# Access Tutorial 12: An Introduction to Visual Basic

## 12.1 Introduction: Learning the basics of programming

Programming can be an enormously complex and difficult activity. Or it can be quite straightforward. In either case, the basic programming concepts remain the same. This tutorial is an introduction to a handful of programming constructs that apply to any “third generation” language, not only Visual Basic for Applications (VBA).

⚠ 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, it is slightly enlarged subset called “Visual Basic for Applications” (VBA). However, in the context of the simple programs we are writing here, these terms are interchangeable.

### 12.1.1 Interacting with the interpreter

Access provides two ways of interacting with the VBA language. The most useful of these is through saved modules that contain VBA procedures. These procedures (subroutines and functions) can be run 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. Interpreted languages are easier to experiment with since you can invoke the interpreter at any time, type in a command, and watch it execute. In the first part of this tutorial, 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 a couple of VBA modules to explore looping, conditional branching, and parameter passing.

## 12.2 Learning objectives

- ❒ What is the debug/immediate window? How do I invoke it?
- ❒ What are statements, variables, the assignment operator, and predefined functions?
- ❒ How do I create a module containing VBA code?
- ❒ What are looping and conditional branching? What language constructs can I use to implement them?
- ❒ How do I use the debugger in Access?
- ❒ What is the difference between an interpreted and compiled programming language?

## 12.3 Tutorial exercises

### 12.3.1 Invoking the interpreter

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

This opens the module window which we will use in Section 12.3.3. 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 *Control-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. As such, you have to use *View > Immediate Window*. The term debug window will be used throughout this tutorial.

## 12.3.2 Basic programming constructs

In this section, we are going to use the debug window to explore some basic programming constructs.

### 12.3.2.1 Statements

Statements are 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:

  ```
  Print “Hello world!”↵
  ```

(the ↵ symbol at the end of a line means “press the *Return* or *Enter* key”).

? In VBA (as in all dialects of BASIC), the question mark (?) is typically used as shorthand for the `Print` statement. As such, the statement: `? “Hello world!”↵` is identical to the statement above.

### 12.3.2.2 Variables and assignment

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

- Type the following:

  ```
  s = “Hello”↵
  ? s & “ world”↵
  ? “s” & “ world”↵
  ```

In the first statement, a variable `s` is created and the string `Hello` is assigned to it. Recall the function of the concatenation operator (&) from Section 4.4.2.

? 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`?).

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**.

Within the debug window, any string of characters in quotations marks (e.g., “`COMM`”) is interpreted as a literal string. Any string without quotation marks (e.g., `COMM`) is interpreted as a variable (or a field name, if appropriate). Note, however, that this convention is not universally true within different parts of Access.

### 12.3.2.3 Predefined functions

In computer programming, a function is a small program that takes one or more **arguments** (or **parameters**) as input, does some processing, and returns a value as output. A *predefined* (or *built-in*) function is a function that is provided as part of the programming environment.

For example, `cos(x)` is a predefined function in many computer languages—it takes some number x as an argument, does some processing to find its cosine, and returns the answer. Note that since this function is predefined, you do not have to know anything about the algorithm used to find the cosine, you just have to know the following:

1. what to supply as inputs (e.g., a valid numeric expression representing an angle in radians),
2. what to expect as output (e.g., a real number between -1.0 and 1.0).

The on-line help system provides these two pieces of information (plus a usage example and some additional remarks) for all VBA predefined functions.

In this section, we are going to explore some basic predefined functions for working with numbers and text. The results of these exercises are shown in Figure 12.1.

- Print the cosine of 2π radians:

```
pi = 3.14159↵
? cos(2*pi)↵
```

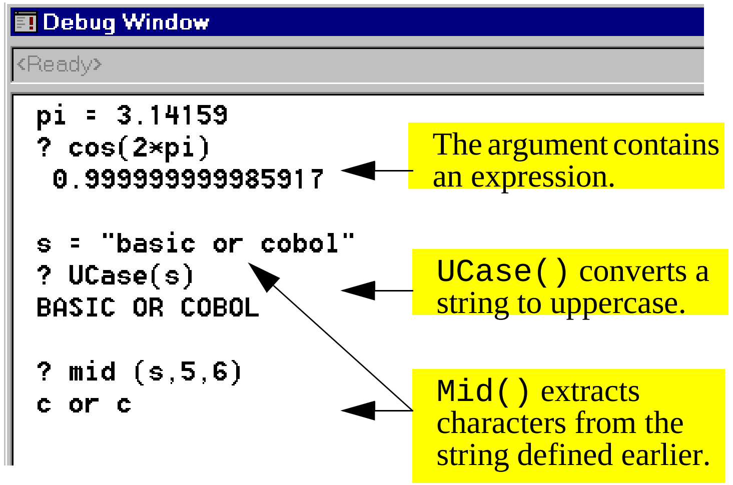
- Convert a string of characters to uppercase:

```
s = “basic or cobol”↵
? UCase(s)↵
```

