

VBA Made Easy

Access VBA Fundamentals

Level 3

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

o5 - Functions, Sub Procedures And Arguments	5
VBA Language in Context	5
What is a Sub Procedure?	5
What is a Function?	6
Calling Sub Procedures And Functions From The Immediate Window.....	6
Calling Sub Procedures from other Sub Procedures.....	8
Calling Functions	9
Built-in Functions	10
Using Expression Builder	10
Commonly Used and Useful Built-In Functions.....	13
String Functions	13
Conversion.....	13
Date and Time Functions	14
Is Functions	15
Custom Functions And Sub Procedures	18
Anatomy of a Sub Procedure	19
Anatomy of a Function	21
Declaring Functions and Procedures	22
Scope.....	22
Declarations in a Module and Global Scope (and a little private-cy).....	22
Declarations in a Form or Report Modules.....	25
Exercises.....	26
Answers	47
o6 – Debugging.....	30
Learning Outcomes	Error! Bookmark not defined.
Break on Unhandled Errors	30
Breakpoints	32
Debug Control Bar	33
Immediate Window.....	34
? and Debug.Print.....	34
: to concatenate commands	34
; to concatenate strings.....	34
Note	34
Call a Procedure.....	35
Immediate Window is in Scope.....	35

Code Window Pop-ups.....	36
Watches Window.....	37
VBE Editor Options.....	38
Compilation Explained	40
Advanced Compilation and ACCDE.....	40
ACCDE files	Error! Bookmark not defined.
Why use an ACCDE file?.....	42
Multi-Users Environments	42
Questions.....	44

05 - Functions, Sub Procedures And Arguments

In this unit you will learn what Functions, Sub Procedures and Arguments are.

VBA Language in Context

The core of the English language is its sentences and paragraphs. The sentence describes some action (verb) that is performed on or by an object (noun), and a paragraph is a set of sentences communicating some overall desired goal or aim. VBA is not unlike English in this sense.

VBA's paragraphs are called Procedures and Functions. Sentences then are the variables, operations, object methods and assignment statements in the Code Block. All recent programming languages share this same structure. To continue the analogy, functions and procedures (the paragraphs) are contained within books called VBA Modules. There are three types of book, or module:

- Forms and Reports Module (Microsoft Office Access Class Object Modules); or,
- Standard Modules.
- Class Modules;

When you write your code it will always be written within a Function...End Function or a Sub...End Sub statement. VBA is what is known as a functional programming language. That is, we cannot just write code within a standard module and expect it to run; Access won't recognise this and will complain terribly, we must put Sub or Function around it.

What is a Sub Procedure?

You may not realise it but you have already used sub procedures in this course. Do you remember this sub procedure from the unit on Variables?

```
1 Sub getPriceIncVATSub()  
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 5.1

In this sub procedure you may notice that all the code is held within the *Sub getPriceIncVAT()* and the *End Sub* statements. These are the outer limits of the sub procedures and any code that comes before *Sub getPriceIncVAT()* and after *End Sub* do not form part of the sub procedure

What is a Function?

Functions are not dissimilar to sub procedures in that they do something but where they differ is that they also return a value.

```
1  Function getPriceIncVATFunction(ItemPrice As Double)
2
3  Dim SalesTax As Double
4  Dim PriceIncVAT As Double
5  SalesTax = 0.2
6  getPriceIncVATFunction = ItemPrice + (ItemPrice * SalesTax)
7
8  End Function
```

Figure 5.2

In Figure 5.2 we have changed the sub procedure into a function and it is now returning a value.

Note

Please only take into consideration the structure of the function as we will be covering the syntax in greater detail later on in this unit.

Calling Sub Procedures And Functions From The Immediate Window

One of the benefits of the immediate window is that we can use it to test sub procedures and functions.

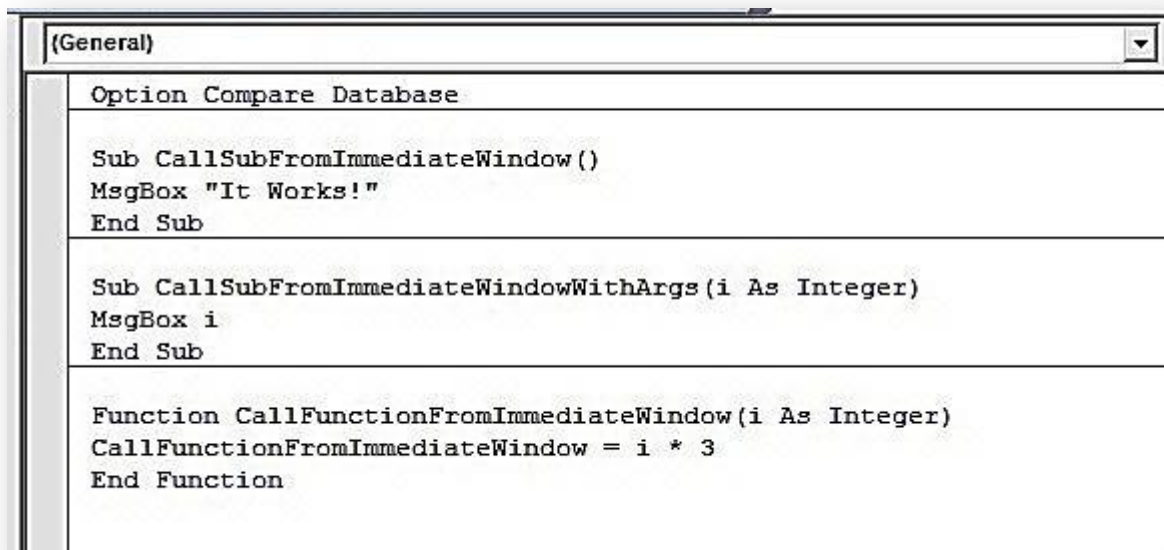


Figure 5.3

Take a look at Figure 5.3 where we have 2 very simple sub procedures and 1 very simple function.

In order to call the procedure *CallSubFromImmediateWindow* using the immediate window we merely need to write its name (without the parentheses).

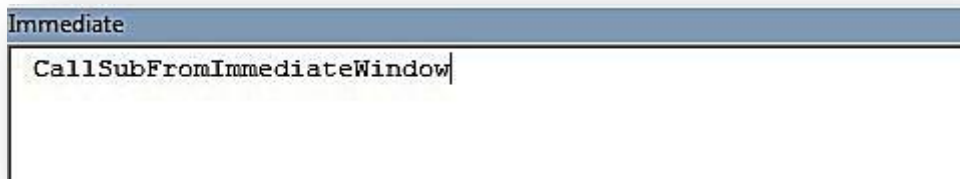


Figure 5.4

This will cause a message box to pop it that states “It works!”

We can also add arguments in the immediate window. In the second sub called *CallSubFromImmediateWindowWithArgs* we need to pass a value *i*. We do this by writing the name of the procedure and then adding the necessary argument to the right.

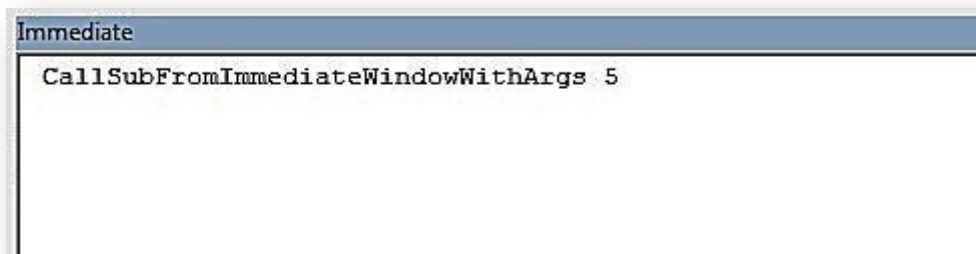


Figure 5.5

In Figure 5.5 we call *CallSubFromImmediateWindowWithArgs* and provide the argument *i*. In this case we pass the value 5 and a message box will pop up with the value 5 in it. Whatever we change the value of the argument to, will be reflected in the value that the message box displays.

We can also test functions. Remember that functions are essentially the same as sub procedures with the difference that they return a value.

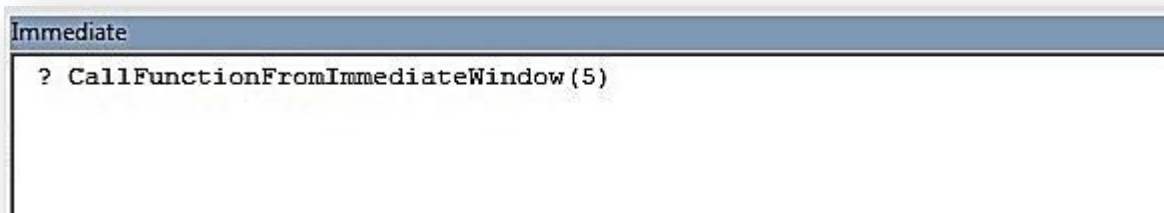


Figure 5.6

To test a function from the immediate window we use a question mark and then we write the name of the function. We follow the function with parentheses and any relevant arguments are placed inside the parentheses. We have done this in Figure 5.6.



Figure 5.7

Figure 5.7 shows that if we provide 5 as an argument for this particular function we get a value returned of 15. Try adding different values as the argument to see what return value you get.

Calling Sub Procedures from other Sub Procedures

One of the most important features of VBA is the ability to call sub procedures from other sub procedures. What do we mean? Take a look at this code to find out:

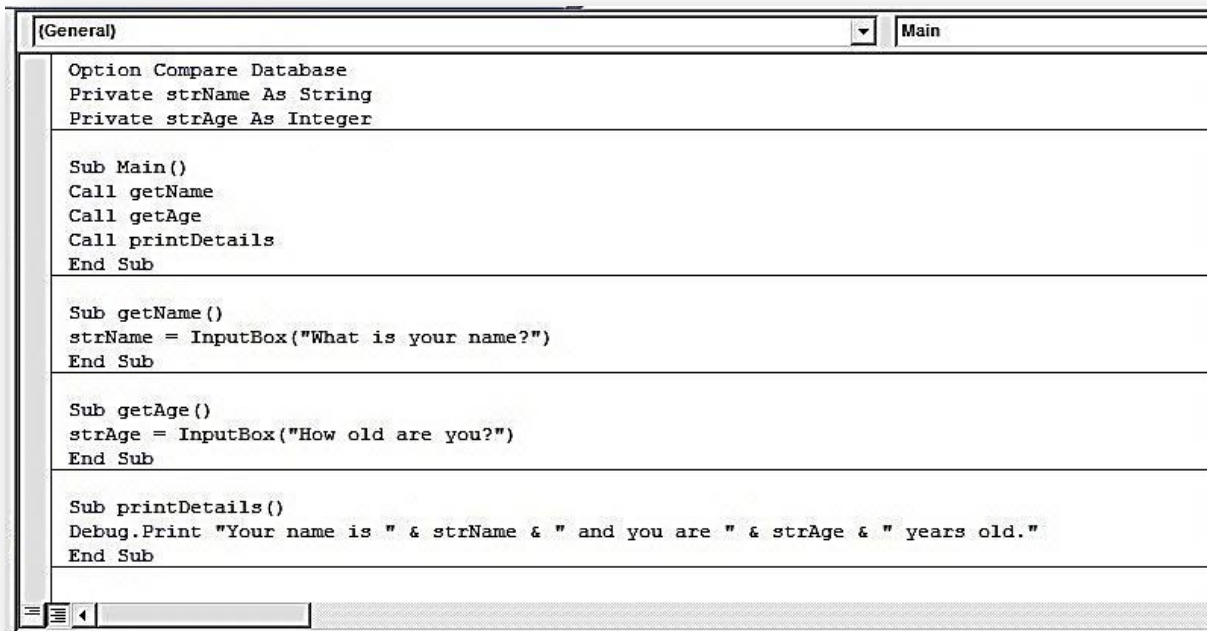


Figure 5.8

In Figure 5.8 we have 4 sub procedures *Main*, *getName*, *getAge* and *printDetails*. The main sub procedure we have cleverly called *Main* and this sub procedure calls all the other sub procedures within the module. It first calls *getName* which has the objective of asking the user's name. This value is then assigned to *strName* which is a module level variable. Next, *getAge* is called which involves another input box asking you for your age and again the value is stored in a module level variable called *strAge*. Finally *printDetails* is called which takes the 2 module level variables and concatenates them in a string which is printed in the immediate window.

In Figure 5.9 below we call the sub Main from the immediate window by writing Main and pressing the return key and then provide Steve and 25 as the values for the variables.

Note

Breaking code down into manageable chunks and having a main procedure that calls other procedures (and functions) is an excellent way to code.

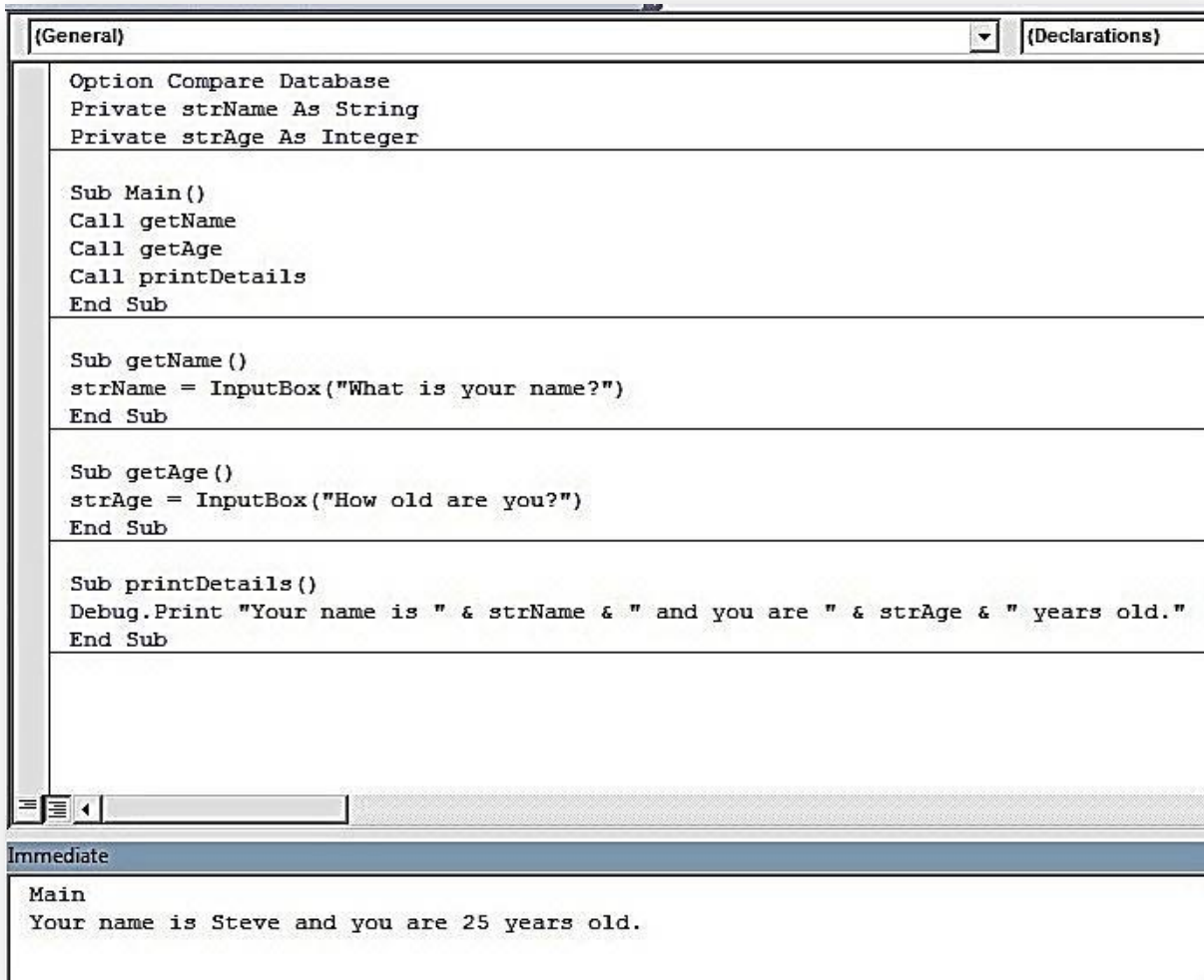


Figure 5.9

Calling Functions

Figure 5.10 has the same concept (you are asked for your name and age which are printed in the immediate window) but this time we are using 1 sub procedure (Main) which is calling functions. As functions return values it is those that are used as the basis for the concatenated string at the end.

