PassingParameters – Walkthrough

Value Types and Reference Types

We will now discuss passing arguments to methods by value and by reference

To understand this, we first need to make a distinction between types in C#

Data types are either value types or reference types

A variable of a value type contains data of that type

A variable of a reference type, in contrast, contains the address of the location in memory where the data are stored

Value types normally represent single pieces of data, such as int or bool values

Reference types, on the other hand, refer to objects, which can contain many individual pieces of data

C# includes built-in value types and reference types

The built-in value types are the integral types (sbyte, byte, char, short, ushort, int, uint, long and ulong), the floating-point types (float and double) and the types decimal and bool

The built-in reference types are string and object

Programmers also can create value types and reference types

The reference types that programmers can create are **classes**, **interfaces** and **delegates**

The table below lists the primitive data types, which are building blocks for more complicated types

Like its predecessor languages C and C++, C# requires all variables to have a type before they can be used in a program

For this reason, C# is referred to as a strongly typed language

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Size in bits** | **Values** | **Standard** |
| bool | 8 | true or false |  |
| char | 16 | '\u0000' to '\uFFFF' | (Unicode character set) |
| byte | 8 | 0 to 255 | (unsigned) |
| sbyte | 8 | –128 to +127 |  |
| short | 16 | –32,768 to +32,767 |  |
| ushort | 16 | 0 to 65,535 | (unsigned) |
| int | 32 | –2,147,483,648 to 2,147,483,647 |  |
| uint | 32 | 0 to 4,294,967,295 | (unsigned) |
| long | 64 | –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 |  |
| ulong | 64 | 0 to 18,446,744,073,709,551,615 | (unsigned) |
| decimal | 128 | 1.0 x 10-28 to 7.9 x 1028 |  |
| float | 32 | ±1.5 × 10-45 to ±3.4 × 1038 | (IEEE 754 floating point) |
| double | 64 | ±5.0 × 10-324 to ±1.7 × 10308 | (IEEE 754 floating point) |
| object |  |  |  |
| string |  |  | (Unicode character set) |

**Table of C# built-in data types**

In C and C++ programs, programmers frequently must write separate program versions to support different computer platforms because the primitive data types are not guaranteed to be identical from computer to computer

For example, an int value on one computer might occupy 16 bits (2 bytes) of memory, whereas an int value on another computer might occupy 32 bits (4 bytes) of memory

In C#, int values are always 32 bits (4 bytes)

Each data type in the table is listed with its size in bits (there are 8 bits to a byte) and its range of values

The designers of C# wanted code to be portable; therefore, they chose to use internationally recognized standards for both character formats (Unicode) and floating-point numbers (IEEE 754)

Passing Arguments: Pass-by-Value vs. Pass-by-Reference

Two ways to pass arguments to methods in many programming languages are pass-by-value and pass-by-reference

When an argument is passed by value, the called method receives a copy of the argument’s value

When an argument is passed using pass-by-reference, the caller gives the method the ability to access and modify the caller’s original data directly

Pass-by-reference can improve performance because it eliminates the overhead of copying large data items such as objects; however, pass-by-reference can weaken security because the called method can modify the caller’s data

To pass an object reference into a method, simply specify the reference name in the method call

Then, in the method body, reference the object using the parameter name

This refers to the original object in memory, which allows the called method to access the original object directly

In the previous section, we discussed the difference between value types and reference types

At this point, understand one of the major differences between the two data types - value-type variables are passed to methods by value, whereas reference-type variables are passed to methods by reference

What if the programmer would like to pass a value type by reference ?

