Access VBA Made Easy

Access VBA Fundamentals

Level 4

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

07 - Conditionals and Branching

Introduction

In everyday life we create scenarios for ourselves and base our actions upon them. An example would be someone saying "If it rains tomorrow then we will stay in; otherwise we will go to the park".

This type of statement is known as a conditional (in both human and computer language). The idea is that we have a statement that can be evaluated to true or false and then act based on that evaluation.

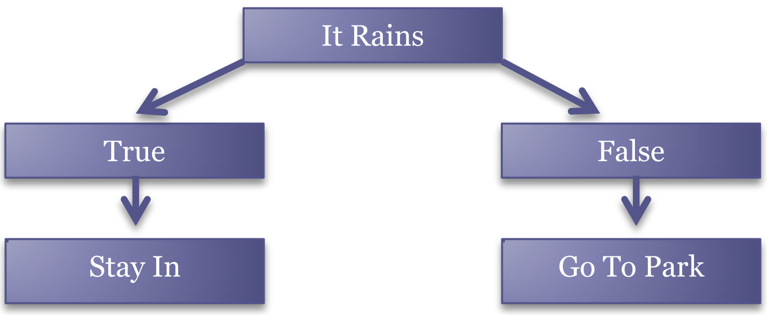

Figure 7.1

Figure 7.1 shows that we are evaluating what will happen if it rains or not. The concept of raining is either true (it is raining) or false (it is not raining) and depending on the answer we either stay in or go out.

Programming languages work in much the same way. A statement is evaluated to be either true or false and the code is executed depending on the answer.

All conditionals use an operator in an expression which concludes that the expression is either True or False. We will start with the straightforward If statement and "=" operator.

If…Then, Evaluating Expressions, Operators, Statement Blocks

Diving straight into some examples, you can execute the following in the immediate window of the VBA Editor. Once you've executed it we'll discuss the code:

```
a = 10 : If a=10 Then Debug.Print “a=10”
```

Figure 7.2

The <u>If statement</u> is a very simple statement that asks a straightforward question – *is an <u>expression</u> True or False?* If the <u>expression</u> is True then execute some code – in this case the "a=10" is printed in the immediate window. You can check this by changing the value of "a" to any other number and re-execute.

The <u>expression</u> above uses what is termed an <u>operator</u>, the "="equals operator. To clearly show what the *expression part* of an If statement is, the above has been rewritten with braces around the expression in the box below.

```
a = 10 : If (a=10) Then Debug.Print “a=10”
```

Figure 7.3

Here are some other examples. All of them evaluate to True.

```
a=20 : If a=20 Then Debug.Print “a=20”
c=5  : If c=5  Then Debug.Print “c=5”
d:10 : If d=10 Then Debug.Print “d=10”
```

Figure 7.4

"AND" Operator

We can also use the keyword <u>AND</u> which asks if two expressions are both equal to True. All the statements below evaluate to True.

```
1   a=10 : b=10 : If (a=10   And b=10)  Then Debug.Print "a and b = 10"
2   c=5  : a=5  : If (c=5)   And (a=5)  Then Debug.Print "c=5 and a=5"
3   a=12 : b=12 : If ((a=12) And (b=a)) Then Debug.Print "a and b = 10"
```

Figure 7.5

In Figure 7.5 you can see that an expression doesn't have to include actual values – numbers like 10 and 5 – but can consist of comparing variable against variable. Line 3 demonstrates this; b is never asked if it equals 12, but asked if it equals a (which does equal 12).

Nested Expressions

Line 3 also shows that the expression ((a=10) And (b=a)) is what is termed a <u>nested expression</u>; that is, there are expressions inside expressions.

1. (a=10)
2. (b=a)
3. ( ) And ( ) which is written as ((a=10) And (b=a))

Nested expressions are more common than non-nested expressions and as programmers you will be using them everywhere. For this reason we will use nested expressions wherever possible in order to get accustomed to dealing with them. Here are some examples of nested expressions (the result of which are all False); so they do not execute the Debug statement.

```
a=11 : b=10 : If (a=10   And b=10)  Then Debug.Print "a and b = 10"
c=5  : a=4  : If (c=5)   And (a=5)  Then Debug.Print "c=5 and a=5"
a=13 : b=12 : If ((a=12) And (b=a)) Then Debug.Print "a and b = 10"
```

Figure 7.6

Exercise
Change the above expressions so that the code after "Then" is executed

If…Then…Else…End If

In Figure 7.6 none of the Debug.Print statements are executed as all of the expressions evaluate to False. So, what do we do if we want to execute some code when the expression evaluates to false?

VBA extends the If…Then statement to include an Else part to tackle such situations. The Else part is executed then the expression evaluates to False. Here are some examples (put the examples in a new Module and call the procedure from the Immediate window:

```
1   Sub conditionalsProc1()
2       myName = "Steve"
3     If myName = "Steve" Then
4       Debug.Print "His name is Shaun"
5     Else
6       Debug.Print "His name is not equal to Shaun"
7     End If
8
9     myBalance = 6000.233: myOD = 0#
10    If (myBalance = 6000) And (myOD = 0#) Then
11      Debug.Print "Balance is £6000.00, OD is £0.00"
12    Else
13      Debug.Print "Either my balance is not £6,000.00 or my OD is not £0.00"
14    End If
15
16    dateEnd = #12/21/2012#
17    daysPostNothing = 2
18    If ((dateEnd = #12/21/2012#) And (daysPostNothing = 2)) Then
19      MsgBox "The date is upon us"
20    Else
21      Debug.Print "Either dateEnd or daysPostNothing or both don't equal expected values"
22    End If
23  End Sub
```

Figure 7.7

Note
You will notice If…Then…Else have been spread across five lines. This format is a standard used in practically every programming language to help us read code more easily.

Now with the Else part in place, all the above If statements will do *something*. For example, line 10 asks a question of myBalance and myOD; if this is true line 11 will execute, if not line

13 will execute. Line 3 compares a string variable with a string literal. Line 18 compares a date variable to a date value and an integer variable to an integer value.

When using the Else part we must also end the whole If statement with the words End If. If this is not done the compile won't execute the code.

Now that we have introduced End If we can also bring in statement blocks.