```
(General) Main1
Sub Main1()
Dim strName1 As String
Dim strAge1 As Integer
Dim strPrintDetails As String

strName1 = getName1
strAge1 = getAge1
strPrintDetails = print1Details(strName1, strAge1)
Debug.Print strPrintDetails
End Sub

Function getName1()
getName = InputBox("What is your name?")
End Function

Function getAge1()
getAge = InputBox("How old are you?")
End Function

Function printDetails1(strName2 As String, strAge2 As Integer)
printDetails = "Your name is " & strName2 & " and you are " & strAge2 & " years old."
End Function
```

Immediate

Figure 5.10

Using functions is another great way to break down your code into manageable bits. In the previous example we wrote custom functions but VBA has plenty of built-in functions all of its own.

Built-in Functions

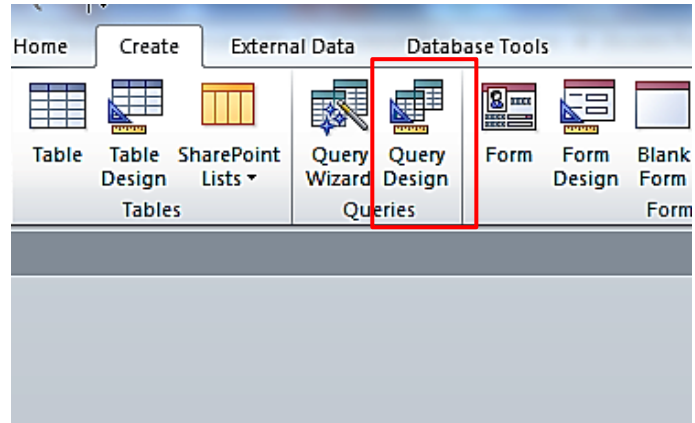
VBA has a wide library of built-in functions. Do look through them and do experiment with them. Most puzzles you try to overcome and actions to be fulfilled can be performed by using these functions, so try not to reinvent the wheel.

Using the Query Expression Builder to locate functions

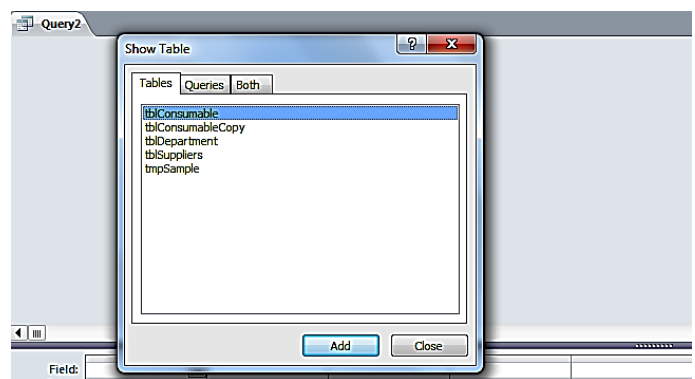
As there are scores of built-in functions in Access/VBA, wouldn't it be great if we had an easily accessible list that listed not only the functions but also their uses. Well, rest assured, we do (kind of). We can use the expression builder in a query to perform this particular function (do you like what we did there?)

Opening the Expression Builder

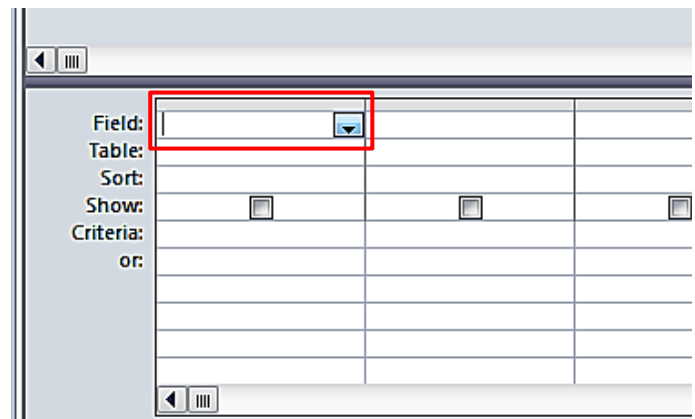
In the main Access window click on the Query Design button which can be found in the Queries group of the Create tab of the Ribbon.



Dismiss the Show Table Dialog Box.



Click in the Field row in any column in the field designer window.



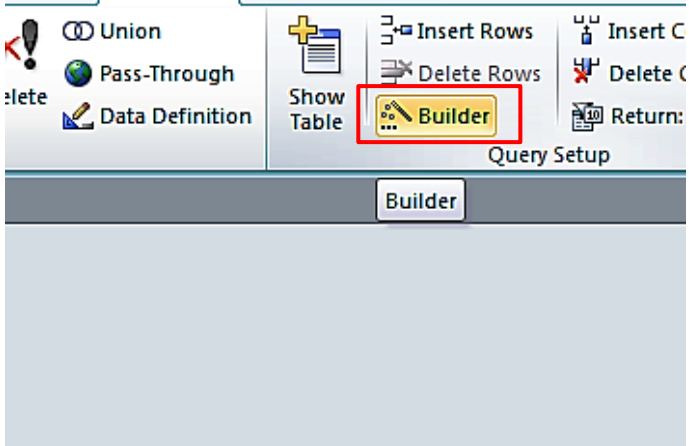
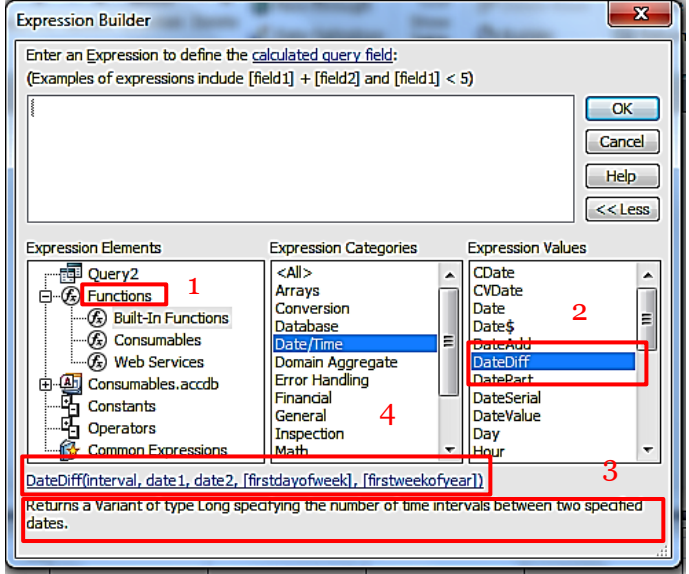
<p>Click on the Builder button which is located in the Query Setup group of the Design tab of the Ribbon.</p>	
<p>The Expression Builder dialog box will pop up.</p> <p>Open the Functions Node (1) in the Expression Elements window and a list of all functions will be displayed.</p> <p>If you click on one of the functions in the Expression Values window (2) you will get a brief explanation of what it does (3).</p> <p>Clicking on the hyperlink text of the function syntax (4) will open up a more detailed explanation of the function in a browser window.</p>	

Figure 5.11

Commonly Used Built-In Functions

This section will provide you with examples of commonly used built-in functions

String Functions

- **Len(s)** – returns the length of String s.
- **Left(s, n)** – returns a substring of s that is n chars long from the left of the string s.
- **Right(s, n)** – returns a substring of s that is n chars long from the right of the string s.
- **Mid(s,nb,ne)** – returns a substring of s from characters nb to ne, inclusive.

