

Back To Basics

Object Orientation

by Peter Parker (in conjunction with his presentation at POSK on January 15th)

Summary

This article is a version of the presentation to the Borland User Group on the same subject. It covers a brief review of object orientation - what it is, how do you implement it and what use it is anyway? It does this by looking a little at the theory with a worked example based upon the Person/Employee object developed through this article. Finally, it looks at the use of some standard objects, and points to some rules described by Marco Cantu.

Overview

Object orientation is not new. It may be new to programming, but the reality began a long time ago. The best example was when Ford set up the factory to produce the Model T. The concept has expanded much since then, but let us review the concept of car manufacture today.

When Ford, Mercedes, BMW, Jaguar want to put a component in a car, they do not worry about the detail. I want a gearbox - is the demand. The parameters of the gearbox are this. I am not interested in what the cogs are made from. I have no interest in how the engine's input is connected to the drive shaft output. I just want it to do the job that is required. Similarly, a carburettor must perform to my specifications. I do not wish to know how it works. Here we have a mechanical engineering parallel to object design.

And in software, it should be no different.

But let us stop a minute and consider the history of software engineering. If you have been in the industry as long as I have, you may remember the original spaghetti coding. My start was in COBOL, with lots of GOTOs and very little PERFORMs. Needless to say, it was horrendous. And had lots of problems with debugging and performance.

But then we moved to structured programming. The concept here was that, if there were tasks that could be done over and over again, then you must set them up as functions and subroutines. The idea was to have a hierarchical structure in a programme, with the programme logic consisting of something simple that was broken down into other simple routines and so on until the final logic was also simple. Structured programming delivered a lot. It made programming more reliable and started on the route to reusable code. You could deliver dynamic linked libraries of routines that were very useful, and could be used by others.

The problem with structured programming, when you moved to complex systems was that they inevitably led to functions and subroutines that had larger and larger parameter lists. This meant that, for complex systems, structured programming became unwieldy.

The demand was to look for a new paradigm. And one of the latest answers is to take the mechanical engineering approach and produce object oriented programming. So let's have a look at what this means.

Some Terms (unexplained)

Object oriented programming came with its own vocabulary. Terms such as

- Objects - the things with which we work
- Classes - actually only another name for an object definition
- Methods - things that objects do - subroutines really
- Encapsulation - well let's look at that later
- Inheritance - we'll also review this by example
- Polymorphism - we'll look at what this means by example
- Hierarchy - Inheritance by another name

And so on.

But our objective is not to go through all the terms. It is more important to look at what object orientation is and what it does for us. So let's get in and try and understand it.

An Example Object

We will build part of this object, so let us put together an object that makes software sense. We will start with the object Person - actually TPerson (in Delphi the standard is to prefix all classes with T - so TAnything is the standard).

We all know what a person is. But we will probably have different ideas of the data that will sit behind this object. So we may each define TPerson differently. Let us consider a Person. An actual Person will be different person to person. So a person may or may not be a sports person. They may be golfers or footballers or have none of these attributes.

A person may be an employee. If they are an employee, they may be either an hourly employee, or a commissioned employee. Each of these will have different attributes. How we would set this up in Delphi?

First Some Enumerated Types

In order for our object to work, we need to introduce some terms and types that we wish to use. I have created a unit to satisfy this aspect, which I have called uTypes. Here is the total source code:

```
unit uTypes;
interface
type stSexType = (Male, Female, NotKnown);
    sptSport = (Football, Golf, Rugby, Swimming);
implementation
end.
```

We have defined two types - stSexType and sptSport. These definitions allow us to use the terms Male, Female, NotKnown, Football, Golf, Rugby and Swimming and know that they will be replaced by numbers that we do not have to worry about.

An Object - The Basic Definition

A person is an object. All objects ultimately are derived from TObject. That is to say they inherit their behaviour from the class TObject.

Three definitions here:

- Inherit (we will see more of this later when we inherit from our own object)
- Class - the definition of the object
- Hierarchy - The current hierarchy is Object then Person.

