Chapter 18: Arrays

Section 18.1: Multidimensional Arrays

Multidimensional Arrays

As the name indicates, multi dimensional arrays are arrays that contain more than one dimension, usually two or three but it can have up to 32 dimensions.

A multi array works like a matrix with various levels, take in example a comparison between one, two, and three Dimensions.

One Dimension is your typical array, it looks like a list of elements.

```
Dim 1D(3) as Variant

*1D - Visually*
(0)
(1)
(2)
```

Two Dimensions would look like a Sudoku Grid or an Excel sheet, when initializing the array you would define how many rows and columns the array would have.

```
Dim 2D(3,3) as Variant
'this would result in a 3x3 grid

*2D - Visually*
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
(2,0) (2,1) (2,2)
```

Three Dimensions would start to look like Rubik's Cube, when initializing the array you would define rows and columns and layers/depths the array would have.

```
Dim 3D(3,3,2) as Variant
'this would result in a 3x3x3 grid

*3D - Visually*
       1st layer                2nd layer                  3rd layer
         front                    middle                     back
(0,0,0) (0,0,1) (0,0,2) ¦ (1,0,0) (1,0,1) (1,0,2) ¦ (2,0,0) (2,0,1) (2,0,2)
(0,1,0) (0,1,1) (0,1,2) ¦ (1,1,0) (1,1,1) (1,1,2) ¦ (2,1,0) (2,1,1) (2,1,2)
(0,2,0) (0,2,1) (0,2,2) ¦ (1,2,0) (1,2,1) (1,2,2) ¦ (2,2,0) (2,2,1) (2,2,2)
```

Further dimensions could be thought as the multiplication of the 3D, so a 4D(1,3,3,3) would be two side-by-side 3D arrays.

Two-Dimension Array

Creating

The example below will be a compilation of a list of employees, each employee will have a set of information on the list (First Name, Surname, Address, Email, Phone ...), the example will essentially be storing on the array

(employee,information) being the (0,0) is the first employee's first name.

```
Dim Bosses As Variant
'set bosses as Variant, so we can input any data type we want

Bosses = [{"Jonh","Snow","President";"Ygritte","Wild","Vice-President"}]
'initialise a 2D array directly by filling it with information, the redult wil be a array(1,2) size
2x3 = 6 elements

Dim Employees As Variant
'initialize your Employees array as variant
'initialize and ReDim the Employee array so it is a dynamic array instead of a static one, hence
treated differently by the VBA Compiler
ReDim Employees(100, 5)
'declaring an 2D array that can store 100 employees with 6 elements of information each, but starts
empty
'the array size is 101 x 6 and contains 606 elements

For employee = 0 To UBound(Employees, 1)
'for each employee/row in the array, UBound for 2D arrays, which will get the last element on the
array
'needs two parameters 1st the array you which to check and 2nd the dimension, in this case 1 =
employee and 2 = information
    For information_e = 0 To UBound(Employees, 2)
    'for each information element/column in the array

        Employees(employee, information_e) = InformationNeeded ' InformationNeeded would be the
data to fill the array
        'iterating the full array will allow for direct attribution of information into the element
coordinates
    Next
Next
```

Resizing

Resizing or `ReDim Preserve` a Multi-Array like the norm for a One-Dimension array would get an error, instead the information needs to be transferred into a Temporary array with the same size as the original plus the number of row/columns to add. In the example below we'll see how to initialize a Temp Array, transfer the information over from the original array, fill the remaining empty elements, and replace the temp array by the original array.

```
Dim TempEmp As Variant
'initialise your temp array as variant
ReDim TempEmp(UBound(Employees, 1) + 1, UBound(Employees, 2))
'ReDim/Resize Temp array as a 2D array with size UBound(Employees)+1 = (last element in Employees 1st
dimension) + 1,
'the 2nd dimension remains the same as the original array. we effectively add 1 row in the Employee
array

'transfer
For emp = LBound(Employees, 1) To UBound(Employees, 1)
    For info = LBound(Employees, 2) To UBound(Employees, 2)
        'to transfer Employees into TempEmp we iterate both arrays and fill TempEmp with the
corresponding element value in Employees
        TempEmp(emp, info) = Employees(emp, info)

    Next
Next

'fill remaining
'after the transfers the Temp array still has unused elements at the end, being that it was increased
```

```
'to fill the remaining elements iterate from the last "row" with values to the last row in the array
'in this case the last row in Temp will be the size of the Employees array rows + 1, as the last row
of Employees array is already filled in the TempArray

For emp = UBound(Employees, 1) + 1 To UBound(TempEmp, 1)
    For info = LBound(TempEmp, 2) To UBound(TempEmp, 2)

        TempEmp(emp, info) = InformationNeeded & "NewRow"

    Next
Next

'erase Employees, attribute Temp array to Employees and erase Temp array
Erase Employees
Employees = TempEmp
Erase TempEmp
```

