

# **C# Fundamentals**

# **via ASP.NET**

**COURSEWARE by Bob Tabor  
(BobTabor.com, LearnVisualStudio.NET)**

**PDF by Steven Nikolic**



001

# Introduction

In this series of lessons, you will be learning the C# programming language by creating dynamic web applications via ASP.NET. These lessons were adapted from the "C# Fundamentals via ASP.NET Web Applications" video series that can be found at:

<http://www.LearnVisualStudio.net>

If you have not purchased a subscription to that video series, doing so will provide you with an excellent, visually-oriented companion piece to the written lessons presented here. It is not strictly necessary to use the video lesson material along with the written lessons, or vice versa, but using both in tandem will significantly fast-track your learning process.

For a broad description of the available video courses, as well as available plans & pricing, visit:

<http://www.learnvisualstudio.net/courses/>

## The Reason This Series Exists

A language like C# is typically learned "in a vacuum," and stripped of potentially distracting visual elements or front-end graphical interfaces. This type of learning process - which is considered the standard approach - is great for communicating the fundamentals. It allows you to focus on the most important and basic programming concepts you will need to understand, and is often presented through the output of a static, text-based console window. However, learning a language in this console vacuum can also be bland and boring, and can often lead to students becoming disengaged entirely.

Students are more likely to remain engaged by learning how to build web and Windows applications from day one in tandem with learning the C# programming language. Therefore, these lessons are designed to keep you engaged with the process of learning C# fundamentals by presenting it through a series of lessons aimed at creating simple, browser-based, web applications using ASP.NET.

In the first lesson, you will learn how to install your main programming tool (Visual Studio) that you will be using throughout these lessons. Right after that you'll be building your first dynamic web application, albeit a simple one, that you're going to use as a launching pad to learn more about the C# language. Along the way, you will also be learning a bit about ASP.NET - specifically ASP.NET Web Forms - which is one of Microsoft's web development technologies.

ASP.NET Web Forms is a combination of several technologies bundled into one cohesive whole. It's C#, HTML, ASP.NET, the Application Programming Interfaces, the Runtimes, along with Visual Studio that make generating ASP.NET code possible. Simply put, our main focus will be on learning C# fundamentals, while the side focus will be on learning ASP.NET. Once you understand the C# fundamentals, you will then be able to delve far deeper into ASP.NET, which will also be made easier having come to understand HTML, CSS, and other web related technologies.

It's important to note that the best way for you to learn in this series is by following along, typing each exercise by hand, and taking pause to review what you have learned in order to make sure you understand a given concept. A large part of the learning process is simply familiarization through muscle-memory and getting a sense of the common workflow: accessing menus, writing code, and so on. If you ever get stuck at any point in this process, feel free to review the lesson material several times in a row, if need be, in order to work through the kinks until the methodology begins to feel like second nature.

You will be able to test and track your progress by taking the supplementary challenges. Keep in mind that a solution to this challenge will be available to you if you find yourself getting stuck on a given challenge. However, you are encouraged only to consult the solution to help yourself become unstuck. The point of the challenges are not just to find the answer elsewhere in order to move forward, but to build valuable problem-solving skills you by trying to work through the problems on your own.

Another learning tool that will be presented to you throughout these lessons - to help you cement in your mind the concepts you are learning - is a cheat sheet that you can use for quick reference. The cheat sheet will usually contain common information at a glance, such as proper syntax for language elements, and workflows.

002

# Installing the Tools You Will Need

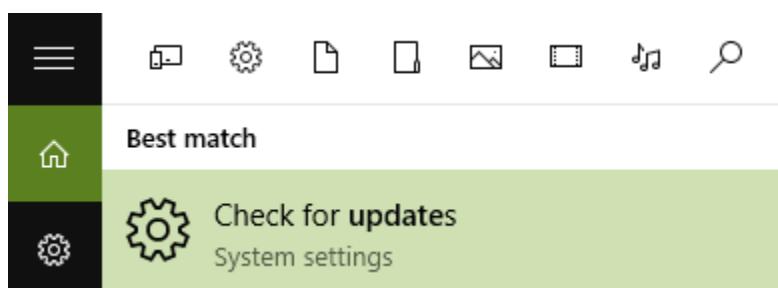
In this lesson, you will learn how to download and install Visual Studio. However, if you already have a version of Visual Studio 2015 installed, you can safely skip these steps and continue on with the next lesson where we build our first web application.

The great news is that you don't have to spend a single penny to get started in building web applications, or learning C#. With the introduction of the free "Community" version of Visual Studio, you have access to everything you need in order to get started building applications on the Microsoft platform. This free version of Visual Studio is essentially the same package that you would get with the paid Professional edition - giving you full access to the same selection of extensions and toolsets. The only differences between the free and paid versions are in the licensing guidelines. Simply put, if you are using Visual Studio within a large team project, or an enterprise business, you are required to purchase the Professional license, otherwise, as an individual or as a part of a small team you can enjoy all of the benefits found in the paid version by using the free version. It was a great move by Microsoft designed to introduce developers into the fold.

## Prepare Your Computer for Visual Studio

Before proceeding with the installation of Visual Studio, you should first make sure that your OS is up to date by downloading and installing any Windows Updates that are available to you. Keep in mind that you can use Visual Studio on either Windows 7, Windows 8 or Windows 10, but in these lessons we are going to assume that you are running the latest Windows OS (which at the time of this writing would be Windows 10).

Assuming that you're using Windows 10, there is a way to force Windows to update itself. Type "windows update" in the search bar located in the lower left corner. Then click on "Check for updates":



The next step is to open up the Microsoft Edge, or Internet Explorer, browser and navigate to

<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>

This will take you to the download page for the current version of Visual Studio Community. However, it may be even easier to simply type into your search engine the search terms:

**"Download Visual Studio Community"**

Once you have downloaded the installer, go ahead and run it. The installer itself is very small but when it runs it will grab everything it needs from Microsoft's servers in order to completely install Visual Studio locally onto your computer.

You will need to have a Microsoft account to complete the installation process, so if you do not yet have one you will need to sign up for one at:

<http://outlook.live.com>

## Installation Errors

That should cover the entire installation process for all of the tools you will need going forward. However, if you happen upon an installation error you have some options available to you in order to diagnose and remedy the issue. Often, a simple reboot of your computer is enough to fix everything. If that doesn't fix the issue, first try to find the solution by using a search engine, making sure to include in your search query the exact error message or error code that you encountered.

If your error persists despite all of your web searching, or you feel that your error is best handled by an actual human being, then you may want to ask for help on popular question/answer forums such as:

<http://msdn.microsoft.com/forums>

<http://www.stackexchange.com>

Keep in mind that there are helpful people in the community who are called "Microsoft Most Valuable Professionals" (MVP's). They are awarded this title as an acknowledgment of their dedication to helping people - such as you - solve problems. To make their job easier, make sure that your question includes the entire error message, including any numerical codes that you don't understand. Also, be sure to include the operating system that you're currently using and anything other details that are pertinent. Also, be sure to mention everything you have tried so far, so that steps that previously failed are not repeated.

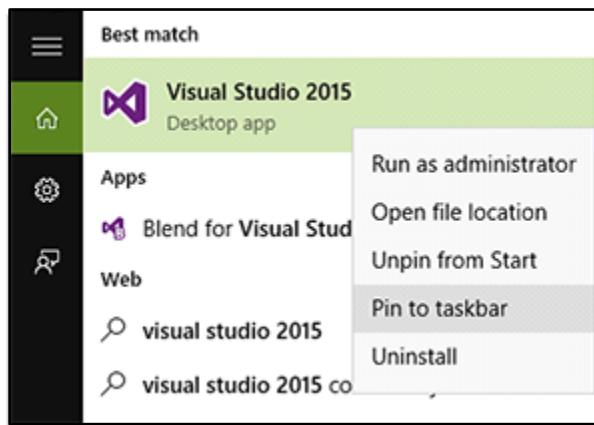
003

# Building Your First Web App

Once Visual Studio is successfully installed, go ahead and launch it in order to build your very first ASP.NET web application using C#. If you are using Windows 10, you can find Visual Studio by typing in the search textbox beside the Start Menu icon:



A good idea would be to right-click and select "Pin to Taskbar" as a convenient way for you to launch the application from the taskbar within Windows:



## Step 1: Getting Familiar with Visual Studio

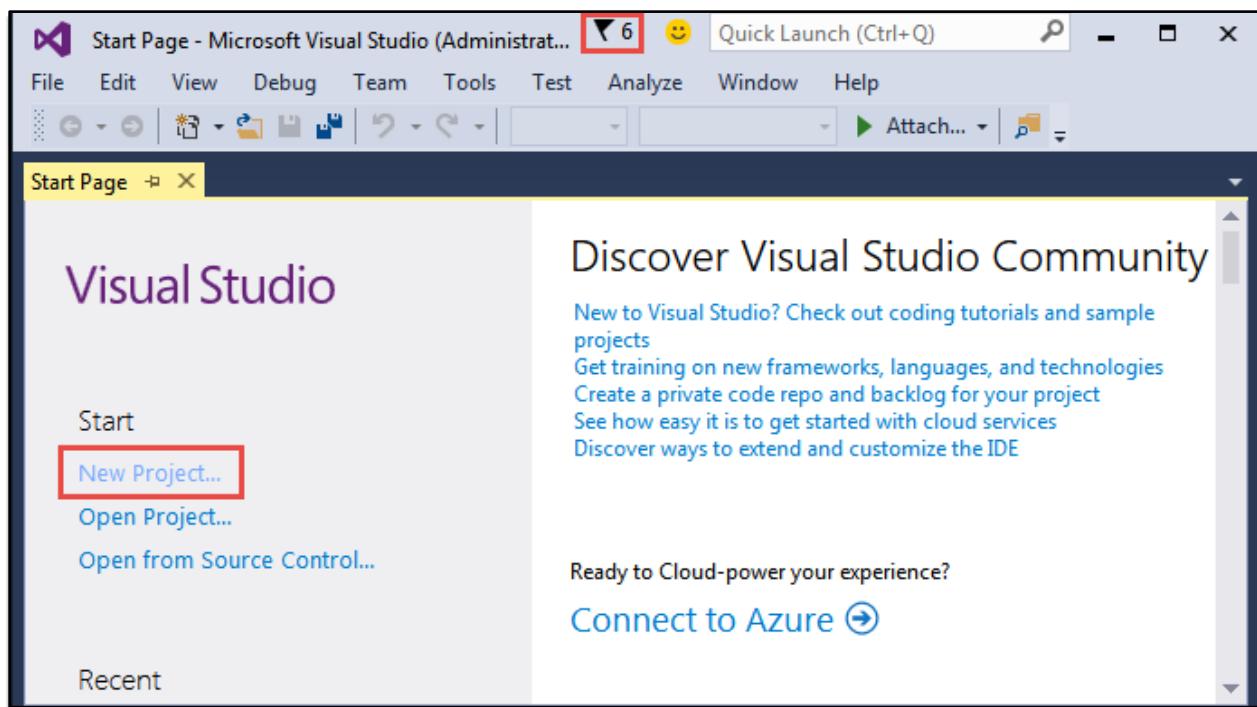
Once Visual Studio launches you will, yet again, be lead to an account login prompt. This second login is for synching up settings from other installs of Visual Studio that you may have signed into previously. This includes settings such as fonts, colors, window placement, and so on.



**Bob's Tip:** It may appear, at first glance, that Visual Studio, C#, and ASP.NET are inextricably linked together. However, that is not necessarily the case. The way you can think of it is that Visual Studio is just a shell environment that lets you edit and manage your projects, which in this case is made up of C# and ASP.NET files.

After logging in you will see a "Start Page" that has a few interesting features. The largest part in the middle is dedicated to providing news related items or links into MSDN (Microsoft Developer Network). This news section features valuable resources, such as, articles and videos that can help you learn more about programming in Visual Studio and keep up to date with the latest features.

On the left-hand side is a convenient way to create a new project, open an existing project or access a list of recent projects, which is a list that will grow larger as we add new projects over the coming lessons. One thing you should keep an eye out for is you might see a little flag in the upper right-hand corner that holds important notifications. If you click it, you will see whether or not there are any suggested updates that you should apply. In specific to this series of lessons, it's a good idea to make sure that Visual Studio - as well as anything relating to C#, and ASP.NET - is up to date:



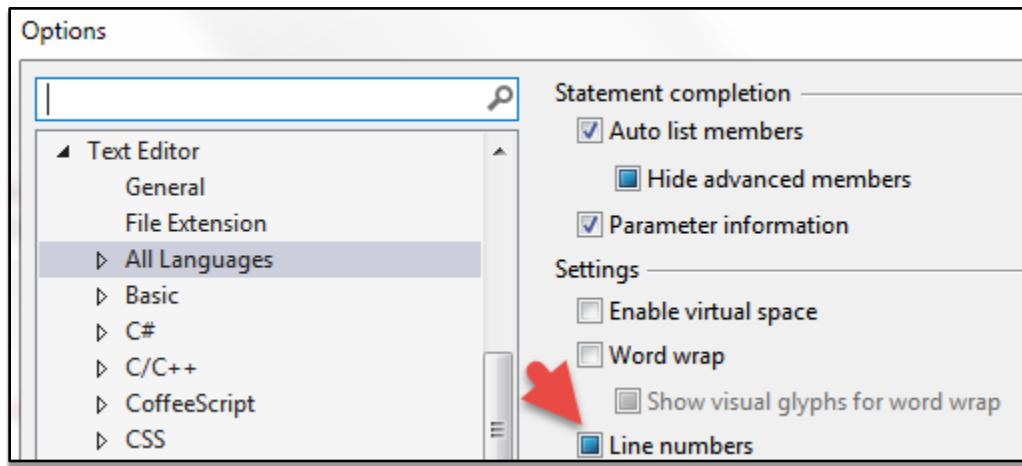
## Step 2: Setting Up Line Numbers and Font Size

A simple adjustment we can make, to make code lines easier to follow, is to enable line numbers for the left margin. To do so, go to the menu bar in Visual Studio and select:

**Tools > Options**

And from there, select from the left-hand menu:

**Text Editor > All Languages > Line Numbers**

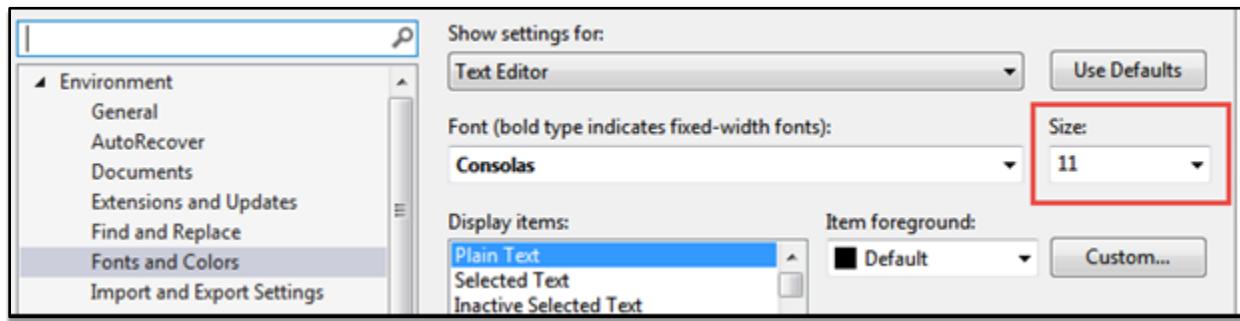


And now you should see line numbers appearing in the left margin:

```
1  using System;
2  using System.IO;
3  using System.Collections.Generic;
```

While you are in the Options menu, note that you can also adjust the font size under:

**Environment > Fonts and Colors**



## Step 3: Create a New ASP.NET Application

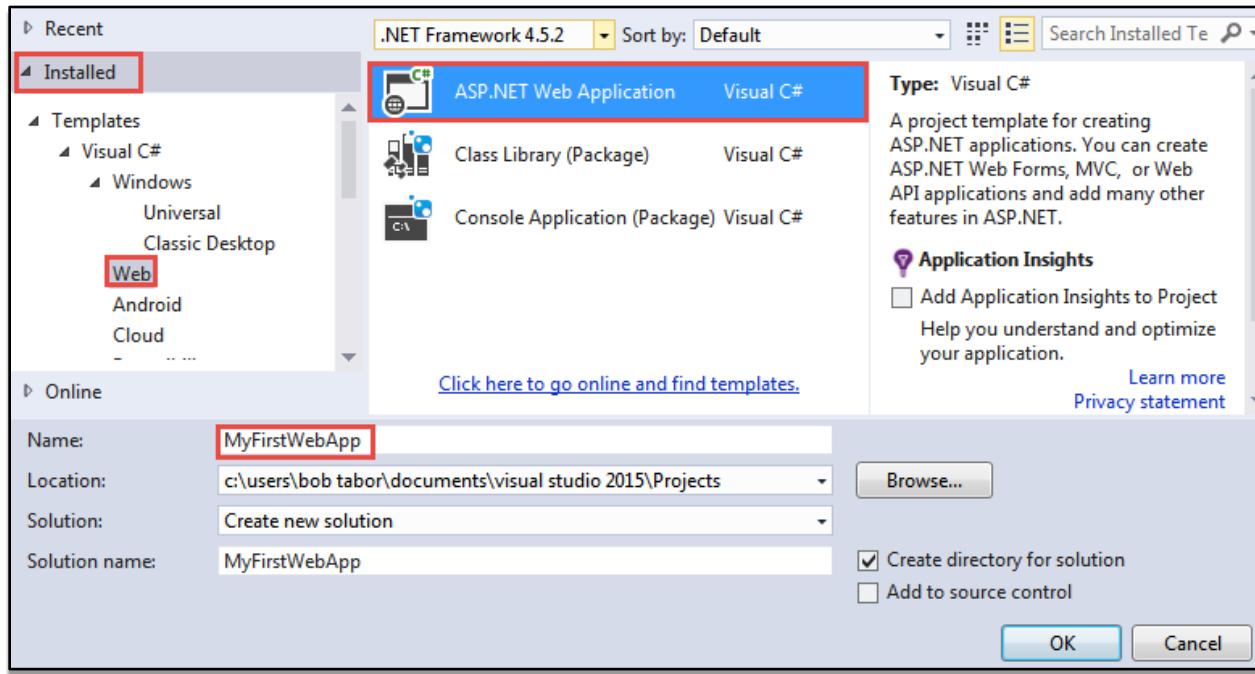
Once all of that is taken care of, go ahead and create a new project by either following the link on the start page or, alternatively, from the file menu:

**File > New Project**

Next you will see the New Project dialogue window. From here you will have to navigate the Installed Templates. Within the Installed Templates menu, continue until you find:

**Templates > Visual C# > Web**

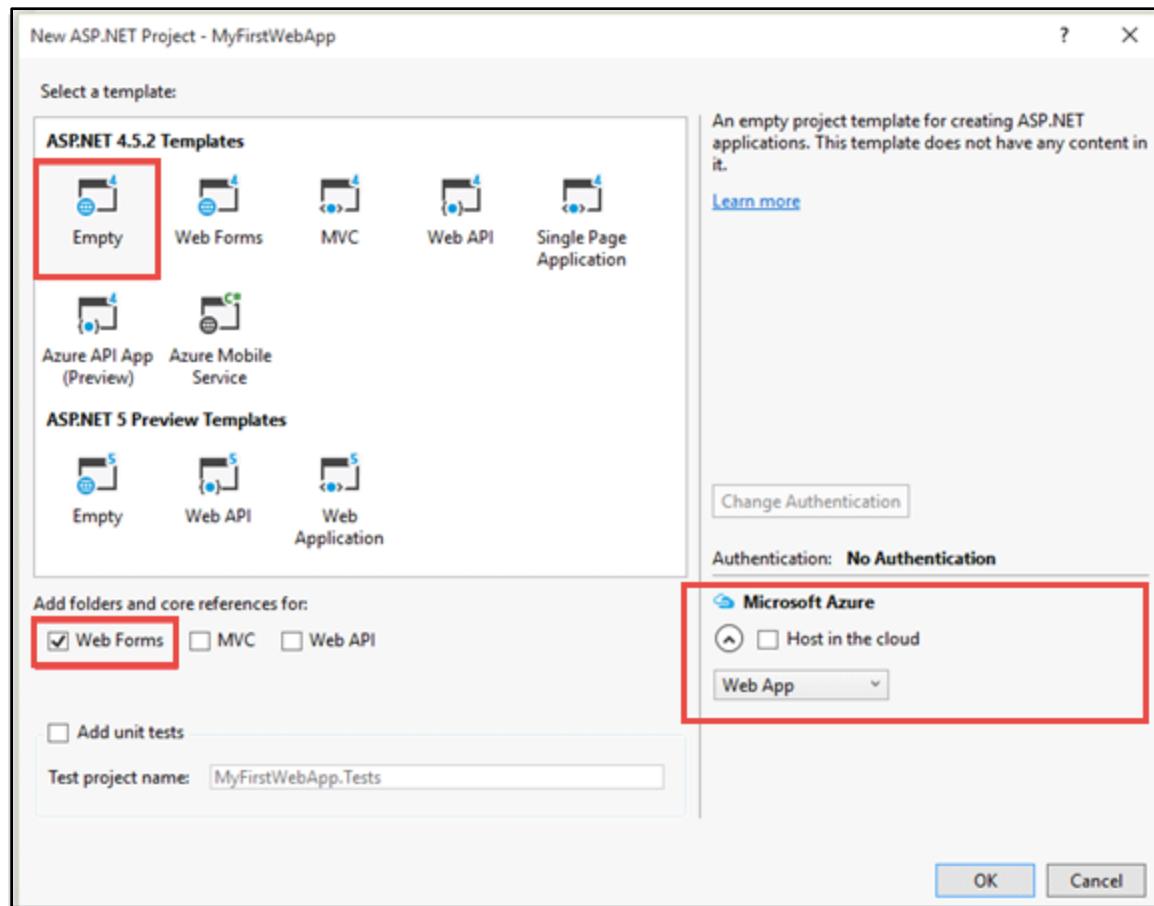
Just make sure that you are using Visual C# templates and *not* Visual Basic. Eventually, you will see an entry in the middle called "ASP.NET Web Application":



Now change the name of your project to "MyFirstWebApp," ensuring that no whitespace is used when naming the application. Also, be aware of the capitalization scheme shown. Go ahead and click "OK" in the lower right-hand corner to continue.

Note that there are several different types of ASP.NET applications, but in this series we will focus on Web Forms. However, in the next step be sure to select "Empty" in order to start from an empty

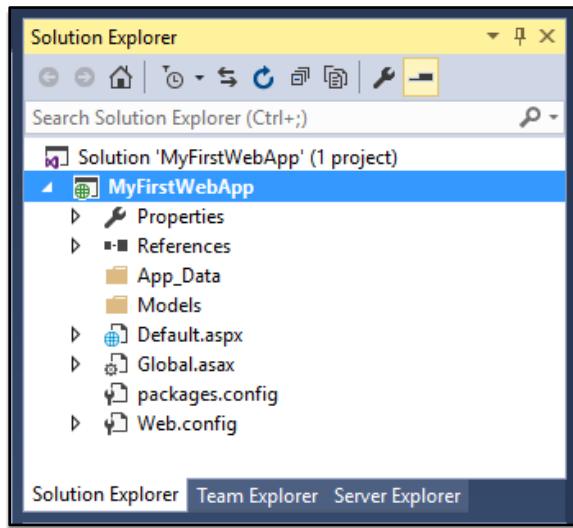
template, letting us add a Web Form later. You will also want to select the option to "Add folders and core references for Web Forms" and make sure that the "Host in the Cloud" option is unselected:



**Bob's Tip:** the process of creating, and setting up, new ASP.NET projects outlined here will be the basis for all of the other lessons. Be sure to refer back to this lesson if you have trouble remembering how to get your projects set up in future lessons.

## Step 4: Introducing the Solution Explorer

After you click the "OK" button, Visual Studio will go through the process of creating a new project for you. A project can be thought of as a collection of all the file settings, references, and assets (such as, images or sounds) that will be used to create your web app. Once your project loads, take a look at the "Solution Explorer" found on the right-hand side of Visual Studio:



The project's "Solution" is almost always found at the very top of the list within the Solution Explorer, which can be viewed as a hierarchy tree of files and references used in the project. Although you can have more than one project per Solution, most of our lessons will stick to just one. Note that if the Solution Explorer is not visible, you can make it visible by selecting from the Visual Studio Menu:

**View > Solution Explorer**

The first thing you will want to do within the Solution Explorer is right-click the "MyFirstWebApp" project (making sure not to select the Solution itself but rather the second entry below the Solution):



You will next see the context menu pop up, select from it:

**Add > Web Form**

Following that, you will be prompted to specify a name. Call it "Default," and then click "OK." This will create a page titled *Default.aspx* (the *.aspx* file name indicates that you are working with Web Forms). Once this launches, you will see some HTML mixed with ASP.NET declaratives:

The screenshot shows the Visual Studio IDE. On the left is the 'Default.aspx.cs' code editor window, which contains the following C# code:

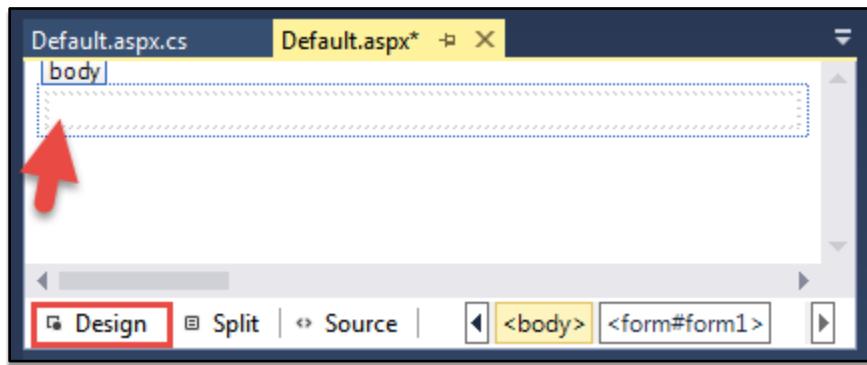
```
1 <%@ Page Language="C#" AutoEventWireup="true" %>
2
3 <!DOCTYPE html>
4
5 <html xmlns="http://www.w3.org/1999/xhtml">
6 <head runat="server">
7   <title></title>
8 </head>
9 <body>
10 <form id="form1" runat="server">
11 <div>
```

On the right is the 'Solution Explorer' window, showing a project named 'MyFirstWebApp' with files like 'Properties', 'References', 'App\_Data', 'Models', and 'Default.aspx'.

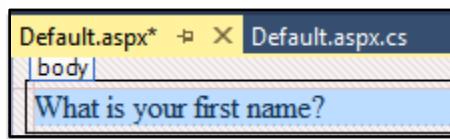
## Step 5: Using the Designer to Make Webpages

---

Ignore the background HTML/ASP markup for the Default.aspx file, for now, as you will be mainly working within the Design mode (or, the “Designer”). This provides a rough approximation of what your web page will look like once it’s running on the client’s web browser. The Designer is a convenient WYSIWYG visual editor that allows you to add markup to your page without having to access the HTML/CSS code behind it. Your initial goal, when first inside the Designer, is to add text and Controls to the page. You should see a dashed box at the very top with the body/div tag above it:



Inside of the dashed box near the body tag, type the question: "What is your first name?" with two spaces following the question mark so that we can insert a Control item:

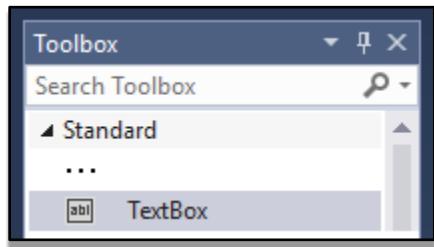


## Step 6: Insert Controls via the ToolBox

---

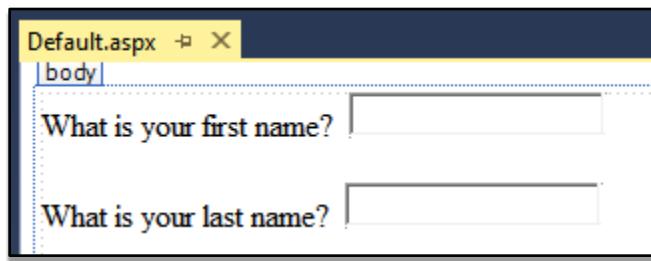
Now, navigate to the Toolbox and find a Server Control called "TextBox." If the Toolbox is not visible for you, you can make it visible by going to the Visual Studio menu and selecting:

**View > Toolbox**



**Bob's Tip:** in the upper right hand corner of the Toolbox, you will notice a little pin that allows you to enable or disable auto-hide for that window. Pinning and unpinning is very useful, especially if you find yourself accessing a menu item frequently. For now, pinning this menu will come in handy.

Double-click the TextBox option in order to insert it into the web page where your cursor was last at (the two spaces after the question). Now, making sure that your cursor is positioned after this TextBox, hit the enter key twice on the keyboard and type "What is your last name?" followed again by two spaces. Now, repeat the previous steps to insert another TextBox control after this phrase. You should now have something that looks like this:



Once again, position your cursor at the end of the last line and hit the Enter key two more times. Returning to the Toolbox you will find, near the top, an option that inserts a button inside of your web page:



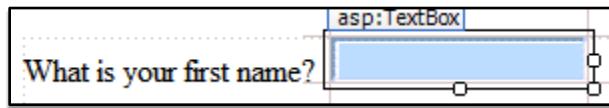
After the button is inserted, once again press enter twice and back in the Toolbox find, and insert, a Label Control from the ToolBox.



