Access VBA Made Easy

Access VBA Fundamentals

Level 2

www.AccessAllInOne.com

This guide was prepared for AccessAllInOne.com by:
Robert Austin

This is one of a series of guides pertaining to the use of Microsoft Access.

© AXLSolutions 2012
All rights reserved. No part of this work may be reproduced in any form, or by any means, without permission in writing.

Contents

03 - Data Types, Variables, Constants and Operators 4
Learning Objectives 4
Variables 4
Declaring variables 4
Dim 5
Note 5
Restrictions on naming variables 5
Naming Conventions 6
Constants 6
Variable Scope 7
Arithmetic Operators 8
Common Errors 9
Not using the Option Explicit Statement 9
Data Types 9
Data types and definition 10
Boolean - (Yes/No) 10
Integer 11
Long 11
Single 11
Double 11
Currency 12
Date 12
String 12
Variant 14
Exercises .... **Error! Bookmark not defined.**
Answers 39
04 - Events 20
Form and Report Events 20
Related Objects 20
How to create an event in the VBA editor 20
Forms, Controls and their events 20
Mouse Events 22
OnClick 22
OnDblClick 24
OnGotFocus and OnLostFocus 25

OnMouseDown, OnMouseUp 27
OnMouseMove 28
OnKeyDown, OnKeyUp 29
OnKeyPress 30
Form Events – OnOpen, OnLoad, OnResize, OnActivate, OnUnload, OnDeactivate and OnClose 31
Recordset Control Events – OnCurrent, BeforeUpdate, AfterUpdate, OnChange 35
OnTimer Events 37
Exercises 38
Answers 39

03 - Data Types, Variables, Constants and Operators

Learning Objectives

After you have finished reading this unit you should be able to:

- Declare a variable
- Say what a data type is
- Say what scope the main data types have
- Declare and instantiate a constant
- Understand naming conventions
- Use arithmetic operators

Variables

When writing code in V.B.A. we often need to do calculations based on values that can change. An example would be a Transaction Processing System that needs to calculate the tax on a sale. Sales tax can change (although not very often) but the price of the item(s) we are adding tax to can and will vary with every transaction. For that reason we use variables in V.B.A.

```
1   Sub getPriceIncVAT()
2
3   Dim ItemPrice As Double
4   Dim SalesTax As Double
5   Dim PriceIncVAT As Double
6
7   ItemPrice = InputBox("What is the price of the item?")
8   SalesTax = InputBox("What is the tax? (20%=0.2)")
9   PriceIncVAT = ItemPrice + (ItemPrice * SalesTax)
10  MsgBox ("The price of the item including VAT is: $" & PriceIncVAT)
11
12  End Sub
```

Figure 3.1

In Figure 3.1 we have 3 variables named ItemPrice, SalesTax and PriceIncVAT that we use to work out the price of an item including V.A.T. The values for ItemPrice and SalesTax the user will be asked to input themselves and the final variable PriceIncVat is a calculated value based on a business rule.

We need to use variables here as the values can and will vary depending on the situation and needs to the business.

Declaring variables

Variable declaration is the act of telling VBA the name of your variables before you actually use them.   You should always declare the variables you will use as soon as logically possible, which usually means at the top of your function or sub procedure. You should also state the data type of your variables (we discuss this later on in this unit). In Figure 4.1 we are telling V.B.A. that we would like to declare a variable called ItemPrice which has a data type double. We do the same for SalesTax and PriceIncVAT.

It is a good idea and standard practice to declare variables and data types

Dim

To declare a variable in VBA we use the keyword **Dim** (which stands for dimension).

```
1   Dim Age As Integer       ' VBA makes space for Age, of type Integer
2   Dim Name as string       ' VBA makes space for Name, of type String
```

Figure 3.2

The name of the variable must follow Dim and the type of the variable *should follow* with the keywords "As" and then the type.

Note
If, at the top of the module, you have the words "Option Explicit" you must let V.B.A. know the data type that you will be assigning the variable (e.g. as Integer, as Double, as String). If, however, you omit "Option Explicit" at the top of the module, you don't have to let V.B.A. know what type of data you are going to use. V.B.A. will assume that you are using the data type "Variant" and proceed accordingly. This is perfectly acceptable but not recommended as the data types are not optimal.

Restrictions on naming variables

The names we can use for variables must conform to a small set of rules:

1. They must begin with a letter or an underscore (_).
2. They must end with a number or letter.
3. They may contain any sequence of numbers or letters or underscores (_).
4. They may contain upper or lower case letters.
5. They must not be one of VBA's keywords.

The compiler will automatically tell you if a variable is illegally named and will not execute unless variables are valid.

```
1   Dim a as String          ' is a valid variable name
2   Dim b_ as String         ' is a valid variable name
3   Dim _b as String         ' variable names must start with a letter
4   Dim 2b as String         ' variable names must start with a letter
5   Dim c1 as String         ' is a valid variable name
6   Dim d12 as String        ' is a valid variable name
7   Dim e_e1 as String       ' is a valid variable name
8   Dim f! as String         ' punctuation not allowed in variable names
9   Dim g as String          ' is a valid variable name
10
11  Dim dim as String        ' is not valid – "Dim" is a keyword
12  Dim string as String     ' is not valid – "String" is a keyword
13  Dim number as String     ' number is not a keyword so this is valid
```

Figure 3.3

Naming Conventions

A naming convention is a way of naming variables which enables us to easily understand both the data type of the variable and the reason for its existence. There are a couple of rules to follow when naming variables.

- Use meaningful variable names – make your variables mean something. Zzxd isn't meaningful, but fileNotFound means something to a human, even though it doesn't affect the computer or VBA in any way.
- Use camelCase for variables – that is, for every word in your variable name make the first letter is upper-case, except the first letter of the first word. thisIsCamelCase .
- Use UPPER_CASE for constants – when you declare a constant the name of that constant is usually capitalised. This means nothing to the compiler but means everything to you (we look at constants later on in this unit).

Another convention is to use up to 3 small letters before the variable name to indicate the data type.

- iMyNumber would be of type Integer
- dblMyOtherNumber would be of type Double
- strText would be of type String

Constants

Constants differ from variables in that their value does not change after they have been declared. This is how we code with constants:

```
1   Dim a as String          ' is a regular variable declaration
2   Const B = 1              ' declare the constant B with a value of 1
3   Const DATABASE_NAME = "accdb_firsttime"
4                            ' new constant called DATABASE_NAME
```

Figure 3.4

You may have noticed that constants are not given a data type; this is because VBA makes some intuitive assumptions about the data. For example, any text surrounded by double quotation marks is a String; any number without a decimal point will fit into a Long; any decimal number will fit into a Double, and any True or False values fit into a Boolean value (True or False). This is the same logic VBA will take if you were not to define your data type on a variable using Dim.

Variable Scope

When you declare a variable in your program you also implicitly determine which parts of your code can access it. In VBA there are three types of declaration that affect the scope of a variable; Procedure, Module and Public.

```
1   Option Explicit
2
3   'Global Declaration
4   Public SalesTax As Double
5
6   'Module Level Declaration
7   Private ItemPrice As Double
8
9   Sub getPriceIncVAT()
10  Dim PriceIncVAT As Double
11  Call getSalesTax
13  Call getItemPrice
14  PriceIncVAT = ItemPrice + (ItemPrice * SalesTax)
15  MsgBox ("The price of the item including VAT is: $" & PriceIncVAT)
16  End Sub
17
18  Sub getSalesTax()
19  SalesTax = InputBox("What is the tax? (20%=0.2)")
20  End Sub
21
22  Sub getItemPrice()
23  ItemPrice = InputBox("What is the price of the item?")
24  End Sub
```

Figure 3.5

Procedure Level Scope

Procedure level scope means that a variable is recognised only within that procedure. In Figure 4.5 the variable PriceIncVAT has a procedure level scope and is only recognised within the sub-procedure getPriceIncVAT. To achieve this we use the dim or static keywords and declare the variable *inside* the sub procedure we wish to recognise it.

Module Level Scope

Module level scope means that a variable can be recognised by any sub procedures within the module. In Figure 4.5 ItemPrice has a module level scope and this is reflected in the fact that the variable is recognised in getItemPrice and getPriceIncVAT. To give a variable a module scope we declare it at the top of the module and use the private keyword (private means it is only available to sub procedures within the module it is declared in).

Public Level Scope (also known as Global Scope)

Public level scope means that a variable is recognised by every sub-procedure and function within the active application. (In Figure 4.5 SalesTax has a public level scope.) This can be useful for variables that should be consistent throughout the application (e.g. SalesTax shouldn't be 20% in one sub procedure and %15 in another). It is convention to create a module with a name like "Globals" where it is possible to keep all of the public variables in one place where they can easily be maintained and modified as required.

