Hands-on Exercise 1: VBA Coding Basics

This exercise introduces the basics of coding in Access VBA. The concepts you will practise in this exercise are essential for successfully completing subsequent exercises. It assumes that you already have some familiarity with basic programming concepts.

Even if you have already coded in VBA you should review this exercise to make sure you are familiar with the topics it contains.

Learning Outcomes

When you have completed this exercise you will:

- Be able to recognise components of the VBA development environment
- Be able to select appropriate variable types
- Be able to name variables appropriately
- Be able to write Sub and Function procedures and understand the difference between them
- Understand the difference between Private and Public procedures
- Use the VBA development environment tools for debugging your code
- Use the VBA Help facility

1.1 Using the VBA development environment

Create a new database

For the first exercise in this module you will familiarise yourself with the basics of VBA programming using a new blank database that you create specifically for this purpose. For the remaining exercises you will use the Yum Juices database, introduced in Hands-on 2.

Create a new blank database (you should know how to do this). As you will just be using this database as a 'scratch-pad' for your initial exploration of VBA you should name it something like vbaTest.mdb

Create a VBA module

Clicking the New button in the modules tab New to open a new module.

You are now taken to the VBA development environment which you use to write all program code for your database. Before you start, spend some time familiarising yourself with the various components of this environment.

Try out code in the Immediate Window

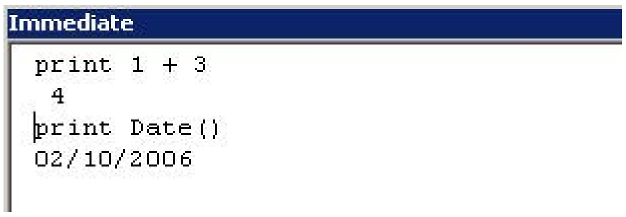
If it is not already open, display the Immediate Window at the bottom of the screen. You can use this window to try enter and test VBA statements.

Note: A statement is a line of program code that will carry out an action.

Try this out by typing the word `print` followed by a VBA statement and then press the Enter key to see the output. Enter the following statements:

```
print 1 +  3
print Date()
```

The output in the Immediate window should appear as in the screenshot below:



```
print 1 + 3
 4
print Date()
02/10/2006
```

Note that in the second example we have used the built-in Access function Date() which displays the current date (so your result will be different from that in the screenshot above!).

You can use VBA to do date arithmetic. Type the following in the Immediate window and press the Enter key.

```
? Date() + 7
```

You should now see the date a week from today in the Immediate window.

In this example you have typed `?` instead of `print`. It works in the same way but reduces the amount of typing you need to do.

You will learn more about date functions later in this exercise.

Write a sub procedure and test it using the Immediate window

As well as entering code directly in the Immediate Window, you can use it to check the output from code you write for your own procedures. In this example, you will write a Sub procedure. (You can also write your own function procedures. The difference between these two types of procedure is covered later in this exercise.)

Type the following code in the code window:

```
 Sub PrintString()
     Debug.Print "Birkbeck"
End Sub
```

Once you press enter at the end of the first line, the VBA editor automatically inserts the `End Sub` statement for you. This is the closing statement of the procedure. The code that you write will be inserted between these two statements.

You'll also notice that these words appear in blue type indicating that they are reserved words in the VBA programming language. What other reserved words have you used in this piece of code?

Note: Although reserved words are case-sensitive – in other words, you must use `Sub` and not `sub`. - you don't need to worry about this when you are typing code as the VBA editor automatically inserts the correct capitalisation for you.

After the word `Sub`, indicating to the VBA compiler that this is a sub procedure, you enter the name for your procedure – in this case `PrintString`. This is a name that you choose yourself and you should aim to use a descriptive name that gives some indication of the purpose of the code. Note that you must not use spaces or reserved words in procedure names.

The procedure name is followed by brackets (). In this case, the brackets are empty as there are no arguments to be passed to the procedure. (Later in this exercise you will write functions and procedures that do take arguments.) If you don't type the brackets, the VBA editor inserts them for you.

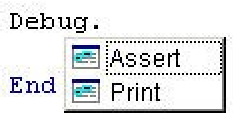
The lines between the `Sub` and `End Sub` statements are the body of the procedure. Each line of the code is a VBA statement. (There is no punctuation used to mark line-endings in VBA as there is in some other programming languages.)

