Windows Forms

An introductory tutorial

IES San Vicente

Table of contents

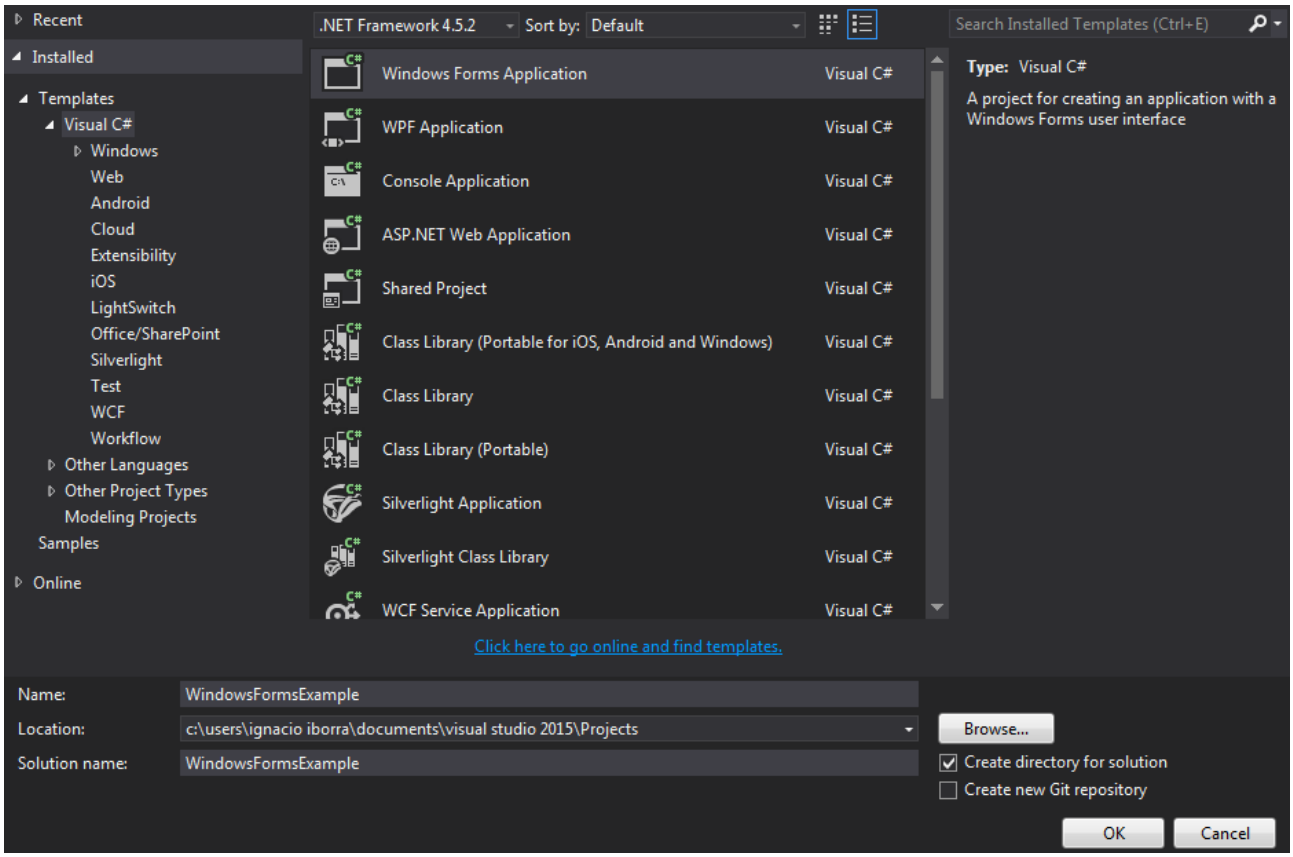
1. Introduction

Now that you have learnt the basics of C# programming language (control structures, arrays, collections, input/output, object oriented programming...), we are going one step further. We are not going to rely on a console application (sometimes they are hard to use), and we are going to see how to create graphical (window based) applications.

Graphical programming in C# consists in developing *Graphical User Interfaces* (GUI). These applications are based on forms, this is, windows that contain a set of controls (menus, buttons, text fields, lists...). In order to develop these applications under Visual Studio, we are going to use **Windows Forms**, a library integrated with Visual Studio for this purpose.

