TwoDimensionalArrays – Walkthrough

Multiple-Subscripted Arrays

So far we have studied single-subscripted (or one-dimensional) arrays - i.e., those that contain single lists of values

In this section, we introduce multiple-subscripted (often called multidimensional) arrays

Such arrays require two or more subscripts to identify particular elements

Arrays that require two subscripts to identify a particular element commonly are called double-subscripted arrays

We concentrate on double-subscripted arrays (often called two-dimensional arrays)

There are two types of multiple-subscripted arrays—rectangular and jagged

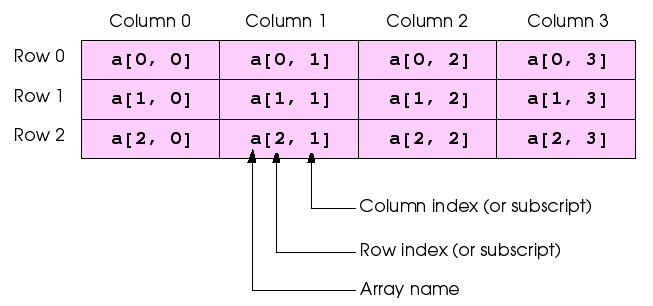
Rectangular arrays with two subscripts often represent tables of values consisting of information arranged in rows and columns, where each row is the same size, and each column is the same size

To identify a particular table element, we must specify the two subscripts—by convention, the first identifies the element’s row and the second identifies the element’s column

Multiple-subscripted arrays can have more than two subscripts

The diagram below illustrates a double-subscripted array, a, containing three rows and four columns (i.e., a 3-by-4 array)

An array with m rows and n columns is called an m-by-n array



Every element in array a is identified in the diagram above by an element name of the form a[ i , j ], in which a is the name of the array, and i and j are the subscripts that uniquely identify the row and column of each element in a

Notice that the names of the elements in the first row all have a first subscript of 0; the names of the elements in the fourth column all have a second subscript of 3

Multiple-subscripted arrays can be initialized in declarations like single-subscripted arrays

A double-subscripted array b with two rows and two columns could be declared and initialized with

**int[,] b = new int[ 2, 2 ];**

**b[ 0, 0 ] = 1;**

**b[ 0, 1 ] = 2;**

**b[ 1, 0 ] = 3;**

**b[ 1, 1 ] = 4;**

or this can be written on one line using an **initializer** list as shown below:

**int[,] b = { { 1, 2 }, { 3, 4 } };**

The values are grouped by row in braces

Thus,

1 and 2 initialize b[ 0 , 0 ] and b[ 0 , 1 ], and 3 and 4 initialize b[ 1 , 0 ] and b[ 1 , 1 ]

The compiler determines the number of rows by counting the number of sub-initializer lists (represented by sets of braces) in the main initializer list

The compiler determines the number of columns in each row by counting the number of initializer values in the sub-initializer list for that row

Method GetLength returns the length of a particular array dimension

In the preceding example, b.GetLength( 0 ) returns the length of the zeroth dimension of b, which is 2

Jagged arrays are maintained as arrays of arrays

Unlike in rectangular arrays, the arrays that compose jagged arrays can be of different lengths

The declaration

**int[][] c = new int[ 2 ][]; // allocate rows**

**// allocate and initialize elements in row 0**

**c[ 0 ] = new int[] { 1, 2 };**

**// allocate and initialize elements in row 0**

**c[ 1 ] = new int[] { 3, 4, 5 };**

creates integer array c with row 0 (which is an array itself) containing two elements (1 and 2), and row 1 containing three elements (3, 4 and 5)