So the simplest definition of TPerson in Delphi would be:

```
type TPerson = class(TObject)
end;
```

This defines the class of TPerson as being derived (inheriting) from TObject. If this is all we did, there would be no difference between TPerson and TObject. But we intend to add more.

Object Attributes

Objects have attributes. We can see this very well by example. The Person object is likely to have a name, an address, a date of birth, a sex and so on. These are attributes that are likely to be different person to person. Let's expand our definition of the object to see how Delphi handles attributes. You will recognise this from the TForm attributes set up.

```
type TPerson = class(TObject)
  fName, fAddress: string;
  fDateOfBirth: Tdate;
  fSex: stSexType;
end;
```

So our object now has some attributes that distinguishes this from the base object.

I prefer to make most of my attributes private, so a better definition would be:

```
type TPerson = class(TObject)
  private
    fName, fAddress: string;
    fDateOfBirth: Tdate;
    fSex: stSexType;
end;
```

Object Methods

Methods are things that objects do. We will have some simple examples here. At the moment, we will define methods such as SetName that will set the name of this object. It is possible, in fact highly probable, that we will have private methods. We will later move these setters and getters to the private area. But, for the moment, let us add the methods to the definition:

```
type TPerson = class(TObject)
  private
    fName, fAddress: string;
    fDateOfBirth: Tdate;
    fSex: stSexType;
  public
    procedure SetName(aName: string);
    procedure SetAddress(aAddress:
      string);
end;
```

Now we choose a feature in Delphi that does code completion. If we press CTRL/Shift/C all together, Delphi will set up the start point for the coding of our class methods (at least, if our cursor is on the procedure definition). Note that in our case SetName is now defined as:

```
procedure TPerson.SetName(aName: string);
begin
end;
```

We can now fill in the code which may only have:

```
fName := aName;
```

Why a Set Method?

In a real life object, we will probably want to do more than this, such as checking that the name is a sensible string and so on. But the concept is simply that of modifying the name attribute. It is much like the working model of a car, where we just change the gear to second. As an object deliverer, we do all the work to ensure that, from the outside, it is a simple method.

But the set method will enable us to ensure that the attribute holds a sensible value - that it only holds valid information.

Much more important, and this is where object orientation starts to come into its own, is that the setting of attributes is done in only one place - inside the object itself. The object itself does the validation of the attribute, and ensures that only valid data is let through.

This leads on to the ability to change the way the object works. If we find that we are not doing correct validation or that the object can do more, there is only one place to make that change - in the object itself. The object is self checking and self consistent. The outside world can only manipulate the object.

Accessing Object Attributes

As the attributes are private, we need to be able to find out details of the attributes. Hence we need functions to return the attribute values. These are set up similarly to the methods. So we may have functions such as:

```
function GetName: string;
function: GetBirthDate: TDate;
function: GetAge: integer;
```

As you may guess, some of these will be simply defined as:

```
function TPerson.GetName: string;
begin
  result := fName;
end;
```

But we can see that GetAge is a calculation.

Properties

Getters and setters are fine, but the use of properties would be better. Let us make the Getters and Setters private, and now look at how we would use properties to access these. Let's take fName as a simple starting point.

```
property Name: string read fName write fName;
```

Here we have defined the property of TPerson: **Name**. We have said that we can read and write direct to the attribute itself.

Much more satisfactory is what we will do with BirthDate

```
property BirthDate: TDate read GetBirthDate
write SetBirthDate;
```

if we make the Getter and Setter private and add a public property. We may now use this property and hide the Getter and Setter action. This allows us to use the property as though it were an attribute, but know that there is real work going on behind the scenes.

Properties - Use - Examples

So if we have a property - say PostCode defined as

```
property PostCode: string read GetPostCode
write SetPostCode;
```

Then the use of the property might be as

```
LocalPostCode := CurrentPerson.PostCode;
```

and

```
CurrentPerson.PostCode := UpdatedPostCode
```

not

```
CurrentPerson.SetPostCode(UpdatedPostCode);
```

The properties approach is much clearer and hides the fact that real validation is being handled by the Getting and Setting routines behind the scenes. The resultant code is much more readable and friendly for debugging purposes.

