Microsoft Office Training Series

Excel VBA

Advanced

- Courses never Cancelled
- 12+ Months Schedule
- 24 Months Online Support
- UK Wide Delivery

STL MicrosoftTraining.net

Learning & Performance Institute
Accredited Learning Provider

ilm Recognised Provider

Microsoft Partner
Certified Silver Partner

Rev 2-1

Welcome to your Excel VBA Advanced training course

- Record macros
- The visual basic Editor
- Understand objects (Object oriented programming)
- Control structure using decision code (If Then Else & Select Case)
- Understand and use loops (Do, For Next, For Each)
- Test code and debugging tools

Contents

Unit 1 - Working with Ranges

In this unit you will learn how to:

- Understand the range object
- Use the Special cell method
- Work with collections

What is a Range?

When we refer to a range in Excel we mean either a singular cell, a rectangular block of cells, or a union of many rectangular blocks. In VBA Range is an object with its own properties and methods. Just to complicate things range can also be a property of the application object, the worksheet object and indeed the range object, where it refers to a specified range object.

Range Property of the Application

You can use the range property of the application to refer to a range object on the active worksheet.

For example;

Range("B2")

Range("A1:B7")

Range("A1:B3,E1:O9")

Note the last example refers to a union, or non-contiguous range.

Cells Property

- The Cells Property of the range object can be used to specify the parameters in the range property to define a range object.

For example the following refers to range A1:E5

Range (Cells(1,1), Cells (5,5))

The cells property can also be used to refer to particular cells within a range; or a range within a range.

The following refers to cell F9

Range (“D10:G20”).Cells (0,3)

You can also shortcut this reference thus

Range (“D10:G20”) (0,3)

The SpecialCells Method

The SpecialCells method allows certain types of cell to be identified within a range.

It has the following syntax:

SpecialCells(*Type*, *Value)*

The type argument specifies the cells to be included

xlCellTypeAllFormatConditions	Cells of any format
xlCellTypeAllValidation	Cells having validation criteria
xlCellTypeBlanks	Empty cells
xlCellTypeComments	Cells containing notes
xlCellTypeConstants	Cells containing constants
xlCellTypeFormulas	Cells containing formulas
xlCellTypeLastCell	The last cell in the used range
xlCellTypeSameFormatConditions	Cells having the same format
xlCellTypeSameValidation	Cells having the same validation criteria
xlCellTypeVisible	All visible cells
xlCellTypeFormulas.	Cells containing formulas
xlCellTypeLastCell.	The last cell in the used range
xlCellTypeSameFormatConditions.	Cells having the same format
xlCellTypeSameValidation.	Cells having the same validation criteria
xlCellTypeVisible.	All visible cells

This argument is used to determine which types of cells to include in the result

xlErrors
xlLogical
xlNumbers
xlTextValues

The following code will delete all the numbers in a worksheet, leaving only text data and formulae in place

```
Sub DeleteNumbersInworksheet()

    Cells.SpecialCells(xlCellTypeConstants, xlNumbers).ClearContents

End Sub
```

Naming Ranges

One of the most useful techniques in Excel is to name ranges. A named range can simplify code as it is possible to refer to the name and not the cell references

To create a named range we use the add method of the workbook's names collection. The following code creates a named range called "NewName" on sheet2 of the active workbook on the range "E5:J10"

```
Sub AddNamedrange()

    Names.Add Name:="NewName", RefersTo:="=Sheet2!$E$5:$J$10"

End Sub
```

Alternatively it is possible to set a name by defining the name property of the range object.

```
Sub AddRangeNameProperty()



   Range("A1:V3").Name = "RangeName"



End Sub
```

Working with Collections

A class is a blueprint for an object, and individual objects are "instances" of a class. A collection is simply a group of individual objects with which we are going to work.

For example in the code above we have defined a class called customers, and code to generate a single instance of that class; i.e. one individual customer. In practice we will be working with more than one customer and we will wish to define them as being part of a collection object so we can process them using some of the methods and properties of the collection object.

The Collection Object

The collection object has a number of properties and methods associated with it; of which the most important are:

Method/Property	**Description**
Count	A method that returns the number of objects in the collection
Add	A method that adds an item to the collection

Remove	Removes an item to a collection
Items(index)	Refers to an individual item in the collection either by its index number (position in collection) or by its name

Explicit creation of a collection

We can create a collection in a class module. This simply requires us to define the collections objects and methods in the normal way

```
Option Explicit

Private FCustomers As New Collection


Public Function add(ByVal value As Customer)

   Call FCustomers.add(value, value.Name)

End Function


Public Property Get Count() As Long

   Count = FCustomers.Count

End Property


Public Property Get Items() As Collection

   Set Items = FCustomers

End Property
```

```
Public Property Get Item(ByVal value As Variant) As Customer
    Set Item = FCustomers(value)
End Property


Public Sub Remove(ByVal value As Variant)
    Call FCustomers.Remove(value)
End Sub
```

The above code simply defines a collection called customers (class module name). The variable FCustomers is defined as a collection object. The various methods and properties are then defined. For example, the remove method is defined in a procedure that uses the remove method of the collection object to remove a specified item from the collection.

Referring to a collection in a standard module

Once defined, a collection can be employed in the same way as any other collection.

```
Dim aCustomer As Customer
Dim theCustomers As New Customers
    Set aCustomer = New Customer
    aCustomer.Name = "Kur Avon"
    aCustomer.MainAddress = "132 Long Lane"
    Call theCustomers.add(aCustomer)


    Set aCustomer = New Customer
    aCustomer.Name = "Fred Perry"
    aCustomer.MainAddress = "133 Long Lane"
```

```
    Call theCustomers.add(aCustomer)


    Set aCustomer = New Customer
    aCustomer.Name = "Jo Bloggs"
    aCustomer.MainAddress = "134 Long Lane"
    Call theCustomers.add(aCustomer)


For Each aCustomer In theCustomers.Items
        Sheets(1).Range("A1").Select
        ActiveCell.value = aCustomer.Name
        ActiveCell.Offset(0, 1).value = aCustomer.MainAddress
        ActiveCell.Offset(1, 0).Select
 Next aCustomer
```

The above code simply defines a "customer" variable and a "customers" variable; assigns three objects to the collection and then writes the name and address to a worksheet in the current workbook, using a "FOR EACH" loop.

