


## 18.1 Introduction: Learning the basics of programming

Programming can be enormously complex and difficult, or it can be easy and straightforward. In many cases, the difference lies in the choice of tools used for writing the program. In other cases, it is a question of scale—large projects are unavoidably complex.

In this project, we are going to focus on solving small problems using an easy-to-use programming language. It is important to recognize, however, that basic programming concepts remain the same regardless of language or project scale. The programs you create here are trivial, but they introduce a handful of programming constructs that can be found in any “third generation” language, not just VISUAL BASIC.

 Strictly speaking, the language that is included with ACCESS is not VISUAL BASIC—it is a subset of the full, stand-alone VISUAL BASIC language (which MICROSOFT sells separately). In ACCESS version 2.0, the subset is called “ACCESS BASIC”. In version 7.0 and above, it is slightly enlarged subset called “VISUAL BASIC FOR

APPLICATIONS” (VBA). In the context of the simple programs we are writing here, these terms are interchangeable.

ACCESS provides two ways of interacting with the VBA language. The most useful of the two is saved **modules** that contain VBA **procedures**. Procedures are batches of programming commands that can be executed to do interesting things like process transactions against master tables, provide sophisticated error checking, and so on.

The second way to interact with VBA is directly through the interpreter. When you type a statement into the interpreter, it is executed immediately. Although there is little business use for this feature, it does make learning the language and testing new statements easier.

In the first part of this lesson, you are going to invoke ACCESS’ VBA interpreter and execute some very simple statements. In the second part of the tutorial, you are going to create VBA modules to explore generic programming constructs such as looping, conditional branching, and parameter passing.



## 18.2 Learning objectives

- invoke and use the debug/immediate window
- understand the difference between fundamental program constructs (statements, variables, the assignment operator, and predefined functions)
- understand the difference between subroutines and functions
- create a module containing VBA code
- gain experience with looping and conditional branching constructs
- use the VBA debugger in ACCESS
- understand the difference between an interpreted and compiled programming language

## 18.3 Exercises

### 18.3.1 Invoking the interpreter

- ➔ Click on the **Modules** tab in the database window and press **New**.

This opens the module window which we will return to in [Section 18.3.3](#).

- ❓ You have to have a module window open in order for the **debug window** to be available from the menu.

- ➔ Select **View** → **Debug Window** from the main menu. Note that **Ctrl-G** can be used in version 7.0 and above as a shortcut to bring up the debug window.



In version 2.0, the “debug” window is called the “immediate” window and you use **View** → **Immediate Window** to activate it.

### 18.3.2 Basic programming constructs

In this section, you will use the VBA interpreter to explore some fundamental programming constructs and VISUAL BASIC syntax.

#### 18.3.2.1 Statements

Statements are built around special keywords in a programming language that do something when executed. For example, the **Print** statement in VBA prints an expression on the screen.

- ➔ In the debug window, type the following:

```
NL Print "Hello world!";
```



The ↵ symbol at the end of a line means “press the **Return** or **Enter** key”. From this point forward, assume that each line is followed by an **Enter**.



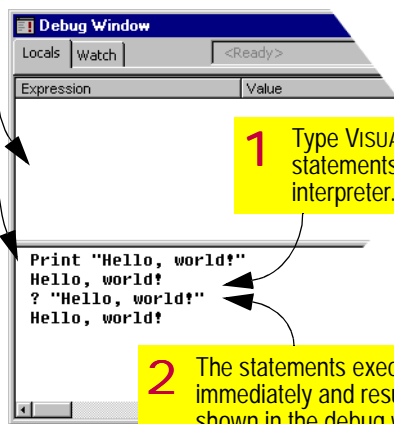
In VBA (as in all dialects of BASIC), the question mark (?) is typically used as shorthand for the **Print** statement.

➔ Type the following into the debug window:  
**NL ? "Hello world!"**

As shown in [Figure 18.1](#), the result is identical to first **Print** statement.

FIGURE 18.1: Interacting with the VISUAL BASIC interpreter.

