VBA Made Easy

Debugging

06

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.

Contents

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 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:

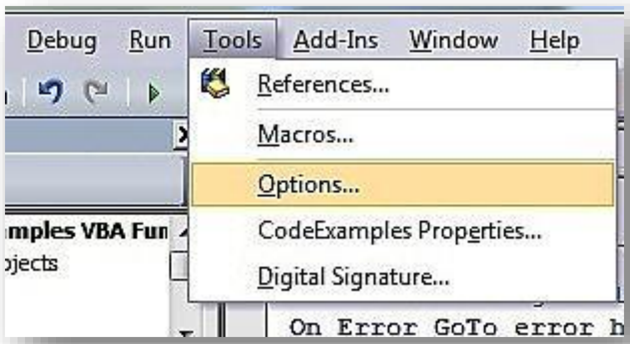
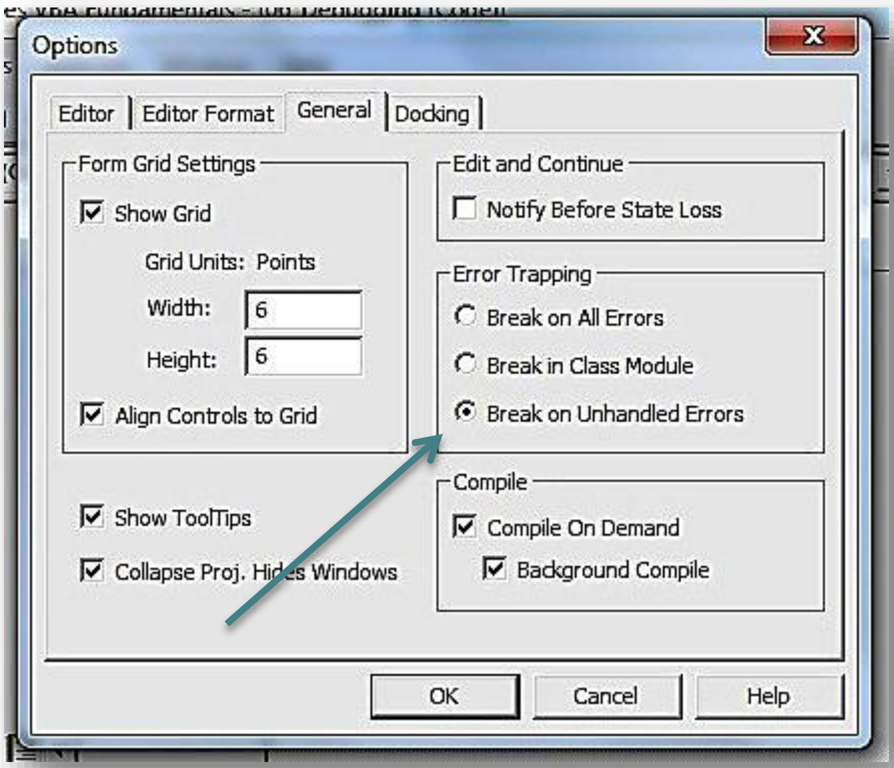


Figure 6.1

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


Options

Editor | Editor Format | General | Docking

Form Grid Settings

[✓] Show Grid

Grid Units: Points

Width: 6

Height: 6

[✓] Align Controls to Grid

[✓] Show ToolTips

[✓] Collapse Proj. Hides Windows

Edit and Continue

[ ] Notify Before State Loss

Error Trapping

( ) Break on All Errors

( ) Break in Class Module

(●) Break on Unhandled Errors

Compile

[✓] Compile On Demand

[✓] Background Compile

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.

```
Sub unhandledError()
Dim i As Integer
i = 5 / 0
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).