If…Then…End If and Statement Blocks

Essentially all your code is divided into statement blocks. Like everything in your Sub or Functions, it is simply a way for us human's to say *here is a list of code I want executed.* A statement block is code between some start keyword and some end keyword, e.g. Sub…End Sub, If…End If, Property Get…End Property, While…End While, For…Next.

```
' Example of statement blocks
Sub myStatementBlock1
  ' Here we put our statement blocks
End Sub

Function myStatementBlock2
  ' Here we put our statement blocks
End Function

Function myStatementBlock3
  ' Statement Block 1
  If (expression) Then
    ' Statement Block 2
    ' Statement blocks are indented by spaces or tab to aid _
      understanding
  Else
    ' Statement Block 3
  End If
  ' Statement Block 1 continues
End Function
```

Figure 7.8

With an If…Then statement the statement block must be only one statement in length – zero or many statements are forbidden.

```
Sub myExampleSub2
  If a=b and c=a Then Debug.Print "executed when evaluates to true"
End Sub
```

Figure 7.9

```
Sub myExampleSub3
  ' subroutine's statement block
  If a=b and c=a Then
    ' If statement block code for True
    Debug.Print "executed when evaluates to true"
    Debug.Print "and so is this"
  End If
```

```

   ‘ Back to subroutine’s code block
   Debug.Print “but this is outside the statement block”

   ‘ it is possible to separate statements using : But it makes for  _
     really difficult reading,
 If a = b And c = a Then
   Debug.Print "executed when evaluates to true": Debug.Print "and so is this"
   End If
```

Figure 7.10

When If is closed off with an End If then all the lines between them are a statement block. This block may contain zero, one or many statement lines or even nested statements; so a statement block may contain yet another If statement within its own blocks of code.

If…Then…Else… End If

An If…Then…Else…End If statement has at least two statement blocks!

```
Sub myExampleSub4
  If a=b and c=a Then
    ‘ This is statement block 1
    Debug.Print “executed when evaluates to true”
    Debug.Print “and so is this”
  Else
    ‘ This is statement block 2
    Debug.Print “this code is in the Else part”
    Debug.Print “Too much cake is not a good thing”
  End If
End Sub
```

Figure 7.11

```
Sub myExampleSub5()
  If Day(Now()) = 1 Then
    ' This is statement block 1
    Debug.Print "Its the first day of the " + CStr(Month(Now())) + " month"
  Else
    ' This is statement block 2
    Debug.Print "It's day number " + CStr(Day(Now())) + " of the month"
  End If
End Sub
```

Figure 7.12

This example works on the expression:

```
Expression: Day(Now()) = 1 or (Day(Now()) = 1)

Now() - function returns the current system date and time
Day() - function accepts a date value and gives us the day in the month
   =1 - does Day(Now()) = 1?
        If so then evaluate to TRUE, otherwise FALSE
And that True or False value determines which block of code ix
executes: block 1 if True, and block 2 if False.
```

Figure 7.13

"=" is not the only operator in VBA, there are several arithmetic operators, logical operators and statements which can be evaluated as an expression.

If...Then...ElseIf...[ElseIf...] End If

The final extension of the If statement is the ElseIf. ElseIf is useful in that it makes nested If statements much easier to read, to code and to handle.

The concept of the ElseIf statement is that each condition will be evaluated in order until one of them evaluates to true at which point the code for that condition (and only that condition) will be executed.

```
Sub myExampleSub6(ageOfChild As Integer)
If ageOfChild < 6 Then
    Debug.Print "Your child is in year 1"
ElseIf ageOfChild < 7 Then
    Debug.Print "Your child is in year 2"
ElseIf ageOfChild < 8 Then
    Debug.Print "Your child is in year 3"
ElseIf ageOfChild < 9 Then
    Debug.Print "Your child is in year 4"
ElseIf ageOfChild < 10 Then
    Debug.Print "Your child is in year 5"
ElseIf ageOfChild < 11 Then
    Debug.Print "Your child is in year 6"
ElseIf ageOfChild < 12 Then
    Debug.Print "Your child is in year 7"
Else
    Debug.Print "Your child is in year 8"
End If
End Sub
```

Figure 7.14

In the above example we can run the code by writing *myExampleSub 8* in the immediate window where 6 is the argument we are passing (*ageOfChild as integer*). The first condition that will be evaluated is whether *ageOfChild* is less than 6 which in this case will evaluate to false. The next condition will be if *ageOfChild* is less than 7 which, again, will evaluate to false. The first condition that will evaluate to true will be *ageOfChild<9* and so the code *Debug.Print "Your child is in year 4"* will execute. The code will then leave the if statement.

The key point here is that the conditions will be evaluated until the first condition that is found to be true. The code will then be executed and then leave the if statement.

Note
The alternative to the ElseIf statement is the Select…Case statement later in this unit.

Expressions: Operators

As mentioned above an expression is a single or list of variables and operators that ultimately evaluates to True or False. Here we will list all the arithmetic operators with example code to introduce the host of operators you'll need when programming.

Boolean as an Expression

As expressions must evaluate to a Boolean value (true or false) one can just use a Boolean value or variable rather than an operator.

```
Sub myExampleSub7(Optional semaphore As Boolean = True)
  If semaphore Then
    Debug.Print "The Semaphore is True"
  Else
    Debug.Print "The Semaphore is False"
  End If
End Sub
```

Figure 7.15

Here there are no operators being used, the variable semaphore *is a Boolean* value so *is an expression* all by itself. This is a very useful construct and is the basis of all loop statements which we'll cover in another unit; but to give you an idea …

```
1   Sub cheekyLoopRoutine()
2     Dim EOF as Boolean, a as String : EOF = false
3     While EOF ' Begin While statement block
4       a = getInputFromFile()
5       If a = "" Then EOF = True
6       Debug.Print a
7     Wend ' End While statement Block
8   End Sub
9
```

Figure 7.16

The cheekyLoopRoutine() routine executes the While…Wend statement block over and over again until "a" is given a zero-length String from the function getInputFromFile(). At that time EOF (meaning End-of-File) will be set to True and the loop will break out of the block to line 8.

Arithmetic Operators

Arithmetic operators work by comparing expression A with expression B. We say comparing expressions because A and B may be nested expressions that must be evaluated first to yield an answer to the If statement, or be values themselves.

A=B	Equal To	Tests for value equality
A>B	Greater Than	Evaluates to True when A is Greater Than B
A>=B	Greater Than or Equal To	Evaluates to True when A is at least the value of B
A<B	Less Than	Evaluates to True when A is Less Than B
A<=B	Less Than or Equal To	Evaluates to True when A is at most B
A<>B	Great than Or Less than or, Doesn't Equal	Evaluates to True when A doesn't equal B

Figure 7.17

Examples of uses for these operators are in myExampleSub8 below.

```
Sub myExampleSub8()
  Dim A As Integer, B As Integer

  ' Test Greater Than
  A = 20: B = 21
  If A > B Then
    Debug.Print "A is Greater than B"
  Else
    Debug.Print "B is Greater than A"
  End If

  ' Test Less Than
  A = 20: B = 19
  If A < B Then
    Debug.Print "A is Less Than B"
  Else
    Debug.Print "B is Less Than A"
  End If

  ' Test Not Equal To
  A = 20: B = 50
  If A <> B Then
    Debug.Print "A and B are Not Equal."
  Else
    Debug.Print "A and B are Equal"
  End If

End Sub
```

Figure 7.18

Exercise
Use the above procedure and change the values of A and B so that the other part of each If statemens is executed.

