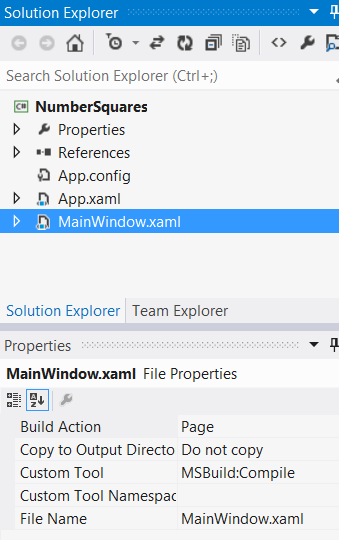
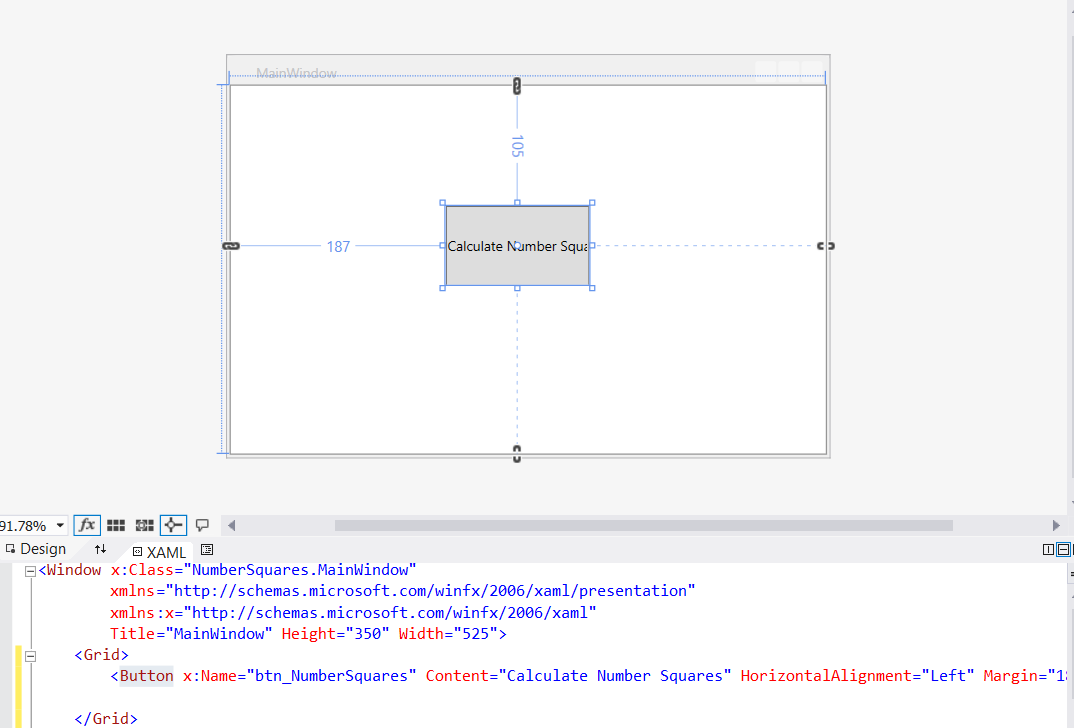
SquareInt – Walkthrough

Using programmer-defined method Square

To build this application, we will use the Windows Presentation Foundation which is a graphical subsystem by Microsoft for rendering user interfaces in Windows-based applications. When a new WPF application is created by default we have the following key files in the project:

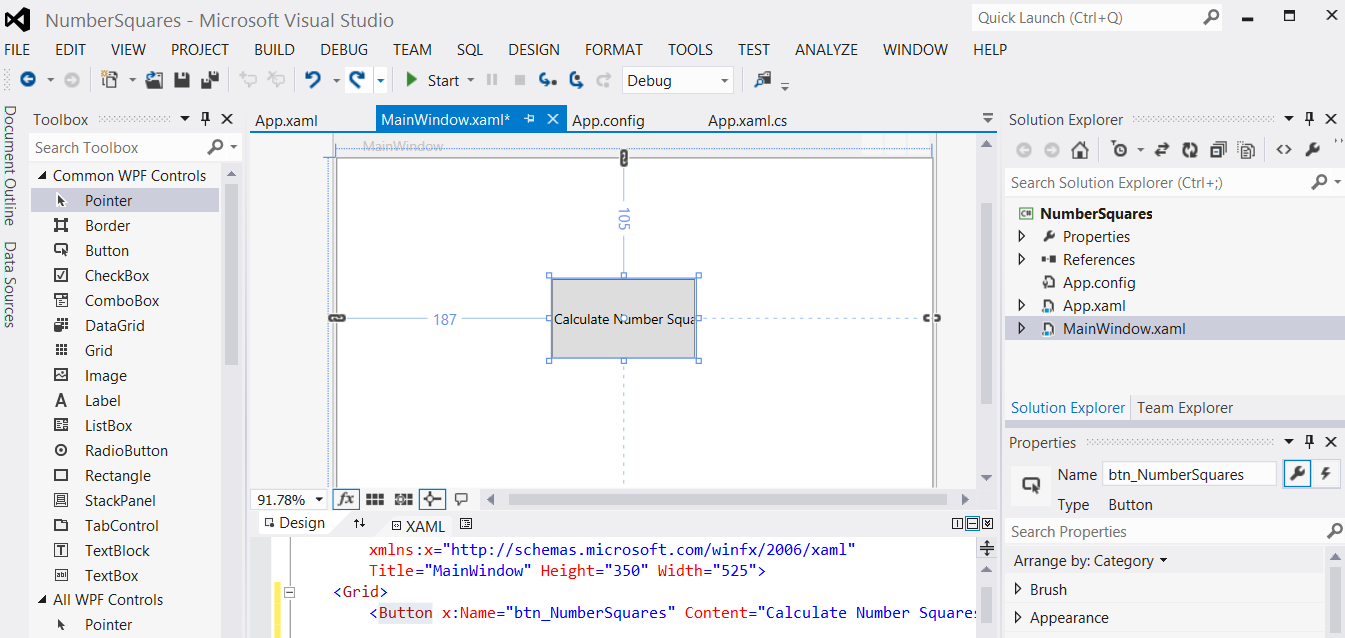
* **MainWindow.xaml**
  + by clicking on this file we get a split window. Typically the top half of the window contains the “designer” - this is a visual representation of the application window. The bottom half of the window shows the XAML code which describes the content of the application window.



XAML

DESIGNER

* + The content of the application window can be updated by dragging objects from the toolbox and dropping them in place on the “designer”. Each objects properties can be updated by highlighting the object on the designer and editing the properties in the “properties” window.