Using the Collections Object Directly

It is possible to create a collection using the VBA collection class directly. The code below creates a collection called employees and assigns three instances of the custom object employees to it.

```
Sub TestEmployeesCollection()
    Dim anEmployee As Employee
    Dim i As Long
        Set anEmployee = New Employee
    anEmployee.Name = "Stephen Williams"
    anEmployee.Rate = 500
```

```
    anEmployee.HoursPerWeek = 50
    Call Employees.add(anEmployee, anEmployee.Name)
        Set anEmployee = New Employee
    anEmployee.Name = "Kur Avon"
    anEmployee.Rate = 50
    anEmployee.HoursPerWeek = 50
    Call Employees.add(anEmployee, anEmployee.Name)
        Set anEmployee = New Employee
    anEmployee.Name = "Bill Bailey"
    anEmployee.Rate = 250
    anEmployee.HoursPerWeek = 50
    Call Employees.add(anEmployee, anEmployee.Name)
        Set anEmployee = New Employee
    anEmployee.Name = "Alexander Armstrong"
    anEmployee.Rate = 250
    anEmployee.HoursPerWeek = 50
    Call Employees.add(anEmployee, anEmployee.Name)
        For Each anEmployee In Employees
          MsgBox anEmployee.Name & " Earns " & "£" &
anEmployee.GetGrossWeeklyPay()
        Next anEmployee


End Sub
```

Unit 2 - Charts

In this unit you will learn how to:

- Create charts using VBA

Creating charts from worksheet data

Charts are created by working with the chart object. The key elements to a chart are:

- Data source
- Type
- Location

These are controlled by the following properties.

Key Properties and methods of the chart object

Properties/Methods	Description
SetSourceData	This specifies the data that will be modelled in the chart. Includes 2 key arguments; Source which specifies the data range, and PlotBy which determines if the series is in rows or columns
ChartType	Select one from a list of chart types recognized by Excel
Location	Specifies if the chart is to be embedded into a worksheet or whether it will occupy a sheet of its own
Add	Adds a new chart to the active workbook

The following code example creates a simple chart object and then sets the above properties.

```
Public Sub EmbeddedChart()



   Set aChart = Charts.Add
```

```
    Set aChart = aChart.Location(Where:=xlLocationAsObject, Name:="Sheet1")
        With aChart
            .ChartType = xl3DBarClustered
             .SetSourceData Source:=Sheets("Sheet1").Range("B2:E6"), PlotBy:=xlRows
            .HasTitle = True
        .  ChartTitle.Text = "Sales Summary"
        End With
End Sub
```

Creating Charts from Arrays

In the example above, the chart's source data was to be found in sheet1 range B2:E6 of the active workbook. It is however possible to set a chart's source data to the contents of an array.

```
Public Sub ChartFromArray()


    Dim SourceRange As Range
    Dim aWorksheet As Worksheet
    Dim aWorkBook As Workbook
    Dim aChart As Chart
    Dim aNewSeries As Series
    Dim intCount As Integer
    Dim SalesArray As Variant
    Dim MonthArray As Variant


    MonthArray = Array("Jan", "Feb", "March")
```

```
Set SourceRange = Sheets("Source Sheet").Range("B2:E6")


Set aWorkBook = Workbooks.Add
Set aWorksheet = aWorkBook.Worksheets(1)
Set aChart = aWorkBook.Charts.Add
    With aChart
        For intCount = 1 To 4
            'create a new series
            Set aNewSeries = .SeriesCollection.NewSeries
    SalesArray = SourceRange.Offset(intCount, 1).Resize(1, 3).Value
    aNewSeries.Values = SalesArray
    aNewSeries.XValues = MonthArray
                Next intCount
            .HasLegend = True
    .HasTitle = True
            .ChartTitle.Text = "First Quarter Sales"


End With
```

The above code creates a new workbook, adds a chart and then populates the chart with data taken from a source workbook.

Within the For...Next loop, four new series are created. At each loop a new series is created with the "NewSeries" method. The appropriate row's data is then assigned directly to the variant "SalesArray", and sales array is assigned to the values property of the new series.

Unit 3 - PivotTable Object

In this unit you will learn how to:

- Use VBA to create PivotTables

Understanding PivotTables

A pivot table is a table that can be used to summarize data from a worksheet or an external source such as a database.

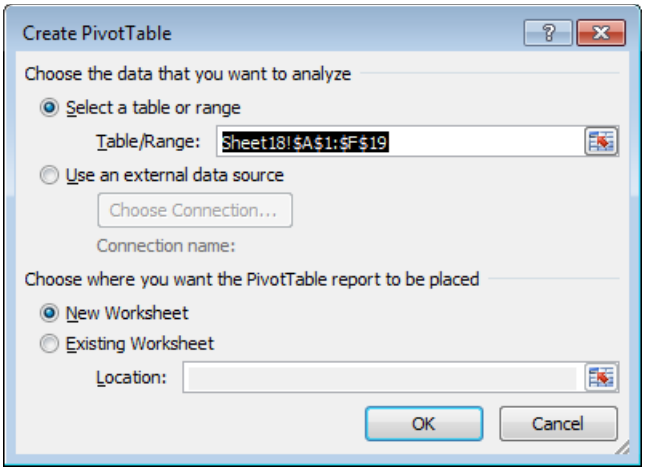
A Pivot table can only be created using the Pivot table wizard.

Creating A PivotTable

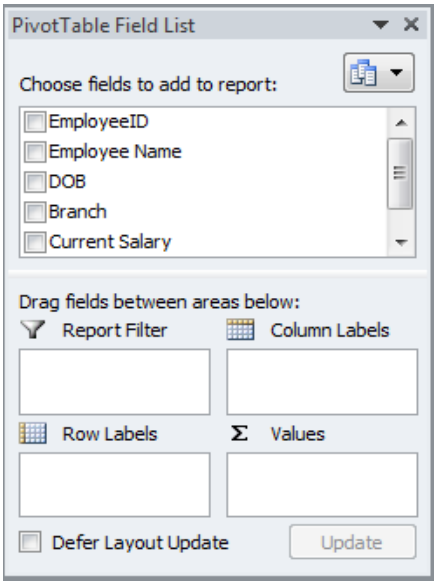
The wizard makes the creation of the pivot table quite easy. By following a series of prompts the wizard takes over and creates the pivot table for you. To do this:

Insert Ribbon > PivotTable Button (Far left)

Procedure



- Select **Where the data is that you want to analyze**
- Select **where you want to create the report**
- Click **OK**.



- Drag the field buttons to the desired page, row, column and data fields.

Using the PivotTable Wizard Method

The **PivotTable Wizard** method of the Worksheet object can be used to create a pivot table in code without displaying the wizard.

The **PivotTable Wizard** method has many arguments. The main ones are described below:

Argument	Definition
SourceType	The source of the PivotTable data. The SourceData argument must also be specified when using this.
SourceData	A range object that specifies the data for the PivotTable.
TableDestination	A range object indicating where the table will be placed.
TableName	The name by which the table can be referred.

An example of the **PivotTable Wizard** method is shown below:

```
Sub MakePivot ()

Dim DataRange As Range
Dim Destination As Range
Dim PvtTable As PivotTable

Set Destination = Worksheets("Sales Summary").Range("A12")
Set DataRange = Range("A9", Range("J9").End(xlDown))

ActiveSheet.PivotTableWizard SourceType:=xlDatabase, _
SourceData:=DataRange, TableDestination:=Destination, TableName:="SalesInfo"

End Sub
```

This code runs the PivotTable wizard, capturing the data in the current worksheet then placing a pivot table in the worksheet called "Sales Summary". In this instance the PivotTable contains no data, because the row, column and data fields haven't been assigned.