```
Sub handledError()
On Error GoTo error_handler
    Dim i As Integer
    i = 5 / 0
Exit_Sub:
    Exit Sub
error_handler:
    MsgBox "There has been an error. " & _
     "Please try running the code again or reloading the form."
    Resume Exit_Sub
End Sub
```

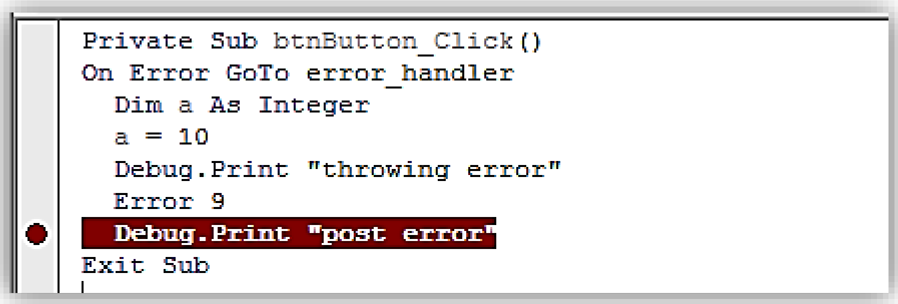
Figure 6.4

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.

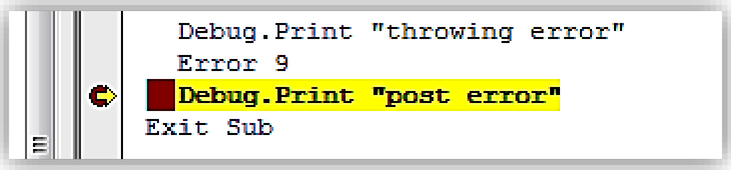
Imagine you have a long calculation and you know there's an error in it but don't know where. By clicking on the column where the red dot is displayed below, the row will become highlighted indicating a breakpoint. Once the breakpoint is reached the code is paused and the VBA editor will go into debug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

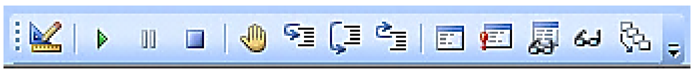


Figure 6.7

▶	The Play button tells the executive to continue running form 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 on 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 Window 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
```

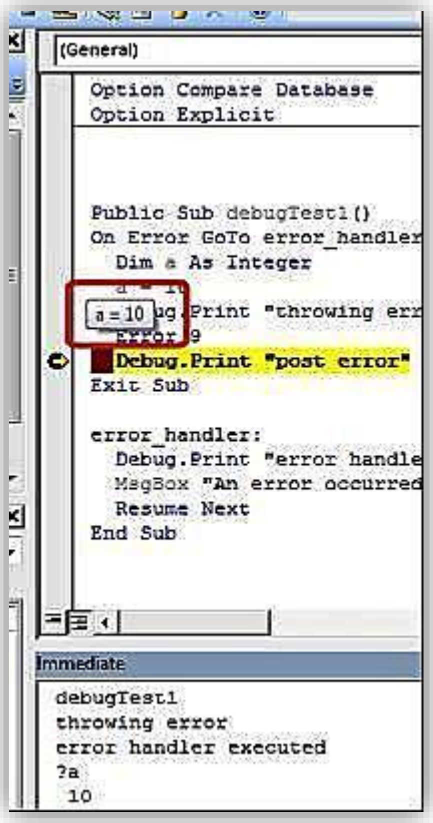
Figure 6.12

Immediate Window 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



```
(General)
Option Compare Database
Option Explicit


Public Sub debugTest1()
On Error GoTo error_handler
  Dim a As Integer
  a = 1
a = 10 ug.Print "throwing err
  Error 9
  Debug.Print "post error"
Exit Sub

error_handler:
  Debug.Print "error handle
  MsgBox "An error occurred
  Resume Next
End Sub
```

```
Immediate
debugTest1
throwing error
error handler executed
?a
 10
```

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.

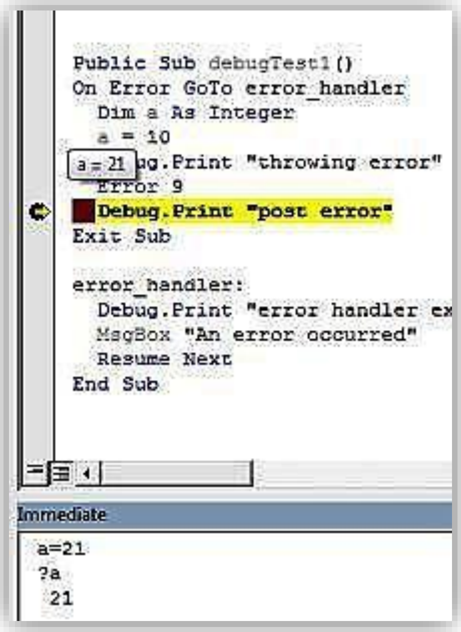
In this simple procedure “a” has been assigned the value of 10. At the debugged statement (the red and yellow line) the code has halted. Place the curser over any of the variables in the procedure and the value will be displayed in a hint. This is very useful when reading code as it is a quick way to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

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

Figure 6.13

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 curser over the variable “a” and the IDE tells us the variable’s value in situ, a=21.