Arithmetic Operators

Like all languages VBA has a set of operators for working on Integer (whole) and floating-point (decimal) numbers. The table below demonstrates all 9 of them. VBA also offers many other operations built in as commands in the language.

```
1
2   Sub ArithmeticOperators()
3
4   Dim a1 As Integer
5   Dim b1 As Integer
6   Dim c1 As Integer
7
8   Dim a2 As Double
9   Dim b2 As Double
10  Dim c2 As Double
11
13  ' + addition
14  a1 = 10
15  b1 = 20
16  c1 = a1 + b2  ' c1 = 30
17
18  ' - subtract
19  a2 = 9.8
20  b2 = 5.3
21  c2 = a2 - b2  ' c2 = 4.5
22
23  ' * multiplication
24  a1 = 9
25  b1 = 8
26  c1 = a1 * b1 ' c1 = 72
27
28  ' / division floating-point
29  a2 = 120.5
30  b2 = 8.12
31  c2 = a2 / b2 ' c2 = 14.8399014778325
32
33  ' \ division integers
34  a1 = 256
35  b1 = 8
36  c1 = a1 \ b1 ' c1 = 32
37
38  ' mod - returns the remainder of a division
39  a1 = 100 Mod 3    ' a1 = 1
40  a2 = 100 Mod 3.1  ' a2 = 1, mod only returns whole numbers
41
42
43  ' ^ powers
44  a1 = 2 ^ 2      ' a1 = 4
45  b1 = 3 ^ 3 ^ 3  ' b1 = 19683
46  End Sub
```

Figure 3.6

Common Errors

Not using the Option Explicit Statement

The option explicit statement is useful because it ensures that we *must* declare our variables. (As mentioned, VBA assumes the data type variant for all non-declared variables when option explicit isn't used).

The Option Explicit statement should go at the top of the application before any code has been written.

Data Types

When working with data in V.B.A. it is important that we don't try to add 15 to the word "Hello" or try to divide 07/02/12 by 53 as V.B.A. will not be able to make sense of these calculations (and, frankly, neither can we). In order to ensure the integrity of the data that we make calculations upon we are required to use data types.

Data types are essentially restrictions that are placed on data that are manipulated in the V.B.A. environment. These restrictions allow us to tell V.B.A. that we are creating a variable and that variable will only accept a specific type of data. An example of this would be the integer data type. Integer is just a fancy way of saying whole number and if we declare a data type as an integer it will only accept a whole number.

```
Dim HouseNumber as integer ‘ This variable will only accept integers
```

Figure 3.7

Here we have created a variable called HouseNumber and informed V.B.A. that we wish this variable to be of type integer. This means that we will only be able to assign a whole number to it. Ergo…

```
Dim HouseNumber as integer

HouseNumber = 5
```

Figure 3.8

…would be a perfectly acceptable assignment statement whilst…

```
Dim HouseNumber as integer

HouseNumber=”Car”
```

Figure 3.9

…would not.

There is also another reason for the existence of data types. Different data types take up different amounts of memory depending on how complex they are. An example of this would be the integer data type vs the double data type.

We can use the double data type to store non-integer numbers. So, for example, whereas we couldn't accurately store the number 2.531 as an integer (it would round it up to 3) we could use the double data type for this value. The double data type, though, uses twice as much memory as the integer data type (8 bytes vs. 4 bytes) and, although not a big drain on the memory, with large applications that use many variables it can and will affect performance if the correct data are not assigned the correct data type. So, if you need to store integers, use the integer data type; if you need to store text strings, use the string data type. And so on.

Data types and definition

Firstly a word on VBA variable names; a variable may be named anything you wish as long as it conforms to VBA's naming rules. Variable names must start with a letter, may contain number characters or underscores (_) but that's it! Punctuation marks are not allowed. Also unlike other languages VBA is **case-insensitive**! This is important to understand and is demonstrated below.

Finally, there are some keywords that cannot be used as variable names.

```
1   Dim a as String          ' a valid variable name
2   Dim b_ as String         ' a valid variable name
3   Dim _b as String         ' variable names must start with a letter
4   Dim 2b as String         ' variable names must start with a letter
5   Dim c1 as String         ' a valid variable name
6   Dim d12 as String        ' a valid variable name
7   Dim e_e1 as String       ' a valid variable name
8   Dim f! as String         ' punctuation not allowed in variable names
9   Dim g as String          ' a valid variable name
10  Dim G as String          ' an invalid variable name.  VBA is case-
11                           '   insensitive, variables also cannot be
12                           '   declared more than once in a code block
13  Dim aVariableName as String    ' a valid variable name
14  Dim a_Variable_Name as String ' a valid variable name
15  Dim HELLOWORLD as String       ' a valid variable name
16
17  Dim dim as String        ' variable name is invalid as Dim is a keyword
    Figure 4.10
```

Figure 3.10

Boolean - (Yes/No)

A variable of type Boolean is the simplest possible data type available in VBA. It can only be set to 0 or -1. These are often thought of as *states* and correspond to Access's Yes/No fields. In VBA you can assign a Boolean variable to True (-1) or False (0) or the numbers indicated in the brackets.

Notice I used capitalised words for True and False, which is because they are VBA keywords and you cannot call a variable a Keyword.

```
1   Public Sub trueOrFalse()
2   Dim foo As Boolean
3   Dim bar As Boolean
4   foo = True         ' foo holds the value True
```

```
5   bar = False      ' bar holds the value False
6   End Sub
```

Figure 3.11

Integer

At the beginning of the section we said that we have to tell the computer what type of data to expect before we can work on it. An Integer is another number data type, but its values must be between -32,768 and 32,767, and they must be whole numbers, that is to say, they mustn't contain decimal places. If you or your users try to save a decimal value (eg 2.5) to an integer variable, VBA will round the decimal value up or down to fit into an Integer data-type.

```
1    Sub IntegerDataType()
2    Dim foo As Integer
3    Dim bar As Integer
4    Dim oof As Integer
5
6    foo = 12345      ' foo is assigned the value 12,345
7    bar = 2.5        ' bar is assigned the value 3 as VBA rounds it up
8    bar = 2.4        ' bar is assigned the value 3 as VBA rounds it down
9    foo = 32768      ' causes an overflow error as 32,768 is too big
10   End Sub
```

Figure 3.12

Long

Long is another number type and works just like Integer except it can hold a much larger range; Any number between -2,147,483,648 and +2,147,483,647.

```
1   Sub LongDataType()
2   Dim foo As Long
3   foo = 74345      ' foo is a variable assigned the value 74,345
4   End Sub
```

Figure 3.13

Single

Single is the smaller of the two "floating point" data types. Singles can represent any decimal number between -3.4028235E+38 through 1.401298E-45 for negative numbers and 1.401298E-45 through 3.4028235E+38 for positive numbers. Put more simply, the single data type has a decimal point in it.

```
1   Sub DoubleDataType()
2   Dim foo As Single
3   Dim bar As Single
4   foo = 1.1        ' foo keeps the .1 decimal part
5   bar = -20.2      ' bar also keep the decimal part
6   foo = foo * bar ' foo equals -22.2200008392334
7   End Sub
```

Figure 3.14

Double

This is a "floating point" number as well and range in value from -1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values and from 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values.

```
1   Sub DoubleDataType()
2   Dim foo As Double
3   Dim bar As Double
4   foo = 1.1        ' foo keeps the .1 decimal part
5   bar = -20.2      ' bar also keep the decimal part
6   foo = foo * bar ' foo equals -22.2200008392334
7   End Sub
```

Figure 3.15

Currency

This data-type is a third "floating-point data" type in disguise. It's a Single which has been engineered to represent behaviours typical of currencies. In particular it rounds off numbers to four decimal places. See the Figure below:

```
1   Sub CurrencyDataType()
2   Dim bar As Single
3   Dim foo As Currency
4   bar = 1.1234567      ' this is the Single
5   foo = bar            ' add the Single to the Currency
6   MsgBox bar      ' bar contains 1.1234567
7   MsgBox foo      ' foo contains 1.1235. Notice that the 4th digit
8                    '   has been rounded up to 5
9   End Sub
```

Figure 3.16

Date

The Date data type is used to perform operations that involve dates AND times. In VBA there are several functions that operate on date variables which perform date and time calculations. It is important to know that date and time operations can be quite complicated and to help ease your burden you can use VBA's DateTime object which encapsulates a lot of the difficulty of working with dates and time and can make them *a little less of a headache* to deal with. Date data types are the most complicated of all the data types to work with.