Using the operators above it can be demonstrated that nested expressions are also values.

```
1   Sub myExample9()
2
3     Dim A As Integer, B As Integer, C As Integer, D As Boolean
4     A = 12: B = 48: C = 24
5     If (C / A) = 2 Then
6       Debug.Print "A multipled by 2 = C"
7     Else
8       Debug.Print "A multipled by 2 <> C"
9     End If
10
11    A = 24: B = 24: D = True
12    If (A >= B) = D Then
13      Debug.Print "A multipled by 2 = C"
14    Else
15      Debug.Print "A multipled by 2 <> C"
16    End If
17
18  End Sub
```

```
There are two expressions in line 5: (C / A) = 2

  (C / A) is the first expression, which equates to 2
  2 = 2   is the second expression which equates to True

On line 12 there are also two expressions: (A >= B) = D

  (A >= A) is the first expression, which equates to True
  (True=D) is the second expression, which equates to True

Ultimately the expression comes down to a True or False value.
```

Figure 7.19

Arithmetic Operators on Strings

It is possible to perform the same operators to Strings as one would numbers.

```
Sub myExample10()

  Dim E As String
  E = "Farming"
  If (E = "farming") Then
    Debug.Print "E equals ";E
  Else
    Debug.Print "E does not equal ";E
  End If

  Dim A As String, B As String
  A = "1": B = "02"
  If (A > B) Then
    Debug.Print "A is higher than B"
  Else
```

```
16      Debug.Print "B is higher than A"
17    End If
18
19    C = "a" : B = "1"
20    If (C >= D) Then
21      Debug.Print C; " is equal to or greater than "; B
22    Else
23      Debug.Print C; " is less than "; B
24    End If
25
26  End Sub

    Line 5 asks the question "does String E equal another String?".  As
    they are both the same content (although different case) the expected
    answer is given, True.

    On line 13: (A > B)

      A = "1", B = "02" is A > B?

    If we performed this expression on Integer variables we would expect
    the answer to be "B is higher than A", but no, "A is greater than B"?

    Line 20 also yields a bizarre answer, that "a" is equal to or greater
    than "1"?
```

Figure 7.20

The reason for the seemingly odd behaviour is down to how Strings are evaluated in expressions, and is actually quite logical.

A String is basically a list of characters, nothing more. When comparing two Strings VBA checks each String character-for-character for equality or value. In the case of line 13 above, a "1" is checked against a "0" and thus A is greater than B.

Logical Operators

Logical operations work with Boolean expressions to yield an answer for expressions. Individually they are quite straightforward but can be brought together.

And Operator

The And operator requires 2 Boolean values, gives a True answer when both sides of the argument are also True, otherwise False. A logic table demonstrates this more clearly.

For the expression:

`Z And X`

	X	
Z	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

```
Sub myExample11()

  Dim X As Boolean, Z As Boolean

  X = True  : Z = True  : Debug.Print X and Z ' True
  X = True  : Z = False : Debug.Print X and Z ' False
  X = False : Z = True  : Debug.Print X and Z ' False
  X = False : Z = False : Debug.Print X and Z ' False

End Sub
```

Figure 7.21

Or Operator

The Or operator requires 2 Boolean values, gives a value of True when either side of the argument is True. A logic table demonstrates this more clearly.

For the expression:

```
Z Or X
```

	X	
Z	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

```
Sub myExample12()

  Dim X As Boolean, Z As Boolean

  X = True  : Z = True  : Debug.Print X or Z ' True
  X = True  : Z = False : Debug.Print X or Z ' True
  X = False : Z = True  : Debug.Print X or Z ' True
  X = False : Z = False : Debug.Print X or Z ' False

End Sub
```

Figure 7.22

Not Operator

The Not operator requires 1 Boolean value, gives a value of True when the argument is False, and False when the argument is True. A logic table demonstrates this more clearly.

For the expression:

```
Not X
```

X	
TRUE	FALSE
FALSE	FALSE

```
Sub myExample13()

  Dim X As Boolean

  X = True  : Debug.Print Not X
  X = False : Debug.Print Not X

End Sub
```

Figure 7.23

Nested If Clauses

“If” statements can also be nested by putting “If” statements inside the execution block of an outer “If” statement.

```
Sub myExample14()
  Dim X As Boolean, Y As Boolean
  X = True : Y = 1

  If X Then ‘ Outer If Start
    Debug.Print ”X is True”

    If Y <2 then ‘ start of nested If statement
      Debug.Print “Y is < 2”
    Else
      Debug.Print “Y is >= 2”
    End If        ‘ end of nested If statement

  Else
    Debug.Print ”X is False”
  End If ‘ Outer If Ended
End Sub
```

Figure 7.24

Select...Case...Else

All languages have alternatives to explaining the same thing; VBA is no exception. In the area of conditionals Select...Case...Else is an alternative to If...Then...ElseIf...End If.

```
Enum StatusCode ' Status of myStoreStatus variable
  CLOSED
  OPENING
  RESTOCKING
  OUT_OF_OPERATION
End Enum

Function myExample15(storeStatus As StatusCode)

  Select Case storeStatus
    Case Is = StatusCode.CLOSED
      Debug.Print "Store CLOSED"

    Case Is = StatusCode.OPENING
      Debug.Print "Store OPENING"

    Case Is = StatusCode.OUT_OF_OPERATION
      Debug.Print "Store OUT_OF_OPERATION"

    Case Is = StatusCode.RESTOCKING
      Debug.Print "Store RESTOCKING"

    Case Else
      Debug.Print "Unknown store code:" + CStr(storeStatus)
  End Select
End Function
```

Figure 7.25

Compared to the ElseIf statement Select…Case is a bit bigger in structure, but the ease of adding new code and its regular structure is appealing in certain situations. In terms of execution speed Select…Case carries the same cost as ElseIf.

Common Problems

Too Many conditionals

A common problem is extending a set of conditionals and making a collection of statements really difficult to read.

Note
Where necessary don't be afraid to alter the structure of your code to improve its readability. Readability is far more important in VBA than execution time, line count or conciseness. Something that is easy to read will naturally contain fewer errors.