There is just one statement in this subroutine:

```
Debug.Print "Birkbeck"
```

`Debug` is an object. VBA is an object-*enabled* programming language, not strictly object-oriented, but it has many of the features of other object-oriented languages such as Java.

Objects have methods – the things that they know how to do. You will have noticed when you typed the full-stop after `Debug` that a box popped up listing the Debug object's methods.



```
Debug.
End
```

You can either type the name of the method or select it from the list to have it inserted automatically. In this case, the `Debug` object has just two methods. `Print` which is used to display the value in the Immediate window - and `Assert`. Use the VBA Help to find out about `Assert`.

The final part of the `Debug.Print` statement is the output list – a list of values to be output in the Immediate Window. In this case we will output the word Birkbeck. This is enclosed in double-quotes as it is a sting literal (in other words its value will remain the same each time the procedure is run).

By now you should know what you expect to happen when you run this subroutine. Run it by positioning the insertion point anywhere within the code and pressing F5 (the function key in the top row of the keyboard). When the procedure runs, the word Birkbeck appears in the Immediate window.

Using variables

Often, rather than printing a literal value, we want to print a value that is stored in a variable. Modify the subroutine to use a variable as follows:

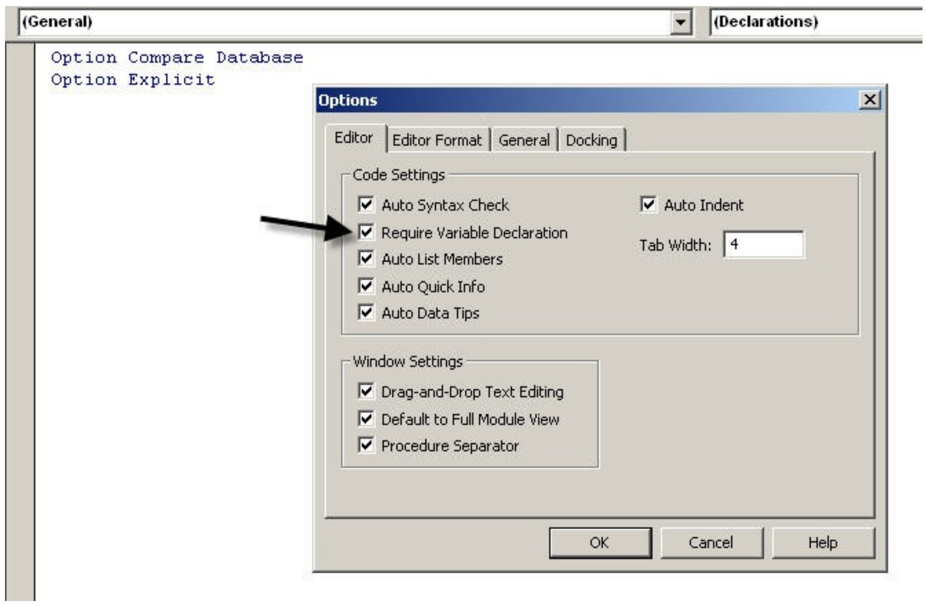
```
Sub PrintString()
 CollegeName =  "Birkbeck"
 Debug.Print CollegeName
End Sub
```

This time you are assigning the value "Birkbeck" to a variable called `CollegeName` and then printing the value of the variable in the Immediate window. Run it as above and you should get the same output as before.

In this example, VBA has automatically created a variable named `CollegeName` the first time you used it. It assumes that any word that is not a reserved word is a new variable and automatically creates one for you. This is dangerous. What happens if you mis-type the variable name in the `Debug.print` statement? Try it. Change `CollegeName` to `CollageName` and run the code again. This time there is no output. VBA creates a new variable called `CollageName` and prints its contents, which is nothing as you have not yet assigned a value to it. To avoid this problem you need to declare your variables before you use them and prevent VBA from automatically creating variables.