Using PivotFields

Once a PivotTable is created pivot fields must be assigned. The **PivotFields** collection is a member of the PivotTable object containing the data in the data source with each Pivot Field getting its name from the column header. PivotFields can be set to page, row, column and data fields in the PivotTable.

In the Sales – April 2004 the fields are: Sales Date, Make, Model, Type, Colour, Year, VIN Number, Dealer Price, Selling Price, Salesperson.

The table below lists the PivotTable destinations for PivotFields.

Destination	**Constant**
Row Field	**xlRowField**
Column Field	**xlColumnField**
Page Field	**xlPageField**
Data Field	**xlDataField**
To Hide A Field	**xlHidden**

The following syntax shows how a PivotField is defined by setting its Orientation property to the desired destination column:

```
.PivotTables(Index).PivotFields(Index).Orientation = Destination
```

```
.PivotTables("SalesInfo").PivotFields("Salesperson").Orientation = xlPageField

PivotTables("SalesInfo").PivotFields("Colour").Orientation = xlRowField
```

To optimize the setting of the Pivot Table orientation use the With Statement:

```
Set PvtTable = Sheets("Sales Summary").PivotTables("SalesInfo")

With PvtTable

    .PivotFields("Salesperson").Orientation = xlPageField
    .PivotFields("Year").Orientation = xlRowField
    .PivotFields("Make").Orientation = xlColumnField
    .PivotFields("Selling Price").Orientation = xlDataField

End With
```

Unit 4 - Working with Arrays

In this unit you will learn how to:

- Understand an array
- Create one dimension and multiple dimension arrays
- Work with lbound & ubound

What is an Array

An array can be regarded as a variable that can hold a collection of values which can be referenced by an index number. Typically an array is defined in the same way as a variable, with the difference that it is followed by brackets.

The following code contains an array that can hold 5 integers

```
Dim intArray(1 to 5) as integer
```

When creating an array it is necessary to specify its size (the number of elements that it can hold) and the number of dimensions contained by the array.

Array Sizes

An array's size can be specified either when it is declared or later during the code's execution. The former case creates a static array, the later a dynamic array.

A static array is an array that is sized in the Dim statement that declares the array.

```
Dim StatArray(1 To 100) As string
```

You can't change the size or data type of a static array.

A dynamic array is an array that is not sized in the Dim statement. Instead, it is sized later with the ReDim statement.

```
Dim DynArray() As string
ReDim DynamicArray(1 To 100)
```

You can change the size of a dynamic array, but not the data type.

One Dimensional Arrays

The arrays considered so far are one dimensional, in that they have a simple row of variables. For example an array defined as;

Dim strArray(1-5) as string

Could be visualized as

1	2	3	4	5

with 5 spaces which can contain string values.

When the array is populated, we could visualize the following

Bill	Ben	Fred	Mary	Jane

In the first view, the numbers are the array's index numbers which are used to identify a particular element. The second refers to the values actually contained in the array.

To allocate a value to a location in array, it is simply necessary to reference the index number and set that equal to the value required. For example:

strArray(3) = "Fred"

…would be the code used to set the value of the array's third element to the string value "Fred".

Arrays with Multiple Dimensions

Arrays can have more than one index number; that is they can have more than one dimension. Typically we will use 2 dimensional arrays which are in effect virtual tables.

The following code

Dim strArray(1 to 5,1 to 3)

….creates the following

1,1	1,2	1,3
2,1	2,2	2,3

3,1	3,2	3,3
4,1	4,2	4,3
5,1	5,2	5,3

This is an array with 5 rows and 3 columns. If we wanted to set the value of the last element in the array to the word "hello", we would need;

StrArray(5,3) = "Hello"

While single and two dimensional arrays are the most commonly used; arrays can have up to 60 dimensions.

Thus the following

StrArray(1 to 3,1 to 9,1 to 6)

….defines a virtual cube containing 162 spaces.

Once we exceed 3 dimensions, mathematically we are working with hypercubes which are hard to visualize! Fortunately, it is unlikely that you will ever need them.

A word about index numbers

Thus far we have explicitly specified the index numbers in an array as follows;

Dim intArray(1 to 4) as integer

…..which specifies that the first location is numbered 1 and the last 4.

We could however define the array as follows

Dim intArray(4) as integer

Here we have again defined an array with 4 locations. However, under normal circumstances the index numbers would be;

0	1	2	3

With the first index number starting zero.

This can be changed by using the Option Base statement in the declarations section of the module containing the code

Option Base 1

…..sets the lower bound index of any array to 1.

It is however better practice to explicitly specify the lower bound index number in the array's declaration.

Ubound and Lbound

The Ubound and Lbound functions return the highest and lowest index numbers in the array. They are useful when cycling through the values contained in an array.

The following code uses the Lbound and Ubound functions to view each item contained in an array

```
Public Sub Array1()
   Dim data(1 To 10) As Integer
   Dim I As Integer
   For I = LBound(data) To UBound(data)
      MsgBox data(I)
   Next I
End Sub
```

Saving arrays in names

As with any variable the array has a limited lifetime which terminates at the latest when the application ceases to run. However, in the same way that we can give a range a name that is saved within the workbook we can also name an array.

This array will then be saved with the workbook and can then be available when the workbook opens.

The following code creates an array, populates it and then saves it to a name; using the add method of the names collection.

This technique allows large volumes of data to be stored in a workbook, outside of the standard worksheets.

```
Public Sub ArrayToName()
   Dim MyArray(1 To 200, 1 To 3) As Integer
   Dim I As Integer
   Dim J As Integer
   For I = 1 To 200
      For J = 1 To 3
         MyArray(I, J) = I + J
      Next J
   Next I
   Names.Add Name:="MyName", RefersTo:=MyArray
End Sub
```

