;
    tblContacts.Post;
    if TContact(lstContacts[i]) is TFriend then //Contact is a friend
    begin
      tblFriends.Insert;
      tblFriends.FieldByName('Name').AsString := TFriend(lstContacts[i]).Name;
      tblFriends.FieldByName('DOB').AsString  := TFriend(lstContacts[i]).DOB;
      tblFriends.Post;
    end
    else //Contact is formal
    begin
      tblFormal.Insert;
      tblFormal.FieldByName('Name').AsString        := TFormal(lstContacts[i]).Name;
      tblFormal.FieldByName('ManagerName').AsString := TFormal(lstContacts[i]).ManagerName;
      tblFormal.FieldByName('ManagerTel').AsString  := TFormal(lstContacts[i]).ManagerTel;
      tblFormal.Post;
    end;
  end;  //for i := 0 to lstContacts.Count-1 do
  Screen.Cursor := crDefault;
end; //SaveToDB
```

Note the following:

a. The fact that cascade deletes are checked makes it unnecessary to delete the records from the three tables separately. When the records in the base table, **Contacts**, are deleted, all records in the two client-tables, **Friends** and **FormalContacts**, are deleted implicitly.

b. Unlike the in the **SaveToTextFile** procedure, we saved the data of the parent class separately. This is in accordance with good database practices to reduce data redundancy and facilitate data integrity. The database design should be in at least third normal form.

13.8 Reading objects of a class hierarchy from a database into an object array

The **LoadFromDB** procedure is slightly more involved:

```
procedure TfrmContacts.LoadFromDB;
var Friend : TFriend;
    Formal : TFormal;
begin
  Screen.Cursor := crHourglass;
  //Prepare
  FreeAll; //Free all previous instances of clsTContact as well as the TList
  lstContacts := TList.Create; //Recreate TList
  tblContacts.Close; tblContacts.Open;
```

```
    //Step through friend records in DB
    tblFriends.Close; tblFriends.Open;
    while not tblFriends.Eof do
    begin
      //Get record in contact table for this friend
      tblContacts.Locate('Name', tblFriends.FieldByName('Name').AsString, []);
      //For each record, create an instance of TContact and fill with data
      Friend         := TFriend.Create;
      Friend.Name    := tblContacts.FieldByname('Name').AsString;
      Friend.Address := tblContacts.FieldByname('Address').AsString;
      Friend.DOB     := tblFriends.FieldByname('DOB').AsString;
      //Add new instance of clsTContact to TList
      lstContacts.Add(Friend);
      //Next record in DB
      tblFriends.Next;
    end; //while not tblContacts.Eof do

    //Step through formal records in DB
    tblFormal.Close; tblFormal.Open;
    while not tblFormal.Eof do
    begin
      //Get record in contact table for this friend
      tblContacts.Locate('Name', tblFormal.FieldByName('Name').AsString, []);
      //For each record, create an instance of Tformal and fill with data
      Formal             := TFormal.Create;
      Formal.Name        := tblContacts.FieldByname('Name').AsString;
      Formal.Address     := tblContacts.FieldByname('Address').AsString;
      Formal.ManagerName := tblFormal.FieldByname('ManagerName').AsString;
      Formal.ManagerTel  := tblFormal.FieldByname('ManagerTel').AsString;
      //Add new instance of clsTContact to TList
      lstContacts.Add(Formal);
      //Next record in DB
      tblFormal.Next;
    end; //while not tblContacts.Eof do
   //Highlight first name in listbox and display details of the corresponding object
    lstbxContactNames.ItemIndex := 0;
    DisplayDetails;

    Screen.Cursor := crDefault;
  end; //LoadFromDB
```

Note the following:

a. As was the case for **`LoadFromTextFile`** we handle the tables for friends and formal contacts separately.
b. Note the usage of the **`Locate`** function above. Read in Delphi help about its usage if you are unfamiliar with it. Basically, it does a search through a database table to locate a specific record based on the field and value given as parameters.

13.9 Polymorphism (again)

The principle of polymorphism was discussed in section 6 above. The word polymorphism means "many forms". In OOP terms it refers to the fact that there may be more than one form (version) of a given method across several sub-classes of the same parent class. The linker will decide during run-time which method to call.

In our example of contacts, let us say we want to display, besides the name and address, also the date of births of the friend contacts and the manager details of the formal contacts.

- We need to define a virtual (dummy) method in the parent class, **`TContact`**, which will be overridden by methods of with the same name in the sub-classes:

```
interface

  type
    TContact = class (TObject)
      private
        ...
      published
        ...
      public
        ...
        procedure PrintDetails; virtual;
      end;

implementation

procedure TContact.PrintDetails;
begin
  //Only virtual method
end; //PrintDetails
```

- For friends the method might be implemented as follows:

```
interface

  type
    TFriend = class (TContact)
      private
        ...
      published
        ...
      public
        ...
        procedure PrintDetails; override;
      end;

implementation

procedure TFriend.PrintDetails;
var s : String;
begin
  s :=   'Name    : ' + self.Name + ENTER
       + 'Address : ' + self.Address + ENTER
       + 'DOB     : ' + self.DOB;
  frmDetails.lblDetails.Caption := s;

  frmDetails.ShowModal;
end; //PrintDetails
```

- For formal contacts the method might be implemented as follows:

```
interface

  type
    TFormal = class (TContact)
      private
        ...
      published
        ...
      public
        ...
        procedure PrintDetails; override;
      end;
```

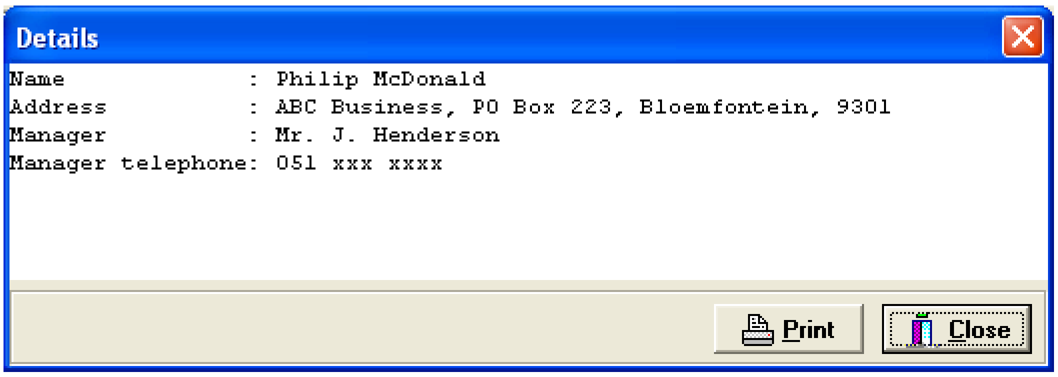
```
implementation

procedure TFormal.PrintDetails;
var s : String;
begin
  s :=    'Name              : ' + self.Name + ENTER
        + 'Address           : ' + self.Address + ENTER
        + 'Manager           : ' + self.ManagerName + ENTER
        + 'Manager telephone: ' + self.ManagerTel;
  frmDetails.lblDetails.Caption := s;
  frmDetails.ShowModal;
end; //PrintDetails
```

Note the following:

a. All three classes, **`TContact`**, **`TFriend`** and **`TFormal`**, have a method **`DisplayDetails`**. In the parent class it is marked with the keyword **`virtual`**`;` and its implementation is basically an empty procedure because it will in fact never be called. In the sub-classes, the declaration of the method is marked with the keyword **`override`**`;` which indicates that the virtual method in the parent class is overridden and replaced with the method in the sub-class.

b. The two **`DisplayDetails`** methods in the two sub-classes are very similar and differ only with regard to the sub-class details to be displayed.

c. You would have noticed the call to **`frmDetails.ShowModal`** in the implementations above. We developed a separate form to display the details:



```
Name             : Philip McDonald
Address          : ABC Business, PO Box 223, Bloemfontein, 9301
Manager          : Mr. J. Henderson
Manager telephone: 051 xxx xxxx
```

- And now for the nice part that shows what polymorphism is all about. The **`OnClick`** event handler of the **Details** button on the main form can be implemented as follows:

```
procedure TfrmContacts.btbtnDetailsClick(Sender: TObject);
var i : Integer;
begin
  i := lstbxContactNames.ItemIndex;
  TContact(lstContacts[i]).PrintDetails;
end; //btbtnDetailsClick
```

Note the following:

a. As was done previously, we use the fact that we are working with two parallel arrays to find the index of a specific object in the array of objects.

b. Remember that **`lstContacts`** contains both **`TFriend`** and **`TFormal`** objects. At the development stage there is no way for the programmer to tell what class **`lstContacts[i]`** will belong to. Therefore, the (virtual) method **`PrintDetails`** of the parent class is called. During run-time, the linker can determine the exact sub-class of **`lstContacts[i]`** at that moment in time and override the call with the method in the specific sub-class. Cool hey??