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

```
? mid (s,5,6)↵
```

**FIGURE 12.1:** **Interacting with the Visual Basic interpreter.**



```
pi = 3.14159
? cos(2*pi)
 0.999999999985917

s = "basic or cobol"
? UCase(s)
BASIC OR COBOL

? mid (s,5,6)
c or c
```

The argument contains an expression.
UCase() converts a string to uppercase.
Mid() extracts characters from the string defined earlier.

### 12.3.2.4 Remark statements

When creating large programs, it is considered good programming practice to include adequate internal documentation—that is, to include comments to explain 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.:

```
  ‘ This is a comment line!
  Print “Hello” ‘the comment starts
   here
```

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

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

### 12.3.3 Creating a module

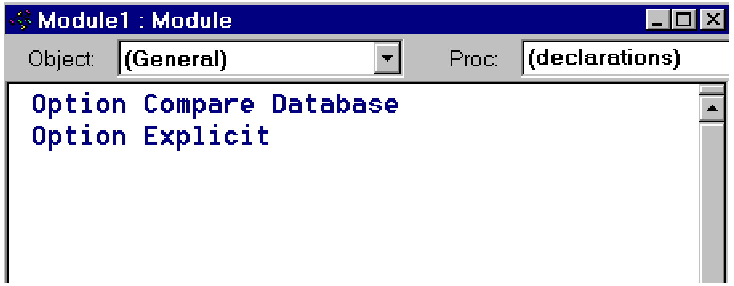
- Close the debug window so that the declaration page of the new module created in Section 12.3.3 is visible (see Figure 12.2).

The two lines:

```
Option Compare Database
Option Explicit
```

are included in the module by default. The `Option Compare` statement specifies the way in which strings are compared (e.g., does uppercase/ lowercase matter?). The `Option Explicit` statement forces you to declare all your variables before using them.

FIGURE 12.2: **The declarations page of a Visual Basic module.**



```
Option Compare Database
Option Explicit
```

In version 2.0, Access does not add the `Option Explicit` statement by default. As such you should add it yourself.

A module contains a declaration page and one or more pages containing **subroutines** or user-defined **functions**. The primary difference between subroutines and functions is that subroutines simply execute whereas functions are expected to return a value (e.g., `cos()`). Since 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.

The VBA editor in version 8.0 has a number of enhancements over earlier version, including the capability of showing multiple functions and subroutines on the same page.

### 12.3.4 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**.

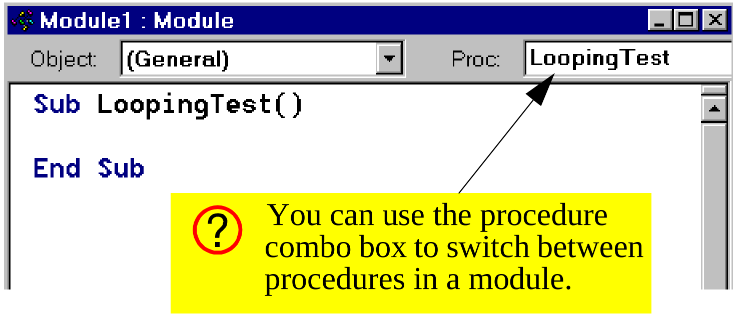
- Create a new subroutine by typing the following anywhere on the declarations page:
  `Sub LoopingTest()↵`

Notice that Access creates a new page in the module for the subroutine, as shown in Figure 12.3.

#### 12.3.4.1 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 is completely dependent on the type of data the variable is going to contain (e.g., string, integer, Boolean, double-precision floating-point, etc.), you

**FIGURE 12.3: Create a new subroutine.**



```
Sub LoopingTest()

End Sub
```

? You can use the procedure combo box to switch between procedures in a module.

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:

```
Dim i as integer
Dim s as string
```

- 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:

```
s = “Loop number: ”
For i = 1 To 10
   Debug.Print s & i
Next i
```

- 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 within the subroutine is prefaced by `Debug`. This is due to the object-oriented nature of VBA which will be explored in greater detail in Tutorial 14.