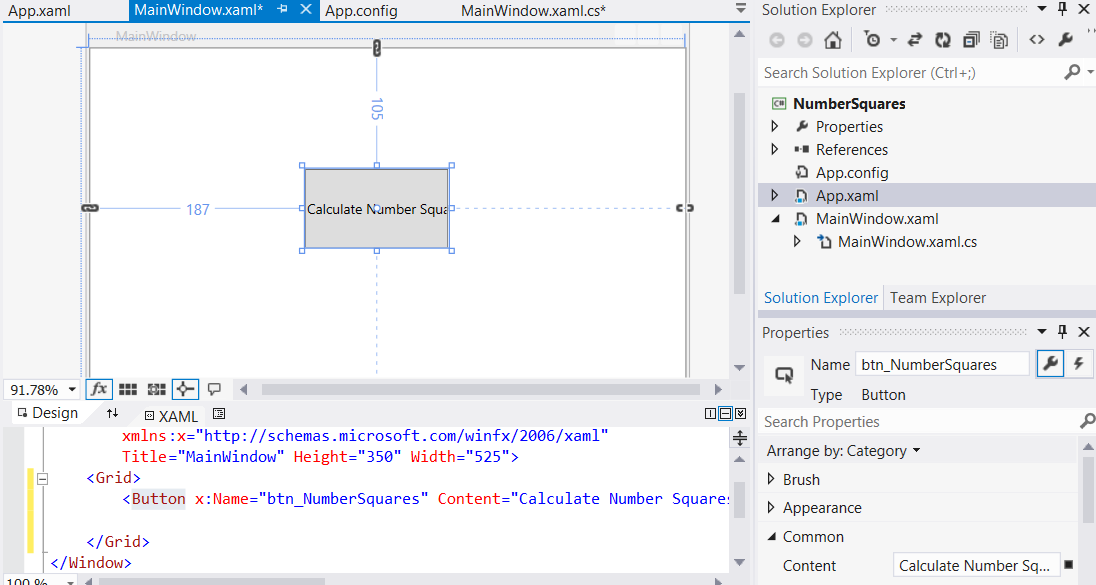
UPDATE PROPERTIES OF ANY OBJECT

DRAG OBJECTS FROM TOOLBOX ONTO DESIGNER AND XAML WILL AUTOMATICALLY UPDATE

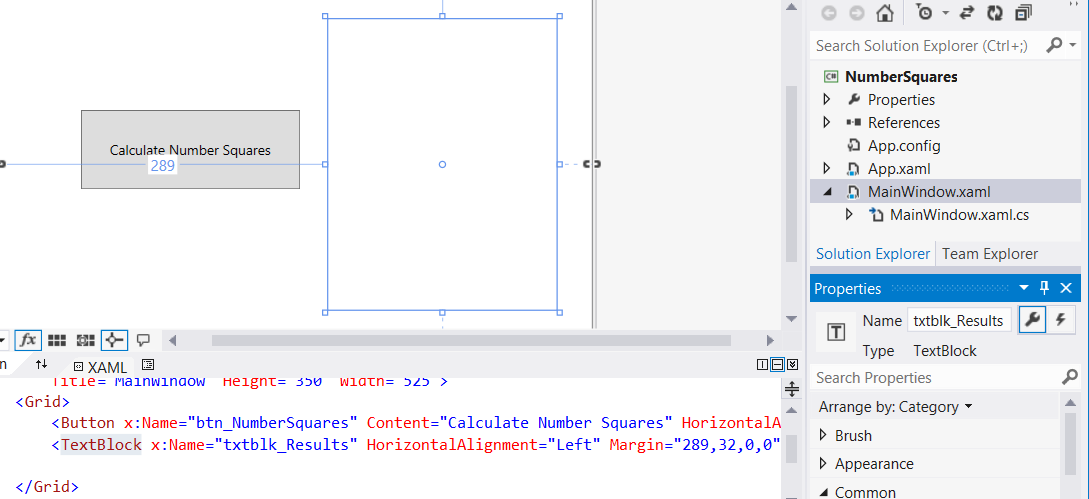
* + You can actually create the GUI in 3 different ways:
    - Drag and drop items onto the designer OR
    - Edit/write the XAML code directly OR
    - Create all the GUI elements in C#
* **MainWindow.cs**
  + This is the “code behind” file where we write all of the instructions that handle each event i.e. each user interaction with the GUI. We can also write any additional “helper” functions in here
* **App.xaml**
  + This file contains configuration information, the key information we are interested in is the StartupUri="MainWindow.xaml". The StartupUri defines what file will be used to load the application at start up, which by default is the MainWindow.xaml which contains a description of how to create the window and its contents.

We are going to build a basic application which has a single button, which when clicked will display the square of each number from 1 through to 10. The aim of this tutorial is to show how to write an “event handler” and a “user-defined helper function”.