Here are a few operations we can do with date data types.

```
1    Sub DateDataTypes()
2    Dim bar As Date
3    Dim foo As Date
4    bar = #11/15/1978#                 ' bar set to this date but has no time
5    foo = #12/10/2012 11:37:00 PM#     ' foo is set to this date and time
6    bar = #1:00:09 AM#                 ' bar is 1 hour and 9 seconds
7    foo = #9:00:00 PM#                 ' foo is 9PM
8    foo = foo + bar                    ' foo is now 22:00:09
9    MsgBox foo
10   foo = foo - bar                    ' foo is back to 9PM
11   MsgBox foo
12   End Sub
```

Figure 3.17

String

A String is any set of characters that are surrounded by double-quotation marks. For example "dog" is a String that contains three characters. Strings are very important to us as they can contain human language, and in fact contain almost any data we want, even numbers and punctuation marks. Strings are very versatile and you will use them extensively in your code. Often when you ask your users for information you will first store

their input into a String before actually using the data provided; in this way Strings are often thought of as a safe data type.

Below are some Figures of Strings in action.

```
1   Sub StringDataTypes()
2   Dim bar As String
3   Dim foo As String
4   Dim foobar As String
5
6   bar = "Hello"                  ' bar now contains "Hello"
7   foo = "world!"                 ' foo contains "world!"
8   foobar = bar & " " & foo       ' foobar now contains "Hello world!"
9   ' notice that foobar has a +" "+ this means a SPACE character has been
10  ' inserted into the String, without it foobar would contain
11  "Helloworld!".
12  foobar = bar + " " + foo       ' This Figure also shows that you can add
13  ' Strings together (but you cannot subtract!)
14  foo = "H" & "E" & "L" & "P"    ' foo now contains "HELP"
15  bar = foo & foo                ' bar now contains "HELPHELP"
16  End Sub
```

Figure 3.18

As stated above, when you collect input from a user you will usually collect it into a String. But be careful not to confuse String with Number data types. For example:

```
1   Dim bar, foo As String
2   Dim foobar As String
3
4   foo = "12.5"         ' user inputs "12.5"
5   bar = "6.3"          ' user inputs "6.3"
6   foobar = foo * bar   ' we multiple 12.5 and 6.3
7   Debug.Print foobar   ' print the result
8   0                    ' It's ZERO!
9
10  ' Remember foo and bar are STRING data types, so multiplying foo and
11  bar as above is like saying "aaa" * "bbb" = ? It doesn't make sense.
12  But we collect data in a String because a String can accept all user
13  input, even if they put a punctuation mark in there.
14
15  foo = "12.5.2"       ' user made a mistake
16  bar = "ifvgj212m"    ' cat walks across the keyboard
17
18  ' When collecting user input the data held in a String can be tested
19  for accuracy and correctness before we load it into an Integer. If the
20  user has not entered data correctly we ignore or display a useful
21  message like "Error"...
```

Figure 3.19

One last thing to mention about Strings. In VBA a String is a “primitive” or simple data type and cannot be accessed as an array, so foo[0] will cause an error. Also a String is “naturally” of variable length and the length need not be specified in the Dim statement. More on these topics later.

Variant

A variant is a special type which can contain any of the data types mentioned above (along with some others).

When a value is assigned to the variant data type the variable *mutates* into the type of the data assigned to it, and in some cases VBA can "detect" the type of data being passed and automatically assign a "correct" data type. This is useful for collecting data from users and also for writing procedures and functions for which you want to be able to call with a variable of any type.

```
1   Sub VariantDataType()
2   Dim bar As Variant
3   Dim foo As Variant
4   Dim foobar As Variant
5   bar = 1              ' bar is now an Integer
6   foo = "oi!"          ' foo is now a String
7   foobar = bar + 1.1   ' foobar is now a Double with the value of 2.1
8   MsgBox TypeName(bar)    ' Integer
9   MsgBox TypeName(foo)    ' String
10  MsgBox TypeName(foobar) ' Double
11  End Sub
```

Figure 3.20

Questions

With Option explicit set in all your modules answer the following questions.

Write each answer in a function called myAnswer_ and end if it your question number. For example:

```
1   Option Compare Database
2   Option Explicit          ' make sure this line is in your code
3
4   Function myAnswer_1()
5
6          ' your code goes here
7
8   End Function
```

Figure 3.21

1. Write code that will declare the following types and set them all to a value
   a. boolean
   b. integer
   c. long
   d. date
   e. single
   f. double
   g. currency
   h. string
   i. date
   j. a Variant String

2. write code that performs the following:
   a. declares a constant with the value "Hello".
   b. declares another constant called YEAR with the value 2012.
   c. declare a variable called myName and assign it your name.
   d. declare a second variable called myMesage and join the constant in (a) with the variable in (c).
   e. now add to myMessage the text ". The year is".
   f. now add to myMessage the constant YEAR (hint: function cstr()).
   g. finally add the following:
      debug.print myMessage

3. What is the output of the following sub?

```
1   Option Compare Database
2   Option Explicit          ' make sure this line is in your code
3
4   Sub myAnswer_3()
5
6          iVar1 = 10
7          iVar2 = "value of iVar1=" + iVar1
8          msgbox iVar2
```

```
    End Sub
```

Figure 3.22

4. What will the next code sequence do and why?

```
1   Option Compare Database
2   Option Explicit          ' make sure this line is in your code
3
4   Sub myAnswer_3()
5         Dim iAnswer as Integer
6         iAnswer = "42"
7
8   End Sub
```

Figure 3.23

5. What is the difference between the following?
    a. A1 = "42.2"
    b. A2 = 42.2
    c. And what would be the result of A1 * A2?

6. You start your code with the following instruction. Why does it not compile?

```
1   Option Compare Database
2   Option Explicit          ' make sure this line is in your code
3
4   Sub myAnswer_3()
5         Dim function as String
6         function = "Hello World!"
7         msgbox function
8
    End Sub
```

Figure 3.24

7. Rewrite the following in camelCase
    a. Calculate the age of a tree
    b. Tape reader file position
    c. User input
    d. File pointer
    e. Input
    f. sMyMessage

8. rewrite the following as constants
    a. semaphore stop
    b. semaphore start
    c. semaphore paused
    d. file open
    e. end of file
    f. carriage return and line feed
    g. new line

9. In a new function write code to do the following:
    a. Define a global constant called database name and give it the value "mysqldb_website1"
    b. Define another global variable with a meaningful name to hold an IP address, eg 127.0.0.1
    c. In local scope, declare a variable named sDBDetails adding the value of the constants from (a) and (b) making sure to add a space between them
    d. Add the following code and execute your code from the immediate window
        i. Msgbox sDBDetails

10. In a new function perform the following mathematical expressions by first assigning the numbers to letters and then saving the result into z, for example:
    a. 12 + 16, would be
        i. Dim a, b, z as integer
        ii. a=12
        iii. b=16
        iv. z=a+b
        v. debug.print z
    b. 100+1+20+2
    c. 76 * 89
    d. (-50 * 3 * -1) / 10
    e. 10 mod 3
    f. 2 to the power of 2 to the power of 2
    g. 2.5 * 7.6, make sure to preserve the decimal number
    h. Assign to an integer the value 2.7 . What is the integer's value?
    i. Concatenate the following Strings with addition spaces between
        i. "Winston, you are drunk! "
        ii. "Yes madam, and you are ugly!"
        iii. "But in the morning, I shall be sober"
    j. What is the square of 27 to the nearest whole number

11. What is displayed in the pop-up message box?

```
1   Option Compare Database
2   Option Explicit         ' make sure this line is in your code
3
4   Sub myAnswer_11()
5         Dim s as String
6         s = "I" + " like " + "Chinese food!"
7         s = s + " The wai-ters never are rude."
8         msgbox s

    End Sub
```

Figure 3.25

12. What must you do to make the following code work?

```
1   Option Compare Database
2   'Option Explicit        ' make sure this line is in your code
3
4   Sub myAnswer_3()
5         Dim d as Date
6         d = 12 Dec 2012
7         msgbox d
8
    End Sub
```

Figure 3.26

13. In a new function assign the following dates to variables
    a. 11 November 1918
    b. 3 December 1992
    c. 18 October 1871
    d. 10 30 PM
    e. 12 – 12 – 2012 00:21
    f. 1969, July, 20th

14. Declare three variant variables, set their values to a person's name, any integer value and any floating-point number, respectively.

