Programming: VBA in MS Office – An Introduction

How to Use this User Guide

This handbook accompanies the taught sessions for the course. Each section contains a brief overview of a topic for your reference and then one or more exercises.

Exercises are arranged as follows:

- A title and brief overview of the tasks to be carried out;
- A numbered set of tasks, together with a brief description of each;
- A numbered set of detailed steps that will achieve each task.

Some exercises, particularly those within the same section, assume that you have completed earlier exercises. Your teacher will direct you to the location of files that are needed for the exercises. If you have any problems with the text or the exercises, please ask the teacher or one of the demonstrators for help.

This book includes plenty of exercise activities – more than can usually be completed during the hands-on sessions of the course. You should select some to try during the course, while the teacher and demonstrator(s) are around to guide you. Later, you may attend follow-up sessions at ITLP called Computer8, where you can continue work on the exercises, with some support from IT teachers. Other exercises are for you to try on your own, as a reminder or an extension of the work done during the course.

Text Conventions

A number of conventions are used to help you to be clear about what you need to do in each step of a task.

- In general, the word **press** indicates you need to press a key on the keyboard. **Click**, **choose** or **select** refer to using the mouse and clicking on items on the screen. If you have more than one mouse button, click usually refers to the left button unless stated otherwise.
- Names of keys on the keyboard, for example the Enter (or Return) key, are shown like this: ENTER.
- Multiple key names linked by a + (for example, CTRL+Z) indicate that the first key should be held down while the remaining keys are pressed; all keys can then be released together.
- Words and commands typed in by the user are shown `like this`.
- Labels and titles on the screen are shown like this.
- Drop-down menu options are indicated by the name of the options separated by a vertical bar, for example File|Print. In this example you need to select the option Print from the File menu or tab. To do this, click when the mouse pointer is on the File menu or tab name; move the pointer to Print; when Print is highlighted, click the mouse button again.
- A button to be clicked will look like this.
- The names of software packages are identified *like this*, and the names of files to be used like this.

Software Used

Excel 2010	*Access 2010*

Files Used

Areas.xlsm	Products.xlsm	SalesData.xlsm
Search.accdb	StationaryBusiness.accdb	

Revision Information

Version	Date	Author	Changes made
1.0	Feb 2015	Gavin Taylor	Created

Copyright

Contents

Exercises

1 Introduction

Welcome to the *Programming: VBA in MS Office* session!

This booklet accompanies the course delivered by Oxford University's IT Learning Programme. Although the exercises are clearly explained so that you can work through them independently, you will find that it will help if you also attend the taught session where you can get advice from the teacher, demonstrator(s) and even each other!

If at any time you are not clear about any aspect of the course, please make sure you ask your teacher or demonstrator for some help. If you are away from the class, you can get help by email from your teacher or from help@it.ox.ac.uk.

1.1. What You Should Already Know

No previous knowledge of VBA is expected. We will assume that you have some knowledge and/or experience of programming and the general principles of programming. This could be gained from other programming courses using different programming languages.

We will also assume that you are familiar with opening files from particular folders and saving them, perhaps with a different name, back to the same or a different folder.

The computer network in our teaching rooms may differ slightly from that which you are used to in your College or Department; if you are confused by the differences please ask for help from the teacher or demonstrator(s).

1.2. What Will You Learn?

This course will help you get started with using VBA in MS Office applications. It will show you the basics of writing VBA code and how the code can be used to make MS Office applications do some very powerful things.

In this session we will cover the following topics:

- Learn what VBA is and how it works
- Understand the syntax and technicalities of the VBA language
- Understand when to use VBA and when to use a simpler macro
- Create modules of code for use in Excel
- Create modules of code for use in Access
- Understand the differences between code in Excel and Access

These notes deal with using VBA in *Office 2010*, however the techniques learned should be able to be applied to future and previous versions of the software as the fundamentals of the language are unchanged between versions.

Getting to grips with VBA is not a quick process. This session will not teach you every function, method and piece of code that can be used. This session will, however, give you the ability and confidence to be able to find out answers to more complicated problems yourselves. With the basic knowledge gained from this course, you will be able to search the internet in a smarter way to get solutions to the more complex problems you may face with VBA.

1.3. Using Office 2010

If you have previously used another version of *Office*, you may find *Office 2010* looks rather unfamiliar. “Office 2010: What’s New” is a self-study guide covering the Ribbon, Quick Access Toolbar and so on. This can be downloaded from the ITLP Portfolio at http://portfolio.it.ox.ac.uk.

For anyone who prefers not to use the mouse to control software, or who finds a keyboard method more convenient, it is possible to control *Office 2010* applications without using a mouse. Pressing ALT once displays a white box with a letter or character next to each visible item on the Ribbon and title bar (shown in Figure 1).

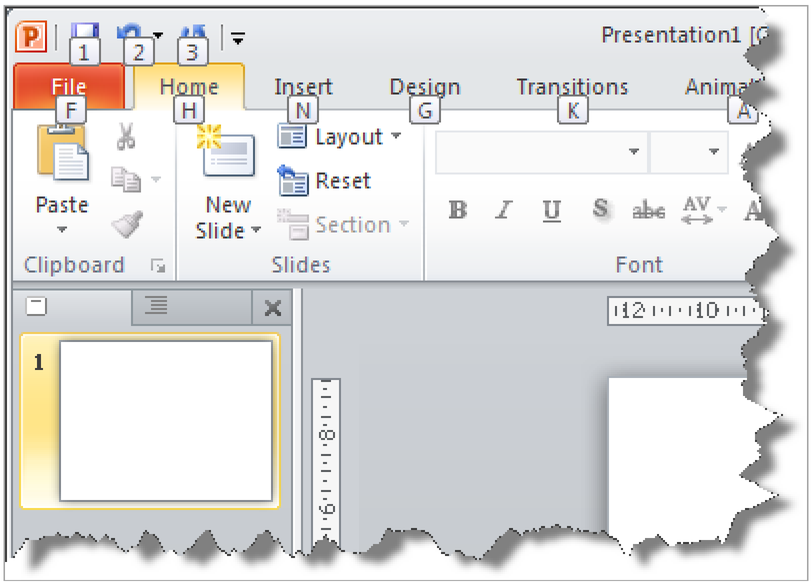


Figure 1 Keystrokes to Control Ribbon Tabs and Title Bar (Press ALT to Show These)

After you type one of the letters/characters shown, the relevant Ribbon tab or detail appears, with further letters/characters for operating the buttons and controls.

The elements of a dialog can be controlled, as usual with *Windows* applications, by using TAB to navigate between items or typing the underlined character shown beside an item.

1.4. What is VBA?

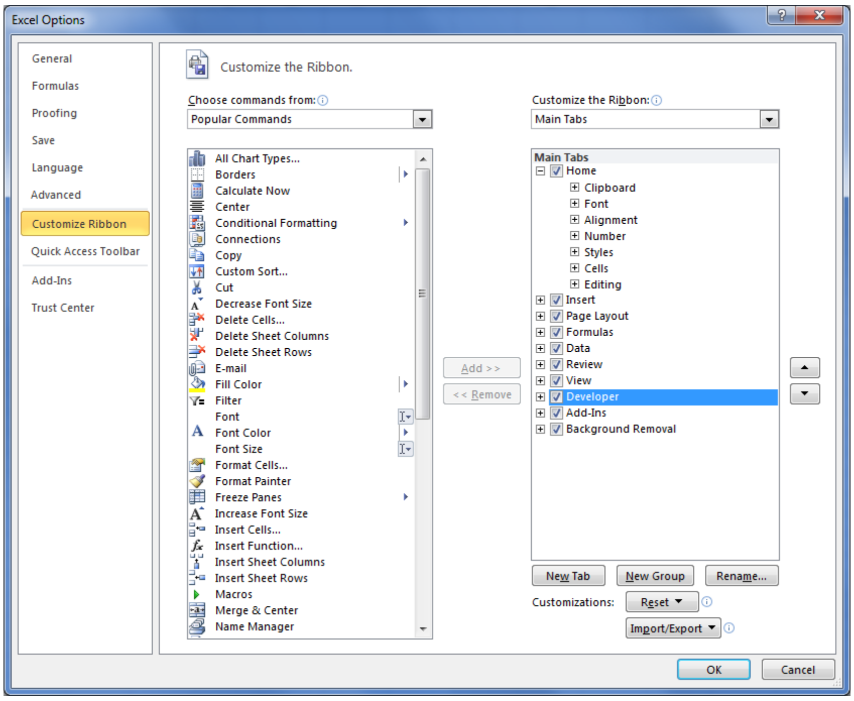
VBA is a high-level programming language that sits behind the Microsoft Office suite of applications. It is made available, through the built-in VBA Editor in each applicable application, to the end user to create code that can be executed within the user's application.