Changing Element Values

To change/alter the values in a certain element can be done by simply calling the coordinate to change and giving it a new value: `Employees(0, 0) = "NewValue"`

Alternatively iterate through the coordinates use conditions to match values corresponding to the parameters needed:

```
For emp = 0 To UBound(Employees)
    If Employees(emp, 0) = "Gloria" And Employees(emp, 1) = "Stephan" Then
    'if value found
        Employees(emp, 1) = "Married, Last Name Change"
        Exit For
        'don't iterate through a full array unless necessary
    End If
Next
```

Reading

Accessing the elements in the array can be done with a Nested Loop (iterating every element), Loop and Coordinate (iterate Rows and accessing columns directly), or accessing directly with both coordinates.

```
'nested loop, will iterate through all elements
For emp = LBound(Employees, 1) To UBound(Employees, 1)
    For info = LBound(Employees, 2) To UBound(Employees, 2)
        Debug.Print Employees(emp, info)
    Next
Next

'loop and coordinate, iteration through all rows and in each row accessing all columns directly
For emp = LBound(Employees, 1) To UBound(Employees, 1)
    Debug.Print Employees(emp, 0)
    Debug.Print Employees(emp, 1)
    Debug.Print Employees(emp, 2)
    Debug.Print Employees(emp, 3)
    Debug.Print Employees(emp, 4)
    Debug.Print Employees(emp, 5)
Next

'directly accessing element with coordinates
```

```
Debug.Print Employees(5, 5)
```

***Remember**, it's always handy to keep an array map when using Multidimensional arrays, they can easily become confusion.*

Three-Dimension Array

For the 3D array, we'll use the same premise as the 2D array, with the addition of not only storing the Employee and Information but as well Building they work in.

The 3D array will have the Employees (can be thought of as Rows), the Information (Columns), and Building that can be thought of as different sheets on an excel document, they have the same size between them, but every sheets has a different set of information in its cells/elements. The 3D array will contain ***n*** number of 2D arrays.

Creating

A 3D array needs 3 coordinates to be initialized `Dim 3Darray(2,5,5) As Variant` the first coordinate on the array will be the number of Building/Sheets (different sets of rows and columns), second coordinate will define Rows and third Columns. The `Dim` above will result in a 3D array with 108 elements (`3*6*6`), effectively having 3 different sets of 2D arrays.

```
Dim ThreeDArray As Variant
'initialise your ThreeDArray array as variant
ReDim ThreeDArray(1, 50, 5)
'declaring an 3D array that can store two sets of 51 employees with 6 elements of information each,
but starts empty
'the array size is 2 x 51 x 6 and contains 612 elements

For building = 0 To UBound(ThreeDArray, 1)
    'for each building/set in the array
    For employee = 0 To UBound(ThreeDArray, 2)
    'for each employee/row in the array
        For information_e = 0 To UBound(ThreeDArray, 3)
        'for each information element/column in the array

            ThreeDArray(building, employee, information_e) = InformationNeeded ' InformationNeeded
would be the data to fill the array
        'iterating the full array will allow for direct attribution of information into the element
coordinates
        Next
    Next
Next
```

Resizing

Resizing a 3D array is similar to resizing a 2D, first create a Temporary array with the same size of the original adding one in the coordinate of the parameter to increase, the first coordinate will increase the number of sets in the array, the second and third coordinates will increase the number of Rows or Columns in each set.