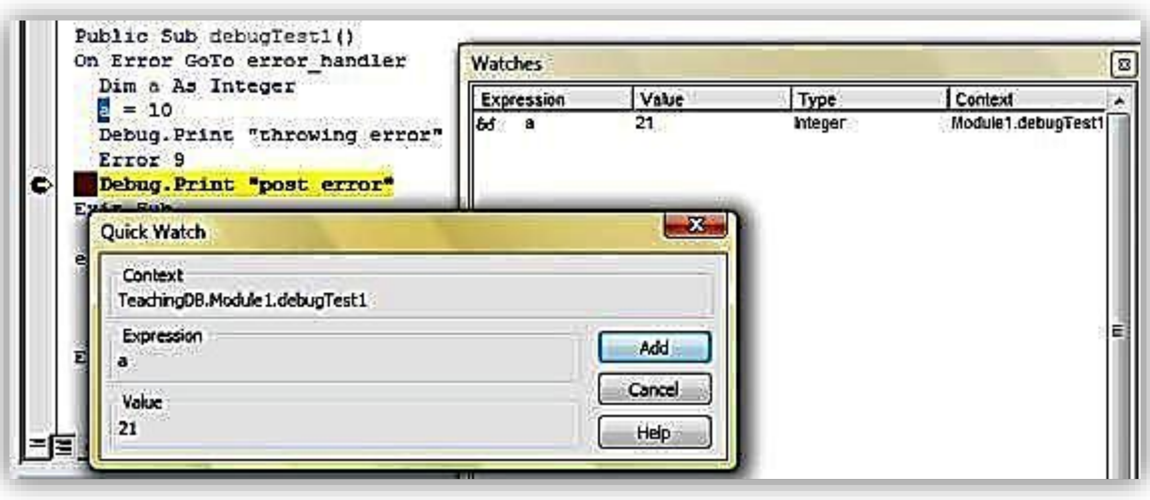
```
Public Sub debugTest1()
On Error GoTo error_handler
  Dim a As Integer
  a = 10
a = 21 ug.Print "throwing error"
  Error 9
  Debug.Print "post error"
Exit Sub

error_handler:
  Debug.Print "error handler ex
  MsgBox "An error occurred"
  Resume Next
End Sub
```

```
Immediate
a=21
?a
 21
```

Figure 6.14

Watches Window



```
Public Sub debugTest1()
On Error GoTo error_handler
  Dim a As Integer
  a = 10
  Debug.Print "throwing error"
  Error 9
  Debug.Print "post error"
```

Expression	Value	Type	Context
a	21	Integer	Module1.debugTest1

Figure 6.15

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

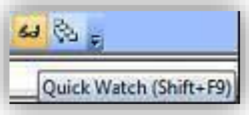
To add a watch to the Watches Window highlight the variable you want to watch ("a" in this case) and click the Quick Watch button or Shift+F9.	

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.

