

Module 2 - Multiplication Table - Part 1

TOPICS COVERED:

- 1) VBA and VBA Editor (0:00)
- 2) Arrays (1:15)
- 3) Creating a Multiplication Table (2:34)
- 4) TimesTable Subroutine (3:08)
- 5) Improving Efficiency of the TimesTable Routine (7:30)
- 6) Excel Worksheet Calculation Engine vs. VBA Engine (8:20)
- 7) A Much Better Way (9:15)
- 8) Learn How to Time Code (11:10)
- 9) Make Your Code Faster (14:20)

What is VBA? (0:00)

Visual Basic for Applications (VBA) is a programming language (based on Visual Basic 6) which is built into most Microsoft Office applications (Excel, Access, Word, Outlook, Visio).

Although functionally rich and flexible, VBA does have many limitations and normally only runs code within its host application rather than as a standalone application.

Within Excel, you can control hundreds of objects from VBA. The Application, Worksheet, and Range, just to name a few.

To reach the VBA IDE (integrated development environment) from within an Office document, just press the key sequence Alt+F11. The IDE is mostly referred to as the VBA Editor in this course.

What is an Array? (1:15)

```
Option Explicit
Option Private Module

Dim MyArray(10, 10)
```

An array is a variable that can hold more than one value.
It is an ordered group of elements in memory.

Arrays can be defined to hold many kinds of info including numbers, text, and objects, and can be defined in many dimensions.

1-dimensional arrays are analogous to a row or a column.

Example: *Dim My Array(10)*

2-dimensional arrays are analogous of a rectangular range, like a worksheet, that has rows *and* columns.

Example: *Dim My Array (10, 10)*

3-dimensional arrays are analogous to a workbook of worksheets (like a cube).

Example: *Dim My Array (10, 10, 10)*

Higher dimensional arrays are possible but not often needed for most applications.

The **Cells** property works like a 2-dimensional array with the first index representing rows and the second index representing columns.

Creating a Multiplication Table (2:34)

	A	B	C	D	E	F	G	H	I	J	K	L
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	14	16	18	20	22	24
3	3	6	9	12	15	18	21	24	27	30	33	36
4	4	8	12	16	20	24	28	32	36	40	44	48
5	5	10	15	20	25	30	35	40	45	50	55	60
6	6	12	18	24	30	36	42	48	54	60	66	72
7	7	14	21	28	35	42	49	56	63	70	77	84
8	8	16	24	32	40	48	56	64	72	80	88	96
9	9	18	27	36	45	54	63	72	81	90	99	108
10	10	20	30	40	50	60	70	80	90	100	110	120
11	11	22	33	44	55	66	77	88	99	110	121	132
12	12	24	36	48	60	72	84	96	108	120	132	144

Make a 12 x 12 table. Or in other words, a 12 x 12 two-dimensional array.

The goal is to create this same table using VBA.

TimesTable Subroutine (3:08)

```
Public Sub TimesTable()

    Dim row As Long
    Dim col As Long

    Sheet1.Cells.ClearContents 1
    For row = 1 To 12
        2 For col = 1 To 12
            Sheet1.Cells(row, col) = row * col
        Next
    Next

End Sub
|
```

The TimesTable subroutine is defined as a Public subroutine which allows us to access it from the Macro dialogue box.

Dissecting this code subroutine:

(1) ClearContents method: This method clears just the values of the cell range specified. It does not erase any formatting of cells.

(2) Nesting Loops: You can next for / next loops as many times as you wish, but for a two-dimensional array only two are needed. The first loop handles rows while the second loop handles columns.

There are several types of loops. The For-Next loop is the most common. It requires a counting variable.

Improving Efficiency of the TimesTable Routine (7:30)

```
Public Sub TimesTable()  
  
    Dim row As Long  
    Dim col As Long  
  
    With Sheet1  
        .Cells.ClearContents  
        For row = 1 To 12  
            For col = 1 To 12  
                .Cells(row, col) = row * col  
            Next  
        Next  
    End With  
  
End Sub
```

Although this nested loop is the most common approach, it is also the *slowest* of the examples. Why?

Each reference to the Cells property, the VBA engine has to do some work in order to evaluate the full reference. In the TimesTable subroutine above, this happens 145 times. Once for the ClearContents method and 144 times for writing to each cell.