Too Many Expressions

It is possible to include too many expressions and operators in a statement and get very confused about which takes precedence over another. Where possible, place brackets around your expressions to make them easier to read.

Very Long Select… Case Statements

Select Case is quite a verbose syntax to use because each Case line is accompanied by a Code Block. To reduce the length of the statement use a function or sub procedure in the code block as this will markedly improve code readability and reliability.

Questions

1. Correct the following code

```
Dim myName as String
myName = getUsername()  ‘ returns user’s name

If myName = “Mat” Then MsgBox “Hi Mat”
If myName = “John Then MsgBox “Hi John”
If myName = “Sarah” Then Print “Hi Sarah”

Dim l as Integer
l = len myName
If l > 4 Amd l < 10 Then
  Debug.Print “Length of myName is  + CStr(l)
Nend If
```

2. What must an expression evaluate to?
    a. Class or Object
    b. True or False
    c. Null or Nothing
    d. Empty or Full
    e. T or F

3. Which of the following are expressions
    a. ((a+b)=c)
    b. (a) < (b-c)
    c. a)b-1
    d. a And b
    e. a Tan b

4. If A is True and C is True and B is False (True or False)
    a. Not A = False
    b. A = C
    c. Not A = B
    d. Not B = A
    e. D = A Or B : D = True
    f. A = C = Not B

5. Why is indentation a good thing?

6. How many statements can an If Statement without an End If have? How many must it have?

7. What are Nested If statements?
    a. A group of statements inside an If Statement
    b. A resting place for birds and bugs
    c. A conditional statement buried in an execution block in an Else clause.
    d. End of an If statement

8. What is wrong with the following ElseIf?

```
Enum StatusCode ' Status of myStoreStatus variable
  CLOSED
  OPENING
  RESTOCKING
  OUT_OF_OPERATION
End Enum

If myStoreStatus = CLOSED Then ' executed on close

ElseIf myStoreStatus = OPENING Then ' executed on OPENING

ElseIf myStoreStatus = CLOSED Then ' executed on Restocking

End If
```

9. Link up the Operator with the Description

1	A=B
2	A>B
3	A>=B
4	A<B
5	A<=B
6	A<>B

	A	Greater Than
	B	Equal To
	C	Less Than or Equal To
	D	Great than Or Less than or, Doesn't Equal
	E	Greater Than or Equal To
	F	Less Than

10. Examine the following code:

```
A = ?
B = ?

If A < B Then

  Debug.Print "Rome"

ElseIf A > B Then

  Debug.Print "Paris"

Else

  Debug.Print "London"

End If
```

a) Set the value of A and B so that London is displayed

b) Set the value of A and B so that Paris is displayed

c) Using A from (b) change B so that Rome is displayed

11. Two variables A and B. Both display 1.1 when Debug prints out their value, yet they are of different data types. What types might they be?

12. Assign the following Logical Operators to the logic diagram below: And, Or, Not

    a. For an extra point, what Logical Operator might (D) be?

(A)	X	
Z	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

(B)	X	
Z	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

(C)	X	
	TRUE	FALSE
	FALSE	FALSE

(D)	X	
Z	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	TRUE

13. Using the above table answer the following True or False questions – substitute () for their logical operator above
    a. True (A) False
    b. True (B) True
    c. (C) True
    d. True (A) ((C) False (B) True)
    e. False (D) False
    f. (C) True (A) (C) True

14. A statement block within a statement block. Explain.

15. Write the following in nicely indented code:
    If a = b And c = a Then: MsgBox "Might be true": Debug.Print "and so may this": Else: Debug.Print "It's Twins!" : End If

16. Rewrite the following as a select statement:

```
Enum Status
  INCREASE_TEMP
  DECREASE_TEMP
  WARM_UP
  COOL_DOWN
  FAN_ON
  FAN_OFF
End Enum

Function P(airconStatus As Status) As Long

```

```
  If airconStatus = INCREASE_TEMP Then
    s = 1
  ElseIf airconStatus = DECREASE_TEMP Then
    s = 2
  ElseIf airconStatus = FAN_ON Then
    s = 4
  ElseIf airconStatus = FAN_OFF Then
    s = 8
  ElseIf airconStatus = WARM_UP Then
    s = 16
  ElseIf airconStatus = COOL_DOWN Then
   s = 32
  Else
   s = 64
  End If
  P = s
End Function
```

17. What is the default value of semaphore?

```
Sub myExampleSub7(Optional semaphore As Boolean = True)
  If semaphore Then
    Debug.Print "The Semaphore is True"
  Else
    Debug.Print "The Semaphore is False"
  End If
End Sub
```

a) myExampleSub7(Not True). What is the outcome of myExampleSub7?
b) myExampleSub7(Not False And Not False). What is the outcome?
c) myExampleSub7(Not False And Not True). What is the outcome?
   a. For an extra point, what's the outcome?
      myExampleSub7(Not False XOR Not Not True)

18. What are the values of a, b and c ?

```
Sub J()
  Dim SMS_a As String, SMS_b As String

  SMS_a = "On the way home!" ' trick question
  SMS_b = "0n the way home!"

  Debug.Print "a="; SMS_a < SMS_b
  Debug.Print "b="; SMS_a = SMS_b
  Debug.Print "c="; SMS_a > SMS_b

End Sub
```

19. How can a With block make code easier to read?

20. Why does this equal False?

```
print CInt(20001.1) = "20001.1"
```

08 - Arrays and Collections

Computing is all about sets of similar looking data; appointments, files, pictures, addresses, UDP packets, tracks, database records, patient records, library records, lots of records. These different data structures inside our programs, computers, hard-drives and memory will be stored as repeating rows making up arrays and collections. Mostly, computing is about processing these arrays and collections of data, and those arrays and collections is what this unit is all about.

This unit will first introduce Arrays as the traditional data structure and also in VBA's somewhat extended variant. This will lay the foundation for understanding Collections and appreciating the differences between the two structures and be able to choose which best suits your particular task.

Traditionally, an Array has always been a block of memory put aside to hold values of a particular type. Its size is set at the time it is initiated and any element within it may be accessed randomly or sequentially. The best way to envisage an Array is like a table of data that is held in memory.

A Collection is an object that holds references to other objects of a similar type. It is somewhat similar to an array, in that it holds a list of things, but a collection is normally dynamic in size and, over all, easier to use than an Array. Objects in a collection can also be randomly or sequentially accessed.

The two, though somewhat similar, are fundamentally completely different. They both hold data, both allow access to the data, both fulfil almost identical roles, except that Arrays handle Primitive Types and Collections handle Primitive Types and Objects.