```
1 Sub testStrings()  
2   Debug.Print Len("Hello World")  
3   Debug.Print Left("Hello World", 10)  
4   Debug.Print Right("Hello World", 7)  
5   Debug.Print Mid("Hello World", 7, 10)  
6 End Sub
```

Output in immediate window:

```
11  
Hello Worl  
o World  
World
```

Figure 5.12

Conversion

- **CInt(*anything*)** – converts anything into an Integer type (if possible).
- **Cdbl(*anything*)** – converts anything into an Double type (if possible).
- **CInt(*anything*)** – converts anything into an Long type (if possible).
- **CStr(*anything*)** – converts anything into a String.
- **CDate(*string*)** – converts a string to a Date type (if possible).

If any of the conversion functions are passed a variable that cannot be parsed – e.g. **CInt("oioi!")** – a Type Mismatch error occurs.

```
1  
2 Sub testConversions()  
3   Dim i As Integer, d As Double, l As Long, s As String  
4   i = 19  
5   d = 12.6  
6   l = 32768  
7   s = "42.001"  
8  
9   ' to display the answers provided by the conversion functions we have to  
10  ' so just to prove that CStr works we'll do it first  
11  Debug.Print "First test CStr on all types"  
12  Debug.Print "CStr(i) = '" + CStr(i) + "'" ' '42'  
13  Debug.Print "CStr(d) = '" + CStr(d) + "'" ' '42.001'  
14  Debug.Print "CStr(l) = '" + CStr(l) + "'" ' '42'  
15  Debug.Print "CStr(s) = '" + CStr(s) + "'" ' '42.001'  
16  Debug.Print ""  
17  Debug.Print "Second, CInt"  
18  Debug.Print "CInt(i) = " + CStr(CInt(i)) ' 19  
19  Debug.Print "CInt(d) = " + CStr(CInt(d)) ' 13  
20  Debug.Print "CInt(l) = Overflow Error. Integers are valued <32768"  
21  Debug.Print "CInt(s) = " + CStr(CInt(s)) ' 42  
22  Debug.Print ""  
23  Debug.Print "Third, Cdbl"  
24  Debug.Print "Cdbl(i) = " + CStr(Cdbl(i))
```

```

24 Debug.Print "Cdbl(d) = " + CStr(Cdbl(d))
25 Debug.Print "Cdbl(l) = " + CStr(Cdbl(l))
26 Debug.Print "Cdbl(s) = " + CStr(Cdbl(s))
27 Debug.Print ""
28 Debug.Print "Fourth, CLng"
29 Debug.Print "CLng(i) = " + CStr(CLng(i)) ' 19
30 Debug.Print "CLng(d) = " + CStr(CLng(d)) ' 13
31 Debug.Print "CLng(l) = " + CStr(CLng(l)) ' 32768
32 Debug.Print "CLng(s) = " + CStr(CLng(s)) ' 42
End Sub

```

Output in immediate window:

```

testConversions
First test CStr on all types
CStr(i) = '19'
CStr(d) = '12.6'
CStr(l) = '32768'
CStr(s) = '42.001'

Second, CInt
CInt(i) = 19
CInt(d) = 13
CInt(l) = Overflow Error. Integers are valued <32768
CInt(s) = 42

Third, Cdbl
Cdbl(i) = 19
Cdbl(d) = 12.6
Cdbl(l) = 32768
Cdbl(s) = 42.001

Fourth, CLng
CLng(i) = 19
CLng(d) = 13
CLng(l) = 32768
CLng(s) = 42

```

Figure 5.13

Date and Time Functions

Date and time functions are quite complex due to the nature of dates. VBA has a special way of handling dates by putting # around them, for example `dMyDate = #18-Dec-2012#`. Here are some of the functions to help with dates.

- `Date ()` – returns the current date.
- `Now()` – returns the current date and time.
- `DateSerial(year, month, day)` – returns a Date object if parameters are valid.
- `Year(date)` – returns the *year* of *date* as an integer.
- `Month(month)` – returns the *month* of *date* as an integer, 1-12.
- `Day(Day)` – returns the *day* of *date* as an integer, 1-31.
- `DateDiff(interval, date, date)` – *date* are dates, *interval* is day, month, year, etc.
- `DateAdd(interval, number, date)` – add to *date* intervals multiplied by *number*

Date Intervals

In the above *interval* refers to one of the following:

Interval	Description
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

Figure 5.14

Note

The Date function returns the current date (as defined by your operating system) so the results you get from the following example will be different from the results we obtained.

```
1 Sub testDateTime ()
2   Debug.Print Date
3   Debug.Print Now()
4   Debug.Print DateSerial(2012, 12, 18)
5   Debug.Print Year(Date)
6   Debug.Print Month(Date)
7   Debug.Print Day(Date)
8   Debug.Print DateAdd("d", 421, Date)
9   Debug.Print DateDiff("d", Date, #1/1/2020#)
10 End Sub
```

```
Output in immediate window:
27/12/2012
27/12/2012 22:50:08
18/12/2012
2012
12
27
21/02/2014
2561
```

Figure 5.15

Is Functions

When inspecting whether a variable has a value we usually use the equals = operator, but equals does not work if a variable is null, empty or is nothing. Nor can equals be used to interrogate the variable for its type. There are special 'Is' operators which provide for that functionality.

- `IsDate(anything)` – returns true if variable is a date.
- `isArray(anything)` – return true if variable is an array.
- `IsNull(anything)` – returns true if variable is Null.
- `IsEmpty(anything)` – returns true when type variable is uninitialized.
- `isObject(anything)` – returns true when variable is an Object.

- `TypeName(anything)` – returns a string.

IsDate and IsEmpty

```
1
2 Option Explicit
3
4 Sub dateAndEmptyFunctions ()
5     Dim myDate
6
7     Debug.Print IsDate(myDate)
8     Debug.Print IsEmpty(myDate)
9
10    myDate = #12/20/2012#
11    Debug.Print IsDate(myDate)
12    Debug.Print IsEmpty(myDate)
13 End Sub
```

Output in immediate window:

```
False
True
True
False
```

Figure 5.16

Note

We will be covering arrays in a future unit.

IsArray and IsNull

```
1
2 Sub arrayAndNullFunctions ()
3     Dim myArray As Variant
4     myArray = Array("first_name", "surname", "dob", "town", Null)
5
6     Debug.Print IsArray(myArray)
7     Debug.Print IsNull(myArray(0))
8     Debug.Print IsNull(myArray(1))
9     Debug.Print IsNull(myArray(2))
10    Debug.Print IsNull(myArray(3))
11    Debug.Print IsNull(myArray(4))
12 End Sub
```

Output in immediate window:

```
True
False
False
False
False
True
```

Figure 5.17

IsObject and TypeName

```
1
2 Sub objectAndTypeNameFunctions()
3   Dim varA, varB As Object, varC As Date, varD As DAO.Recordset
4
5   Debug.Print
6   Debug.Print "isObject(varA) = "; CStr(IsObject(varA)); Tab; "TypeName(varA) =
7   "; TypeName(varA)
8   Debug.Print "isObject(varB) = "; CStr(IsObject(varB)); Tab; "TypeName(varB) =
9   "; TypeName(varB)
10  Debug.Print "isObject(varC) = "; CStr(IsObject(varC)); Tab; "TypeName(varC) =
11  "; TypeName(varC)
12  Debug.Print "isObject(varD) = "; CStr(IsObject(varD)); Tab; "TypeName(varD) =
13  "; TypeName(varD)
14 End Sub
```

Output in immediate window:

```
isObject(varA) = False      TypeName(varA) = Empty
isObject(varB) = True       TypeName(varB) = Nothing
isObject(varC) = False     TypeName(varC) = Date
isObject(varD) = True       TypeName(varD) = Nothing
```

Figure 5.18

DFunctions - Database Functions

Sometimes it is necessary to retrieve certain data from the database - e.g. a manufacturer's name – or perform a quick count on records. Rather than having to create objects and write SQL statements VBA offers a couple of smart and concise routines to obtain what you need without all the object/SQL hassle.

All DFunctions have the same signature *expression, table[, criteria]* which is similar in structure to SQL itself.

- DLookup (*expression, table, [criteria]*) – Looks up a value in a table or query.
- DCount (*expression, table, [criteria]*) – Counts the records in a table or query.
- DSum(*expression, table, [criteria]*) – Returns the sum of a set of records in a range.
- DMax (*expression, table, [criteria]*) – Retrieves the largest value from a range.
- Dmin(*expression, table, [criteria]*) – Retrieves the smallest value from a range.
- DAvg(*expression, table, [criteria]*) – Returns the average set of numeric values from a range.
- DFirst (*expression, table, [criteria]*) – Returns the first value from a range.
- DLast (*expression, table, [criteria]*) - Returns the last value from a range.