## Step 7: Customizing Controls via the Properties Window

---

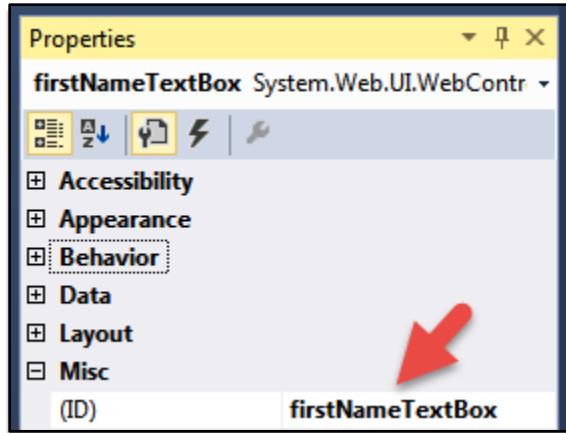
The next thing to do is set some properties for these Controls (which are actually called ASP.NET Server Controls). Go ahead and select the first TextBox:



And while it is selected go to its Properties in the right-hand pane. If you don't see the Properties pane go to the menu and select:

**View > Properties Window**

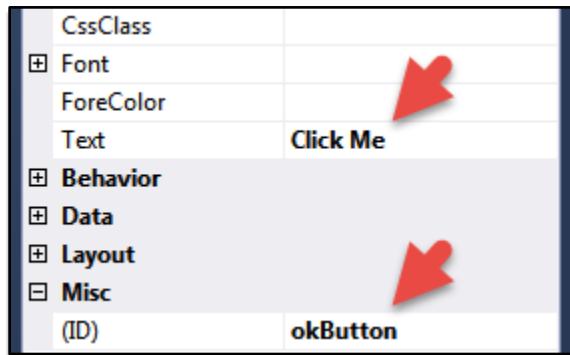
Alternatively, you can just press the F4 key to display the Properties window. In this window, change the programmatic ID for the first TextBox to "firstNameTextBox":



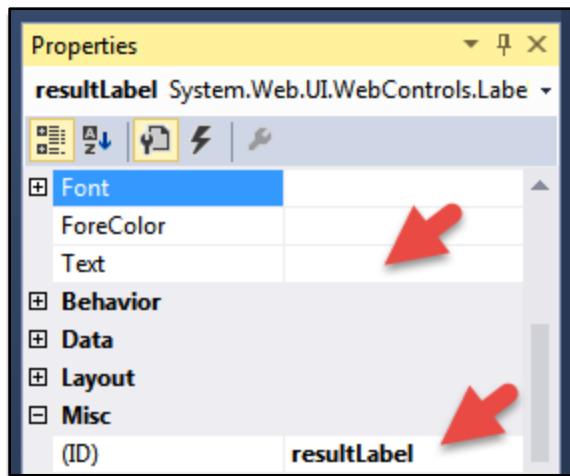
This name will let us keep track of what was entered in the TextBox, so we can then later reference it within code. While naming is somewhat arbitrary, a good rule of thumb to follow is to pick one that communicates the intent of the Control. Another good rule of thumb for naming is to use the capitalization convention shown, called camel-case (lower-case first letter, followed by upper-case first letters for each word, while using no spaces to separate the words). Next, select the second TextBox and set its ID accordingly, with "lastNameTextBox":



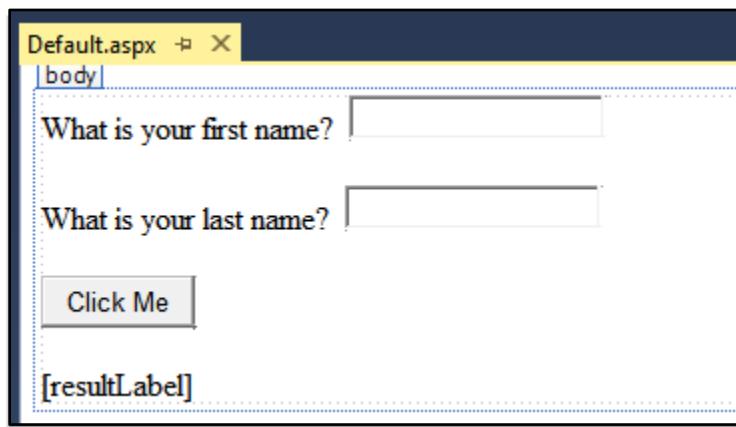
Now, selecting the button Control in the Designer, go to its Properties window and change its ID to "okButton," and also change its Text label (under "Appearance") to "Click Me" and now the button in the Designer should change as well:



And finally, select the Label control in the Designer and change its programmatic ID to “resultLabel” and also remove the value in the “Text” attribute:



Removing the text for the Label ensures that nothing is shown by default (this will instead be determined in code later). The part that says “[resultLabel]” will only be visible in the Design view, but will not be visible within the public-facing web page. At the end of all of this, the Design view should look like this:



## Step 8: Write C# Code to Determine Page Behavior

Now, double click the “Click Me” button and a new tab will launch for a file called *Default.aspx.cs*. In this file, you will see a bunch of C# code, with the cursor situated in between curly braces for a block of code called `okButton_Click`:

```
protected void okButton_Click(object sender, EventArgs e)
{
}
}
```

Within this set of curly braces type in the following code exactly as shown here:

```
protected void okButton_Click(object sender, EventArgs e)
{
    string firstName = firstNameTextBox.Text;
    string lastName = lastNameTextBox.Text;

    string result = "Hello " + firstName + " " + lastName;

    resultLabel.Text = result;
}
```



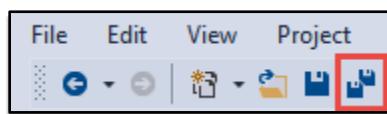
**Bob's Tip:** Beginners will often run into errors that usually trace back to incorrectly typed code. Double-check that your code follows along exactly with the lesson material. If you see red squiggly underlines it is an indicator that you omitted, or mistyped, something in your code. Unlike writing text messages and emails, coding is an exercise in precision!

## Step 9: Save and Run Your Application

---

The final thing you will want to do for this lesson is save all of your work (meaning all of the changes we have made to all of the files in our entire project). To do so, click on the “Save All” icon in the Visual Studio menu, or else select from the menu:

**File > Save All**



Now, let's see the result of all of our work by running the application. Click on the green arrow icon in the Visual Studio menu to run the application in your web browser of choice:



You can now test out the application in the browser by entering your information in the form, and see the result posted back to you after hitting “Click Me”:

What is your first name?

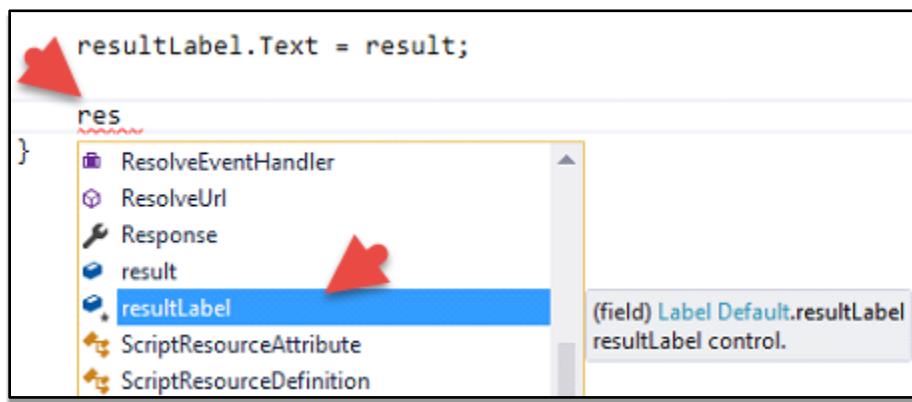
What is your last name?

Hello Bob Tabor

004

## Understanding What You Just Did

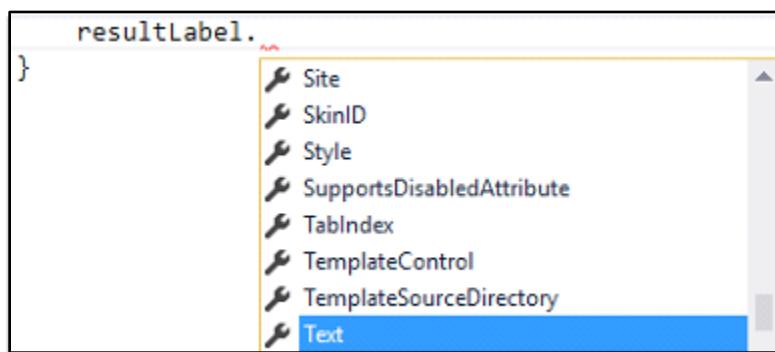
In this lesson, we will take a closer look at some key concepts we came across in the previous lesson. First, you may have noticed a handy tool called “Intellisense” when typing out code in Visual Studio. For instance, when you type out the words “result” you would see Intellisense pop up giving you a list of options for completing that bit of code:



### Step 1: Using Intellisense

---

You can then step through the options with your arrow keys until you find what you need, and insert it into your code. In this case, the code reference we’re looking for is the Server Control variable called `resultLabel`. If you hit a completion key on your keyboard (tab, space, or enter), it will autocomplete and write the rest of the variable name out for you. However, we knew ahead of time that we also wanted to access the `Text` property that is contained within `resultLabel`. So, you can instead hit the period key on the keyboard to input the dot accessor, giving you access to a new list of Intellisense options:



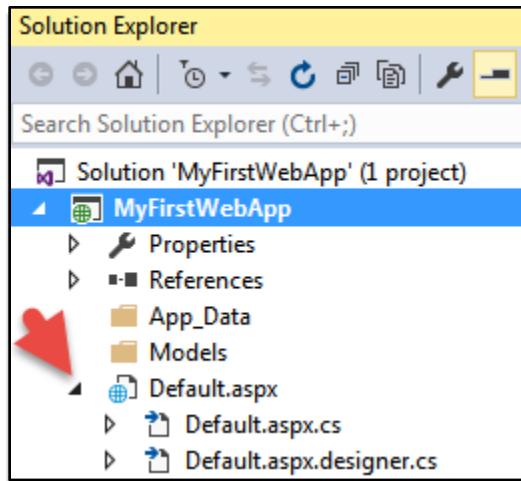
Now, with the Text property highlighted, you can hit the equals sign on the keyboard to complete that variable name and put the cursor in a position where you can assign whatever text you would want to that property.

As you see, Intellisense can become your best friend when coding in Visual Studio. It makes your typing quicker, and more accurate, while also narrowing down a list of options available to you based on the context of the coding element you are accessing.

## Step 2: Distinguishing Default.aspx vs Default.aspx.cs

---

The next thing to go over is understanding the relationship between the *Default.aspx.cs* and *Default.aspx* files within your project. These two project files represent two sides of the same coin, and are found by clicking on the arrow beside *Default.aspx* in the Solution Explorer:



As implied from this file hierarchy, the files ending in *.cs* (denoting “c-sharp” files) can be seen as representing the *code behind* the *Default.aspx* form. When you double-click the *Default.aspx* file - and look at its “Source” view - you will even find this referenced on the first line of code:

```
1 <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
```

And when you double-click the *Default.aspx.cs* file you will see this “CodeBehind” for the *Default.aspx* form:

```
namespace MyFirstWebApp
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void okButton_Click(object sender, EventArgs e)
        {
            string firstName = firstNameTextBox.Text;
            string lastName = lastNameTextBox.Text;

            string result = "Hello " + firstName + " " + lastName;

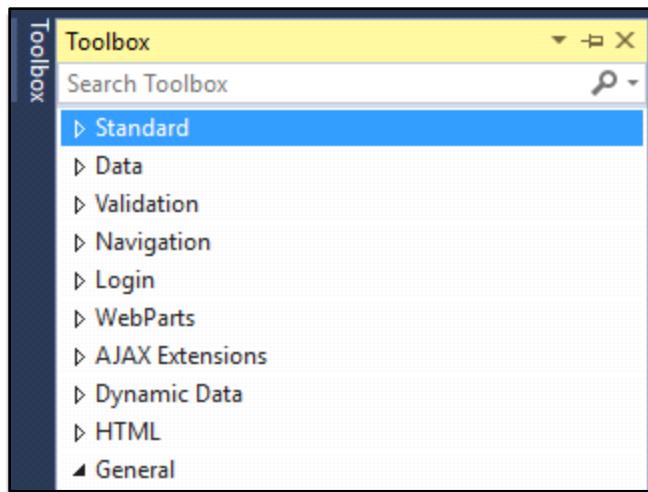
            resultLabel.Text = result;
        }
    }
}
```

When running on the web server, this code will be responsible for handling the back-end logic for the form, while the *Default.aspx* code handles what's delivered to the front-end and is immediately viewable to the user. It's worth noting, however, that once all of this code becomes compiled down to a machine readable format, both of these files will merge into a single file called *Default.aspx* (more on this later).

## Step 3: Server Controls in Detail

---

Turning our attention now to Server Controls, you will notice that the Toolbox contains an exhaustive list of Server Controls organized by category:



We've been working with several Server Controls in the Standard category, but some other common Controls available here are:

- **Data Controls** – bind to a data source like a database.
- **Validation Controls** – ensure end-users are providing inputs in an acceptable format.
- **Navigation Controls** – for creating menus, and site navigation.
- **Login Controls** – handle securing a site with password protection and retrieval.

While these stock Server Controls are useful for general use, there are also entire libraries of purchasable Server Controls that provide additional – and often very specific – features. Every Control has one thing that it does well that distinguishes it from all of the other controls, or else it wouldn't exist. As we saw in the previous lesson, the TextBox Control that we used is good at receiving alphanumeric characters from the end-user, while the Button Control that we also used is great for allowing the end-user to click on it in order to take some action (such as processing the form). And, the Label Control is good at displaying information, such as the information the end-user provided and executed using those two other Controls.

## Step 4: Referencing Server Controls in Code

---

Now, the execution of all of this occurs within the code behind the Button Control. When you double-click the button in the Design view it takes you to this code block called `okButton_Click` which is enclosed within a set of curly braces (each opening/closing set of curly braces is referred to as a “code block”):

```

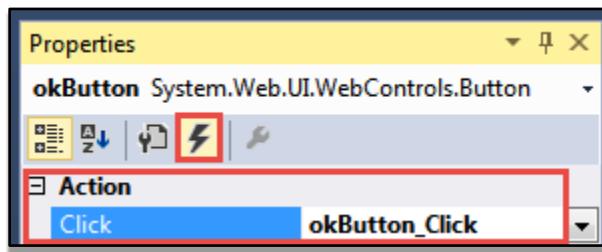
protected void okButton_Click(object sender, EventArgs e)
{
    string firstName = firstNameTextBox.Text;
    string lastName = lastNameTextBox.Text;

    string result = "Hello " + firstName + " " + lastName;

    resultLabel.Text = result;
}

```

In this case, this code block is labeled based on the name we chose for the Button ID, with “\_Click” appended to the end of it. Going back to the Properties window for the button you can see the reference – by clicking on the Events icon (the lightning bolt beside the Properties icon):



This, essentially, binds the `okButton_Click` code block together with the act of clicking on the Button; when the button is clicked, it executes the code contained in `okButton_Click`. Whereas the fields and properties we previously dealt with described the appearance of the Button, this click-action Event – along with its code block(s) – describes what the button *does*. As such, you could consider this code block as the *event handler* for this button, in that it *handles* what the button executes in the *event* of a click.

## Step 5: Understanding the okButton\_Click Event

---

Let's now turn to understanding what this particular code block does. You can see that we are using the names/programmatic IDs for several of the Server Controls:

- firstNameTextBox
- lastNameTextBox
- resultLabel

The first two references take whatever text the user typed in, each contained in a preset Property called `firstNameTextBox.Text` and `lastNameTextBox.Text`, respectively. And then the values for each of these text Properties become stored into separate variables, labeled `firstName` and `lastName`:

```
string firstName = firstNameTextBox.Text;
string lastName = lastNameTextBox.Text;
```



**Bob's Tip:** these variables can be thought of as “buckets” that are just big enough to store information of the type specified (before the named labels). In this case, that type is of type `string`, meaning a “string of characters.” Picture strings as a set of alpha-numeric characters strung together, one after the other, as though on a clothes line.

You declare a variable by first specifying its type, and then its label (so we can reference it by name, somewhere later in code). This tells the computer to set aside in memory a “bucket” just large enough to store that information for later use, as we did here:

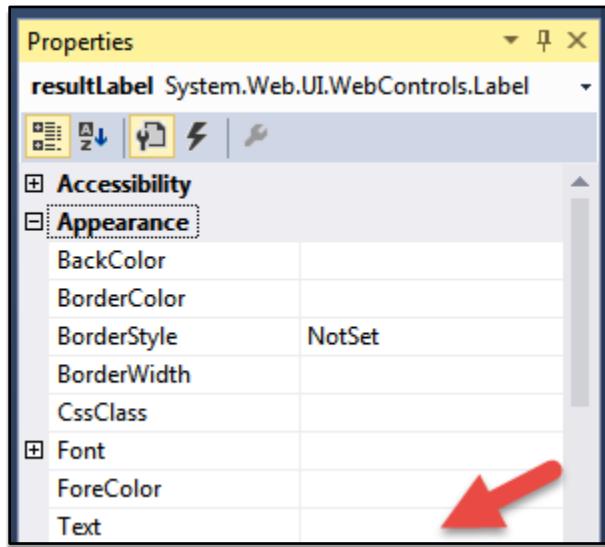
```
string result = "Hello " + firstName + " " + lastName;
```

On this line, we are creating yet another “bucket” called `result`, as it’s the result of combining the values contained within `firstName` and `lastName` along with the literal strings for “Hello ” and an empty space for nicer formatting. The plus sign, referred to as the “concatenation operator,” essentially glues together all of these separate string values, before dumping it into the `result` bucket by using the equal’s sign, or “assignment operator.”

