Creating Windows Forms Applications with C#

by Jason Pursell, University of Washington, Bothell (2000-2003)
jpursell@u.washington.edu

PART 1 – INTRODUCTION

CONTENTS

OBJECTIVE

The obvious objective is to introduce Windows Forms by creating a small application similar to the dialog-based application created in previous tutorials (see **History** section below). I don't try to teach C# or Windows Forms in their entirety, because plenty of books and websites already discuss the subject. On the other hand, I do provide plenty of extra information, such as an explanation of the code generated by Visual Studio .NET. I assume that you are not a beginner and some knowledge of programming, preferably in C++, is helpful. The only other requirement is your desire to know more than just step 1, step 2, and so on.

UWB CSS Students: This tutorial is primarily written for current and former students of *CSS 450 – Computer Graphics* and *CSS 451 – 3D Computer Graphics* at the University of Washington, Bothell. As of February 2004, Professor Sung uses MFC as the application framework for his class examples, so it's necessary to provide a "translation" for those who would like to use C# and the .NET Framework for completing their assignments. This is important since I provide C# versions of the class examples and this tutorial serves to lay a common foundation for those class examples – you could call this "class example zero."

TUTORIAL HISTORY

This tutorial is really the third version of an original tutorial. It's a translation of a translation! It is based on the following two tutorials:

- *Creating Dialog Based Applications with MFC 7*
  I created this tutorial for the Fall 2002 CSS 450 class. I created it, because Professor Sung decided to use MFC as the user interface framework and many of the students wanted to use MFC 7 (as part of Visual Studio .NET) instead of MFC 6.

- *A Beginner's Guide to Dialog Based Applications – Part One*
  (by Dr. Asad Altimeemy on http://www.codeproject.com)
  This is what my tutorials are based on. It's fairly small and uses MFC 6, so it might be useful for those who are used to the "old way" or just want to continue with MFC 6. I reference it frequently in my MFC 7 tutorial.

REQUIREMENTS

I used Visual Studio .NET 2003 (with .NET Framework 1.1) and Windows XP to create this tutorial. However, there should be no problems using Visual Studio .NET 2002 (with minor adjustments).

If you encounter any problems, please email me and I will correct the issue or note the difference.

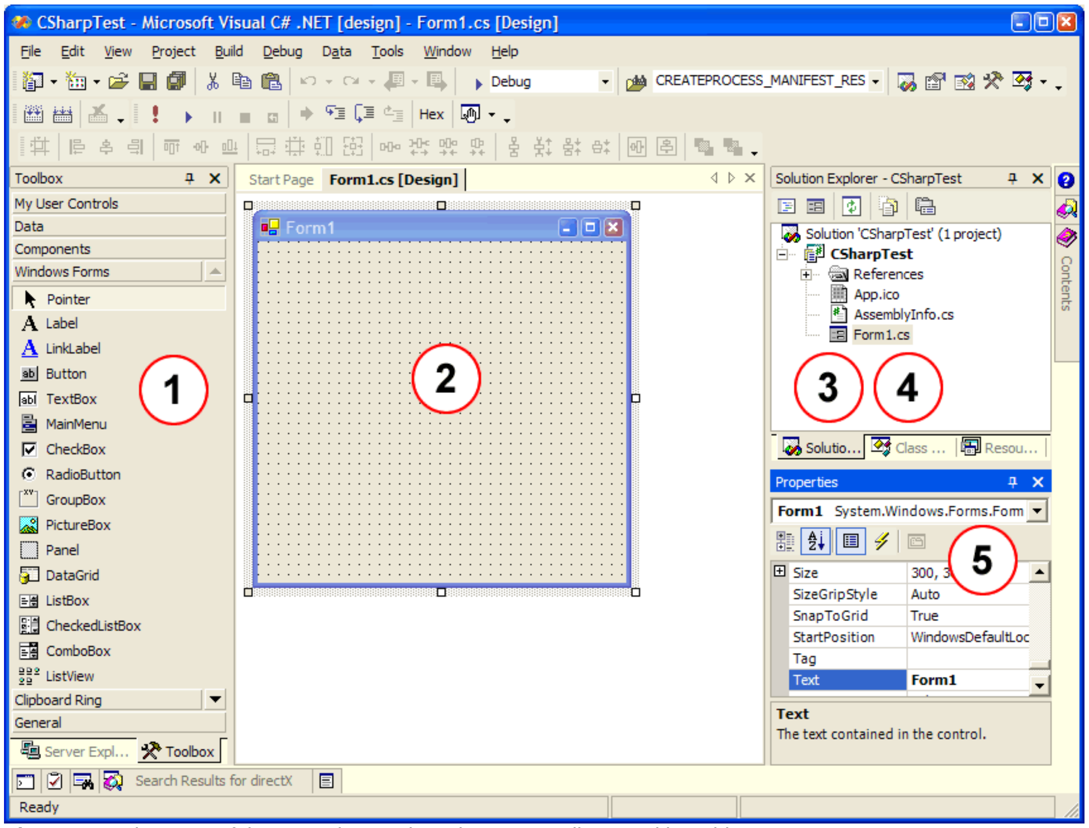
jpursell@u.washington.edu

PART 2 – CREATING THE APPLICATION

VISUAL STUDIO .NET IDE

For the purposes of this tutorial, I will focus only on the parts related to creating Windows Forms with C#. For a much more in-depth discussion of the new features found in Visual Studio .NET, please see this reference from the MSDN library: Introduction to Visual Studio .NET. Even though it uses Visual Basic .NET, it is still an excellent resource.

Visual Studio .NET allows multiple programming languages to share the Integrated Development Environment (IDE), not just Visual C++, so there are new parts to the IDE that are common to all the languages, and some familiar parts of Visual Studio 6.0are gone or replaced. See **Figures 1**, **2a**, **2b**, and **Table 1**.



Size	300, 3
SizeGripStyle	Auto
SnapToGrid	True
StartPosition	WindowsDefaultLoc
Tag	
Text	Form1

Text
The text contained in the control.

Figure 1. Major parts of the IDE. The numbered areas are discussed in Table 1.

IDE Component	Description
1. Toolbox	– Displays a list of controls and components for you to add to your form. The items available can include .NET components, COM components, HTML objects, code fragments, and text.
2. Windows Forms Designer	– This is a design-time visual representation of a System.Windows.Forms.Form derived class that you can manipulate and add controls to.
3. Solution Explorer	– Shows the Solution and its Projects. Each project contains its files and References.

4. Class View	– Provides a hierarchical view of the methods, fields, properties, and base classes of each project's objects. Namespaces are honored in this view.
5. Properties Window	– Provides a compact place to view and change the properties and events of the currently selected control, resource, or file at design-time.

Table 1. Major IDE components from Figures 1, 2a, and 2b.

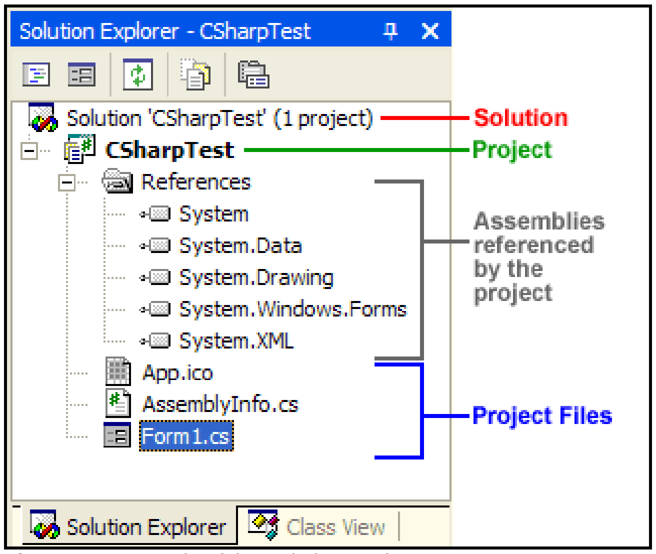


Figure 2a. Example of the Solution Explorer.