Inheritance

So we have now set up our basic person as an object or class. We may wish to have other classes of person that behave in a different way. So we could have a TSportster inherited from TPerson.

This object, if only defined as inheriting from TPerson would be no different from the Person object. However, a Sportster could have a further attribute, say Sport that defined this person as a golfer or footballer and so on.

But let us look at a more concrete example. There is also an Employee. Let us consider the employee as an object.

```
TEmployee = class(TPerson)
private
  fEmpNo: integer;
  fDateStarted: TDate;
  fBaseSalary: real;
  procedure SetEmpNo(aEmpNo: integer);
  procedure SetDateStarted(aDate: TDate);
  procedure SetBaseSalary(aSalary: real);
  function GetEmpNo: integer;
  function GetDateStarted: TDate;
  function GetBaseSalary: real;
```

```
function GetSalary: real; virtual;
abstract;

public
  property EmployeeNumber: integer read
    GetEmpNo write SetEmpNo;
  property DateStarted: TDate read
    GetDateStarted write SetDateStarted;
  property BaseSalary: real read
    GetBaseSalary write SetBaseSalary;
  property Salary: real read GetSalary;
end;
```

Polymorphism

The Employee allows us to look not only at inheritance but at polymorphism. Let us first take the inheritance route.

All employees will have a date when they started employment, an employee number and a base salary. So these attributes will be set up as attributes of the TEmployee object. Similarly there will be Set methods and Get functions for setting and retrieving these attributes. However, we ought also to provide a method called GetSalary which returns a real value, but that we do not know how to calculate at this time. We know we will need this function. So we define it as virtual. We are saying that the ultimate object will define this and that the ultimate object will return a value.

A rule worth considering here is the rule that there should be one unit per object. If we do this, the compiler will give us a problem with a definition where we do not have an implementation. Although we have stated that GetSalary is virtual so we will replace it later, we may not wish to define it now. If we only want to define it in child objects, then we have to inform the compiler that the method definition is more than virtual, it is abstract. By doing this, the compiler still complains (warnings unless you set the switches) but produces the output anyway.

In the example we are using, we are not taking advantage of the abstract mechanism (the code supporting this article takes this a little further). If we are the ultimate object, we will need to define this and so, in our first implementation, we will return a value of 0. But this is not necessary as we will see.

So we have inherited an employee object from a person object. Now we need to consider both an hourly and a commissioned employee. These objects will both inherit from Employee. However, the salary calculation will be different. We will have two different calculations for salary. Both of these calculations will be visible to the Employee object and will be called correctly by that object. This is polymorphism in action. Let us see the code to do this.

We now inherit from our Employee object:

```
THourlyEmployee = class(TEmployee)
private
  fHoursWorked, fHourlyRate: real;
  procedure SetHoursWorked(aHours: real);
  procedure SetHourlyRate(aRate: real);
  function GetHoursWorked: real;
  function GetHourlyRate: real;
  function GetSalary: real; override;
public
  property HoursWorked: real read
    GetHoursWorked write SetHoursWorked;
  property HourlyRate: real read
    GetHourlyRate write SetHourlyRate;
end;
```

As we can see, the hourly employee has hours worked and rate. We have the standard set and get methods in place, but we also have defined GetSalary which overrides our parental version of this definition.

Also note that we do not redefine the property Salary. We do not need to reconsider this.

For the hourly employee the GetSalary function is:

```
function THourlyEmployee.GetSalary: real;
begin
  result := fBaseSalary + (fHoursWorked *
    fHourlyRate);
end;
```

Let us consider the similar code for the commissioned employee:

First the definition:

```
TCommissionedEmployee = class(TEmployee)
private
  fSalesMade, fCommissionRate: real;
  procedure SetSalesMade(aSalesMade: real);
  procedure SetCommissionRate(aRate: real);
  function GetSalesMade: real;
  function GetCommissionRate: real;
  function GetSalary: real; override;
public
  property SalesMade: real read
    GetSalesMade write SetSalesMade;
  property CommissionRate: real read
    GetCommissionRate write SetCommissionRate;
end;
```

and the code for GetSalary:

```
function TCommissionedEmployee.GetSalary: real;
begin
  result := fBaseSalary + (fCommissionRate *
    fSalesMade);
end;
```

So we have our definitions of GetSalary, which gives us our polymorphism. We will see this in action later. I would like to develop the ideas before we actually see the concept working.