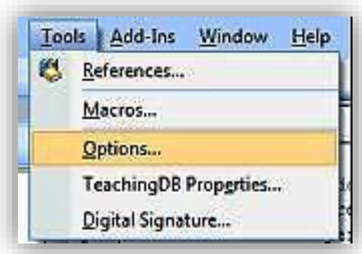
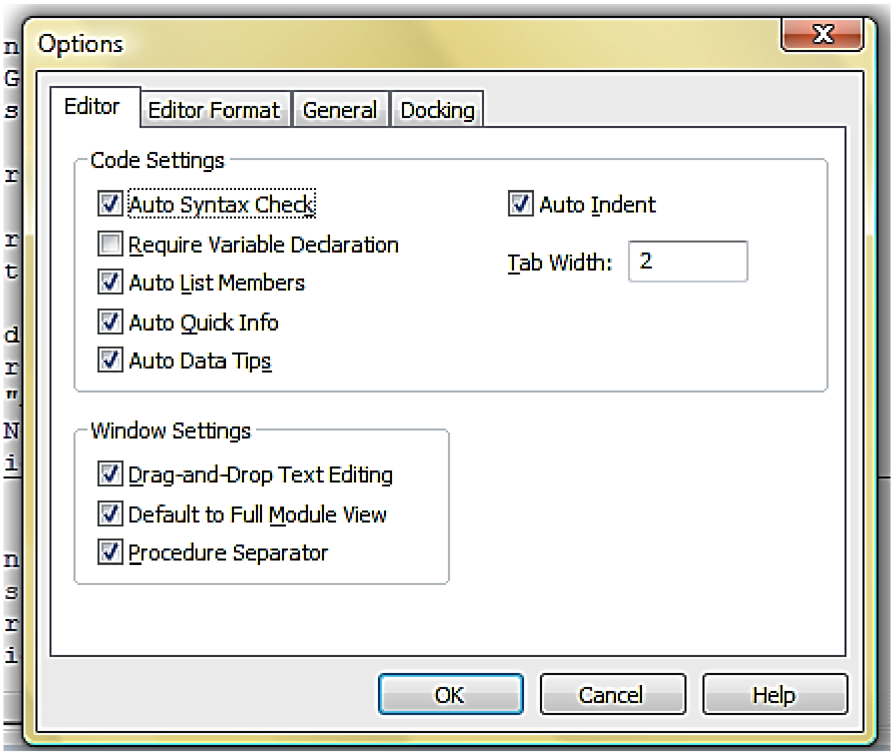
To access the Editor Options click on the Tools menu and select Options…	 Tools Add-Ins Window Help References... Macros... Options... TeachingDB Properties... Digital Signature...

Figure 6.17



Code Settings

- [✓] Auto Syntax Check
- [ ] Require Variable Declaration
- [✓] Auto List Members
- [✓] Auto Quick Info
- [✓] Auto Data Tips
- [✓] Auto Indent

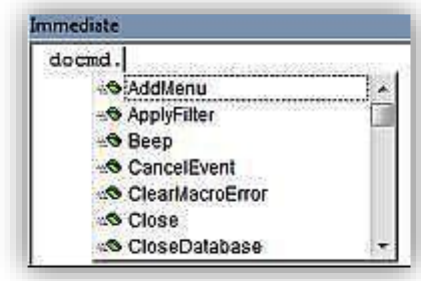
Tab Width: 2

Window Settings

- [✓] Drag-and-Drop Text Editing
- [✓] Default to Full Module View
- [✓] Procedure Separator

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 to always have ticked.



```
docmd.
```

- **Auto List Members**

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

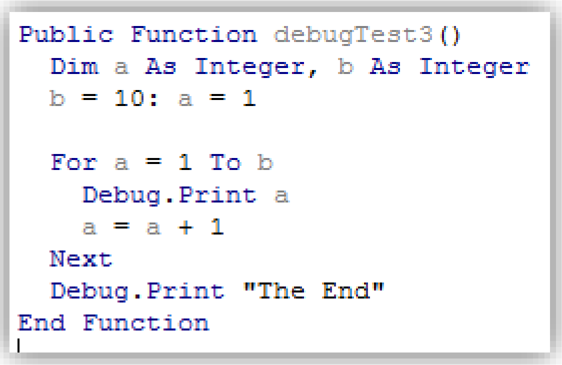
Here DoCmd members are shown. This is a feature of the IDE and can be done on any object or class.

Figure 6.19

- **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)**

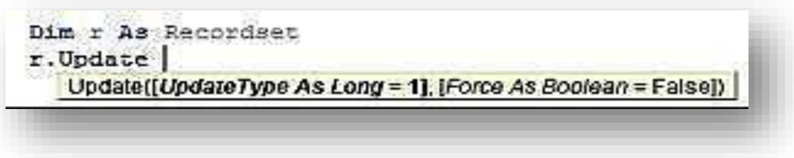
This option tells the debugger to break execution and let the programmer see the debugger and investigate what's happening on ALL errors (handled or otherwise). If this option is not checked the debugger will not cut in and the program is left to perform default actions based on the nature of the error.

- **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.



```
Dim r As Recordset
r.Update
Update([UpdateType As Long = 1], [Force As Boolean = False])
```

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 curser 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.