Declaring Arrays

One can think of an array as a row of boxes with a number on each, 0 to *n*. When we first declare an array we must state at least its type and may also state its size, but we can also set the size later.

Firstly, we will create an array that will hold Integer types.

```
Dim myIntegerArray() as Integer
```

myIntegerArray : Array of Integers

Figure 8.1

At this point VBA is aware that myIntegerArray will be an array containing Integers but it doesn't know how large we want it.

So we will use the ReDim statement to set the size of the array which reserves memory for it. We will make a 10 integer array. *Each Integer takes up 4 bytes.*

```
Dim myIntegerArray() as Integer
ReDim myIntegerArray(10)
```

myIntegerArray: Array of Integers (0..9)

0	1	2	3	4	5	6	7	8	9

0	0	0	0	0	0	0	0	0	0

Figure 8.2

Unlike C and some other languages VBA clears the array elements so guaranteeing the developer a clean slate to work with.

All other primitive typed arrays are allocated the same way. Strings:

```
Dim myStringArray() as String
ReDim myStringArray(10)
```

myStringArray: Array of Strings (0..9)									
0	1	2	3	4	5	6	7	8	9
“”	“”	“”	“”	“”	“”	“”	“”	“”	“”

Figure 8.3

VBA initialises Strings to “”, an empty String. Each character of a string takes up at least 2 bytes.

```
Dim myFloatArray() as Float
ReDim myFloatArray(10)
```

myFloatArray: Array of Floats (0..9)									
0	1	2	3	4	5	6	7	8	9
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 8.4

VBA initialises Floats to 0.0. A float takes up 8 bytes.

```
 Dim myDateArray() as Date
 ReDim myDateArray (10)
```

myDateArray: Array of Dates (0..9)									
0	1	2	3	4	5	6	7	8	9
00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

Figure 8.5

VBA initialises Dates to 00:00:00. A date takes up 8 bytes.

```
Dim myBooleanArray() as Boolean
ReDim myBooleanArray (10)
```

myBooleanArray: Array of Boolean (0..9)									
0	1	2	3	4	5	6	7	8	9
False	False	False	False	False	False	False	False	False	False

Figure 8.6

VBA initialises Boolean values to False. A Boolean takes up 1 byte.

Hang on; what is all the fuss about??

Arrays take up REAL physical resources, they require REAL physical memory space and REAL physical CPU power to operate and move them around. They and Collections are the main data structure that takes up system resources. The bigger these structures are the harder computers must work to maintain them.

This is particular a concern for old computers and mobile devices; these have very limited amounts of memory and limited CPU speeds and capability. So an awareness of arrays, their physical characteristics and their impact on your program,your computer resources and performance is vital for any programmer.

Referencing Arrays

Continuing with our row of boxes analogy, an array is referenced by its name and the box we wish to work with. For example, to get the value of box 0 we use myIntegerArray(0); to reference box 9 we use myIntegerArray(9).

```
myIntegerArray(0) = 10
myIntegerArray(1) = 36
myIntegerArray(2) = 77
myIntegerArray(4) = 87
myIntegerArray(5) = -10
```

myIntegerArray: Array of Integers (0..9)									
0	1	2	3	4	5	6	7	8	9
10	36	77	87	-10	0	0	0	0	0

Figure 8.7

Here are some useful Strings, 10 Top-Level Domains:

```
myStringArray (0) = "UK" : myStringArray (5) = "ME"
myStringArray (1) = "RU" : myStringArray (6) = "COM"
myStringArray (2) = "HR" : myStringArray (7) = "INFO"
myStringArray (3) = "DE" : myStringArray (8) = "NET"
myStringArray (4) = "FR" : myStringArray (9) = "EU"
```

myStringArray: Array of Strings (0..9)									
0	1	2	3	4	5	6	7	8	9
UK	RU	HR	DE	FR	ME	COM	INFO	NET	EU

Figure 8.8

Fixed Length and Dynamic Arrays

One of the headaches with arrays is that they are static blocks of memory and are not designed to change in size. If we want to add another 5 domain names to myStringArray we have to re-declare the array. Oh, btw, doing so usually gives you back a new clean array!

ReDim and Preserve

VBA offers the ReDim function which performs much of the leg-work involved in changing an array's size. ReDim also has a useful keyword Preserve which preserves the data in your array as you change its size.

```
Dim myIntegerArray() as Integer ' define array variable
ReDim myIntegerArray(10)        ' set array size and memory allocation
myIntegerArray(0) = 22          ' set (0) to 22
ReDim Preserve myIntegerArray(20)' extend array preserving (0)=22
```

Figure 8.9

The standard ReDim function would destroy the old array and make a new one; with the Preserve keyword included VBA creates the new array of the new size and copies over the previous arrays values, making them available to us.

Fixed vs Dynamic Arrays

A fixed-length array is what the above arrays are called – they cannot be changed, or at least not without a great deal of effort. A dynamic array is more flexible allowing the array to grow and shrink in size over time without having to recreate the array data and structure.

VBA doesn't innately support dynamic arrays but using the ReDim function does provide semi-dynamic behaviour as shown above. VBA does however support Strings dynamically.

A String is an array containing an arbitrary number of Characters and by default has no size limit. myString below is being allocated and reallocated, extended, memory managed and whatever is needed to make our Strings persist in memory, and all in the background – we have no idea what VBA is doing and nor do we care. The boxes below illustrates line 4.

```
3   myString = "Hello"
4   myString = "HELLO WORLD"
5   myString = "foobar"
6   myString = "foo foo bar bar"
```

myString: Array of Chars (0…9)									
0	1	2	3	4	5	6	7	8	9
H	E	L	L	O		W	O	R	L

Figure 8.10

Variant Arrays

Another type of array that VBA implements is the Variant Array. Variant arrays handle all primitive types and each element of the array can be loaded with any data type. This contrasts with "standard" arrays which can hold only one primitive data type.