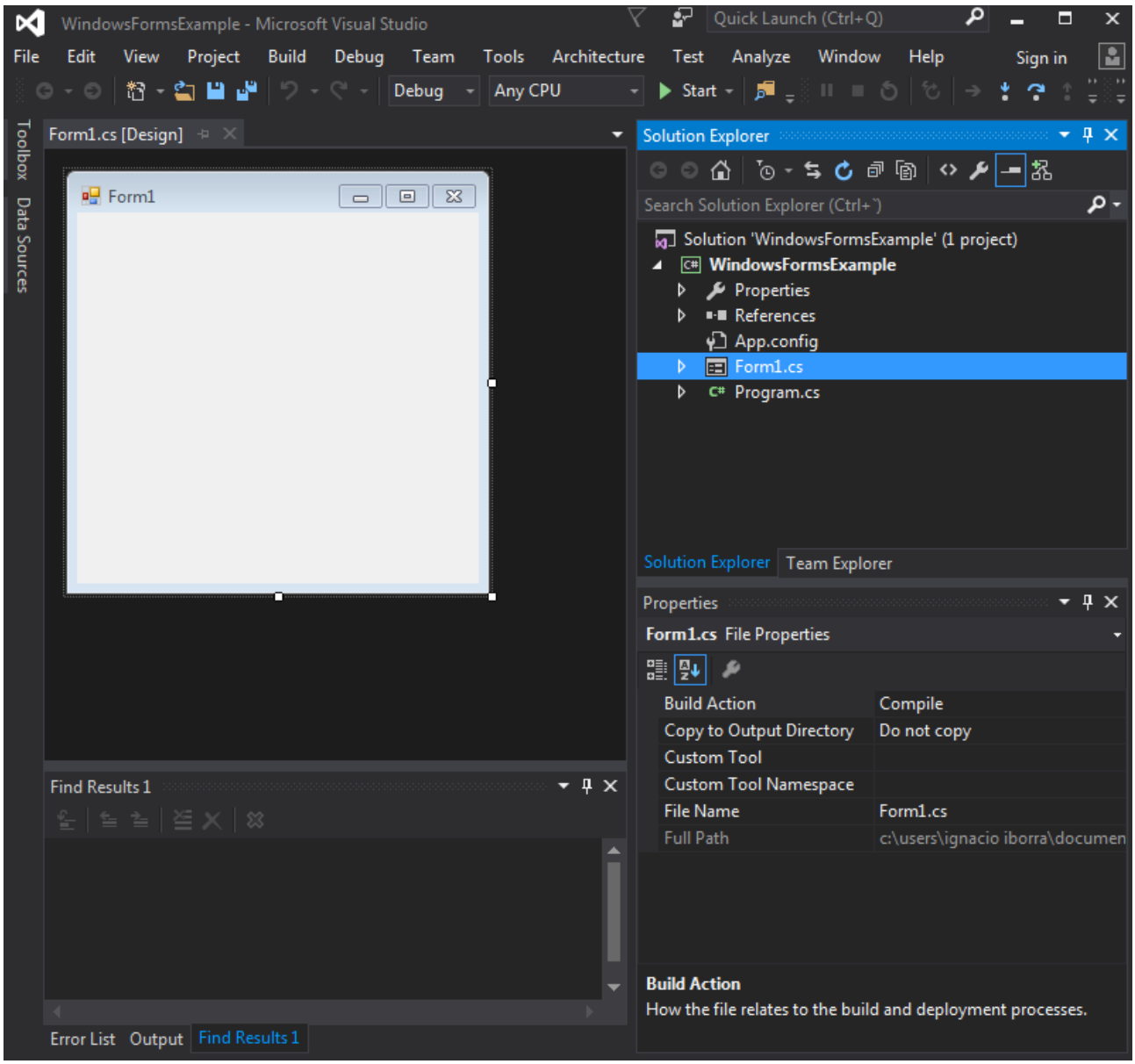
1.1. Creating the project

To create a Windows application project, we just need to go to *File* > *New project* menu, and choose *Windows Forms Application*.



When we create the project, a main program will be created (called *Program.cs*), along with a main form controlled by *Form1.cs* class (although we can rename any class in Visual Studio, just right clicking on it and choosing the *rename* option).

In the main work area (on the center-left) we can see this form, and we can edit it with a visual interface, as we are going to see right now.