15. Write code to answer the following expressions:
    a. 20-True
    b. True+ True+ True-False
    c. (7656 mod 7) / 3
    d. 12 + 66 / 11
    e. #12-dec-2012# + #01/01/01#

16. Explain the differences between a Long number and a floating point number.

17. Explain why "10:26 PM" and #10:26 PM# are not the same?

18. If you are asking the user for their birth date, which data type would you / could you temporarily store their answer for further checking?

19. True or false:

    a. 20-20 = true?
    b. True and true = false?
    c. False or true = true?

20. Which of the following are valid variable names
    a. aVariable
    b. aFunction

c. end
d. while
e. STATE_HOLD
f. STATE OVER
g. File1
h. outputFile_10
i. input-file2
j. $helloWorld
k. 9LivesACatHas
l. todayisyesterday
m. Tomorrow NeverComes

04 - Events

Form and Report Events

An event is any interaction that a human has with the application or when parts of the application change state, which is invariably because the human has requested something. Usually this will involve the user clicking a button or entering some text but can also involve touching the screen, just leaving the mouse curser over a box or form, tabbing around, cycling through records or part of a chain of events.

The events that we will be concentrating upon in this unit are those associated with Access forms; reports also have some of these events and operate in the same way but forms are interactive mediums whilst reports format data in a static report like fashion.

Related Objects

Please open up the CodeExamplesVBAForBeginners application. The objects we will be using will be frmEvents, frmStudentsDataEntry and frmTimer.

How to create an event in the VBA editor

Modules for forms are automatically created by Access when we click on the ellipsis in the Properties | Events tab.

The form must be open in design view when you first create an event.

All events that a Form or Object can react to are in the Events tab.

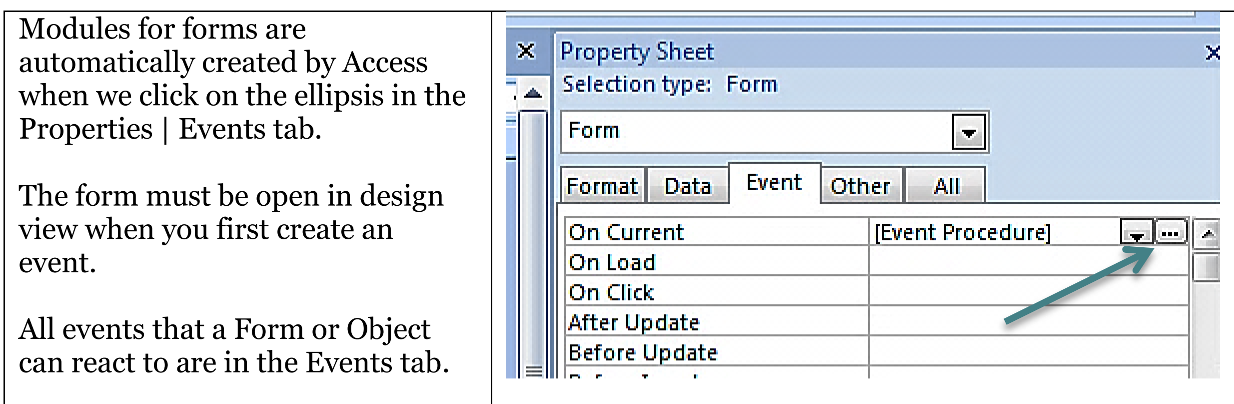

Figure 4.1

In figure 4.1 an On Current event already exists. We know this because [Event Procedure] is written in the On Current field of the property sheet.

Forms, Controls and their events

Forms are not simple objects. They are made up of a Header, a Detail, a Footer and the Form itself. Each of these parts of the form has their own set of events which you can see change as you click on each of these parts of on the form. The little square in the top left is the form itself. You can add controls to the Header, Detail and Footer areas.

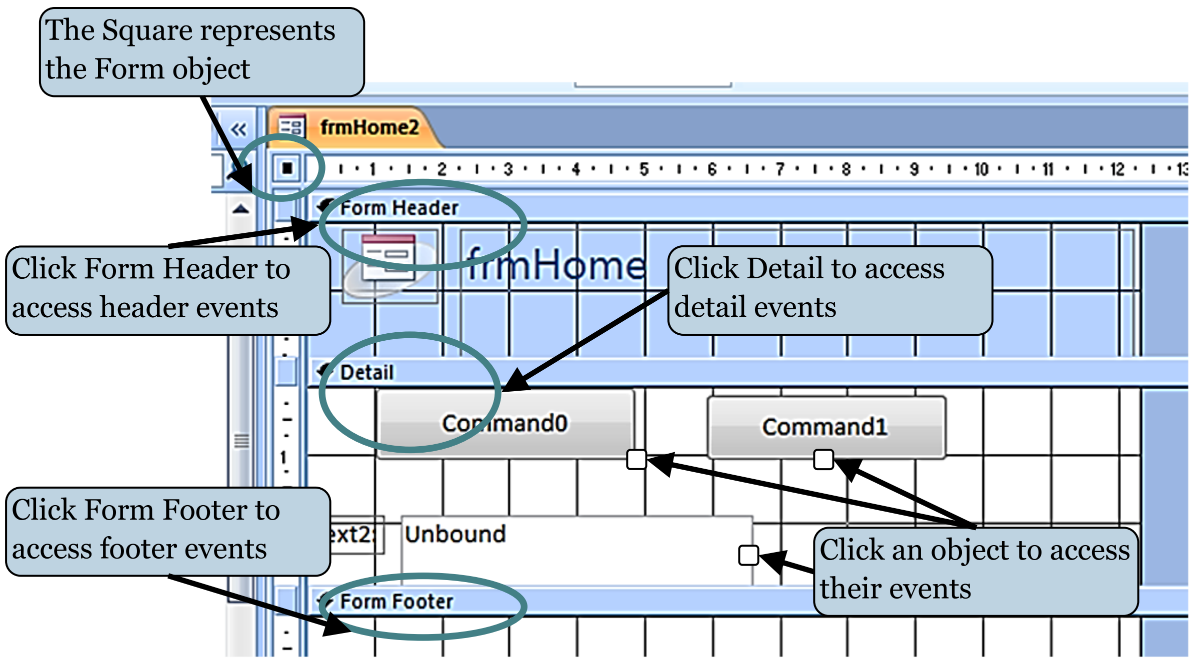

Figure 4.2

Note
Although the form is broken down into several parts the vast majority of the time you will be dealing with events related to opening the form, closing the form and events for different controls (combo-boxes, text-boxes, command buttons) that are usually found in the detail section of the form. This has been reflected in the material for this unit.

Mouse Events

The main mouse events occur when you click an object such as a section of the form or a control. A click event actually consists of a MouseDown, MouseUp, MouseClick and MouseDblClick. These can then also trigger another set of events LostFocus, GetFocus, Enter, Exit .

Please open frmEvents

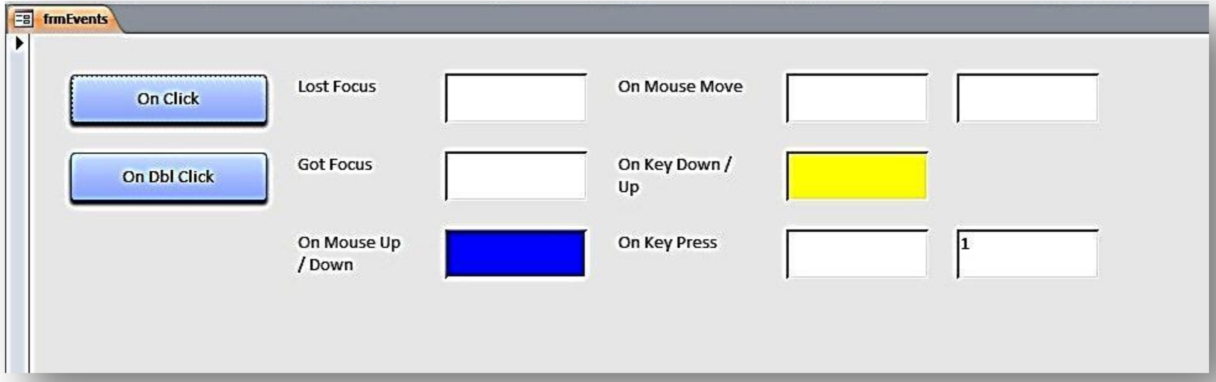

Figure 4.3

Using frmEvents we have set up several controls to demonstrate what certain events are triggered by and how they behave.

OnClick

The OnClick occurs when a Control object is clicked.. This event is most commonly associated with a command button but can also be used with controls such as text-boxes and combo-boxes.

To get to the code associated with the OnClick event of cmdOnClick button we open the form in design view, select cmdOnClick in the property sheet and click on the ellipsis on the far right hand side.