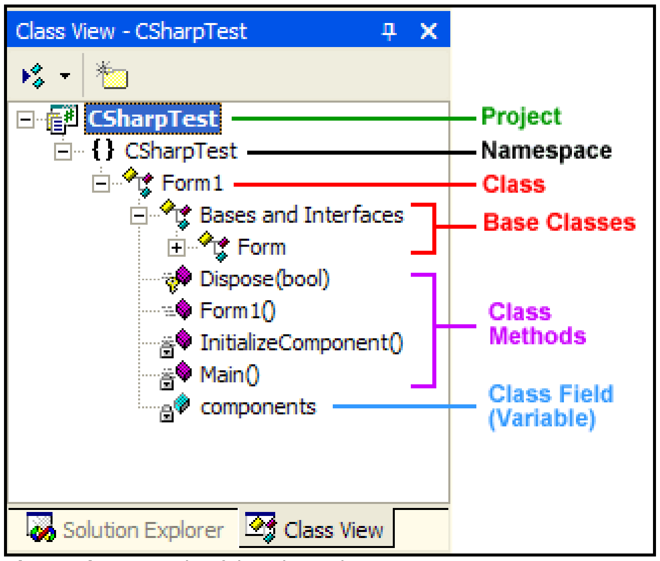


Figure 2b. Example of the Class View

Note What about the Resource View? For those who use MFC, you'll notice that I left it out. You can show it, but it doesn't display anything for **managed only** applications. If your application uses both managed and unmanaged code, then it might be needed. This tutorial won't use mixed languages.

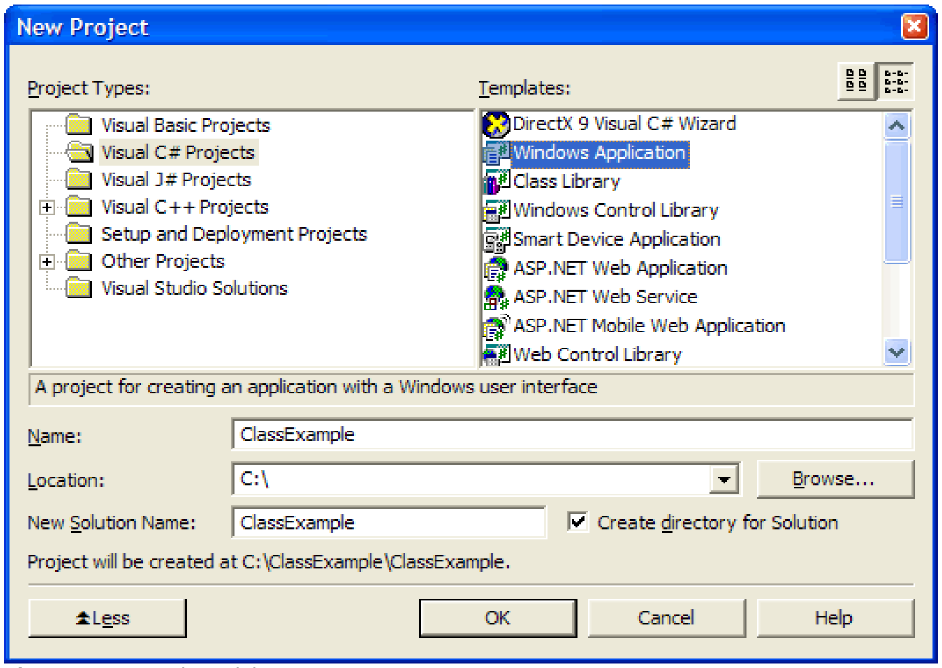
CREATING A NEW PROJECT (AND SOLUTION)

1. Start Visual Studio .NET. On the main menu, choose **File | New | Project** .

2. Once the New Project Dialog opens, choose *Visual C# Projects* in the **Project Types** pane choose. In the **Templates** pane choose *Windows Application*. Provide a **Name** and **Location** for the Project. If this is a brand new solution (and project), then it is good practice to check the **Create Directory for Solution** checkbox. If it isn't showing, then click the **More** button. The dialog will expand and provide space for defining a **Solution Name**. The default Solution Name is the name of the first project, but it doesn't have to be. For now just accept the default.

   The reason for doing this is found in the "Extra Information" section below. Alternatively, you could create and empty solution and then add a new project to it, but this is easier, because you are combining two steps into one.

3. When all the fields are filled in, click the **OK** button.



A project for creating an application with a Windows user interface

Name: ClassExample

Location: C:\

New Solution Name: ClassExample

[✓] Create directory for Solution

Project will be created at C:\ClassExample\ClassExample.

Figure 3. New Project Dialog.

4. Visual Studio will work for a few seconds and then load up with something similar to **Figure 1** from the previous section. Depending on the profile you've set for the IDE the windows might be in different places; the IDE is highly configurable.

   Good, the project is now created and ready to go. If you want to know more about the files that were just created, read the following extra information, otherwise skip to the next section to start designing the application.

HELPFUL INFORMATION (not needed to finish the tutorial)

What exactly was created? **Figure 4a** shows the file structure and the files created, assuming you check the *Create directory for Solution* checkbox. The Solution files (.sln, .sou) are by themselves in the top folder. Each project contains its own Project file (.csproj). Nice and clean, huh? Later on you can add other projects to the solution. **Figure 4b** shows what that might look like.

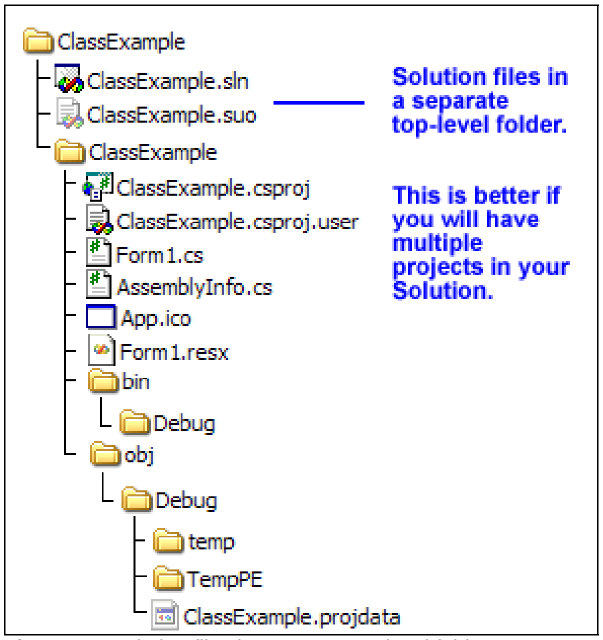


Figure 4a. Solution files in separate top-level folder.

