

7.0 Object-oriented programming

Topic A: Understanding classes

Structured vs. object-oriented programming

Various approaches to program development have evolved over the years, e.g. structured programming, object-oriented programming, generic programming, etc. These approaches are not mutually exclusive but can build on techniques and experience gained from earlier developments.

The motivation for the development of these approaches has been the need to develop large applications. What is a large application?

A very small application could be written as simply a sequence of statements (source code). As an application becomes larger, it can become unwieldy and the code needs to be divided into more manageable chunks of code (procedures). Each procedure should be about a single page and no more. This is not for any reason to do with computers, but because we as human beings make far fewer mistakes if we can see an entire procedure on a single screen. This is why many programmers set their font to be very small – they can get more code on a page!

Structured programming is based on the approach of dividing the functionality of the application into procedures. Data that these procedures use can tend to be scattered and needs to be passed around or held globally (which is not recommended).

In the old days, before object-oriented programming, you had your data, and the procedures that acted upon the data. These were two separate things. It is possible to write very large programs that successfully join the 2 together. However, as programs grow larger, it becomes far more likely that the programming team will introduce errors into the code. For example, if a piece of data needs to be accessed by many different parts of an application, it is tempting to simply make the data Global. This is an easy option for a programmer, but is asking for trouble – how do you ensure that each procedure is going to obey all your business rules when it accesses or amends that data?

Let's look at a couple of examples, to see where the problems lie.

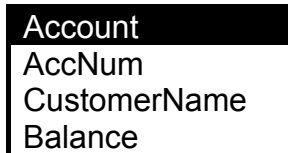
For example, we could have a program to hold bank account details. Each bank account has an account number, a customer name and a balance. Each of these could be held as separate items of data, as shown below:

```
Dim AccNum As Integer
Dim CustomerName As String
Dim Balance As Integer
```

Here, we're not even collecting our data together into a single structure. When manipulating these values, it's easy to make mistakes, for example update bits of different accounts.

Collecting the data together

The most obvious step to help avoid making mistakes is to collect all the data together in some way. Traditionally, this was done by creating a Structure, with fields for each of the important values. For example, we might create a Bank Account structure:



In code:

```
Public Structure AccountStruct
    Dim Balance As Integer
    Dim AccNum As Integer
    Dim CustomerName As String
End Structure
```

We can now create variables of this type, and define procedures to access the fields, e.g.:

```
Dim a1, a2 As AccountStruct
```

We could then write procedures to access the data, e.g.:

```
a1.CustomerName="Fred"
Withdraw(a1,50)
```

Within these procedures, we could place any required business rules (e.g. can't withdraw more than overdraft limit etc).

However, there's nothing to stop a programmer accessing the data directly, e.g.:

```
a1.Balance = 50
a2.Balance = 250
```

Although this method can and has been used to create extremely robust applications, it's still error-prone. For example, each programmer is free to access the data in whatever way they want – some programmers will write excellent code, others will write awful code.

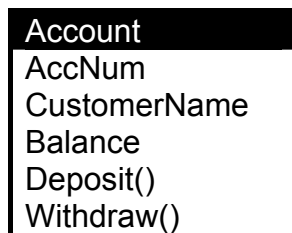
Object oriented programming (OOP)

Object Oriented Programming (OOP) was invented in an attempt to force programmers to adopt good programming practices – e.g. associate all data with appropriate methods, and force programmers to use these methods to access each piece of data. OOP does this by providing a way of associating data and code. Many items of data within an application are not independent. With OOP, we create a **Class**, which defines both the data and the procedures that can be used with that data.

Amongst other useful tools, OOP provides the following techniques:

- Aggregation: group the related data together (use of Classes)
- Encapsulation: associating functionality with data (Methods and Properties).

Diagrammatically, we could show this as:



We can then go on to force other programmers to use the procedures Deposit() and Withdraw() in order to access the data. Within these procedures, we will see that it's relatively easy to incorporate any business rules we need (e.g. can't withdraw more than overdraft limit etc).

- AccNum, CustomerName and Balance are referred to as Properties.
- Deposit() and Withdraw() are referred to as Methods

Creating a new class

Here's the code required to create a new class:

```
Public Class Account
    ' 1st the Fields (Properties). The "c" in front of the variable name
    ' indicates ' it is "Class-wide" ie accessible from anywhere in our
    ' class.

    Public cBalance As Integer
    Public cAccNum As Integer
    Public cCustomerName As String

    ' Now the Methods:
    Public Function Deposit(...) As ...
        ...
    End Function

    Public Function Withdraw(...) As ...
        ...
    End Function
End Class
```