The example below increases the number of Rows in each set by one, and fills those recently added elements with new information.

```
Dim TempEmp As Variant
'initialise your temp array as variant
ReDim TempEmp(UBound(ThreeDArray, 1), UBound(ThreeDArray, 2) + 1, UBound(ThreeDArray, 3))
```

```
'ReDim/Resize Temp array as a 3D array with size UBound(ThreeDArray)+1 = (last element in Employees
2nd dimension) + 1,
'the other dimension remains the same as the original array. we effectively add 1 row in the for each
set of the 3D array

'transfer
For building = LBound(ThreeDArray, 1) To UBound(ThreeDArray, 1)
    For emp = LBound(ThreeDArray, 2) To UBound(ThreeDArray, 2)
        For info = LBound(ThreeDArray, 3) To UBound(ThreeDArray, 3)
            'to transfer ThreeDArray into TempEmp by iterating all sets in the 3D array and fill
TempEmp with the corresponding element value in each set of each row
            TempEmp(building, emp, info) = ThreeDArray(building, emp, info)

        Next
    Next
Next

'fill remaining
'to fill the remaining elements we need to iterate from the last "row" with values to the last row in
the array in each set, remember that the first empty element is the original array Ubound() plus 1
For building = LBound(TempEmp, 1) To UBound(TempEmp, 1)
    For emp = UBound(ThreeDArray, 2) + 1 To UBound(TempEmp, 2)
        For info = LBound(TempEmp, 3) To UBound(TempEmp, 3)

            TempEmp(building, emp, info) = InformationNeeded & "NewRow"

        Next
    Next
Next

'erase Employees, attribute Temp array to Employees and erase Temp array
Erase ThreeDArray
ThreeDArray = TempEmp
Erase TempEmp
```

Changing Element Values and Reading

Reading and changing the elements on the 3D array can be done similarly to the way we do the 2D array, just adjust for the extra level in the loops and coordinates.

```
Do
' using Do ... While for early exit
    For building = 0 To UBound(ThreeDArray, 1)
        For emp = 0 To UBound(ThreeDArray, 2)
            If ThreeDArray(building, emp, 0) = "Gloria" And ThreeDArray(building, emp, 1) =
"Stephan" Then
            'if value found
                ThreeDArray(building, emp, 1) = "Married, Last Name Change"
                Exit Do
                'don't iterate through all the array unless necessary
            End If
        Next
    Next
Loop While False

'nested loop, will iterate through all elements
For building = LBound(ThreeDArray, 1) To UBound(ThreeDArray, 1)
    For emp = LBound(ThreeDArray, 2) To UBound(ThreeDArray, 2)
        For info = LBound(ThreeDArray, 3) To UBound(ThreeDArray, 3)
            Debug.Print ThreeDArray(building, emp, info)
        Next
```

```
    Next
Next

'loop and coordinate, will iterate through all set of rows and ask for the row plus the value we
choose for the columns
For building = LBound(ThreeDArray, 1) To UBound(ThreeDArray, 1)
    For emp = LBound(ThreeDArray, 2) To UBound(ThreeDArray, 2)
        Debug.Print ThreeDArray(building, emp, 0)
        Debug.Print ThreeDArray(building, emp, 1)
        Debug.Print ThreeDArray(building, emp, 2)
        Debug.Print ThreeDArray(building, emp, 3)
        Debug.Print ThreeDArray(building, emp, 4)
        Debug.Print ThreeDArray(building, emp, 5)
    Next
Next

'directly accessing element with coordinates
Debug.Print Employees(0, 5, 5)
```

Section 18.2: Dynamic Arrays (Array Resizing and Dynamic Handling)

Dynamic Arrays

Adding and reducing variables on an array dynamically is a huge advantage for when the information you are treating does not have a set number of variables.

Adding Values Dynamically

You can simply resize the Array with the `ReDim` Statement, this will resize the array but to if you which to retain the information already stored in the array you'll need the part `Preserve`.