Declaring variables

To prevent VBA from automatically creating variables you need to add the `Option Explict` declaration at the top of the module. The VBA editor will then flag an error if you attempt to use variables without declaring them. As it is good practice to use this declaration in each module that you write, you should set Access so that this declaration will always be inserted at the start of a module by clicking Tools > Options and then checking the Require Variable Declaration box.



```
Option Compare Database
Option Explicit
```

Options

Editor | Editor Format | General | Docking

Code Settings
- ☑ Auto Syntax Check
- ☑ Require Variable Declaration
- ☑ Auto List Members
- ☑ Auto Quick Info
- ☑ Auto Data Tips
- ☑ Auto Indent
- Tab Width: 4

Window Settings
- ☑ Drag-and-Drop Text Editing
- ☑ Default to Full Module View
- ☑ Procedure Separator

Declaring your variables also makes your code more efficient as any variable that is not declared uses the Variant data type which uses more storage and slows down the program as Access needs to work out what type of data it contains before using it.

Add a declaration for your variable to the procedure.

```
Sub PrintString()
    Dim strCollegeName As String
    strCollegeName = "Birkbeck"
    Debug.Print strCollegeName
End Sub
```

It is good practice to follow the naming convention when naming your variables. In this case we have used the three-letter prefix str to indicate that the variable is intended to hold a string. This is simply a convention to make the code easier to maintain, it does not create a string variable. To create a variable of a particular type, the variable name must be followed by the word `As` followed by the data type for the variable. In this case, `String`. You should notice that once you press the spacebar after typing the word `As`, a dropdown list of all the available variable types appears. This is another of the ways that the VBA editor helps you when you are entering code. You can either type the name of the data type or select the one you want to use from the list as you did for the `Print` object's method above. If you do not explicitly declare a type for a variable it will be created with the Variant data type.

Now that you have named your variable strCollegeName, don't forget to change all the uses of the variable name in the code before running it.

What happens now if you misspell the variable name in the `debug.Print` statement and then try to run the code? What happens if you try to run a procedure using a variable that has not been declared?

Type 'Comparison of data types' into the VBA Help search box to get a full listing of the data types you can use in VBA.

Passing arguments to procedures

Rather than hard-coding the value to be printed, as in the example above, we can make the code more flexible by passing an argument to the procedure to allow us to print a different value each time it is run.

To do this you place argument(s) representing the value(s) you want to pass to the procedure inside the parentheses following the procedure name. If there is more than one argument to be passed, they should be separated with commas.

Modify your procedure (or write a new one) to take the string as an argument. (Don't forget to use a different name for the procedure if you write a new one. You can't have two procedures with the same name.)

```
Sub PrintString(strCollegeName As String)
    Debug.Print strCollegeName
End Sub
```

The code is now much shorter as the argument is used as a variable so we don't need to declare the variable within the body of the code or assign a value to it.

Now that you need to pass an argument to the procedure you can no longer run it using F5. To run it, type the following in the immediate window:

```
Call PrintString3("UCL")
```

Note that you need to enclose the argument in double quotes as you are passing a string to the function. The example above uses the string "UCL" but you can call the procedure using any string you like.

Other useful coding techniques

Comments

It is good practice to add comments to your code as this makes it easier to maintain. You should generally add a short comment at the start of each procedure to explain its purpose. Within a procedure there is no need to comment every line but it can be helpful to add comments at points where you think the code is not self-explanatory.

To insert a comment in your code, start the line with an apostrophe. Comments appear in green in the VBA editor. Comment lines are ignored by the compiler and do not slow down the execution of your code.

Add a comment to explain what your PrintString procedure does. For example, this could be

```
'prints the string passed as an argument
```

You can also use comments to record the name of the person writing the procedure and the date on which it was written.

Continuation lines

The lines in the code above are short but some lines of code can be rather long. When a line becomes too long to read comfortably in the editing window you can't simply press Enter to start a new line as VBA regards the new line character as the signal for a new statement. In order to continue the same statement on a new line you need to use the line continuation character – a space followed by an underscore. Although this is not necessary in the code you have written above you can test it out by splitting the print line as in the example below:

```
Sub PrintString3(strCollegeName As String)
    Debug.Print _
    strCollegeName
End Sub
```

Line continuation character

Save the module

When you create a new module from the database window as you did at the start of this exercise, it is listed in the Project Explorer (left-hand top) as Module1 (or Module*n* where *n* represents the number of new modules you have created since the last time you opened your database). When you close the VBA editor you will be prompted to save any modules that you have not already saved or changes to modules that you have modified since they were first created. You can also save a module at any time by clicking the save button or using the Save command from the File menu.