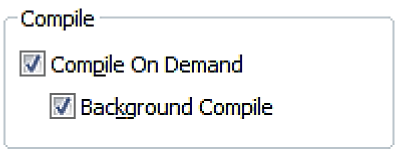
 Compile: ☑ Compile On Demand, ☑ Background Compile	Generally you will not notice the compiler as the settings on the left are set to automatically compile during execution and whilst you type.

Figure 6.22

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

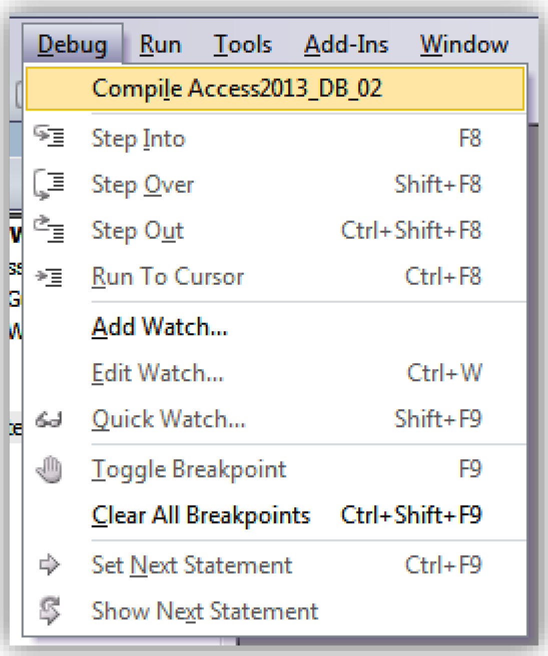
Before you release your Access product to fellow users, it is always a good idea to explicitly execute the Compile item in Debug menu. Compiling before release also ensures the VBA code executes as fast as possible.	 Debug Run Tools Add-Ins Window Compile Access2013_DB_02 Step Into F8 Step Over Shift+F8 Step Out Ctrl+Shift+F8 Run To Cursor Ctrl+F8 Add Watch... Edit Watch... Ctrl+W Quick Watch... Shift+F9 Toggle Breakpoint F9 Clear All Breakpoints Ctrl+Shift+F9 Set Next Statement Ctrl+F9 Show Next Statement

Figure 6.23

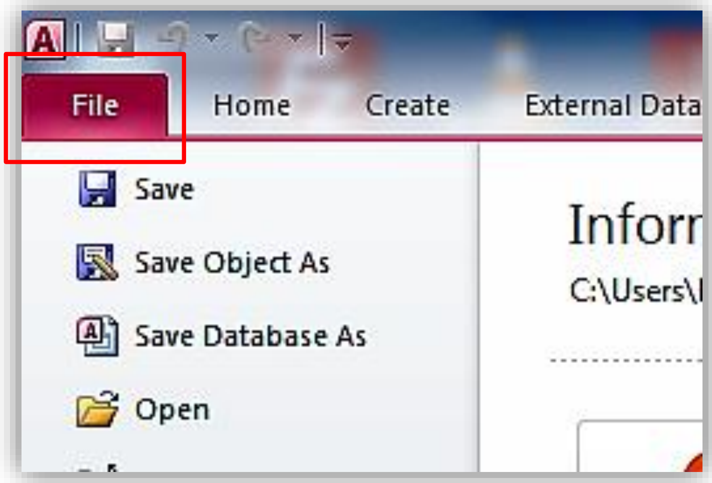
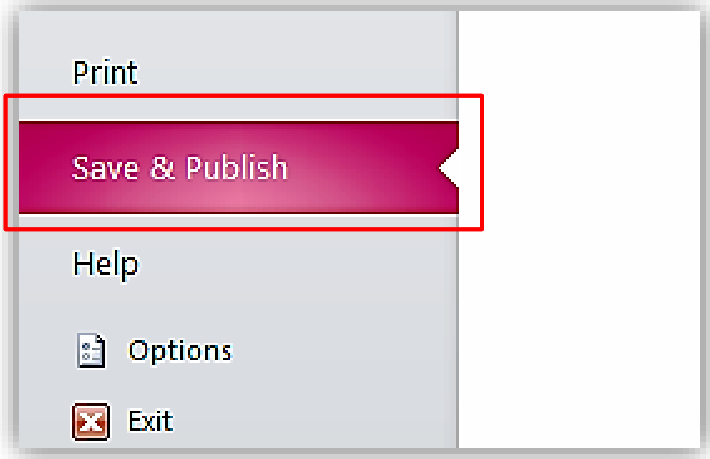
That's really all you need to know about compilation. For interested readers there is a little more below about ACCDE files.