Variant arrays handle just like regular arrays, requiring us to ReDim to change the number of variables it can store.

```
' A variant array can hold any primitive data type, but it is
' actually stored as an object

Dim myVariableArray As Variant
myVariableArray    = Array(10)
myVariableArray(0) = "First element"
myVariableArray(1) = 2
myVariableArray(2) = new Date(#12-09-1989#)
```

Figure 8.11

Erasing an Array

Erasing an array is so important that VBA – a language that usually makes things easy for programmers – provides a dedicated function to release memory held by an array. If one doesn't remove an array VBA will *garbage collect* memory space left when variables go out of scope, but you are advised to explicitly erase array structures when finished with them. Once erased the variable must be ReDim'd.

```
Dim myVariableArray() As Variant
ReDim myVariableArray(10)
myVariableArray(0) = 1
myVariableArray(1) = 2
myVariableArray(2) = 3

Erase myVariableArray ' myVariableArray has no more data and must be
ReDim'd to be used
```

Figure 8.12

Split Function

The split function splits a string into an array of strings based on some delimiter. The following example demonstrates splitting a string based on spaces. Execute it in the Immediate window.

```
Sub SplitFunction()
Dim i As Integer
    A = Split("here;we;go;again!", ";")
    For i = LBound(A) To UBound(A)
        Debug.Print A(i)
    Next i
End Sub
```

Figure 8.13

Join Function

Join does the exact opposite of split; it requires an array and a delimiter and returns a single string.

```
Sub JoinFunction()
A = Array("here", "we", "go", "again", "!")
Debug.Print Join(A, " ")
End Sub
```

Figure 8.14

Multi-Dimensional Arrays

All the arrays shown above are one-dimensional arrays. It is also possible to create an array with more than one dimension. For example, you may have an array of week numbers with days to hold an Integer number.

```
Dim myIntegerArray() as Integer
ReDim myIntegerArray(52,7)
```

Figure 8.15

While myIntegerArray (52) tells VBA to allocate 52 boxes for Integers values, myIntegerArray (52, 7) requires 364 boxes of integers (52 x 7)! The one dimensional array took up 208 bytes of memory, the two dimensional array takes up 1456 bytes.

Three-dimensional arrays take up yet more space and can be declared by appending another dimension to the Dim or ReDim statement, e.g. myIntegerArray(52,7,24) and takes up 34944 bytes.

```
Sub MultiDimendionalArrays()
Dim myIntegerArray() As Integer
ReDim myIntegerArray(7, 52)

myIntegerArray(0, 0) = 1: myIntegerArray(1, 2) = 2
myIntegerArray(3, 3) = 3: myIntegerArray(2, 3) = 4
myIntegerArray(3, 3) = 5: myIntegerArray(6, 1) = 6

End Sub
```

myIntegerArray: Array of Integer (0..7)(0...52)								
Y	X 0	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	6	0
2	0	2	0	0	0	0	0	0
3	0	0	4	3	0	5	0	0
...	0	0	0	0	0	0	0	0

Figure 8.16

The example above illustrates clearly the idea of two-dimensional arrays; like a spreadsheet. Three-dimensional arrays would be like an Excel spreadsheet file with its multiple sheets, and a four-dimensional array is like many Excel files (contained within are sheets, which contain rows and columns, which contain cells), and so on.

Collections

A Collection is an object that stores other objects. Usually a collection will store objects of a particular type so servicing those objects with functionality specifically required by them. When using collections we don't need to worry about ReDim'ing them; the collection will increase in size all by itself so we need only declare a variable to point to a Collection Object and instantiate a new Collection Object.

```
Function makeCar() As Collection
  Dim parts As New Collection
  Dim part As Variant

  parts.Add "Volvo"
  parts.Add 5
  parts.Add "V70R"
  parts.Add "Sunroof"
  parts.Add "Drive"
  parts.Add #12/24/2012#

  Dim t As Integer
  For Each part In parts
    t = t + 1
    Debug.Print t, part, TypeName(part)
  Next

End Function
```

Figure 8.17

The above code creates a new Collection object, adds some primitive types to it then cycles through the Collection outputting it as position, value and data type. Below is the output.

```
     1            Volvo         String
     2             5            Integer
     3            V70R          String
     4            Sunroof       String
     5            Drive         String
     6            24/12/2012    Date
```

Figure 8.18

Relationship with Objects

Collections are used everywhere in VBA and Access. For example, AllForms, AllQueries, AllReports, AllMacros, AllModules, AllViews, Form.Controls, Page.Properties, Form.Properties … Report.Controls … basically a collection is used to hold everything about your application, even collections inside collections. In VBA there are dozens of different collections and although they all inherit from a generic Collections Class one must work the particular Collection for a particular object.

Properties Associated with Objects

Every class of object in VBA and Access have a Properties Collection that for the most part is built-in to the class. It is possible to create user-defined properties and add them to a class instance. Overall the properties of an object uniquely characterise the object and differentiate it.

Practical Uses of Collections : Form and Report Controls

```
Function listCtrlsAndProperties()
  Dim ctrls As Access.controls
  Dim ctrl As Access.control
  Dim prop As Property, tmp As String

  ' the Forms collection is only populated with instantiated forms
  DoCmd.OpenForm "frmCoursesNav", acDesign,,,,acHidden

  Set ctrls = Application.Forms("frmCoursesNav").controls

  For Each ctrl In ctrls
    Debug.Print ctrl.name
    For Each prop In ctrl.Properties
      tmp = tmp & "," & prop.name
    Next
    Debug.Print tmp
    tmp = ""
  Next
  DoCmd.Close acForm, "frmCoursesNav"
End Function
```

Figure 8.19

Note
The Set operator is required because we are dealing with Objects and not primitive types – this is a VBA specific requirement.

The above function opens a form in design view, cycles through the controls collection of the form, then for each control cycles through its properties collection. This demonstrates that a Collection can hold a Collection – the Controls collection has individual Controls which in turn have a Properties Collection and individual properties, like Height, BackColor and ForeColor.

You can also change the appearance of objects on the screen by changing related properties. Use the following instruction to change the colour and height of a button on a form

- Open a new form and save it with the name “frmButtonChangeTest”.
- Add to the form a button and call it “btnChangeAppearance”.
- Navigate to the Events tab and double-click the On Click event and open the VBA Editor.
- Insert the following code

```
Private Sub btnChangeAppearance_Click()

  Dim lHeight As Double
  Dim btn As CommandButton

'  Set btn = Forms!frmButtonChangeTest.controls!btnChangeAppearance
'  lHeight = btn.Height    : btn.Height = lHeight + 50

  lHeight = Me.controls("btnChangeAppearance").Properties("Height")
  Me.controls("btnChangeAppearance").Properties("Height")=lHeight+50

End Sub
```

Figure 8.20

Commented out is one way of manipulating the height property of the button control.
Another valid way is to access properties directly via the Properties collection of the control.
The brackets show a key and the result is a value.

So, *btnChangeAppearance* is a key of controls – this selects the button.
Chained to this is a request for the property Height – height is a key here.
The value of Height is changed by assigning lHeight+50.