To do this, C# provides the **ref** and **out** keywords

The ref keyword specifies that a value-type argument should be passed by reference, which enables the called method to modify the original variable

This keyword is used for variables that already have been initialized

The out keyword specifies an output parameter, which is an argument to which the called method will assign a value

Normally, when a method receives an uninitialized value, the compiler generates an error

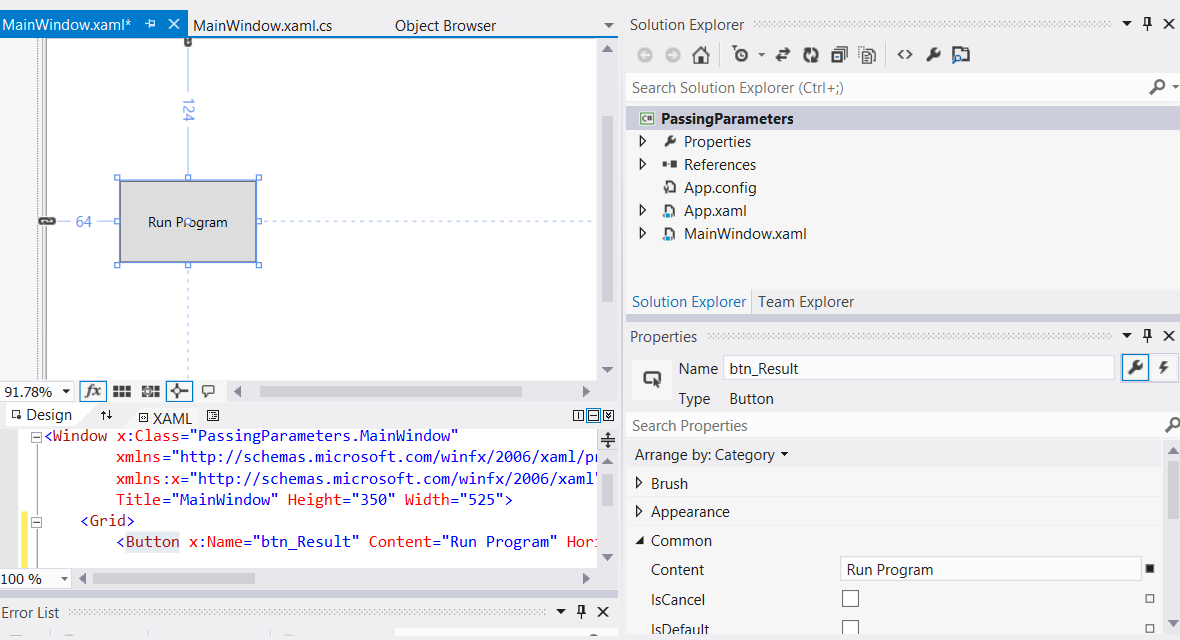
Preceding the parameter with keyword out specifies that the called method will initialize the variable and prevents the compiler from generating an error message for the uninitialized variable

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

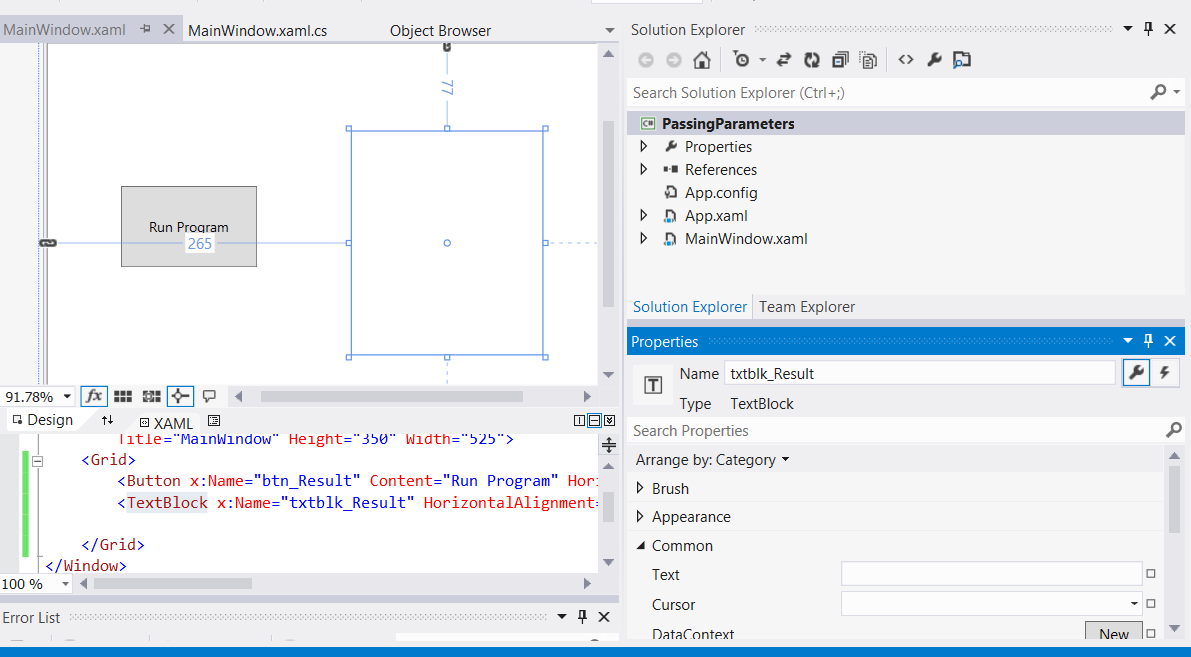
* display the original values of 2 integer variables, followed by
* their new values having being passed **by reference** to a method that squares the value, followed by
* their new values having being **passed by value** to a method that squares the value.