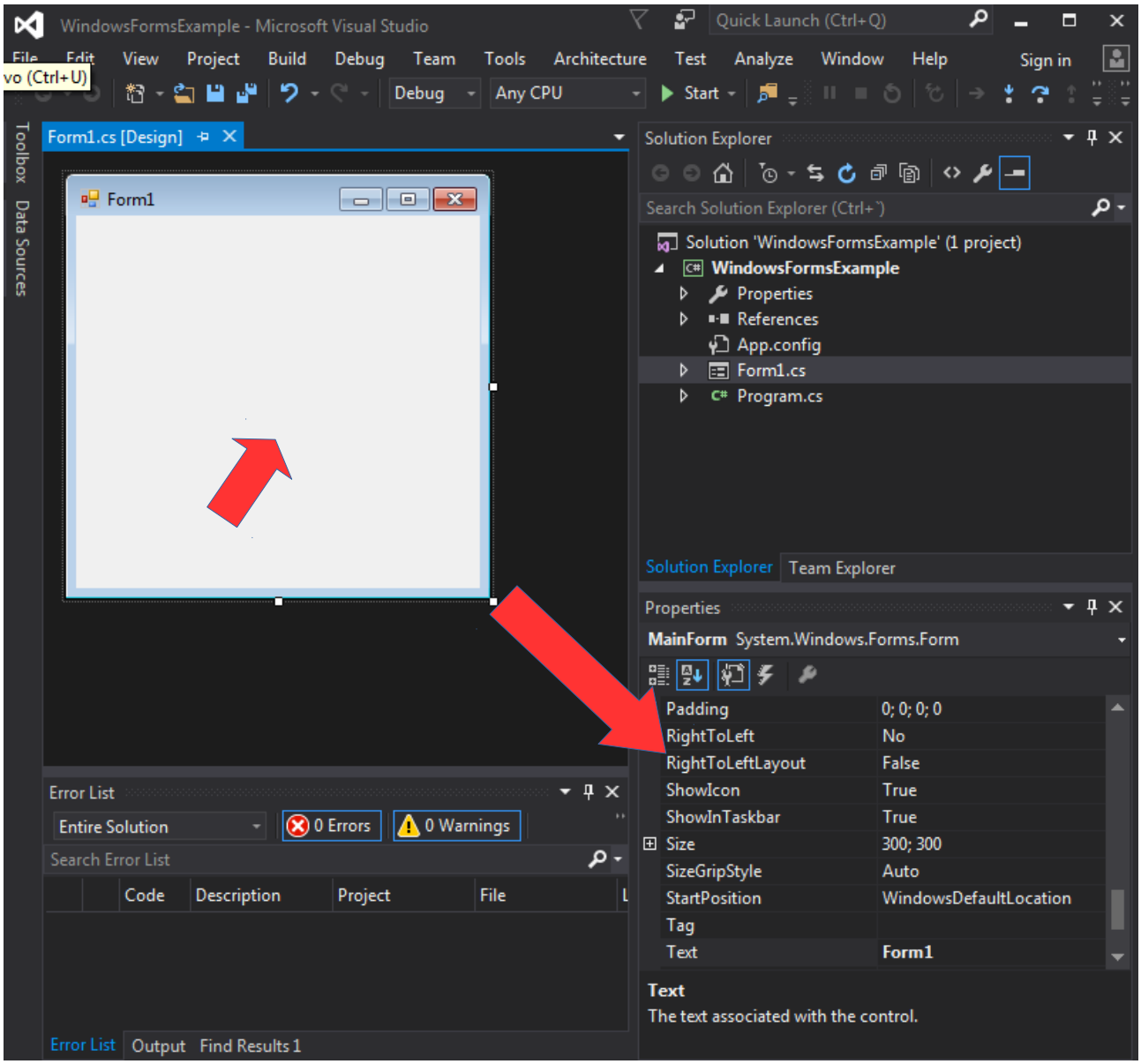

WindowsFormsExample - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Sign in
Debug
Any CPU
Start
Toolbox
Data Sources
Form1.cs [Design]
Form1
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'WindowsFormsExample' (1 project)
WindowsFormsExample
Properties
References
App.config
Form1.cs
Program.cs
Solution Explorer
Team Explorer
Properties
Form1.cs File Properties
Build Action
Compile
Copy to Output Directory
Do not copy
Custom Tool
Custom Tool Namespace
File Name
Form1.cs
Full Path
c:\users\ignacio iborra\documen
Build Action
How the file relates to the build and deployment processes.
Find Results 1
Error List
Output
Find Results 1

2. Controls and properties

Once we have our main form created, we can edit it, by adding controls (buttons, labels, menus...), and changing some of their properties (colors, font types...).