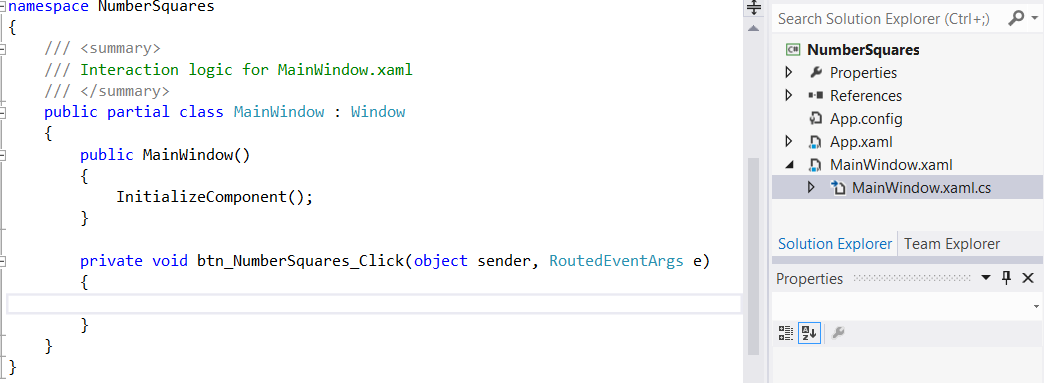
1. Drag a button object from the toolbox onto the designer
   1. Update the “Name” property to be btn\_NumberSquares
   2. Update the “Content” property to be Calculate Number Squares



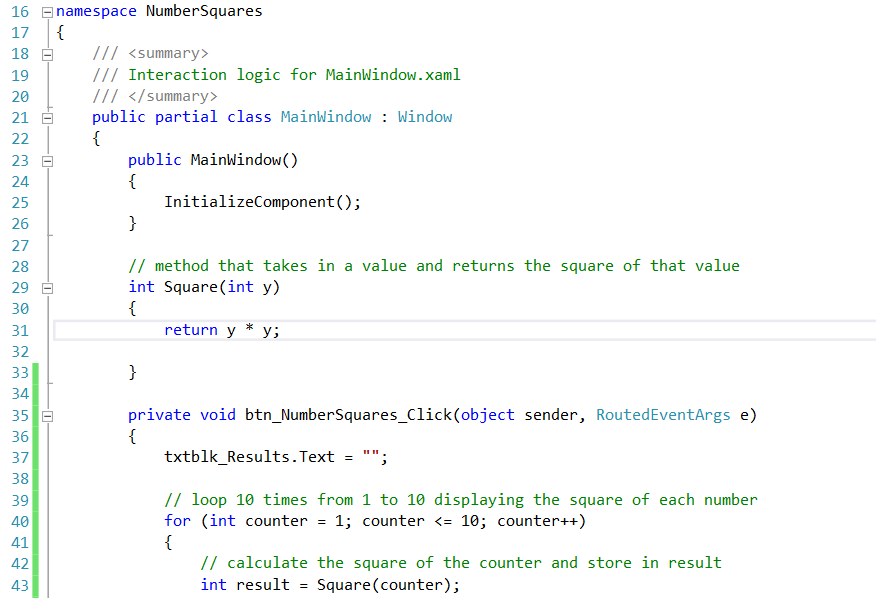
1. Drag across a TextBlock object from the toolbox onto the designer, this will be used to display the results of calculating the first ten squares.
   1. Update the “Name” property to be txtblk\_Results
   2. Update the “Content” property to be blank

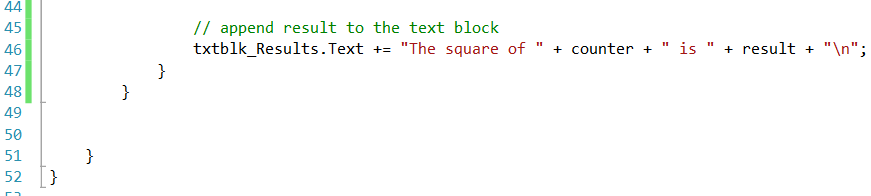


1. Double-click on the “Calculate Number Squares” button to automatically create an event handler in the MainWindow.xaml.cs file, which is a method called btn\_NumberSquares\_Click



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 program invokes a special method, known as an event handler, when the user clicks the “Calculate Number Squares” button.

An event handler is a method that performs some action in response to an event.

Events occur when certain actions take place in a graphical user interface, such as when the user clicks a button.