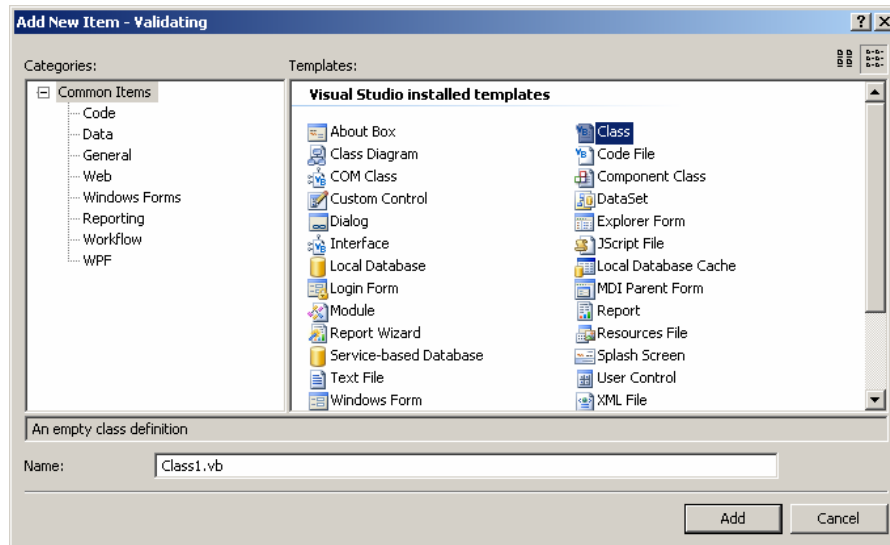
There's still a lot more detail to learn, but as we expand this fairly simple idea, we'll see how we can force our programmers to adopt good programming practices, which means our applications are far more likely to be robust and scalable (i.e. work!).

Classes

The code we saw above could easily be placed in the same .vb file that your application uses to display its opening (main) form. However, we can do better than that. It's better to create a separate file which contains just the code for the new class. In Visual Studio, there are predictably several ways to achieve this. For example, you can choose any of the following:

- Project, Add Class...
- Project, Add Component...
- Project, Add Module...
- Solution Explorer, Right-Click on the Project, Add Class...
- Solution Explorer, Right-Click on the Project, Add New Item...

Selecting Add Class leads to the Add New Item dialog, shown on the next page:



If we change the name to Account.vb and click Add Visual Studio.NET will now present you with a new Tabbed window, called Account.vb. It looks much the same as when you create a new project, except it has no Designer window, and it only contains the following code:

```

1 Public Class Account
2
3 End Class
4

```

You simply type in the code shown previously into this window.

As we said, storing data and functions in a single unit (class) is known as Encapsulation.

Creating variables using classes

Having written the code to create the class, you might be tempted to declare variables with the following code. Back in your main form (i.e. your code test harness front-end), perhaps behind a button labelled "Create New Account":

```
Dim a1 As Account
```

Note that as soon as you create your new Class (e.g. Account), Intellisense automatically knows about it, and will present Account as one of the choices after you type in:

```
Dim a1 As
```

However, the code shown above is insufficient. We need to also create a new instance of our class.

Creating an instance of a class

When you define a Class this only defines a template for a new type. Each instance of that type you create will hold all of the data for the declared fields within that class.

The code above would produce no compilation errors. However, if you try and use your new variable (e.g. assign values into it), your program will fall over. Instead, you must create a new instance of the class by using the keyword **New**. You can create a new instance of a class at one of two times:

1. At the point of declaration of a variable, or
2. assigning to a variable that was declared earlier.

Examples of these are shown below:

```
Dim a1 As New Account ' New instance of Account created, or:
```

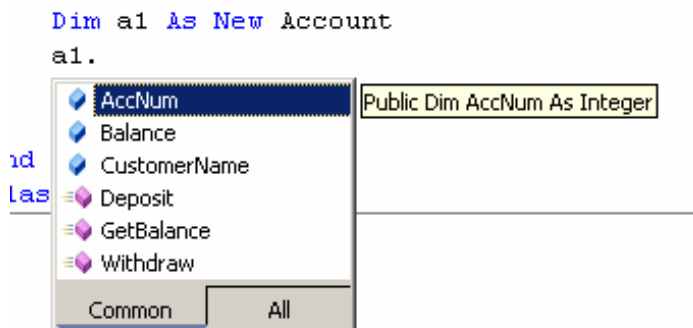
```
Dim a1 As Account ' Declare the variable  
a1 = New Account() ' New instance of Account created
```

Class - new reference type

When you define a class you are defining a new type. These new types are Reference types and declaring a variable of a class type does not allocate any memory for an object of that type. What you will have allocated is a reference (can be the special value, Nothing), which can hold the location (address) of an object of the class type. It's not until you type **New** that the compiler actually allocates memory to hold your object.