Before proceeding further, save your module with the name modTest. Note that we are using the prefix *mod* to indicate that this is a module. It will be listed in the modules pane of the database window with this name. Some developers use the prefix *bas* to indicate that this is a VBA code module. You can use this prefix if you prefer but don't mix and match prefixes; either name all your modules as mod*Name* or all as bas*Name*.

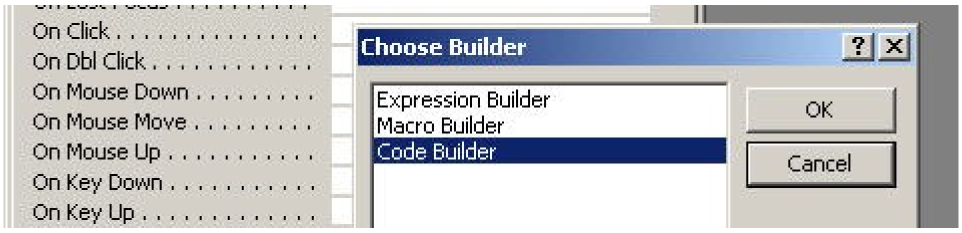
1.2 Event Procedures

A common use of VBA is to handle events that occur for forms and their associated controls or for reports. In later exercises you will be using events to automate your database. This section introduces the basic concepts.

From the database window, create a new, blank form in design view. Name your form frmTest. You will now create a button on this form that will close the form and at the same time check whether this is really what the user wants to do.

Add a command button with the wizard switched off. Set the button's Name property to cmdClose (note the use of the naming convention) and its Caption property to Close. (You should know how to do this.)

Open the properties for the button and select the Event tab. Click the ellipsis next to the On Click Event and then select Code Builder from the Builder dialog box.



This takes you to the VBA editor where Access has created a procedure stub for the On Click event of your button. In this case, it has also created a new module for the form that now appears in the Project Explorer list. Any new code that is added to the form will be included in this module.

```
Private Sub cmdClose_Click()

End Sub
```

You will write your code inside this stub.

Note that Access has named the procedure with the name of the button followed by the name of the Event that the procedure will handle. If you did not name the button as described above, it will have the default Access name of Command*n* (where *n* is a number representing the number of command buttons you have created). If you later change the name of the button you will also have to change the name of the procedure. Access will not change it automatically. You should also be aware that if you delete the button, you also need to delete any related code from the form's module. This will not be done automatically for you.

Enter the following code:

```
Private Sub cmdClose_Click()

 MsgBox "Close form?", vbOKCancel
 DoCmd.Close

End Sub
```

The first line of code calls the MsgBox function. This function takes five arguments but only the first – the prompt to be displayed to the user – is required, the other four are optional. The second argument specifies the buttons to be displayed. In this case the box will contain two buttons Ok and Cancel. If no value is passed for this argument, the box will just display the Ok button. Check the VBA Help to find out about the other MsgBox arguments and the other types of button that can be displayed.

The second line uses the DoCmd object. You use the DoCmd object to run Access actions from VBA code. The methods of the DoCmd object largely correspond to the macro actions that you have used in the macro design pane.

Check the VBA Help to find out more about the DoCmd object.

To test the code, switch back to form view and click the command button.

As your code does not specify what should happen when the buttons are clicked, all that happens at the moment is that the message box is displayed and the form closes when we click either of the buttons. To make Access respond differently to each of the buttons we need to use the return value of the message box in the code.

Note: MsgBox is a *function*. This means that it returns a value – in this case the return value corresponds to the button that the user has clicked. You can use this value in your code to select between different courses of action depending which button the user clicks. To implement this conditional execution, you need to use the If statement.

To do this, modify your code as follows:

```
Private Sub cmdClose_Click()
    Dim intResponse As Integer
    intResponse = MsgBox("Close form?", vbOKCancel)
    If intResponse = vbOK Then
        DoCmd.Close
End If
```

The modified code first declares an integer variable and then assigns the return value of the message box to this variable. The If statement checks this value and only executes the close command if the user has clicked the OK button. As no action is specified for the Cancel button, clicking this button simply closes the message box and leaves the form open.

Change back to form view and click the button to test your modified code.