In the example below we create an array and increase it by one more variable in each iteration while preserving the values already in the array.

```
Dim Dynamic_array As Variant
' first we set Dynamic_array as variant

For n = 1 To 100

    If IsEmpty(Dynamic_array) Then
        'isempty() will check if we need to add the first value to the array or subsequent ones

        ReDim Dynamic_array(0)
        'ReDim Dynamic_array(0) will resize the array to one variable only
        Dynamic_array(0) = n

    Else
        ReDim Preserve Dynamic_array(0 To UBound(Dynamic_array) + 1)
        'in the line above we resize the array from variable 0 to the UBound() = last variable, plus
one effectivelly increeasing the size of the array by one
        Dynamic_array(UBound(Dynamic_array)) = n
        'attribute a value to the last variable of Dynamic_array
    End If

Next
```

Removing Values Dynamically

We can utilise the same logic to to decrease the the array. In the example the value "last" will be removed from the array.

```
Dim Dynamic_array As Variant
Dynamic_array = Array("first", "middle", "last")

ReDim Preserve Dynamic_array(0 To UBound(Dynamic_array) - 1)
' Resize Preserve while dropping the last value
```

Resetting an Array and Reusing Dynamically

We can as well re-utilise the arrays we create as not to have many on memory, which would make the run time slower. This is useful for arrays of various sizes. One snippet you could use to re-utilise the array is to `ReDim` the array back to `(0)`, attribute one variable to to the array and freely increase the array again.

In the snippet below I construct an array with the values 1 to 40, empty the array, and refill the array with values 40 to 100, all this done dynamically.

```
Dim Dynamic_array As Variant

For n = 1 To 100

    If IsEmpty(Dynamic_array) Then
        ReDim Dynamic_array(0)
        Dynamic_array(0) = n

    ElseIf Dynamic_array(0) = "" Then
        'if first variant is empty ( = "") then give it the value of n
        Dynamic_array(0) = n
    Else
        ReDim Preserve Dynamic_array(0 To UBound(Dynamic_array) + 1)
        Dynamic_array(UBound(Dynamic_array)) = n
    End If
    If n = 40 Then
        ReDim Dynamic_array(0)
        'Resizing the array back to one variable without Preserving,
        'leaving the first value of the array empty
    End If

Next
```

Section 18.3: Jagged Arrays (Arrays of Arrays)

Jagged Arrays NOT Multidimensional Arrays

Arrays of Arrays(Jagged Arrays) are not the same as Multidimensional Arrays if you think about them visually Multidimensional Arrays would look like Matrices (Rectangular) with defined number of elements on their dimensions(inside arrays), while Jagged array would be like a yearly calendar with the inside arrays having different number of elements, like days in on different months.

Although Jagged Arrays are quite messy and tricky to use due to their nested levels and don't have much type safety, but they are very flexible, allow you to manipulate different types of data quite easily, and don't need to contain unused or empty elements.

Creating a Jagged Array

In the below example we will initialise a jagged array containing two arrays one for Names and another for

Numbers, and then accessing one element of each

```
Dim OuterArray() As Variant
Dim Names() As Variant
Dim Numbers() As Variant
'arrays are declared variant so we can access attribute any data type to its elements

Names = Array("Person1", "Person2", "Person3")
Numbers = Array("001", "002", "003")

OuterArray = Array(Names, Numbers)
'Directly giving OuterArray an array containing both Names and Numbers arrays inside

Debug.Print OuterArray(0)(1)
Debug.Print OuterArray(1)(1)
'accessing elements inside the jagged by giving the coordenades of the element
```

Dynamically Creating and Reading Jagged Arrays

We can as well be more dynamic in our approx to construct the arrays, imagine that we have a customer data sheet in excel and we want to construct an array to output the customer details.

```
   Name -   Phone   -  Email  - Customer Number
Person1 - 153486231 - 1@STACK - 001
Person2 - 153486242 - 2@STACK - 002
Person3 - 153486253 - 3@STACK - 003
Person4 - 153486264 - 4@STACK - 004
Person5 - 153486275 - 5@STACK - 005
```