? The debug window in version 8.0 is shared with the "locals" window.



### 18.3.2.2 Action statements

Actions are special statements that allow programmers to use elements of ACCESS' macro language in VBA programs. Some actions are stand-alone (such as the **MsgBox** action below) and others are actually methods of the **DoCmd** object (see [Section 18.4.1](#) for more information on the evolution of VBA and its increasing use of object-oriented concepts). Although the inclusion of so many different types of statements results in a conceptual mess, it provides programmers with a great deal of flexibility.

? At this point, you should ignore theoretical language issues and accept VBA as a powerful-but-inconsistent friend.

➔ Type the following into the debug window:  
**NL MsgBox "Hello, world!"**

The **MsgBox** macro action provides an easy way to create custom messages using the standard WINDOWS message box format.

The **DoCmd** object is just a kludge that brings VBA, object-orientation, and the macro language together. Ugly as it is, the **DoCmd** object has many methods that are worth learning about.



➔ Type the following into the debug window:

```
NL DoCmd.OpenForm "frmOrders"
```

You will notice that your order form opens in the background behind the debug window.



If you do not have a form called **frmOrders**, this method will result in an error.

### 18.3.2.3 Variables and assignment

A variable is space in memory to which you assign a name and a value. When you use the variable name in expressions, the programming language replaces the variable name with its assigned value at that particular instant.

➔ Type the following:

```
NL s = "Hello"
NL ? s & " world"
NL ? "s" & " world"
```

In the first statement, the variable named **s** is created and the string "Hello" is assigned to it. Recall the function of the concatenation operator from [Section 11.4.1](#). When the second statement is executed, VBA recognizes that **s** is a variable, not a string (since it is not in quotations marks). The interpreter replaces **s** with its value ("Hello") before executing the **Print** command. In the final statement, **s** is in

quotation marks so it is interpreted as a **literal string**.



Contrary to the practice in languages like C and PASCAL, the equals sign (=) is used to **assign** values to variables. It is also used as the **equivalence operator** (e.g., does  $x = y?$ ).



Within the debug window, any string of characters in quotations marks (e.g., "Hello") is interpreted as a literal string. Any string without quotation marks (e.g., **strName**) is interpreted as a variable or some other objects (such as a field) that is defined within the ACCESS environment.

### 18.3.2.4 Predefined functions

You were introduced to predefined functions in [Section 11.4.2](#). In this section, you are going to explore some basic predefined functions for working with numbers and text. The results of these exercises are shown in [Figure 18.2](#).

➔ Print the cosine of  $2\pi$  radians and then nest the cosine function within a conversion function to round to the nearest integer:

```
NL pi = 3.14159
NL ? cos(2*pi)
NL ? CInt(cos(2*pi))
```



➔ Convert a string of characters to uppercase:

```
NL s = "basic or cobol"
NL ? UCase(s)
```

➔ Extract the middle six characters from a string starting at the fifth character:

```
NL ? mid (s,5,6)
```

➔ Calculate the monthly payment for a \$12,000 loan over four years with 8.5% interest (nominal) compounded monthly:

```
NL ? Pmt(0.085/12, 4*12, -12000)
```



In VBA, as in ACCESS, case is ignored. As such `MID(s,5,6)` is identical to `mid(s,5,6)`. In addition, the amount of "whitespace" (space between elements of the statement) is irrelevant.

### 18.3.2.5 Remark statements

When creating large programs, it is considered good programming practice to include adequate internal documentation. In other words, you should include comments throughout your code to explain to others and your future self what the program is doing.

Comment lines are ignored by the interpreter when the program is run. To designate a comment in VBA, use an apostrophe to start the comment, e.g.:

```
NL `This is a comment line!
```

FIGURE 18.2: Using the VISUAL BASIC interpreter to test predefined functions.

The debug window can be enlarged by dragging the dividing line.

The argument for `cos()` contains an expression.

`UCase()` converts a string to uppercase.

`Mid()` extracts characters from a string.