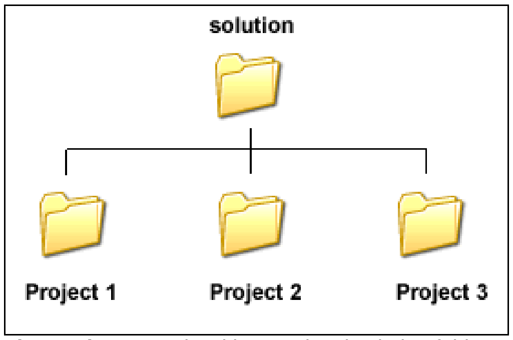
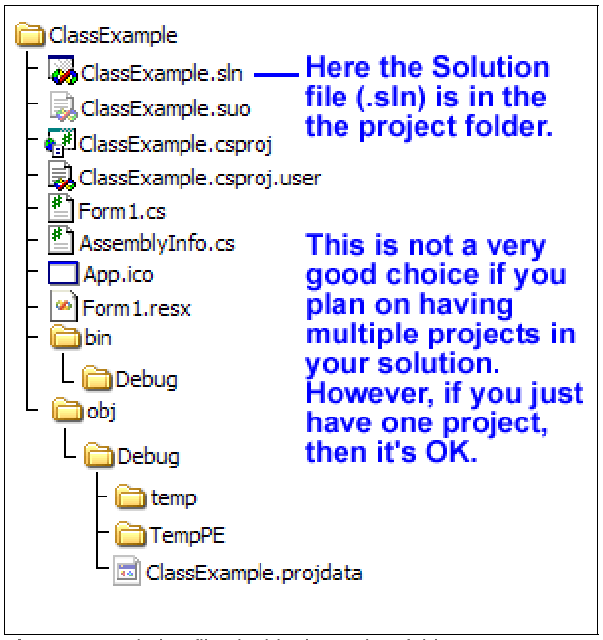


Figure 4b. Example with a top-level solution folder.

If you don't check the *Create directory for Solution* checkbox, all files are created at one level inside the project folder itself as in **Figure 5a**. Additional projects are added at the same level as shown in **Figure 5b**.


ClassExample
ClassExample.sln
Here the Solution file (.sln) is in the the project folder.
ClassExample.suo
ClassExample.csproj
ClassExample.csproj.user
Form1.cs
AssemblyInfo.cs
This is not a very good choice if you plan on having multiple projects in your solution. However, if you just have one project, then it's OK.
App.ico
Form1.resx
bin
Debug
obj
Debug
temp
TempPE
ClassExample.projdata
Figure 5a. Solution files inside the project folder.

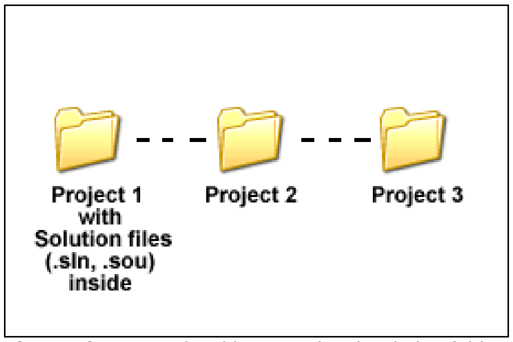


Figure 5b. Example with no top-level Solution folder.

CSS 450/451 Students: Prof. Sung seems to use this method most often; however, the first method is arguably better, because when you add multiple projects, it becomes more manageable. If you just have one project, then it's not a big deal. In CSS450 and CSS451 you will have multiple projects in a solution, so it's best to just get in the habit of checking the *Create directory for Solution* checkbox.

What are these files exactly? The following files are created as part of a C# Windows Application called *ClassExample*.

- **App.ico** The default icon for the application.

- **AssemblyInfo.cs** Used to describe the assembly and specify versioning information.
- **Form1.cs** The C# class source code for the windows form.
- **Form1.resx** An XML file used to edit and define application resources. (**normally hidden*)
- **ClassExample.csproj** The C# project file.
- **ClassExample.csproj.user** Records all of the options that you might associate with your *project* so that each time you open it, it includes customizations that you've made. There is one per project.
- **ClassExample.projdata** Hidden file used by Visual Studio .NET. Not human readable. (**normally hidden*)
- **ClassExample.sln** Organizes projects, project items and solution items into the *solution* by providing the environment with references to their locations on disk.
- **ClassExample.suo** Records all of the options that you might associate with your *solution* so that each time you open it, it includes customizations that you have made. There is one per solution. (**normally hidden*)

* Some of these files might only be seen if you have chosen to *show hidden files and folders* in the Windows 2000/XP Folder Options dialog.

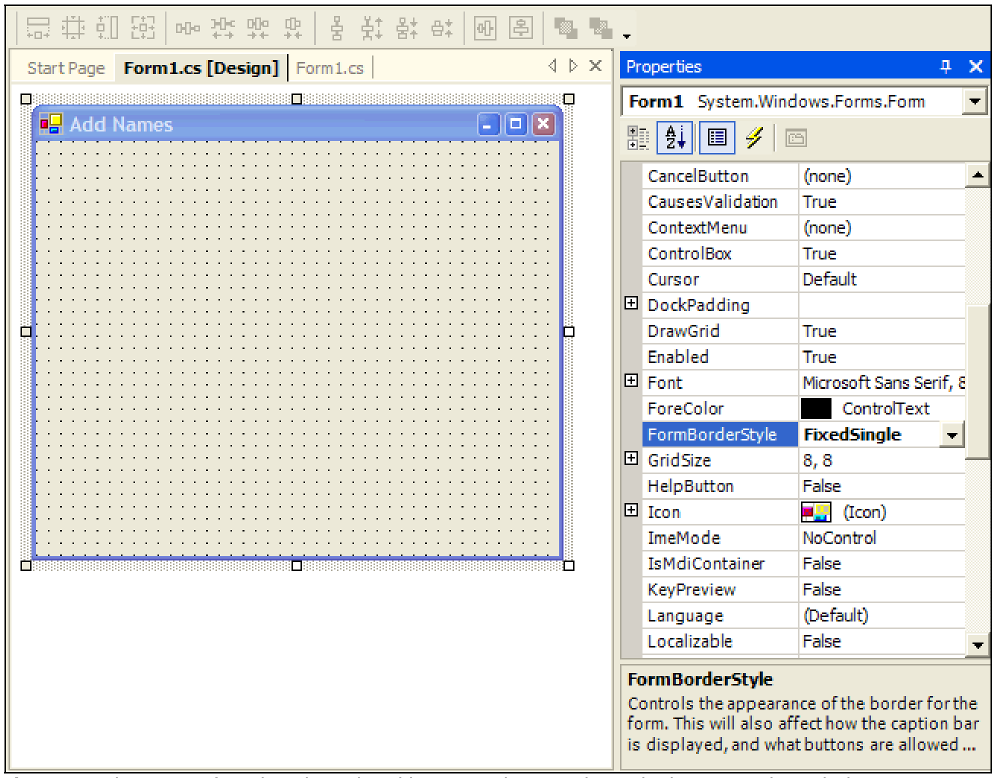
Note Unlike in Visual C++, the folder and files that you see in the Solution Explorer in a C# project represent the physical folders and files on the hard drive (with a few exceptions). If you click on a file and press delete, it will delete the physical file. Thankfully, a message box will prompt you for permission. In a Visual C++ (and MFC) project, if you select a file in the Solution Explorer and "delete" it, then it just removes it from the project; however, the file is not physically deleted from the hard drive.

One major exception to the above rule – the References folder of the project. If you choose one of the assemblies in the References folder and delete it, it still exists; however, you won't be able to program against the classes in that assembly.

Designing the (Windows) Form

OK, finally we can start designing the application. What we will do in this tutorial is create all the controls first, then we will add code afterwards.

1. Make sure that Form1 is displayed and selected in the Designer View window as in **Figure 6** below.



Form1 System.Windows.Forms.Form

CancelButton	(none)
CausesValidation	True
ContextMenu	(none)
ControlBox	True
Cursor	Default
⊞ DockPadding	
DrawGrid	True
Enabled	True
⊞ Font	Microsoft Sans Serif, 8
ForeColor	■ ControlText
FormBorderStyle	**FixedSingle**
⊞ GridSize	8, 8
HelpButton	False
⊞ Icon	(Icon)
ImeMode	NoControl
IsMdiContainer	False
KeyPreview	False
Language	(Default)
Localizable	False

FormBorderStyle
Controls the appearance of the border for the form. This will also affect how the caption bar is displayed, and what buttons are allowed ...