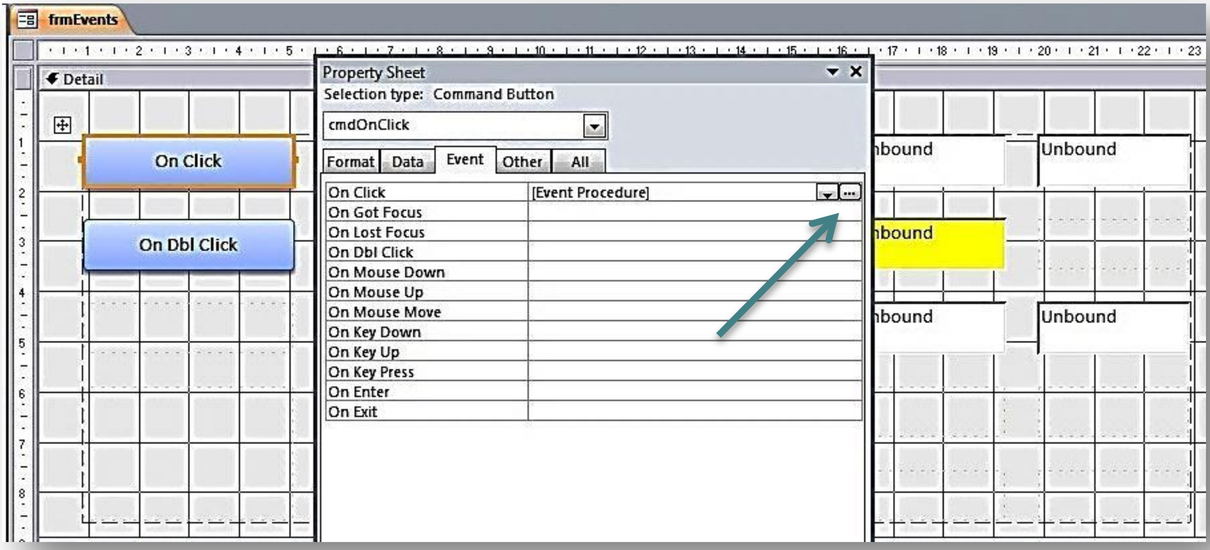

Figure 4.4

The VBA editor will open up with all the procedures related to that form on display. The curser should be flashing in *Private Sub cmdOnClick_Click()*.

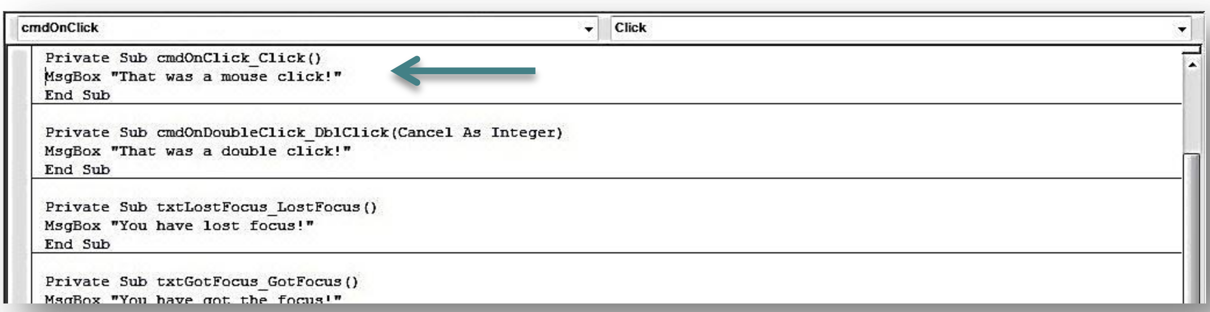

Figure 4.5

The code associated with cmdOnClick is displayed in Figure 4.6

```
1   Private Sub cmdOnClick_Click()
2   MsgBox "That was a mouse click!"
3   End Sub
```

Figure 4.6

Go back to frmEvents, change it to Form view and click the button to see what happens.

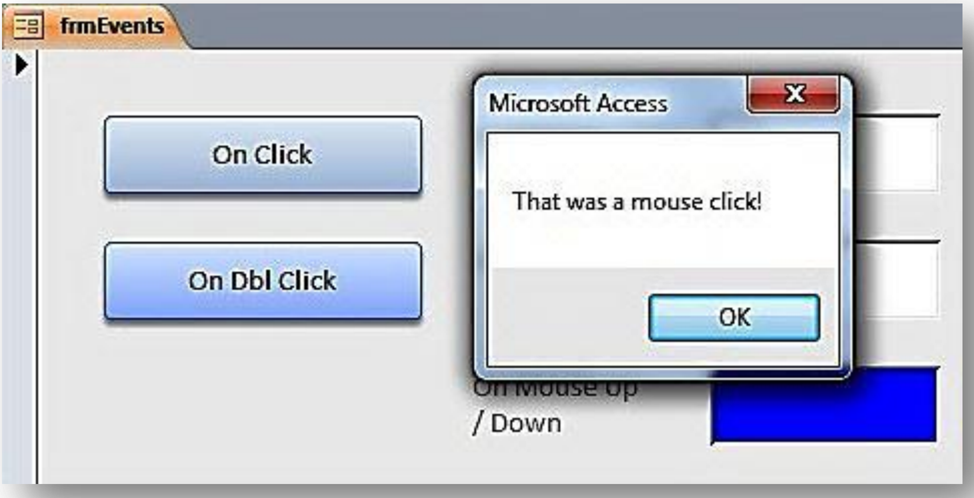

Figure 4.7

The OnClick event fired and the statement *MsgBox "That was a mouse click!"* was executed.

OnDblClick

The double click event occurs when the system identifies that the user has double-clicked an object.

Here is the code associated with the double click event for the cmdOnDoubleClick button.

```
1   Private Sub cmdOnDoubleClick_DblClick(Cancel As Integer)
2   MsgBox "That was a double click!"
3   End Sub
```

Figure 4.8

Double click cmdOnDoubleClick and this is what you should see:

Figure 4.9

OnGotFocus and OnLostFocus

The Got Focus event occurs when a control receives the focus. This can be either by clicking the control or tabbing into it. If a text-box receives the focus the curser flashes inside it whereas when a button receives the focus you can just make out a faint dotted line around the edge.

Figure 4.10

In figure 4.10 the On Dbl Click button has the focus and the dotted line is just visible.

We can trigger the OnGotFocus event of txtGotFocus by either clicking into txtGotFocus or tabbing over from cmdOnDblClick. Either way the OnGotFocus event will produce this reaction:

Figure 4.11

```
1   Private Sub txtGotFocus_GotFocus()
2   MsgBox "You have got the focus!"
3   End Sub
```

Figure 4.12

The code associated with the OnGotFocus event is displayed in Figure 4.12.

The OnLostFocus event triggers when a control loses the focus. If the focus is on a button (cmdOnDoubleClick) and you tab or click into txtOnGotFocus, cmdOnDoubleClick loses the focus right before txtOnGotFocus gets the focus.

To demonstrate this concept click into txtOnLostFocus (not txtOnGotFocus). The curser should be flashing within the text-box. Now click into txtOnGotFocus. You should see two messages come up one after another. The first will read:

Figure 4.13

And the second will read:

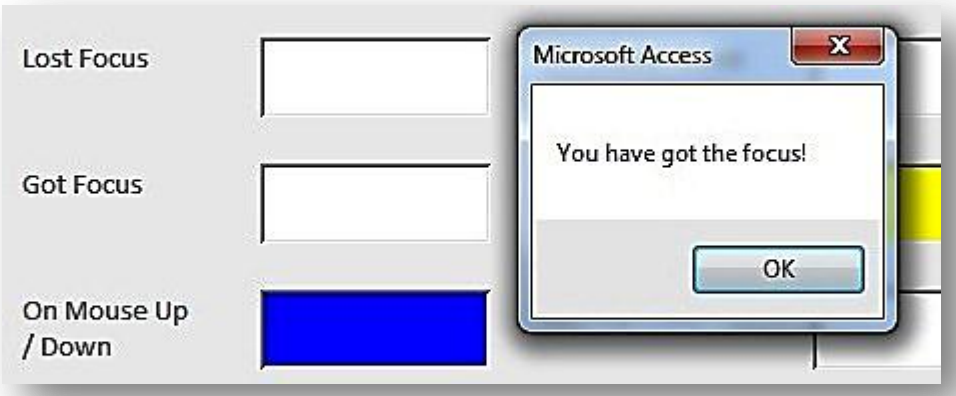

Figure 4.14

What has happened is that the first event to fire was the OnLostFocus event of txtOnLostfocus which brought up the message box in Figure 4.13 and second event to fire was the OnGotFocus event of txtOnGotFocus which brought up the message box in Figure 4.14.