Many specialized financial and scientific functions are available.

```
NL Print "Hello" `the comment starts here
```

The original REM (remark) statement from BASIC can also be used, but is less common.

```
NL REM This is also a comment (remark)
```

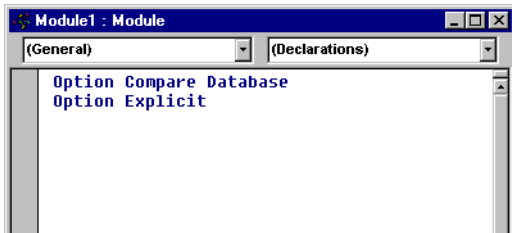


### 18.3.3 Creating a module

So far, you have written and executed VBA statements one at a time. A more useful technique is to bundle a number of VBA commands together in a procedure and run them in sequence. A module is a collection of statements that is saved with the database (like a form or a query).

- ➔ Close the debug window so that the declaration page of the new module created in [Section 18.3.3](#) is visible (see [Figure 18.3](#)).

FIGURE 18.3: The declarations page of a VISUAL BASIC module.



The two lines:

```
NL Option Compare Database
NL Option Explicit
```

are included in the module by default. The **Option Compare** statement specifies the way in which strings are compared (e.g., are “wire whisk” and “WIRE WHISK” considered identical?). The **Option Explicit** statement forces you to declare all your variables before using them (variable declaration is discussed in [Section 18.3.4.1](#)).

- 2 In version 2.0, ACCESS does not automatically add the **Option Explicit** statement. You should add it yourself to the declarations section.

A module contains a declaration page and one or more pages containing procedures. In VBA, there are two types of procedures: subroutines and functions. The primary difference between the two is that subroutines simply execute whereas functions execute and return a value (e.g., `cos()`).

- 2 In version 2.0, only one subroutine or function shows in the window at a time. You must use the **Page Up** and **Page Down** keys to navigate the module.



## 18.3.4 Creating subroutines with looping and branching

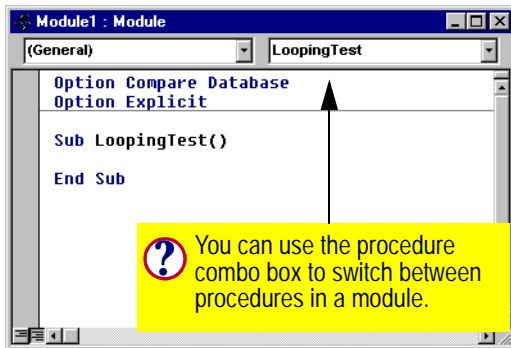
In this section, you will explore two of the most powerful constructs in computer programming: **looping** and **conditional branching**.

- ➔ Create a new subroutine by typing the following anywhere on the declarations page of the open module:

```
NL Sub LoopingTest()
```

Notice that Access creates a new section/page in the module for the subroutine, as shown in Figure 18.4.

FIGURE 18.4: Create a new subroutine.



### 18.3.4.1 Declaring variables

When you declare a variable, you tell the programming environment to reserve some space in memory for the variable. Since the amount of space that is required depends on the type of data the variable is expected to contain (e.g., string, integer, Boolean, double-precision floating-point, etc.), you have to include data type information in the declaration statement.

In VBA, you use the **Dim** statement to declare variables.

- ➔ Type the following into the space between the **Sub... End Sub** pair:

```
NL Sub LoopingTest()  
NL   Dim i as integer  
NL   Dim s as string  
NL End Sub
```

- ➔ Save the module as **basTesting**.

One of the most useful looping constructs is **For <condition>... Next**. All statements between the **For** and **Next** parts are repeated as long as the **<condition>** part is true. The index **i** is automatically incremented after each iteration.

- ➔ Enter the remainder of the **LoopingTest** program:



```

NL Sub LoopingTest()
NL   Dim i as integer
NL   Dim s as string
NL   s = "Loop number: "
NL   For i = 1 To 10
NL     Debug.Print s & i
NL   Next i
NL End Sub

```

➔ Save the module.



