ArrayReference – Walkthrough

Passing Arrays by Value and by Reference

In C#, a variable that “stores” an object, such as an array, does not actually store the object itself

Instead, such a variable stores a reference to the object (i.e., the location in the computer’s memory where the object itself is stored)

The distinction between reference variables and primitive data type variables raises some subtle issues that programmers must understand to create secure, stable programs

When a program passes an argument to a method, the called method receives a copy of that argument’s value

Changes to the local copy do not affect the original variable that the program passed to the method

If the argument is of a reference type, the method makes a local copy of the reference itself, not a copy of the actual object to which the reference refers

The local copy of the reference also refers to the original object in memory

Thus, reference types are always passed by reference, which means that changes to those objects in called methods affect the original objects in memory

C# also allows methods to pass references with keyword **ref**

This is a subtle capability, which, if misused, can lead to problems

For instance, when a reference-type object like an array is explicitly passed with **ref**, the called method actually gains control over the **passed reference** itself, allowing the called method to replace the original reference in the caller with a different object or even with null

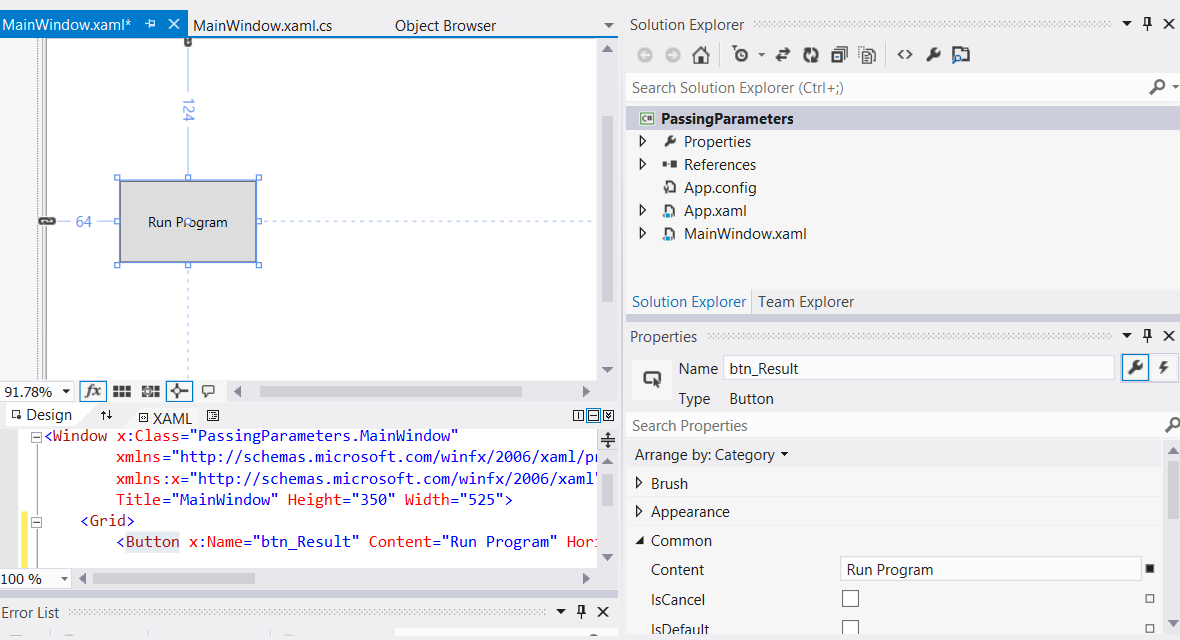
Such behaviour can lead to unpredictable effects, which can be disastrous in mission-critical applications