We will Dynamically construct an Header array and a Customers array, the Header will contain the column titles and the Customers array will contain the information of each customer/row as arrays.

```
Dim Headers As Variant
' headers array with the top section of the customer data sheet
    For c = 1 To 4
        If IsEmpty(Headers) Then
            ReDim Headers(0)
            Headers(0) = Cells(1, c).Value
        Else
            ReDim Preserve Headers(0 To UBound(Headers) + 1)
            Headers(UBound(Headers)) = Cells(1, c).Value
        End If
    Next

Dim Customers As Variant
'Customers array will contain arrays of customer values
Dim Customer_Values As Variant
'Customer_Values will be an array of the customer in its elements (Name-Phone-Email-CustNum)

    For r = 2 To 6
    'iterate through the customers/rows
        For c = 1 To 4
        'iterate through the values/columns

            'build array containing customer values
            If IsEmpty(Customer_Values) Then
                ReDim Customer_Values(0)
                Customer_Values(0) = Cells(r, c).Value
            ElseIf Customer_Values(0) = "" Then
```

```
                Customer_Values(0) = Cells(r, c).Value
            Else
                ReDim Preserve Customer_Values(0 To UBound(Customer_Values) + 1)
                Customer_Values(UBound(Customer_Values)) = Cells(r, c).Value
            End If
        Next

        'add customer_values array to Customers Array
        If IsEmpty(Customers) Then
            ReDim Customers(0)
            Customers(0) = Customer_Values
        Else
            ReDim Preserve Customers(0 To UBound(Customers) + 1)
            Customers(UBound(Customers)) = Customer_Values
        End If

        'reset Custumer_Values to rebuild a new array if needed
        ReDim Customer_Values(0)
    Next

    Dim Main_Array(0 To 1) As Variant
    'main array will contain both the Headers and Customers

    Main_Array(0) = Headers
    Main_Array(1) = Customers
```

To better understand the way to Dynamically construct a one dimensional array please check Dynamic Arrays (Array Resizing and Dynamic Handling) on the Arrays documentation.

The Result of the above snippet is an Jagged Array with two arrays one of those arrays with 4 elements, 2 indention levels, and the other being itself another Jagged Array containing 5 arrays of 4 elements each and 3 indention levels, see below the structure:

```
Main_Array(0) - Headers - Array("Name","Phone","Email","Customer Number")
          (1) - Customers(0) - Array("Person1",153486231,"1@STACK",001)
                Customers(1) - Array("Person2",153486242,"2@STACK",002)
                ...
                Customers(4) - Array("Person5",153486275,"5@STACK",005)
```

To access the information you'll have to bear in mind the structure of the Jagged Array you create, in the above example you can see that the `Main Array` contains an Array of `Headers` and an Array of Arrays (`Customers`) hence with different ways of accessing the elements.

Now we'll read the information of the `Main Array` and print out each of the Customers information as `Info Type: Info`.

```
For n = 0 To UBound(Main_Array(1))
    'n to iterate from first to last array in Main_Array(1)

    For j = 0 To UBound(Main_Array(1)(n))
        'j will iterate from first to last element in each array of Main_Array(1)

        Debug.Print Main_Array(0)(j) & ": " & Main_Array(1)(n)(j)
        'print Main_Array(0)(j) which is the header and Main_Array(0)(n)(j) which is the element in
the customer array
        'we can call the header with j as the header array has the same structure as the customer
array
    Next
Next
```

REMEMBER to keep track of the structure of your Jagged Array, in the example above to access the Name of a customer is by accessing `Main_Array -> Customers -> CustomerNumber -> Name` which is three levels, to return `"Person4"` you'll need the location of Customers in the Main_Array, then the Location of customer four on the Customers Jagged array and lastly the location of the element you need, in this case `Main_Array(1)(3)(0)` which is `Main_Array(Customers)(CustomerNumber)(Name)`.

Section 18.4: Declaring an Array in VBA

Declaring an array is very similar to declaring a variable, except you need to declare the dimension of the Array right after its name:

```
Dim myArray(9) As String 'Declaring an array that will contain up to 10 strings
```

By default, Arrays in VBA are **indexed from ZERO**, thus, the number inside the parenthesis doesn't refer to the size of the array, but rather to **the index of the last element**

Accessing Elements

Accessing an element of the Array is done by using the name of the Array, followed by the index of the element, inside parenthesis:

```
myArray(0) = "first element"
myArray(5) = "sixth element"
myArray(9) = "last element"
```

Array Indexing

You can change Arrays indexing by placing this line at the top of a module:

```
Option Base 1
```

With this line, all Arrays declared in the module will be **indexed from ONE**.