It is customary in most programming languages to use the **Tab** key to indent the elements within a loop slightly. This makes the program more readable.

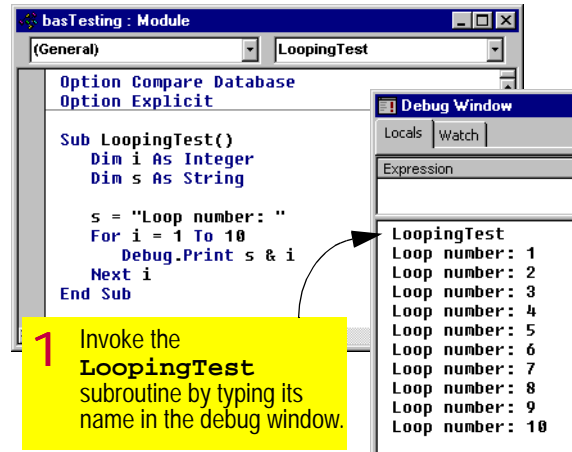
Note that the **Print** statement in Figure 18.5 is prefaced by **Debug**. You can get away with the old style BASIC **Print** statement in the debug window. But within modules, VBA enforces a form of object-orientation. All stand-alone statements are replaced by methods of objects. In this case, the **Print** method window belongs to the **Debug** object.

### 18.3.4.2 Running the subroutine

Now that you have created a subroutine, you need to run it to see that it works. To invoke a subroutine, you simply use its name like you would any statement.

- ➔ Select **View** → **Debug Window** from the menu (or press **Ctrl-G** in version 7.0 and above).
- ➔ Type: **LoopingTest** in the debug window, as shown in Figure 18.5.

FIGURE 18.5: Run the *LoopingTest* subroutine in the debug window.



### 18.3.4.3 Conditional branching

We can use a different looping construct, **Do Until <condition>... Loop**, and the conditional branching construct,



If **<condition>** Then... Else, to achieve the same result.

- Type the following anywhere under the **End Sub** statement in order to create a new page in the module:

```
NL Sub BranchingTest
```

- Enter the following program:

```
NL Sub BranchingTest
NL   Dim i As Integer, s As String
NL   Dim blDone As Boolean
NL   s = "Loop number: "
NL   i = 1 'initialize counter
NL   blDone = False
NL   Do Until blDone
NL     If i > 10 Then
NL       Debug.Print "All done"
NL       blDone = True
NL     Else
NL       Debug.Print s & i
NL       i = i + 1
NL     End If
NL   Loop
NL End Sub
```

- Run the program

### 18.3.5 Using the debugger

ACCESS provides a very good debugger to help you step through your programs and understand

how they are executing. The two basic debugging constructs explored here are **breakpoints** and **stepping** (line-by-line execution).

- Move to the "Do Until blDone" line in the **BranchingTest** subroutine and select **Run → Toggle Breakpoint** from the menu (you can also press **F9** to toggle the breakpoint on a particular line of code).

Note that the line becomes highlighted, indicating the presence of an active breakpoint. When the program runs, the interpreter will suspend execution at this breakpoint and pass control of the program back to you.

- Run the subroutine from the debug window. Execution should halt at the breakpoint.
- Step through a couple of lines in the program line-by-line by pressing **F8**, as shown in [Figure 18.6](#).

By stepping through a program line by line, you can usually find any program bugs. In addition, you can use the debug window to examine the value of variables while the program's execution is suspended.

- Click on the debug window and type **? i** to see the current value of the variable **i**, as shown in [Figure 18.7](#).



FIGURE 18.6: Step through the each line in the program individually.