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

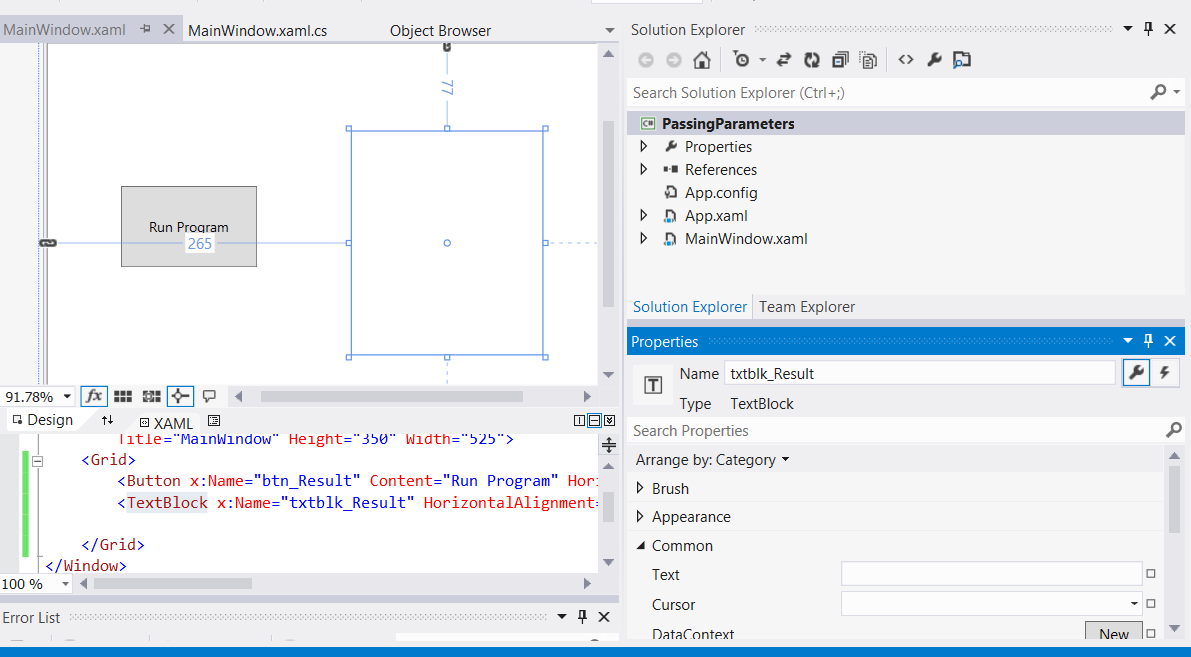
* Pass an array by normal value reference (get copy of reference), update the values and attempt to modify the reference. The data will be updated but only the local reference will have a different object and original will remain unchanged.
* Pass an array by actual reference (refers to original), update the values and attempt to modify the reference. The data will be updated and the original reference will then be assigned a different object.

The aim of this tutorial is to understand that under normal circumstances arrays are passed by reference which takes a “copy” of the reference which points to the original object i.e. there will now be 2 references pointing to the same object, but there is also the ability to use the **ref** keyword which results in the original reference being accessible to the called method.

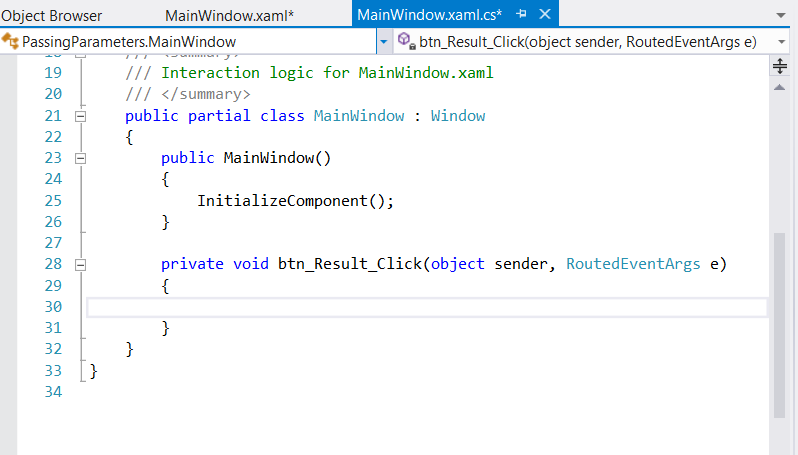
1. Create a new WPF project
2. Drag across a button from the toolbox
   1. set the “Name” property to btn\_Result
   2. set the “Content” property to “Run Program”



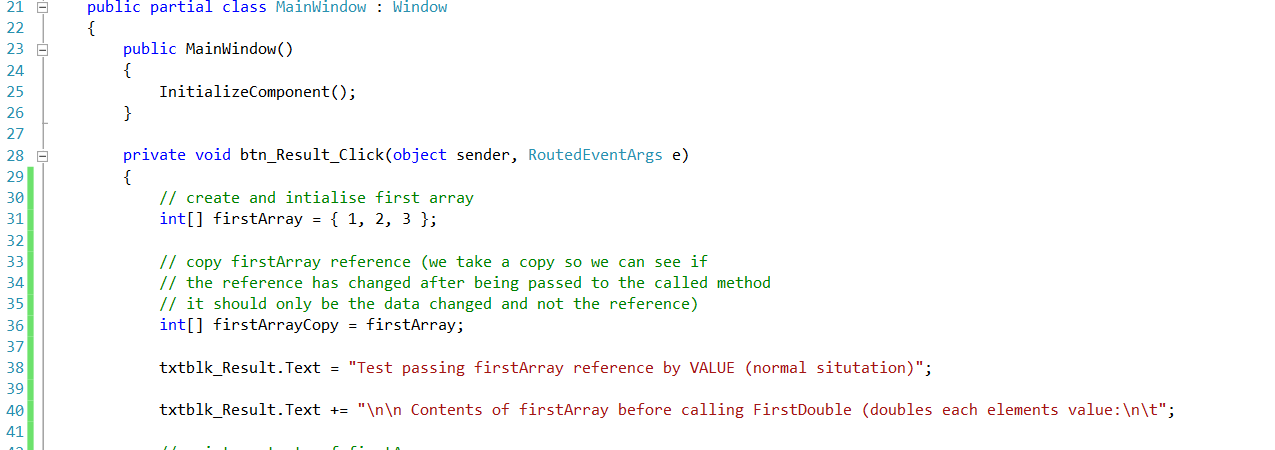
1. Drag across a text block and from the toolbox
   1. set the “Name” property to txtblk\_Result
   2. set the “Text” property to blank