```
1
2 Option Explicit
3 Sub DFunctions()
4
5   'These D-Functions will be using data from the teachers table
6
7   Debug.Print DLookup("[LastName]", "tblTeachers", "[FirstName]='Anna'")
8   'We are looking up a value in the [LastName] field of tblTeachers.
9   'We are looking up states that the [FirstName] field must be equal to Anna
10
11  Debug.Print DLookup("[Address]", "tblTeachers", "[FirstName]='Anna'")
12
13  Debug.Print DCount("*", "tblTeachers")
14  'The asterix (*) means that we are counting all the records in the table
15
16  Debug.Print DCount("*", "tblTeachers", "[zippostal]='98052'")
```

```

15  Debug.Print DMax("[City]", "tblTeachers")
16  Debug.Print DMin("[Zippostal]", "tblTeachers")
17  End Sub
18

```

Output in immediate window:

```

Gratacos Solsona
123 2nd Street
9
5
Salt Lake City
98004

```

Figure 5.19

Custom Functions And Sub Procedures

Having looked at built-in functions we are now going to create our own custom function.

Let's write a function that calculates the age of a student given the date of birth. The details we know are as follows:

- A returned value is needed, so we must use a function.
- The value returned will be somebody's age, so we should return an Integer.
- The function needs to know the student's DOB, so a Date parameter is needed.
- We also need a relevant function name; let's call it *calculateAge*.

The signature of the function then is:

```

Function calculateAge(DOB As Date) As Integer
End Function

```

We need a variable to store the age and to store today's date:

```

Dim iAge as Integer
Dim dToday as Date

```

Figure 5.20

Now we need to know the difference between DOB and today's date in years. VBA has a function for that, `DateDiff`. Let's set `dToday` to today's date and use `DateDiff` to give us the age in years.

```

dToday = Date()
iAge = DateDiff("yyyy", DOB, dToday) \ yyyy interval date

```

Figure 5.21

Finally, we also need to return `iAge` to the calling method by doing the following:

```

calculateAge = iAge

```

Figure 5.22

The whole function now looks like this:

```
1 Function calculateAge(DOB As Date) As Integer
2     Dim iAge As Integer
3     Dim dToday As Date
4
5     dToday = Date
6     iAge = DateDiff("yyyy", DOB, dToday) ' yyyy interval date
7     calculateAge = iAge
8 End Function
```

Figure 5.23

In the immediate window we call the function with a known anniversary date, e.g. today's date minus 1 year:

```
Output in immediate window:
Print calculateAge (#19/12/2011#)
1
```

Figure 5.24

Let's try with another known date, your own age:

```
Output in immediate window:
? calculateAge (#15/11/1978#)
34
```

Figure 5.25

Note

The Date function returns the current date (as defined by your operating system) so the results you get from the following example will be different from the results we obtained.

So, we know how to use sub procedures and functions. Let's take a closer look at the syntax of each one.

Anatomy of a Sub Procedure

In VBA the *Sub* keyword denotes a *procedure*. Procedures are designed to perform some action.

The syntax of a procedure is:

```
Sub nameOfSub (arguments | optional arguments As Datatype[=defaultValue] )  
    [Code Block]  
End Sub
```

nameOfSub – name of the sub procedure.

Arguments – are a list of values and types that are collected and used within the sub procedure.

Optional arguments As Datatype[=defaultValue] – an argument may be optional and if it is then you may provide a default value.

```
1  \ Declarations of Procedures-syntax highlighting to aid understanding  
2  \ put this section in the module window  
3  
4  Sub DoNothing()  \ basic procedure  
5      MsgBox "Do Nothing ☹"  
6  End Sub  
7  
8  Sub DoNothing2(name as String)  \ one argument provided  
9      MsgBox "the name is " + name  
10 End Sub  
11  
12 Sub DoNothin3(optional name as String)  \ one optional argument  
13     MsgBox "The name is " + name  
14 End Sub  
15  
16 \ one optional argument which defaults to Julia  
17 Sub DoNothing4(optional name as String = "Julia")  
18     MsgBox "The name is " + name  
19 End Sub  
20  
21  
22 Sub DoNothing5(name as String, age as Integer)  \ two arguments provided  
23     MsgBox "the name is " + name + " with age " + CStr(age)  
24 End Sub  
25  
26 \ put this section into the immediate window  
27 DoNothing  \ Simple call  
28 DoNothing2 "Julia"  \ Julia displayed  
29 DoNothing3  \ optional name left out, blank appears  
30 DoNothing4  \ optional name left out but will default to Julia  
31 DoNothing5 "Julia", 32  \ two arguments
```

Figure 5.26

Anatomy of a Function

In VBA a Function is a Procedure that returns a value. Functions accept data through arguments, they perform operations internally just like a procedure, but finish with a value which may be returned by the function.

```
Function nameOfFunction (arguments | optional arguments As  
Datatype[=defaultValue] ) _  
    As returnDataType  
    [Code Block]  
    [nameOfFunction = expression]  
End Function
```

nameOfFunction – is the name of the function.

Arguments – are a list of values and types that are collected and used within the function.

optional [arguments] [=defaultValue]] – an argument may be optional and if it is then you may provide a default value.

returnDataType – If stated, this is the value returned by the function, its data type.