VBA can be used in a spreadsheet to carry out complicated tasks that may be too time-consuming or impossible for a user to do manually.
VBA can be used in Access to carry out tasks on data stored in tables, or run code when a button on a form is pressed. For example, you may have a form to enter data with a submit button. VBA could be used to create a piece of code that checks the entered data and stores it in multiple tables, perhaps making changes to the data first, based on assigned rules.
The possibilities for how VBA could be used are endless, and only limited by your inventiveness.

1.5. Where Can I Use VBA?

If you have a copy of *Microsoft Office*, then you already have the applications required to use VBA. This course will focus on using Excel and Access.

You may need to make the Developer tab visible in each application in order to access the VBA Editor. To do this, go to the File Menu, then select Options. On the screen that appears, go to the Customize Ribbon option, then make sure Developer is checked in the list on the right-hand side.



Excel Options

Customize the Ribbon.

Choose commands from: Popular Commands

Customize the Ribbon: Main Tabs

If you are a member of staff, you can obtain a copy of *Microsoft Office* from the IT Services on-line shop. Students can obtain a Microsoft Student Licence, but this must be bought through a Microsoft Authorised Education Reseller.

2 Using Macros

2.1. What are Macros

Macros are very useful tools when working in MS Office applications. They allow you to 'record' actions that you carry out (such as highlighting a range of cells and making them bold, or adding the sum of a column to the bottom of a table of data). Once you have finished recording your actions you can save the macro and run it whenever you want. When run, the macro will replicate the actions you took on whatever fields you specify.

Technically, macros are VBA functions that are automatically generated based on the inputs you record. The VBA is written for you and you are never expected to look at it.

2.2. Macros vs VBA

So, if macros are just VBA functions, why do we need to use VBA at all? Why not just create macros for everything we need?

In answer to that, macros are very useful at automating repetitive tasks that you would normally do manually. If what you are doing involves doing basic things, using standard controls in the ribbon and standard formulas, then macros are definitely the way to go.

However, if what you want to do cannot easily be done using the standard ribbon controls and standard formulas, then macros aren't going to be much help. That's when you want to write the VBA code yourself.

As a guide, if what you want to do is repetitive and easy to carry out manually, then record a macro for it. If what you want to do is more complex and can't easily be done within the standard user interface, then you need to use VBA.

3 Basic VBA Coding Principles

3.1. Accessing VBA

To start writing VBA code you need to open the VBA Editor. To do this, go to the `Developer` tab in your Office application and click on the `Visual Basic` button.

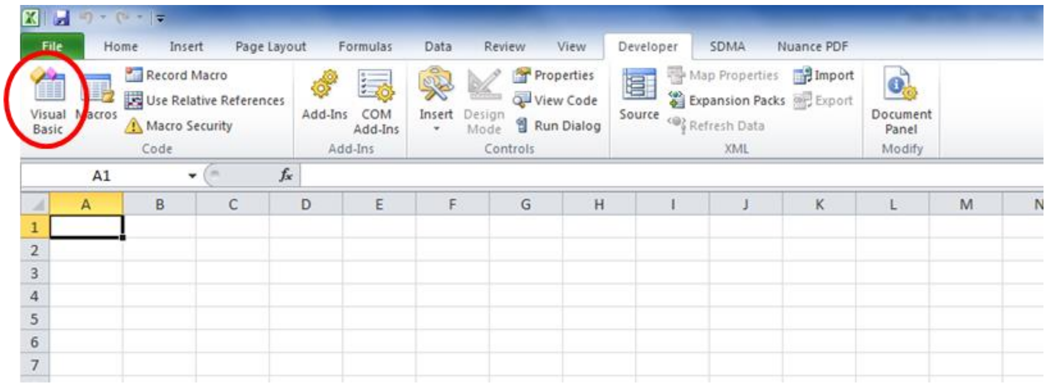


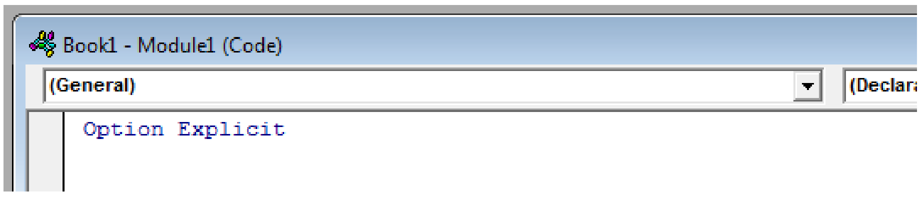
Figure 2 View of the Developer tab with the Visual Basic button

3.2. Coding Structure and Objects

3.2.1. Modules

Modules in VBA are the windows you write code in in the Visual Basic editor. Each window represents a module and will contain any number of functions and sub functions. Within a module it is also possible to declare global variables, which will be accessible by all functions within that module.

When starting a new module, it is a very good idea to add the following code to the top of the module:



```
Option Explicit
```

Figure 3 Top of a module with Option Explicit written in the code window

Typing `Option Explicit` at the top of the code window means that any variables you define within the code window must be declared with an object type. This ensures the computer can assign the correct amount of memory to each variable without having to guess at what it might be used for.

3.2.2. Functions and Sub Functions

Functions are the blocks of code that will be called within your application. They are what you will be writing when solving your problems with VBA.

Functions can be stand-alone methods that can be called from anywhere, or they could be linked to specific events, such as clicking on a button in an Access form.

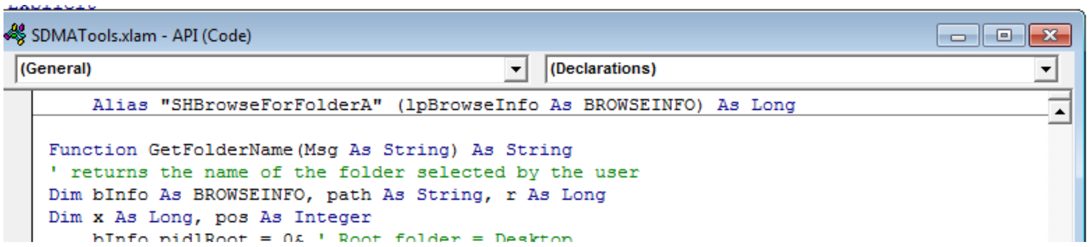
Functions start with a name and a list of parameters in brackets, followed by the object type that will be returned by the function (this may not always be needed if you don't want the function to return anything). An example function is written in VBA as follows:

```
Function ExampleFunctionName(ExampleParameter As Integer) As String
```

In the above example, the name of the function is `ExampleFunctionName`, and this will be used to call the function in other parts of the code and from elsewhere, such as a form in Access or as a cell formula in Excel.
The function also has a single parameter, in this case called `ExampleParameter`, which has been specified as an Integer. This means that wherever the function is called, the name of the function must always be followed by brackets with an integer, or a variable or object that is an integer, inside the brackets, for example:

```
ExampleFunctionName(12)
ExampleFunctionName(varNumber)          Where varNumber is an integer
```

Finally, when the function has finished executing, it is expected to return an output. This is specified by the use of `As String` at the end. In this case, the output should be of the String data type.


SDMATools.xlam - API (Code)
(General) (Declarations)

```
    Alias "SHBrowseForFolderA" (lpBrowseInfo As BROWSEINFO) As Long

Function GetFolderName(Msg As String) As String
' returns the name of the folder selected by the user
Dim bInfo As BROWSEINFO, path As String, r As Long
Dim x As Long, pos As Integer
    bInfo.pidlRoot = 0& ' Root folder = Desktop
```

Figure 4 Example of a function definition

In the example above, the function is defined with the name **`GetFolderName`**, it has one parameter (**`Msg`**, which should be a String variable), and will return the result of the function as a string variable.

In order for the function to return something (such as a string in the above example) you need to make sure you specify what that value to be returned is. This is done as follows:

```
FunctionName = ValueToReturn
```

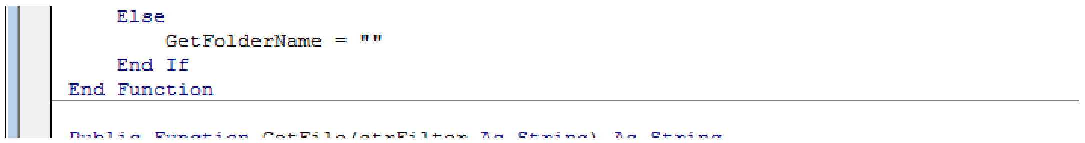
Using the example function `ExampleFunctionName`, as stated above, the function should return a string, so at some point within the function, the following line of code could be included to return the result "Finished":

```
ExampleFunctionName = "Finished"
```

Alternatively, we may have a variable within the function that holds a string, in which case we could return the contents of the variable:

```
ExampleFunctionName = varString
```

In the example below you will see that it says **`GetFolderName = ""`**. This means that the function would return `""` at the end, unless another line of code changed that before the function ended.



```
    Else
        GetFolderName = ""
    End If
End Function

Public Function GetFile(strFilter As String) As String
```

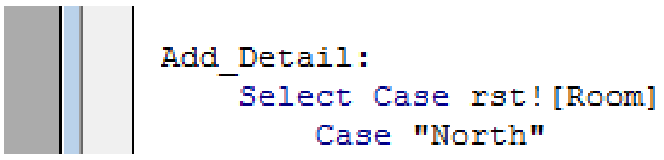
Figure 5 Example of the end of a function

At the end of the function, so that the application knows it has reached the end, you type **`End Function`**.