Specific Index

You can also declare each Array with its own index by using the `To` keyword, and the lower and upper bound (= index):

```
Dim mySecondArray(1 To 12) As String 'Array of 12 strings indexed from 1 to 12
Dim myThirdArray(13 To 24) As String 'Array of 12 strings indexed from 13 to 24
```

Dynamic Declaration

When you do not know the size of your Array prior to its declaration, you can use the dynamic declaration, and the `ReDim` keyword:

```
Dim myDynamicArray() As Strings 'Creates an Array of an unknown number of strings
ReDim myDynamicArray(5) 'This resets the array to 6 elements
```

Note that using the `ReDim` keyword will wipe out any previous content of your Array. To prevent this, you can use the `Preserve` keyword after `ReDim`:

```
Dim myDynamicArray(5) As String
myDynamicArray(0) = "Something I want to keep"
```

```
ReDim Preserve myDynamicArray(8) 'Expand the size to up to 9 strings
Debug.Print myDynamicArray(0) ' still prints the element
```

Section 18.5: Use of Split to create an array from a string

Split Function

returns a zero-based, one dimensional array containing a specified number of substrings.

Syntax

Split(expression [, **delimiter** [, **limit** [, **compare**]]])

Part	Description
expression	Required. String expression containing substrings and delimiters. If *expression* is a zero-length string("" or vbNullString), **Split** returns an empty array containing no elements and no data. In this case, the returned array will have a LBound of 0 and a UBound of -1.
delimiter	Optional. String character used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If ***delimiter*** is a zero-length string, a single-element array containing the entire ***expression*** string is returned.
limit	Optional. Number of substrings to be returned; -1 indicates that all substrings are returned.
compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values.

Settings

The ***compare*** argument can have the following values:

Constant	Value	Description
Description	-1	Performs a comparison using the setting of the **Option Compare** statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

Example

In this example it is demonstrated how Split works by showing several styles. The comments will show the result set for each of the different performed Split options. Finally it is demonstrated how to loop over the returned string array.

```
Sub Test

    Dim textArray() as String

    textArray = Split("Tech on the Net")
    'Result: {"Tech", "on", "the", "Net"}

    textArray = Split("172.23.56.4", ".")
    'Result: {"172", "23", "56", "4"}

    textArray = Split("A;B;C;D", ";")
    'Result: {"A", "B", "C", "D"}

    textArray = Split("A;B;C;D", ";", 1)
    'Result: {"A;B;C;D"}
```

```
    textArray = Split("A;B;C;D", ";", 2)
    'Result: {"A", "B;C;D"}

    textArray = Split("A;B;C;D", ";", 3)
    'Result: {"A", "B", "C;D"}

    textArray = Split("A;B;C;D", ";", 4)
    'Result: {"A", "B", "C", "D"}

    'You can iterate over the created array
    Dim counter As Long

    For counter = LBound(textArray) To UBound(textArray)
        Debug.Print textArray(counter)
    Next
End Sub
```

Section 18.6: Iterating elements of an array

For...Next

Using the iterator variable as the index number is the fastest way to iterate the elements of an array:

```
Dim items As Variant
items = Array(0, 1, 2, 3)

Dim index As Integer
For index = LBound(items) To UBound(items)
    'assumes value can be implicitly converted to a String:
    Debug.Print items(index)
Next
```

Nested loops can be used to iterate multi-dimensional arrays:

```
Dim items(0 To 1, 0 To 1) As Integer
items(0, 0) = 0
items(0, 1) = 1
items(1, 0) = 2
items(1, 1) = 3

Dim outer As Integer
Dim inner As Integer
For outer = LBound(items, 1) To UBound(items, 1)
    For inner = LBound(items, 2) To UBound(items, 2)
        'assumes value can be implicitly converted to a String:
        Debug.Print items(outer, inner)
    Next
Next
```

For Each...Next

A `For Each...Next` loop can also be used to iterate arrays, if performance doesn't matter:

```
Dim items As Variant
items = Array(0, 1, 2, 3)

Dim item As Variant 'must be variant
For Each item In items
    'assumes value can be implicitly converted to a String:
```

```
    Debug.Print item
Next
```

A `For Each` loop will iterate all dimensions from outer to inner (the same order as the elements are laid out in memory), so there is no need for nested loops:

```
Dim items(0 To 1, 0 To 1) As Integer
items(0, 0) = 0
items(1, 0) = 1
items(0, 1) = 2
items(1, 1) = 3

Dim item As Variant 'must be Variant
For Each item In items
    'assumes value can be implicitly converted to a String:
    Debug.Print item
Next
```

Note that `For Each` loops are best used to iterate `Collection` objects, if performance matters.

All 4 snippets above produce the same output:

```
0
1
2
3
```