```
1  ' Declarations of functions -syntax highlighting to aid understanding  
2  ' put this section in the module window  
3  
4  Function returnName1() ' basic procedure  
5      returnName1 = "returnName1 Called"  
6  End Function  
7  
8  Function returnName2(name as String) as String ' return name  
9      returnName2 = name  
10 End Function  
11  
12 Function returnName3(optional name as String) ' return name or Shaun  
13     If name="" Then returnName3="Julia" else returnName3=name  
14 End Function  
15  
16 ' one optional argument which defaults to Julia  
17 Function returnName4(optional name as String = "Julia")  
18     returnName4 = name  
19 End Function  
20  
21  
22 Function returnName5(name as String, age as Integer) ' two arguments  
23     returnName5 = "the name is " + name + " with age " + CStr(age)  
24 End Function  
25  
26 ' put this section into the immediate window  
27 Debug.Print returnName1()  
28 Debug.Print returnName2("Shaun")  
29 Debug.Print returnName3()  
30 Debug.Print returnName3("Shaun")  
31 Debug.Print returnName4()  
32 Debug.Print returnName5("Shaun", 34)
```

Figure 5.27

Declaring Functions and Procedures

Above we've read about what the differences are between functions and procedures

Scope

As we have seen, it is possible to call functions and sub procedures from other functions and sub procedures. But you can restrict which sub procedures and functions can be called. This is known as the scope of a function or sub procedure and is dependent on the location in which it is written and also the modifiers you put before the function or sub procedure name.

Possible modifiers are:

- Private - eg. `Private Sub txtName_Click()`
- Public - eg. `Public Function getCustomerName() As string`
- Nothing - eg. `Function isLeapYear() As Boolean`

For all modules Private stops anything seeing the private function or sub procedure except for other functions or sub procedures in the same module.

Putting Public before a method in a Standard Module, or putting nothing at all means that the method is available anywhere in the application, its GLOBAL! The reason for this is that Standard Modules are in global context.

Declarations in a Module and Global Scope (and a little private-cy)

In the example below we have a sub procedure and a function.

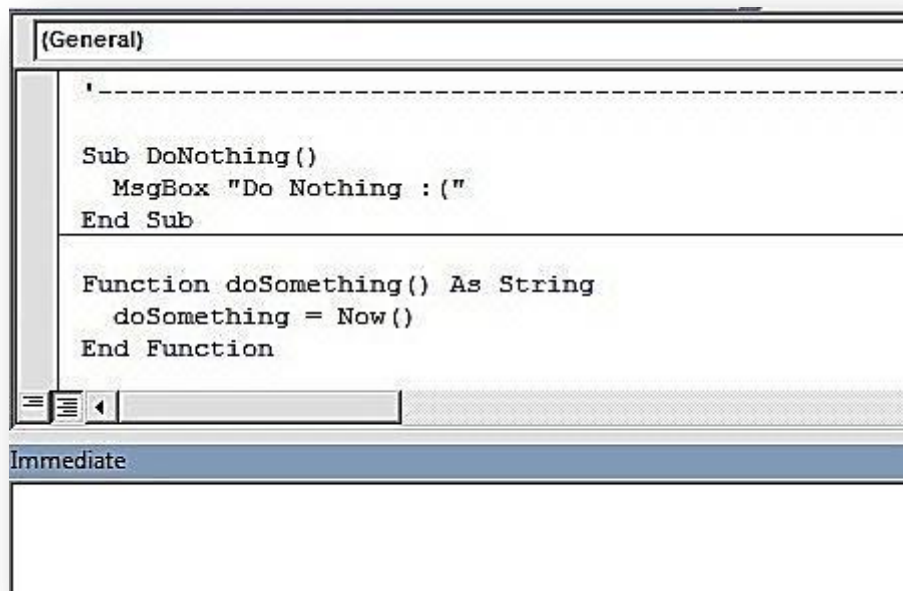


Figure 5.28

You can execute this function and sub procedure by entering their names directly into the immediate window one after the other:

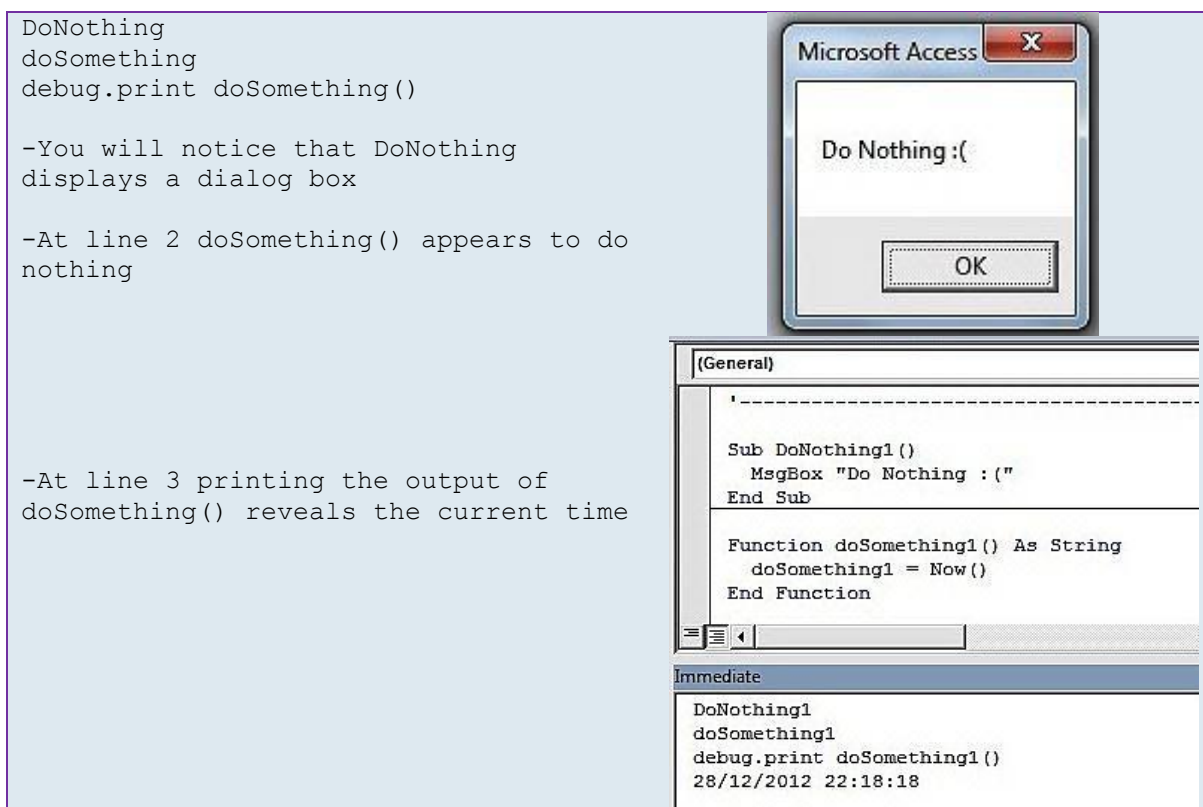


Figure 5.29

In fact you can execute this function and sub procedure from anywhere in your application. For example, navigate to the Module *FromAnywhere* and call *CallFromHere* from the immediate window.

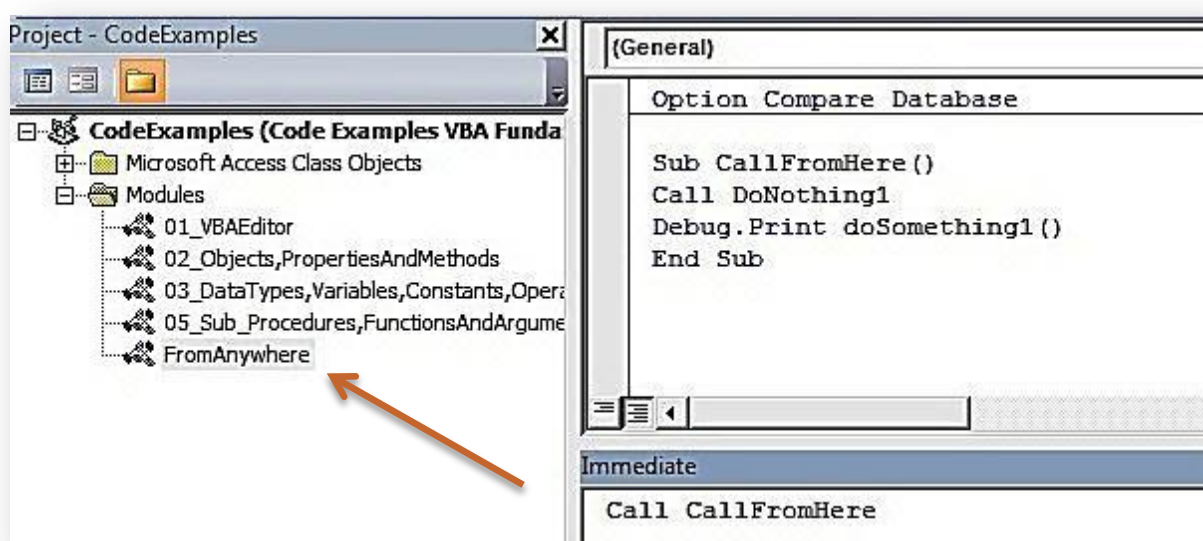


Figure 5.30

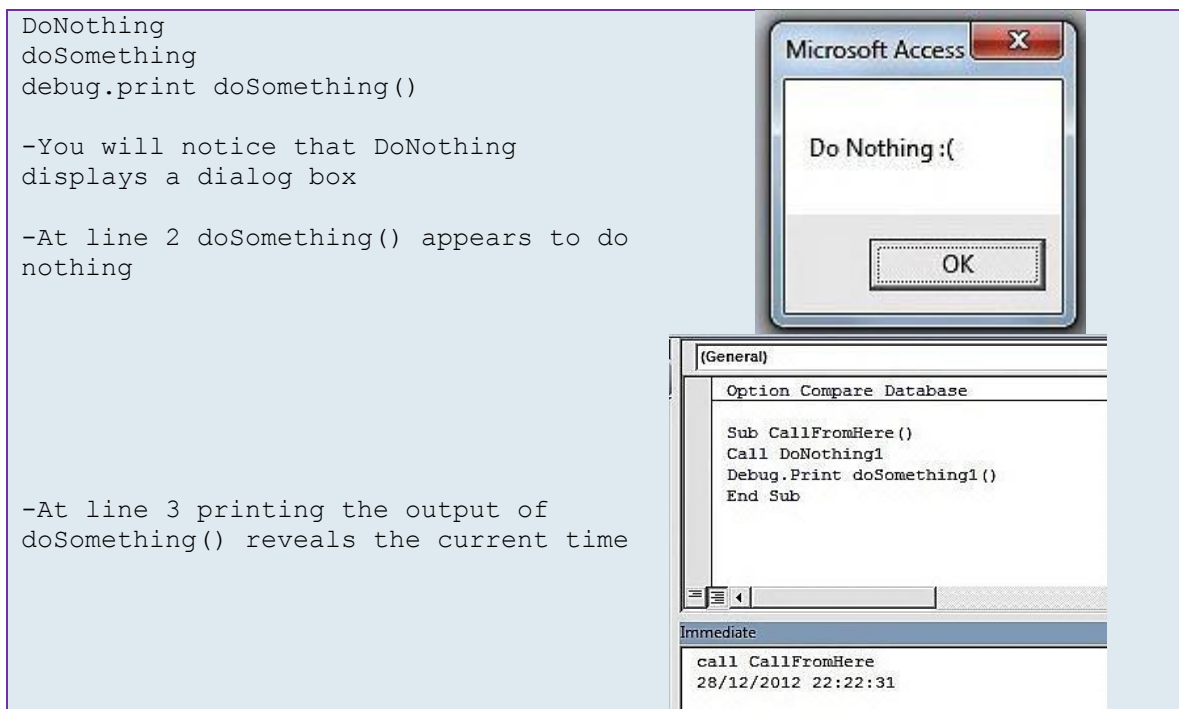


Figure 5.31

To demonstrate scoping with the Private modifier, add Private to the sub procedure *DoNothing1* and the function *doSomething1* and rerun the immediate window tests

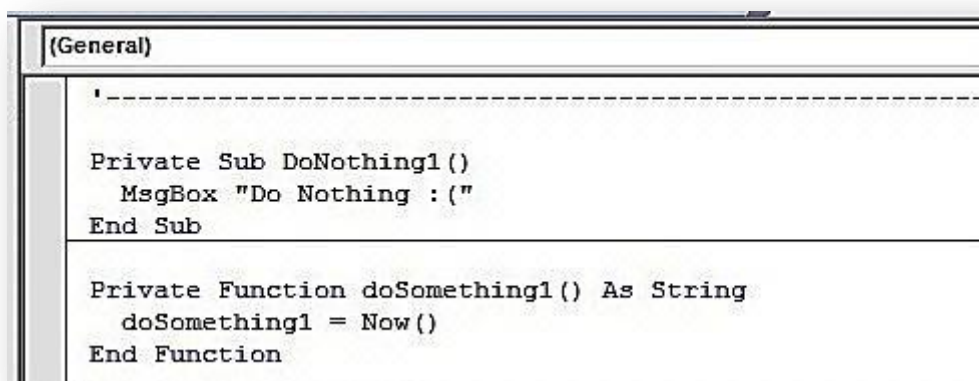


Figure 5.32

.Now *DoNothing* does nothing, except give you the error below! Private in a module means no VBA code outside the Module can see this sub procedure or function.

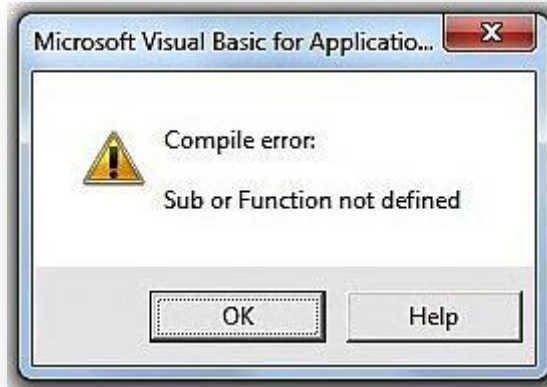


Figure 5.33

Declarations in a Form or Report Modules

In the Events unit you may have seen that all event subs created by the IDE are declared with the Private modifier. Private ensures that it is not possible for code outside the Form to call its own code. This is particularly important for Forms as executing any of the event procedures could cause a modification of data! That is why all Event Procedures are Private.

In Form and Report modules only put that code which is unique and specific to that form or report. You may include Public sub procedures if you need to give access to some functionality unavailable by conventional mechanisms.

Forms and reports do not need to be open for public sub procedures to be called and variables set or actions performed.

Exercises

1. Why would you want to use a function instead of a sub procedure?
2. Which one of the following signatures is valid for a function called **appointmentDate**?
 - a. `Function appointmentDate(customerID As Integer) As Date`
 - b. `Function Date appointmentDate(Integer customerID)`
 - c. `Sub appointmentDate(customerID As Integer) As Date`
 - d. `Date appointmentDate(Integer customerID)`
3. The signatures below have been extracted from a Standard Module. Which are available in Global scope?
 - a. `Private Function getNewID() As Integer`
 - b. `Public sub updateCustomerName(id as Integer, name as String)`
 - c. `Function IsClass(text As String) As Boolean`
 - d. `Sub updateModificationDate(recorded As Long)`
 - e. `Private Sub GetNextRecord()`
4. Match each DFunction on the left with its description on the right
 - a. DSum
 - b. DCount
 - c. DLookup
 - d. DMin
 - a. Returns the value of a field in a table for which ID=20.
 - b. Ordered by invoice number the function will return the smallest numerical value.
 - c. Returns a value equal to the number of records in a table.
 - d. For a table of invoices this function will return the total value of all invoices
5. Using the expression builder find the mathematical functions which do the following:
 - a. Calculates the square of a number.
 - b. Returns today's date.
 - c. Returns the time now.
 - d. Returns the difference between two dates.
 - e. Converts a Boolean value to a string.
 - f. Returns true when an object reference is empty.
 - g. Returns false when a recordset field doesn't have the value of null.
 - h. Gives back the aggregate sum value of a table's tax field.
 - i. Converts a string into a date.
6. Which function returns the string value of a variable type?

7. Function giveMeTime(name As String) As Date
 - a. What is the return data type?
 - b. Is this a procedure or a function?
 - c. With time As Date can time=giveMeTime("Mike")?
 - d. Which of the following will give a compiler error
 - i. A = giveMeTime "Mike"
 - ii. giveMeTime "Mike"

8. Match the following String functions on the left with their description on the right

<ol style="list-style-type: none"> a. Mid(s, a, b) b. Len(s) c. Left(s, a) d. Right(s, a) e. InStr(1, s, c) 	<ol style="list-style-type: none"> a. Gives the ending of a string from character position A to the end b. Returns a substring of a string c. From the beginning returns a smaller string from position no with length a d. Searches for one string inside another e. Give a count of the characters in a string
--	---

9. What does Now() provide you with that Date() does not?
10. What is the return value of Month(#29-February-2012#)
11. Write the following function called textAddNumber:
 - a. Parameters of myText and myNumber.
 - b. Returns a string equal to the text of myText with myNumber appended to the end.
 - c. Such that "Your score is" and 13 returns "Your score is 13".
12. Write the following procedure called calculate:
 - a. Parameters of a(integer), b(string), c(string)
 - b. Allocate a to houseNo, b to teleNum, c to Surname
 - c. Concatenate c+b+a to d
 - d. Write debug,print d
13. Using DLookup, write an expression that retrieves the [surname] of a [pupil] with [id] of 1192.

14. Using DCount write an expression that counts the number of [students] with a [telephone] number beginning with “555”.

15. Match the following date intervals with the description

Interval	Description
d	Weekday
h	Year
m	Month
n	Day
s	Second
w	Minute
yyyy	Hour

16. True or False (; a semi colon denotes a new line)?

- IsDate(#05/11/2012#)
- IsDate(#01:36:01#)
- Dim var As Application; IsObject(var)
- Dim foobar; IsEmpty(foobar)
- Dim foo as String; TypeName(foo) = "String"
- Dim bar as Object; TypeName(bar) = "Empty"

17. Write a function that, given an array (myArray) and an integer (i), returns the value of the myArray element i

18. In which module would you place the following code? Answer a) Standard Module, b) Form Module or c) Class Module.

- A globally available function?
- A procedure that can only be used by a form?
- A procedure that operates on a form but is available outside the form?
- A function that is specific to a class?
- A class function that can only be used by the same class?
- A procedure available to the whole project that minimises all windows and opens the form MainMenu?

19. On a new form you place three buttons named btnButton1, btnButton2, btnButton3.
When btnButton1 is clicked a message is displayed.
When btnButton2 is double-clicked the form closes.

When btnButton3 is clicked nothing happens.
What has buttons 1 and 2 that button 3 doesn't?

20. Read the following code