Unit 5 - Triggers and Events

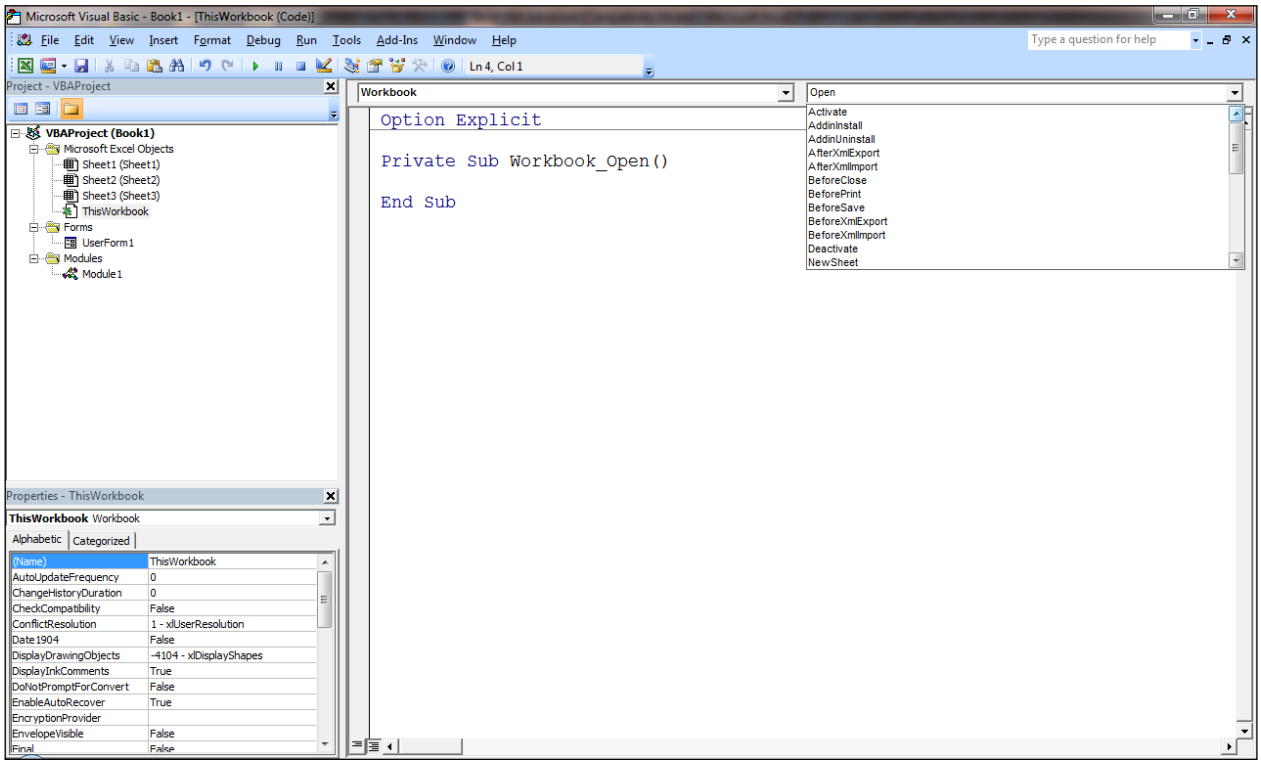
In this unit you will learn how to:

- Create workbook events
- Create worksheet event
- Work with timed events

An event is "something that happens" to an object, and usually occurs when an object undergoes a change of state.

For instance when a workbook is selected, its status changes from deactivated to activated and the Activate event fires. Code can be embedded in special event procedures and will run whenever certain events occur.

The screenshot below shows how to select an object and then access the relevant event



Workbook Events

Events for the Workbook object occurs when the workbook is changed or a sheet in the workbook is changed.

Select the desired project in the Project-window of the VBE and activate the object ThisWorkbook by double-clicking it. Any event procedures in the workbook will now be displayed in the Code-window on the right side of the screen. You can create a new event Procedure by selecting Workbook in the Object dropdown, and then select an event in the Procedure dropdown.

The main workbook events are:

- Activate (When the workbook is selected. Also fires when the workbook opens, after the open event)
- AddinInstall
- AddinUninstall
- BeforeClose (Can be used to "clean up" workbook before it closes. Also you can run the save method at this point to ensure the file always automatically saves any changes)
- BeforePrint
- BeforeSave
- Deactivate (Fires when another workbook or application is selected)
- NewSheet (when a new sheet is created)
- Open
- SheetActivate
- SheetBeforeDoubleClick
- SheetBeforeRightClick
- SheetCalculate
- SheetChange
- SheetDeactivate
- SheetSelectionChange
- WindowActivate
- WindowDeactivate
- WindowResize

Worksheet Events

In the worksheet dropdown you can access the following events

- Activate
- BeforeDoubleClick
- BeforeRightClick
- Calculate (Runs whenever a formula's dependent cell value is changed, or when F9 is pressed.)
- Change
- Deactivate
- SelectionChange

Timer Controlled Macro

Timer Event example code:

```
Private Sub Workbook_Open()

'Application.OnTime Now() + TimeSerial(0, 0, 10), "TimeMe"

Application.Wait Now() + TimeValue("00:00:05")

MsgBox "Hi its me"

End Sub
```

If using the OnTime you need to specify another routine for it to go to.

Use this line if you want to run a macro at a preset time e.g. here set for 9.17pm

```
Application.OnTime TimeSerial(21, 17, 10), "TimeMe"
```

Use either TimeSerial or TimeValue.

Unit 6 - Working with Text Files

In this unit you will learn how to:

- Import a text file
- Understand FileStream

For people that deal with databases and large systems, the text file is the common 'language' that they can all converse in. This could be in a variety of formats, such as: TXT, PRN, CSV, TSV and many more. It is beneficial to be aware of routines to handle importing and exporting text files as VBA can use them as an input / output between systems.

Importing a Text File

This procedure allows you to import data from a delimited text file. Each line in the text file is written to one row in the worksheet. Items in the text file are separated into separate columns on the worksheet row based on the character you specify.

Minimum code required to import a text delimited file:

```
With
ActiveSheet.QueryTables.Add(Connection:="TEXT;C:\VBA\EmployeeData.txt",
Destination:=Range("A1"))
      .TextFileStartRow = 1
      .TextFileCommaDelimiter = True
      .Refresh BackgroundQuery:=False
   End With
```

FileStream

Use the FileStream class to read from, write to, open, and close files on a file system, and to manipulate other file-related operating system handles, including pipes, standard input, and standard output.

FileStream buffers input and output for better performance.

FileStream can be used across different VBA models (Excel, Word, Outlook).

FileStream needs to be loaded from the VBA Library before use.

- To reference this file, load the Visual Basic Editor (ALT-F11)
- Select Tools - References from the drop-down menu
- A listbox of available references will be displayed
- Tick the check-box next to 'Microsoft Scripting Runtime'
- The full name and path of the scrrun.dll file will be displayed below the listbox
- Click on the OK button

The two most useful lines of code are

```
FileSystemObject.OpenTextFile

FileSystemObject.CreateTextFile
```

To read in and create a text file, respectively.

Unit 7 - Working with Procedures and Parameters

In this unit you will learn how to:

- Passing arguments
- Use optional arguments
- Passing arguments ByVal & ByRef

Procedure Arguments

There are two types of procedure; sub procedures and function procedures. The difference between them is that function procedures return values, and sub procedures do not. Both sub procedures and function procedures accept arguments. An argument is simply a piece of information that the procedure is to process.

Passing Arguments

The arguments of a procedure are defined within the brackets after the procedure's name. They are then processed within the procedure.

The following function accepts 2 string variables and then concatenates them together.

```
Function StringJoiner(Name1 As String, Name2 As String) As String


   StringJoiner = Name1 & Name2


End Function
```

It is then called from the following sub procedure, with the two arguments defined.

```
Sub RunStringJoiner()

```

```
    Dim strResult As String

    strResult = StringJoiner("Stephen ", "Williams")

    MsgBox strResult

End Sub
```

Optional Arguments

You can specify that some or all of the arguments in a procedure are optional.