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:

Click on the File tab in the Ribbon to expose BackStage view.	 File Home Create External Data Save Save Object As Save Database As Open Inforr C:\Users\
Click on Save & Publish.	

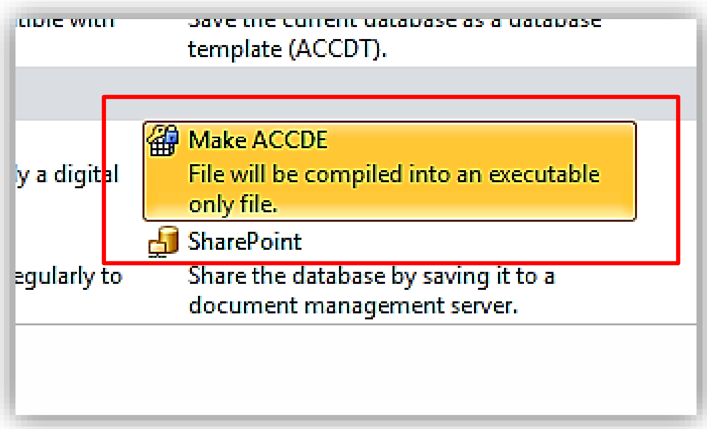
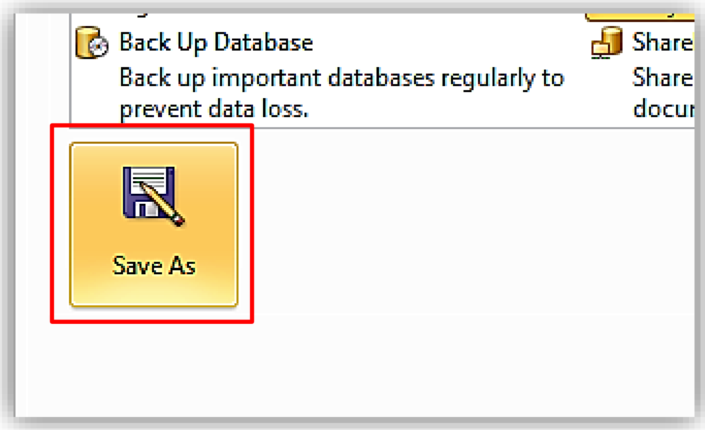
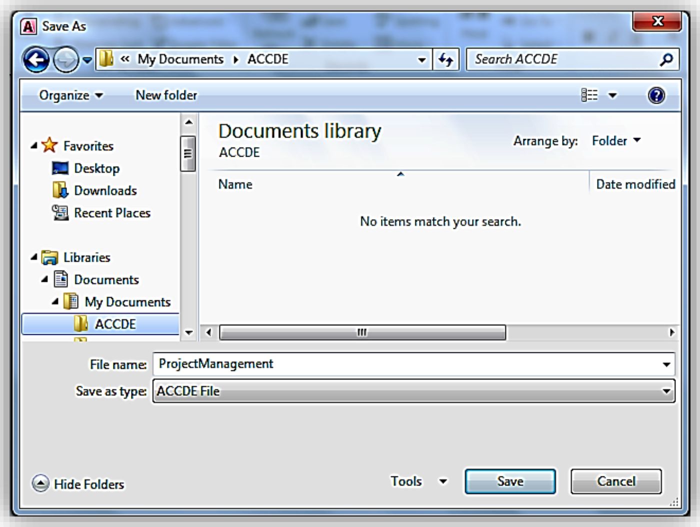
Click on Make ACCDE.	 template (ACCDT). Make ACCDE File will be compiled into an executable only file. SharePoint Share the database by saving it to a document management server.
Click on Save As.	 Back Up Database Back up important databases regularly to prevent data loss. Save As
A Save As dialog box will pop up. You choose where you would like to save the ACCDE file and under what name.	 Save As My Documents ▸ ACCDE Documents library ACCDE No items match your search. File name: ProjectManagement Save as type: ACCDE File

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 curser 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 databse 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

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