Accessing properties and methods

So how do you access the Properties and Methods associated with your new object variable? Easy! Type the object name, followed by a "." Intellisense will automatically present you with a list of all the known Properties and Methods for that type, e.g.:



We can see several important points here:

- Fields get a little blue box next to them
- Methods get a little purple box, which looks like it's just flown in from the left.

So all we need to do is something like:

```
Dim a1 as New Account  
  
a1.AccNum = 001  
a1.CustomerName = "Fred Bloggs"  
a1.Balance = 100
```

Of course, in a real program, you wouldn't hard-code these values as above. Instead, you might launch a dialog box, and prompt the user to enter the appropriate details. You could then put in some validation rules to check the input (e.g. it's a number; AccNum must be 6 digits etc).

So what are we going to put in our functions? Let's keep it simple, and expect our functions to work something like this:

```
a1.Deposit(450)  
a2.Withdraw(200)
```

In which case, all we need is something along these lines:

```
Public Function Deposit(ByVal amt As Integer) As Integer  
    cBalance += amt  
    Return cBalance  
End Function
```

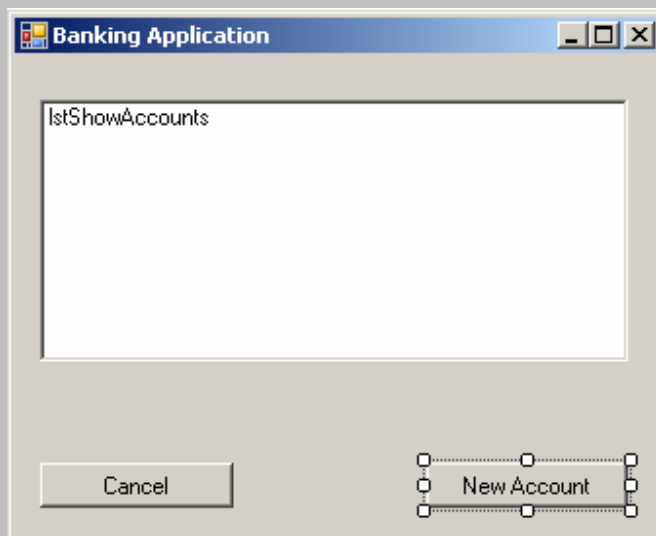
```
Public Function Withdraw(ByVal amt As Integer) As Integer  
    cBalance -= amt  
    Return cBalance  
End Function
```

In fact, we've got quite a long way to go yet before we've written an OO program. However, it's time to pause for a while, and get you to digest what we've learned so far (that means it's time for an Exercise, where you can type this lot in!)

Creating a new class

This exercise gets you to create a new Class of your own, and add a couple of functions.

1. Open a new Visual Basic Windows Application project, called BankAccounts. Design a main form called frmMain that looks something like this:



The big white space is a ListBox, called something like lstShowAccounts.

2. Add a New Class to your project, called Account. Inside this class, define 3 public variables, called cAccNum, cBalance and cCustomerName. Also define 2 methods, Deposit() and Withdraw().
3. Back in frmMain, behind the New Account Button, hard-code a new account. Give values for the Account Name, Number and Opening Balance.
Display your results in the ListBox, perhaps using code something like the following:

```
lstShowAccounts.Items.Add(a1.AccNum & " " & _  
    a1.CustomerName & " " & a1.Balance)
```

Best of all, create a subroutine called, for example, Display_Account(ByVal a As Account) which does the displaying.

4. Try out your Deposit() and Withdraw() methods. Again, just hard-code a deposit or withdraw into the click event of the New Account button. Again, display the results in the list box.

Answers to exercise

The completed code for the Account class is as follows:

```
Public Class Account
    Public cBalance As Integer
    Public cAccNum As Integer
    Public cCustomerName As String
    Public Function Deposit(ByVal amt As Integer) As Integer
        cBalance += amt
        Return cBalance
    End Function
    Public Function Withdraw(ByVal amt As Integer) As Integer
        cBalance -= amt
        Return cBalance
    End Function
    Public Function GetBalance() As Integer
        Return cBalance
    End Function
End Class
```

The code for the Form1 is as shown:

```
Public Class Form1
    Private Sub btnNew_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnNew.Click
        Dim a1 As New Account
        a1.cAccNum = 1001
        a1.cCustomerName = "Fred"
        a1.cBalance = 200
        Call Display_Account(a1)
        a1.Deposit(50)
    End Sub
    Sub Display_Account(ByVal a As Account)
        lstShowAccounts.Items.Add(a.cAccNum & " " & a.cCustomerName & " " & a.cBalance)
    End Sub
End Class
```