Use of Objects

But before we go on, let us review where we are at:

We have derived or inherited an Employee from a Person object. This Employee has attributes that are peculiar to employees, but are not necessarily available to all Persons. However, an Employee has a property Salary that is dependent upon something that is not part of a base Employee object.

The hourly employee inherits from Employee. The hierarchy is:

```
Object
  Person
    Employee
      Hourly Employee
```

But at this point we now know how to calculate a salary.

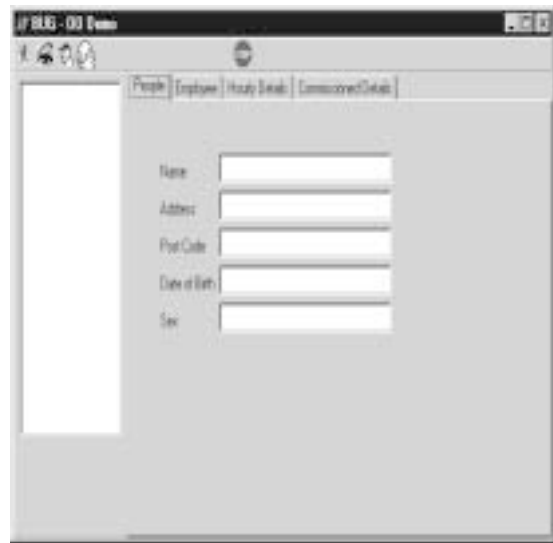
Similarly a commissioned employee has the hierarchy:

```
Object
  Person
    Employee
      Commissioned Employee
```

And we have a different method of calculating a salary.

The Interface - Base Form

Let us now consider the user interface. For the purposes of this article I have considered the following form as a demonstration of one way of implementing objects:



You will note a number of items, first the buttons. These are on a panel and offer the options of “Add Person”, “Add Employee”, “Add Hourly Employee” and “Add Commissioned Employee” - and of course “Stop” the program.

Below the panel is a list box and a tabbed set of pages that have information for each of the parts of the object that we are interested in.

Local Definitions

Now we can start defining some local information. I am not suggesting that this is particularly the best way of doing things, but it highlights a number of object aspects that show inheritance and polymorphism in action, rather than ultimately correct techniques.

So our application (TForm1) will contain the following definitions:

```
private
  aPeople: array of TPerson;
  intArraySize: longint;
```

What we have here is a dynamic array of the base class in which we are going to work, and a counter for the size of the array.

The Procedure DisplayPerson

As you will have noted from the form, there are fields that are set up to display the information about a person. So here is the procedure in TForm1 that does that display.

```
procedure TfrmMain.DisplayPerson(intThisPerson:
  Integer);
begin
  txtName.Text := aPeople[intThisPerson].Name;
  txtAddress.Text :=
    aPeople[intThisPerson].Address;
  txtPostCode.Text :=
    aPeople[intThisPerson].PostCode;
  txtDateOfBirth.Text :=
    DateToStr(aPeople[intThisPerson].BirthDate);
end;
```

OK so I lied, the form is called frmMain. But you get the idea. Given an offset, this procedure will display the items from the dynamic array. We can see the simple calling of the object details now held in the array.

“But how do we get this information into the object?” I can hear you shouting.

The Add Person Button

Well, let’s look at the Add Person button. Before we do we must initialise some variables. These could be set initialised, but I am of the old school that says take complete control, and then you know what is happening.

The Form Create action is very simple:

```
intArraySize := 1;
pgPerson.ActivePageIndex := 0;
```

All it does is to say the dynamic array size is one, and that the page control active page is page zero.