One simple tweak to enhance performance... by using **With Sheet1** allows us to reference any method

or property of **Sheet1** by just using the dot (".") before the property or the method. In the example above, we are interested in the Cells property, so we use **.Cells**.

While this is more efficient because the reference to Cells is only evaluated once versus 145 times. But for such a small sample size, VBA is fast enough so that we probably will not perceive much difference.

Excel Worksheet Calculation Engine versus VBA Engine (8:20)

The Excel worksheet calculation engine and VBA engine are separate domains and, in fact, use separate DLL's.

*****So effectively, there is a barrier between them, and passing through that barrier is slow.*****

The VBA engine has to do a lot of work behind the scenes to setup for each call through the barrier and to tear down after each call.

Although the TimesTable subroutine is algorithmically sound, it forces VBA to do this "setup values / pass values through the barrier / tear down construct" process 144 times.

A Much Better Way (9:15)

```
[a1:L12] = [row(1:12)*column(1:1)]
```

On the right-side of the equation, we use an array formula to produce the desired array of values. On the left-side, we write the entire array to the multiplication table in one shot.

This one little line of VBA code replaced the entire TimesTable subroutine. It does it much quicker and without declaring variables or using any loops. *It is a very sophisticated line of code, indeed.*

It uses the short form of the Evaluate method of the Application object to do Worksheet calculations on both sides of the equal sign.

Just how fast is this? Learn How to Time Code (11:10)

```
Option Explicit
'Option Private Module
Private Declare Function GetTickCount Lib "ke
```

Need the help of a Windows API called **GetTickCount**. It's very simple to use. Just declare it at the top of the module in which you wish to use it.

The full line of code should read: *Private Declare Function GetTickCount Lib "kernel32" As Long*

Learn How to Time Code - Create the Timing Procedures

```
Private Sub TStart()

    Cells.ClearContents
    lngStart = GetTickCount()

End Sub

Private Sub TEnd()

    MsgBox GetTickCount() - lngStart & " ms"

End Sub
```

These procedures will actually do the "timing" for us. Subroutines TStart() and TEnd().

Learn How to Time Code - Global Variable

```
Option Explicit
'Option Private Module
Private Declare Function GetTickCount Lib "ke
Private lngStart As Long
```

Since we are sharing the variable **lngStart** between these two subroutines, it must be declared as a *global variable*.

A *local variable* is only accessible to the routine in which it is dimensionalized. A global variable, though, is accessible anywhere within the module.

Simply declare the variable at the top of the module.

Learn How to Time Code - Add Calls to the Timer Within Your Routine (12:50)

```
Public Sub TimesTable()  
  
    Dim row As Long  
    Dim col As Long  
  
    With Sheet1  
        TStart  
        For row = 1 To 12  
            For col = 1 To 12  
                .Cells(row, col) = row * col  
            Next  
        Next  
        TEnd  
    End With  
  
End Sub
```

Add a call to TStart and to TEnd to each routine you wish to time.

Make It Even Faster! (14:20)

```
[a1:L12] = [row(1:12)*column(1:1)]
```

In the above code, we reference ALL columns on the entire first row (1:1). This forces the array formula to take into account *all* 16,000+ columns in the row when calculating *each* of the 144 answers.

Make It Even Faster!

```
TStart  
[sheet1!a1:L12] = [row(1:12)*column(a1:L1)]  
TEnd
```

To make this even faster still, we change the **column(1:1)** to **column(a1:L1)**. Now, our subroutine will only calculate the 12 columns within the times table cells.

This offers a *significant* performance improvement, especially as more and more calculations are required.

While this one line of code is the most condensed and most readable, it does have one final bottleneck.

This code forces the Excel worksheet calculation engine to calculate the value array.

Send it to the VBA Engine, through the barrier.

Then return the array *back* through the barrier to the destination range.

Two trips through the barrier.

Fastest of All... (17:10)

```
Public Sub TimesTable2()  
  
    Dim row As Long  
    Dim col As Long  
    Dim a(1 To 1000, 1 To 100) As Long  
  
    With Sheet1  
        TStart  
        For row = 1 To 1000  
            For col = 1 To 100  
                a(row, col) = row * col  
            Next  
        Next  
        [sheet1!a1:cv1000] = a  
        TEnd  
    End With  
  
End Sub
```

The quickest routine of all is to create the array on the VBA-side. Then send that array of values through the barrier to the destination range.

One trip through the barrier between VBA engine and the worksheet calculation engine leads to optimal performance!