OnMouseDown, OnMouseUp

Although the OnClick event represents the simple clicking of a mouse, it is actually possible to break it down into two separate events; the OnMouseDown event and the OnMouseUp event. The OnMouseDown event is fired when the mouse button is depressed and the OnMouseUp event is fired when the button is released. Before we go to frmEvents to test it out, have a look at the code associated with the two events. In this case we are using both these events on one control – txtOnMouseUpDown.

```
1   Private Sub txtOnMouseUpDown_MouseDown(Button As Integer, Shift As
2   Integer, X As Single, Y As Single)
3   Me.txtOnMouseUpDown.BackColor = vbRed
4   End Sub
5
6   Private Sub txtOnMouseUpDown_MouseUp(Button As Integer, Shift As
7   Integer, X As Single, Y As Single)
8   Me.txtOnMouseUpDown.BackColor = vbBlue
9   End Sub
```

Figure 4.15

Try and work out from the code in Figure 4.15 what is going to happen when the two events fire.

Note

The arguments that the OnMouseDown and OnMouseUp events take may seem complicated but are anything but.

- *Button refers to which mouse button was pressed or released to cause the event to trigger.*
- *Shift refers to whether any of the SHIFT, CTL or ALT keys were depressed at the time the event fired.*
- *X and Y refer to the mouse coordinates.*

We will be using the X and Y arguments when discussing OnMouseMove later on.

When the mouse button is depressed the BackColor property of txtOnMouseUpDown changes to VbRed:

Figure 4.16

When the mouse button is released the BackColor property of txtOnMouseUpDown changes to VbBlue.

Figure 4.17

Press and release the mouse button slowly to really see the difference between the two events.

OnMouseMove

The OnMouseMove event corresponds to the mouse curser hovering over a control that contains that event procedure. The clicking of buttons makes no difference as it is merely the position of the curser that is important.

In form events there is a text-box named txtOnMouseMove. This text-box has the OnMouseMove event procedure and the code looks like this:

```
1   Private Sub txtOnMouseMove_MouseMove(Button As Integer, Shift As
2   Integer, X As Single, Y As Single)
3   Me.txtOnMouseMoveCoordinates.Value = X & " " & Y
4   End Sub
```

Figure 4.18

txtOnMouseMoveCoordinates is the text-box immediately to the right of txtOnMousemove and *X* and *Y* refer to the coordinates of the mouse. What do you think will happen when you hover the mouse curser over txtOnMouseMove?

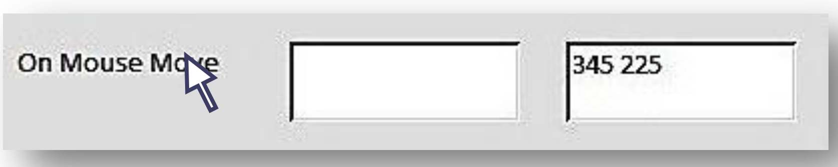

Figure 4.19

As you hover the mouse cursor over txtOnMouseMove the *X* and *Y* coordinates are being displayed in txtOnMouseMoveCoordinates and as you move the position of the curser, the coordinates change.

OnKeyDown, OnKeyUp

The OnKeyDown and OnKeyUp events are very similar to the OnMouseDown and OnMouseUp events but are triggered by the depressing and releasing of certain keys. On frmEvents we have a text-box called txtOnKeyUpDown where we will be testing out the two events. Before testing out the events let's take a look at the code behind the text-box.

```
1   Private Sub txtOnKeyUpDown_KeyDown(KeyCode As Integer, Shift As
2   Integer)
3   Me.txtOnKeyUpDown.BackColor = vbGreen
4   End Sub
5
6   Private Sub txtOnKeyUpDown_KeyUp(KeyCode As Integer, Shift As Integer)
7   Me.txtOnKeyUpDown.BackColor = vbYellow
8   End Sub
```

Figure 4.20

What do you think will happen when you press a key within the txtOnKeyUpDown text box?

Let's say you were pressing the *ctrl* key (the key pressed doesn't matter in this example as we are merely interested in firing the event).

Figure 4.21

After pressing the *ctrl* key the BackColor property of txtOnKeyUpDown changes to vbGreen (Figure 4.21).

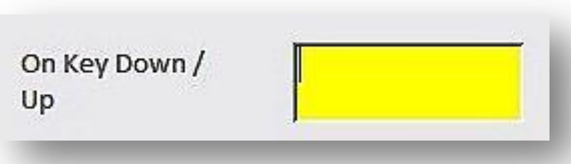

Figure 4.22

After releasing the *ctrl* key the BackColor property of txtOnKeyUpDown changes to vbYellow (Figure 4.22).

If you press and hold a key it will repeatedly fire the OnKeyDown event (along with the OnKeyPress)event.

OnKeyPress

The OnKeyPress event is very similar to the OnKeyDown event with the main exception being that the key that is pressed must return a character. In the examples illustrated in Figures 4.21 and 4.22 pressing the *ctrl* key would *not* trigger the onKeyPress event.

If you click into txtOnKeyPress and start tapping keys you will notice that txtOnKeyPressCounter increments by 1 (until it reaches 100) if the key pressed returns a charcter.

Form Events - OnOpen, OnLoad, OnResize, OnActivate, OnUnload, OnDeactivate and OnClose

Opening a Form

There are two types of form specific events, the first being those associated with the graphical user interface, and the second associated with data and recordsets.

When a form is opened or closed there are a number of stages which a form goes through in order to be capable of displaying itself. The following are the states and each has an associated event:

When the form is opening: **Open →Load →Resize →Activate →Current**

When the form is closing: **Unload →Deactivate →Close**

The Open Event is the first to be fired. In this event you can check whether data exists in the database for the form to work with, and if it doesn't you can Cancel = True to prevent the form from opening.

The Load Event is significantly different from the Open Event in that it cannot be cancelled.

The Resize Event deals with positioning of controls on the form. It is also called whenever the form is minimised, resized, moved, maximised or restored.

The Activate Event is associated with the GetFocus event except Activation is to windows (forms, reports and dialog boxes) what focus is to controls. You may want your code to refresh its view of the recordset in case any data has been updated since it was last active.

The Current Event occurs when the form is ready and retrieves data from the underlying recordset. This event is also the first step the form takes in its efforts to handle recordset data.

Please open up frmStudentsDataEntry. We will be using the immediate window to help us ascertain the correct order of events . To open the immediate window you:

- Click on the view drop-down box

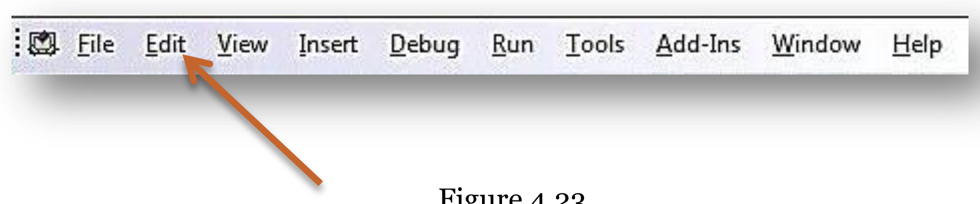

Figure 4.23

- Choose Immediate Window

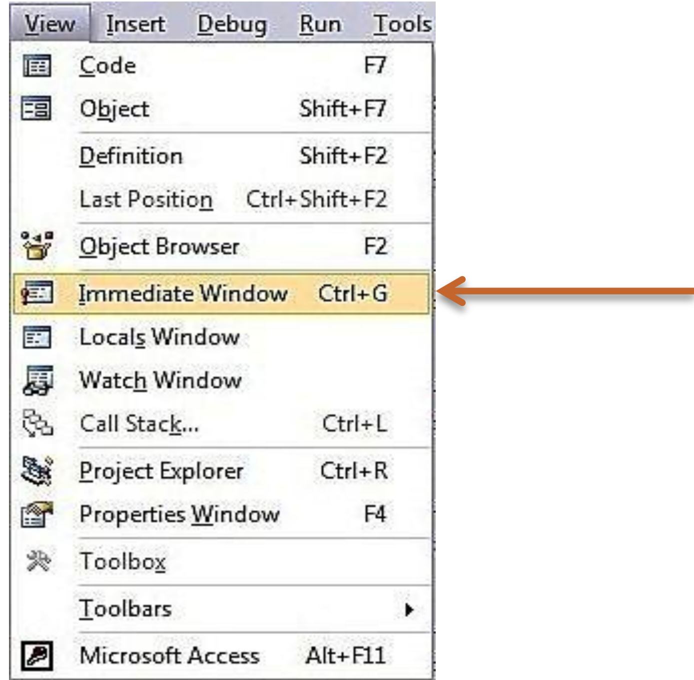

Figure 4.24

- It should be visible at the bottom of your screen (the immediate window can be docked in many different places but it is typical to have it docked below the code window)