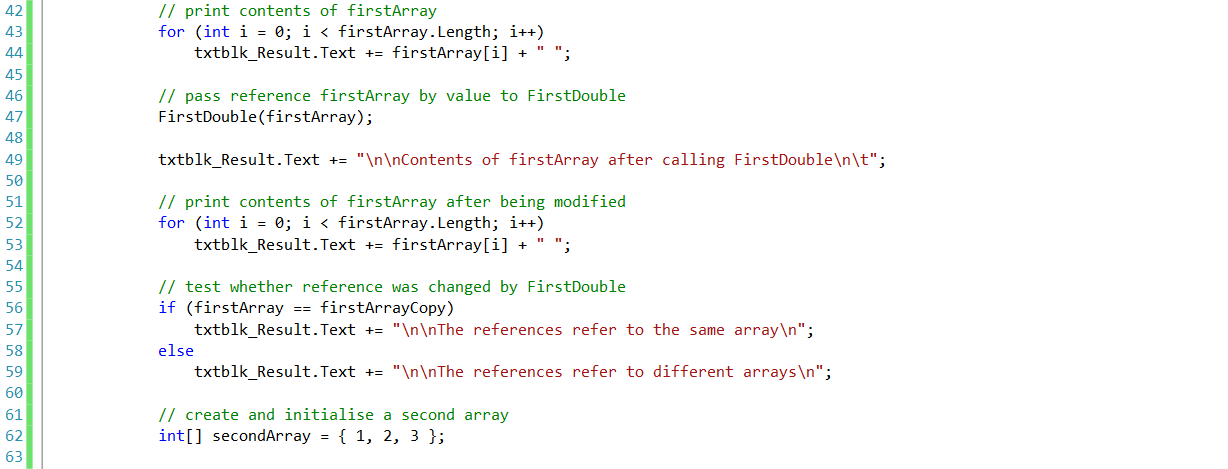


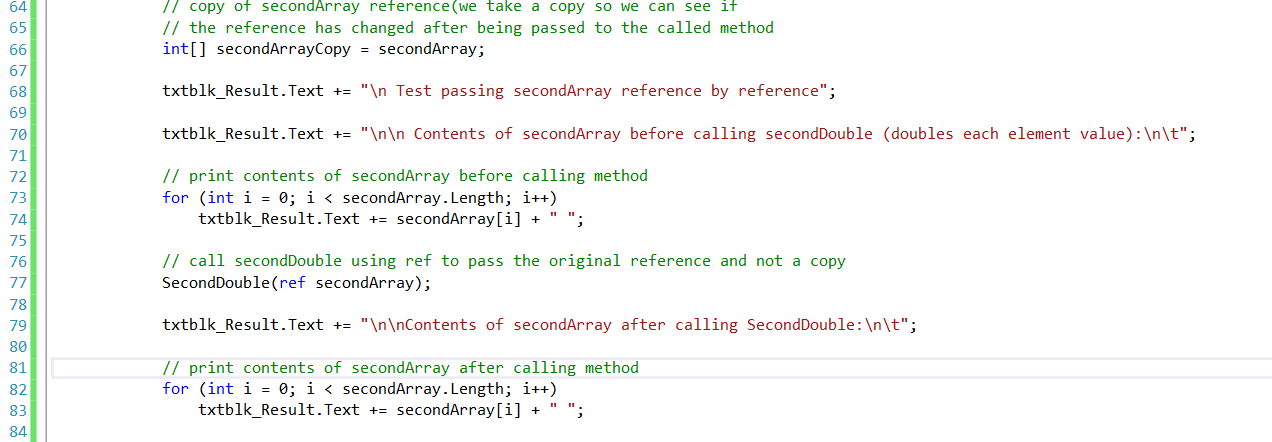
1. Double-click on the “Run Program” button to automatically create an event handler in the MainWindow.xaml.cs file, which is a method called btn\_Result\_Click