Sub functions are small chunks of code that you may want to run repeatedly at different points in your main function. Rather than writing the same code out every time you want to use it, you can write the code chunk as a sub-function within the main function and then reference it within the code.

Strictly speaking it is not really a function, more of a signpost telling the system to go somewhere new and execute code, then come back and continue where it left off.

When writing a sub-function you need to start by declaring that is a sub-function by writing the name, followed by a colon, as shown in the picture below:



```
Add_Detail:
    Select Case rst![Room]
        Case "North"
```

Figure 6 Example of declaring a sub-function

In this case, the sub-function is called **`Add_Detail`**. Once we have written the code for our sub-function we end it by writing **`Return`**, this tells the system to go back to where it was in the main code.

Finally, to call a sub-function at any time within the main function, we type **`GoSub`** followed by the name of the sub-function.

Be aware that sub-functions can only be referenced within the function they are written in.

3.2.3. Variables and Objects

Variables and objects are what will store the data for you as your code runs.

Variables are defined within code the following way:

```
Dim variableName as VariableType
```

If you have put **`Option Explicit`** at the top of your code window (highly recommended) then you must declare the variable type when declaring a variable. If you do not then the code will not compile and you won't be able to run your code.

There are a number of different data types that variable can be declared as, too many to list here. The following is a list of the most common ones but I suggest looking at the help pages in the VBA editor for a better idea of what is available:

```
String
Integer
Double
Array
Float
```

In Excel:

```
Worksheet
Workbook
```

In Access:

```
Database
Recordset
```

Once declared, variables can be assigned values as follows:

```
variableName = valueToBeAssigned
```

If you try to assign a value that is of a different data type to the variable then the code will fail, so make sure you assign values that match the data type of the variable. Some more examples of assigning values to variables are as follows:

```
varString = "Hello World"
varInteger = 34 + 56
varFloat = 56/100
```

Some variables can be declared as complex data types and so are more like objects. For example, in Access, you can declare a variable as a **`Recordset`**. This variable will act as a mini table and will hold lots of data of different types in a table format.

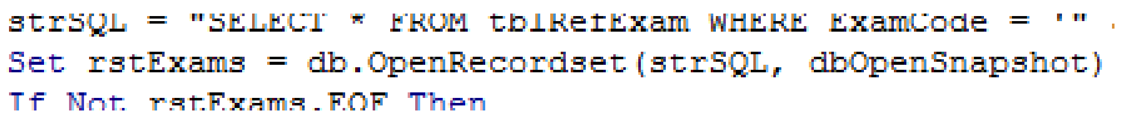
Objects (and some variables) have built-in methods that can be called on them to carry out certain tasks. For example, returning to the recordset data type again, there is a method called **`MoveNext`** that tells the recordset to move to the next row in its table. To call such methods you do the following:

```
objectName.methodName
```

Notice the full stop between the object name and the method name, this is what tells the system to run the built-in method.

Some methods require parameters to work. These are added as follows:

```
objectName.methodName(parameter1, parameter2)
```



```
strSQL = "SELECT * FROM tblRefExam WHERE ExamCode = '" 
Set rstExams = db.OpenRecordset(strSQL, dbOpenSnapshot)
If Not rstExams.EOF Then
```

Figure 7 Example of a Database object (db) and the method call OpenRecordset with two parameters

3.2.4. Coding Expressions and Statements

If Statements

If statements allow different code to be run depending on a condition. They are written in VBA as follows:

```
If Condition then
…
Else
…
End If
```

Note that the **`End If`** statement is very important, without it the code will not compile correctly.

`Else` is not mandatory and can be left out if needed. You can also use the statement **`ElseIf`**, which allows you to specify another condition (for example if a variable could have three values rather than two).

It is also possible to insert If statements inside other If statements, known as nested If statements, you just have to make sure you include the **`End If`** statement correctly for each If statement you create.

For example, look at the code block below:

```
Dim varInteger as Integer
Dim result as String
varInteger = 15 * 3
If varIntger < 30 then
```

```
        result = "Your calculation is less than 30"
Else
        result = "Your calculation is greater than or equal to 30"
End If
```

In this example, the If statement is used to determine whether the value of varInteger is less than 30. If it is then it stores the string "Your calculation is less than 30" in the result variable. Otherwise, it stores the string "Your calculation is greater than or equal to 30" in the result variable. At the top you can see that the value of varInteger is the result of the calculation 15*3, which equals 45, therefore the result variable will hold the string "Your calculation is greater than or equal to 30".

An example of a nested If statement can be seen below:

```
varInteger = CInt(Inputbox("Enter an integer"))
If varInteger > 0 then
        If varInteger < 10 then
                result = "Your number is between 1 and 9
        ElseIf varInteger < 20 then
                result = "Your number is between 10 and 19
        Else
                result = "Your number is 20 or more
        End If
Else
        result = "Your number is less than or equal to 0"
End If
```

You can see that the second If statement is only reached if the first If statement evaluates to True. We have also used the ElseIf statement to include another evaluation, in this case whether the variable is less than 20.
Another thing to notice about the code example above is the use of `Inputbox` and `CInt`. `Inputbox` is an in-built function that will display a box to the user where they can enter text when requested. `CInt` is another function that takes whatever is inside the brackets (in this case whatever the user enters) and turns it into an integer if possible.

Fill in the Blanks

Below are some code blocks involving If statements. Fill in the blanks to complete the code in each block:

1:

```
Dim varArea as Float
Dim returnString as _________
__________ = 10 * 20
If varArea > ____ Then
      returnString = "The area is over 100!!!"
Else
      resturnString = "The area is less than 100"

_________
```

2:

```
Dim userName as String
Dim outputString as _________
userName = ________("Please enter your name")
___ userName = "John" _____
      outputString = "Your name is the same as mine, " & userName
ElseIf __________ = "Sebastian" Then
      __________ = "I don't like your name, " & __________
______
      outputString = "Hello, " & userName
End If
```

3:

```
      _____ varLength __ ________
      _____ varWidth __ ________
      Dim resultArea as Integer
      Dim errorMessage as _______
      varLength = CInt(________("Please enter the length of the plot"))
      varWidth = ___(________(____________________________________))
      If varLength >= ______ _____
            If _________ >= _______ Then
                  resultArea = _________ * __________
            Else
                  errorMessage = "The width of the plot needs to be at least 5"
            End If
      ______
            ____________ = "The length of the plot needs to be at least 10"
      ______
```

Case Statements

Case statements should be used where If statements would be too complicated to use. For example, if the condition you need to check is a variable that could hold a large number of different options, then a Case statement should be used to specify what code to run in each of the cases.

Case statements are written as follows:

```
Select Case variableName
   Case [option1]
         …
   Case [option2]
         …
   Case [option3]
         …
   Case Else
         …
End Select
```

As with If statements, it is very important that you finish the Case statement with **`End Select`** otherwise the system will not compile your code.

An example of a Case statement can be seen below:

```
Select Case varString
      Case "Cheese"
            varCrisps = "Wotsits"
      Case "Pickled Onion"
            varCrisps = "Monster Munch"
      Case "Cool Original"
            varCrisps = "Doritos"
      Case Else
            varCrisps = "Walkers"
End Select
```

In the example above, the Case statement takes the value of the variable varString and compares it to each of the cases ("Cheese", "Pickled Onion", etc). If it matches one of these cases then it will run the code for that case (eg, if it matches

"Cheese", then it will set the value of varCrisps to "Wotsits"). If it doesn't match any of the cases then it will run the code in the Else block instead.

Loops

There are two types of loops that are used in VBA: For loops and While loops.

For loops should be used when you know how many times code should be looped through.

While loops should be used when you don't know how many times code should be looped through.

The idea of using a loop is to carry out the same bit of code repeatedly, usually changing what you are working on each time you go through the loop, such as moving to the next row in a set of data.

For loops are written in the form:

`For variableName = number to number`

…

`Next variableName`

The numbers could be specified, eg 1, 2, etc, or they could refer to other variables or method calls such as **`len(variable)`**.