And, finally, we simply set the `resultLabel.Text` property with the contents of `result`:

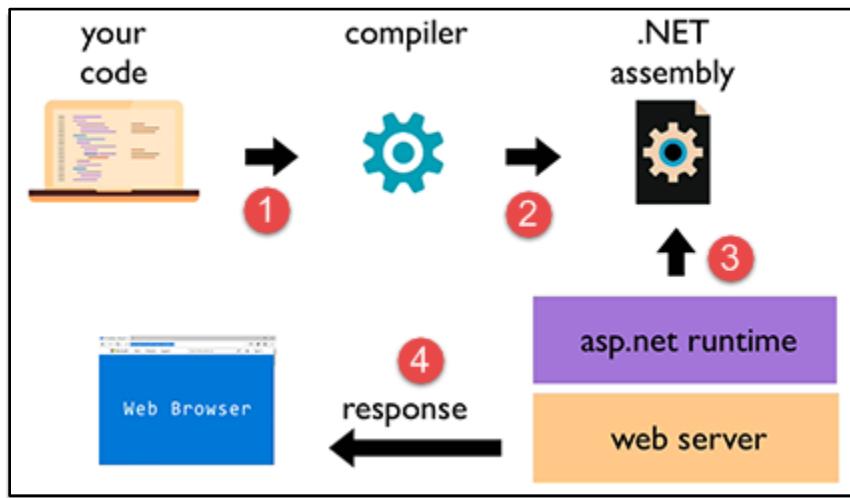
```
resultLabel.Text = result;
```

Recall that during *design time* (before the application runs) `resultLabel.Text` was set, by default, as an empty string. However, the code block that references this property is, nonetheless, able to manipulate the value at *runtime*.



## Step 6: Understanding Design-time vs Runtime

The process by which this design-time/human readable code becomes compiled down to a runtime/machine-readable format, and ultimately displayed for the end-user, can be visualized as follows:



1. Compile human-readable code, to a machine readable format.
2. Compile this code into a .NET Assembly - typically a single file with a `.dll` file extension in the case of a web application - that will be used and referenced whenever we want to serve up a

**web page.**

3. A web server will launch ASP.NET, that in return calls our .dll assembly into play, requesting a definition for *default.aspx*
4. Our code will execute and ASP.NET will give it back to the web server. The web server, in return, will send it back on its way to the end-users web browser.

This entire compilation process gets kick-started whenever you click on the button that sends the results to the browser:



How is this able to display in the browser without first having a web server running the compiled .NET assembly? The answer is that you do, in fact, have a web server running on your computer. Visual Studio runs a mini web server called IIS Express (Internet Information Services Express), which you will notice pops up as a Desktop tray icon, and is meant for local development-only:



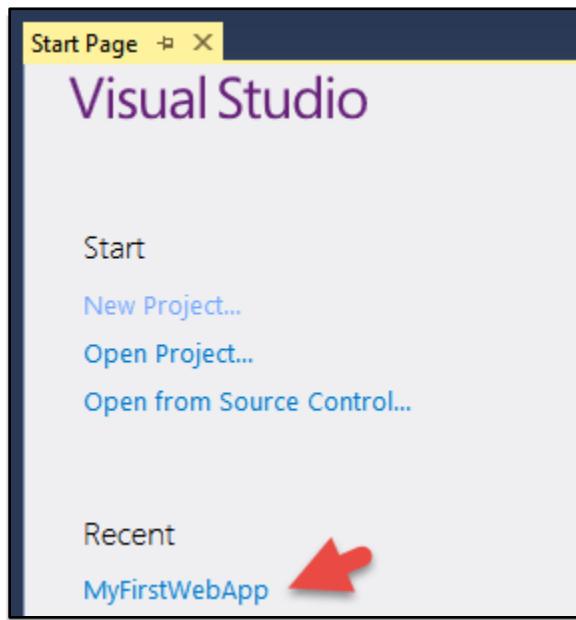
In the next lesson, we will cover a bit more on Server Controls, and how to style web pages, before delving deeper into C# language fundamentals.

005

# Working with Projects in Visual Studio

In this lesson, we'll talk about working with Solutions and projects, as well as the relationship between the two concepts. You will learn how to find your Projects and Solutions on your hard drive so that you can continue working with them at a later date.

When you re-open Visual Studio, you will notice that the last project you were working on is available via a quick-link, under the heading titled "Recent":



Eventually, this list will grow and push older projects off of the list, unless you were to pin the project (hover your mouse over the project name to see the pin and click it to pin it downwards).



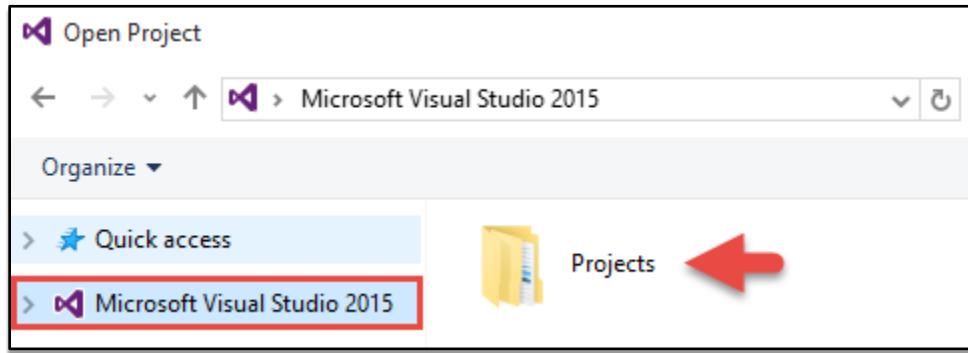
You can also access your recent projects by accessing the Visual Studio menu:

**File > Recent Projects and Solutions**

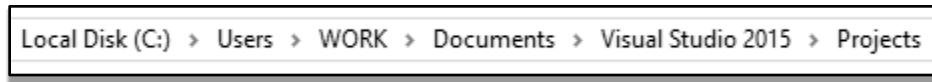
If your project is no longer visible in your list of recent projects, you can locate it by going to the Visual Studio menu, and then manually searching through your Visual Studio projects folder:

**File > Open > Project/Solution**

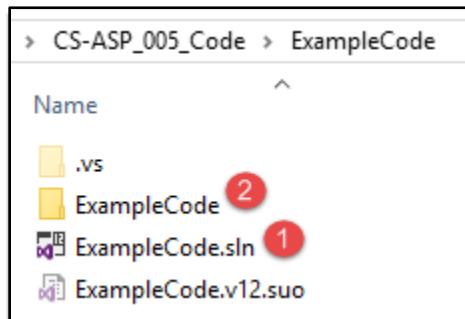
You will find all of your projects in the “Projects” folder after you click on the “Microsoft Visual Studio 2015” tab on the left:



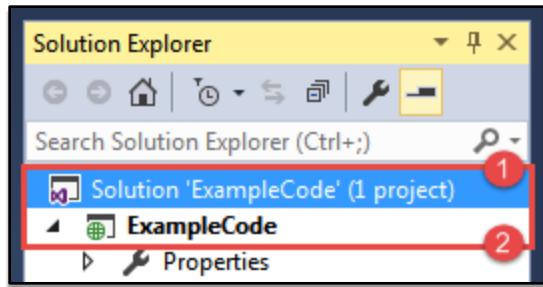
You can also manually browse your computer to find the Projects folder for your Windows user account:



A project folder will contain one or more projects managed within a “Solution.” The Solution is designated with a **(1)** .sln file extension, while the project folders associated with the Solution can have any name but by default are **(2)** named the same as the Solution:

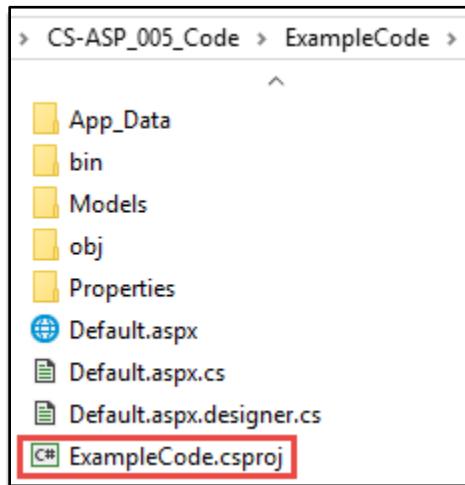


When you click on the Solution file, your main Solution launches in Visual Studio along with all of the projects that are associated with it. If you look at the Solution Explorer you will find the hierarchy reflected as the Solution **(1)** at the top-level and containing the project(s) within it **(2)**:



**Bob's Tip:** It may not appear immediately obvious why anyone would want more than one project per solution. However, when dealing with application architecture there's a need to sometimes separate code based on individual responsibility: each project handling a specific responsibility within the context of the broader Solution.

Back in the project folder, you'll find the project file with the `.csproj` extension. Whereas the `.sln` file keeps track of all projects in the solution, the `.csproj` file keeps track of all of the files, folders, settings and references that make up the individual project:



006

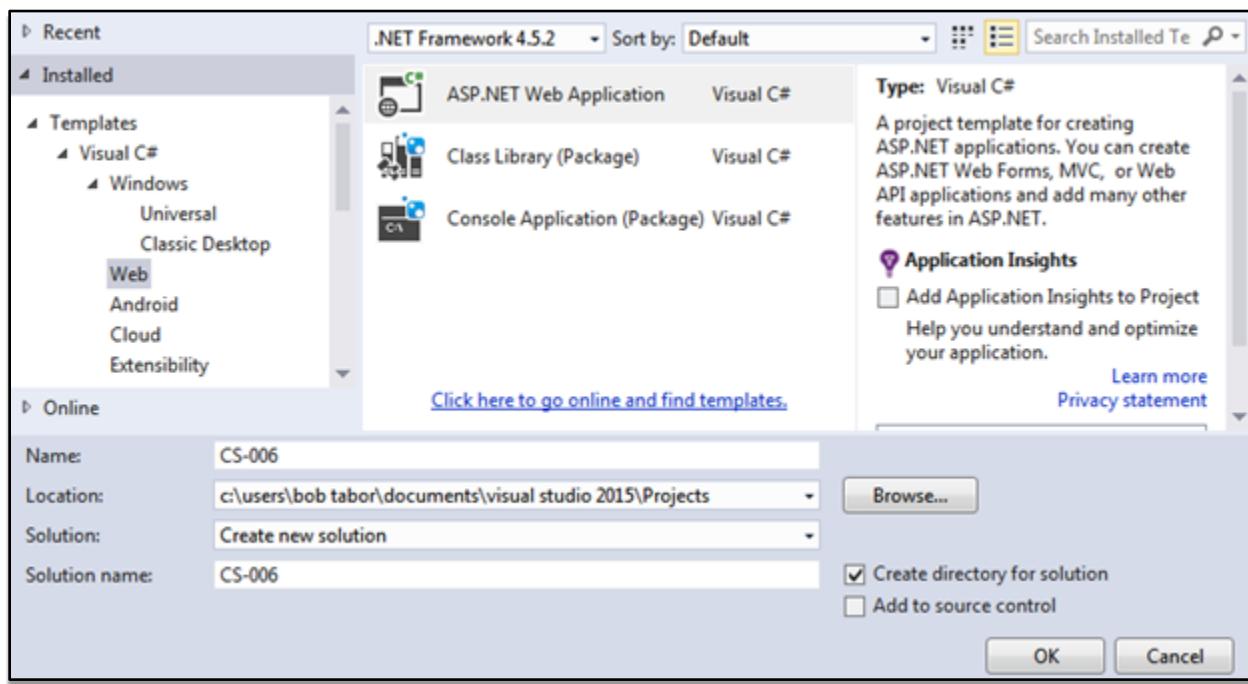
# Simple Web Page Formatting in Visual Studio

In this lesson, we'll look at a few Visual Studio tools that will help you build better web pages without having to know HTML5, CSS3, and other frontend user interface technologies. Having said that, this is not going to be a long-term solution for you if you desire to be a well-rounded web developer. You will eventually have to learn HTML5, CSS3 and other web-related technologies in order to progress as a web developer. However, demonstrating these tools will expose you to some handy features in Visual Studio and will allow us to do a bit of design upfront without having to confront all of the details of HTML or ASP.NET just yet.

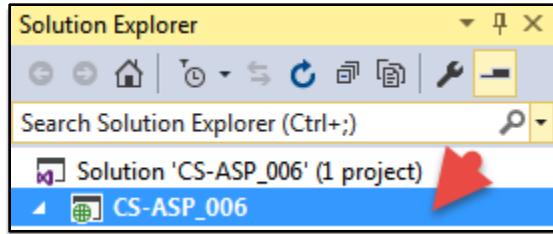
## Step 1: Start a New ASP.NET Project

---

Let's get a fresh start by creating a new ASP.NET project, calling it "CS-006":



At the next screen, select an "Empty" template, and checkmark the "Web Forms" box to add it your Project. Once the Project loads, add a new .aspx file to the project by right-clicking on the project name in the Solution Explorer:

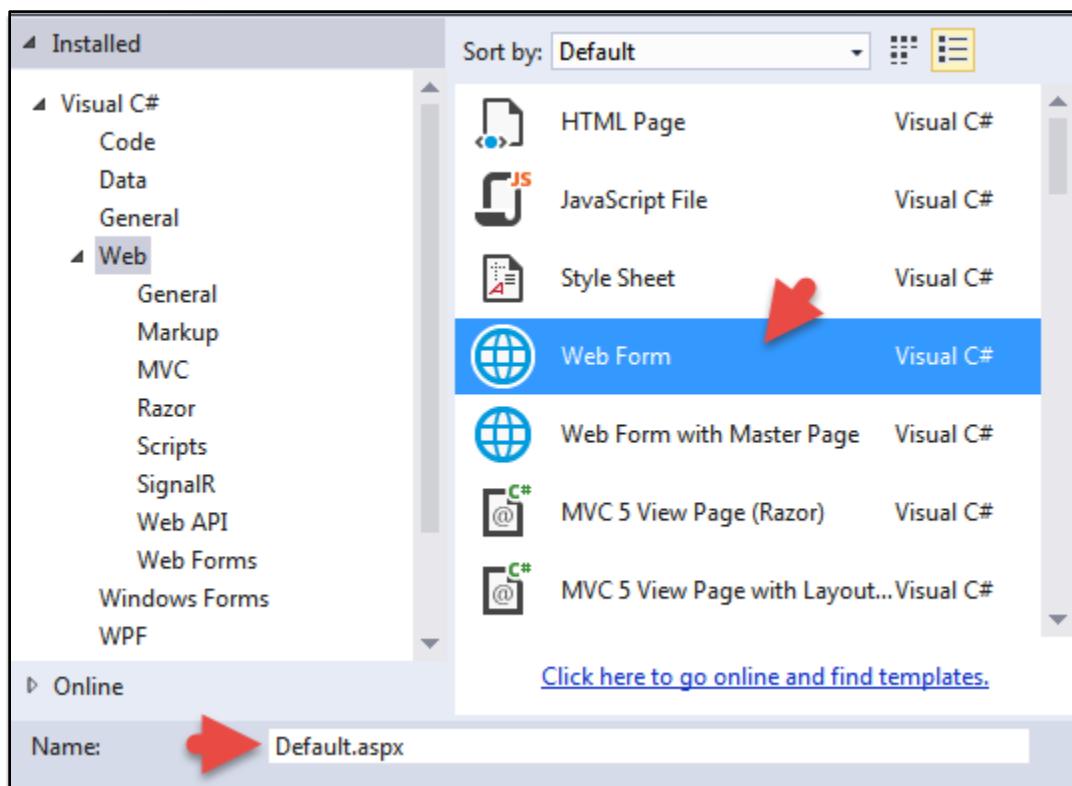


And choose from the menu options:

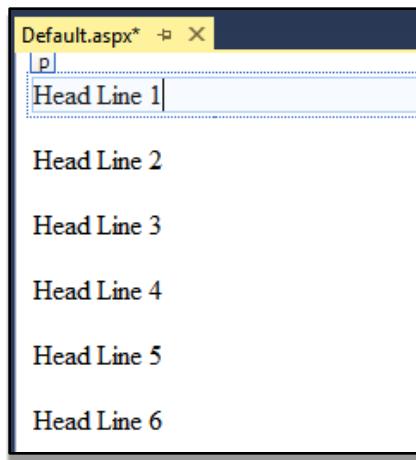
**Add > New Item**

## Step 2: Add a Default.aspx Web Form

In the “Add New Item” dialogue, select “Web Form” from the list of options, and name it “Default.aspx”:



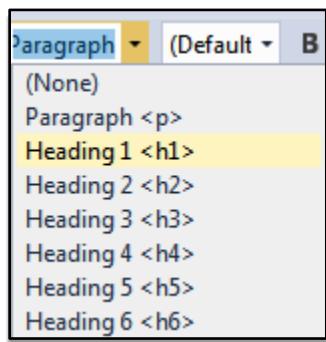
Once it loads within your project, click on the *Default.aspx* file in the Solution Explorer to open it up in the main window, and click on “Design” to go into Design View. Once there, click in an empty space within the Design View and enter the text “Head Line.” Do this six times and enumerate each one. After you are done, it should look like this:



## Step 3: Add HTML Heading Tags

---

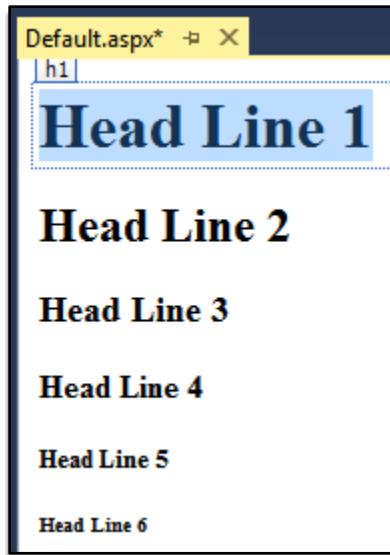
The purpose of this is to show you how you can access HTML formatting options directly in the Designer, without even having to access the background markup. Each of these lines are formatted as a paragraph by default but we can change the HTML tag for each of these elements within Visual Studio by accessing the tag options directly within the Formatting toolbar menu:



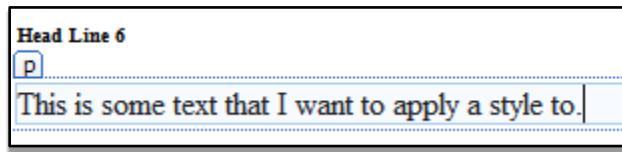
If you do not see the Formatting toolbar, make sure it is visible by checkmarking it within the Visual Studio menu:

**View > Toolbars > Formatting**

Go ahead and change each element with the heading tag that matches each numbered "Head Line." In other words, <h1> for Head Line 1, <h2> for Head Line 2... etc. The result will look like this:



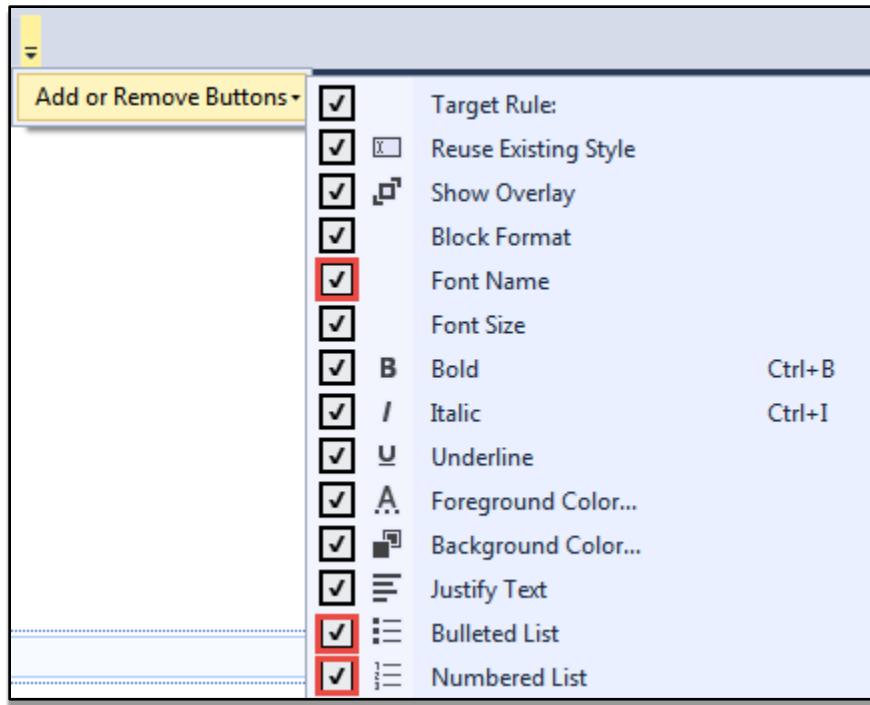
Next, add another paragraph tag element below “Head Line 6” and write inside of it “This is some text that I want to apply a style to”:



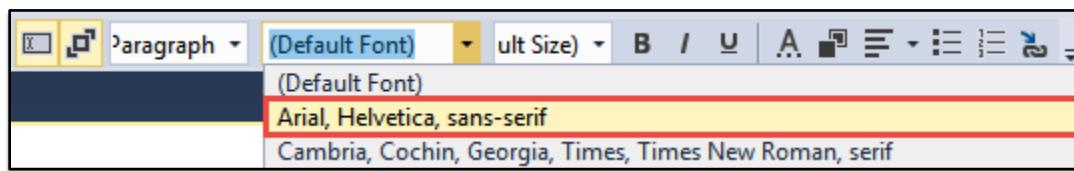
## Step 4: Apply Customized HTML Font and Color Styles

---

To create a unique styling for this element, access the formatting options via the Visual Studio Formatting toolbar. Click on the downward arrow beside the other formatting options and then click on “Add or Remove Buttons.” Now, check “Font Name,” “Bulleted List,” and “Numbered List”:

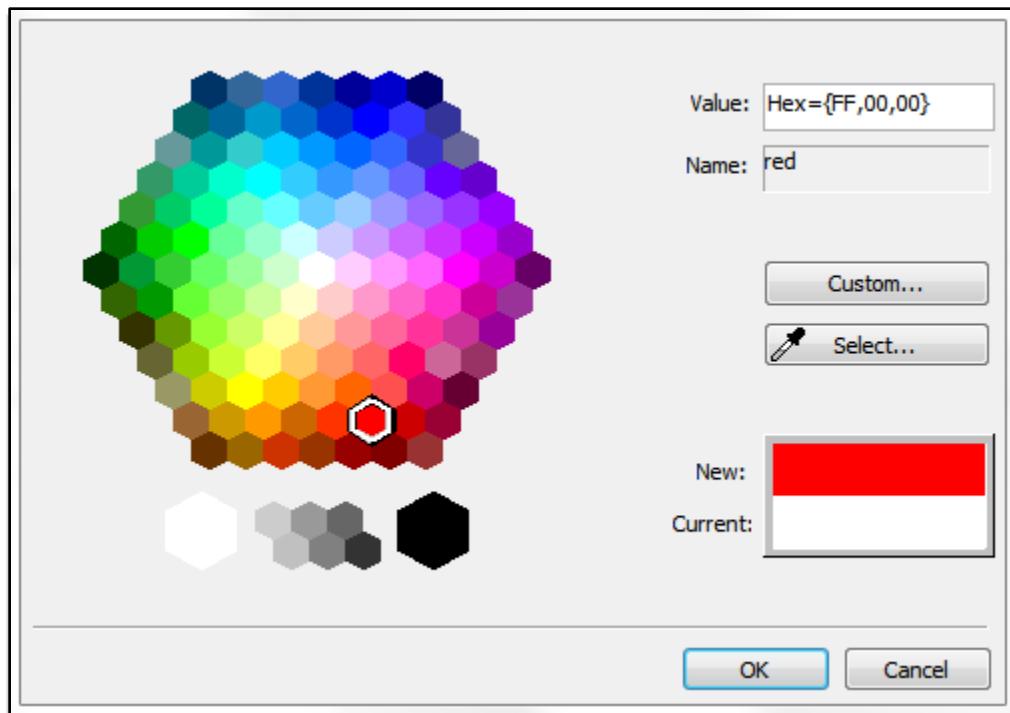


With this new set of formatting options available to you, make sure the text is selected. From the Formatting menu change the default font to “Arial, Helvetica, sans-serif”:



Now, select a part of the text and change its color by accessing the text Foreground Color option:

The screenshot shows a text editor with a selected paragraph of text. The text contains the sentence "This is some text that I want to apply a style to.". Below the text is a toolbar with several buttons. One button, which is the foreground color button (a small square icon), has a red arrow pointing to it, indicating it should be clicked.



Note that you can use the “Custom” or “Select” color options to use a customized color of your choosing, but here we are just selecting a simple red color so the text stands out.

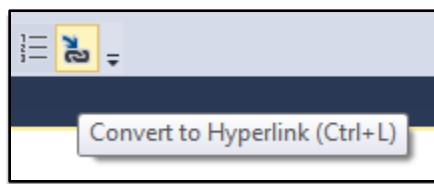
## Step 5: Add a Hyperlink

---

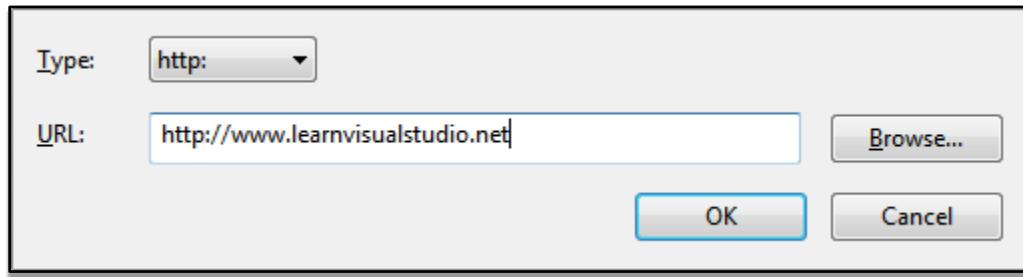
Now, let's add a hyperlink by first writing some text inside a div below the previous paragraph:

This is some text that I want to [apply](#) a style to.  
`div`  
[Add a hyperlink.](#)

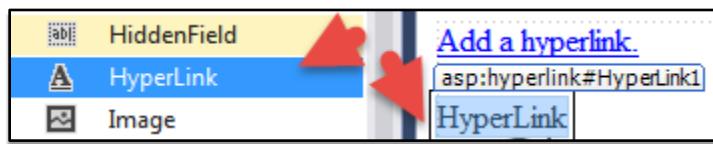
With “Add a hyperlink” highlighted, then click on the “Convert to Hyperlink” menu item in Visual Studio:



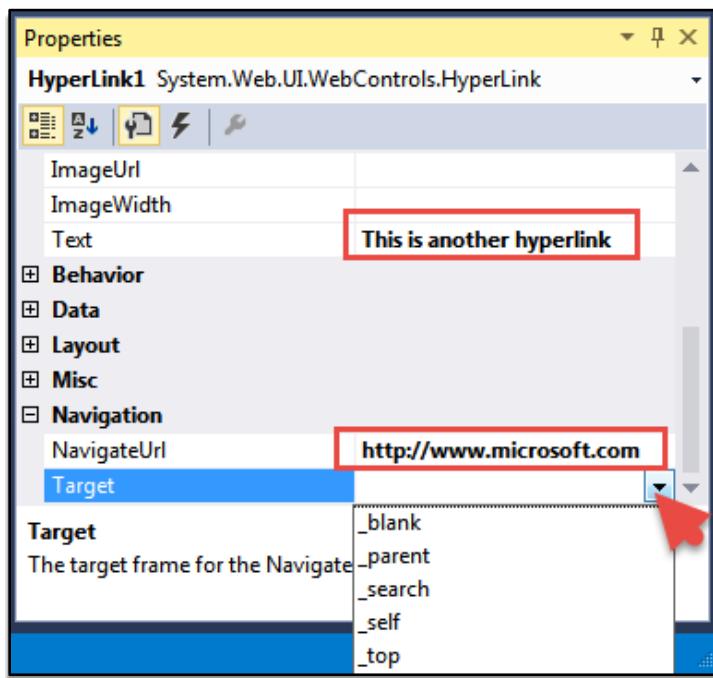
In the resulting dialogue box, note the protocol options for the link beside “Type,” as well as a field for adding the URL link itself. Choose “http:” for the protocol and add a link as shown here. Also, note that the “browse” option lets you reference another page from within the project if you want to link to it:



There's another way to add a hyperlink, and that's by inserting it as a Server Control. Although you would normally do this only when the link needs to dynamically change at runtime, we will do so here for demonstration purposes. Write some text under the previous hyperlink and from the Toolbox insert the HyperLink Server Control:



And you can set the link text, URL and other properties via the Properties Window (F4):



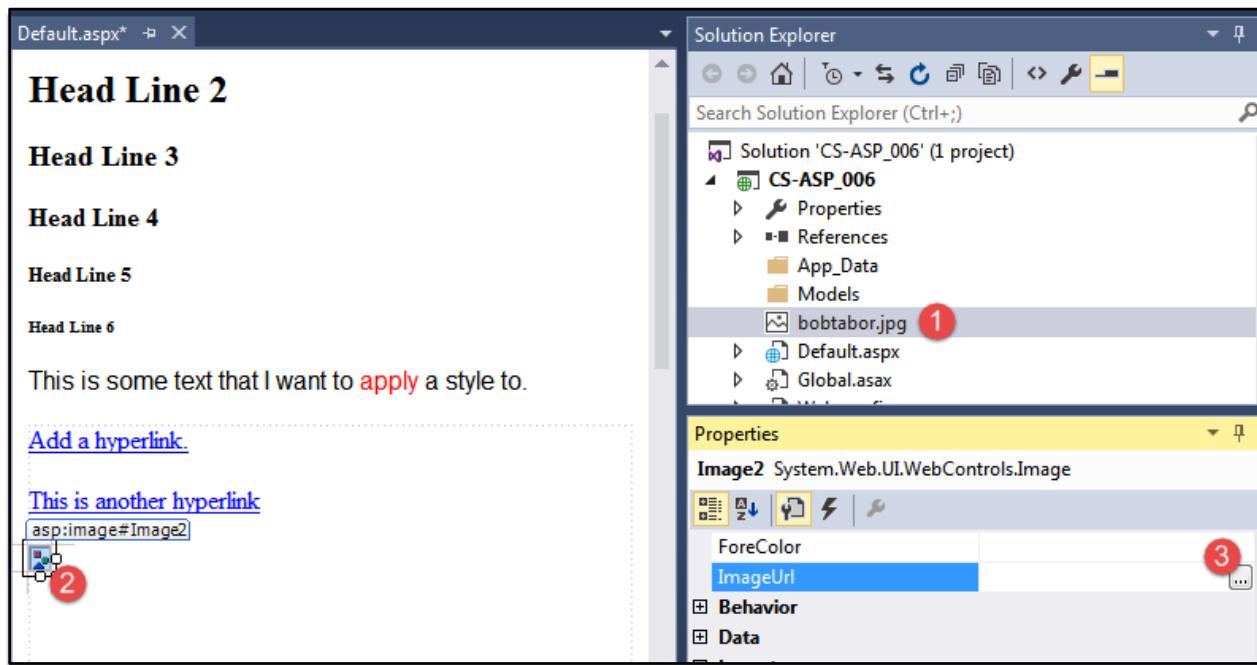


**Bob's Tip:** experiment with the “Target” option to see how it changes the way the link behaves. For example “\_blank” opens up the link in a new window.