2.1. Form properties

If we left click on the main form in the work area, we can see its main properties in the lower right section:



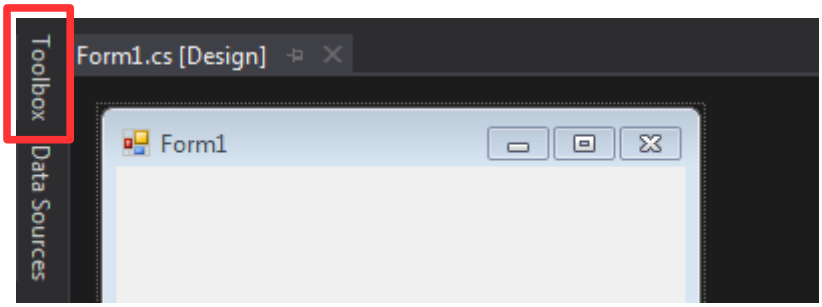
Some of the most interesting form properties are:

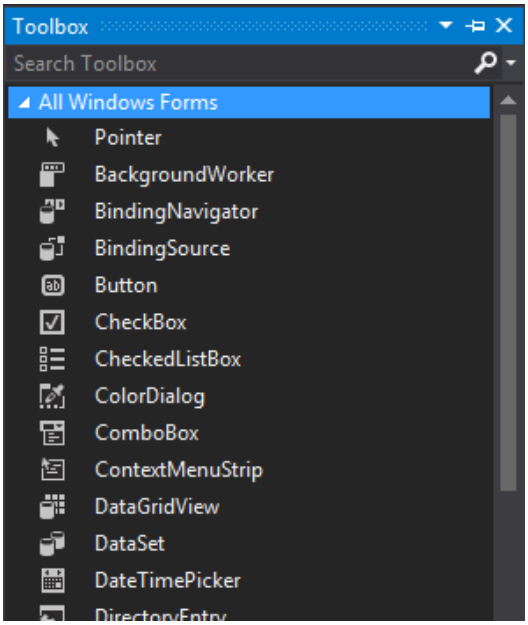
- **Text**, which is shown in the window upper bar
- **BackColor**, background color
- **BackgroundImage**, in case we want to add a background image to our form

- **Forecolor**, foreground (text) color
- **MaximumSize** and **MinimumSize** allowed for the window
- **Size**, current (initial) size (it must be a value between *MinimumSize* and *MaximumSize* properties)
- **StartPosition**: initial position in screen
- **WindowState** to set if we want the main window to start minimized, maximized or normal

2.2. Some typical controls

If we want to explore the control toolbox, we must click on Toolbox section in the upper left corner:





Some of the most typical controls in Windows Forms applications are:

- **Button**: to define action buttons, such as *Accept, Cancel, Save...*
- **Checkbox**: to define controls to check or uncheck some options
- **ComboBox**: to define dropdown lists
- **Label**: to add some explanatory texts next to some controls
- **ListBox**: to display a set of elements in a list with a fixed size (not a dropdown list)
- **PictureBox**: to add images to our application

- **RadioButton**: to define a set of buttons in which only one of them can be pressed at any time
- **TextBox**: to add either single line or multiline texts

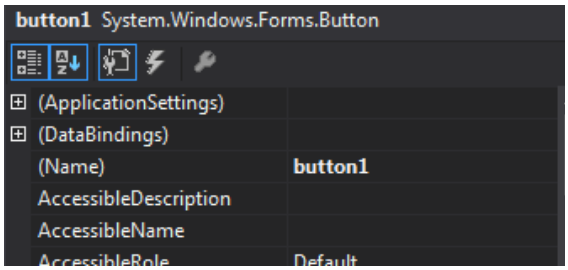
However, there are a lot of additional controls that we can use in our applications. If we want to use any of these controls in our application, we just select it from the *Toolbox* and drag it in the form, right in the place where we want the control to be. Besides, if we have other controls already placed in our form, you will be able to see some guides to help you align the controls properly.

2.2.1. Some control properties

As we have seen for the main form, there are some interesting properties regarding each control. Some of them are the same than the ones explained for the form: background color, foreground color, text, size... But some other properties are more specific: *Visible* (it makes the control visible or not), *Enabled* (it enables/disables the control)...

2.2.2. Names and naming conventions

Among all the properties, there is one of them who is really important: the **Name** property (actually, it is *(Name)* in the properties list, and it is at the beginning of the list). This property lets us define a variable name for the control, so that we can reference the control with this name in our C# code.



Although you can use any valid name for your controls, there are some conventions that you should take into account. Depending on the control type, there is a prefix that is usually placed in the control name. Some of the most typical are:

- *btn* for buttons
- *txt* for text boxes
- *lbl* for labels
- *chk* for checkboxes
- *cmb* for combo boxes
- *lst* for list boxes

2.3. Containers or layout managers

Containers or layout managers are a special set of controls that help us group a subset of controls and treat them as a whole: we can move them as a unit, or draw a border that surrounds all of them... Some of the most typical containers are:

- **FlowLayoutPanel**: controls placed inside this container are automatically arranged from left to right and from up to down.