So moving quickly to the Add Person button, here is the code:

```
var
    intThisEmployee: longint;
begin
    SetLength(aPeople, intArraySize);
    intThisEmployee := intArraySize - 1;
    inc(intArraySize);
    aPeople[intThisEmployee] := TPerson.Create;
    DisplayPerson(intThisEmployee);
    lbxPerson.Items.Add(aPeople[intThisEmployee].Name);
```

First we have to deal with controlling the size of the dynamic array. Next we add a new person to the array, and then display that person. Finally, we add the name of that person to the list box.

Simple really! But haven’t we missed a step? How did we get the name of the person, and all the other details into the object?

TPerson.Create

Well, the answer is that we have modified the constructor. All objects have a constructor. This is normally called Create and, if not defined, is inherited from the parent object. But we have our own constructor and here it is:

```
inherited;
frmThisPerson :=
    TfrmCreatePerson.Create(nil);
try
    frmThisPerson.ShowModal;
    SetName(frmThisPerson.GetName);
    SetAddress(frmThisPerson.GetAddress);
    SetPostCode(frmThisPerson.GetPostCode);
    SetBirthDate(frmThisPerson.GetBirthDate);
    SetSex(frmThisPerson.GetSex);
finally
    frmThisPerson.Free;
end;
```

This needs examining carefully. First we do call the inherited Create. We do this with the simple statement *inherited*.

Next we create and show a modal form to collect the data. Then from the form we collect the data and use the methods to set the relevant details. Finally releasing the form.

Note a number of things here:

- First, calling the form is part of the object (though the form isn’t).
- Next, as we will see, the form does the validation.
- Finally, the constructor sets the necessary object values at creation time.

Other trivia, such as error trapping, I hope is well understood, and although not in the remit of this article, is worth a slight mention.

The hourglass is always the best error trapping example. How do I ensure that the hourglass is removed when an error occurs? The **try...finally...end**; construct is the best for this. This is what we have used here. But the real issue is that object programming is about using error notification (the raising of error conditions) as the error model, and this is the direction to be thinking in terms of error handling.

The Person Create Form

Here is the form:



It has all the relevant data within the form itself. This raises an interesting point about objects. Although it is not mandatory, it is clear common sense that the creation of an object should initiate all the mandatory data for that object at creation time. We have chosen to do that by collecting the data through a form. It could just as well be done by defining a constructor with all the parameters or a data structure to do that. It could be that the object works out the media being used, and as necessary issues a web form. In some respects, it does not matter which way you go. In fact you may choose to do it both ways by defining two constructors - one that takes data as parameters, and one that uses a form, calling whichever one is appropriate in the circumstances. The important thing to consider is that it is very worth while making data collection and validation part of the object implementation.

By doing this, you place all the validation in one place and do not have to write validation code in all your applications. When you find bugs or need enhancement to your validation, there is only one place to do the work.

The OK Button

Back to the mainstream, we need to ensure that the data is valid, so let’s look at the code for the OK button on the form:

```

var
  blGo: boolean;
begin
  blGo := False;
  if length(trim(txtName.Text)) > 0 then
    if length(trim(txtAddress.Text)) > 0 then
      if length(trim(txtPostCode.Text)) > 0 then
        blGo := True
      else
        showmessage('No Post Code')
    else
      showmessage('No Address')
  else
    showmessage('No Name');
  if blGo then
    Close
  else
    ModalResult := 0;

```

The validation is pretty trivial here, but it serves its purpose. The data is checked for being there and, if it is not a message, is displayed. The form is only allowed to close if all the data is there. So our Person constructor demands the data and does not complete creation until the data is valid.

Reviewing the progress so far.

The Add Person button

- Controls the dynamic array size.
- Adds a newly created person to the array.
- Displays the person.
- Adds the person to the listbox.

In creating a person the Add Person button has called the Person constructor - Create. This

- Opens the Person Create form.
- Which collects and validates the data.
- Collects the information from the form.
- Sets the relevant data items of the object.
- Releases the form.

The Add Employee Button

From now on, the next stage is so obvious, it is relatively simple. But let's look at it anyway. How does the add employee button work?

Well, to be realistic, it does exactly the same as the Add Person button.

- It controls the dynamic array size.
- It adds a newly created Employee to the array.
- It displays the employee.
- It adds the employee to the list box.