Using GUI component objects and events together allows programmers to create applications that interact with users in sophisticated ways.

In Visual Studio .NET’s Designer, double clicking on a GUI component object causes Visual Studio .NET to generate an empty event handler method in the code-behind file (MainWindow.xaml.cs).

The event handler method’s name defaults to the GUI component’s name, followed by an underscore and the name of the event:

private void btn\_NumberSquares\_Click(object sender, RoutedEventArgs e)

{

}

The programmer then can fill in the event handler method with code that performs a particular task.

Method btn\_NumberSquares\_Click (lines 35 - 48) is an event handler method for btn\_NumberSquares’s Click event

(i.e. the event that occurs when the user clicks the button)

When the user clicks the button, line 37 assigns the empty string ("") to txtblk\_Results’s Text property to ensure that the output does not scroll off the form if the user presses the Calculate Squares button more than once

Lines 40 - 47 repeatedly invoke method Square to calculate the squares of the integers from 1 to 10

Line 43 invokes the Square method and passes the variable counter as an argument

A text block called txtblk\_Results is created to display the program’s output

Every text block contains a string property called Text, which can be accessed using the dot operator (.)

We append the results of the square calculations to this text block’s Text property

Line 43 declares int variable result to store the result of each square calculation

Lines 40 - 47 contain a for repetition structure in which each iteration of the loop calculates the Square of the current value of control variable counter and stores the value in result

Lines 46 concatenates each result to the Text property of txtblk\_Results

At the end of the loop, the text block contains the results of squaring the values from 1 to 10

The program invokes method Square on line 43

The parentheses, (), after Square represent any inputs (arguments) we wish to pass to the method

At this point, the program makes a copy of the value of counter (the argument to the method call), and program control transfers to method Square

(defined at lines 29 - 33)

Method Square receives the copy of the value of counter in the parameter y

Then, Square calculates y\*y (line 31)

Method Square uses a return statement to return (i.e., give back) the result of the calculation to the statement that invoked Square (located in line 43)

Line 43 then assigns the returned value to variable result

Line 46 concatenates "The square of", the value of counter, "is", the value of result and a newline character to the end of txtblk\_Results Text property

The for repetition structure repeats this process 10 times

The definition of method Square (line 29) shows (inside the parentheses) that Square expects an integer parameter y

Parameter y is the name that holds the value passed to Square as an argument

The parameter name provides access to the argument value, so that code in the method body can use the value

Keyword int, which precedes the method name, indicates that method Square returns an integer result

The return statement in Square (line 31) passes the result of the calculation y\*y back to the calling statement

Note that the entire method definition appears inside the braces of class MainWindow

All methods must be defined inside a class definition

The format of a method definition is:

**return-value-type method-name( parameter-list )**

**{**

**declarations and statements**

**}**

The first line is sometimes referred to as the method header

The method-name is any valid identifier

The return-value-type is the data type of the result that the method returns to its caller

The return-value-type void indicates that a method does not return a value

Methods can return at most one value

The parameter-list (arguments) is a comma-separated list in which the method declares each parameter’s type and name

The method call must specify one argument for each parameter in the method definition and the arguments must appear in the same order as the parameters in the method definition

The arguments also must be compatible with the parameter’s type

For example, a parameter of type double could receive values of 7.35, 22 or –.03546, but not "hello" because a double variable cannot contain a string

If a method does not receive any values, the parameter list is empty (i.e., the method name is followed by an empty set of parentheses)

Each parameter in a method’s parameter list must have a data type; otherwise, a syntax error occurs

The declarations and statements within braces form the method body

The method body is also referred to as a block

A block is a set of declarations and statements enclosed in curly braces

Variables can be declared in any block, and blocks can be nested

There are three ways to return control to the point at which a method was invoked

If the method does not return a result (i.e., the method has a void return type), control returns when the program reaches the method-ending right brace or when the statement

**return;**

executes

If the method does return a result, the statement

**return expression;**

returns the value of expression to the caller

When a return statement executes, control returns immediately to the point at which the method was invoked