The aim of this tutorial is to show how to pass a value-type by reference and to point out that it updates the original variable, whereas passing by value makes a copy of the original and therefore leave the original value unchanged.

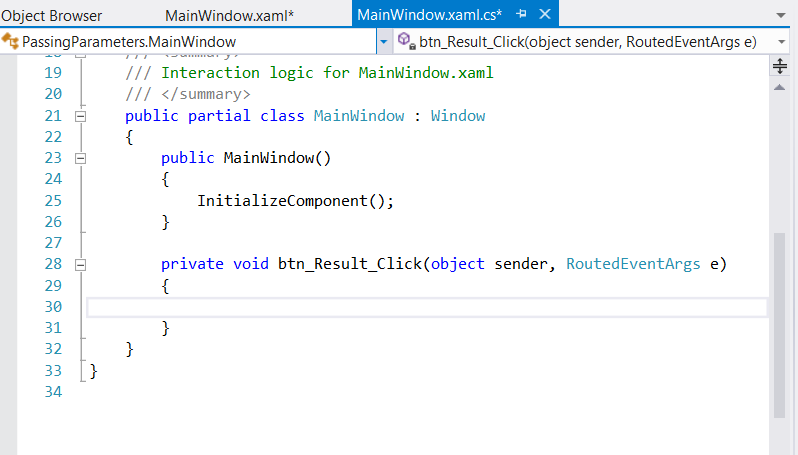
1. Create a new WPF project
2. Drag across a button from the toolbox
   1. set the “Name” property to btn\_Result
   2. set the “Content” property to “Run Program”