Figure 6. The Form1 form is selected and its properties are shown in the Properties Window.

2. We want to change the title of the form. We also want to change the size and make it fixed, thereby not allowing the user of the application to resize the window. Make the following changes to these properties in the Properties Window:

Property	New Value
FormBorderStyle	FixedSingle
Size	370, 225
Text	Add Names

Adding Controls

1. Now, from the Toolbox window, add a Button control to the form. We also want to change some of the properties of this new button. Make the following changes:

Control Type	Property	New Value
Button	Name	closeButton
	Location	273, 16
	Text	Close

2. Now add the other controls. **Table 2** shows a list of the controls and their properties with new values.

Control Type	Property	New Value
Button	Name Location Text	addButton 16, 16 Add
Label	Text	Title
Label	Text Location	First Name 132, 56
Label	Text Location	Last Name 248, 56
TextBox	Name Location Text	firstTextBox 132, 80 (blank)
TextBox	Name Location Text	lastTextBox 248, 80 (blank)
ComboBox	Name Items Location MaxDropDownItems Size Text	titleComboBox Mr.;Mrs.;Ms.;Miss.;Dr. 16, 80 4 100,21 (blank)
ListBox	Name Location Size	namesListBox 16, 112 336, 69

***see side note on next page**

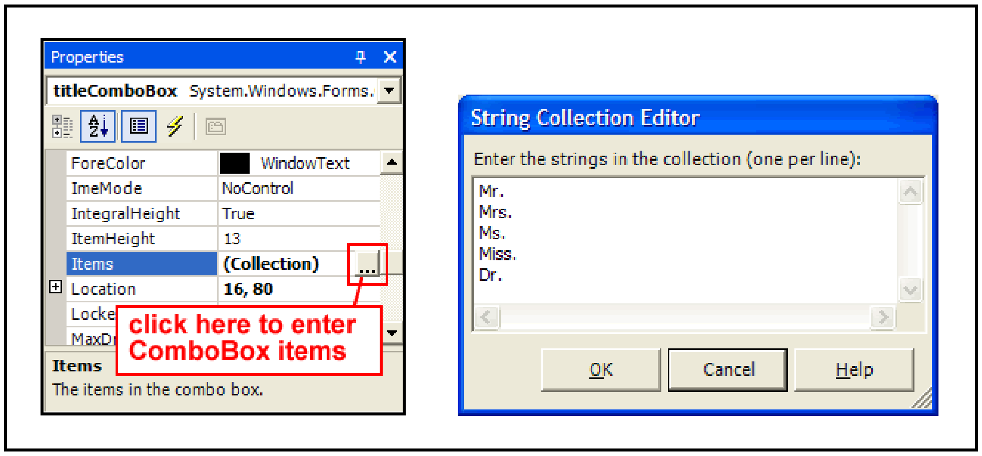
Table 2. Control properties and their new values.

Note I've only included a new size when the control is different from the default size. Also, I've included the location of all of the controls to aid you in their positioning; however, aesthetic properties such as location and size are normally determined by you, depending on the application requirements.

Additionally, some of the control's Text properties are given a value of *(blank).* This just means that it is an empty string (i.e. no text at all).

Lastly, there are multiple ways to add a control to a form. Here is the MSDN page that explains the ways to do it. Adding Controls to Windows Forms

Side Note for the ComboBox – enter Items in the following manner:



```
Mr.
Mrs.
Ms.
Miss.
Dr.
```

3. If you completed all of Step 2, the form should look like Figure 7.

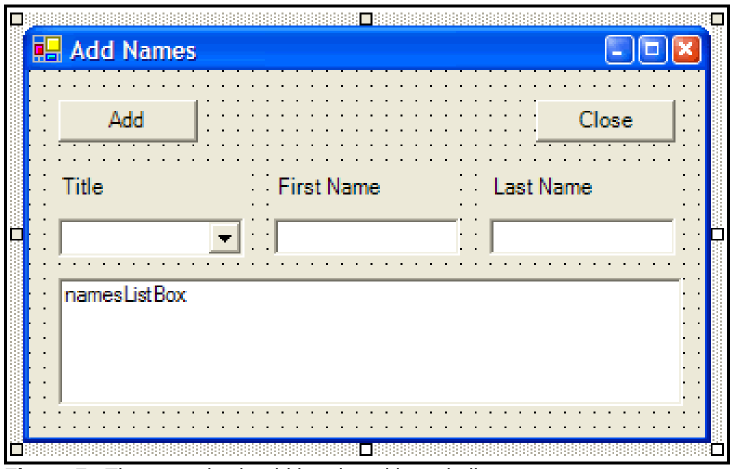


Figure 7. The controls should be placed in a similar manner.

HANDLING EVENTS

Alright, now that you have the form and all its controls in place, we can start on the code. The first thing we will do is to add an event handler for the Close button. Use the Properties Window and its Events View (see **Figure 8**).

1. Select the **Close** Button on the form by left-clicking it **once**.

2. Click the Events button in the Properties Window.

3. In the space for the Click event type: `closeButtonClick`.

4. Press Enter.

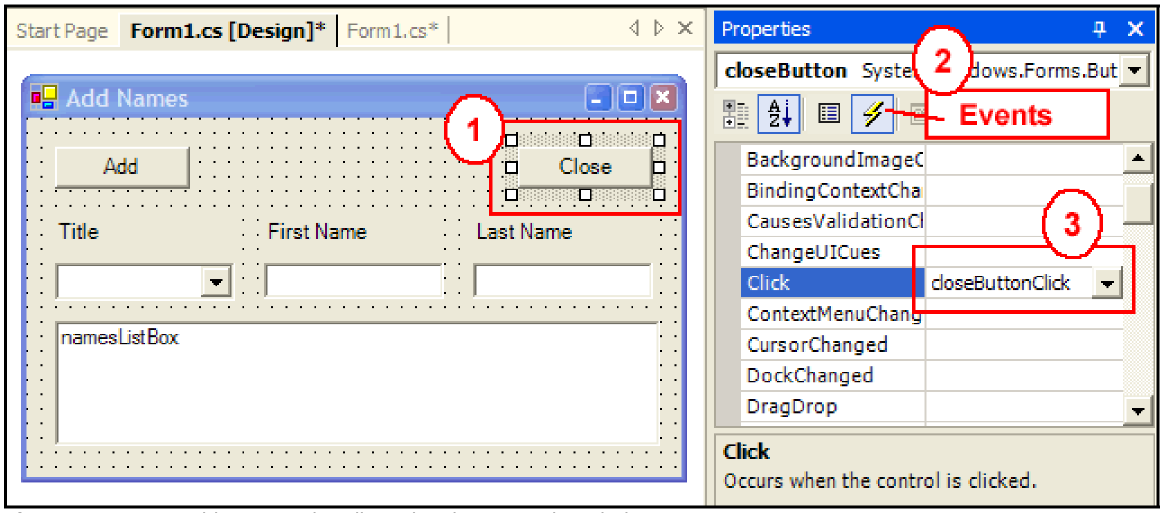


Figure 8. Steps to add an event handler using the Properties window.

Important Alternatively, you can just double-click the control. This will accomplish the same thing as the steps above, but you must accept the default name generated by Visual Studio .NET. The default name separates the control name and the default event with an underscore. So if you use this method, the name for the event method would be *closeButton_Click* instead of *closeButtonClick*. If you use this method, please make the necessary mental adjustments during the remainder of the tutorial.