This procedure has an optional argument strMessage

```
Sub OptArgument(Optional strMessage As String)

    If strMessage <> "" Then

        MsgBox strMessage

    Else

        MsgBox "I have nothing to say"

    End If

End Sub
```

It is called from the following procedure. The first line returns a message box with the word argument value hello as the message. The second has no value for the argument and returns the message “I have nothing to say”.

```
Sub CallOptArg()



   Call OptArgument("Hello")

   Call OptArgument



End Sub
```

Default Values

It is common to include a default value with an optional argument. This will be the value of the argument if it is omitted when the procedure is called.

```
Sub OptArgument(Optional strMessage As String =  “I have nothing to say”)

      MsgBox strMessage

End Sub
```

The above procedure has exactly the same results as the previous example, but is clearly a lot simpler to code and understand.

Passing arguments by value and reference

By default, arguments in VBA are passed by reference. This means that if you pass a variable as an argument from one procedure to another then the called procedure is working with the exact same copy of the variable as the calling procedure. When you pass a variable by value, then the calling procedure makes a copy of the variable, hands that to the called procedure; but retains the original itself. As a result, the variable in the calling procedure is unaffected by the changes made in the calling procedure.

The following procedure sets a variable intVar to the value of 10 and then passes it to another procedure by reference. This procedure adds 10 to it and hands it back, where the final value of 20 is displayed in a message box.

```
Sub PassByRef()

    Dim intVar As Integer

    intVar = 10

    Call RecByRef(intVar)

    MsgBox intVar

End Sub
```

```
Sub RecByRef(IntArgument As Integer)
   IntArgument = IntArgument + 10
   End Sub
```

In the following example the intVar is passed by value to the sub RecByRef. Here a copy of the variable is processed, which means it is not passed back to the calling procedure. As a result the message box returns the value 10

```
Sub PassByVal()
   Dim intVar As Integer
      intVar = 10
       Call RecByVal(intVar)
      MsgBox intVar
End Sub
Sub RecByVal(ByVal intArgument As Integer)

   intArgument = intArgument + 10

End Sub
```

The key is the argument in the called procedure

Sub RecByVal(ByVal intArgument As Integer)

The byVal keyword specifies that the argument has been passed by value; that is that it is a copy and that the original value will be retained after the called procedure has completed.

Unit 8 - Active X Data Objects

In this unit you will learn how to:

- Understand ADO (Active X Data Object)
- Use then connection object and the recordset object
- Create a universal data link tool

Microsoft's ActiveX Data Objects (ADO) is a set of objects for accessing data sources. It provides a layer between VBA and the OLE DB, for example an Access Database.

ADO allows a developer to write programs that access data without knowing how the database is implemented. You must be aware of your database for connection only. No knowledge of SQL is required to access a database when using ADO, although you can use ADO to execute arbitrary SQL commands.

Key Objects

There are two key objects that concern us; the Connection Object and the Recordset Object.

The Connection Object is the link between your Excel Spreadsheet and the database itself. The link must be opened initially, it must be closed when you're finished, and it has varying qualities. These qualities are the *properties* and methods of the Connection Object.

The Recordset is the object you're going to be doing almost all of your work with. A RecordSet object is a container for what is called a *Cursor*. A cursor is a temporary table, which is constructed by performing a query on a table in a database. It doesn't exist in a file; it exists in memory, but other than that, it has all the characteristics of a database table. It has rows (records) and columns (fields), and the rows and columns have properties of their own.

The Connection Object

The connection object has a child object known as the connection string. The connection string provides the path to the database, together with additional information concerning the database's properties.

The following code creates a constant to take the required connection string. It then creates an ADODB connection object and then sets that object's connection string to the defined constant. The connection is then opened, using the connection string's open method.

```
Const ConnString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Documents and Settings\Storage\Work\Access\Simulated Server\Northwind 2003v2.mdb;Persist Security Info=False"

Dim Connection As ADODB.Connection

Connection.ConnectionString = ConnString

Connection.Open
```

The RecordSet Object

The RecordSet Object is used to represent a table or query in the database defined by the connection string. An object variable is defined as an ADODB RecordSet and is then set to the required table using an SQL statement.

The Recordset can then be manipulated with the following methods and properties.

MoveFirst	Move to first record
MoveNext	Move to next Record
MovePrevious	Move to previous Record
MoveLast	Move to last Record
Edit	Edit current record
AddNew	Add new record
Update	Update changes
Fields()	Used to specify a particular field in the current record either by index number or name
EOF	Specifies whether the cursor is at the end of the file
BOF	Specifies that the cursor is at the beginning of the file

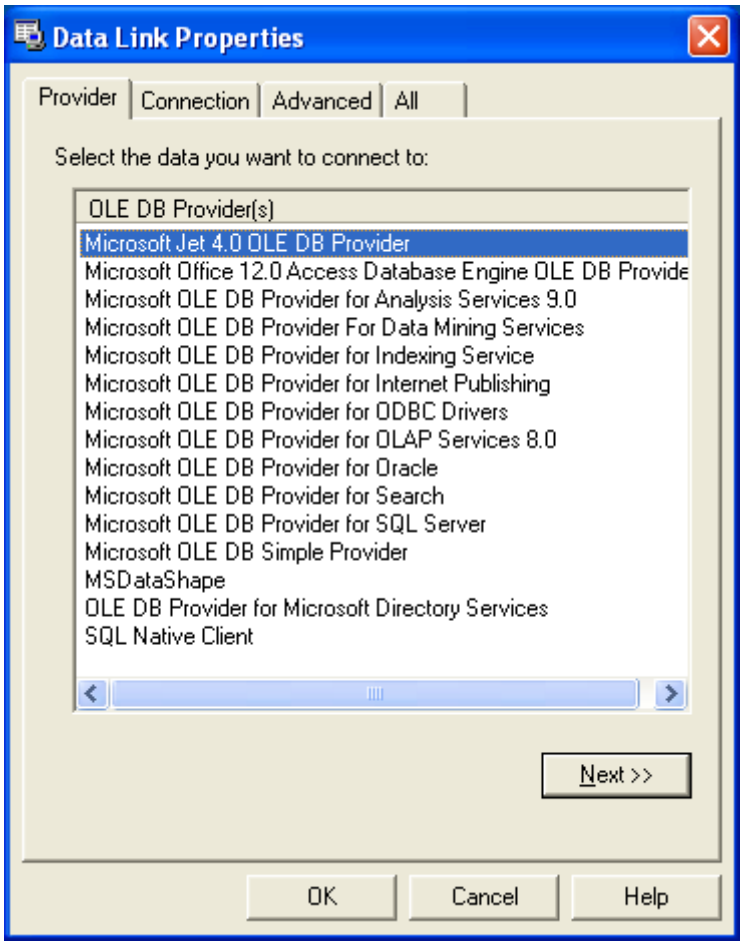
The following code follows on from that shown for the connection object it opens the customer's table as a RecordSet, moves to the first record and then cycles through to the end of the file, writing the customer name to a cell in the active worksheet.

```
Const SQL = "SELECT * FROM customers"

Set rstCustomer = Connection.Execute(SQL)

With rstCustomer

      .MoveFirst

        Sheets(1).Range("a1").select

      Do While Not .EOF

         Activecell.value = .Fields("CompanyName")

         .MoveNext

        Activecell.Offset(1,0).Select

      Loop

End with
```