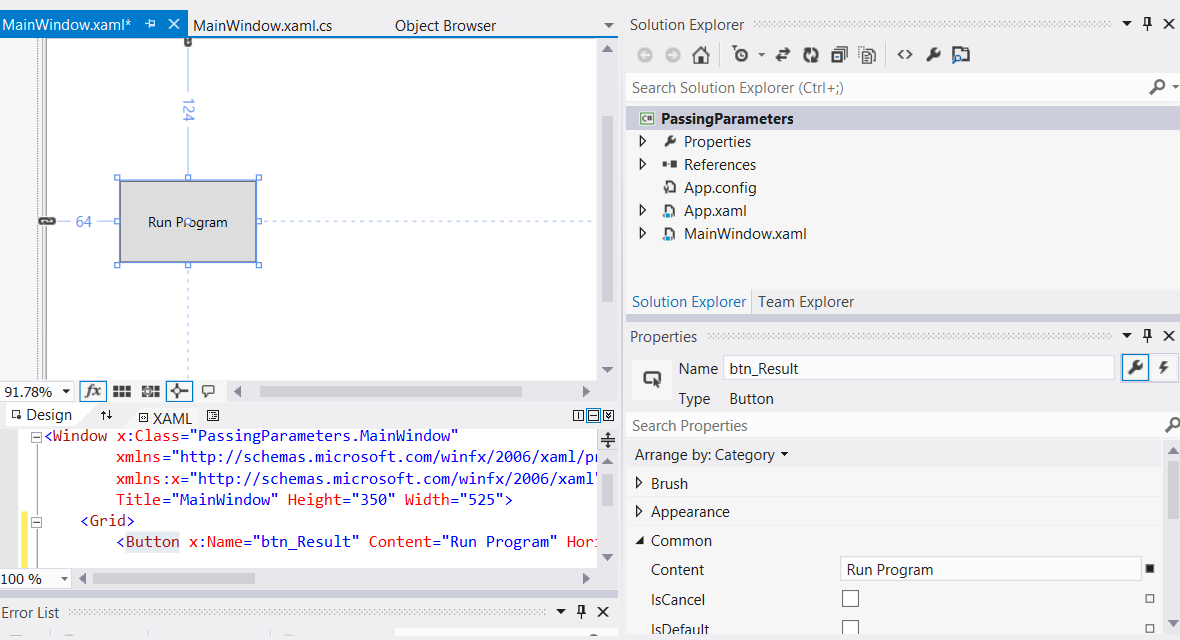
The Length property of each sub-array can be used to determine the size of each column

For the jagged array c, the size of the zeroth column is c[ 0 ].Length, which is 2

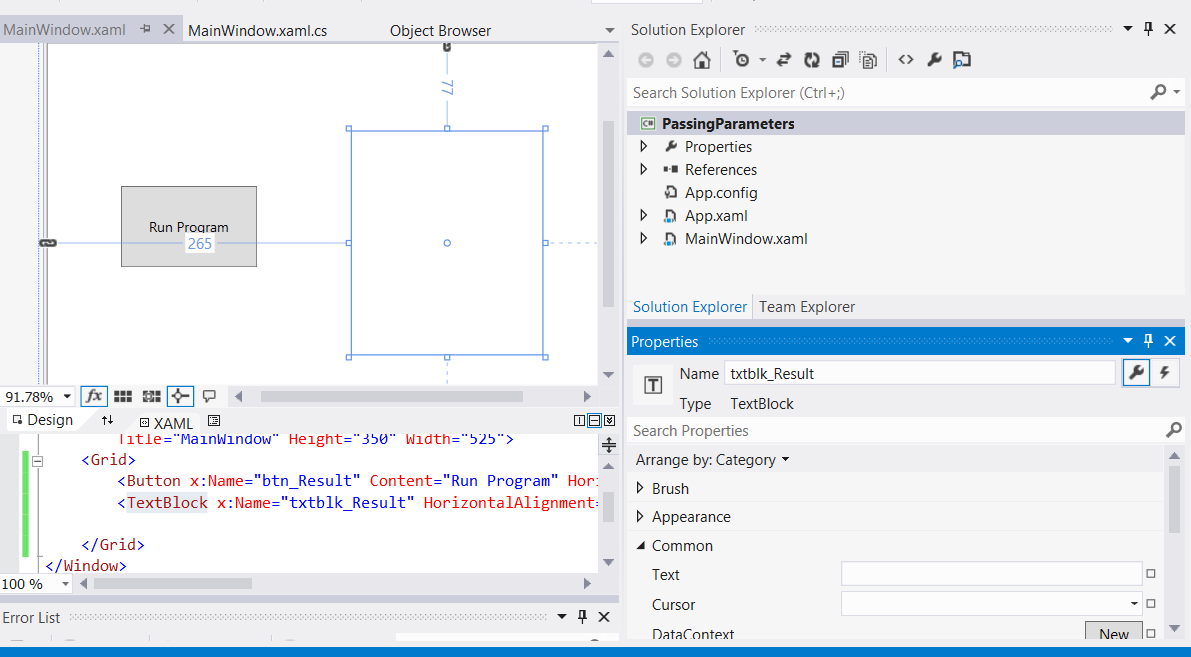
We are going to build a basic application which has a single button, which when clicked will:

* Create a 2D array and a jagged array
* Use nested for loops to traverse each array

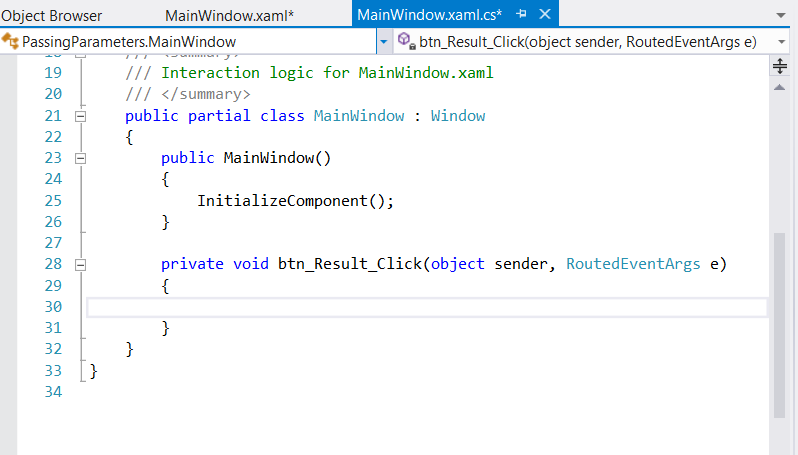
1. Create a new WPF project
2. Drag across a button from the toolbox
   1. set the “Name” property to btn\_Result
   2. set the “Content” property to “Run Program”



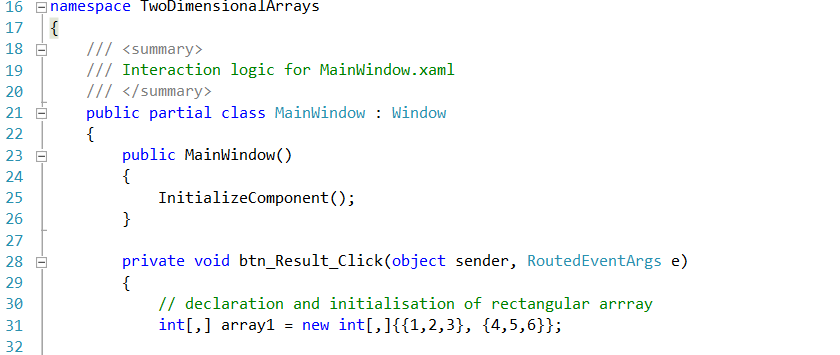
1. Drag across a text block and from the toolbox
   1. set the “Name” property to txtblk\_Result
   2. set the “Text” property to blank

