Getting Started with VS2013

Once you’ve installed VS2013, let’s create a GUI with several types of controls and do something simple with them. There’s no great point to this program, other than giving you a head start on things.

* I’m going to go into some depth first as to what’s going on. I certainly wouldn’t do this for a beginning programmer, but I’m assuming you’d find the background info useful in understanding how things are happening.
* I’m going to assume you’ve created a directory called C:\WRR\C#
* Bring up Visual Studio. You’ll probably see two panes, one captioned *Toolbox* and one labelled *Start Page*. On the *Start Page*, click *New Project*. Alternatively, click File | New | Project.



* This will create, among others, a .sln (Solution) file. A Solution is one or more projects. For example, at some point you might want to create a .exe and its accompanying .dll file. There would be one .csproj file for the .exe, and another .csproj file for the DLL. But in this case, of course, there will be only one project. But if you want to launch VS from File Explorer, you’d normally (double-)click on the .sln file, not the .csproj file.
* From the dialog that appears, choose Templates | Windows Desktop, then choose Windows Forms Application. At the bottom of the dialog, give it a name, say, WRR1, and set the Location to C:\WRR\C#. Click OK.
	+ 
* VS will create Form1.cs as follows. More on this shortly.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WRR1 {

 public partial class Form1 : Form {

 public Form1() {

 InitializeComponent();

 }

 }

}

* You should also see the Solution Explorer pane. If not, click menu View | Solution Explorer. Right-click on Form1.cs, then click Rename from the popup menu. Overwrite “Form1” with, say, “WRR1”. You’ll get a box that says, “You are renameing a file. Would you also like to perform a rename in this project of all references to the code element ‘Form1’?”. Click Yes. Note that the code now says

public partial class WRR1 : Form {

 public WRR1() {

 InitializeComponent();

 }

* If you now click the Start icon on the toolbar (), you’ll run the program. It will just be an empty window with the caption “Form1”. But it can be maximized, minimized, dragged on the screen and closed.
* Now let’s look at the code a bit.

Namespaces

Suppose you’re in the (unlikely) situation of having to write a program that includes both banking data and hobbies. You might want to have two quite different classes named Interest. This can be done by defining each in its own namespace, which can be used to qualify the names. In this case, VS has defined the WRR1 namespace for us.

More to the point, all those *using* statements at the top of the program reference namespaces. For example, the *System* namespace has definitions for many common classes, for example, *string*. The *System.Drawing* namespace (namespaces can be nested) defines routines for drawing, say, Rectangles, Ellipses and many others. And so on.

 If you reference a class without a namespace qualifier, the compiler will search all defined namespaces. If it finds only one definition, it will implicitly use that one. If it finds more than one, you’ll get a compile time error and you’ll have to explicitly qualify it. But we normally don’t have to worry about such things.

*Using* Statements and References

In the Namespaces section, I was careful to say that naespaces *defined* classes, as opposed to saying that they *contained* classes.

Unlike C and C++, C# executable files (.exe or .dll) contain metadata that defines all the things in the program. Class names. function names. function parameters, data types for all fields, variables and parameters. And more.

The actual code for the routines in, say, the *System* namespace (and the classes, etc in it, such as *string*) is contained in C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.5.1\System.dll

If you look in the Solution Explorer, you’ll see a tree node for References, and within that, an entry for System.

So the way it works is this.

* References point to (usually) DLLs that have one or more namespaces, each with their own set of classes and the corresponding code.
* A *using* statement finds the namespaces in the DLLs and reads the corresponding metadata. It then enters this into the program’s symbol table.
* And away you go.

Note: All *using* statements must appear before the first line of code (i.e. non blank lines or comments).

The Good News About Namespaces and Usings, But a Small Warning

The good news is that Visual Studio provides the most commonly used references and usings by default, so you can mostly ignore them.