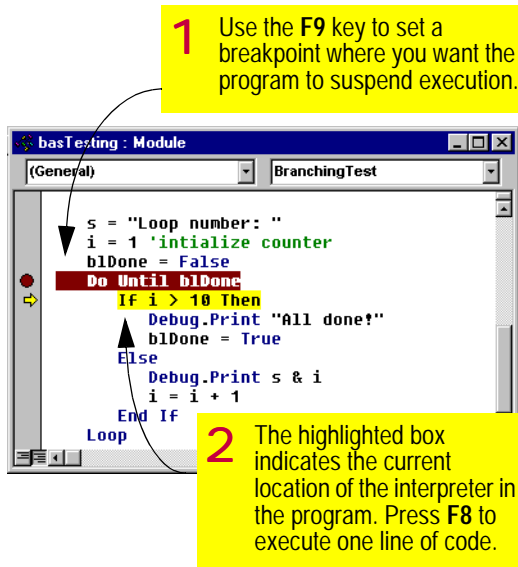
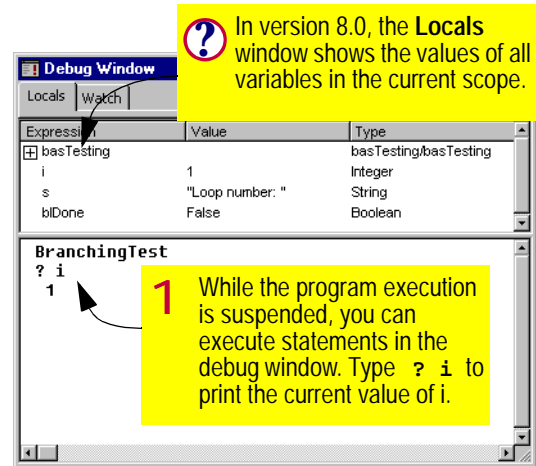


FIGURE 18.7: Use the debug window to print variable values while the program is running.



pass **parameters** (or **arguments**) to the subroutine.

The main difference between passed parameters and other variables in a procedure is that passed parameters are declared in the first line of the subroutine definition. For example, following subroutine declaration

```
NL Sub BranchingTest(intStart as Integer, intStop as Integer)
```

not only declares the variables `intStart` and `intStop` as integers, it also tells the subroutine

### 18.3.6 Passing parameters

In the `BranchingTest` subroutine, the loop starts at 1 and repeats until the counter `i` reaches 10. It may be preferable, however, to set the start and finish quantities when the subroutine is executed. To achieve this, you



to expect these two numbers to be passed as parameters.

To see how this works, create a new subroutine called `ParameterTest` based on `BranchingTest`.

- ➔ Type the declaration statement above to create the `ParameterTest` subroutine.
- ➔ Switch back to `BranchingTest` and highlight all the code except the `Sub` and `End Sub` statements
- ➔ Cut and paste the code into the `ParameterTest` procedure.

To incorporate the parameters into `ParameterTest`, you will have to make the following modifications to the pasted code:

- ➔ Replace `i = 1` with `i = intStart`.
- ➔ Replace `i > 10` with `i > intStop`.
- ➔ Call the subroutine from the debug window by typing:

```
NL ParameterTest 4, 12
```

The results are shown in [Figure 18.8](#).



If you prefer enclosing parameters in brackets, you have to use the

FIGURE 18.8: Create a parameterized subroutine.

The screenshot shows the Visual Basic IDE with the `ParameterTest` subroutine code in the `basTesting : Module` window. The code is as follows:

```
Sub ParameterTest(intStart As Integer, intStop As Int
Dim i As Integer, s As String
Dim bIDone As Boolean

s = "Loop number: "
i = intStart 'initialize counter
bIDone = False
Do Until bIDone
If i > intStop Then
Debug.Print "All done!"
bIDone = True
Else
Debug.Print s & i
i = i + 1
End If
Loop
End Sub
```

The `Debug Window` shows the output of the subroutine call:

```
ParameterTest 4,12
Loop number: 4
Loop number: 5
Loop number: 6
Loop number: 7
Loop number: 8
Loop number: 9
Loop number: 10
Loop number: 11
Loop number: 12
All done!
```

A yellow callout box with a red number '1' points to the parameter declarations in the code, containing the text: "Include the names of the parameters in the subroutine's declaration."

Call `<sub name>(parameter1, ..., parametern)` syntax, e.g.:

Call `ParameterTest(4, 12)`. See [Section 18.4.3](#) for more information on the use of brackets in VISUAL BASIC.