So the only difference between the two buttons is that:

```

aPeople[intThisEmployee] := TPerson.Create;
DisplayPerson(intThisEmployee);

```

is changed to

```

aPeople[intThisEmployee] := TEmployee.Create;
DisplayEmployee(intThisEmployee);

```

Well there is another important difference: the name of the button that starts the task is different. This is important, as we will see later.

But let us first look at the two line code change. Note that we still use the same array. We are using the array to hold an Employee object as well as a Person object. The beautiful thing about the dynamic array is that we can do exactly this. We can hold objects that are inherited from the base class that the array is defined to hold. So we can hold any of our descendants of TPerson.

For the moment, let us take a side stream and look at how we can use Delphi objects to take advantage of what we have already done, before we look at the employee and other of our own object creations.

Single Code for All Buttons

I have placed speed buttons on a panel. These speed buttons are named:

```

spbtnPerson
spbtnEmployee
spbtnHourlyEmployee
spbtnCommissionedEmployee

```

So my name can tell me how to do the two lines of code that are different for each object. There are multiple ways of handling this, and each of you will have your own favourite. Here is a simple piece of code that is the event handler for all of the buttons:

```

procedure TfrmMain.spbtnCreateObjectClick
  (Sender: TObject);
var
  intThisEmployee: longint;
  chrDefineCreate: char;
begin
  chrDefineCreate :=
    TSpeedButton(Sender).Name[6];
  {spbtn = 5, so next char starts one of
  Person
  Employee
  HourlyEmployee
  CommissionedEmployee}
  SetLength(aPeople, intArraySize);
  intThisEmployee := intArraySize - 1;
  inc(intArraySize);
  case chrDefineCreate of
    'P':
      begin
        aPeople[intThisEmployee] := TPerson.Create;
        DisplayPerson(intThisEmployee);
      end;
    'E':
      begin
        aPeople[intThisEmployee] := TEmployee.Create;
        DisplayEmployee(intThisEmployee);
      end;
    'H':
      begin
        aPeople[intThisEmployee] :=
          THourlyEmployee.Create;
        DisplayHourlyEmployee(intThisEmployee);
      end;
    'C':
      begin
        aPeople[intThisEmployee] :=
          TCommissionedEmployee.Create;
        DisplayCommissionedEmployee(intThisEmployee);
      end;
    else
      showmessage('Error is system - sbtn ` +
        TSpeedButton(Sender).Name);
      end;
  lbxPerson.Items.Add(aPeople[intThisEmployee].Name);
end;

```

To take you through the code:

First, we know that the object initiating this code is a speed button. If it could be initiated by other objects, we would have to consider how to deal with those, but for the sake of this example, we are trying to be simple. As we know it is a speed button, we may cast the Sender to a speed button and extract the sixth character from its name. The first five characters are "spbnt" so the sixth will uniquely identify the operation we wish to perform.

Next we do the initial common work of handling the dynamic array growth.

Then we use our character to define which two lines of code we run next.

And, finally, we add the name of the person to the list box. Note that in this we treat the object as a Person object, whatever we have done, and take it from the dynamic array.

But we haven't written the Employee code yet, so let's look at the form first.

The Employee Form

Here is the Employee form:

It collects the data for the Employee object that is different to the Person object. Note that by the time this form is shown, we already know the Person's name, and address. In fact, by the time we try to collect the Employee data, we already know everything about the Person as a person. Let us see why.

TEmployee Constructor

The code for this is:

```
inherited;
frmThisEmployee := TfrmEmployee.Create(nil);
try
  frmThisEmployee.Caption := 'Employee
    Details for ' + Name;
  frmThisEmployee.pnlAddress.Caption :=
    Address;
  frmThisEmployee.ShowModal;
  SetEmpNo(StrToInt(frmThisEmployee.txtEmpNo.Text));
  SetDateStarted(frmThisEmployee.dteDateStarted.DateTime);
  SetBaseSalary(StrToFloat(frmThisEmployee.txtSalary.Text));
finally
  frmThisEmployee.Free;
end;
```

Note that the first thing we call is inherited. Who do we inherit from? The answer is TPerson. So the first thing we are doing is calling the TPerson constructor which collects the information for the Person part of our object.