Note that the return value of the message box is an *integer* but the code then compares it to the VBA *constant* vbOK. Each of the message box buttons has an integer value associated with it but it is easier to use constants in the code as it means it is not necessary to memorise the return value for each button. The return value for the vbOK button is 1. If you modify your code to use this value in place of the constant it will work in exactly the same way. You can test this for yourself but you should use the constants when writing code as it makes it clearer. Check the VBA Help to find out the return values for the other message box buttons.

1.3 Date functions and conditional execution

VBA has several built-in functions for dealing with dates. You have already used the Date() function to display today's date in the Immediate window. You will now create a new module to store procedures to deal with dates. Create a new module and name it modDateUtilities.

You will first write a Sub procedure using the DateDiff function to calculate the number of days until Christmas. The DateDiff function finds the difference between two dates. It takes five arguments, but the last two are optional so we only need to concern ourselves with the first three. These are: *Interval*, *Date1*, *Date2. Interval* is a string which identifies whether the difference should be returned in days, “d”, months, “m” or years “yyyy”. The date arguments are the two dates we want to compare.

Try this out by entering the following code:

```
Sub DaysToChristmas()
    Dim datToday As Date
    datToday = Date
    Debug.Print "There are " & DateDiff("d", datToday, #12/25/2007#) _
                     & " days until Christmas"
End Sub
```

Use the F5 key to run this code from the editing window and you will get a message in the Immediate window telling you the number of days until Christmas.

Points to note in this code:

The first line of code declares a variable with the data type Date. The variable name uses the prefix dat.

The variable is assigned the value of today’s date using the inbuilt Date function.

The values passed to the DateDiff function are “d” indicating that we want to find the difference between the dates as a number of days, today’s date (stored in the variable) and the date of Christmas day as a date literal. Literal dates need to be enclosed by hash # symbols. What happens if you forget to do this? (Also note that the date appears in US format mm/dd/yyy. When you enter it in UK format, dd/mm/yyyy, the VBA editor converts it to US format.)

The Debug.Print statement concatenates the return value of the DateDiff function with an explanatory message. As this statement is rather long, the example above uses the line continuation character to enter it on two lines

Other useful date functions are:

`DatePart` allows us to find a specific part of a date. It takes four arguments but the last two are optional. The first two arguments are *`Interval`* and *`Date`*. The Interval is a string which identifies the part of the date to return; use "`m`" to return the month, "`d`" to return the day part and "`yyyy`" to return the year part.

`DateAdd` lets us add values to particular parts of the date. It takes three arguments. *`Interval`* as above, the number of intervals we want to add to the date and a date to be added to. Note: `DateAdd` provides different functionality to simply using the + sign to add to a date as we did in the first example in this exercise. When you use + it just adds the specified number of days to the date.

Try these out by inserting the following lines of code into your procedure:

```
Debug.Print DatePart("m", DateToday)
Debug.Print "In five years time it will be " & DateAdd("yyyy", 5, DateToday)
```

Run the procedure again to see how these other date functions work.

Calling a function from a sub

It would be more useful if the date difference procedure was a function rather than a sub. We could then use it to return the difference between two dates and use that value elsewhere in our program code.

Modify the procedure (or write a new one) as follows:

```
Public Function DaysDiff(datFuture As Date)
    Dim datToday As Date
    datToday = Date
    DaysDiff = DateDiff("d", datToday, datFuture)
End Function
```

Note the changes in this code (highlighted).

Firstly we have changed the procedure declaration from `Sub` to `Public Function`. Declaring a procedure as `Public` means that it can be called from other modules in your program code. Procedures should all be declared as either `Private` (can only be used in the module in which they appear) or `Public` (can be called from other modules). Note that code for Event Procedures will always be Private.

Secondly, the `debug.print` statement has been replaced with an assignment statement. This assigns the return value of the built-in `DateDiff` function to `DaysDiff` which is the name of your function. This then becomes the return value of your function.

Test this out by calling the function from a button and using a message box to display the number of days until Christmas.

Create a new button on your frmTest form.

Give the button a descriptive name, cmdXmas, for example.