### 12.3.4.2 Running the subroutine

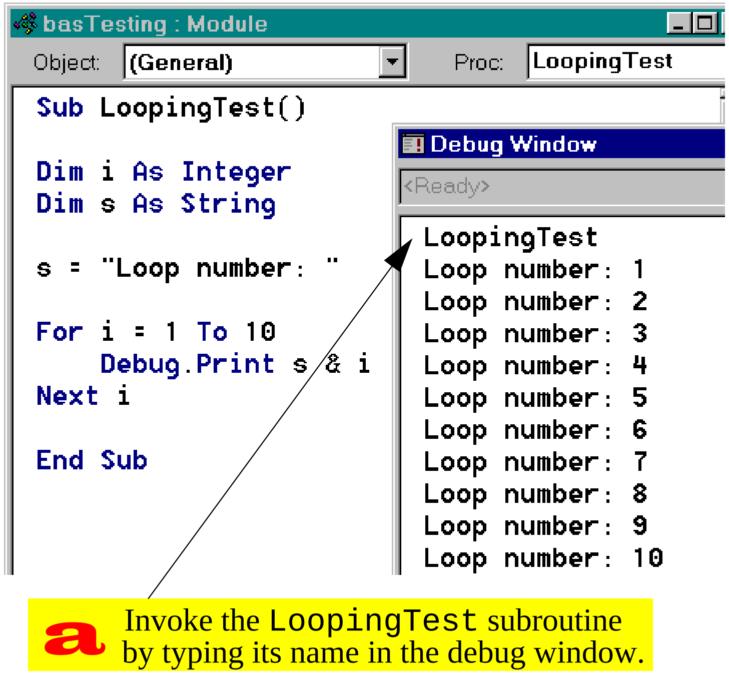
Now that you have created a subroutine, you need to run it to see that it works. To invoke a subroutine, you simply use its name like you would any statement.

- Select *View > Debug Window* from the menu (or press *Control-G* in version 7.0).
- Type: `LoopingTest↵` in the debug window, as shown in Figure 12.4.

### 12.3.4.3 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.

**FIGURE 12.4: Run the `LoopingTest` subroutine in the debug window.**



```
Sub LoopingTest()

Dim i As Integer
Dim s As String

s = "Loop number: "

For i = 1 To 10
    Debug.Print s & i
Next i

End Sub
```

```
LoopingTest
Loop number: 1
Loop number: 2
Loop number: 3
Loop number: 4
Loop number: 5
Loop number: 6
Loop number: 7
Loop number: 8
Loop number: 9
Loop number: 10
```

a Invoke the `LoopingTest` subroutine by typing its name in the debug window.

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

  ```
  Sub BranchingTest↵
  ```

- Enter the following program:

  ```
  Dim i As Integer
  Dim s As String
  Dim intDone As Integer
  s = “Loop number: “
  i = 1
  intDone = False
  Do Until intDone = True
     If i > 10 Then
        Debug.Print “All done”
        intDone = True
     Else
        Debug.Print s & i
        i = i + 1
     End If
  Loop
  ```

- Run the program

### 12.3.5 Using the debugger

Access provides a rudimentary debugger to help you step through your programs and understand how they are executing. The two basic elements of the debugger used here are **breakpoints** and **stepping** (line-by-line execution).

- Move to the `s = “Loop number: ”` line in your `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, it will suspend execution at this breakpoint and pass control of the program back to you.

- Run the subroutine from the debug window, as shown in Figure 12.5.
- Step through a couple of lines in the program line-by-line by pressing *F8*.

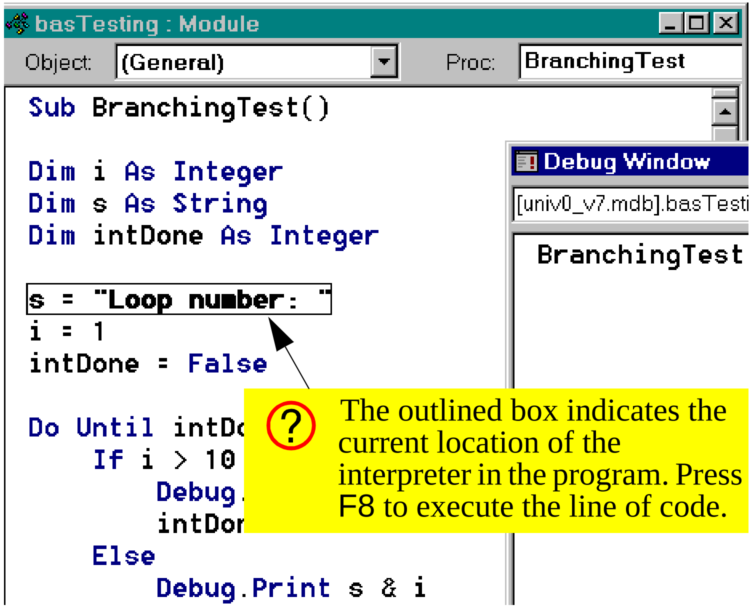
By stepping through a program line by line, you can usually find any program bugs. In addition, you can use the debug window to examine the value of variables while the program's execution is suspended.