Collections: Control. ControlType

Using the collection Controls of a form all controls can be cycled and only those of a particular type can be targeted. In the following example all controls of the form are checked for 1) their type and 2) the section in which they appear. If a control is in the Detail of the form their values are output to the Immediate Window.

```
Private Sub Form_BeforeUpdate(Cancel As Integer)

  Dim c As Variant
  For Each c In Me.Form.controls

    If c.ControlType = acTextBox And c.Section = acDetail Then
      Debug.Print c.name & " = '" & c & "'"
    End If

  Next

End Sub
```

Figure 8.21

Checking if a Form is loaded

To check whether a form is currently open or not use the CurrentProject.AllForms collection which has an IsLoaded function which returns true if the form is loaded. CurrentProject also contains all the other All* collections.

```
Function isMyFormOpen(frmName As String) As Boolean
  isMyFormOpen = CurrentProject.AllForms(frmName).IsLoaded
End Function
```

```
? isMyFormOpen("frmStudentsDataEntry")
False
```

Figure 8.22

Referencing Controls

Me keyword

The Me keyword is associated with classes and object modules – using it in the standard module will result in a compilation error. In a Form module, Me refers to the form itself. Writing "me" tells VBA to reference the current form or report.

```
Option Compare Database

Private Sub Command11_Click()
  MsgBox Me.Form.name  ' msgbox opens with the form's name
End Sub

Private Sub Command12_Click()
  MsgBox Me.Form!field1  ' msgbox opens display content of field1
End Sub
```

```
? isMyFormOpen("frmClassesNav")
False
```

Figure 8.23

Full Form Reference

Referencing the form itself can be performed by writing:

```
Option Compare Database

Private Sub Command0_Click()
  MsgBox Forms(Form.name).name  ' msgbox opens with the form's name
End Sub
```

Figure 8.24

You may also reference another form if it is open. All open forms are held in the Forms collection. Accessing other forms is very helpful when passing data between forms or setting up a form that edits a child record of the first form.

```
Private Sub Command2_Click()
  If AllForms("otherform").IsLoaded Then
    Forms("otherform").controls("customerID") = Me![CustomerID]
    Forms("otherform").FilterOn = True
  End If
End Sub
```

Figure 8.25

Sub Form Reference

A form may be embedded into a parent form so showing records of some child table. The subform can be accessed by accessing the embedded form's name. The subform is added to the parent form's Controls collection so is referenced like *any other control on the form*.

```
Private Sub Command2_Click()
    Me.frmCarDataSub.Form.Detail.BackColor = vbRed
End Sub
```

Figure 8.26

This last item demonstrates what this whole unit is about. Arrays and Collections are the containers of all our data and highly versatile. They are only lists of primitive data or lists of objects but they take up the most space and the most resources. Creating an array can sink a system or make it run lightning fast, as long as it is well maintained.

Common Errors

Not Releasing Memory

Whenever you instantiate an object you should always release the memory. Explicitly releasing memory by erasing arrays or removing an object from a collection forces VBA, .Net or Java to process that memory hole. Leaving objects floating and relying on garbage collectors can slow down you application, and worse cause memory leaks.

Out of Memory

Not releasing arrays and collections, or requesting too much space can result in an Out of Memory error. This was quite frequent 10 years ago, and even now with virtual memory on TB hard drives, running out of memory is possible

Slooooooow Response Times

Again, creating arrays and collections you will not use. When you request a block of memory your computer will allocate it. When that memory isn't in use or doesn't fit into physical memory, it will be swapped out to a hard drive or SD Card, and getting that data back into memory can result in serious slowdown.

Exception: Out of Bounds

Make sure not to attempt to access elements of an array that don't exist by knowing the upper and lower bounds of your arrays. Collections in VBA start at 1. Arrays usually start at 0 but may start at 1. The upper bounds of arrays shouldn't be passed either; this can cause Out of Bounds exceptions, or in a really bad situation may try to execute data as if it were instructions – that is how viruses get their code executed.

Questions

1) Describe the structure of an array?

2) What is the difference between a dynamic array and a fixed length array?

3) Which of the following defines an array or pointer to an array correctly in VBA?
    a. Integer[] myIntegerArray;
    b. Dim myStringArray = new String(10)
    c. Dim myStringArray;
    d. ReDim myIntegerArray(10)
    e. Dim myDateArray() as Date

4) Why can arrays and collection cause many problems?
    a. They fire too many rounds
    b. They can take up a lot of memory
    c. CPU time can be huge
    d. Virtual memory can be used up
    e. Collections are never a problem

5) The following code wipes out the old data. Correct it to maintain old data.

```
Dim integerArray(3) as Integer
integerArray(0) = 20
integerArray(1) = 99
integerArray(2) = 887

ReDim integerArray(10)
integerArray(3) = 44
```

6) aString = “My son went to market and brought dried bananas”
    What letter appears in the following?
    a. aString(9)
    b. aString(31)
    c. a = 20 : aString(a)
    d. c = 4 * 8 : aString(c)
    e. instr(1,aString,"y")
    f. aString(instr(1,aString,"i"))

7) Which of the following are not VBA or Access collections?
    a. AllForms
    b. AllModules
    c. AllStrings
    d. Report.Controls
    e. Properties
    f. Fields
    g. Recordset.Fields

h. Me.Controls

8) Fill out the following table.

```
myIntegerArray = array(8,9,10,5,3,23,65,99,121,00)
```

myIntegerArray: Array of Integers (0..9)									
0	1	2	3	4	5	6	7	8	9

9) Fill out the following table.

```
myString = "There,follows,a,party,political,broad,cast,!,?"
myStringArray = Split(myString, ",")
```

myStringArray: Array of Strings									
0	1	2	3	4	5	6	7	8	9

10) From (9) complete the following Immediate window statement to print all array elements.

```
For Each ___ In _____________ : ? a : next
```

11) Fill in the missing numbers.

```
3   myStringArray (___) = "INFO" : myStringArray (___) = "DE"
4   myStringArray (___) = "RU"   : myStringArray (___) = "HR"
5   myStringArray (___) = "COM"  : myStringArray (___) = "FR"
6   myStringArray (___) = "DE"   : myStringArray (___) = "NET"
7   myStringArray (___) = "ME"   : myStringArray (___) = "UK"
```

myStringArray: Array of Strings (0..9)									
0	1	2	3	4	5	6	7	8	9
UK	RU	HR	DE	FR	ME	COM	INFO	NET	EU

12) Why does the following code not work? Correct it. What is the output?

```
Function collectionsTest1()
  Dim col As New Collection
  Dim num As Integer

  num = 10: col.Add num
  num = 30: col.Add num
  num = 88: col.Add num
  num = 30: col.Remove num
  num = col.Item(1): Debug.Print num
  collectionsTest1 = num
End Function
```

13) If the following needs changing, change it so that line 11 returns littleArray(1) = 66.

```
1   Function arrayTest2()
2     Dim littleArray(4) As Integer
3     littleArray(0) = 1
4     littleArray(1) = 99
5     littleArray(2) = 5
6     littleArray(3) = 67
7     Erase littleArray
8     littleArray(0) = 1
9     littleArray(1) = 66
10    littleArray(2) = 5
11    arrayTest2 = littleArray(1)
12  End Function
```

14) Write a multi-dimensional array that is called and represents a chessboard that could hold the text queen, king, bishop, knight, rook, pawn.