Create the following Event Procedure for the Click Event. (Follow the same steps as you used above to do this.)

```
    Private Sub cmdXmas_Click()
       MsgBox "There are " & DaysDiff(#12/25/2007#) & " days left until
    Christmas."
    End Sub
```

The one line of code in this Sub calls your DaysDiff function and concatenates the return value with the message.

Getting input from the user

The main reason to supply arguments to functions is that they can be varied with each call to the function. For example, rather than hard-coding the argument to the DaysDiff function as you did above, it would be more useful to get the user to supply the date to pass to the function.

To get user input we use the `InputBox` function. The return value of this function is the value entered by the user.

You will now create a button on your form that will prompt you to enter the date on which your next assignment is due and will then display a message telling you the number of days left until this date.

Create another command button on your frmTest form and create the following Event Procedure for the Click Event. (Remember to give your button a meaningful name.)

```
Private Sub cmdDue_Click()
    Dim datDue As Date
    datDue = InputBox("When is your assignment due?")
    MsgBox "Your assignment is due in " & DaysDiff(datDue) & " days from
today."
End Sub
```

Test it out.

When you click the button, an input box will open prompting you to enter a date. The code assigns the date that you enter to the `datDue` variable and passes it to the `DaysDiff` function. The return value of this function is then displayed in the message box.

Conditional execution using If ... Else

What happens if you enter a date earlier than today's date? You get a negative number indicating that your assignment is overdue. In this case, rather than simply displaying the negative number in the message box, we would like to display an alternative message box with the message “Your assignment is overdue”.

You used the `If` statement for conditional execution in your Close button code but in that case there was only one course of action to be taken – either close the form or do nothing. Now there are two possible courses of action: either output the message box showing the number of days remaining or the message box displaying the overdue message. To choose between two courses of action you need to use `If … Else`.

To do this, modify your Event Procedure code as follows:

```
Private Sub cmdDue_Click()
    Dim datDue As Date
    Dim intDays As Integer
    datDue = InputBox("When is your assignment due?")
    intDays = DaysDiff(datDue)
    If intDays < 0 Then
        MsgBox "Your assignment is overdue!"
    Else
        MsgBox "Your assignment is due in " & intDays & " days from
today."
    End If
End Sub
```

The code now tests the value of the `intDays` variable using the condition <0. If the condition is true, the overdue message is displayed, otherwise the original message box showing the number of days left appears.

Test this out by entering a date earlier than today’s date. The overdue message should display.

Now extend the code so that a message box containing the message “Your assignment is due today!” will display if you enter today's date.

```
Private Sub cmdDue_Click()
    Dim datDue As Date
    Dim intDays As Integer
    datDue = InputBox("When is your assignment due?")
    intDays = DaysDiff(datDue)
    If intDays < 0 Then
        MsgBox "Your assignment is overdue!"
    ElseIf intDays = 0 Then
            MsgBox "Your assignment is due today!"
    Else
        MsgBox "Your assignment is due in " & intDays & " days from 
today."
    End If
End Sub
```

Conditional execution using the Select Case statement

You can have as many different conditions as you want using `ElseIf` but when you have several different conditions to test, you may find that a `Select Case` statement is clearer. You will now use `Select Case` to display the same message boxes as your `If … Else` code above and an additional message box with the message "You have less than a week to do your assignment!" if there are less than 7 days remaining.

Create a new button on your frmTest form and add the following code to the Click Event. The highlighted lines show the `Select Case` code. You should be able to see how this relates to the `If … Else` that you used above.

```
Private Sub cmdDue2_Click()
    Dim datDue As Date
    Dim intDays As Integer
    datDue = InputBox("When is your assignment due?")
    intDays = DaysDiff(datDue)
    Select Case intDays
        Case Is < 0
            MsgBox "Your assignment is overdue!"
        Case Is = 0
            MsgBox "Your assignment is due today!"
        Case Is < 7
            MsgBox "Your assignment is due in less than a week from 
today!"
        Case Else
             MsgBox "Your assignment is due in " & intDays & " days from 
today."
    End Select
End Sub
```

Test the new button using different dates to test all possibilities.

Although you can achieve the same effect with an `If … Else` construction, the Select Case code will often be clearer when you have many alternatives to choose from.

1.4 Repeated execution of code

You will often want to run a segment of your code more than once.

Running code a set number of times

To run a piece of code a set number of times you use the For construction.

We can use this to print out the first ten numbers of the five times table.