Note that it is very important that you include the variable name next to **`Next`**. If you don't then the variable would never change and the loop would repeat infinitely, since the ending condition would never be met. This is what's called an infinite loop.

For example:

```
For varInteger = 1 to 10
      varResult = varResult + varInteger
Next varInteger
```

In the code above, the loop is set to run while the value of varInteger is between 1 and 10 (starting at 1). During each loop the value of varResult will be increased by the amount of varInteger. Once done, the loop will increment varInteger by 1 and start from the beginning. By the time the loop finishes running you should find that the value of varResult is the result of the sum 1+2+3+4+5+6+7+8+9+10.

While loops are similar but will continue looping until a condition is met. They are written as follows:

`Do While condition`

…

`Loop`

The condition must be an expression that will evaluate to true or false. If true, then the loop will run the code inside it. If false, then the loop will end.

It is very easy to create a while loop that results in an infinite loop as it does not enforce an increment of a value or a change to anything before looping again (unlike for loops). It is very important that you include within the code inside the loop, something that will eventually result in the loop condition evaluating to false.

An example of a While loop can be seen below:

```
varInteger = 1
Do While varInteger <= 10
      varResult = varRsult + varInteger
```

```
        varInteger = varInteger + 1
Loop
```

Notice in the example above that we have done exactly the same thing as in the For loop, you should find that the result is the same after the loop finishes running. However, notice that we have had to include the line `varInteger = varInteger + 1`, which increments the value of `varInteger` and so makes sure the loop does eventually end.

Fill in the Blanks

Below are some code blocks involving For and While loop statements. Fill in the blanks to complete the code in each block:

1:

```
For I = 1 to ______
        ' Loop through this code 12 times
        ' Sum the values of I
        sumOfI = sumOfI + __
_____ __
```

2:

```
I = 3
Do While I < _____
        'Loop through this code 12 times
        'Sum the values of I
        sumOfI = ______ + ___
        ___ = ___ + ___
______
```

3:

```
'Loop through code until Finished equals True
tmpCount = 1
Finished = False
Do _______ Finished ___ False
        If tmpCount = 14 _____
                Finished = True
        ________
        tmpCount = ________ + 2
_______
```

Note: Number 3 will result in an infinite loop and the code will never stop executing the loop. Can you explain why this is the case and what you could do to stop it happening?

With Statements

A With statement can be very useful, especially in Access, in avoiding excessive code typing and potential spelling errors.

A With statement allows you to specify a variable, or more commonly, an object, and then call methods and reference parts of the object without having to write out the object's name in full every time.

They are written as follows:

```
With objectName
   …
End With
```

With this in place, you don't need to write the **`objectName`** when calling methods on it or referencing data within it. For example, instead of writing **`objectName.MoveNext`** for a recodset you can write **`.MoveNext`** within the With statement.

This saves on typing and the potential for human error. One thing to note is that they cannot be nested as the system wouldn't know which object you are referring to.

For example, in Access:

```
Set rst = db.OpenRecordset("Table1", dbOpenSnapshot)
With rst
       .MoveFirst
       tmpID = !ID
       tmpName = !Name
       .MoveNext
End With
```

In the code block above, the variable rst is a recordset object that contains the contents of Table1 in a database. By using the With statement we avoid having to write rst in every line. Without using the With statement the code would look as follows:

```
Set rst = db.OpenRecordset("Table1", dbOpenSnapshot)
rst.MoveFirst
tmpID = rst!ID
tmpName = rst!Name
rst.MoveNext
```

3.3. Formatting Code – Best Practice

3.3.1. Naming Conventions

When naming functions and variables in VBA there are only a few limitations on what you can write. Not using spaces or some special characters are the main ones. However, it is a very good idea to get into the habit of using a consistent naming convention.

A good convention to use is as follows:

- All names should have meaning. Variables should be named based on what they hold; functions should be named based on what they do
- All names should be of a reasonably short length. Very long names make typing difficult, while very short names make it difficult to follow the first convention
- When writing a function name, make sure there are no spaces and capitalise the first letter of each word in the name, eg, **`GetFolderName`**
- When writing a variable name, make sure there are no spaces and capitalise the first letter of each word EXCEPT for the first word, eg, **`varUploadFile`**
- When naming a constant, use all caps, eg, **`ACADYEAR`**
- When naming variables it can be helpful to include the data type in the name, eg, **`strName`**, **`rstStudents`**, **`intScore`**

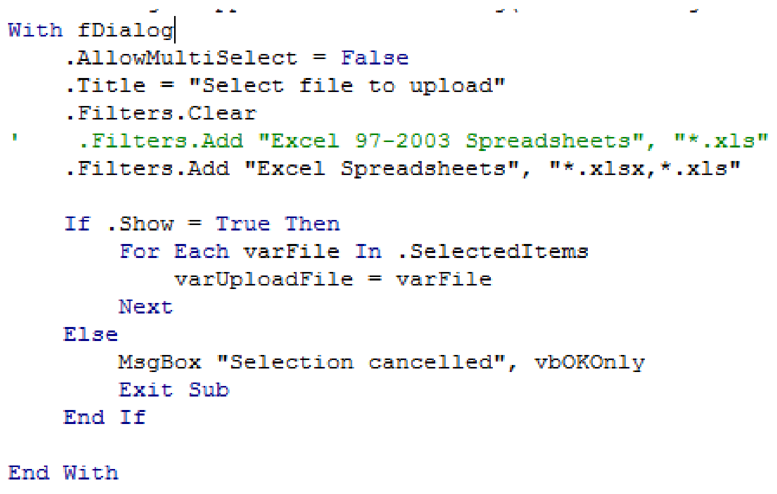
Following these conventions will give you consistency in your code and greatly improve readability for yourself and other people looking at your code.

3.3.2. Indenting Code

When writing your code in any language, but particularly with VBA, it is a very good idea to indent your code. This means having sections of code (such as the code being executed within an If statement) indented from the If statement itself.

The benefit of indenting code is that it makes it much easier to read for others; it makes it easier for you to follow as you are writing your code; and it makes it easier to see where you may have missed important statements such as **`End If`**.

The picture below shows code with multiple indentations. Firstly, the code within the With statement has been indented, then the code within the If statement has been further indented, while the code within the For loop has been indented even further.



```
With fDialog
    .AllowMultiSelect = False
    .Title = "Select file to upload"
    .Filters.Clear
'    .Filters.Add "Excel 97-2003 Spreadsheets", "*.xls"
    .Filters.Add "Excel Spreadsheets", "*.xlsx,*.xls"

    If .Show = True Then
        For Each varFile In .SelectedItems
            varUploadFile = varFile
        Next
    Else
        MsgBox "Selection cancelled", vbOKOnly
        Exit Sub
    End If

End With
```

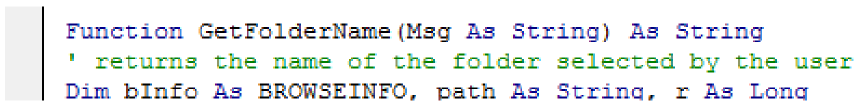
Figure 8 Code showing good use of indentation

Using indentation will make your code much easier to understand and work with. Your fellow coders will thank you for it.

3.3.3. Commenting Code

Regardless of which programming language you are using, you should always comment your code. Commenting makes it easy for other people to see what is going on in your code and can even help you if you have to return to code long after writing it, when you have forgotten what you did.

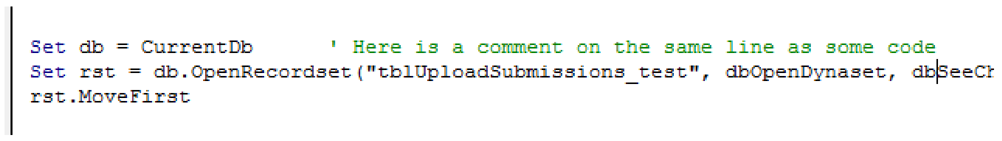
Commenting in VBA is done by inserting an apostrophe at the start of the line. Doing this will then turn the whole line green in the editor, which tells you that it is now a comment and will be ignored by the system.



```
Function GetFolderName(Msg As String) As String
' returns the name of the folder selected by the user
Dim bInfo As BROWSEINFO, path As String, r As Long
```

Figure 9 Example of a commented out line

In addition to commenting out entire lines, you can also type an apostrophe after the code on a line and then add a comment on the same line as the code.



```
Set db = CurrentDb      ' Here is a comment on the same line as some code
Set rst = db.OpenRecordset("tblUploadSubmissions_test", dbOpenDynaset, dbSeeCh
rst.MoveFirst
```

Figure 10 Example of an in-line comment

Commenting code is a balancing act between providing enough detail and not filling the screen with green text. A good rule of thumb is to comment every 5 lines or so or when something different is happening in the code. If you are doing the same thing repeatedly in the code then you don't need to comment every one, just comment the block.