A word about the connection string

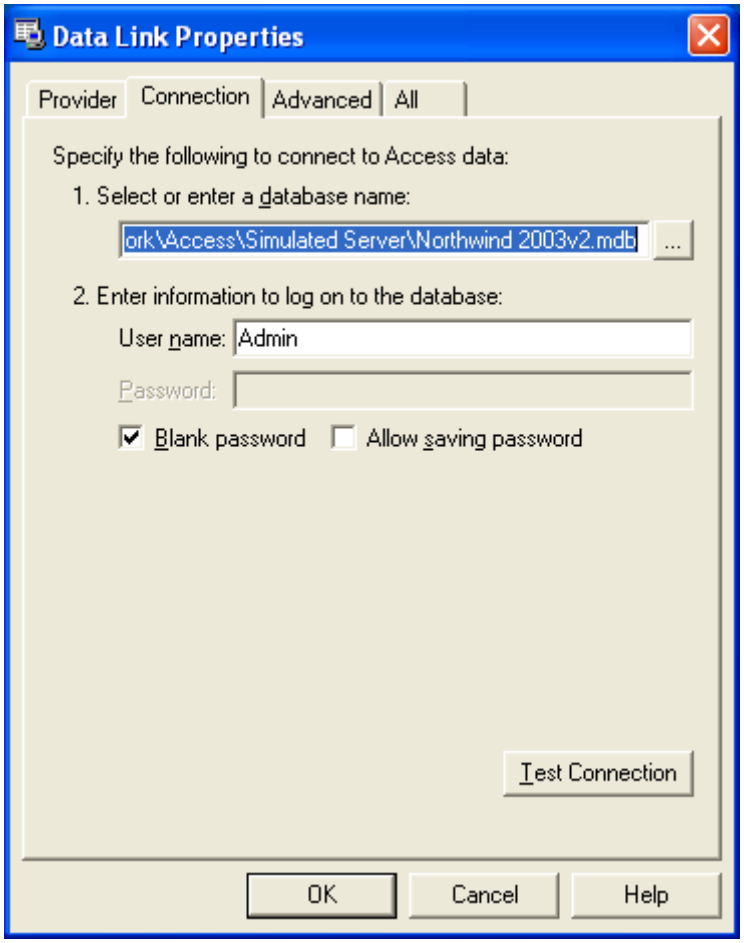
The connection string can often be difficult to code correctly. Fortunately there exists a simple technique to define the string, which involves creating a GUI tool that allows you to browse for the source file and then automatically calculate the connection string.

Create an empty text file and save it with the extension .udl. It will then open as a dialog box.

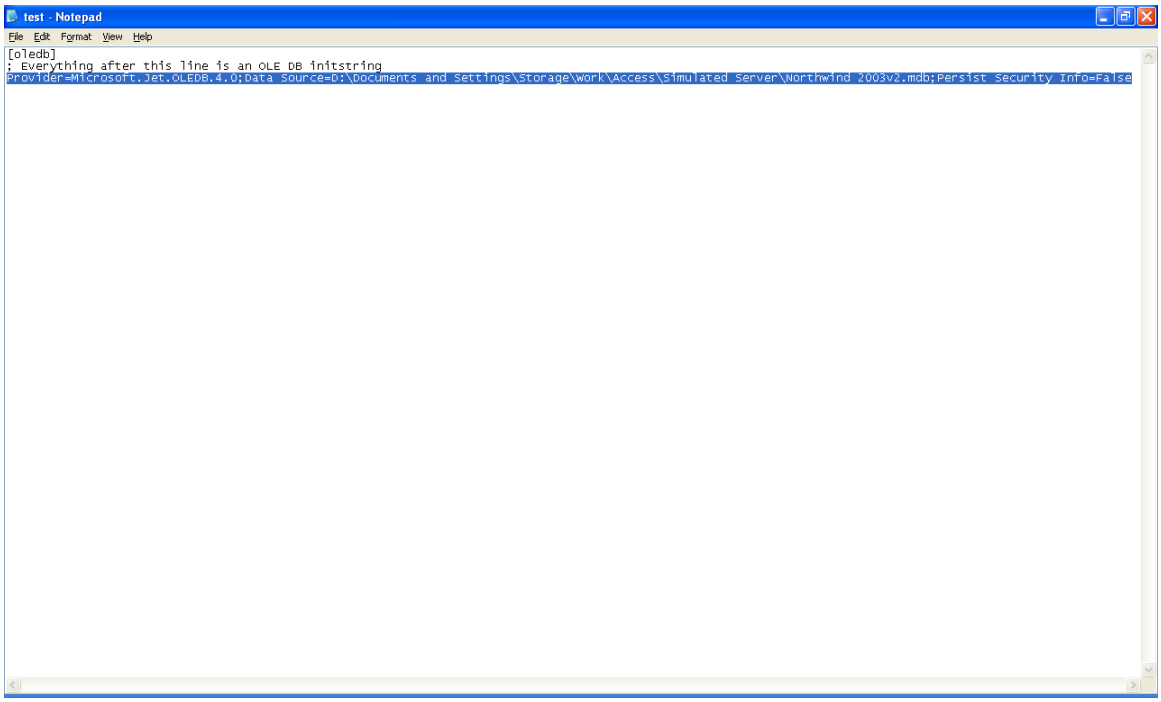


On the provider tab select the appropriate OLE DB Provider.

On the connection tab, click the browse button and select the required database



Then close the dialog box and open the file using notepad. Within the file you will see the connection string clearly labeled.



You can the copy and paste the code into your procedure.