15) True or false

a. Collections are a string of characters
b. Less memory is used by an object in a collection than an integer in an array
c. Arrays are slower to access than a collection
d. A variant array may hold objects
e. Arrays are instantiated
f. To increase the size of a collection we used ReDim
g. Preserving an array maintains its size and clears the content
h. c = 10 / 2: Dim A() As Integer: ReDim A(5): Debug.Print UBound(A) = c
i. Arrays are instantiated

16) What is special about the Forms collection?

17) SubForm KOL can be found where in relation to Me?

18) Are Strings, by default, dynamic or fixed length arrays of characters?

19) How does VBA implement dynamic arrays for primitive types?

20) If an Double takes up 8 bytes of memory space, and a Float takes by 8 bytes of memory space, and one character of a String takes up 2 bytes of memory space, rank the following in order of size, smallest to largest:

Float(5)	
Double(6)	
String “Foobar”	

Float 6.77	
String(1)	
Double(2,5)	
Double(3,3,3)	
Float(6,1)	
String(10,2)	

Answers - Conditionals and Branching

1.

```
Dim myName as String
myName = getUsername()  ' returns user's name

If myName = "Mat" Then MsgBox "Hi Mat"
If myName = "John" Then MsgBox "Hi John"
If myName = "Sarah" Then Print "Hi Sarah"

Dim l as Integer
l = len(myName)
If l > 4 And l < 10 Then
  Debug.Print "Length of myName is " + CStr(l)
End If
```

2. True or false

3. a) true, b) true, c) false, d) true, e) false

4. a) false, b) true, c) true, d) true, e) true, f) true

5. It aids readability which reduces the likelihood of errors.

6. 1 and 1

7. (a), (c)

8. Line 12 should read RESTOCKING not CLOSED.

9. See Arithmetic Operators on page 11

10. a) A and B must be the same
    b) A must be larger than B
    c) A must not change and B > A

11. One may be a String, one may be a single or float.

12. (A) = AND, (B) = OR, (C) = NOT, (D) = XOR

13. a) False, b) True, c) False, d) True, e) True, f) False

14. Think nested expressions!

15.

```
If a = b And c = a Then
  MsgBox "Might be true"
  Debug.Print "and so may this"
Else
  Debug.Print “It’s Twins!”
End If
```

16.

```
Enum Status
  INCREASE_TEMP
  DECREASE_TEMP
  WARM_UP
  COOL_DOWN
  FAN_ON
  fan_off
End Enum

Function P(airconStatus As Status) As Long

  Select Case airconStatus
    Case Is = Status.INCREASE_TEMP: s = 1
    Case Is = Status.DECREASE_TEMP: s = 2
    Case Is = Status.FAN_ON: s = 4
    Case Is = Status.fan_off: s = 8
    Case Is = Status.WARM_UP: s = 16
    Case Is = Status.COOL_DOWN: s = 32
    Case Else: s = 64
  End Select
  P = s

End Function
```

17. a) The Semaphore is False
b) The Semaphore is True
c) The Semaphore is False

18. a=False, b=False, c=True

19. With can make a block easier to read by taking out repetitive code. Particularly useful when you are accessing deeply nested properties of objects.

20. Because the types are not equal. If you put <> between them the answer is True.

Answers - Arrays and Collections

1) Traditionally, an Array has been a block of memory put aside to hold values of a particular type. Its size is set at the time it is initiated and any element within it may be accessed randomly or sequentially.
2) Dynamic can change over time whilst a fixed cannot
3) Yes or no
    a. No
    b. No
    c. Yes
    d. Yes
    e. Yes
4) Yes or no
    a. No
    b. Yes
    c. Yes
    d. Yes
    e. No
5) Line 6: `ReDim Preserve integerArray(10)`
6) Letters below
    a. e
    b. g
    c. e
    d. h
    e. 2
    f. i
7) yes or no
    a. yes
    b. yes
    c. no
    d. yes
    e. yes
    f. yes
    g. yes
    h. yes
8) see below

```
myIntegerArray = array(8,9,10,5,3,23,65,99,121,00)
```

myIntegerArray: Array of Integers (0..9)									
0	1	2	3	4	5	6	7	8	9
8	9	10	5	3	23	65	99	121	0

9) see below

```
myString = "There,follows,a,party,political,broad,cast,!,?"
myStringArray = Split(myString, ",")
```

myStringArray: Array of Strings									
0	1	2	3	4	5	6	7	8	9
There	follows	a	party	political	broad	cast	!	?	

```
10)      For Each _a_ In __myStringArray__ : ? a : next
```

11) See below

```
3   myStringArray (_7_) = "INFO" : myStringArray (_3_) = "DE"
4   myStringArray (_1_) = "RU"   : myStringArray (_2_) = "HR"
5   myStringArray (_6_) = "COM"  : myStringArray (_4_) = "FR"
6   myStringArray (_3_) = "DE"   : myStringArray (_8_) = "NET"
7   myStringArray (_5_) = "ME"   : myStringArray (_0_) = "UK"
```

myStringArray: Array of Strings (0..9)									
0	1	2	3	4	5	6	7	8	9
UK	RU	HR	DE	FR	ME	COM	INFO	NET	EU

12) Line 8 causes an out of bounds error
change to num=2
collectionsTest1 = 10

13) Cheeky answer, comment out line 7
otherwise place a ReDim littleArray(3) after line 7

14) Dim Chessboard(8,8) As String

15) True or false

a. False
b. False
c. False
d. True
e. False
f. False
g. False
h. True (sorry)
i. False

16) Forms only contains those forms that are open

17) Me.KOL or Me.Controls("KOL")

18) Dynamic

19) By using ReDim

20) See below

Float(5)	40	4
Double(6)	48	6
String “Foobar”	12	3
Float 6.77	8	2
String(1)	2	1
Double(2,5)	80	8
Double(3,3,3)	216	9
Float(6,1)	48	7
String(10,2)	40	5