Don't be intimidated by commenting, it is difficult to get it just right but will get easier with practice.

3.4. Checking Code Works and Handling Errors

3.4.1. Compiling Your Code

In order for the application to be able to run your code it needs to be compiled.

To compile your code select the `Compile` option from the `Debug` menu in the VBA Editor window. The application will then compile your code.

At this point, if you have made any obvious syntax errors or logic errors (such as forgetting to end an If statement) the application will alert you to this and stop compiling. You will not be able to compile your code until you resolve the error and recompile.

Please note that even if your code compiles, doesn't mean that it will work when run. The compiler just checks that the code makes sense to the computer, it doesn't check to make sure it will do what you want it to.

3.4.2. Error Handling

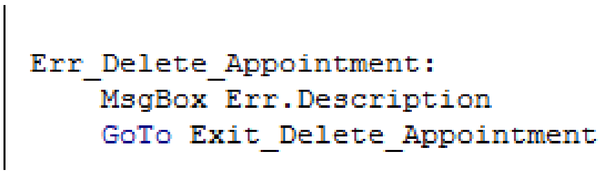
Error handling is very important in any programming you do. Having good error handling means you can easily identify what has gone wrong when your code fails (and be assured, you will get lots of failures before you get to your final working program).

To handle errors in VBA you need to write the following line of code:

```
On Error GoTo ErrorName
```

This tells the system to go to the location with the name **`ErrorName`** whenever the code errors.

At the bottom of your function you will then have a block of code for the error handling that will look a bit like the screenshot below.



```
Err_Delete_Appointment:
    MsgBox Err.Description
    GoTo Exit_Delete_Appointment
```

Figure 11 Example of error handling code block

You can execute any code you like within the error handler, allowing you to reverse any changes that may have been made, for example, before ending the function. As a good rule of thumb, you should at the very least include the line **`MsgBox Err.Description`**. This will display a pop-up box with a system level description of the error. They are often not very informative but they are better than nothing at all.

As another rule of thumb you should always put your error handler declaration (the `On Error` statement) just after the function declaration. Error handlers are local to the function and can't be accessed outside of the function they are created in.

It is also possible to have multiple error handlers within your function. All you have to do is write new declaration statements; give each new error handler a different name; and create the error handler code block at the end of the function.

Although you can have multiple error handlers in a function, the system will only recognise one at a time. Each time you declare a new one it will ignore the previous one. This can be useful to help you understand where in your code the failure occurs.

4 Using VBA in Excel

Before looking at some examples of VBA being used in Excel, we need to look at how data is stored in Excel and how VBA will manipulate it.

4.1. Excel Hierarchy

In Excel you work with workbooks, that contain worksheets, that are made up of rows and columns of cells that contain data.

In VBA you use a hierarchy to refer to these different elements. The hierarchy is achieved using the same method as when calling methods on objects, namely the full stop.

The basic hierarchy in VBA is as follows:

```
Application.Workbooks(1).Sheets(1).Cells(1,1).Value
```

The 1s refer to the fact that there could be any number of the particular object and so, when writing code, you would need to specify which one you were working with inside the brackets, for example if you wanted to reference worksheet 2 then you would write `Sheets(2)`.

As well as cells, worksheets can also contain ranges (eg, A1:B35), rows and columns.

4.2. Attributes

Each of the objects within the hierarchy have attributes associated with them that can be changed, for example the name of a worksheet. We won't list all of the attributes for all of the object types but we will look at the some of the attributes of cells as these will probably be the most commonly used.

The attributes that you will likely use most are as follows:

```
.Value = newValue
.Formula
.Font.Name
.Font.Size
.Font.Bold = True/False
.Font.Italic = True/False
.NumberFormat = format, e.g "@" or "0.00"
.Borders.LineStyle
.Borders.Weight
.Interior.Color
```

As you can see, the names are quite self explanatory. By referencing any of these attributes you can retrieve it (such as retrieving the value of a cell) or change it (such as changing the font size of a cell).

There are many other attributes to be retrieved or set. When writing code in the VBA editor you will find that when you type the name of an object followed by the full stop, as long as the editor knows what type of object it is, it will display a list of things you can type (attributes and methods). It is a good idea to spend a bit of time browsing this list so you can start to learn exactly what you can do with each object type.

4.3. Some Examples of VBA Code

Below are various examples of VBA code used in Excel.

4.3.1. Creating a New Workbook

To create a new workbook:

```
Set wbNewWorkbook = Application.Workbooks.Add(strTemplate)
```

You can then use that object to refer to the new workbook:

```
wbNewWorkbook.Worksheets(WorkSheetNameOrIndex)
```

4.3.2. Iterating Through Worksheets

To iterate through the sheets in the current workbook:

```
For each s in workbook(n).sheet
  do something with s
Next
```

4.3.3. Changing Rows and Columns

To delete columns or rows:

```
Rows(6).Delete Shift:=xlUp

Columns("H:H").Delete Shift:=xlToLeft
```

To change the format of a row:

```
Rows(15).NumberFormat = "0.00"
```

To insert rows into the current worksheet:

```
Rows(11).Insert Shift:=xlDown,CopyOrigin:=xlFormatFromLeftOrAbove
```

4.3.4. Example Functions

This function takes an input string (NotIn), checks whether it is one character long and, if it is, checks whether it contains one of the list of allowed characters (C,I,K,M,O,V). If it does contain one of those characters then the function returns False, otherwise it will return True.

```
Function NotAllowed(NotIn As String) As Boolean
Dim Allowed(6) As String
Dim x As Integer

Allowed(1) = "C"
Allowed(2) = "I"
Allowed(3) = "K"
Allowed(4) = "M"
Allowed(5) = "O"
Allowed(6) = "V"
NotAllowed = True

If Not Len(NotIn) <> 1 Then
      For x = 1 To 6
            If Allowed(x) = NotIn Then
                  NotAllowed = False
            End If
      Next x
End If
End Function
```

This next function explains what it does in the comment below the function declaration, can you understand how it works?

```
Function GetUniqueColumnValues(rColumn As Range)
'This function goes through the range (it should be a single column and
MUST include a column header)
'and outputs all unique values to a new worksheet

Dim iUniqueValues As Long
Dim i As Long
Dim iVal As Variant
Dim iValCount As Long
Dim aOutput() As Variant
Dim sColumnHeader As String

sColumnHeader = rColumn.Cells(1, 1)

' now resize the input range to exclude the header
Set rColumn = rColumn.offset(1, 0).Resize(rColumn.Rows.Count - 1, 1)
iUniqueValues = CountUniqueValues(rColumn, True, True)

ReDim aOutput(iUniqueValues - 1, 2)

' Go through the unique values and count the instances
For i = 1 To iUniqueValues
      iVal = UniqueItem(rColumn, i)
      If iVal = "" Then
          iValCount = WorksheetFunction.CountBlank(rColumn) +
          WorksheetFunction.CountIf(rColumn, " ")
      Else
         iValCount = WorksheetFunction.CountIf(rColumn, iVal)
      End If

      aOutput(i - 1, 0) = sColumnHeader
      aOutput(i - 1, 1) = IIf(iVal = "" Or iVal = " ", "<blank>", iVal)
      aOutput(i - 1, 2) = iValCount
Next  'i

GetUniqueColumnValues = aOutput
End Function
```

4.4. Building Your First Function

Now that you have looked at the examples of VBA code above and have an idea of what VBA can do in Excel, try the exercises below, where you will be writing your own VBA functions and running them.

Exercise 1 – Building your First Function (and Running it)

See Exercise 1 in Appendix 2.

Exercise 2 – Amending Someone Else's Code

See Exercise 2 in Appendix 2.

5 Using VBA in Access

5.1. How is it Different to Excel?

For the most part VBA works in the same way in Access as it does in Excel, so what you have done in Excel will be transferable to Access. There are some differences though, mainly in the way you work with data.

5.1.1. Using Recordsets and Working With Tables

In Excel, your VBA code works with spreadsheets, looping through rows and columns, dealing with whatever data is stored in those cells.

In Access, because it is a database, data is stored in tables that may be linked together by relationships.

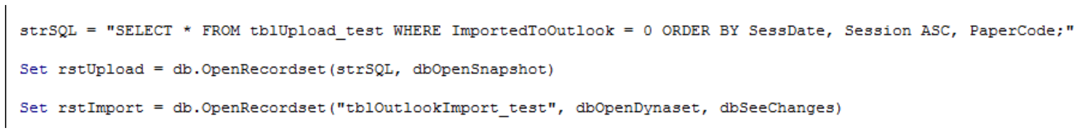
Although tables look very much like Excel spreadsheets, they can be worked with differently because you can reference particular column names, rather than column A or B, etc.