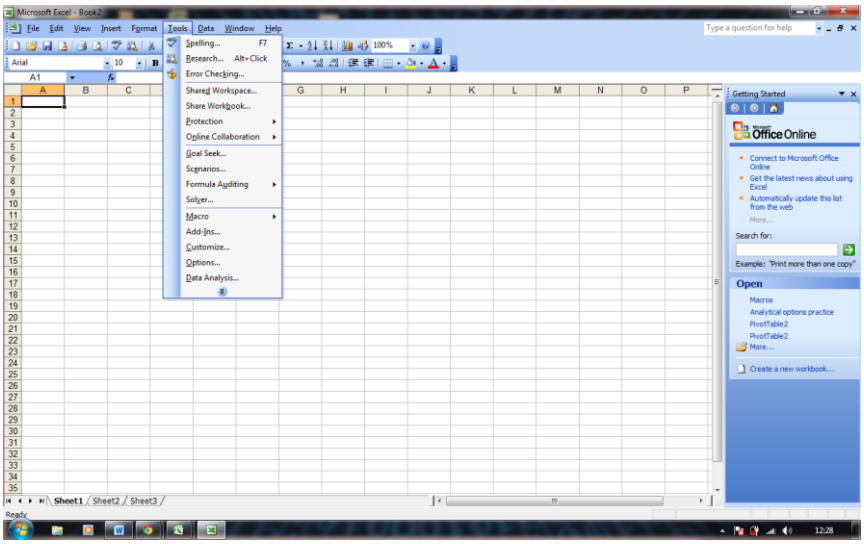
Unit 9 - Creating Add-Ins

When we create a customised function, that function will typically only be available within that workbook. To make the function available to all workbooks, we must create and then open an Add-In file.

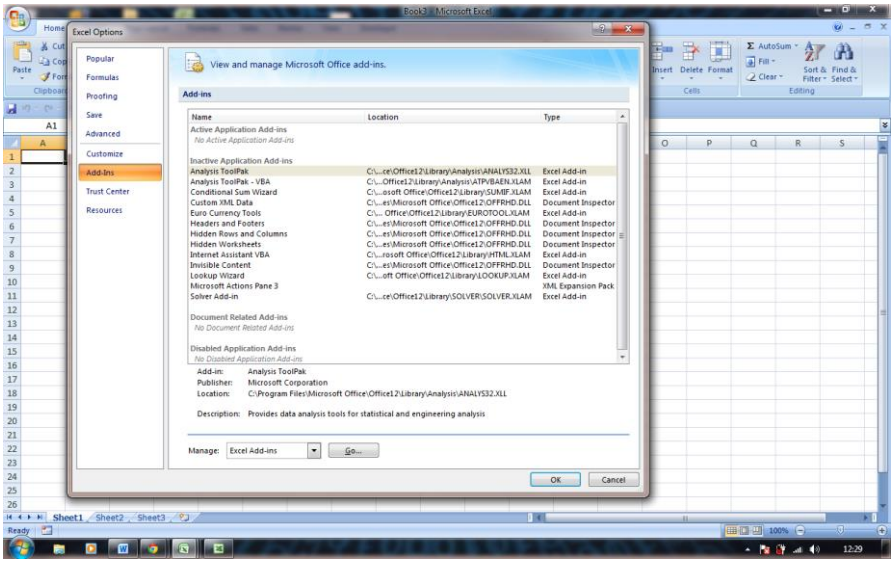
1. Create all the required functions in a separate workbook. This workbook should contain no data as it is sole purpose is to hold the functions

2. The file should then be saved and the file type should be Excel AddIn

3. It is then necessary to install the Addin file.



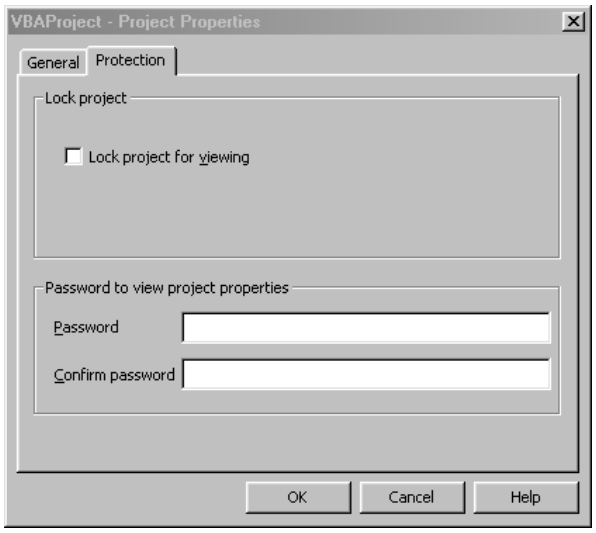
4. Click on the office button/file tab, select Excel options and click on the AddIns command. Alongside manage click "Go" and then select the required AddIn



VBA Password Protection

When we write VBA code it is often desirable to have the VBA Macro code not visible to end-users. This is to protect your intellectual property and/or stop users messing about with your code.

To protect your code, from within the Visual Basic Editor



- Open the **Tools** Menu
- Select **VBA Project Properties**

The Project **Properties** dialog box appears.

- Click the **Protection page tab**
- Check "**Lock project for viewing**"
- Enter your password and again to confirm it.
- Click **OK**

After doing this you must **Save and Close** the Workbook for the protection to take effect.

The safest password to use is one that uses a combination of upper, lower case text and numbers. Be sure not to forget it.

Unit 9 - About Macro Security

In this unit you will learn how to:

- Set security level
- Add a password to the code

In Excel, you can set a macro security level to control what happens when you open a workbook that contains a macro.

Macro security settings and their effects

The following list summarizes the various macro security settings. Under all settings, if antivirus software that works with 2007 Microsoft Office system is installed and the workbook contains macros, the workbook is scanned for known viruses before it is opened.

- **Disable all macros without notification** Click this option if you don't trust macros. All macros in documents and security alerts about macros are disabled. If there are documents that contain unsigned macros that you do trust, you can put those documents into a trusted location.

- **Disable all macros with notification** This is the default setting. Click this option if you want macros to be disabled, but you want to get security alerts if there are macros present. This way, you can choose when to enable those macros on a case by case basis.

- **Disable all macros except digitally signed macros** This setting is the same as the Disable all macros with notification option, except that if the macro is digitally signed by a trusted publisher, the macro can run if you have already trusted the publisher. If you have not trusted the publisher, you are notified. That way, you can choose to enable those signed macros or trust the publisher. All unsigned macros are disabled without notification.

- **Enable all macros** (not recommended, potentially dangerous code can run) Click this option to allow all macros to run. Using this setting makes your computer

vulnerable to potentially malicious code and is not recommended.

- **Trust access to the VBA project object model** This setting is for developers and is used to deliberately lock out or allow programmatic access to the VBA object model from any Automation client. In other words, it provides a security option for code that is written to automate an Office program and programmatically manipulate the Microsoft Visual Basic for Applications (VBA) environment and object model.

Change Macro Security Settings

You can change macro security settings in the Trust Center, unless a system administrator in your organization has changed the default settings to prevent you from changing the settings.

1. On the Developer tab, in the Code group, click Macro Security.
2. In the Macro Settings category, under Macro Settings, click the option that you want.

NOTE Any changes that you make in the Macro Settings category in Excel apply only to Excel and do not affect any other Microsoft Office program.

Appendix
Class Modules

What can be done with Class Modules?

Class modules allow you to create and use your own object types in your application. This implies the following;

- You can easily write code that works with any workbooks that do not have any code.
- Two or more procedures for the event of a command button can be consolidated in one
- The code is easy to use by concealing logic and data.

Why use Class Modules?