- click on the debug window and type
  `? i↵`
  to see the current value of the variable `i`.

## 12.3.6 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 called from the debug window. To achieve this, we have to pass **parameters** (or **arguments**) to the subroutine.

FIGURE 12.5: **Execution of the subroutine is suspended at the breakpoint.**



```
Sub BranchingTest()

Dim i As Integer
Dim s As String
Dim intDone As Integer

s = "Loop number: "
i = 1
intDone = False

Do Until intDo
    If i > 10
        Debug.
        intDor
    Else
        Debug.Print s & i
```

```
BranchingTest
```

The outlined box indicates the current location of the interpreter in the program. Press F8 to execute the line of code.

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

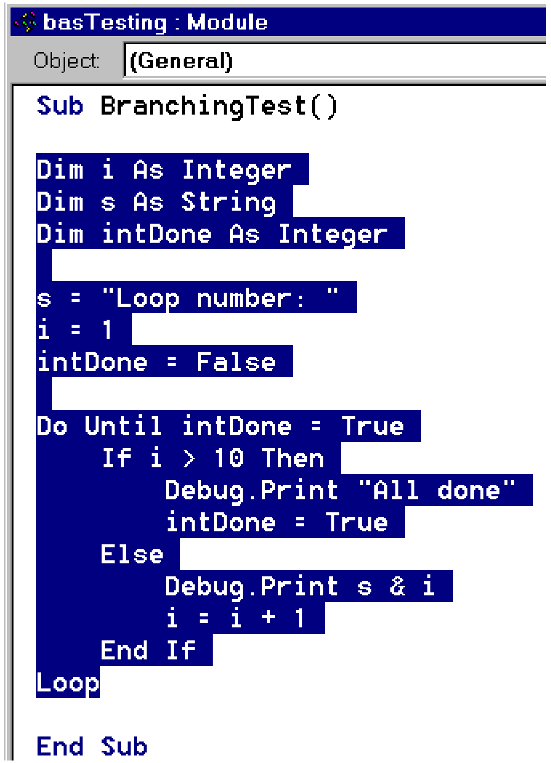
```
Sub BranchingTest(intStart as
  Integer, intStop as Integer)
```

not only declares the variables `intStart` and `intStop` as integers, it also tells the subroutine 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, as shown in Figure 12.6.

**FIGURE 12.6: Highlight the code to copy it.**



basTesting : Module

Object: (General)

```
Sub BranchingTest()

Dim i As Integer
Dim s As String
Dim intDone As Integer

s = "Loop number: "
i = 1
intDone = False

Do Until intDone = True
    If i > 10 Then
        Debug.Print "All done"
        intDone = True
    Else
        Debug.Print s & i
        i = i + 1
    End If
Loop

End Sub
```

- Copy the highlighted code to the clipboard (*Control-Insert*), switch to `ParameterTest`, and paste the code (*Shift-Insert*) 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:
  `ParameterTest 4, 12↵`

? If you prefer enclosing parameters in brackets, you have to use the `Call <sub name>(parameter`$_1$`, ..., parameter`$_n$`)` syntax. For example:
`Call ParameterTest(4,12)↵`

### 12.3.7 Creating the `Min()` 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 to create a new function:
  ```
  Function MinValue(n1 as Single, n2
  as Single) as Single↵
  ```

This 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. As such, the basic syntax of a function declaration is:

```
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:

```
If n1 <= n2 Then
   MinValue = n1
Else
   MinValue = n2
End If
```

- Test the function, as shown in Figure 12.7.

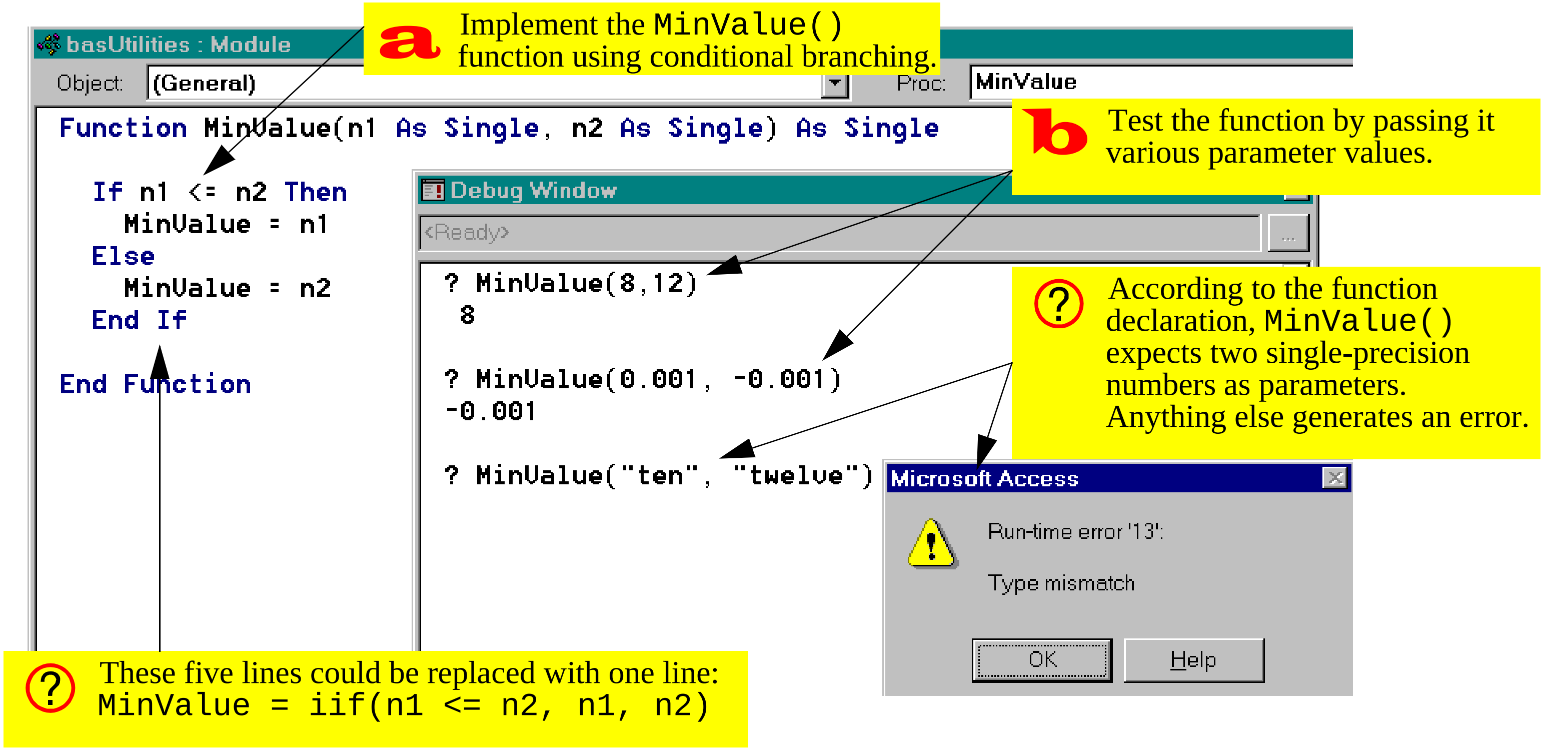
## 12.4 Discussion

### 12.4.1 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. This executable file is run instead of the source code.

Predictably, compiled languages run much faster then interpreted languages (e.g., compiled C++ is generally ten times faster than interpreted Java). However, interpreted languages are typically easier to learn and experiment with.

**FIGURE 12.7: Testing the `MinValue()` function.**



a Implement the `MinValue()` function using conditional branching.

```
Function MinValue(n1 As Single, n2 As Single) As Single

   If n1 <= n2 Then
      MinValue = n1
   Else
      MinValue = n2
   End If

End Function
```

b Test the function by passing it various parameter values.

```
? MinValue(8,12)
 8

? MinValue(0.001, -0.001)
-0.001

? MinValue("ten", "twelve")
```

Run-time error '13':

Type mismatch

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

? These five lines could be replaced with one line:
`MinValue = iif(n1 <= n2, n1, n2)`

## 12.5 Application to the assignment

You will need a `MinValue()` function later in the assignment when you have to determine the quantity to ship.

- Create a `basUtilities` module in your assignment database and implement a `MinValue()` function.

To ensure that no confusion arises between your user-defined function and the built-in SQL `Min()` function, do not call you function `Min()`.