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

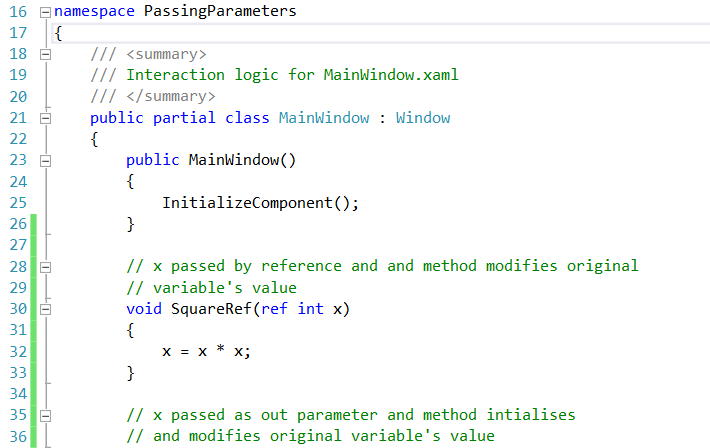


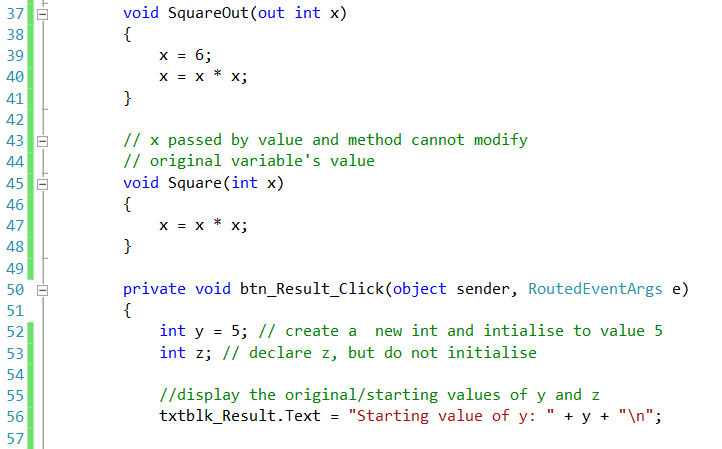
1. Double-click on the “Run Program” button to automatically create an event handler in the MainWindow.xaml.cs file, which is a method called btn\_Result\_Click

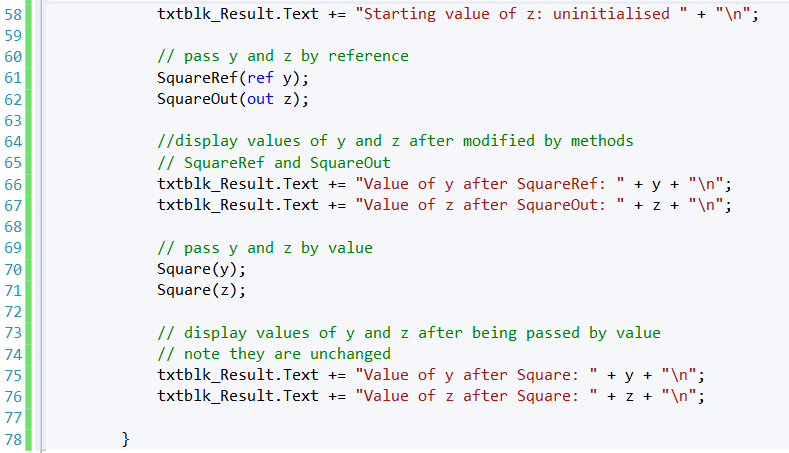


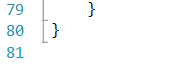
1. We are now going to write the code for this event handler and additional “helper” methods called “Square”, “SquareRef” and “SquareOut”. The final code can be viewed in the screenshot below followed by a full explanation

The code below demonstrates using the ref and out keywords to manipulate integer values:









This program contains three methods to calculate the square of an integer

The first method, SquareRef (lines 30 - 33), multiplies its argument x by itself and assigns the new value to x

SquareRef receives its argument as a **ref** int, specifying that x is an integer that is passed by reference to the method

As a result, the assignment at line 32 modifies the original argument’s value, rather than a copy of that value

The second method, SquareOut (lines 37 - 41), does the same thing, but initializes x to 6 on line 39

SquareOut receives its argument as an **out** int, which indicates that x is an integer variable that the caller passes to method SquareOut by reference and that SquareOut can assign a new value to this variable

The final method, Square (lines 45 - 48), simply takes x as a **value**-type integer argument and squares its value

Method btn\_Result\_Click (lines 50 - 78) is an event handler that invokes methods SquareRef, SquareOut and Square when the user clicks the Run Program button

This method begins by initializing y to 5 and declaring (but not initializing) z

Lines 61 - 62 call methods SquareRef and SquareOut

Notice the syntax used for passing y and z - in each case, we precede the argument either with **ref** or with **out**

The output displays the values of y and z after the function calls

Notice that y has been changed to 25 and z has been set to 36

Finally, on lines 70 - 71 we call method Square

Arguments y and z both are passed by value—only **copies of their values** are passed to the method

As a result, the values of y and z remain 25 and 36, respectively