5. As soon as you press the Enter on the keyboard you are taken to Code View and the cursor is placed in Visual Studio .NET generated code. The new method is in the Form1.cs code file.

6. Where the cursor is, add the code that is in bold. The *Close* method is from the base class – System.Windows.Forms.Form. It closes the form and return execution to the Main method, causing the application to end.

```
private void closeButtonClick(object sender, System.EventArgs e) {
   this.Close(); //add this code
}
```

Note When you define an event for a control, code is added to the *InitializeComponent* method. This method is the one in the “Windows Form Designer generated code” region. The added code below is how C# connects the *closeButtonClick* method to the **closeButton**’s Click event. Basically, the closeButton is saying, “when I am clicked, call the closeButtonClicked method.” This is known as “subscribing” to the event and C# accomplishes this with delegates; however, delegates are used for much more than a form’s control events.

```
//
// closeButton
//
 ...
this.closeButton.Click += new system.EventHandler(this.closeButtonClick);
```

7. Now, repeat steps 1 through 5 for the **Add** button, but name the method *addButtonClick.*

8. After you press Enter and are placed in the *addButtonClick* method type in the following bolded code:

```
private void addButtonClick(object sender, System.EventArgs e) {
    //add the following code
    string title = titleComboBox.Text;
    string first = firstTextBox.Text;
    string last  = lastTextBox.Text;

    string nameToAdd = title + " " + first + " " + last;

    namesListBox.Items.Add(nameToAdd);
}
```

Note If you are adding many items to a ListBox or ComboBox, then it is good practice to enclose the calls to *Items.Add* with the BeginUpdate and EndUpdate calls. This will help reduce any flicker that might happen otherwise. For example:

```
namesListBox.BeginUpdate();
namesListBox.Items.Add(nameToAdd);
namesListBox.EndUpdate();
```

If you are adding only one item at a time, then it's not a big deal.

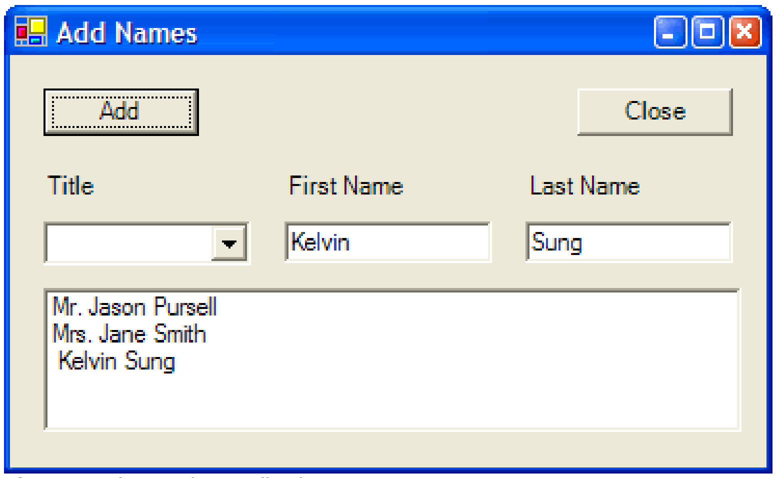
9. You're all done! Now we just need to build the application and run it.

Note The .NET Framework naming convention suggests all methods should start with a capital letter. This is called Pascal casing. The naming convention for fields (class member variables) is to start with a lower case letter. This is called camel casing. There is an exception to the convention with respect to Windows Forms controls. The control is added as a private member field to the form class and should start with a lower case letter. However, when you create an event handler for that control, it is not capitalized. It's one of those grey areas that all naming conventions have. In this case, the control is a member field of the form class, so it uses camel casing. Since the event handler is usually private, the fact that the method starts with a lower case letter, and therefore "breaks" the naming rules, is not that big of a deal.

BUILDING AND RUNNING THE PROJECT

This is the easy part. Here you have a few choices. You can build the application. You can build and start (with debugging) or you can build and start without debugging.

1. On the main menu, choose **Debug | Start** (or press F5 on the keyboard).
2. When the form is visible, test the application by entering information into the fields and clicking the Add button. If everything was done correctly, the names are added to the list box. Here is an example of the application in action:



Kelvin
Sung

Mr. Jason Pursell
Mrs. Jane Smith
Kelvin Sung

Figure 9. The running application.

3. Click the **Close** button when your done.

Note If you don't need to debug you code, then use **Start without Debugging** (Ctrl+F5). Your application will load much faster.

4. That's it. You're all done. This application doesn't perform any real error checking or special case formatting, but it should get you started.

   If you would like to learn more about the actual code, then continue on to the next section. In my opinion, it is the most important part of this tutorial. Otherwise, immediately following that section, I list the references that I used for creating this tutorial.

PART 3 – UNDERSTANDING THE CODE

This section is really an added bonus for you. I really want to help you understand what is going on in the code and feel that this section might actually be the most important part of the tutorial. It's not meant to be exhaustive in nature and I don't focus much on the syntax of C#. Instead, I focus on the larger purpose of a line (or lines) of code. In order to help you more, I've included relevant links to the MSDN library at the end of each code block's description. Please email me if you have any questions or corrections that you feel I should know about.

This section will explore and describe all of the code from the ***Form1.cs*** class file (from the top down). Code from the file is in the gray shaded blocks.

Starting at the top, the *using* directives allow you to avoid fully qualified namespaces when declaring types. They are not required, but if you don't use them, then you will have to fully qualify every type that you use. These *using* directives are the ones that the C# Windows Application wizard generates for you. Note that even though the wizard includes them, all of them are not used by this tutorial's application. Namely, no types in the *System.Data* or *System.Collections* namespaces are referenced in this tutorial. Microsoft made the decision that these were the most commonly used and included them.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

Anyway, the main point is that the *using* directive allows you to type a lot less when declaring or referencing types. It's really not a big deal for one or two lines, but for entire programs it becomes tedious to fully qualify every type. As an example, wouldn't you rather type this code?

```
Button btn = new Button();
```

Instead of this,

```
System.Windows.Forms.Button btn = new System.Windows.Forms.Button();
```

It's important to understand that even though the *using* directives are optional, the assemblies in the References folder of the project (in the Solution View) are not. If you want to use the types from the *System.Windows.Forms* namespace, then you must reference the assembly that contains its types. In Visual Studio .NET, each C# project contains a References "folder". This folder contains the default set of referenced assemblies as pictured in figure x.

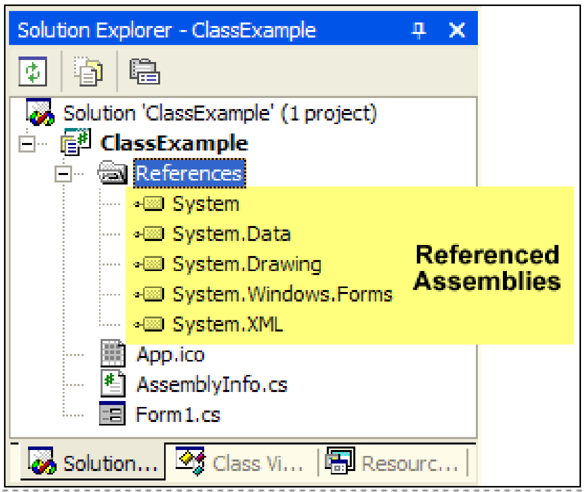


Figure x. The default assemblies referenced by a C# Windows Forms Application created with Visual Studio .NET.