The main way to work with data in Access is through the use of recordsets. Recordsets are VBA objects that act as a temporary table. They can store any data in a tabular format as long as what you are putting in is in a tabular format.

For example, you can populate a recordset with the entire contents of a table and then loop through the records. You could write a query that pulls out certain records in a table, or several related tables, and then put the results of the query in to a recordset and work with that data.

Recordset objects can be populated with a snapshot of data, which means the data is what was retrieved at that point in time and won't reflect any changes that may occur to the data in the mean time. Alternatively, you can create a recordset that will keep track of any changes made to the data and display them as they occur. This can be useful if you need to update records within your code. There are other types of recordset although we won't go through them here.

Examples of recordset objects being populated with data can be seen below:



```
strSQL = "SELECT * FROM tblUpload_test WHERE ImportedToOutlook = 0 ORDER BY SessDate, Session ASC, PaperCode;"

Set rstUpload = db.OpenRecordset(strSQL, dbOpenSnapshot)

Set rstImport = db.OpenRecordset("tblOutlookImport_test", dbOpenDynaset, dbSeeChanges)
```

Figure 12 Examples of recordsets being populated with data

In the example above you can see that the recordset **`rstUpload`** has been populated with the query found in the variable **`strSQL`**. This is something that is also possible. You can write your query in a string variable and then use that when populating the recordset. It has also been set as a snapshot using the parameter **`dbOpenSnapshot`**.

The second recordset, **`rstImport`**, has been populated with the entire contents of the table **`tblOutlookImport_test`**. It has also been declared as a **`dynaset`**, which means it is possible to update the recordset (add, remove and edit records), using the parameter **`dbOpenDynaset`**. It also has the parameter **`dbSeeChanges`** which means any changes made in the recordset will be immediately visible.

5.1.2. SQL

Another important part of Access is the use of Structured Query Language (SQL) for creating queries that can look at data in several tables that are linked by relationships between them.

If you have not used SQL before and plan to use VBA in Access it is a good idea to learn the basics of writing SQL statements as it will prove invaluable when writing your VBA code.

We will not explain SQL here as it would take too long. Just be aware that it is possible to use SQL within VBA (especially in Access) and it will be very useful to know how to write your own queries.

5.2. Creating VBA Code in Access

Now that we have described the big differences you will find when using VBA in Access, here are some examples of functions and event handlers written in VBA. All of the examples are working pieces of code that are helping University staff members carry out their work on a day-to-day basis.

5.2.1. Creating Functions

This first example of a VBA function in Access is used in a database that connects to a shared calendar in MS Outlook in order to add/remove appointments in the calendar.

This particular function deletes a selected appointment from the shared calendar.

```
Private Function Delete_Appointment(argStartDate As Date, argEndDate As
Date, argSubject As String)
On Error GoTo Err_Delete_Appointment
Dim db As Database
Dim rst As Recordset
Dim strSQL As String

' Set up Outlook Objects.
Dim ol As New Outlook.Application
Dim olns As Outlook.Namespace
Dim cf As Outlook.MAPIFolder
Dim cfFolder As Outlook.MAPIFolder
Dim c As Outlook.AppointmentItem
Dim Prop As Outlook.UserProperty
Dim myRecipient As Outlook.Recipient
Dim sFilter As String

Set db = CurrentDb

If argSubject = "Exam Schedule" Then
      If varTestCalendar = True Then
            strSQL = "SELECT * FROM tblOutlookImport_Archive_test WHERE
Calendar = 'Test' AND Start = #" & CStr(Format(argStartDate, "mm/dd/yyyy
hh:mm:ss")) & "# AND End = #" & CStr(Format(argEndDate, "mm/dd/yyyy
hh:mm:ss")) & "# AND Subject = '" & argSubject & "' ORDER BY ImportID
DESC;"
            Set rst = db.OpenRecordset(strSQL, dbOpenSnapshot)
      Else
            strSQL = "SELECT * FROM tblOutlookImport_Archive_test WHERE
Calendar = 'Live' AND Start = #" & CStr(Format(argStartDate, "mm/dd/yyyy
hh:mm:ss")) & "# AND End = #" & CStr(Format(argEndDate, "mm/dd/yyyy
hh:mm:ss")) & "# AND Subject = '" & argSubject & "' ORDER BY ImportID
DESC;"
```

```
            Set rst = db.OpenRecordset(strSQL, dbOpenSnapshot)
      End If
Else
      If varTestCalendar = True Then
            strSQL = "SELECT * FROM
tblOutlookImportSubmissions_Archive_test WHERE Calendar = 'Test' AND Start
= #" & CStr(Format(argStartDate, "mm/dd/yyyy")) & "# AND End = #" &
CStr(Format(argEndDate, "mm/dd/yyyy")) & "# AND Subject = '" & argSubject
& "' ORDER BY ImportID DESC;"
            Set rst = db.OpenRecordset(strSQL, dbOpenSnapshot)
      Else
            strSQL = "SELECT * FROM
tblOutlookImportSubmissions_Archive_test WHERE Calendar = 'Live' AND Start
= #" & CStr(Format(argStartDate, "mm/dd/yyyy")) & "# AND End = #" &
CStr(Format(argEndDate, "mm/dd/yyyy")) & "# AND Subject = '" & argSubject
& "' ORDER BY ImportID DESC;"
            Set rst = db.OpenRecordset(strSQL, dbOpenSnapshot)
      End If
End If

Set olns = ol.GetNamespace("MAPI")

Set cf = olns.GetDefaultFolder(olFolderCalendar)
Set cfFolder = cf.Folders("Exams Form Test")

With rst
      If Not .EOF Then
            .MoveFirst
            sFilter = "[Mileage] = " & !ImportID & " AND [Subject] = " &
!Subject & ""
            Set c = cfFolder.Items.Find(sFilter)
            If Not c Is Nothing Then
                  c.Delete
            End If
      End If
End With

Exit_Delete_Appointment:
On Error Resume Next
      Exit Function

Err_Delete_Appointment:
      MsgBox Err.Description
      GoTo Exit_Delete_Appointment

End Function
```

Don’t worry if it doesn’t make sense right now (particularly the parts that deal with connecting to Outlook). Focus on recognising the parts of VBA code, such as the function declarations, the error handlers, the comments, what variables are used, etc. As you use VBA yourself, the more you will understand.

This next function is very long and quite complicated. What it does is allocate exam candidates to desks in the Extra Time room. Candidates sitting papers in the Extra Time room are seated so that the candidates with the shortest exam duration are sat closest to the door (in this case, desk 1 is closest to the door, with the rest being allocated sequentially).