## Step 6: Add an Image

Now, underneath the second link, let's insert an image with the following steps:

1. Add an image to the project (the fastest way is to drag and drop it from your drive/desktop, straight to the Solution Explorer).
2. With the cursor under the hyperlink text, insert an Image Server Control from the Toolbox (Ctrl+Alt+X).
3. Browse to the image in the project from the Properties Window for the selected Image.





**Bob's Tip:** you may have noticed that HTML options are available via the HTML Server Control in the Toolbox. You can certainly use that, however, it requires a bit more explanation so we will stick to the basic Server Controls for HTML elements for now.

## Step 7: Insert a Table

---

Now, let's add a table, which is useful for tabular data, such as statistics. Note, do not use tables for styling your graphical elements in a grid-like format. This is a very outdated technique with inherent limitations! To insert the table, place the cursor underneath the image you inserted and in the Visual Studio menu click on:

**Table > Insert Table**

From the table dialogue choose layout of three rows and three columns:

The screenshot shows the 'Table' dialog box with the title 'Size'. It has two input fields: 'Rows:' containing '3' and 'Columns:' also containing '3'. Both fields have up and down arrows for adjustment.

Note other useful features from this dialogue such as the option to set the default values for new tables (useful when you have a similar table structure you want to re-use) as well as the cell spacing (space between the cells) and cell padding (space surrounding elements in the cells). Now, you can enter tabulated data within the cells:

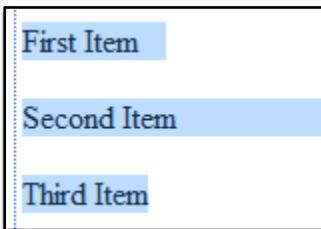
The screenshot shows a web page with a table. At the top left, there is a link labeled "This is another hyperlink.". Below it is a placeholder for an image with two small circular icons. A large red arrow points downwards towards the table. The table has three columns: "Player", "Year", and "Home runs". It contains two rows of data: "Sammy Sosa" in 2005 with 100 home runs, and "Mark MacGuire" in 2005 with 102 home runs.

Player	Year	Home runs
Sammy Sosa	2005	100
Mark MacGuire	2005	102

## Step 8: Create Ordered and Unordered Lists

---

To create an ordered list, input three separate paragraph elements under the table data and highlight them all:



And select from the menu the ordered list icon, rendering an automatically numbered list of items:

Two screenshots illustrating the creation of an ordered list. The top image shows the "List" icon in a toolbar, which is a red-bordered square containing a white list icon. The bottom image shows the resulting numbered list: "1. First Item", "2. Second Item", and "3. Third Item", all contained within a single rectangular selection box.

By contrast, an unordered list is any kind of bulleted list that has no particular sequence. To create an unordered list, repeat the last steps. Except now click on the unordered list icon instead to render a result that looks like this:

Two screenshots illustrating the creation of an unordered list. The top image shows the "List" icon in a toolbar, with the "Bulleted List" option selected (indicated by a red border). The bottom image shows the resulting bulleted list: "• This is an idea", "• This is an equally good idea", and "• Yet one more idea to consider", all contained within a single rectangular selection box.

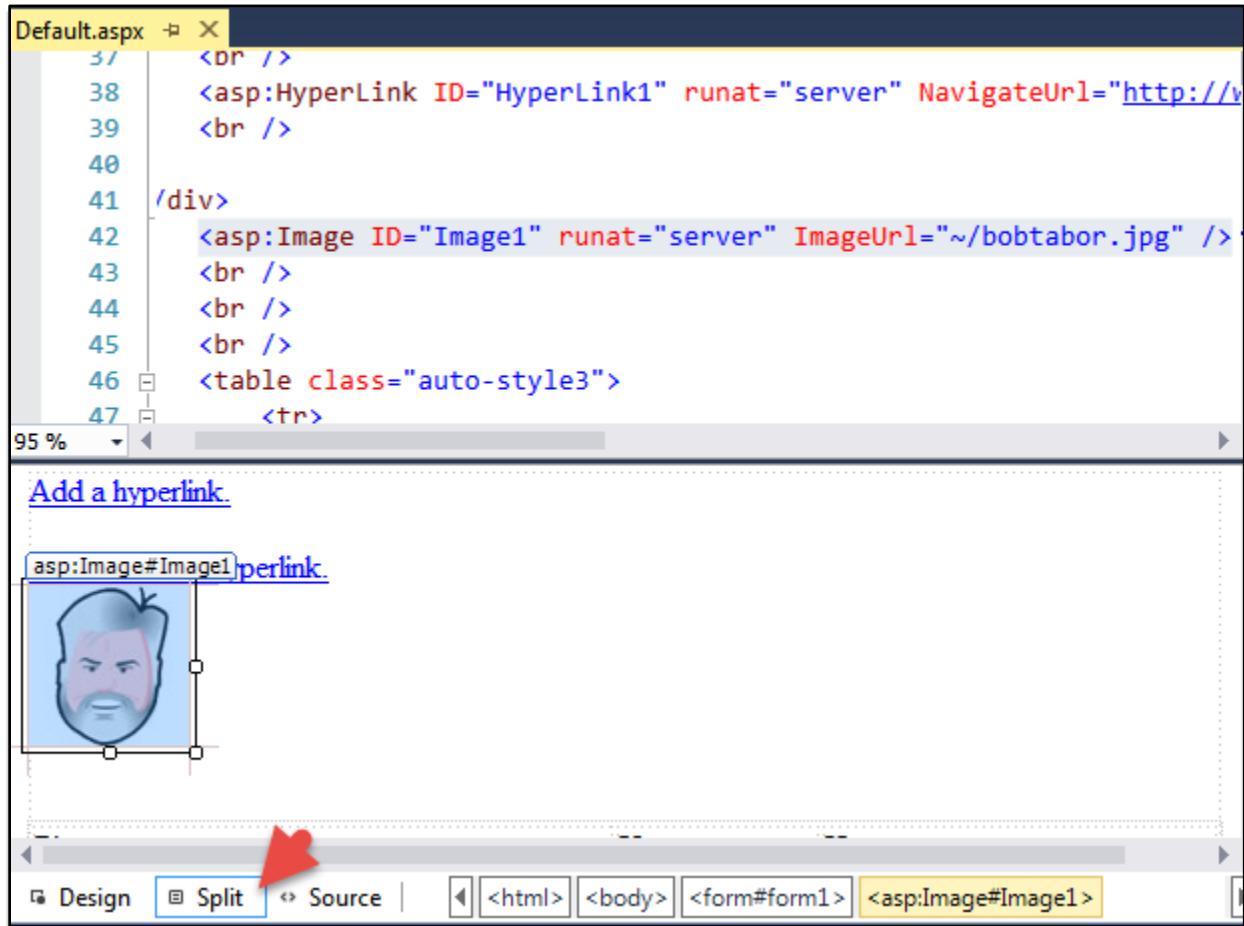
Experiment with styling these elements by highlighting them and changing attributes, such as the background color:



- This is an idea
- This is an equally good idea
- Yet one more idea to consider

## Step 9: Review HTML Generated by Design Mode

And, finally - although we won't be getting into HTML right now - take a look at the code generated by clicking on the "Split" tab to give you an idea of what is actually being created in the background HTML. In this Split View, you can click on different elements in the Design View that correspond to the background code elements to highlight them:





**Bob's Tip:** as mentioned, while the Design Mode tools are useful for structuring “quick and dirty” visual elements, you should eventually get familiar with CSS and HTML in order to gain full control over styling your web page. This doesn’t mean, however, that you will need to write every piece of HTML and CSS yourself. In many cases you can simply purchase existing templates and - having come to understand how to put together HTML and CSS - you can then edit the template to tailor to your needs.

007

# Variables and Data Types

In this lesson, we're going to examine variables and data types more closely. We briefly looked at these coding concepts in "003 – Building Your First Web App" where it was mentioned that variables are like named "buckets" that store values of the type declared. Let's revisit, and expand on, this concept by looking once again at the code within the MyFirstWebApp project.

## Step 1: Declaring Variables and Assigning Values

---

In the okButton\_Click code block, a variable is first declared with an arbitrary name, `firstName`, and the type of information it that can store `string`. It is then assigned the string value contained in another variable called `firstNameTextBox.Text`:

```
string firstName = firstNameTextBox.Text;
```

In the above example, we are completing two separate operations on a single line of code. The first step declares the variable type, as well as its unique name (identifier) that can be used to reference it elsewhere in code. This step simply sets aside an empty "bucket" in memory that is large enough to store a `string` value, and labels this bucket with a name that uniquely identifies it:

```
string firstName;
```

The next step takes that "bucket" and puts an existing string value into it:

```
firstName = firstNameTextBox.Text;
```

You can, of course, write out these steps on separate lines, which would look something like this:

```
string firstName;
firstName = "Bob";
```

Here, we chose the *literal* string "Bob" – which can be any alpha-numeric sequence of characters enclosed in quotes – to make the two-step process clearer. Instead, you could encapsulate that literal string into another variable and it would work just the same (this is closer to what you saw in the original example above):

```
string name = "Bob";
string firstName;
firstName = name;
```

Be careful when declaring variables. It is perfectly acceptable to reference already declared variables, however, you cannot re-declare already declared variables. Here we (1) declared `firstName`, then (2) referenced it by assigning a value to it, but (3) erroneously re-declared the same variable we declared in (1). To fix this, simply remove the type declaration (`string`) in (3) and it would be valid C# syntax:

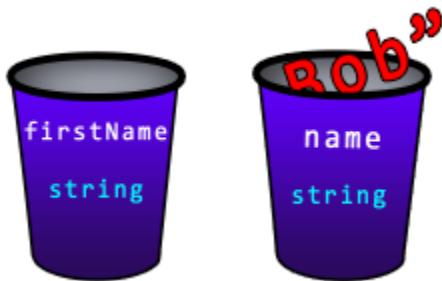
```
① string name = "Bob";
② string firstName;
③ firstName = name;

④ string firstName = "Steven";
```

## Step 2: Understanding Variables as “Buckets”

---

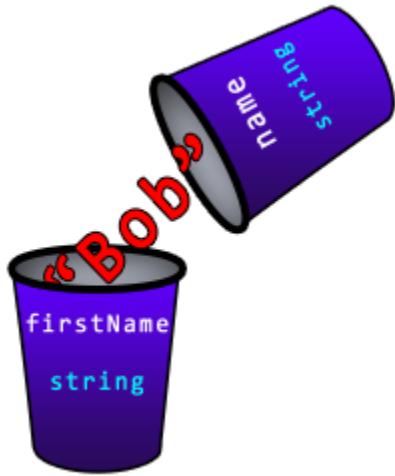
The first two lines of code here can be visualized, using the “bucket” analogy, as looking something like this:



In the above code, you will notice how the variable called “name,” is later referenced by taking the value that it currently holds (“Bob”) and assigning it, via the equal’s assignment operator, *into* the variable called “firstName”:

```
firstName = name;
```

This is a hint as to the versatility you get by creating a variable identifier for referencing throughout your code (such as, in this case, not having to constantly write out the literal string “Bob”). The result of this assignment operation, (taking the value held in `name` and placing it into `firstName`) can be visualized as follows:



Note that the “bucket” variable dumping its value into another bucket (via the assignment operator) *still holds on to the value itself*. The data is actually *copied* from one bucket to another during the assignment operation, which would leave us with two different variables that each hold the same string value:



### Step 3: Understanding Variable Naming Limitations

---

A variable identifier can be labeled as anything you choose, with a few general limitations. The identifier should:

- Be at least one alphabetical character.
- Not start with a number.
- Not contain non-alpha-numerical characters other than dash and underscore.
- Not contain any spaces.
- Not be exactly the same as a reserved word in C#.

Visual Studio will tell you that your variable is not named correctly by producing a red squiggly line, indicating an error:

```
string 1stname;
```

## Step 4: Adding Comments in Code

You will also get a green squiggly line, indicating a warning, when you declare a variable that is never used. Here, we commented out the line of code that makes use of the variable called name, leading to a warning:

```
string name = "Bob";
string firstName;
//firstName = name;
```

You can also comment out multiple lines of code, using `/*` and `*/` as follows:

```
/*
string name = "Bob";
string firstName;
firstName = name;
*/
```



**Bob's Tip:** Commenting out code tells the compiler to *ignore the line entirely*.  
Commenting can be useful for a number of reasons

- **Adding a note to yourself, or others who may read your code.**
- **Removing parts of your code in order to track down problems.**
- **Preserving code that you don't want to throw out entirely, yet has no current use (think code that fulfills some occasional testing purpose, yet shouldn't be in the final build).**

## Step 5: Primitive C# Data Types

We already know that a `string` is a sequence of alpha-numerical characters (numbers and letters) but there are many other primitive, pre-defined data types in C# that hold whole numbers, decimal numbers, characters, true/false values, and so on:

<b>byte</b>	0 to 255
<b>sbyte</b>	-128 to 127
<b>short</b>	-32,768 to 32,767
<b>ushort</b>	0 to 65,535
<b>int</b>	-2,147,483,648 to 2,147,483,647
<b>uint</b>	0 to 4,294,967,295
<b>long</b>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<b>ulong</b>	0 to 18,446,744,073,709,551,615
<b>float</b>	-3.402823e38 to 3.402823e38
<b>double</b>	-1.79769313486232e308 to 1.79769313486232e308
<b>decimal</b>	-79228162514264337593543950335 to 79228162514264337593543950335
<b>char</b>	A single Unicode character
<b>string</b>	A string of Unicode characters
<b>bool</b>	True or False
<b>object</b>	An object (the base type for all other types)



**Bob's Tip:** the main reason for these different sized “buckets” is generally for sake of economy. In other words, don’t use a bucket large enough to hold a quintillion-sized number (long) when all you need is a tiny bucket, big enough to hold a number up to a few thousand (short).

## Step 6: Uninitialized Variable Default Values

---

Here are the **default** values for uninitialized variables (those that are declared, but not yet given any value by using the assignment operator):

- Whole number data types (byte, sbyte, short, ushort, int, uint, long, ulong) – **0**
- Decimal number data types (float, double, decimal) – **0.0**
- string data types – **"" (empty quotes)**
- bool data types – **False**
- char data types – **null**



**Bob's Tip:** this may seem like a rather large list of basic types to keep track of. However, in general the basic data types that you will really need to remember are **string, int, double and bool**.

# Data Type Conversion

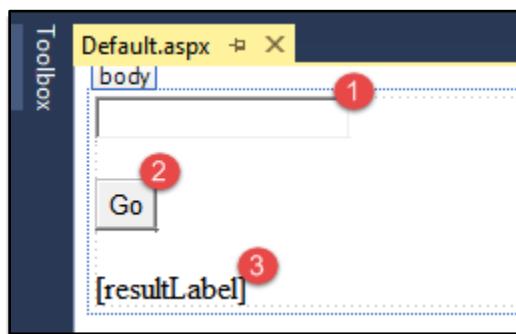
At times, you will need to convert from one data type into another. In this lesson, you will learn how to perform some basic data type conversions, as we touch upon common situations requiring this – most notably when receiving user input from Server Controls via the Web Form.