Note that although many namespaces and assemblies have the same name (minus the ".dll" extension), they don't have to. In fact, an assembly can contain multiple namespaces and namespaces can span multiple assemblies. For instance, the *System.Collections* namespace is in the *System.dll* assembly and the *System.Data.dll* assembly contains part, but not all, of the *System.XML* namespace.

For a more in-depth treatment of the *using* directive and the relationship between namespaces, assemblies, and references, please check out these MSDN library pages:

[MSDN Links]
The using directive in C#
Using fully qualified names
Namespaces in the .NET Framework Class Library
Assemblies
Project References

This one line of code is the namespace of the project. It defaults to the name of the project, but you can change it if you want. The use of namespaces is highly recommended and Visual Studio .NET assumes that you will use them, but it doesn't require it. If you don't use a namespace, then the types that you create will be at the "global" namespace, which is basically no namespace. Some words of advice – use namespaces for your own code.

```
namespace ClassExample
{
```

Here is the beginning of the *Form1* class definition. It is derived from the *System.Windows.Forms.Form* class. In other words, *Form1*'s base class is *Form*, found in the *System.Windows.Forms* namespace.

```
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
```

[MSDN Links]
Defining classes in C#

When you added controls to the form at design time, the Windows Form Designer generated this code. These are private member fields of the *Form1* class.

```
        private System.Windows.Forms.Button closeButton;
        private System.Windows.Forms.Button addButton;
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.Label label3;
        private System.Windows.Forms.TextBox firstTextBox;
        private System.Windows.Forms.TextBox lastTextBox;
        private System.Windows.Forms.ListBox namesListBox;
        private System.Windows.Forms.ComboBox titleComboBox;
```

Notice that the control member fields use the fully qualified namespace. This really isn't necessary, because of the using directive (`using System.Windows.Forms`) at the beginning of the code. Again, Visual Studio .NET is just being explicit. You can remove the *System.Windows.Forms* portion from the following member field declarations and everything will still build and run fine.

> **Note** C++ developers should be careful here. These fields are `null` at this point. In C#, these are references and you must create an object using the `new` keyword. This is done in the *InitializeComponent* method later in the code. If you try to use a member field (member variable in C++) before it has been initialized using `new`, you will be presented with a Null Reference exception. It might be helpful to think about this way – they are more like C++ pointer variables, but without the indirection operator (`*`).

[MSDN Links]
Comparison between C++ and C#
Static and instance members

This next bit of code declares a private member field which is used by Visual Studio .NET and Windows Forms for tracking the components used by the form.

```
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;
```

That raises the question, "what's a component?" Well, there is more than one meaning depending on what the context is. The context here is Windows Forms programming and the controls found in the Toolbox. The Toolbox window has both controls and components of which you can add to the form. The Timer, ToolTip, and SaveFileDialog are examples of components found in the Toolbox window. The Button, TextBox, and Label are examples of controls. Note that all controls are components, but all components are not controls. The technical distinction can be found at this MSDN page: Class vs. Component vs. Control. The brief, but less accurate, distinction is that controls are components that have an editable user interface.

As far as you are concerned though, the distinction is best demonstrated in Visual Studio .NET when adding the control or component to the form. Components that are added to a form will show in the Component Tray below the form. Please note that we don't use any components in this tutorial, but here is an example picture:

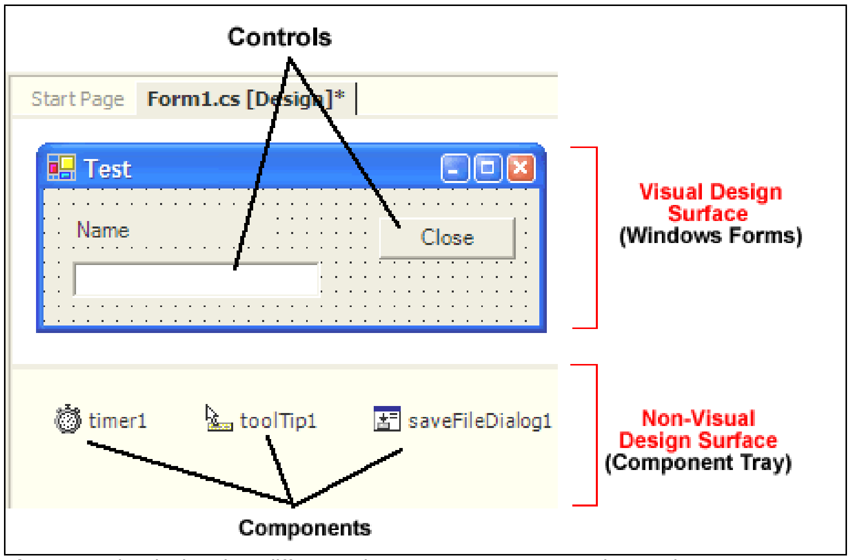


Figure x. The design-time difference between components and controls.

The following links are helpful:

[MSDN Links]
Class vs. Component vs. Control
Controls You Can Use on Windows Forms

This is the Form1 class constructor. It makes a call to the *InitializeComponent* method. That method is found in the special Windows Form Designer generated code region and is discussed a little later. It's where the controls you've added to the form are instantiated and initialized. Just like the TODO comment says – add any initialization code of your own after the call to *InitializeComponent*.

```
        public Form1()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            //
            // TODO: Add any constructor code after InitializeComponent call
            //
        }
```

Also, understand that the *InitializeComponent* method is not some magic C# function, it's just how Visual Studio .NET chooses to initialize everything. If you created a Windows Forms class manually, you could use a differently named function or none at all and put everything in the constructor itself. It doesn't matter, as long as you're sure to initialize everything that needs to be initialized.

[MSDN Links]
Here is a list of the different types of constructors. The first one is the normal public constructor that we all know and love. The other two are special ones that might come in handy.

C# Class Instance Constructors
C# Class Private Constructors
C# Class Static Constructors

We are now presented with the *Dispose* method. This section is a bit long, albeit important. If you want to skip this section and come back later, that's fine. In order to fully explain this would take many pages, so I leave most of the topic for your study. For a start, please see the MSDN links at the end of this code section. I will, however, lay out a brief explanation here; just as long as you understand that my explanation is less than adequate because of its brevity.

Ok, let's look at the method signature first. The *protected* keyword means that a class member is accessible from within the class in which it is declared, and from within any classes derived from that class. The *override* keyword is used to override a virtual method in a base class. In C++, we would use the *virtual* keyword in the base class method and optionally in the derived class method. In C#, the *virtual* keyword is also used in the base class method, but unlike C++, C# requires the *override* keyword in the derived class method. I think you'll agree that C# is clearer about virtual methods.

```
         /// <summary>
         /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }
```

The *Dispose* method is where you dispose of managed and unmanaged resources used by the form. If the parameter *disposing* is true, then the managed resources used by the form will be disposed of. In this tutorial the *components* variable is *null*, because the form doesn't contain any managed components like a Timer or a SaveFileDialog (see previous page for more information).

If the parameter *disposing* is false, only the unmanaged resources used by the form will be disposed of. With unmanaged resources, you must call their dispose method manually. This tutorial doesn't use any unmanaged resources, but if it did you would call their *Dispose* method outside of the `if` statements, but before the call to the base *Dispose* method. The very last thing is to call the base class *Dispose* method.

Now we'll look at the purpose of the *Dispose* method. I believe it's impossible to understand this method's purpose without first understanding how the Common Language Runtime, or CLR, uses *garbage collection* to manage memory. The .NET Framework uses what is called a *generational garbage collector*. What this basically means is that you don't have to worry about managing your own memory, because the CLR will do it for you. This is why there is no *delete* or *free* keyword in C# like there is in C or C++. Instead, the garbage collector will eventually come along and collect any objects that are no longer in use. The CLR is highly optimized in determining the best time to collect garbage and is (normally) not under your control.