However, you might want something outside these defaults. Say you want to use a Regular Expression. Once you do some querying and find that there’s a Regex class, you can’t just say

 Regex MyRegex = new Regex();

The Regex class is defined in namespace System.Text.RegularExpressions. The actual code is in the same DLL that has the System.Text namespace, but you haven’t told the compiler to load the metadata (symbol table info) for that namespace. So you’d have to do either

 System.Text.RegularExpressions.Regex MyRegex = new System.Text.RegularExpressions.Regex();

or (more commonly)

 using System.Text.RegularExpressions;

 …

 Regex MyRegex = new Regex();

The Code

Again, the code is…

namespace WRR1 {

 public partial class WRR1 : Form {

 public WRR1() {

 InitializeComponent();

 }

 }

}

As you can see, C# is one of the curly-brace languages. I’ll want to talk about source code formatting issues, but I’ll worry about that later.

**Namespace:** We understand namespaces, so let’s skip over that.

**Class:** The next line defines a class called *WRR1*. (I’m assuming you’re familiar with object-oriented programming concepts, such as classes, inheritance, etc. if not, let me know and we’ll talk about that Ilater.

It’s a *public* class, which means that, if we ever use this .exe file as a Reference, it would be visible to the outside world. Of course, this doesn’t concern us now.

It’s also a *partial* class. A Windows Forms program puts the definition of the GUI into a separate source module but wants all the objects readily available to the main class (WRR1). So in that source module, it also says that this is a partial class named WRR1.

In the Solution Explorer, expand the WRR1.cs node and see the WRR1.Designer.cs source module. You’ll see the other partial class there, also named WRR1. Note that this file will be automatically updated as you create your user interface using the GUI designer component of Visual Studio.

So when you actually compile the program, it will treat the multiple source files as a single logical file, find that there are two definitions for class WRR1, but since they’re defined as partial, it will merge the definitions.

In other words, you don’t have to worry about the partial keyword. It’s there mostly for the GUI builder component.

Now note the “WRR1 : Form” part. The “:” means that WRR1 is derived from class Form, which is the class that defines a window.

**Constructor:** As you’re of course aware, perhaps the most common programming bug (other than logic errors) concerns undefined variables. C# has the concept of a *constructor*. This is a subroutine (identified by having the same name as the class itself) that is called automatically when an instance of the class is created. In this case, it calls the InitializeComponent() routine. The is in the above-mentioned source module that defines the GUI.

The Other Pieces

There’s also a Program.cs source module created for you. It has a subroutine called *Main*. When you run the executable, initialization code looks for a Main() routine and runs it. As you can see, it instantiates a new copy of class WRR1 and runs it.

You’ll also see AssemblyInfo.cs. In .NET terminology, an Assembly has nothing to do with machine language. Loosely speaking, an Assembly is the set of files (mostly .exe and .dll) required to run a program. See [http://en.wikipedia.org/wiki/Assembly\_(CLI)](http://en.wikipedia.org/wiki/Assembly_%28CLI%29) for more information, but you really don’t have to bother.

Looking around the Solution Explorer some more, you’ll find a Resource file that can contain embedded graphics (e.g. icons), strings (e.g. for international apps, you might want text strings in English, French and German, but not hard-coded into the code), and so on.

There are also files that can be used for configuration info.

But at this stage, you can ignore every one of these.

OK, Let’s Build That GUI

In the Solution Explorer, right-click on WRR1.cs, and click View Designer.

On the left you should see a Toolbox pane. If not, click View | Toolbox. This contains all your controls.

In the middle, you should see a tab captioned “WRR1.cs [Design]”. It will be a blank window with the caption “Form1”.

On the right, you should see two panes, stacked on top of each other. The top would be the Solution Explorer, and the one underneath, Properties. Again, if not visible, they can be made so from the View menu.

Change the Caption

Click inside the empty window (captioned “Form1”). Over in the Properties pane, scroll down until you find the property named *Text*. Change that to, say, “My First C# Program” (no quotes) and hit Enter. Notice that the caption in the designer has changed.

If you were to look at WRR1.Designer.cs, you’d notice the insertion of the line

 this.Text = "My First C# Program";

Now run the program (the ) icon. The window still doesn’t do much, but at least the caption has changed!

Resize the Form

There are 3 small sizing rectangles on the My First C# Program window. Click and drag on any of them to make the form wider, larger, smaller, etc.