- **GroupBox**: we use this container to just group controls and give them a common border and/or title
- **Panel**: a simple panel to group controls and let us move them as a unit
- **SplitContainer**: it makes two different areas within the container
- **TabControl**: to create tabbed panels
- **TableLayoutPanel**: to create tables, with their rows and columns. We can also specify the number of columns, and the title and width of each column.

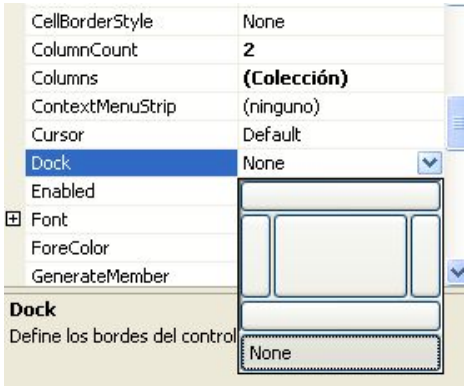
Regarding **properties**, containers have more or less the same set of properties than controls (colors, fonts, sizes, name and so on).

2.4. Docks and anchors

If we just place containers and/or controls inside our form, we may make a big mistake. What if we resize the window? Maybe the original position and size of each control is not suitable for the new window size.

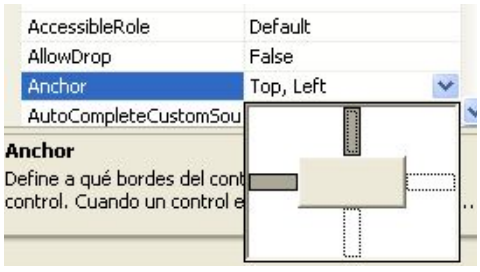
To avoid this problem, there are two properties in each control and container that we can edit:

- **Dock**: it specifies in which part of the window the control/container is added:



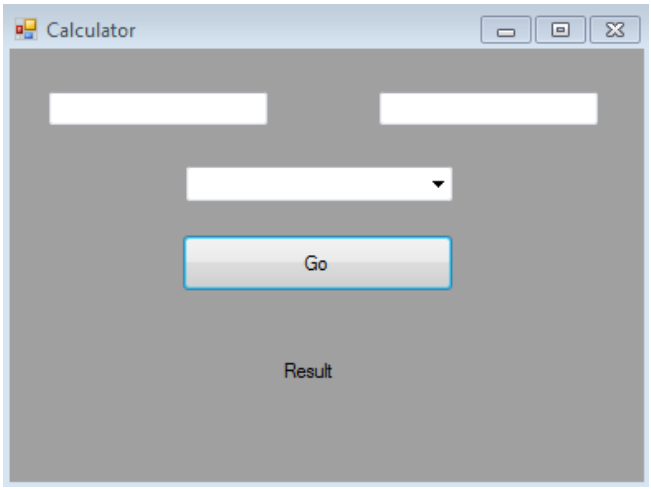
  Then, if we resize the main window, the control/container will remain stuck on the area that we specified.

- **Anchor**: it lets us determine the reference borders for each control. So, the control/container is going to keep the same distance from these reference borders, regardless of any window resize.



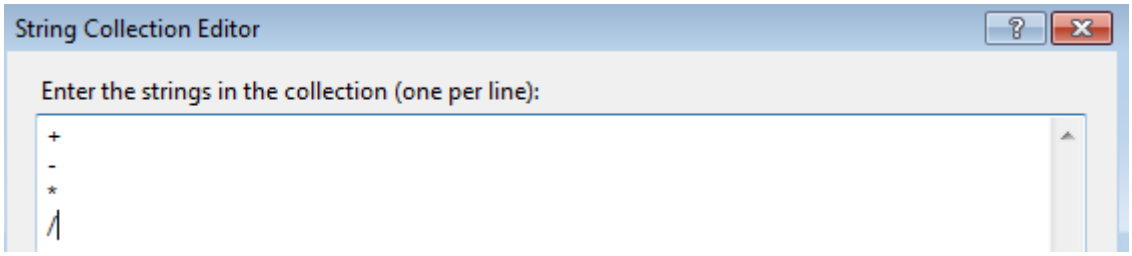
2.5. A simple project: Calculator

Let's create a simple Windows Form application called *Calculator*. Use two text boxes, a combo box, a button and a label, and place them in the form to get something like this:



First of all, we are going to change some properties:

- Set form title (text) to *Calculator*
- Set form background color to some dark gray
- Set button text to *Go*
- Set label text to *Result,* and label font and size to an appropriate value to see the result clearly
- Set the *Items* property of the combo box to a list with these four strings: "+", "-", "*" and "/"



Let's give a name to each control:

- Set left text box to *txtNumber1*
- Set right text box to *txtNumber2*
- Set combo box to *cmbOperator*
- Set button to *btnGo*
- Set label to *lblResult*

Now let's change the *Anchor* properties:

- Anchor left text box to top and left borders
- Anchor right text box to top and right borders

- Anchor combo box and button to top, left and right borders
- Anchor the label to the bottom, left and right borders

Finally, let's specify the form size:

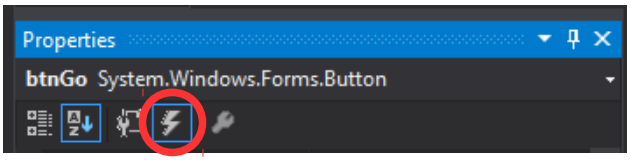
- Define a default (initial) size of 400 x 300
- Define a minimum size of 300 x 200, and a maximum size of 500 x 500.

Try to run the application when you finish, and check the resizing operations. You won't be able to do anything else for now, until we add some events in next section.

3. Events

Events are what make an application really useful. They are code blocks that run when something happens. For instance, if we want to save data when we click on a *Save* button, we need to define a *click* event on that button that runs a method with the instructions to save the corresponding data.

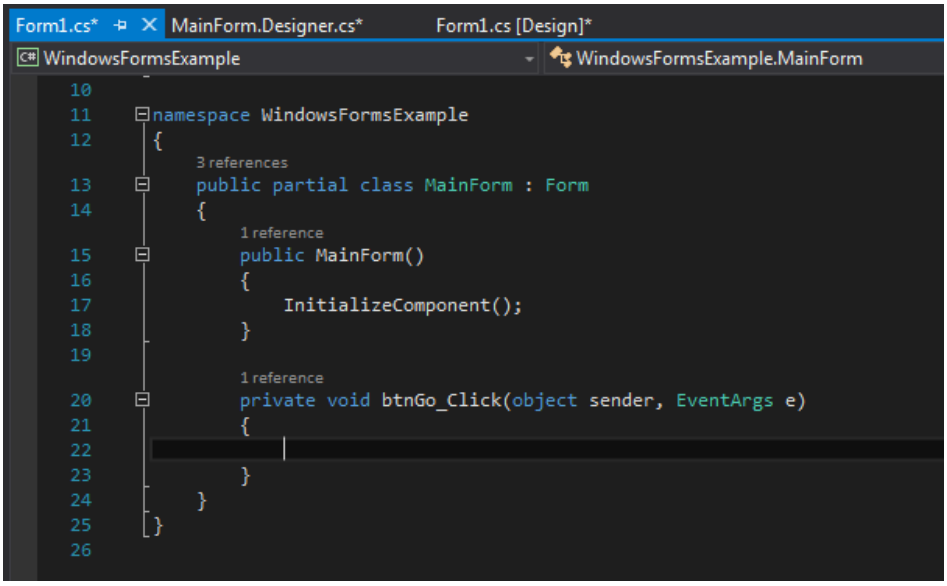
In order to define events on any control or container (or form), there is a specific tab in the properties panel (the one with the lightning icon):



If we click on it, we can see the list of all the available events for the selected control. Some of the most useful are *Click* or *MouseClick* (it fires when we click on the control), *KeyPress* (it fires when we press any key in the control), *TextChanged* (for text boxes, for instance, if fires when the text inside them changes)...

3.1. Choosing an event. Design and code view

If we double click on any event in the list, a code page will be opened. If you pay attention to the file name, it has the same name than our form but without the *Design* suffix.



Besides, a new method is added representing the event that we have just selected (in our case, a *Click* event on a button). Inside this method we can add the code that we want for that event. We can even reference all the controls of our application, as long as we have set a name for them. Don't worry, the whole set of controls, names and properties are automatically stored in *Form.Designer.cs* file. You can edit this file, although it is not necessary, normally. Everything that we do in the design view is automatically updated in the *InitializeComponent* method inside *Form.Designer.cs* file.

Whenever we double click on an event, Visual Studio adds a new line in *Form.Designer.cs* file indicating the relationship between the control and the event. For instance, for previous example, a line like this would be added:

```
this.btnGo.Click += new System.EventHandler(this.btnGo_Click);
```

3.1.1. Default events

Each control has a default event. For instance, button's default event is *Click*. So if we double click on a button (instead of double clicking the *Click* event), then we will go to the same place: an empty *Click* method to be filled.