> **Note** There is a way to force garbage collection with the System.GC class, but second guessing the CLR as to when to run the garbage collector can be detrimental to performance; however, there may be times when you might want to use the *System.GC* class, but be careful.

In C++, when an object goes out of scope or the object is deleted with the *delete* keyword, its destructor is called. This is called *deterministic destruction*, because you know exactly when the object's memory is freed. However, .NET Framework languages only provide *nondeterministic destruction*, because you don't know when an object will be collected by the garbage collector. So even though C# has what looks like a C++ destructor (~), it is only the syntax that is the same. In fact, the CLR doesn't know anything about destructors. Behind the scenes, the C# destructor syntax is translated into a method called *Finalize*. In C# the destructor (or *Finalize* method) is called when the garbage collector reclaims the memory. There are a couple of important aspects to the destructor in C#:

1. Just like in C++, you can't call the C# destructor (*Finalize* method) explicitly.
2. The Finalize method it is never guaranteed to be called.

For unmanaged resources like file handles or database connections, you need to be able to release them deterministically. This is where the *Dispose* method comes in. For now, just think of the *Dispose* method as a way to simulate the behavior of a true C++ destructor.

> **Note** At the end of this tutorial I list the references that I used for this tutorial. One book in particular, that I highly recommend, is *Applied Microsoft .NET Framework Programming* by Jeffrey Richter. This book has an absolutely outstanding chapter called "Automatic Memory Management (garbage collection)." This chapter provides a lot of important information about the garbage collection process, including the reasons why the *Dispose* method exists. I'm not exaggerating when I say that this book should be on every .NET developer's bookshelf.

[MSDN Links]
Access Modifiers
override keyword
Garbage Collection in .NET
Dispose method (general)

Implementing a Dispose Method
Dispose Method (of the Form – inherited from System.ComponentModel.Component)
C# Destructor – Finalize Method

The `#region` preprocessor directive lets you specify a block of code that you can expand or collapse when using the Code Outlining feature of the Visual Studio Code Editor. The matching `#endregion`, found at the end of the *InitializeComponent* method, marks the end of this block.

```
        #region Windows Form Designer generated code
```

Note Even though C# has preprocessor directives such as `#region`, `#endregion`, and `#define`, there is no real preprocessor as in C/C++. The name was kept, primarily because of the similarity to the C/C++ syntax. The main difference is that you can't use pseudo constants or macros with `#define`.

[MSDN Links]
Code Outlining
C# Preprocessor Directives
#region

Here is the *InitializeComponent* method that is called from the class constructor. This is where the controls are actually instantiated (with the *new* keyword); their properties are initialized a little bit later. Since the control variables are members of *this* class, Visual Studio .NET appends the *this* keyword; however, it is not required. It is just Visual Studio .NET playing it safe again. There are times when the *this* keyword is needed, such as when a method parameter and a class member field have the same name. The *this* keyword will distinguish the class's member field.

```
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.closeButton = new System.Windows.Forms.Button();
            this.addButton = new System.Windows.Forms.Button();
            this.label1 = new System.Windows.Forms.Label();
            this.label2 = new System.Windows.Forms.Label();
            this.label3 = new System.Windows.Forms.Label();
            this.firstTextBox = new System.Windows.Forms.TextBox();
            this.lastTextBox = new System.Windows.Forms.TextBox();
            this.namesListBox = new System.Windows.Forms.ListBox();
            this.titleComboBox = new System.Windows.Forms.ComboBox();
```

Now that the controls are instantiated, you can reference them without getting a Null Reference Exception.

[MSDN Links]
new Operator
this Operator

Continuing with the *InitializeComponent* method, the call to `SuspendLayout` causes the program to suspend the application of property changes such as Size, Location, Dock, and Anchor. The call to *ResumeLayout* (towards the end of the *InitializeComponent* method) causes the suspension to be lifted and the changes to take effect. This is meant to improve redraw performance.

```
            this.SuspendLayout();
```

[MSDN Links]
SuspendLayout

This part of the *InitializeComponent* method is where the changes you made in the Properties window in the *Designing the (Windows) Form* section of this tutorial actually take effect. Notice the event handlers for the *closeButton* and *addButton* controls. This is explained in the note found in Step 6 of the *Handling Events* section earlier in the tutorial. This code demonstrates another important topic – Properties. Properties are the C# equivalent of *get* and *set* assessors methods in a C++ class, but better. C# understands the Property organically and provides a special syntax for it. Understanding properties is easiest when you are designing your own class, so I'll wait until a later tutorial for an example, but if you would like to learn more about them, please see the MSDN link at the bottom of this code block. Notice how you set the Text property. You don't have to call a function and pass in the string; rather, you just perform an assignment. Behind the scenes, you really are calling a function, but C# properties abstract that from you.

```
            //
            // closeButton
            //
            this.closeButton.Location = new System.Drawing.Point(273, 16);
            this.closeButton.Name = "closeButton";
            this.closeButton.TabIndex = 0;
            this.closeButton.Text = "Close";
            this.closeButton.Click += new System.EventHandler(this.closeButtonClick);
            //
            // addButton
            //
            this.addButton.Location = new System.Drawing.Point(16, 16);
            this.addButton.Name = "addButton";
            this.addButton.TabIndex = 1;
            this.addButton.Text = "Add";
            this.addButton.Click += new System.EventHandler(this.addButtonClick);
            //
            // label1
            //
            this.label1.Location = new System.Drawing.Point(16, 56);
            this.label1.Name = "label1";
            this.label1.TabIndex = 2;
            this.label1.Text = "Title";
            //
            // label2
            //
            this.label2.Location = new System.Drawing.Point(132, 56);
            this.label2.Name = "label2";
            this.label2.TabIndex = 3;
            this.label2.Text = "First Name";
            //
            // label3
            //
            this.label3.Location = new System.Drawing.Point(248, 56);
            this.label3.Name = "label3";
            this.label3.TabIndex = 4;
            this.label3.Text = "Last Name";
            //
            // firstTextBox
            //
            this.firstTextBox.Location = new System.Drawing.Point(132, 80);
            this.firstTextBox.Name = "firstTextBox";
            this.firstTextBox.TabIndex = 6;
            this.firstTextBox.Text = "";
            //
            // lastTextBox
            //
            this.lastTextBox.Location = new System.Drawing.Point(248, 80);
            this.lastTextBox.Name = "lastTextBox";
            this.lastTextBox.TabIndex = 7;
            this.lastTextBox.Text = "";
            //
            // namesListBox
            //
            this.namesListBox.Location = new System.Drawing.Point(16, 112);
            this.namesListBox.Name = "namesListBox";
            this.namesListBox.Size = new System.Drawing.Size(336, 69);
            this.namesListBox.TabIndex = 8;
            //
            // titleComboBox
            //
            this.titleComboBox.Items.AddRange(new object[] {
                                                  "Mr.",
                                                  "Mrs.",
                                                  "Ms.",
                                                  "Miss.",
                                                  "Dr."});
            this.titleComboBox.Location = new System.Drawing.Point(16, 80);
            this.titleComboBox.MaxDropDownItems = 4;
            this.titleComboBox.Name = "titleComboBox";
            this.titleComboBox.Size = new System.Drawing.Size(100, 21);
            this.titleComboBox.TabIndex = 9;
```