## Step 1: Create a New Project

---

To demonstrate this, go ahead and set up a new ASP.NET project called “CS-ASP\_008”. And using the steps detailed in “003 – Building Your First Web App” create a *Default.aspx* with TextBox, Button, and Label Server Controls with the following programmatic IDs:

- (1) `inputTextBox`
- (2) `okButton`
- (3) `resultLabel`



## Step 2: Understanding Data Type Assignment Mismatch

---

Then go to the *Default.aspx.cs* file and write the following in the `okButton_Click` code block:

```
protected void okButton_Click(object sender, EventArgs e)
{
    int i;
    i = "Hello World";
}
```

Notice that the literal string “Hello World” becomes underlined with a red squiggly, indicating an error. That’s because we’re trying to put a **string** into a variable that is only able to hold an **int** (integer/whole number) value:



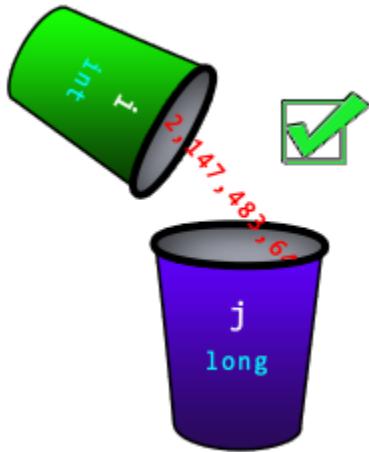
**Bob’s Tip:** if you hover over the underlined text, Visual Studio will give you a hint as to why it ran into an error. In this case, hovering over the underlined string will say “**Error: Cannot implicitly convert type string to int.**” Error messages – when they aren’t cryptic – will often be helpful towards tracking down the problem.

## Step 3: Implicit Conversions

*Implicit* conversions can often be done between similar data types. However, a **string** and an **int** are completely different from one another. It’s not immediately obvious what it would mean to convert a bunch of letters into a number, even if we were to do something like this:

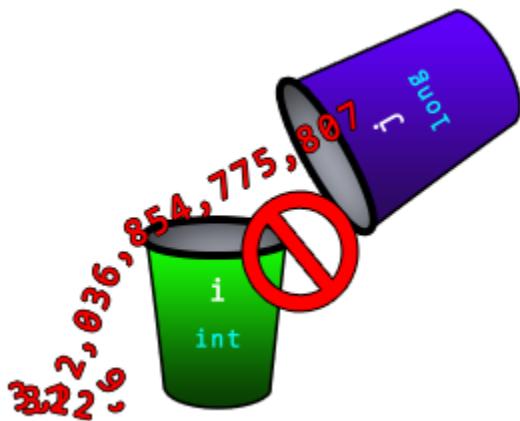
```
int i;  
i = "3";
```

In such cases, where two data types are so fundamentally different, an *explicit* conversion will be needed instead (which we will get to in a moment). By contrast, an example of a simple, *implicit* conversion would be in the event of an *upcast* from a smaller data type to a larger data type of the same base value format. For example, an **int** and a **long** are data types that both store whole numbers. The only difference is that a **long** can simply store a much larger number than an **int**. Upcasting in this scenario would be like taking the contents of a smaller bucket and putting it into a larger bucket:



```
int i = 2000000000;
long j = i;
```

However, this kind of implicit conversion can't be done *downcasting* from a larger bucket into a smaller bucket, even though each bucket stores the same basic value format. Where would all of the extra data from the larger bucket, that can't fit into the smaller bucket, go?:



```
long j = 2000000000;
int i = j;
```

## Step 4: Explicit Conversion via Casting

---

Even though an implicit conversion is not possible in this scenario, you can perform an explicit *cast* operation on the `long` variable `j`. This conversion effectively shrinks `j` down to an `int` before we attempt to copy its contents over to the `int` variable `i`. This explicit conversion process is possible because the underlying data types are both whole numbers. Casting is very simple and consists of

prefixing the variable that you want to convert - with the type you want to convert to - enclosed in parentheses:

```
long j = 2000000000;
int i = (int)j;
```



**Bob's Tip:** there are ramifications of performing the cast operation. By doing so you, the programmer, are taking control and overriding the default behavior, which is to not allow a *downcast* from a larger type into a smaller type. An unintended error/exception or data loss might occur in certain cases.

## Step 5: Double to Int Truncation

Note that in the above example, we were able to cast `j` to an `int` without data loss because an `int` can hold an integer value up to 2,147,483,647. If, on the other hand, `j` held a number larger than that, it simply would not fit into `int i` and would yield a totally different result than we were expecting. Let's consider a similar scenario where we try to do an explicit conversion between a `double` and an `int`. We see that this is possible, in principle, as the compiler doesn't show any errors:

```
double k = 2.5;
int i = (int)k;
```

However, we know that an `int` won't be able to represent the decimal place, as it can only represent whole numbers. Let's try to output this to see the result of casting `k` to an `int`:

```
double k = 2.5;
int i = (int)k;
resultLabel.Text = i;
```

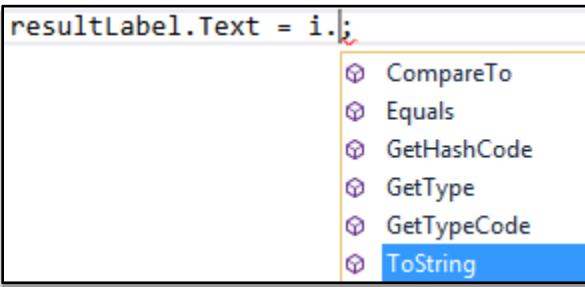
(local variable) int i  
Cannot implicitly convert type 'int' to 'string'

## Step 6: Convert to String via `ToString()` Helper Method

We see this error when trying to copy the integer value from `i` into the string value for `resultLabel.Text`. You may be tempted to try something like this:

```
resultLabel.Text = (string)i;
```

However, that doesn't work because casting is possible only between similar underlying data types. You would be forgiven for thinking that 2.5 and "2.5" are not terribly different. They may be of different data types, but the only real difference is one is in quotes while the other one isn't. Fortunately, there are helper methods (we will cover methods in depth in later lessons) that can handle conversions such as these that are simple in nature, yet can't be done with explicit casting. In this case, we will want to access the `ToString()` method, that you can find as an option when using Intellisense and the "dot accessor" (period) after the variable:



Be sure to *invoke* the helper method by inputting empty parentheses after the method name:

```
resultLabel.Text = i.ToString();
```

When you run the application, and click the "Go" button, here is what you will see:



This is an example of the kind of error/data loss that can occur when casting. In this case, casting from `double k` to `int i` truncated whatever the decimal value would have been, owing to the fact that an `int` simply won't recognize decimal values.



**Bob's Tip:** *truncating* a value is very different from *rounding* a value to the nearest whole number. Truncation simply chops off the decimal value as if it never existed. Truncating can be very useful in many circumstances, but can also cause problems in your code if it was not the result you anticipated.

Going back to the application, let's allow the user to type in a number, perform a calculation on that number, and then display the result. We know ahead of time that at least two conversions will have to take place:

1. **The TextBox Control stores user input into a string, so we will want to convert that string to an int in order to perform a calculation on it.**
2. **The Label Control stores the result as a string, so we will want to convert the result of the int calculation back into a string.**

## Step 7: String to Int via int.Parse() Helper Method

We already saw that you can't cast an int to a string. So, instead, we use a helper method to perform the conversion:

```
string i = inputTextBox.Text;
int j = int.Parse(i);
```



**Bob's Tip:** this looks a little bit different from the `ToString()` helper method we saw when going from an int to a string, but the same principle is in effect. The main difference is that the `Parse()` helper method comes from inside of the int "class" and takes in a string as input, in between the method parentheses. This will all become much clearer when we delve into methods a bit later.

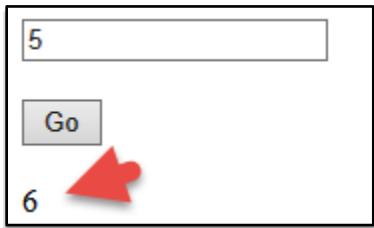
Now that we have the user input stored as an integer in `int j`, we can safely perform a calculation on that integer:

```
string i = inputTextBox.Text;
int j = int.Parse(i);
int k = j + 1;
```

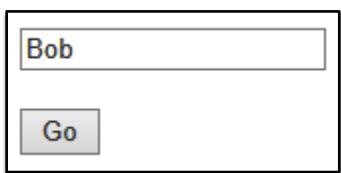
And then to display this through our Label Control, we simply convert `k` to a string, as we did before:

```
string i = inputTextBox.Text;
int j = int.Parse(i);
int k = j + 1;
resultLabel.Text = k.ToString();
```

Now, when we run the application we see the process worked as intended:

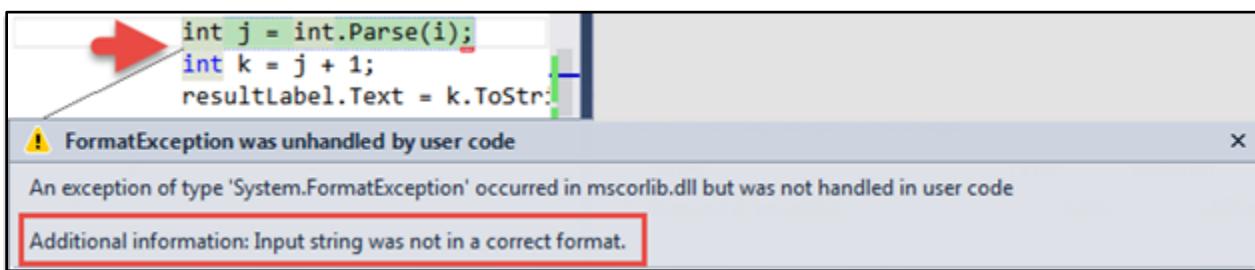


At this point, you may be tempted to see what happens when you input a value that can't be converted to an integer, for instance:



## Step 8: Failed Conversion Runtime Error

A *runtime exception/error* will occur, and Visual Studio will inform you with exactly what line of code failed. Unsurprisingly, it was the line that attempted to convert the string to an int. Essentially, `int.Parse()` will only work when the string looks exactly like a number, and nothing more:





**Bob's Tip:** runtime errors are insidious because they cannot be caught by the compiler. We've seen how compile-time errors will be caught by Visual Studio, and warn you with a red squiggly line before you try and run your application. By contrast, runtime errors can linger in your code until they are discovered through rigorous testing, or else a user-submitted bug report.

The way to combat this situation is to perform some validation that checks if that the user provided input in the correct format. Validation requires some logic that we are not yet ready to get into yet, but note that we will come back to this issue in a later lesson.

009

# Arithmetic Operators

In this lesson, we will uncover arithmetic operators, which entail nothing more than common math calculations. There are a few other “fancier” math operations available, but these are the ones that are most commonly used in business applications:

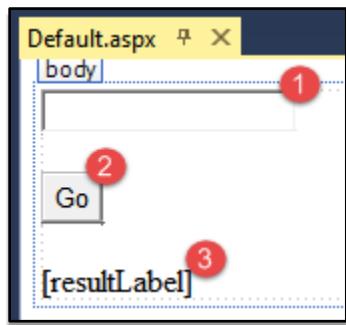
- **Addition**
- **Subtraction**
- **Multiplication**
- **Division**

## Step 1: Create a New Project

---

For this lesson, create an ASP.NET application called “CS-009” using the previously detailed workflow. Add to this project a *Default.aspx* file with the following Server Controls and programmatic IDs:

- (4) **inputTextBox**
- (5) **okButton**
- (6) **resultLabel**



Before going into math operations, it is important to distinguish the way you would normally think of the equal's sign (=) with the way it is used in C#. In math, the equal's sign is used to designate equivalence between elements on both sides of that symbol: **1+1 = 2**

However, in C#, the equal's sign is not used as an equivalence operator (there is another operator used for that operation). Rather, the equals sign designates *assignment*, which was touched upon in the previous lessons where it was mentioned that dumping the contents of one variable “bucket” into another assigns that value to it. And value assignment can be of any data type available, not just numerical values.

## Step 2: Addition, Subtraction, Multiplication and Division in C#

---

With this in mind, let's write out some basic math operations using familiar mathematical operators, while assigning the results of those calculations to a variable "bucket". You can write this out in the `okButton_Click` event in `Default.aspx.cs`, commenting to avoid re-declaring previously declared variables:

```
int i = 1;  
int j = 2;  
int result = i + j; //Addition
```

```
int i = 1;  
int j = 2;  
int result = i - j; //Subtraction
```

```
int i = 1;  
int j = 2;  
int result = i * j; //Multiplication
```

```
int i = 1;  
int j = 2;  
int result = i / j; //Division
```



**Bob's Tip:** it might be helpful to read these statements from right-to-left. In other words, this code first does the calculation and then stores that calculated value by assigning it with the equal's sign operator into the variable `result`.

You can treat the variable used in the calculation as the literal value that it holds. That means that there is no essential difference between the formulations shown above, and one such as this:

```
int result = 1 + j;
```

You can also take whatever current value is contained in the variable that you end up storing the result of the calculation into, and use it within the calculation:

```
int i = 1;  
i = i + 1; //i ends up storing 2
```

## Step 3: Using Calculation Assignment Shortcuts (`+=`, `-=`, `++`, `--`)

---

There is a shortcut when performing this calculation, by combining the arithmetic operator with the assignment operator as follows:

```
int i = 1;  
i += 1; //i ends up storing 2
```

The above example has the effect of incrementing by 1. You have yet another shortcut for representing a *single increment* by writing this:

```
int i = 1;  
i++; //i ends up storing 2
```

Or, you could decrement in much the same way:

```
int i = 1;  
i--; //i ends up storing 0
```

Note that you can perform these operations, one after the other, as much as you want:

```
int i = 1;  
i += 5; //i ends up storing 6  
i++; //i ends up storing 7  
i++; //i ends up storing 8  
i--; //i ends up storing 7
```

## Step 4: Operator and Calculation Precedence

---

When performing calculations, it is important to be aware of operator precedence (also called “order of operation”). For example, notice the result we get when running this code:

```
int myInteger = 5 + 1 * 7;  
resultLabel.Text = myInteger.ToString();
```

A screenshot of a Windows application window. It contains a single-line text input field at the top, a blue rectangular button labeled "Go" below it, and a black rectangular label at the bottom displaying the number "12".

This happens because C# adheres to the operator precedence dictated by ordinary math. In other words, multiplication and division occur before addition and subtraction. However, what if we intended to first add  $5 + 1$ , and then multiply the result of that with 7? You can create your own precedence by wrapping the calculation in parentheses, again, just like in ordinary math:

```
int myInteger = (5 + 1) * 7;  
resultLabel.Text = myInteger.ToString();
```

A screenshot of a Windows application window. It contains a single-line text input field at the top, a blue rectangular button labeled "Go" below it, and a black rectangular label at the bottom displaying the number "42".

## Step 5: Calculating with Mixed Data Types

---

Next, let's see what happens when we mix a decimal value with a whole number in a calculation:

```
double myDouble = 5.5;  
int myInteger = 7;  
  
double myResult = myDouble + myInteger;  
resultLabel.Text = myResult.ToString();
```

A screenshot of a Windows application window. It contains a single-line text input field at the top, a blue rectangular button labeled "Go" below it, and a black rectangular label at the bottom displaying the number "12.5".