Enter the following code in your modDem module:

```
Sub TimesTable()

Dim intIndex As Integer

For intIndex = 1 To 10
    Debug.Print "5 x " & intIndex & " = " & intIndex * 5
Next intIndex
```

```
End Sub
```

In this code the `For` statement defines the starting and ending value for the loop. The code between the `For` and `Next` statements will then be executed the specified number of times.

Test your code by running it with the F5 function key; the five times table should be displayed in the Immediate Window.

Do While ... Loop

This construction will run a segment of code repeatedly so long as a condition is met. You can test the condition at the start or end of the loop. . Both code examples below will give the same output. Enter them in your mod Dem Module and test them using the Immediate Window.

```
Sub ChkFirstWhile()
    Dim intCounter As Integer
    Dim intMyNum As Integer

    intCounter = 0
    intMyNum = 20
    Do While intMyNum > 10
        intMyNum = intMyNum - 1
        intCounter = intCounter + 1
    Loop
    Debug.Print "The loop made " & intCounter & " repetitions."
End Sub
```

```
Sub ChkLastWhile()
    Dim intCounter As Integer
    Dim intMyNum As Integer

    intCounter = 0
    intMyNum = 20
    Do
        intMyNum = intMyNum - 1
        intCounter = intCounter + 1
    Loop While intMyNum > 10
    Debug.Print "The loop made " & intCounter & " repetitions."
End Sub
```

Note If you make an error and your program goes into an endless loop, you can stop execution by pressing ESC or CTRL+BREAK.

Sub or function?

In this exercise you have created both Sub and Function procedures and should by now have some idea of the difference between the two. In general you use Sub procedures when your procedure simply carries out an action and does not need to return a value and Function procedures when you need to use the value returned elsewhere in your code. However, if you want to call the procedure from elsewhere in your code, it needs to be a Function regardless of whether or not it returns a value.

Public or Private?

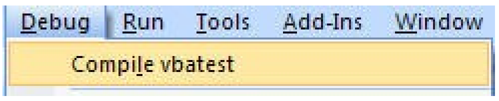
You have also created both Public and Private procedures. All procedures, apart from those you simply create for testing in the Immediate Window, should be declared as one or the other. If a procedure will simply be used in the Module in which it appears, it should be declared Private and all Event Procedures are Private. If you are creating a general purpose Module containing procedures for use elsewhere in your code the procedures should be declared as Public.

Debugging and compiling your code

You may have made errors when entering some of the code in this exercise so you will already have some experience of dealing with them!

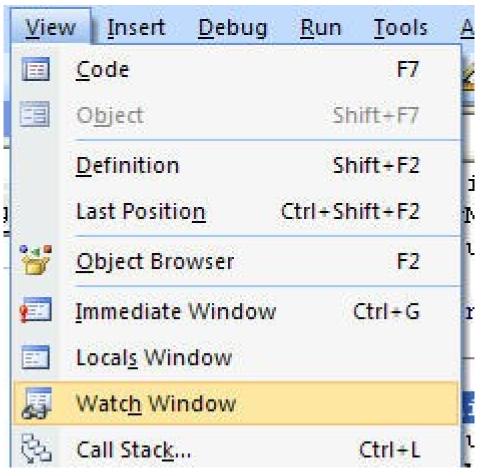
Compilation

It is good practice to compile your code before running it. To do this, click Compile from the Debug menu.