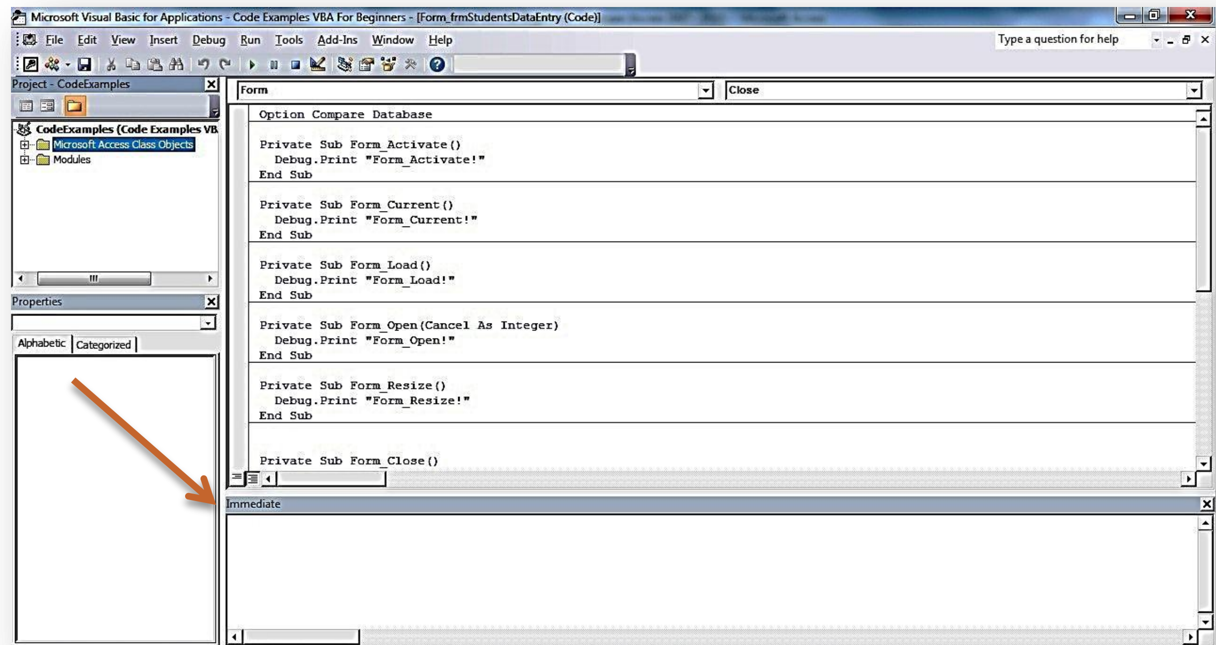

Figure 4.25

Note
The immediate window is a tool that can be used for debugging purposes and to call sub procedures and functions. We will be discussing the immediate window in much more detail in a later unit. For now, you just need to know that when you write Debug.Print in a subprocedure or function, whatever follows will be printed to the immediate window. Ergo, Debug.Print "Form_Activate!"will print Form_Activate! In the immediate window. We will be using this technique to demonstrate the order in which form events are fired.

Here is the code for all the events associated with the opening of a form. If you select Form_frmStudentsDataEntry from the Object Explorer (window in top right of screen) of the VBA editor you will see this code.

```
1     Option Compare Database
2
3     Private Sub Form_Activate()
4       Debug.Print "Form_Activate!"
5     End Sub
6
7     Private Sub Form_Current()
8       Debug.Print "Form_Current!"
9     End Sub
10
11    Private Sub Form_Load()
12      Debug.Print "Form_Load!"
13    End Sub
14
15    Private Sub Form_Open(Cancel As Integer)
16      Debug.Print "Form_Open!"
17    End Sub
18
19    Private Sub Form_Resize()
20      Debug.Print "Form_Resize!"
21    End Sub
22
```

Figure 4.26

Opening a form we see the order in which this series of events prints to the immediate window.

Figure 4.27

Closing a Form

Closing a form has fewer events than opening a form but is equally structured. Just to remind us: When the form is closing: **Unload →Deactivate →Close**

The Unload Event (and the load event) is Cancellable. Setting Cancel = True will prevent the form from being closed. This is very useful when users haven't saved their data and you wish for them to confirm that the changes are desired.

The Deactivate Event is the window equivalent of LostFocus. One cannot do anything about it but one could save data to the database which hasn't been committed.

The Close Event is a form and report object function. At this stage the object will be deleted once the event has finished.

```
1     Option Compare Database
2
3     Private Sub Form_Close()
4       Debug.Print "Form_Close!"
5     End Sub
6
7     Private Sub Form_Deactivate()
8       Debug.Print "Form_Deactivate!"
9     End Sub
10
11    Private Sub Form_Unload(Cancel As Integer)
12      Debug.Print "Form_Unload!"
13    End Sub
14
```

Figure 4.28

After closing the form the immediate window will look like this (I have removed the printouts from the opening of the form):

Figure 4.29

Cancel Form_Close Event

```
12      Cancel = True
13      Debug.Print "Form_Unload!"
```

Insert the above code into your form to test out the Cancel Unload operation	Form_Unload!

Figure 4.30

Recordset Control Events - OnCurrent, BeforeUpdate, AfterUpdate, OnChange

Data in a form is stored in the form's recordset property. All these events are associated with the interaction between the form and this underlying Recordset object.

The Current Event occurs when data in a form or report is refreshed. It typically fires when the active record on a bound form is changed.

The Before Update Event executes *just before the form changes are saved to the database*. This can be seen as an application implementation of update and insert triggers. Here you would carry out any final data validations, check business rules, populate hidden fields, and cancel the action altogether. As Access doesn't implement triggers (as that is a job for the Jet engine or other data source) this is probably the place where final validation checks should be done.

The After Update Event executes once the data has been committed to the database. Useful for updating other tables like audit trails, updating graphics to indicate a save, disable fields from being changed, close and open up a View type form.

The Change Event executes when data within a text object's content is changed and before the Before Update and After Update Events. This means you can validate the content of the control before it loses focus and before its data is committed to the database. If the Form is bound to a recordset then changing focus from a *changed* text control to another control will automatically attempt to commit the *change* to the field / record in the database.

Using frmStudentsDataEntry cycle through the records and every time you change a record you will see Form_Current! being printed in the immediate window.

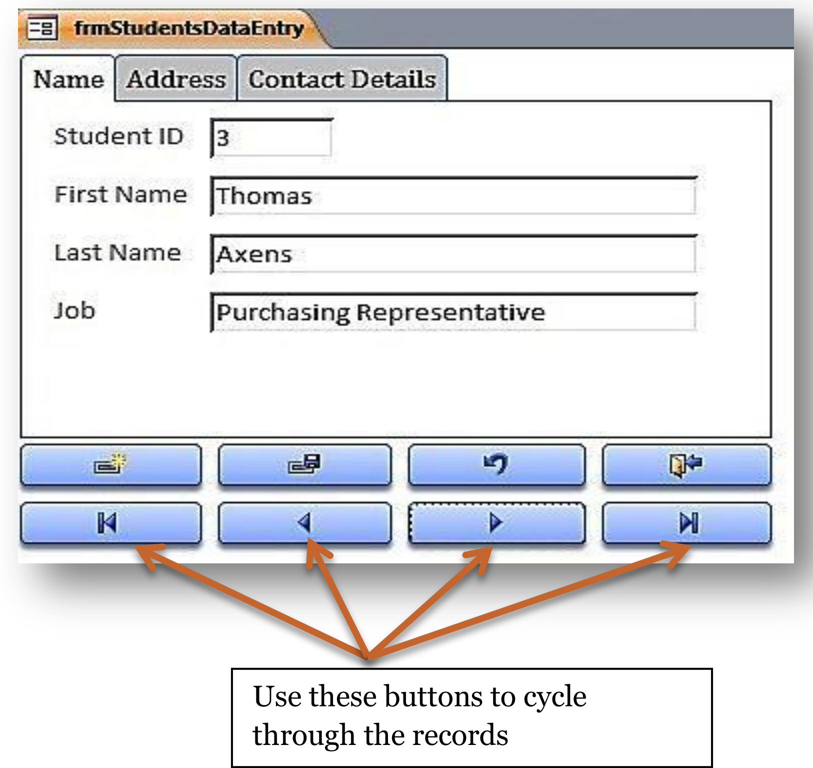

Figure 4.30

```
1    Option Compare Database
2    Private Sub Form_Current()
3      Debug.Print "Form_Current!"
4    End Sub
5
6    Private Sub Form_AfterUpdate()
7      Debug.Print "Form_AfterUpdate!"
8      Msgbox “Data change saved!”
9    End Sub
10
11   Private Sub Form_BeforeUpdate(Cancel As Integer)
12     Debug.Print "Form_BeforeUpdate"
13     If (MsgBox("are you sure", vbYesNo) = vbNo) Then
14       Cancel = True
15       Me.Undo
16     End If
17   End Sub
```