```
Sub DoNothing4(optional name as String = "Julia")  
  
    MsgBox "Morning Dave. My name is " + name  
  
End Sub
```

- a. What does optional mean?
- b. What is the default value of name?
- c. What is the name of the method?
- d. When the method is execute with the following values, what is the result?
DoNothing4 ("Hal 9000")

06 - Debugging

In VBA when we write code, it often doesn't work how we expect it to or we think it is working fine but need to be sure before handing it over to a client. For this reason we use debugging tools to enable us analyse our code whilst it is running.

Note

When writing code it is completely normal for it not to work as expected. Very few programs work 100% error free (if any at all) and our job as coders is to eliminate major errors and bullet-proof our code by making sure that any unforeseen errors are handled in some way by the IDE and not just left to confuse the end user.

Break on Unhandled Errors

Before going any further it is important to make sure the option to break on unhandled errors is on. We do this by selecting Options from the Tools tab:

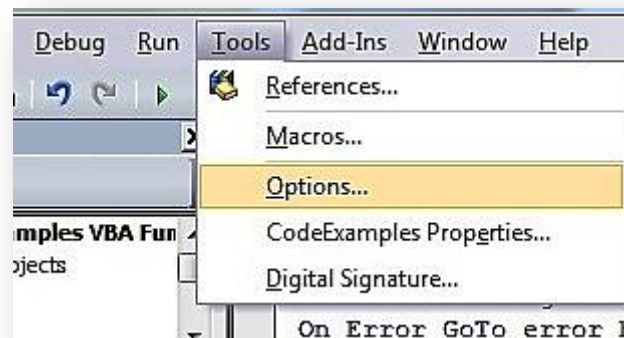


Figure 6.1

- Select the General tab of the dialog box.
- Tick “Break On Unhandled Errors” in the “Error Trapping” Option box.

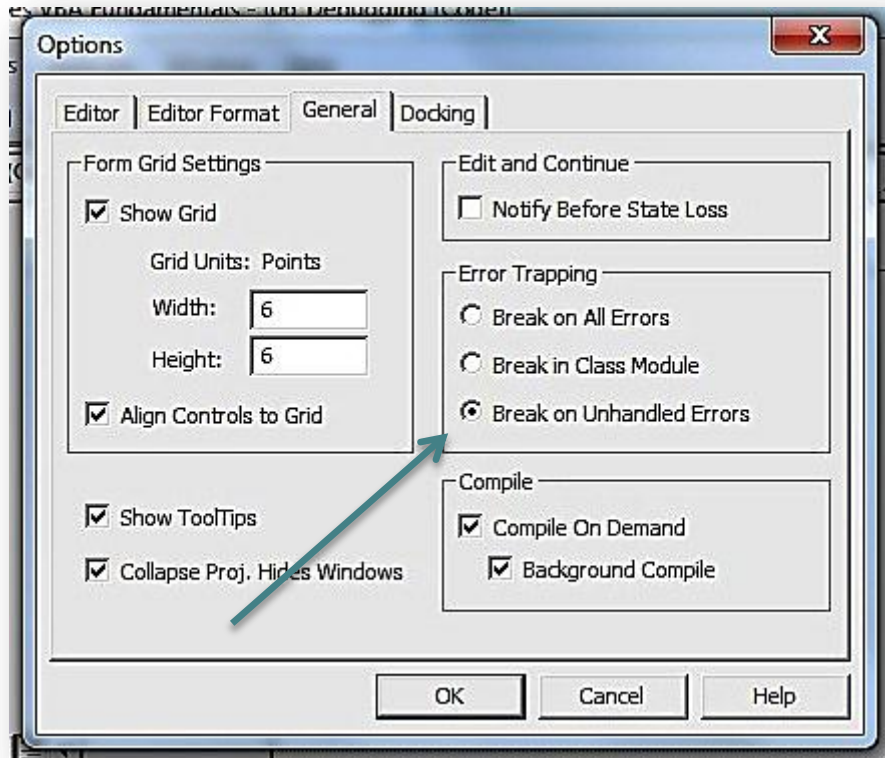


Figure 6.2

Ok, now we have done that we need to identify the difference between a handled error and an unhandled error. We can never fully anticipate all errors that will occur so we need to have a kind of safety mechanism to ensure that if errors do occur, they are handled accordingly. When we write code that achieves this we are *handling* errors.

In this first example we have an error because we are trying to divide 5 by 0. This is a common error that occurs in VBA. The code can be found in the “06_Debugging” module of the accompanying Access file. Test the code and see what happens. An ugly dialog box appears and gives us some information which may be useful to a developer but not to an end user.

```

1 Sub unhandledError()
2   Dim i As Integer
3   i = 5 / 0
4 End Sub

```

Figure 6.3

In the second example we have included an error handler. The snippet of code that says *On Error GoTo error_handler* tells the IDE that if an error is encountered the code should immediately jump to the section entitled *error_handler*: where we have some lines of code that bring up a much more informative and instructional dialog box (that we created and can modify to suit our means).


```

1 Sub handledError()
2 On Error GoTo error_handler
3     Dim i As Integer
4     i = 5 / 0
5 Exit_Sub:
6     Exit Sub
7 error_handler:
8     MsgBox "There has been an error. Please try running the code again
9 or reloading the form."
10    Resume Exit_Sub
11 End Sub

```

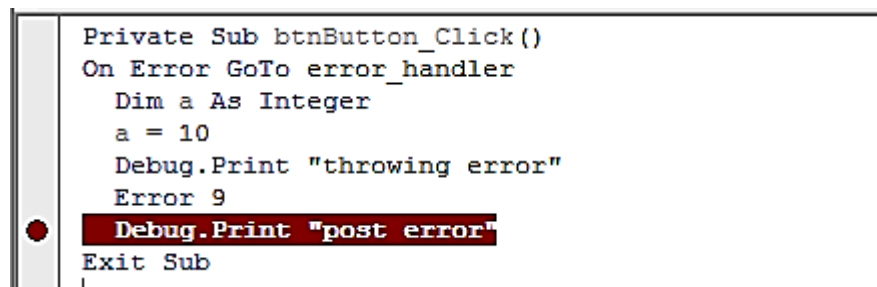
Figure 6.4

Error handling such as this is typical in VBA code and is the mark of a bullet-proofed application.

Breakpoints

A breakpoint is a marker one places on the code and at which point execution breaks and stops to allow the debugger to operate. There are many cases when such an activity is really useful.

Say you have a long calculation and you know there's an error in it but don't know where. By clicking on the column where the red dot is displayed below, the row will become highlighted indicating a breakpoint. Once the breakpoint is reached the code is paused and the VBA editor has gone into debug mode.



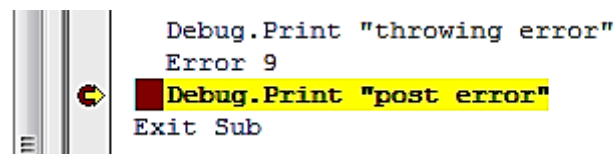
```

Private Sub btnButton_Click()
On Error GoTo error_handler
Dim a As Integer
a = 10
Debug.Print "throwing error"
Error 9
Debug.Print "post error"
Exit Sub

```

Figure 6.5

Debug mode in the VBA editor isn't much different to normal mode except that the debug control bar's controls are enabled and you can see a yellow line indicating the line of code waiting to be executed. In order to resume executing code from this point, press F5. If you would like to step through the code one line at a time you can press F8.



```

Debug.Print "throwing error"
Error 9
Debug.Print "post error"
Exit Sub

```

Figure 6.6

Debug Control Bar

The debug toolbar is your next companion in battle. When this bar is active it allows you to step through your program and examine it in great detail.

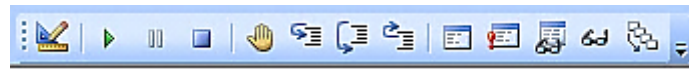


Figure 6.7













	The Play button tells the executive to continue running from the line that is currently highlighted.
	If the debugger isn't yet engaged you can pause the execution and enter debug mode immediately.
	Stop forces an immediate cessation of execution and the call stack to be cleared.
	The hand toggles a breakpoint on current line.
	On a line which contains a user-defined function the Step-In follows execution into the next function or procedure.
	Or rather than stepping into the function one can jump over the function, allowing it to execute as needed, and take up debugging once the sub-method has finished.
	Step out tells the debugger to continue executing the rest of the current procedure until it completes.
	Displays the Locals Window which displays all local variables in use and their values.
	Toggles the visibility of the immediate window.
	Toggles the Watches Dialog box. This box is navigable allowing you to drill down into all local variables currently in use and inspect them in minute detail.
	Quick Watch creates a quick watch item using the currently selected variable.
	Call Stack displays a list of functions and procedures that have lead up to this point and will be returned to.

Figure 6.8

Immediate Window

As mentioned before the immediate window is fantastic for testing code snippets, but it can also be used as a great debugging tool. Here are a few simple commands:

```
Debug.print "Hello World"  
  
Print "Foobarbar"  
  
? "bar foo foo bar bar"
```

Figure 6.9

? and Debug.Print

Although we use these 3 methods to print from the immediate window we must use Debug.Print when inside the code window.

: to concatenate commands

Another shorthand notation is ":" which allows multiple commands on one line. As the Immediate Windows doesn't execute commands over several lines – just one line – you can use ":" to overcome this limitation. So now looping and conditionals structures are available to you:

```
Debug.print "Hello World": print "Foobar": ? "barfoo"  
  
For t=1 to 10: ?t:Next  
  
T=True: if T then ?"It's true": else : ?"it's false :("  
  
T=False: if T then ?"It's true": else : ?"it's false :("
```

Figure 6.10

; to concatenate strings

Just like in the Code Window you can also use ";" to concatenate Strings together rather than "+".

Note

You cannot use the "Dim" keyword in the Immediate Window. The good news though is that this is because it is not required; just assign values to variables as required.