We’ll be adding a number of controls, so make it large-ish.

As with all these steps, you can see what’s changed in WRR1.Designer.cs

Add a Button

From the Toolbox pane, find the Button icon and drag it onto the design surface, somewhere near the top left.

In the Properties pane, change its Text to Push Me.

Also change its (Name) from the generic (and not very self-documenting) to, say, btnPushMe.

Now double-click the button on the design surface. You’ll see

 private void btnPushMe\_Click(object sender, EventArgs e) {

 }

This (empty) routine will be invoked when you click the button.

How does this work? Look at the Designer (WRR1.Designer.cs) code. It’s added

 this.btnPushMe.Click += new System.EventHandler(this.btnPushMe\_Click);

A Button has a property called *Click*, which is a list of event handlers to be invoked whenever the button is clicked. We’ve added (“+=” operator) a new event handler with an address of btnPushMe\_Click.

Let’s add just a bit of functionality to it. Place the cursor inside the body of the function.

Terminology Note: Functions and subroutines within a class are usually referred to as *methods*.

Type in “mbox” (no quotes) then hit the TAB key twice. Change the text from “Test” to, say, “Don’t do that. It tickles!”.

Run it.

Hey, congratulations. You’ve just written your first functional C# GUI program!

Note: The *sender* parameter tells you what control generated this event. You can have the same handler for more than one control; this lets you know which one you’re talking about. The *e* parameter doesn’t particularly apply to buttons, but it’s there for consistency’s sake.

Help!

Click the mouse on the “Show” method name and hit F1. The Help Viewer should show up, describing the syntax of the Show method.

Note that this method is *overloaded*. There are many methods named Show, each with a different parameter list. The compiler infers which method to use based on the data types in the parameter list.

Try playing around with the extra parameters, especially for the caption, buttons and icon.

Label and TextBox

Using the Toolbox, below the button, add a Label. Notice that as you drag it along, a blue vertical line may appear. It does so when the label would be exactly under the button. Very helpful for aligning the GUI.

Change its Text property to “Name” (no quotes). Usually for labels I don’t bother changing their (Name) property.

Now add a TextBox to the right of the label. A blue horizontal line will help align the bottoms of the label and textbox. But once you’ve dropped the textbox, click on it and drag it down just a pixel or two. You’ll see now a ref horizontal line. This is how you align the text inside the two controls, which is normally what you’d want.

Give it a (Name) of txtName.

The textbox has two sizing handles. Grad the right-hand one and make the textbox wider, most of the way to the right-hand side of the form (window). Scroll towards the bottom of its Properties pane and click on the Anchor property. It should say Top, Left. Click the down-arrow to the right. You’ll see a little diagram with the Top and Left boxes grayed. Click on the Right box as well.

Run the program, then resize the runtime window to be either wider or narrower. You’ll notice the textbox automatically resizes, staying the same distance from the right-hand side of the window.

Another Button

Add a second button, with Text “Hello” (no quotes) and (Name) btnHello. Align the text (red line) with the label and textbox.

Place it on the same line as the Label and Textbox, to the right of the textbox, near the right-hand edge of the form. Set its Anchor property to Top, Right. (Side note: In this case, the Top setting doesn’t do anything.)

Run the program and change the width of the runtime window. Note how the new button hugs the right side of the window, while the textbox remains the same distance from it. (Although if you make the window too narrow, the textbox resizes into nothing.)

The Second Button and the Textbox

Double-click on the Hello button. As you might expect, you’ll get generated for you

private void btnHello\_Click(object sender, EventArgs e) {

}

Add the following code:

private void btnHello\_Click(object sender, EventArgs e) {

 string txt = txtName.Text.Trim();

 if (txt.Length == 0) {

 MessageBox.Show("And whom do I have the pleasure of addressing?");

 } else {

 string Hiya = "Hello " + txt;

 MessageBox.Show(Hiya);

 }

}

The only thing here that might be strange is the Trim() function. This is a method of the *string* class that trims off any leading and trailing blanks.

You’re On Your Own At This Point

Give it a spin. Have fun! And email/phone me for help.

And install Teamviewer; it’s a great tool!