Then we create the Employee data collection form and place the person's name and address in this form.

Then we show the modal form, collect the data and move on. Much as we have done before.

Display Employee

The code for displaying the employee is much as would be expected:

```
DisplayPerson(intThisPerson);
txtEmpNo.Text := IntToStr(TEmployee(aPeople
  [intThisPerson]).EmployeeNumber);
txtDateStarted.Text := DateToStr(TEmployee
  (aPeople[intThisPerson]).DateStarted);
txtBaseSalary.Text := FloatToStr(TEmployee
  (aPeople[intThisPerson]).BaseSalary);
lblSalary.Caption := 'f' + FloatToStr
  (TEmployee(aPeople[intThisPerson]).Salary);
```

First we call the DisplayPerson code, and then we display the employee specific details.

Note, however, that it is at this point that we display the salary. For this purpose, I have made the employee object return a value of 0 for the salary. But if we never saw the TEmployee object (as is most likely) this would not be a necessary step.

The Employee Page

So our application can show the employee page with relevant data displayed:

The person's name has been added to the list box, and the details are there including calculated salary.

Collecting the rest of the object data collection is really more housekeeping and much of the same. However, it should be noted that the Hourly employee and the commissioned employee have different ways of calculating salary, but the correct answer is still displayed.

The List Box

The list box has been used to store all the names of the people that we are adding to our system. It is also useful for displaying the data. The code for the On Click event of the list box is:

```

if aPeople[lbxPerson.ItemIndex].ClassName =
    'THourlyEmployee' then
    begin
    DisplayHourlyEmployee(lbxPerson.ItemIndex);
    ClearCommissionedEmployee;
    end
else if aPeople[lbxPerson.ItemIndex].ClassName
    = 'TCommissionedEmployee' then
    begin
    DisplayCommissionedEmployee(lbxPerson.ItemIndex);
    ClearHourlyEmployee;
    end
else if aPeople[lbxPerson.ItemIndex].ClassName
    = 'TEmployee' then
    begin
    DisplayEmployee(lbxPerson.ItemIndex);
    ClearHourlyEmployee;
    ClearCommissionedEmployee;
    end
else
    begin
    DisplayPerson(lbxPerson.ItemIndex);
    ClearHourlyEmployee;
    ClearCommissionedEmployee;
    ClearEmployee;
    end;

```

Notice here that we are relying on the indexing working simply. But this is just a demonstration. The index of the list box will be the same as the index of the dynamic array. So we can now extract the object from the dynamic array. Again for the sake of simplicity, the code is not very elegant.

We do know that we have stored various different types of object in the dynamic array. So when we extract the object, we need to know which type of object it is so that we can deal with it appropriately. We do this by asking value of the property `ClassName`. (We could also use this `is` syntax)

Having worked out the class we have stored, we know exactly how to deal with it. We display the relevant information, and clear the fields that are not relevant.

A Recap

Inheritance

A Person is inherited from an object, but as far as we are concerned, this is the base class with which we have been working.

Employee and its descendants do not need to worry about the implementation of the Person part of the object. This is all taken care of by the parent.

Abstraction

An Employee object abstracts the behaviour of a Person object. There is no code within the Employee object to do the work required to manipulate the Person part of the object.

Polymorphism

An hourly employee is still an employee, as is a commissioned employee. So a salary calculation can be placed at the employee level. The correct calculation is called dependant upon the ultimate class of the object. This is polymorphism. We write as many variants of the Salary routine as we have polymorphs, and the correct calculation is used.

Some Rules for Objects

As plagiarism is the highest form of flattery, I thought that I might end this with some good rules defined by Marco Cantu in an article in the Delphi Magazine. I would tend to agree



with these, even though the code samples associated with this article do not completely go along with this. However, it is more appropriate to point you to the article. It is "20 Rules for OOP in Delphi" and is in issue 47 of The Delphi Magazine.

For more information on what EPCoT does, please see our site at www.epcot.co.uk