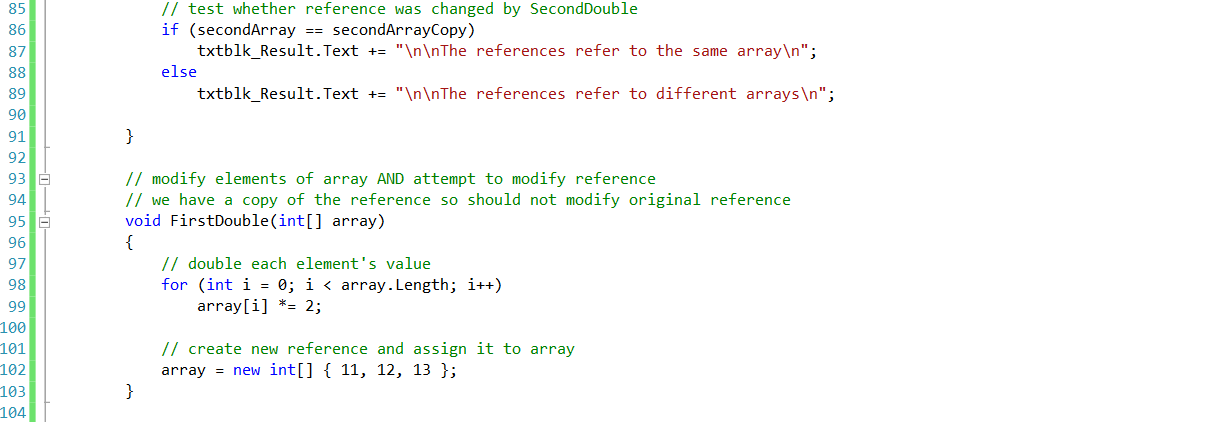


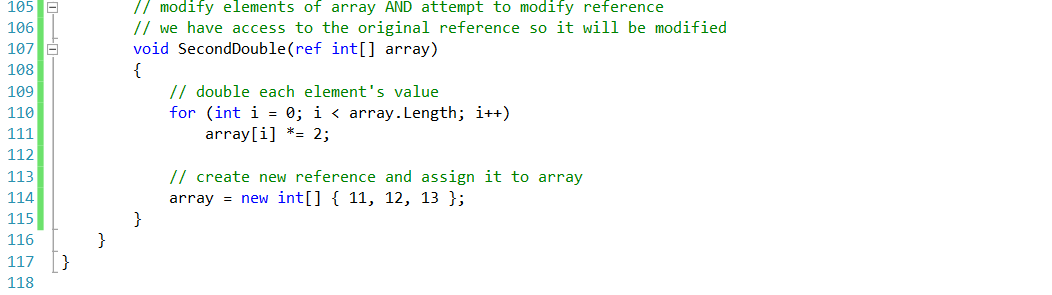
1. We are now going to write the code for this event handler and additional “helper” methods called “FirstDouble” and “SecondDouble”. The final code can be viewed in the screenshot below followed by a full explanation











Passing an array reference by value and by reference - code walkthrough

Lines 31 and 36 declare two integer array variables, firstArray and firstArrayCopy (we make the copy so we can determine whether reference firstArray gets overwritten)

Line 31 initializes firstArray with the values 1, 2 and 3

The assignment statement on line 36 copies reference firstArray to variable firstArrayCopy, causing these variables to reference the same array object in memory

The for structure on lines 43 - 44 prints the contents of firstArray before it is passed to method FirstDouble (line 47) so we can verify that this array is passed by reference (i.e., the called method indeed changes the array’s contents)

The for structure in method FirstDouble (lines 98 - 99) multiplies the values of all the elements in the array by 2

Line 102 allocates a new array containing the values 11, 12 and 13; the reference for this array then is assigned to parameter **array** (in an attempt to overwrite reference firstArray - this, of course, will not happen, because the reference was passed by value)

After method FirstDouble executes, the for structure on lines 52 - 53 prints the contents of firstArray, demonstrating that the values of the elements have been changed by the method (and confirming that in C# arrays are always passed by reference)

The if/else structure on lines 86 - 89 uses the == operator to compare references firstArray (which we just attempted to overwrite) and firstArrayCopy

The expression on line 86 evaluates to true if the operands to binary operator == indeed reference the same object

In this case, the object represented is the array allocated in line 31 - not the array allocated in method FirstDouble (line 102)

Lines 62 - 89 perform similar tests, using array variables secondArray and secondArrayCopy and method SecondDouble (lines 107–115)

Method SecondDouble performs the same operations as FirstDouble, but receives its array argument using keyword **ref**

In this case, the reference stored in secondArray after the method call is a reference to the array allocated on line 114 of SecondDouble, demonstrating that a reference passed with keyword **ref** can be modified by the called method so that the reference actually points to a different object, in this case an array allocated in procedure SecondDouble

The if/else structure in lines 86 - 89 demonstrates that secondArray and secondArrayCopy no longer refer to the same array