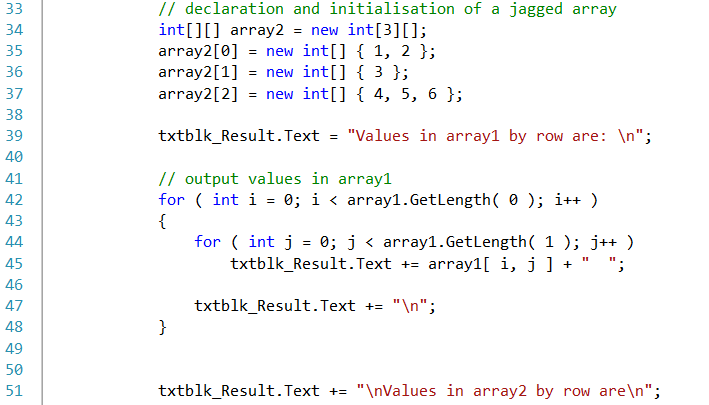


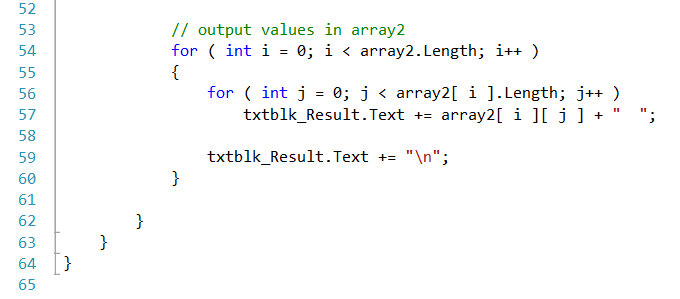
1. Double-click on the “Run Program” button to automatically create an event handler in the MainWindow.xaml.cs file, which is a method called btn\_Result\_Click



1. We are now going to write the code for this event handler. The final code can be viewed in the screenshot below followed by a full explanation







Initializing multidimensional arrays - code walkthrough

The declaration of array1 (line 31) provides six initializers in two sublists

The first sublist initializes the first row of the array to the values 1, 2 and 3

The second sublist initializes the second row of the array to the values 4, 5 and 6

The declaration of array2 (line 34) creates a jagged array of 3 arrays (specified by the 3 in the first set of square brackets)

Lines 35 - 37 initialize each subarray so that the first subarray contains the values 1 and 2, the second contains the value 3 and the last contains the values 4, 5 and 6

The for structure on lines 42 - 48 appends the elements of array1 to string output

Note the use of a nested for structure to output the rows of each double-subscripted array

In the nested for structures for array1, we use method GetLength to determine the number of elements in each dimension of the array

Line 42 determines the number of rows in the array by invoking array1.GetLength( 0 ), and line 44 determines the number of columns in the array by invoking array1.GetLength( 1 )

Arrays with additional dimensions would require more deeply nested for loops to process

The nested for structures on lines 54 - 60 output the elements of jagged array array2

Recall that a jagged array is essentially an array that contains additional arrays as its elements

Line 54 uses the Length property of array2 to determine the number of rows in the jagged array

Line 56 determines the Length of each subarray with the expression

array2[ i ].Length

Many common array manipulations use for repetition structures

For the remainder of this section, we will focus on manipulations of jagged arrays

Imagine a jagged array a, which contains 3 rows, or arrays

The following for structure sets all the elements in the third row of array a to zero:

**For ( int col = 0; col < a[ 2 ].Length; col++ )**

**a[ 2 ][ col ] = 0;**

We specified the third row; therefore, we know that the first subscript is always 2 (0 is the first row and 1 is the second row)

The for loop varies only the second subscript (i.e., the column subscript)

Notice the use of a[ 2 ].Length in the for structure’s conditional expression

This statement demonstrates that each row of a is an array in itself, and therefore the program can access a typical array’s properties, such as Length

Assuming the length of array a[ 2 ] is 4, the preceding for structure is equivalent to the assignment statements

**a[ 2 ][ 0 ] = 0;**

**a[ 2 ][ 1 ] = 0;**

**a[ 2 ][ 2 ] = 0;**

**a[ 2 ][ 3 ] = 0;**

The following nested for structure determines the total of all the elements in array a

We use a.Length in the conditional expression of the outer for structure to determine the number of rows in a, in this case, 3

**Int total = 0;**

**For ( int row = 0; row < a.Length; row++ )**

**For (int col = 0; col < a[ row ].Length; col++)**

**Total += a[ row ][ col ];**

The for structure totals the elements of the array one row at a time

The outer for structure begins by setting the row subscript to 0, so the elements of the first row may be totalled by the inner for structure

Then the outer for structure increments row to 1, so the second row can be totalled

Finally, the outer for structure increments row to 2, so the third row can be totalled

The result can be displayed when the nested for structure terminates