3.1.2. Removing an event

What if we have made a mistake and we don't want to use an event that we have previously created? We need to follow these steps:

1. Remove the event method from the *Form1.cs* file (or whatever file name you have chosen)
2. Go to *Form.Designer.cs* file and remove the line of code that links the control to the event.

3.2. Defining events in our calculator example

Let's go with our calculator project. Define a *Click* event in the *Go* button, as the example shown before. Inside this method, we are going to:

- Store the values typed in the two text boxes in two integer variables
- Check the selected value in the combo box
- Depending on the selected value, we are going to do the corresponding arithmetic operation and print the result in the label

Our code should look like this one:

```
private void btnGo_Click(object sender, EventArgs e)
{
    int number1 = Convert.ToInt32(txtNumber1.Text);
    int number2 = Convert.ToInt32(txtNumber2.Text);
    string selOperator = (string)(cmbOperator.SelectedItem);
    switch (selOperator)
    {
        case "+":
            lblResult.Text = "" + (number1 + number2);
            break;
        case "-":
            lblResult.Text = "" + (number1 - number2);
            break;
        case "*":
            lblResult.Text = "" + (number1 * number2);
            break;
        case "/":
            lblResult.Text = "" + (number1 / number2);
            break;
```

```
    }
}
```

Try to run the code now, and see how our new event works.

3.3. Showing error messages

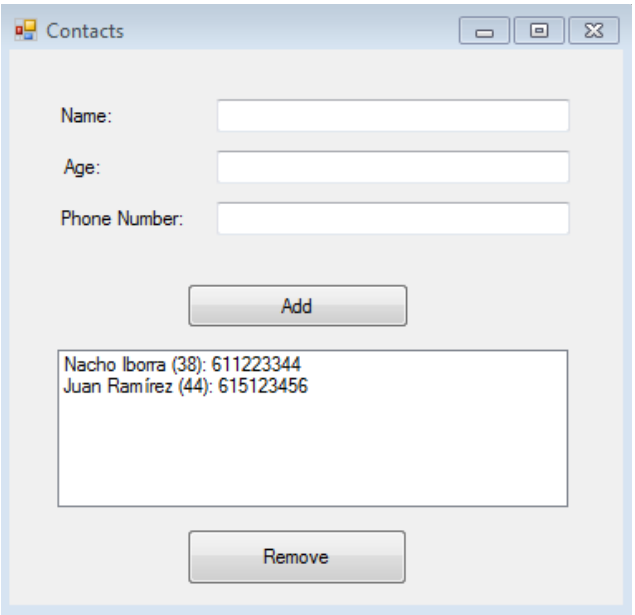
If we want to alert the user that something goes wrong, we can no longer rely on the console (this is not a console application, remember). So we need to show some message in a window. To help us with this issue, we can use the *MessageBox.Show* method. For instance, we can show an error or alert message this way:

```
MessageBox.Show("No operator is selected");
```

MessageBox.Show methods is overloaded, so you can use it with some more parameters if you want to, specifying the message title, icon and some other properties. See the *MessageBox.Show* official documentation for more details.

4. A final project

To practice with all the concepts learnt in this short tutorial, let's create another Windows Forms application called *Contacts*, with the following appearance:



The application must have:

- Three labels and text boxes in the upper section.
- An *Add* button below them
- A list box below the *Add* button
- A *Remove* button below the list box

Whenever we fill the three text boxes and click on the *Add* button, a new contact must be added to the list box. To deal with contacts, create your own *Contact* class inside the project, with three attributes: name (string), age (integer) and phone number (string). When we click on the *Add* button, we will check if there is any empty field (if so, we will show an error message). If everything is OK, a new *Contact* object will be created with this data, and added to the list box.