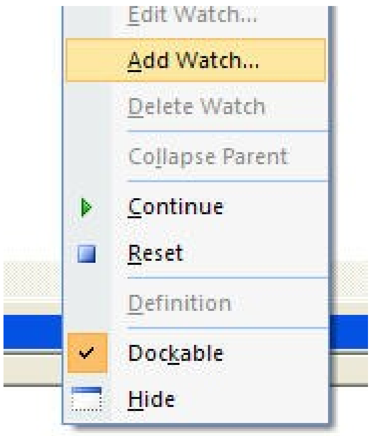


Stepping through the code

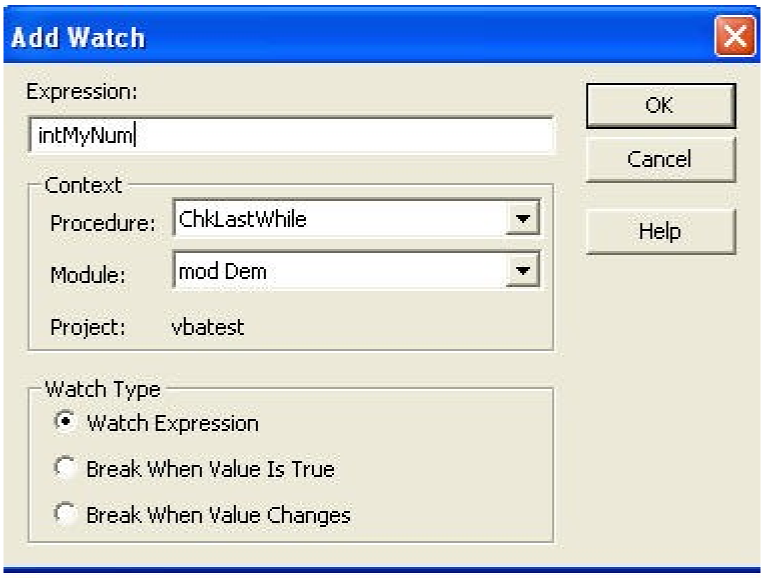
If you want to trace a problem with your code, you can step through it one line at a time. You can also use the Watch window (bottom right) to display the values of variables and expressions during code execution. Test this out using some of the code you have written for this exercise. This example uses the `ChkLastWhile` procedure from the modDem module.



If the Watch Window is not open, display it by selecting Watch Window from the View Menu.



To add a watch to the window, right-click in the Watch Window and select Add Watch.


Add Watch

Expression: intMyNum

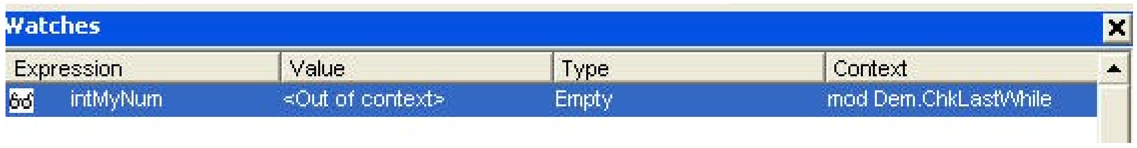
Context
Procedure: ChkLastWhile
Module: mod Dem
Project: vbatest

Watch Type
[✓ Watch Expression]
Break When Value Is True
Break When Value Changes

Enter the value of the variable or expression whose value you want to watch in the Add Watch dialog. In this case, we will watch the change in the value of the `intMyNum` variable as we step through the code.

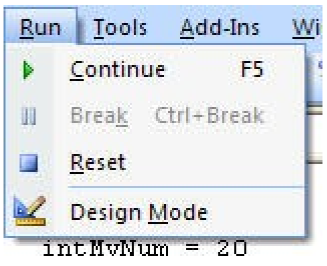
Enter this variable name in the Expression text box and click OK.

This value now appears in the Watch Window. (If your insertion point is not in the procedure it will be labelled <Out of context>.)


Watches

Expression	Value	Type	Context
intMyNum	<Out of context>	Empty	mod Dem.ChkLastWhile

You will now watch the way in which this variable changes as you step through the code for the `ChkLastWhile` procedure. Position your insertion point in the procedure and press the F8 (Step Into) function key to step through your code. (This command is also available from the Debug Menu. Each time you press the key, the next line of your procedure will be executed and changes in the value of your variable will be shown in the Watch Window.



To exit from debug mode, select Reset from the Run menu.

This combination of watching variables and stepping through the code is a useful method for tracking down problems in your code.

Try it out with some of the other procedures you have written in this exercise. For example, you could use Step Into to see the different paths through the code where you have used `If ... Else` or `Select Case` for conditional execution.

Stop and reflect

What is an Event Procedure and how do you create one?

What are the arguments for the MsgBox function?

What function do you use to get input from the user?

How does the DoCmd object relate to the Macro Actions?

What is the difference between Sub and Function procedures?

When should procedures be Public? When should they be Private?

What programming statements can you use for conditional execution?

How do you create a loop that will execute code a specified number of times?

From here...

You will now build on the coding techniques introduced in this exercise to automate the Yum database.