```
` this will not work  
Dim t As Integer: For t=1 to 10: ?t:Next  
  
` this will work  
T=0 : For t=1 to 10 : ?t : Next
```

Figure 6.11

Call a Procedure

To execute a procedure you need only type its name. If you want to highlight the fact and document that you are actually calling a procedure and function you use the word *Call* before the method's name. One note of caution, when trying to call a function, *Call* does not return any values and will give an error if you try to capture a function's return value. *Call* only calls a function as if it were a procedure.

```
Public Function testAA() As String
    testAA = "done"
End Function

` this will not work
Call testAA()
Call (testAA)
A = call(testAA)

` this will work
call testAA
a=testAA()
testAA
```

Figure 6.12

Immediate Window is in Scope

Commands you type in the Immediate Window are executed immediately and in scope. If you are not debugging, the window will operate at Global Scope; if you are debugging, the window operates at that function or procedure Scope.

And a final comment to make; when working in the Immediate Window any code you write using variables of the code being debugged will cause the program's variables to be changed. This is a highly desirable feature to help resolve bugs.

Code Window Pop-ups

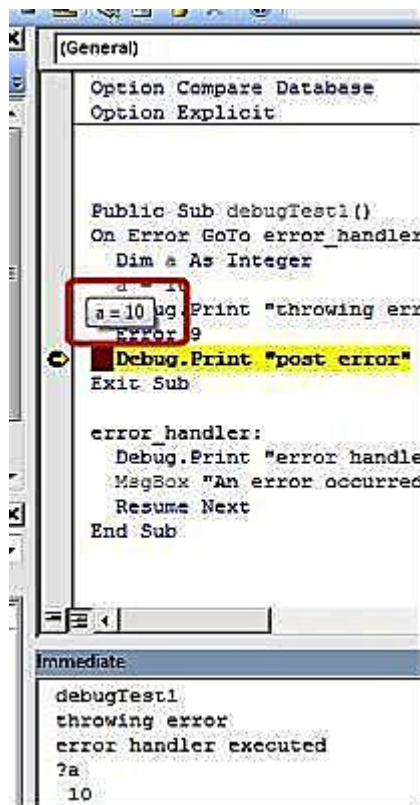


Figure 6.13

In this example we are auto-generating an error. The line that says Error 9 will generate:

“Subscript out of range”. But we are using an error handler to enable us to “trap” the error.

In this simple procedure “a” has been assigned the value of 10. At the debugged statement (the red and yellow line) the code has halted. Place the cursor over any of the variables in the procedure and the value will be displayed in a hint. This is very useful when reading code as it is a quick way to determine the values of any variables.

In the immediate window below we’ve also added the statement: `?a` which prints 10.

Here we have placed a break point on the line that reads `Debug.print “post error”`. The code has halted execution and now we will use the immediate window to manipulate the variable “a”.

In the immediate window we’ve assigned the value 21 to “a” and printed it out to verify this has happened. Next we placed the cursor over the variable “a” and the IDE tells us the variable’s value in situ, `a=21`.

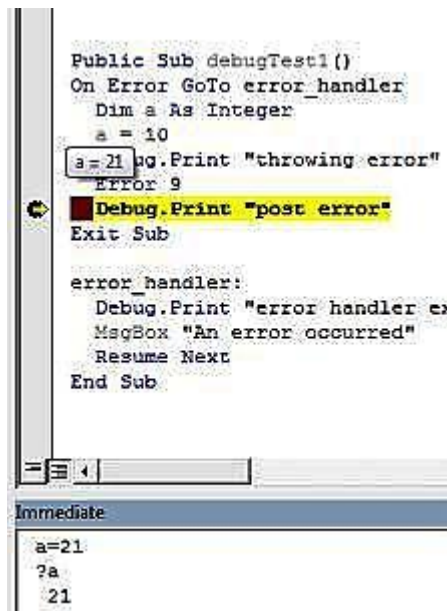


Figure 6.14

Watches Window

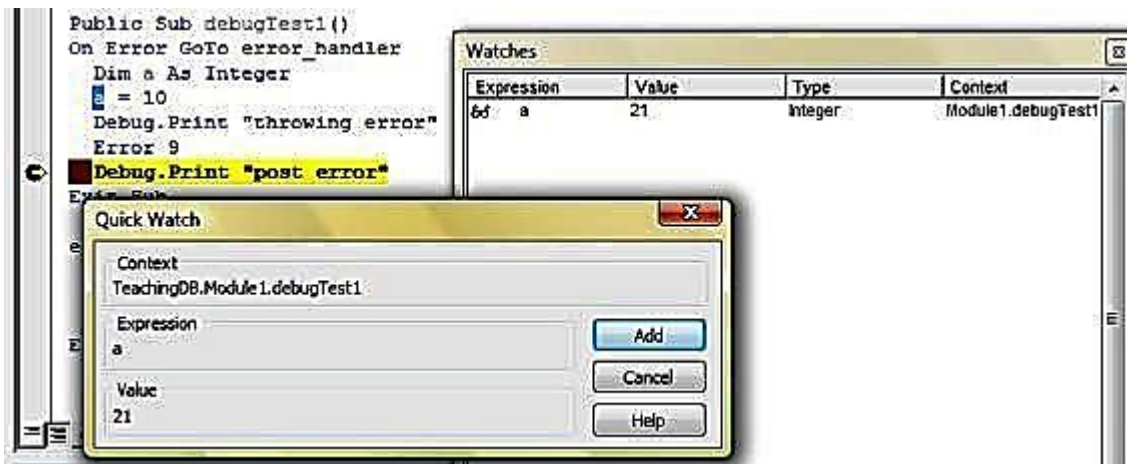


Figure 6.15

The Quick Watch feature and Watches Window allows us to see a set of variables without having to place the cursor over anything or type anything in the immediate window.

To add a watch to the Watches Window highlight the variable you want to watch (“a” in this case) and click the Quick Watch button or Shift+F9.



Figure 6.16

Above you can see that “a” is now in the Watches Window, displaying its value, type, context and other details. When you bug out of the procedure “a” will become <Out of Context>.

VBE Editor Options

VBA has a concise set of options and tools which you can set to change behaviour of the editor and debugger. All are useful tools to help make coding easier and quicker for developers.

To access the Editor Options click on the Tools menu and select Options...

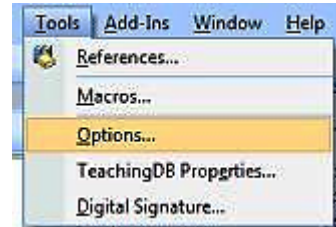


Figure 6.17

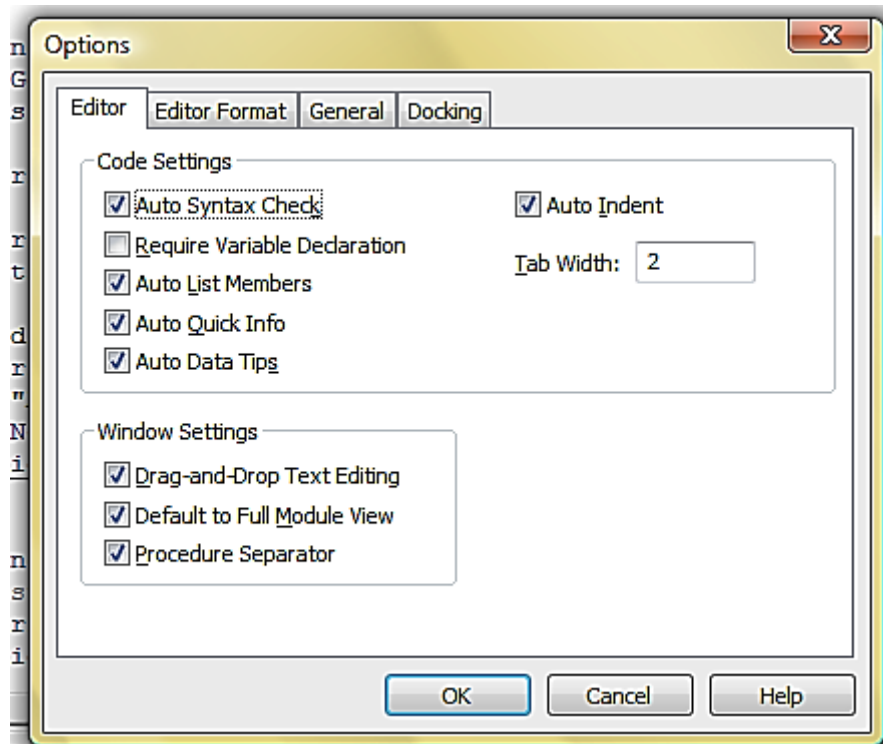


Figure 6.18

- **Auto Syntax check** – check as you type syntax checker.
- **Require Variable Declarations** – this adds “Option Explicit” to the top of all modules and is a good one to always have ticked.

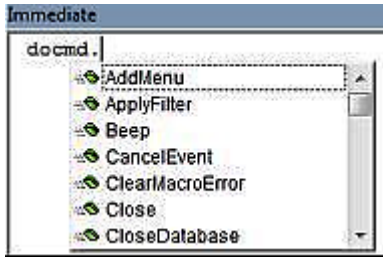


Figure 6.19

- **Auto List Members**

The . dot operator allows you to access members / properties / fields of an object.

Here DoCmd members are shown. This is a feature of the IDE and can be done on any object or class.

- **Auto indent**

Auto indent option forces the Editor to indent your code which makes ascertaining the beginning and end of nested methods easier.

In the image to the right there is a For...Next loop and because the inner code block is indented it is easy to make out what is being executed and when.

```
Public Function debugTest3()
    Dim a As Integer, b As Integer
    b = 10: a = 1