The function sorts the candidates by duration (within the SQL statement that creates the recordset of candidates) and then allocates the next available desk, unless they have already been allocated a desk (some candidates sit two papers in a single session so we wouldn't want to allocate them to two different desks).

```
Private Function AllocateETDesks(argSessionID As Integer)
On Error GoTo Err_AllocateETDesks
Dim db As Database
Dim rstWPSession As Recordset
Dim rstCandidates As Recordset
Dim rstWPCandidates As Recordset
Dim rstWPCandPaper As Recordset
Dim strSql As String
Dim tmpSessionDate As Date
Dim tmpSession As String
Dim tmpID As Integer
Dim strSQL4 As String
Dim rstCandInfo As Recordset
Dim rstLinkTable As Recordset
Dim tmpCounter As Integer
Dim CandDict As Dictionary
Dim DuplicateCand As Boolean
Dim candNumber As Long

Set db = CurrentDb

'Create the dictionary
Set CandDict = Nothing
Set CandDict = New Dictionary

'Set up the counter that sets the desk number. Starts at 1 and increments
after each allocation of a desk.
tmpCounter = 1

DoCmd.Hourglass True

'Start by getting session record for updating
strSql = "SELECT * FROM tblWPSessionSummary WHERE ID = " & argSessionID &
";"
Set rstWPSession = db.OpenRecordset(strSql, dbOpenSnapshot)
rstWPSession.MoveFirst

'Record session date and session type (AM or PM) in temp variables for use
in retrieving candidates
tmpSessionDate = rstWPSession!SessionDate
tmpSession = rstWPSession!Session

'Close recordset as we are finished with it now
rstWPSession.Close
Set rstWPSession = Nothing

'Open recordset on WP candidates table to create records there, it is
going to be used for WP and ET candidates, so there ;)
Set rstWPCandidates = db.OpenRecordset("tblWPCandidates", dbOpenDynaset,
dbSeeChanges)

'OPen recordset on WPCandPaper table to create records there as well,
still using it for both ET and WP, so ignore the poor naming convention
Set rstWPCandPaper = db.OpenRecordset("tblWPCandPaper", dbOpenDynaset,
dbSeeChanges)

'Now, get the ET candidates that need to be allocated to desks
strSql = "SELECT * FROM tblPapers WHERE PaperDate = #" &
```

```
Format(tmpSessionDate, "mm/dd/yyyy") & "# AND PaperSession = '" &
tmpSession & "' AND CandLoc LIKE 'Spec*' ORDER BY PaperFinish ASC,
PaperCode;"
Set rstCandidates = db.OpenRecordset(strSql, dbOpenSnapshot)

'Check whether there are any ET candidates for the session. If not, move
on to next session without doing anything
If rstCandidates.RecordCount = 0 Then
    'No ET candidates, so close open recordsets and go to exit
    rstCandidates.Close
    Set rstCandidates = Nothing
    rstWPCandPaper.Close
    Set rstWPCandPaper = Nothing
    rstWPCandidates.Close
    Set rstWPCandidates = Nothing
    db.Close
    Set db = Nothing
    GoTo Exit_AllocateETDesks
End If

rstCandidates.MoveFirst

'loop through ET candidates and allocate desks
With rstCandidates

'Open link table recordset to record session-candidate links
Set rstLinkTable = db.OpenRecordset("tblETSessionDeskCandidateLink",
dbOpenDynaset, dbSeeChanges)

Do While Not .EOF
    candNumber = !CandNo
    'First, check the dictionary to make sure candidate isn't doing
another paper during the same session
    DuplicateCand = CandDict.Exists(candNumber)
    If DuplicateCand Then
        'Candidate exists in the dictionary, so we only need to create a
WPCandPaper record and link it to the correct WPCandidate record (using
the dictionary)
        rstWPCandPaper.AddNew
        rstWPCandPaper!WPCandidateID = CandDict.Item(candNumber)
        rstWPCandPaper!PaperCode = !PaperCode
        rstWPCandPaper!PaperTitle = !PaperTitle
        rstWPCandPaper!FinishTime = !PaperFinish
        rstWPCandPaper!PaperDuration = !PaperDuration
        rstWPCandPaper!ExamCode = !ExamCode
        rstWPCandPaper!IDtblPapers = !PaperID
        rstWPCandPaper.Update
    Else
        'New candidate so create a record in the WPCandidates table and
retrieve the ID for it
        'Note, left out CandName and AddComments as these are found in
another table. Will run a mass update at the end to add these details to
all records in table
        rstWPCandidates.AddNew
        rstWPCandidates!CandNo = !CandNo
        'rstWPCandidates!PaperCode = !PaperCode
        'rstWPCandidates!PaperTitle = !PaperTitle
        'rstWPCandidates!PaperDuration = !PaperDuration
        'rstWPCandidates!FinishTime = !PaperFinish
        'rstWPCandidates!ExamCode = !ExamCode
        'Actually, let's pull those extra details now and close the
recordset afterwards
        strSQL4 = "SELECT * FROM tblCandidates WHERE CandNo = " & !CandNo
& ";"
        Set rstCandInfo = db.OpenRecordset(strSQL4, dbOpenSnapshot)
        rstCandInfo.MoveFirst
        rstWPCandidates!CandName = rstCandInfo!CandName
        rstWPCandidates!AddComments = rstCandInfo!Comments
        'rstWPCandidates!IDtblPapers = !PaperID

        tmpID = rstWPCandidates!ID
        rstWPCandidates.Update

        'Next, add the candidate to the dictionary along with the ID
number for checking later in the loop
        CandDict.Add candNumber, tmpID

        'Now, create WPCandPaper record and link it to the newly created
WPCandidate record
        rstWPCandPaper.AddNew
        rstWPCandPaper!WPCandidateID = tmpID
        rstWPCandPaper!PaperCode = !PaperCode
        rstWPCandPaper!PaperTitle = !PaperTitle
        rstWPCandPaper!FinishTime = !PaperFinish
        rstWPCandPaper!PaperDuration = !PaperDuration
        rstWPCandPaper!ExamCode = !ExamCode
        rstWPCandPaper!IDtblPapers = !PaperID
        rstWPCandPaper.Update

        rstCandInfo.Close
        Set rstCandInfo = Nothing
    End If

    'If cnadidate is already doing a paper this session then there is no
need to create another desk allocation so skip this step
    If Not DuplicateCand Then
        'Add record in the link table for use with the mail merge query,
this is all we have to do for allocation
        rstLinkTable.AddNew
        rstLinkTable!SessionID = argSessionID
        rstLinkTable!deskNumber = tmpCounter
        rstLinkTable!CandidateID = tmpID
        rstLinkTable.Update

        'Increment counter and move on to next candidate
        tmpCounter = tmpCounter + 1
    End If

    candNumber = 0
    .MoveNext
Loop
End With

'Close recordsets and database
rstLinkTable.Close
Set rstLinkTable = Nothing
rstWPCandPaper.Close
Set rstWPCandPaper = Nothing
rstWPCandidates.Close
Set rstWPCandidates = Nothing
rstCandidates.Close
```

```
Set rstCandidates = Nothing
db.Close
Set db = Nothing

Exit_AllocateETDesks:
On Error Resume Next
    DoCmd.Hourglass False
    Exit Function

Err_AllocateETDesks:
    MsgBox Err.Description
    GoTo Exit_AllocateETDesks

End Function
```

As with the first example function, don't worry if a lot of this doesn't make sense right now. Focus on picking out the bits that do make sense, like variables and objects, loops, if statements, error handlers, etc.

5.2.2. Creating Code for Events

Events in Access are things that happen. For example, an event could be the user clicking on a button on a form, or the user typing in to a text field, or ticking a check box. It could also be when a form opens or closes. Almost anything that "happens" in Access is considered an event and can have VBA code associated with it.

All of the examples below are code blocks that are linked to events and will execute every time that event occurs.

Event handlers are usually declared as `Private Sub` (As you will see in the example code blocks below) and tend to include the event in the name of the routine (for example `_Click` in the example below).

This first example is a very simple event handler that closes the current form when the Close button is pressed.

```
Private Sub CloseForm_Click()
On Error GoTo Err_CloseForm_Click

    DoCmd.Close

Exit_ CloseForm_Click:
    Exit Sub

Err_CloseForm_Click:
    MsgBox Err.Description
    Resume Exit_CloseForm_Click

End Sub
```

This next example runs when the `Upload SC05` button is pressed on a form in the database. It allows the user to select a spreadsheet and upload it in to a temporary table in the database.

```
Private Sub UploadSC05_Click()
On Error GoTo UploadSC05_Click_err
Dim varFile As Variant
Dim varUploadFile As Variant
Dim fDialog As Object

Set fDialog = Application.FileDialog(3)
With fDialog
    .AllowMultiSelect = False
    .Title = "Select file to upload"
    .Filters.Clear
    .Filters.Add "Excel Spreadsheets", "*.xlsx,*.xls"

    If .Show = True Then
        For Each varFile In .SelectedItems
            varUploadFile = varFile
        Next
    Else
        MsgBox "Selection cancelled", vbOKOnly
        Exit Sub
    End If

End With

DoCmd.Hourglass True

'Import spreadsheet into tblImport
DoCmd.TransferSpreadsheet acImport, , "tblImport", varUploadFile, True

MsgBox "Process Complete", vbOKOnly

UploadSC05_Click_Exit:
    DoCmd.Hourglass False
    Exit Sub

UploadSC05_Click_err:
    MsgBox Err.Description
    GoTo UploadSC05_Click_Exit

End Sub
```

Exercise 3 - Making a Database React to User Input

See exercise 3 in Appendix 3.

Appendix 1: Further Study

There are two courses on Lynda.com that look at using VBA. One course focuses on VBA in Excel, while the other focuses on VBA in Access. I highly recommend working through these courses to build on the basics covered in this course.

Course: Up and Running with VBA in Access

http://www.lynda.com/Access-2003-tutorials/Up-and-Running-with-VBA-in-Access/87007-2.html?org=ox.ac.uk

Course: Up and Running with VBA in Excel

http://www.lynda.com/Excel-tutorials/Up-Running-VBA-Excel/62906-2.html?org=ox.ac.uk

Although both courses are graded as advanced, they should be manageable with the basics introduced during this course.

Appendix 2: Student Exercises

Exercise 1 Building Your First Function (and Running It)

- *Open the spreadsheet Areas.xlsm*
- *Add a module for our new function*
- *Write a function that will calculate the area of rectangle*

Task 1 Start *Excel* and open *Areas.xlsm*	**Step 1** Open *Excel*
	Step 2 Open the file *Areas.xlsm*
	Step 3 If a security warning appears just below the ribbon, open the message and make sure you enable macros.
Task 2 Go to the VBA Editor and create a new module	**Step 1** Go to the Developer tab and click on the Visual Basic button. If the tab isn't visible follow the instructions on page 3 of this handbook to make it appear.
	Step 2 In the VBA Editor, go to the Insert menu and choose Module

Task 3

Create a function called **`CalcArea`** that takes two parameters, **`length`** and **`width`**, with suitable variable types.