[MSDN Links]
Event Handlers in Windows Forms
Properties in C#

Here is the section of *InitializeComponent* that defines the form itself. It also adds the controls to the form, since a Form is a container control. This helps, because when the form is disposed, the controls it contains will be disposed of as well. Other container controls include controls such as the GroupBox and Panel.

```
            //
            // Form1
            //
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(364, 198);
            this.Controls.Add(this.titleComboBox);
            this.Controls.Add(this.namesListBox);
            this.Controls.Add(this.lastTextBox);
            this.Controls.Add(this.firstTextBox);
            this.Controls.Add(this.label3);
            this.Controls.Add(this.label2);
            this.Controls.Add(this.label1);
            this.Controls.Add(this.addButton);
            this.Controls.Add(this.closeButton);
            this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedSingle;
            this.Name = "Form1";
            this.Text = "Add Names";
```

[MSDN Links]
Controls Property
Controls.Add

Here is the match to *SuspendLayout* from the beginning of the *InitializeComponent* method. Again this will cause all of the suspended layout changes to take effect. This method is overloaded.

```
            this.ResumeLayout(false);
        }
```

[MSDN Links]
ResumeLayout

Here is the match to `#region` from just before the *InitializeComponent* method.

```
        #endregion
```

Here is the ubiquitous *Main* method. Notice that in C# the *Main* method has a capital 'M'. This is in keeping with the naming convention of capitalizing class methods. The *Main* method must be *static*, because there can be only one per class. *Main* can be declared with either a *public* or *private* access modifier; it doesn't matter, because *Main* is treated as a special method by C# and the access modifier is essentially ignored. If you don't specify *public* or *private* (as in the code below), it defaults to *private*. *Main* can return *void* or an *int*. It can also accept command line arguments, but these variations of the *Main* method aren't covered here.

The other thing you'll notice is the strange thing inside square brackets (*[STAThread]*). This is called an attribute. An attributes is a powerful programming concept used throughout the .NET Framework. This particular attribute specifies a Single Threaded Apartment for the Component Object Model (COM). This is only needed if your application makes use of COM through a process called COM Interop. You would do this if you used things like ActiveX controls. In this application, this is ignored completely by the compiler, because we don't use anything from the COM world. In other words, it doesn't hurt anything to just leave that line in the code, just in case you decide to use a COM component like Drag and Drop. That's why Visual Studio .NET generated the code for you.

```
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
```

The next important thing is the call to the *Application.Run* method. *Application* is a convenience class from the *System.Windows.Forms* namespace that provides static methods for things like starting, stopping, and showing the form. In this case, the *Run* method is called with the form that you want to start and show. The *Run* method also starts the message loop for the form.

[MSDN Links]
The Main method
STAThreadAttribute
Application class

These are the methods that you added to handle the Close and Add buttons. The first method just closes the form. The *this* keyword is optional. When this method is done, program control will return to the *Main* method.

The second of the two event handler methods gets the text in each of the editable controls, concatenates them, then adds the result to the listbox. Notice the use of the Text property. This demonstrates the "get" version of using a property. The "set" version was used in the *InitializeComponent* method.

The *`BeginUpdate`* and *`EndUpdate`* method calls are not necessary here, but might improve performance (reduce redraw flicker) if many items are added at one time.

```
        private void closeButtonClick(object sender, System.EventArgs e) {
            this.Close();
        }


        private void addButtonClick(object sender, System.EventArgs e) {
            string title = titleComboBox.Text;
            string first = firstTextBox.Text;
            string last  = lastTextBox.Text;

            string nameToAdd = title + " " + first + " " + last;

            namesListBox.BeginUpdate();
            namesListBox.Items.Add(nameToAdd);
            namesListBox.EndUpdate();
        }
    }
}
```

Let's look at these last two methods in a different light. The parameter *sender* is the object that caused the event to happen. Since we have explicitly connected the button controls on the form to these two methods, we know what object is really the *sender*. For example, the addButton is the only control that will call the addButtonClick method – ever. It is the *sender* object. You know this, because no other button has "connected" to this event handler method.

An Alternate Method – One Event Handler for Multiple Buttons
Now, if you are paying attention, you'll realize that you could use one method to handle all the buttons (or groups of buttons) on a form. Let's see how you would do this. I've only included the necessary parts of the code to get the point across. Notice that the method name passed to the *System.EventHandler* is the same for both buttons. I called this method *buttonClick*.

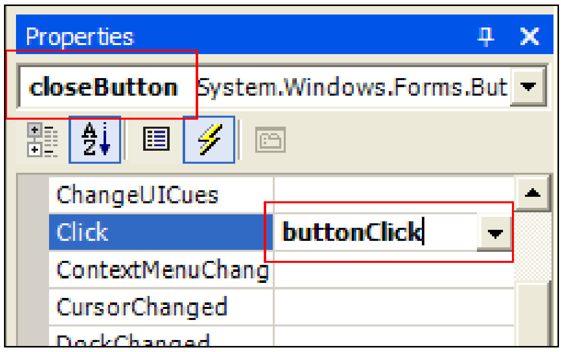
```
    InitializeComponent()
    {
        …
        // closeButton stuff
        this.closeButton.Click += new System.EventHandler(this.buttonClick);
        …
        // addButton stuff
        this.addButton.Click   += new System.EventHandler(this.buttonClick);
        …
    }
```

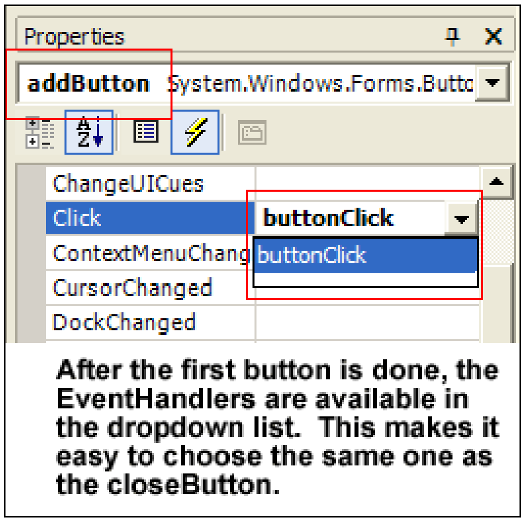
Now instead of two separate methods (*closeButtonClick* and *addButtonClick*), you only have one called *buttonClick*.

```
    private void buttonClick(object sender, System.EventArgs e) {

        if (sender == closeButton) {
            this.Close();
        }
        if (sender == addButton) {
            // code for adding stuff just like in the original
            // addButtonClick goes here.  I'm lazy! ☺
        }
    }
```

The choice is yours – one method for each control or one method for multiple controls. However, if you choose to use this alternate version, **you must make sure not to double-click the control** to add an event handler at design-time. You should use the Properties window like I discussed earlier in the tutorial. Instead of using different names, just use the same name. The following pictures show what this would look like – the *closeButton* first, then the *addButton*.




Properties
addButton System.Windows.Forms.Butto
ChangeUICues
Click buttonClick
ContextMenuChang buttonClick
CursorChanged
DockChanged

After the first button is done, the EventHandlers are available in the dropdown list. This makes it easy to choose the same one as the closeButton.

[MSDN Links]
[Form.Close method]
[ListBox Class]
[Multiple Events to a Single Event Handler]

That's it. For more in-depth information, please see the next section.

PART 4 – REFERENCES USED

This is a listing of the references I used while creating this tutorial.

- *Microsoft Visual C# .NET*. MS Press (2002). Mickey Williams.
- *Applied Microsoft .NET Framework Programming*. MS Press (2003). Jeffrey Richter.
- *Inside C#, 2nd Edition*. MS Press (2002). Tom Archer and Andrew Whitechapel.
- *Windows Forms Programming in C#*, Addison Wesley (2003). Chris Sells.
- *Programming Microsoft .NET*. MS Press (2002). Jeff Prosise.
- MSDN Library