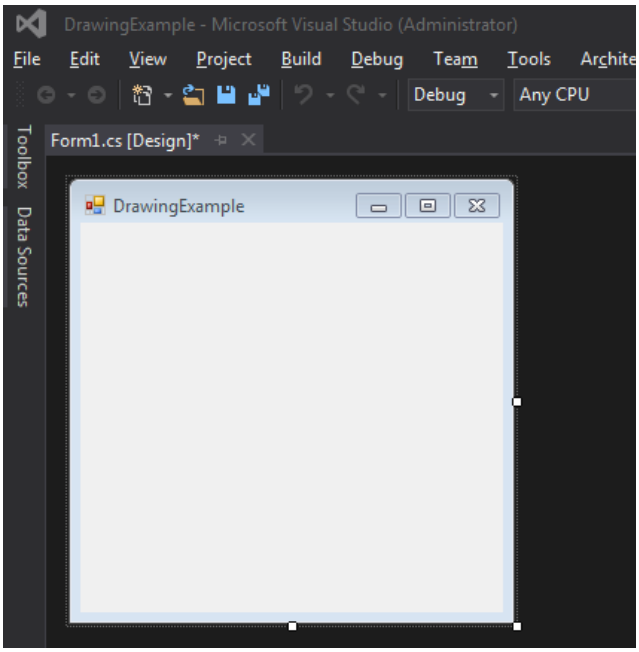
To make contacts be shown in the list box as we see in previous image, you may need to override the *ToString* method of *Contact* class, so that we print the name, the age (between parentheses) and the phone number.

Finally, if we click on the *Remove* button, we will remove the selected item from the list (if any item is selected).

To finish the project, try to store the data in a text file, with a given format (for instance, all the attributes in the same line, separated by ';'). We will need to save the data every time we add or remove a contact to/from the list.

5. Drawing with Windows Forms

In this last section of this tutorial we are going to see the basics of how to draw geometrical figures or images in a Windows Forms application. To start with this, we just need to create a Windows Forms application.



We really don't need any additional control if we don't want to, we can just draw in the main form by firing the corresponding events. But let's see how to define the objects to draw in the form, and the main instructions for drawing geometrical figures.

5.1. Colors, pens and brushes

First of all, we need to define the border and/or the fill color. To do this, we need to create a *Pen* object (for the border) or a *SolidBrush* object (for the filling). When we create them, we can specify the color to draw with. All these elements are inside *System.Drawing* namespace.

```
using System.Drawing;
...
Pen redBorder = new Pen(Color.Red);
SolidBrush blueFill = new SolidBrush(Color.Blue);
```

5.2. Getting the graphics environment

Before drawing anything, we need to get the graphics environment of the application, from the event or method where we want to draw.

```
Graphics currentGraphics = this.CreateGraphics();
```

Then, we will use this object to draw anything on it (lines, circles, images and so on).

5.3. Drawing basic figures

Once we have the pens or brushes established, we can draw some basic figures with them. As we have said before, we need to get an instance of the graphics environment. Then, we can draw figures in this graphics context. For instance, we can draw lines:

```
currentGraphics.DrawLine(redBorder, 100, 100, 200, 200);
```

The four parameters next to the *Pen* object are the (X,Y) coordinates of the starting point and the (X,Y) coordinates of the finish point, respectively.

If you want to fill circles/ovals, you can do this:

```
currentGraphics.FillEllipse(blueFill, new Rectangle(200, 200, 100, 100));
```

In this case, we define a rectangle that surrounds the ellipse or oval, specifying its initial point (200, 200), and its width and height (100 x 100)

We can also draw a border and a filling. Try this rectangle:

```
currentGraphics.FillRectangle(blueFill, new Rectangle(0, 0, 100, 50));
currentGraphics.DrawRectangle(redBorder, new Rectangle(0, 0, 100, 50));
```

There are some more useful methods inside *Graphics* class to draw many different figures, such as *DrawArc, DrawCurve, DrawImage, DrawPolygon, DrawString...*

5.3.1. Freeing memory after drawing

It is important to free memory resources when we finish drawing. To do this, we just call the *Dispose* method of every pen and brush that we have created, and also for the graphics context.

```
redBorder.Dispose();
blueFill.Dispose();
currentGraphics.Dispose();
```

5.4. Some useful drawing options

5.4.1. Drawing images

A special feature of this *System.Drawing* namespace is the ability of drawing images in the window. To do this, we use the *DrawImage* method. But before this, we must create an *Image* object with the image file name, and then call *DrawImage* method indicating the image to be drawn, the initial coordinates (upper left corner), and the image width and height. This code draws an image called *mario.bmp* in coordinates X = 0, Y = 100, with 50px width and 70px height. Image is automatically scaled to the specified dimensions.

5.4.2. Moving figures. Clearing the graphics context

We could implement a video game embedded in a Windows Forms application by using the drawing techniques explained above. To do this, we need to implement the classical game loop, and clean the graphics context at the beginning of each iteration. To clean the graphics context, we can just call its *Clear* method. We can specify a background color to clear the image, or *Color.Transparent*.

```
currentGraphics.Clear(Color.Black);
```

Then, we just need to draw the moving image in a different location.