Classes make your code:

- Development simpler
- More manageable
- Self-documenting
- Easier to maintain

What is a Class?

A Class is a Blueprint or template of an Object.

In Excel VBA, an Object can mean Workbooks, Worksheets, User forms and Controls etc. Normally an Object has Properties or Methods. A Property stands for Data that describes the Object, and a Method stands for an action that can be ordered to the object.

Properties and Methods of the Object depend on the kind of Object.

For Example;

Worksheet (1).Select

... selects the first worksheet in the workbook. Select is a method of the worksheet object.

How Does a Class Module Work?

A Class Module is a place where a Class is defined. The procedures in a class module are never called directly from other modules like the procedures placed in the standard modules.

In the view of a standard module, the class module doesn't exist.

The thing that exists in the view of a standard module is an instance of the object generated by the class defined by the class module. The methods and procedures of the class are defined within the class module.

Key Elements in a class module

The class module defines all the properties and methods associated with the class. In the example below the "customer" class has two properties associated properties; Name and Main Address.

These are defined by the Property Get and Property let Procedures (see below).

The Customer ID is calculated by taking the leftmost 3 characters from the customer's Name and concatenating that with the 5 leftmost characters from the main Address. This is the result of the method GetCustomerID, and is defined in a function in the class module

Property Get and Let Procedures

A property is implemented using a property let and a property get procedure. When someone sets a value for a property the property let procedure is called with the new value. When someone reads the value of a property the property get procedure is called to return the value. The value is stored as an internal private variable.

Read only properties can be created by implementing a property get procedure without a corresponding property let procedure.

Example of a Class Module

```
Option Explicit

Private strName As String

Private strAddress As String

Public Property Get Name() As String

   Name = strName

End Property


Public Property Let Name(ByVal value As String)

   strName = value

End Property


Public Function GetCustomerID()

   GetCustomerID = Left(strName, 3) & Left(strAddress, 5)

End Function


Public Property Get MainAddress() As String

   MainAddress = strAddress

End Property


Public Property Let MainAddress(ByVal value As String)

   strAddress = value

End Property
```

Referring to user defined Objects in Code

This simply involves creating an instance of the Class in Code and then manipulating it is the way you would any other object.

The following code would be placed in a standard module, and refers to the customer object defined previously.

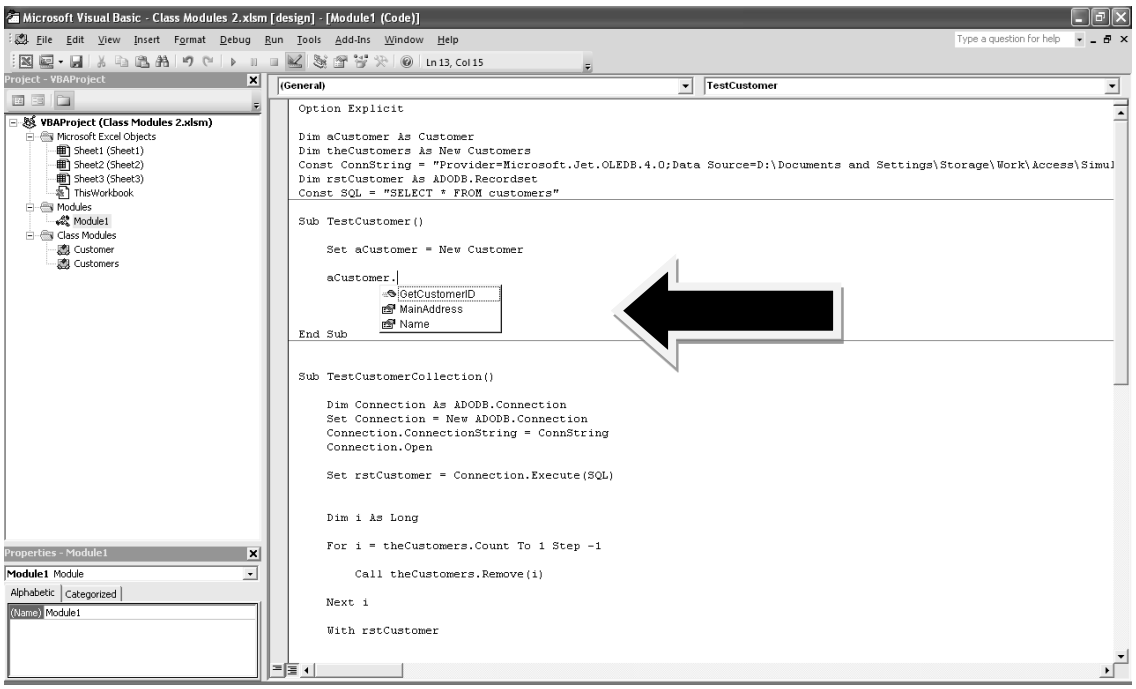
```
Option Explicit

Dim aCustomer As Customer (1)

Sub TestCustomer()

  Set aCustomer = New Customer    (2)

   aCustomer.Name = "Evil Genius" (3)

   aCustomer.MainAddress = "123 the Hollowed out Volcano" (4)

   MsgBox "Company ID is " & vbCrLf & aCustomer.GetCustomerID() (5)

End Sub
```

Line 1 defines an object variable as a Customer variable, and line 2 sets it as a new customer object. Line 3 assigns a value to its name property and line 4 a value to its Main Address property.

Line 4 uses the GetCustomerID Method to generate the CustomerID value and returns it in a message box.

Using IntelliSense™

Microsoft IntelliSense is a convenient way to access descriptions of objects and methods. It speeds up software development by reducing the amount of name memorization needed and keyboard input required. Once a class is defined in a class module, Intellisense will automatically provide drop down lists showing the methods and properties of objects the names of which have been entered into the VBE.



Programming Techniques

Writing effective code is both a science and an art. There are obviously rules of logic and syntax that define what will and will not work. Outside of this however, there are usually a multiplicity of ways to achieve a particular end, and as your experience grows you will find a style that suits you. However there are various conventions and best practises that when followed will make your code simpler and more efficient.

Best Practice for Excel Programming

When writing VBA code for Excel, it is best whenever possible to make use of Excel's built in functionality rather than trying to replicate it in your code. For example, if you were working with a list and wanted to place subtotal within it at the end of various groups, you could:

1. Write code that inserts a blank row at the end of each group and then insert a function to total the rows above it.

2. Alternatively, you could use the subtotalling method of the current region to accomplish the same end.

The later technique is the more efficient as it is briefer, easier to understand and executes more quickly. There are also associated methods for removing the subtotals.

```
Sub SubTotals()

   Range("A6").CurrentRegion.Sort key1:=Range("a7:a41"), Header:=xlYes

   Range("A6").CurrentRegion.Subtotal Groupby:=1, Function:=xlSum, _
TotalList:=Array(6, 11, 16)

End Sub
```

The code above sorts a database, and then inserts subtotals based on the sorted column.