    For a = 1 To b
        Debug.Print a
        a = a + 1
    Next
    Debug.Print "The End"
End Function
```

Figure 6.20

- **Break On All Errors (General Tab)**

This option tells the debugger to break execution and let the programmer see the debugger and investigate what's happening on ALL errors (handled or otherwise). If this option is not checked the debugger will not cut in and the program is left to perform default actions based on the nature of the error.

- **Compile On Demand (General Tab)**

Compilation is an operation that converts our VBA into executable code. Compile On Demand should generally always be on, and will be executed by the IDE or the module compiled when a function or procedure held within is required.

- **Auto-Quick Info (Editor Tab)**

Quick information enables the edit to provide you with the signature of a method displaying its accepted arguments and their data types – including enumeration types.

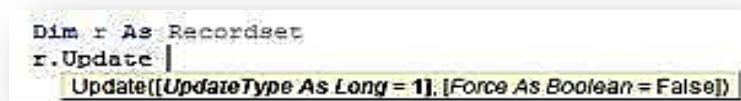


Figure 6.21

The hint below the Update tells you all the arguments this method requires. Both arguments in this method are optional as they have [] around them. UpdateType is of data type Long with a default value of 1, Force is of Boolean data type with a default value of False.

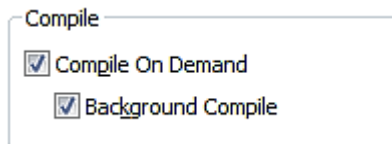
- **Auto Data Tips (Editor Tab)**

Earlier we saw that placing the cursor over a variable in debug mode displays a hint. This option turns that feature on and off.

Compilation Explained

To round off the unit we will look at compilation.

Compilation is the act of converting our human readable code (VBA) into code the computer understands. It may also be that your code is compiled into an intermediary format often called object code. Either way this just illustrates that our programs are actually just a human understandable representation of what we are telling our computers to do.



Generally you will not notice the compiler as these setting on the left are set to automatically compile during execution and whilst you type.

Figure 6.22

You can explicitly force VBA IDE to compile all modules in the project.

Before you release your Access product to fellow users it is always a good idea to explicitly execute the Compile item in Debug menu.

One has often found debug code lingering around forms and modules that would later cause problems, especially once multiple users are using the file.

Compiling before release also ensures the VBA code executes as fast as possible.



Figure 6.23

That's really all you need to know about compilation. For interested readers there is a little more below about ACCDE files.

Advanced Compilation and ACCDE

While compilation as described above allows your application to execute in a multiuser environment, it leaves all the form data, report data and VBA code available to be edited by anyone with a full installation of MS Access. You can use the runtime/command-line switch in a shortcut to reduce the chance of a user stumbling across the designer tools.

Alternatively we can strip out all design information make it impossible for users to edit forms and modules. If you do this procedure **make sure** you keep a backup of the accdb file; if you lose it you will never get the design information back.

To Create an ACCDE file:

<p>Click on the File tab in the Ribbon to expose BackStage view.</p>	
<p>Click on Save & Publish.</p>	
<p>Click on Make ACCDE.</p>	

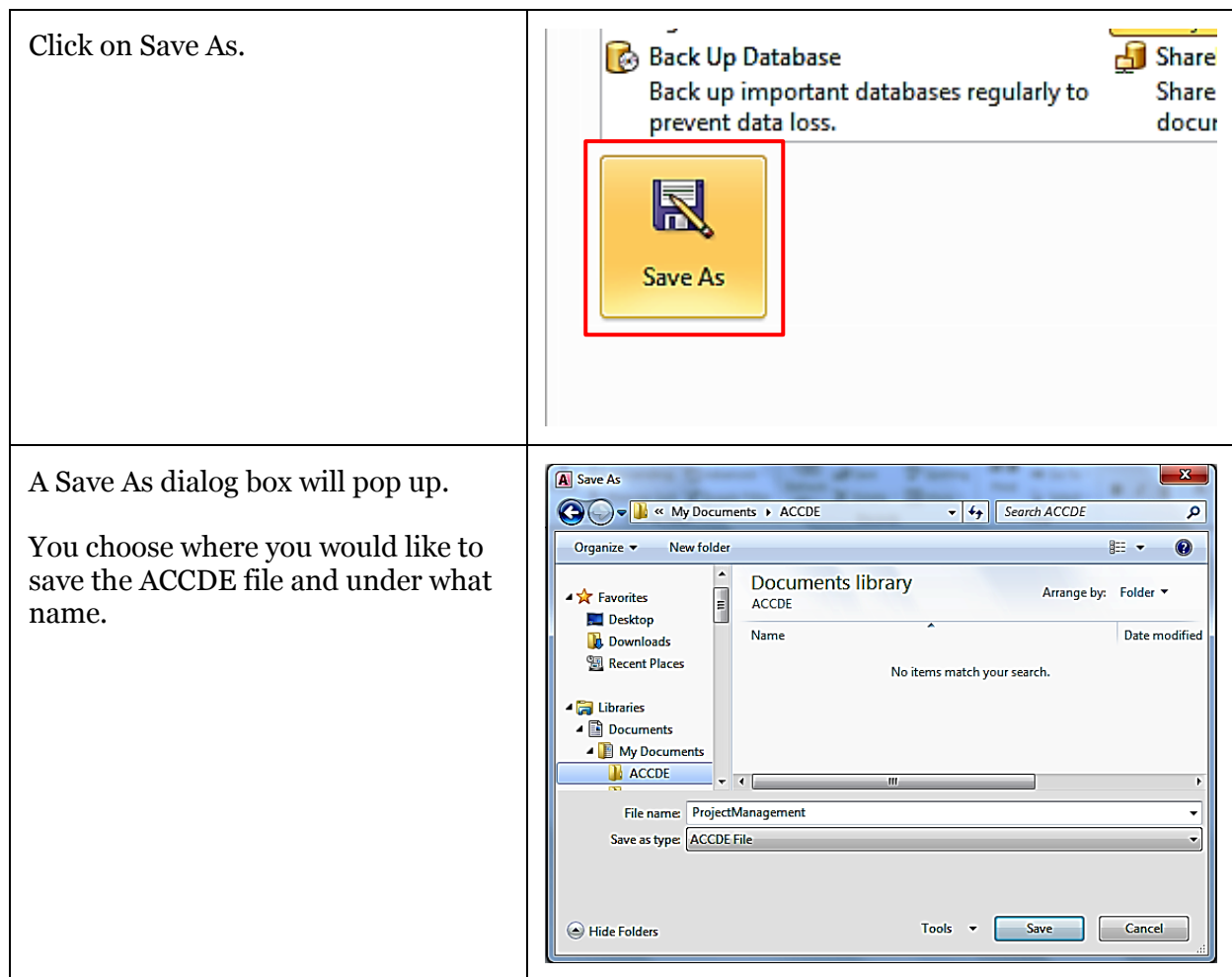


Figure 6.24

Why use an ACCDE file?

Creating an ACCDE file removes all design code so that Forms may only be opened in Form View and Modules cannot be debugged. So even if the SHIFT Key entry method is used no Forms, Reports or Code can be changed.

This is useful for the following situations:

- You don't want users to change your forms, reports or code.
- It creates a more stable Access Database for multi-user environments.
- You want to protect your intellectual property.
- You want to publish your Access Database.

Multi-Users Environments

From first-hand experience in multi-user environments you are advised to only give end-users ACCDE or MDE files and split your back- and front-ends. Giving access to the ACCDB runs the possibility of inadvertent changes to form properties – e.g. when a user applies a filter to the form – and when saved this very well may corrupt your database.






Corrupt databases *can be recovered* but on the off-chance it is not possible it is not worth the risk. You'll lose a day's work at least (if not everything) if backups of your file server haven't been kept.

Questions

1. Which of the following describes a breakpoint?
 - a. A red dot on the left.
 - b. A green bar across the highlighted code.
 - c. A point in the code that interrupts execution.
 - d. Max number of function called before crashing.
 - e. A corrupt database.

2. Which of the following describe debugging?
 - a. Ridding code of errors.
 - b. Cleaning the mouse.
 - c. A Honey trap for VBA code.
 - d. Inspecting run-time code for errors.
 - e. Command-line interface for IDE functions.

3. What do the following icons mean

4. Explain the uses of the following characters in the Immediate Window

?	
:	
;	
Dim	

5. What is the result of executing the following code

```
t=-1 : For t=t to 5 : ?t : Next t
```

6. When typing the DoCmd object and hitting “.” What may happen?

- a. Object members are listed.
- b. Quick Info may display.
- c. Computer may beep at you.
- d. Compiles your VBA code.
- e. Resets the IDE’s editor tools.

7. In the immediate window what can you not do?

- a. Use the keyword Dim to instantiate objects.
- b. Access scoped variables.
- c. Inspect variable values.
- d. Execute snippets of code.

8. What is wrong with the following code?

```
? a= ; a; "oioi"; "."
```

9. What does the Debug menu item **Clear All Breakpoints** do?

10. How to use quick watch?

11. Which of the following lines will cause an error?

- a. A = myFunc(a)
- b. A = myFunc a
- c. C = A + myFunc (12)
- d. Call myFunc(12)
- e. C = myFunc(12)

12. True or False?

- a. The debugger will compile and execute your code.
- b. The IDE will assist with most syntax problems.

- c. Option Explicit should be used as little as possible.
- d. Debugging Modules involves (by default) a lot of red and yellow lines.
- e. Watches window displays the time.
- f. The immediate window is always in execution scope.
- g. In debug mode placing the cursor over a variable displays its value.

13. An ACCDE is a compiled version of an ACCDB file?
14. What's the difference between an ACCDB and MDB?
15. Breakpoints are activated at run-time?
16. ACCDE files do not have breakpoints? True or False
17. True or False? Using Debug.Print slows down your application.
18. True or False? You should tick the box that says "Break on All Errors" when handing over the database to an end user.
19. True or False? Functions can't be called from the immediate window.
20. True or False? The Debug control bar is always visible and cannot be removed.

Answers - Functions, Sub Procedures and Arguments

1. If you want a returned value
2. a
3. b, c, d
4. a-d, b-c, c-a, d-b
5. a
 - a. sqr
 - b. date()
 - c. now()
 - d. datediff
 - e. CStr
 - f. IsEmpty
 - g. IsNull
 - h. DSum
 - i. CDate
6. TypeName
7.
 - a. Date
 - b. Function
 - c. Yes
 - d. i
8. a-b, b-e, c-c, d-a, e-d
9. Now() has a time element, Date() has only date
10. 2
11. Function
 - a. Function textAddNumber (myText As String, myNumber as Long) As String
 - b. textAddNumber = myText + " " + CStr(myNumber)
 - a. End Function
 - b.
 - c. Function textAddNumber (myText As String, myNumber as Long) As String
 - d. textAddNumber = myText; " "; CStr(myNumber)
 - c. End Function
12. Sub
 - e. Sub calculate(a As Integer, b String, c String)
 - f. Dim houseNo As Integer
 - g. Dim telNum As String
 - h. Dim Surname As String
 - i. Dim d As String
 - j. D = CStr(houseNo) + telNum + Surname
 - k. Debug.print d
 - l. End Sub
13. eg. DLookup("[surname]","[pupils]","id=1192")
14. eg. DCount("[*]","[students]", "left([telephone],3)="555")
15. see page on dates for answers
16. All are true :)
17. Function
 - a. Function getElement(myArray as Variant, i as Integer)
 - b. getElement = myArray(i)

c. End Function

18. Multi choice

a. A

b. B

c. B

d. C

e. C

f. A

19. Button 3 doesn't have an event procedure, specifically no onClick or onDbClick

20. Multiple answers

a. Optional means name doesn't have to be passed






b. Julia

c. Donothing4

d. "Morning Dave. My name is Hal 9000"

Answers - Debugging

1. (a) and (c)
2. (a) and (d)
- 3.

	This button allows the developer to “step over” a function.
	This button toggles the visibility of the immediate window.
	On a line which contains a user-defined function the Step-In follows execution into the next function or procedure.
	The hand toggles a breakpoint on current line.
	Quick Watch creates a quick watch item using the currently selected variable.

4.

?	Prints out a variable or return value of a function.
:	Concatenates commands.
;	Concatenates a string.
Dim	Cannot be used in the immediate window.

5. It prints out -1, 0, 1, 2, 3, 4, 5 in the immediate window.
6. (a)
7. (a)
8. A requires a value - ? $a=1$; a ; "oioi"; "."
9. It removes all breakpoints from the code in the active module.
10. Highlight a variable you would like to watch click on the quick watch button in the debug bar.
11. (b)
12. See answers below.
 - a. False
 - b. True
 - c. False
 - d. False
 - e. False
 - f. False
 - g. True
13. True
14. MDB are legacy files, pre Access 2007.
15. True
16. True
17. True
18. False – This should only be ticked when a developer uses it.
19. False
20. False