Put the form into form view and cycle back and forth. For each record movement the immediate window will have a Form_Current! Line member	Form_Current! Form_Current!
Change the value in the textbox and try to move to the next or previous record. This dialog should appear. The BeforeUpdate routine presents you with this dialog. If you press No the Cancel argument is set to True which forces the form not to update the database and not to progress to the next record. BTW, to cancel any changes press ESC and you'll be able to navigate again.	Microsoft Office Access are you sure Yes No Form_BeforeUpdate!
This time allow the changes to be saved. This will fire the After update event and display this message.	Microsoft Office Acc... Data change saved! OK Form_AfterUpdate!

Figure 4.31

OnTimer Events

The Timer Event is a special form event that is activated after a set period of time. The exact time of the event is at least the value of the Timer Interval property.

Open frmTimer to see the event at work. You should see a speedboat speeding across an ocean.

Figure 4.32

The code that goes behind the form is this:

```
1     Option Compare Database
2     Dim intCounter As Integer
3     Private Sub Form_Load()
4         Me.imgSpeedboat.Top = 2750
5         intCounter = 12500
6       Me.TimerInterval = 100
7     End Sub
8
9     Private Sub Form_Timer()
10    intCounter = intCounter - 500
11    If intCounter < 200 Then
12        intCounter = 12000
13    End If
14      Me.imgSpeedboat.Left = intCounter
15    End Sub
```

Figure 4.33

Although the code in figure 4.32 may look complicated it is actually fairly simple. Essentially every 1/10 of a second (*Me.TimerInterval = 100*) the Form-Timer() sub procedure is fired. And every time the the Form-Timer() subprocedure is fired the image of the speedboat is moved 500 twips to the left (a twip is a unit of measurement in Access. 1440 twips = 1 inch). And when there is no more left left (so to speak) the image is moved to 12500 twips from the left. And the whole thing repeats ad infinitum.

Exercises

1. What should be written in the On Current field of the property sheet to indicate that an event procedure exists for the On Current event?
2. When is the OnMouseUp event triggered?
3. Will the OnMouseDown event fire if you right-click a mouse?
4. If you tab from txtFocus1 to txtFocus2, which event fires first? The OnLostFocus event or the OnGotFocus event.
5. Look at this code:

```
1   Private Sub txtOnMouseUpDown_MouseDown(Button As Integer, Shift As
2   Integer, X As Single, Y As Single)
3   Me.txtOnMouseUpDown.BackColor = vbRed
4   End Sub
```

Figure 4.34

True or False: The argument Button (highlighted in red) refers to the button or text-box clicked on a form.

6. In the above code snippet what do the buttons X and Y represent?
7. What causes the OnMouseMove event to fire?
8. Look at this code:

```
1   Private Sub txtOnKeyPress_KeyPress(KeyAscii As Integer)
2   MsgBox "You have pressed a key!"
3   End Sub
```

Figure 4.35

If txtOnKeyPress had the focus, what would happen if we pressed the *ctrl* key?

9. These are the 5 events associated with opening a form:

Activate
Load
Current
Resize
Open
In what order are these events executed when a form opens?

10. In what order should these will these events associated with closing a form be fired?

Close
Unload
Deactivate

11. Although similar in nature, what is the difference between the Activate event and the OnGotFocus event?
12. When does the BeforeUpdate event fire?

Answers - Data types, variables, constants and operators

1. If written as a statement in a function (1 mark)
   If all given different names (1 mark)
   Otherwise 1 mark for each of the following
   a. Dim a As Boolean /cr/lf/ a = true or false or -1 or 0
   b. Dim a As Integer
   c. Dim a As Long
   d. Dim a As Date
   e. Dim a As Single
   f. Dim a As Double
   g. Dim a As Currency
   h. Dim a As String
   i. Dim a As Date
   j. Dim a As Variant
2. 1 mark for each line
   a. Dim CONSTANT_NAME = “Hello “ ‘ there’s a space at the end
   b. Dim YEAR = 2012
   c. Dim myName as String
      i. myName = “pupil’s name”
   d. Dim myMessage as String
      i. myMessage = CONSTANT_NAME + myName
   e. myMessage = myMessage + “. The year is “
   f. myMessage = myMessage + CStr(YEAR)
   g. debug.print myMessage
3. No output as this subroutine does not compile
4. 1 mark for stating 42, +1 mark VBA automatically converts “42” into Integer type
5. 1 mark for each
   a. The String “42.2” is added to A1
   b. The Double 42.2 is added to A2
   c. Type mismatch error
   d. Cannot multiply string by integer
6. Function is a keyword
7. 1 mark for each
   a. calculateTheAgeOfATree
   b. TapeReaderFilePosition
   c. UserInput
   d. FilePointer
   e. Input
   f. sMyMessage
8. 1 mark for each
   1. SEMAPHORE_STOP
   2. SEMAPHORE_START
   3. SEMAPHORE_PAUSED
   4. FILE_OPEN
   5. END_OF_FILE
   6. CARRIAGE_RETURN_AND_LINE_FEED
   7. NEW_LINE
9. 1 mark for each

a. Const DATABASE_NAME = "mysqldb_website1"
b. Dim IP – IP may be anything as long as its meaningful and camelCase
    i. In function – IP="127.0.0.1"
c. In function
    i. sDBDetails = DATABASE_NAME + " " + (b variable)
d. Msgbox sDBDetails
e. Everything in a function

10. 1 mark for each. They should all follow the same basic format given in (a)
    8. Value is 3, rounding
    9. "Winston, you are drunk! " + "Yes madam, and you are ugly!" + "But in the morning, I shall be sober"
    10. Trick question, "nearest whole number"
        10.1.1. Dim a, z as Integer
        10.1.2. a = 27
        10.1.3. z = a*a
11. 1 mark
    a. "I like Chinese food! The wai-ters never are rude."
12. 1 mark,
    a. Line 6 needs #'s: `d = #12 Dec 2012#`
13. 1 mark for each
    a. All should have #'s around them EXCEPT f, which needs to be rewritten without the "th"
14. 1 mark for each
    a. Dim [variable name] as Variant / or Dim name
    b. Followed by respective values
15. 1 mark for putting all into a single function
    1 mark for each expression
    a. 21
    b. -2
    c. 1.66666666666667
    d. 18
    e. 15/12/2113 – yes, that's the answer; VBA doesn't make sense here
16. 1 mark for each
    a. An integer holds whole numbers
    b. An floating-point number holds decimals / numbers and fractions
    c. An integer holds less data than a floating-point number / or vice-versa
    d. Integers are calculated in the CPU
    e. Floating-point numbers are calculated in the FPU / ALU
17. 1 mark – the first is a string, second a date
18. 1 mark – String
19. 1 mark for each
    a. False
    b. False
    c. true
20. 1 mark for each
    a. aVariable - valid
    b. aFunction - valid
    c. end – invalid keyword

d. while – invalid, keyword
e. STATE_HOLD - valid
f. STATE OVER – invalid, no spaces allowed
g. File1 - valid
h. outputFile_10 - valid
i. input-file2 – invalid, means input subtract file2, input is also a keyword
j. $helloWorld – invalid, variables must start with a letter
k. 9LivesACatHas - valid, variables must start with a letter
l. todayisyesterday – valid, but should be camelCase
m. Tomorrow NeverComes – invalid, nospaced in strings

Answers - Events

1. [Event Procedure]
2. When a depressed mouse button is released.
3. Yes.
4. The OnLostFocus event.
5. False: it refers to which mouse button was pressed.
6. The Coordinates of the mouse curser.
7. Hovering the curser over an object that has an event procedure for OnMouseMove.
8. Nothing. The onKeyPress event is only triggered by keys that return characters.
9. **Open →Load →Resize →Activate →Current**
10. **Unload →Deactivate →Close**
11. The activate event fires when a window (such as a form, report or dialog box) receives the focus whilst the OnGotFocus event fires when a control receives the focus.
12. Just before committing changes to the server.