### 18.3.7 Creating a MinValue() function

In this section, you are going to create a user-defined function that returns the minimum of two numbers. Although most languages supply such a function, ACCESS does not (the `Min()` and `Max()` function in ACCESS are for use within SQL statements only).

➔ Create a new module called `basUtilities`.

➔ Type the following (on one line) to create a new function:

```
NL Function MinValue(n1 as Single, n2
as Single) as Single
```

The statement above defines a function called `MinValue` that returns a single-precision number. The function requires two single-precision numbers as parameters.

Since a function returns a value, the data type of the return value should be specified in the function declaration. Accordingly, the basic syntax of a function declaration is:

```
NL Function <function name>(parameter1
As <data type>, ..., parametern As
<data type>) As <data type>
```

The function returns a variable named `<function name>`.

➔ Type the following as the body of the function:

```
NL Function MinValue(n1 as Single, n2
as Single) as Single
NL If n1 <= n2 Then
NL MinValue = n1
NL Else
NL MinValue = n2
NL End If
NL End Function
```

➔ Test the function, as shown in [Figure 18.9](#).

## 18.4 Discussion

### 18.4.1 The evolution of BASIC

An important thing to keep in mind when using VBA is that the BASIC language has been around since the mid 1960s and has evolved in a relatively uncontrolled, organic matter. Because of this, VISUAL BASIC lacks the purity and simplicity of a teaching language like PASCAL or a newcomer like JAVA.

For new programmers, the result is often frustration. There are multiple conflicting ways to accomplish the same task and some of the language constructs (e.g., `Dim`, `Goto`) are only understandable in their historical context.

The latest transformation in the language's evolution is quasi-object-orientation. Stand-alone statements are being de-emphasized in favor of methods that belong to objects. For

FIGURE 18.9: Testing the `MinValue()` function.

**1** Implement the `MinValue()` function using conditional branching.

```

Function MinValue(n1 As Single, n2 As Single) As Single
    'function that returns the smaller of two numbers
    IF n1 <= n2 Then
        MinValue = n1
    Else
        MinValue = n2
    End IF
End Function

```

**2** Test the function by passing it various parameter values.

Expression	Value	Type
? MinValue(.12, 100)	0.12	
? MinValue(100, 101)	100	
? MinValue("cat", "dog")		

Microsoft Access  
Run-time error '13':  
Type mismatch

These five lines could be replaced with an "immediate if" function: `MinValue = iif(n1 <= n2, n1, n2)`.

According to the function declaration, `MinValue()` expects two single-precision numbers as parameters. Anything else generates an error.

example, recall the `Debug.Print` method in Section 18.3.4.

## 18.4.2 Interpreted and compiled languages

VBA is an **interpreted language**. In interpreted languages, each line of the program is interpreted (converted into machine language)

and executed when the program is run. Other languages (such as C, PASCAL, FORTRAN, etc.) are **compiled**, meaning that the original (source) program is translated and saved into a file of machine language commands. The resulting executable file is run instead of the source code.



Predictably, compiled languages run much faster than interpreted languages (e.g., compiled C++ is generally ten times faster than interpreted JAVA). However, interpreted languages are generally easier to debug since program execution can be stopped at any point and the current line of source code viewed and manipulated.

created and tested the function in [Section 18.3.7](#).

### 18.4.3 Brackets and parameters

Recall that subroutines in VISUAL BASIC execute a batch of commands, whereas functions execute a batch of commands and return a single value. The distinction is important because, when it comes to passing parameters to procedures, VISUAL BASIC adopts an odd convention:

- If the procedure returns a value (i.e., it is a function), enclose the parameters in brackets, e.g., **MinValue(2, 7)**.
- If the procedure does not return a value (i.e., it is a subroutine), do not enclose the parameters in brackets, e.g.:

**ParameterTest 4, 12.**

## 18.5 Application to the project

You will need a **MinValue()** function in [Section 19.5](#) when you have to determine the default quantity to ship. Ensure you have