The function should then return the area of the rectangle based on the **`length`** and **`width`** parameters. The area is calculated as:

Area = length * width

Once you have created your function, compile and save it, then return to the spreadsheet and try running it.

To run it go into the empty cell C2 and type:

```
=CalcArea(A2,B2)
```

This should calculate the area based on the values in row 2 of the spreadsheet. You should find it returns the answer as 360.

Notes:

Make sure to specify **`Option Explicit`** at the top of the module.

Although the values in the spreadsheet are all integers you may want to consider making sure your function will work with decimals as well.

If you are struggling to create your function, here is the declaration you should use:

```
Function CalcArea(length As Double, width As Double)
```

Task 4

If you complete your function and it works try this next task.

Rather than supply your function with the length and width parameters we want the function to calculate the area for each row in the table with a single function call.

Amend and run your function so that it calculates and returns the area for each row in the table. You should only call the function once, so you will have to use a loop to work through the table of data.

Exercise 2 Amending Someone Else's Code

The function ***`FindAndTotal`*** will take an item as a parameter and then return the sum of the total column for all rows with the matching item. Unfortunately, it doesn't work at the moment.

- *Open the spreadsheet Products.xlsm*
- *Run the function FindAndTotal*
- *Fix the problems with the function and make sure it runs correctly*

Task 1 Open the spreadsheet *Products.xlsm*	**Step 1** Start *Excel*
	Step 2 Open the spreadsheet *Products.xlsm*
	Step 3 If a security warning appears just below the ribbon, open the message and make sure you enable macros.
Task 2 Test the function **`FindAndTotal`**	**Step 1** In an empty cell type: **`=FindAndTotal("Pencil")`** And press Enter.
	Step 2 Observe the results

Task 3

Now that you have seen what happens, you should open the VBA Editor, find the module containing the function and fix the function. The VBA Editor is found on the Developer tab of the ribbon. If the tab isn't visible follow the instructions on page 3 of this handbook to make it appear.

There are several problems with the function, including formatting issues, so fixing one problem will likely highlight another one.

Notes:

To help you on your way you should consider adding in some error handling.

Is the loop working correctly?

If you want to get the **value** of a cell what do you need to type?

Keep retesting the function as you go. If working correctly, the total returned for "Pencils" should be 2135.14.

Task 4
If you complete the previous task, help out the creator of this function by making sure it is formatted in a readable way (as discussed) and add some comments to the code so that other people can see what is going on with it.

Exercise 3 Making a Database React to User Input

- *Open the database Search.accdb*
- *Open the form frmSearch*
- *Write VBA code to make the search button work*

Task 1 Start *Access* and open the database	**Step 1** Open *Access*
	Step 2 Open the database *Search.accdb*
	Step 3 If a security warning appears just below the ribbon, open the message and make sure you enable macros.
Task 2 Open the form `frmSearch` in design view	**Step 1** **Right-click** on the form `frmSearch`
	Step 2 Select Design View
Task 3 Create an event handler for the Search button	**Step 1** **Right-click** on the Search button and select Properties
	Step 2 In the properties box, go to the Event tab and **click** on the … button next to the On Click row
	Step 3 Select Code Builder and press OK

Task 4

You will now have a space to start writing your code.

Your code should take the text entered into the search field (called **`SearchField`**) and check the Item column in the table **`tblStock`** for a row that matches the search term.

For the purposes of this exercise, only exact search terms will be considered.

If a match is found, the code should then populate the Item, Price and Quantity fields with the relevant data from the correct row of the table.

If no match is found a suitable message should be displayed to the user informing them of this.

You can check your results by comparing them to the data in the table.

Notes:

You can search the table in one of two ways:

- Use an SQL statement to search the table and return the results in the form of a recordset
- Create a recordset of the table and loop through the records until you find the item you are looking for

In either case you will find the recordset method **`.EOF`** very useful. This stands for End Of File and will return true if it has reached the end of the recordset (as in, there are no more records available).

To display a message to the user you should use the command **`MsgBox`** followed by your message in double quotes (“”).

Make sure you are commenting your code and following best practices.

Task 5

If you manage to complete task 4 and would like to try something more complex, create On Click event handlers for the Take Stock and Replenish Stock buttons on the form.

Take Stock should ask the user how many they want to take, then reduce the quantity by that amount, making sure that the quantity does not go below zero. It should then display a message to the user telling them the total cost of what they are taking (amount * price).

If there isn’t enough stock then the user should be shown a message informing them of this and then end the code.

Replenish stock should ask the user how many of the item they are adding to the stock and then update the quantity field by that amount. The user should then be shown a message confirming it has been done and tell the user what the new quantity is.

Exercise 4 Excel: Long Exercise

Your boss has a spreadsheet full of sales data that he wants you to add fancy things to. However, he is 'macrophobic' – he has an irrational fear of recorded macros. Code written manually is fine, but he can't go near recorded macros (he just doesn't trust them).

Your task is to take the spreadsheet and add VBA functions and subroutines to it that will allow your boss to do the fancy things he wants to do with the data.

One thing he has asked for is the ability to call a function that will return the item that has the highest total sales in the sales data. It won't need any parameters and should just return the item name.

Other than the requested function, your boss has been very vague about what he wants. It is up to you to surprise him with fancy functions. Just remember that macros aren't allowed, not even for simple tasks.

Task 1 Start *Excel* and open the spreadsheet *SalesData.xlsm*	**Step 1** Start *Excel*
	Step 2 Open the spreadsheet *SalesData.xlsm*
	Step 3 If a security warning appears just below the ribbon, open the message and make sure you enable macros.

Task 2

With the spreadsheet open you can now start to work your magic.

Spend a little time considering what you want to create and how you might go about it.

Start with some easy functions to get in to the swing of it, then move on to the more complicated ones, such as the total sales function.

If you are feeling confident straight away then go straight to working on the total sales function, this should give you some ideas about other functions you could create afterwards.

You may remember from Exercise 3 that you fixed a function that returned the total sales for a given item. Conveniently for you, that function has been included in your spreadsheet already. Perhaps there is a way you could make use of it in your total sales function.

Exercise 5 Access: Long Exercise

You have a database that contains the inventory for your stationary business.

You have been tasked with improving the database with some VBA.

Firstly, you need to write code for the **`Process Sales`** *button on the form* **`frmProcess`**. *This code will process the sales data imported in to the table* **`tblSales`**.

There are a few things to consider when putting together the code. Firstly, sales are handled in order of value to the company. This means that orders that have a higher unit price for the same item are fulfilled first. After this, they are sorted by date.

When an item runs out of stock, any unfulfilled orders need to be added to the table **`tblDisappointments`** *so that the clients can be informed. It is not possible to partially fulfil an order so orders that can't be completely fulfilled should also be added to this table.*

You should, however, make best use of the stock, so if there isn't enough stock to fulfil one order but a different order can be fulfilled, then you should fulfil the order that can be completed.

Finally, you should mark on the sales table whether the order has been fulfilled or not.

Relevant messages should be displayed to the user at various points so they are aware of the progress of the function.

If you complete the task above then the management would like you to add some functionality that works out each sales rep's commission. This is worked out as 5% of the total of any fulfilled orders. The details of each rep's commission should be added to the table **`tblCommission`** *using VBA.*

Task 1 Start *Access* and open the database *StationaryBusiness.accdb*	**Step 1** Open *Access*
	Step 2 Open the database *StationaryBusiness.accdb*
	Step 3 If a security warning appears just below the ribbon, open the message and make sure you enable macros.

Task 2

Start by opening the form **`frmProcess`** in Design View and add the VBA to the Process Sales button that will process the sales data.

Spend some time thinking about how the function will work first. As a guide you can declare the function and then add comments within the function to guide you through what you need to add in terms of code.

This will be a very complex function so don't be disheartened if it doesn't work straight away. Keep testing it bit by bit as you build it up and remember to include your error handlers so you can see what is going wrong.

You may find you need to use SQL to work out which orders should be fulfilled first. A good way of getting a usable SQL statement without knowing SQL is to use the Query Builder in Access and then switch to the SQL view. You can then copy and paste the SQL in to your code (you may need to modify it slightly to make it work though).

Be aware that your testing may alter data in the tables so you may want to take a copy of the database before starting so you can reverse any changes you make by mistake.

Task 3

If you complete the task above you can try to improve the functionality by amending the button code to allow a user to import a spreadsheet into the tblSales table. You can use the spreadsheet salesforupload.xls to test with.

Don't forget about adding the functionality to calculate the reps' commission as well.