The calculation performs as we would expect because `myInteger` gets implicitly upcast to a `double` in order to make the calculation work. However, let's see what happens when we try to store this calculation to an `int` instead:

```
//double myResult = myDouble + myInteger;  
int myResult = myDouble + myInteger;  
resultLabel.Text = myResult.ToString();
```

As you can see with the red squiggly underline, this produces an error immediately recognized by the compiler. It warns us that we are trying to implicitly convert `myDouble` to an `int`, however, we know that we can do the conversion ourselves by simply casting as follows:

```
int myResult = (int)myDouble + myInteger;  
resultLabel.Text = myResult.ToString();
```

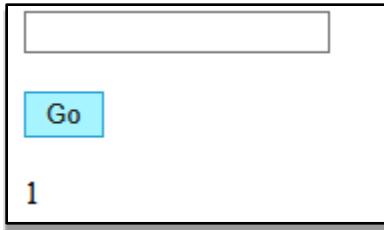
This now works because we are downcasting `myDouble` to an `int`, however, you may recall from the previous lesson that this will result in truncating the decimal data:



You should always be aware of truncation occurring when dealing with integers, especially when dividing whole numbers and wanting a decimal value result:

```
int myInteger = 7;  
int myOtherInteger = 4;   
  
int myResult = myInteger / myOtherInteger;  
resultLabel.Text = myResult.ToString();
```

When dealing with doubles we would expect the result of this division to be 1.75. However, because we are dealing with integers, the resulting calculation ignores the decimal data, leaving us with 1:



You may be tempted to solve this problem by storing the result of the calculation into a double instead:

```
double myResult = myInteger / myOtherInteger;
```

However, this still yields the same truncated result as before. The reason for this is because the calculation is still being performed on ints. The way we fix this is by upcasting the ints to doubles:

```
double myResult = (double)myInteger / (double)myOtherInteger;
```



## Step 6: Calculating a Value That “Overflows the Bucket”

---

In the previous lesson, we confronted the possibility of data overflow, where a given data type isn't large enough to store all of the information it is given. Recalling that an int is only capable of storing a number up to 2147483647, let's see what happens when we overflow an int variable with a number that is too large:

```
int firstNumber = 2000000000;
int secondNumber = 2000000000;
int resultNumber = firstNumber * secondNumber;

resultLabel.Text = resultNumber.ToString();
```

**Go**  
-1651507200

You are probably quite confused by this result. While it is clear that the result of this calculation would be way outside the boundaries of what an `int` can store, why would it result in a negative value? The answer is quite technical and not necessarily informative to a beginning programmer. Just be aware that this can happen when doing calculations that may involve larger numbers than you anticipated.



**Bob's Tip:** the moral of the story is that you should always use larger data types whenever you're performing arithmetic operations with the *potential* to overflow.

The most obvious way of solving this problem is to use the largest data type we know of for storing such a large number:

```
long resultNumber = (long)firstNumber * (long)secondNumber;
```

  
**Go**  
4000000000000000000000000

If you *only* cast the assigned-to variable to `long`, you will get the overflow, for a similar reason as we saw with truncation by dividing `ints` earlier:

```
long resultNumber = firstNumber * secondNumber;
```

  
**Go**  
-1651507200

## Step 7: Using “checked” to Catch Silent Overflows

---

Perhaps the most important thing to realize about these kinds of overflow problems is that they can happen quietly, without your knowledge, unless you take steps to:

- Consider the possibility of it happening ahead of time
- Use a “checking” measure to report when an overflow occurs

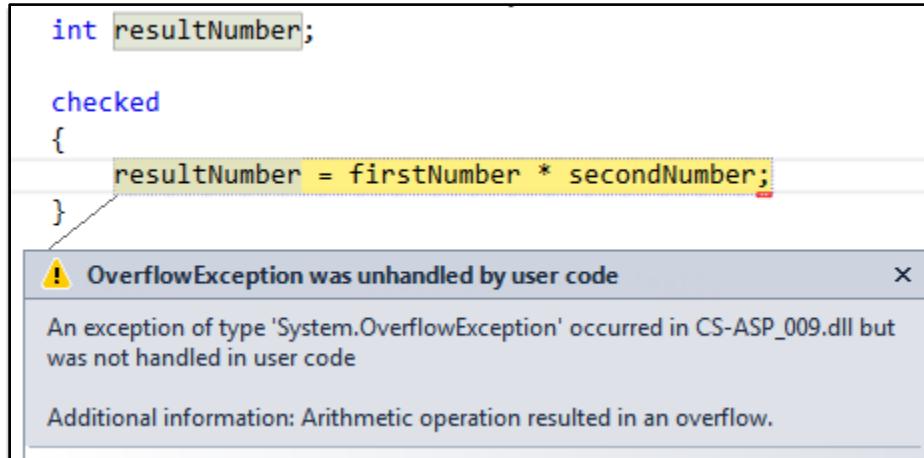
There is a built-in checking measure that you can use in C#:

```
int firstNumber = 2000000000;
int secondNumber = 2000000000;
int resultNumber;

checked
{
    resultNumber = firstNumber * secondNumber;
}

resultLabel.Text = resultNumber.ToString();
```

When the application runs, it will no longer be silent about the overflow, crashing on this line of code:





**Bob's Tip:** don't worry about how odd this code might look right now. This will become clearer when you learn about *scope* in a later lesson. This is just an illustration of the kinds of issues you can run into when you are not careful about properly handling your data types. In the end, no programmer can rely on foresight alone; apps will crash, users will submit bug reports, and mistakes will need to be weeded out accordingly. However, what programmers *can* do is account for these possibilities, and write safeguards to gracefully handle such occurrences.

There is more to arithmetic operations than what was covered in this lesson. However, with these basics under your belt, you will be well-equipped to tackle the majority of problems needing to be solved when working with numerical data types in the coming lessons.

010

# C# Syntax Basics

This lesson will cover the basic syntax, or “grammar,” of C#. Programming languages are a bit like natural languages – such as English – except you are communicating your intent to a compiler instead of a human being. Whereas humans can infer details communicated in a natural language - even when they are ambiguous - a compiler is quite dumb and demands precise, explicit instructions. If you forget even a capital letter, a period, or a semi-colon, the C# compiler will complain and refuse to render the code to a format readable by the machine.

## Step 1: Combining Operators/Operands for Expressions/Statements

---

C# is similar to a language like English in that both are composed of statements and expressions constructed from elements that are “things,” and “actions done by those things.” In English, we would call these elements nouns and verbs, respectively. However, in C# you would say these elements are *operands* and *operators*. Operands - as we have come to know them - can be literal or variable:

```
//two string variables
string mySimpleName = inputTextBox.Text;
//a literal string and a string variable
string myString = "This is a string literal" + mySimpleName;
//an integer variable and a literal integer
int myInt = 5;
```

There are many operators in C# that we looked at, such as assignment, mathematical, concatenation, and so on:

```
//two string variables
string mySimpleName = inputTextBox.Text;
//a literal string and a string variable
string myString = "This is a string literal" + mySimpleName;
//an integer variable and a literal integer
int myInt = 5;
```

For an exhaustive list of operators, categorized in lists describing what they do, take a look at:

<http://is.gd/operators>

As you can see, we form complete expressions by combining operands and operators in much the same way that we combine nouns and verbs to express an idea in a natural language. These programmatic expressions even use a form of punctuation, such as terminating with a semi-colon (which is similar to how a period terminates a sentence).



Bob's Tip: there is a technical difference between *expressions* and *statements* in C#. An expression is a combination of operators and operands that evaluate to a single value, such as the simple operations seen in this lesson so far. Statements, on the other hand, are complete actions. This can be a block of code that executes, a complete line of code ending in a semi-colon, calling a method, etc.

For an in-depth description of statements, visit:

<http://is.gd/statement>

## Step 2: Understanding Code Flow of Execution

---

The order in which statements are executed in a program is called the flow of control/flow of execution. Up to this point, we've been looking at code executed line-by-line, from top to bottom, following a linear flow order. Soon, we're going to learn about statements that can change the flow of execution based on different conditions being met, including:

- If statements.
- Switch statements.
- Looping statements.

## Step 3: Understanding Code Style

---

Another important aspect of syntax in C# is code style. This is one case where programming is not as strict as a written, natural language. If you submitted an English paper that had a sentence looked like this:

Here is My AweSome  
English Paper.

You would get a failing grade simply because it doesn't follow the basic rules of English formatting which economizes whitespace. However, in a programming language like C#, these two different ways of writing a statement are treated identically by the compiler, as it just ignores whitespace:

```
string s = "Hi" + firstName + " " + lastName + ".";
```

```
string s = "Hi" + firstName  
    + " " + lastName  
    + ".";
```



Bob's Tip: while code styling is a matter of taste, there are some general rules to follow that you will come to notice as we progress in these lessons. The basic rule to follow is to write code that is, above all, readable by a human being. Consistent variable naming conventions, blank lines, indentations, spaces and so on, are there to make code easier for you to read.

## Step 4: Real-Time Error Detection in Visual Studio

---

The compiler is the ultimate arbiter of C# syntax. However, Visual Studio works on behalf of the compiler - and to your benefit - to warn you when your code will not compile (with a little red squiggly line underneath the code in question). This is the compiler's way of saying you are using unacceptable grammar (syntax) as per the rules of constructing C# expressions/statements. In the following example, the compiler is telling you that it has no idea what to do with this code as there is no expression that resolves to a single value (such as taking the value returned from this calculation and assigning it to a variable):

```
int i;  
int j;  
i + j;
```

➊ (local variable) int i

Only assignment, call, increment, decrement, and new object expressions can be used as a statement

Use of unassigned local variable 'i'

Even if a calculation were possible, by assigning values to `i` and `j`, the last statement still has no meaning since the compiler has no idea what you want to do with the result of that calculation (it has to be part of a valid expression in order to make sense).

There is still a lot more to say about syntax rules, but if you keep these basic rules in mind - and see a compilation error in your code - understand that it's probably because you didn't form the code correctly, according to C#'s "grammar" rules.

011

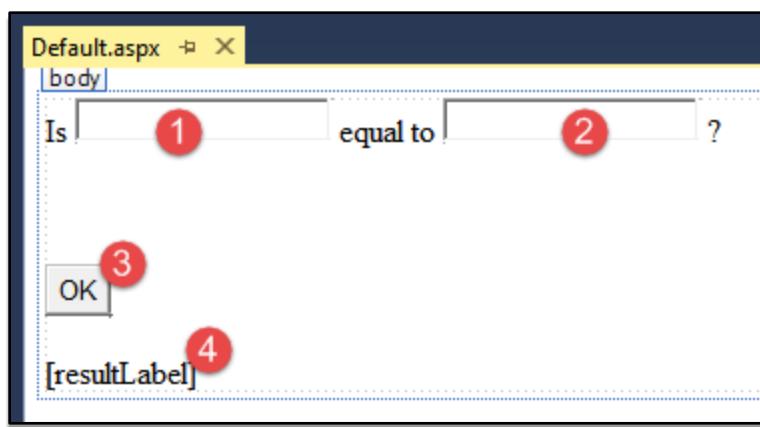
# Conditional if(), else if() and else() Statements

This lesson will cover changing the flow of code using conditional logic. You will be making branching if() statements in code that represent paths that your code can take - like a fork in the road - depending on whether or not a particular condition is met.

## Step 1: Create a New Project

---

Create a new ASP.NET project and call it "CS-ASP\_011," and in Design Mode set up *Default.aspx* as follows:



The programmatic IDs for these Controls are:

- (1) firstTextBox
- (2) secondTextBox
- (3) okButton
- (4) resultLabel

The purpose of this application will be to evaluate what the user enters for each TextBox Control. A message will then be displayed depending on whether or not the values are considered to be equivalent.

## Step 2: Create an if() Statement

---

First, write an empty if() statement inside of the okButton\_Click event:

```
protected void okButton_Click(object sender, EventArgs e)
{
    if (true)
    {
    }
}
```

The part that says “true” will be replaced by an expression that will be evaluated as either true or false. The squiggly brackets directly beneath this `if()` statement will execute *only in the case that* the expression evaluates as true. Let’s now evaluate whether or not each TextBox is equivalent to the other:

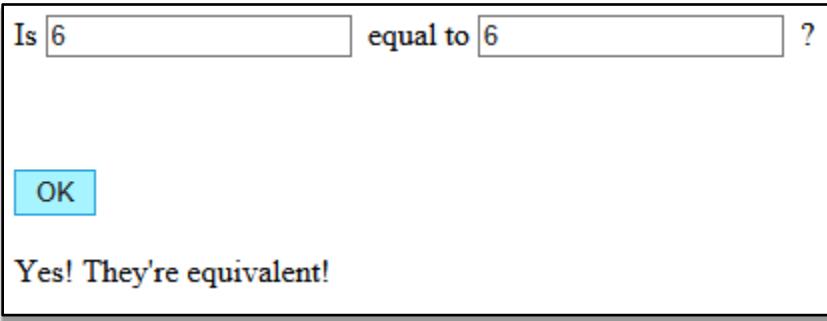
```
if (firstTextBox.Text == secondTextBox.Text)
{
}
```

### Step 3: The Equivalence (==) Operator

---

Notice that we are using the double-equal’s sign, which is the *equivalence* operator. It’s easy to forget that the single-equal’s sign is the assignment operator, and accidentally use that instead. In this example, *we do not wish to assign anything at all*. We just want to check for equivalence. Now, let’s specify what happens if the values entered for each TextBox are equivalent and test it out by running the application:

```
if (firstTextBox.Text == secondTextBox.Text)
{
    resultLabel.Text = "Yes! They're equivalent!";
}
```



## Step 4: Clear Out resultLabel.Text on Button Click

---

You may notice that once the message is displayed, it will continue to stay there even when inequivalent values are set. What we will need to do is reset resultLabel.Text each time the okButton\_Click event is run. We achieve this by (1) clearing the string value held in resultLabel.Text and initializing it with an empty string. And now, when inequivalent values are entered, the code will *completely skip* the conditional code block (2) leaving an empty resultLabel.Text to display:

```
protected void okButton_Click(object sender, EventArgs e)
{
    ① resultLabel.Text = "";
    if (firstTextBox.Text == secondTextBox.Text)
    {
        ② resultLabel.Text = "Yes! They're equivalent!";
    }
}
```

## Step 5: Add an else() Clause to the If() Statement

---

If we want to display a message in the case that the values are not equivalent, we just add an else() clause after the first if() statement:

```
if (firstTextBox.Text == secondTextBox.Text)
{
    resultLabel.Text = "Yes! They're equivalent!";
}
else
{
    resultLabel.Text = "No! They're not equivalent!";
}
```

Is  equal to  ?

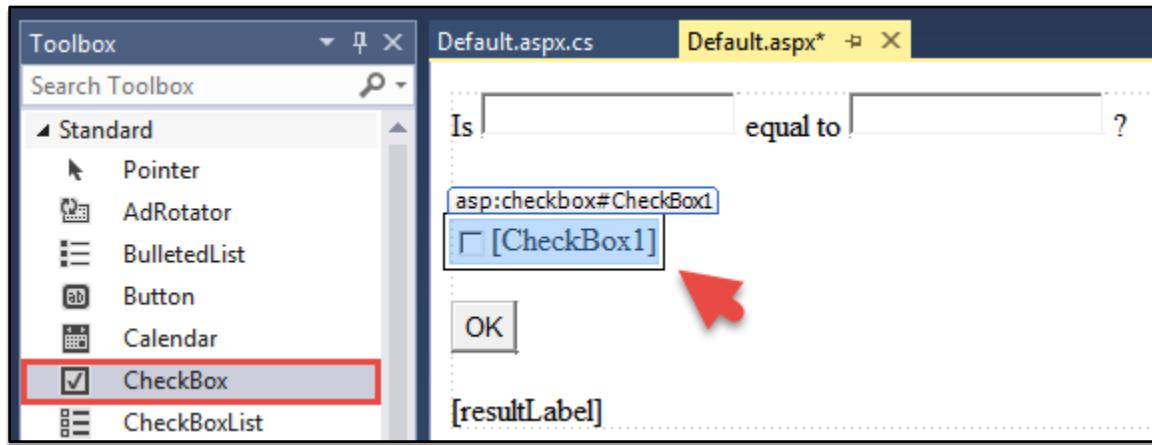
No! They're not equivalent!



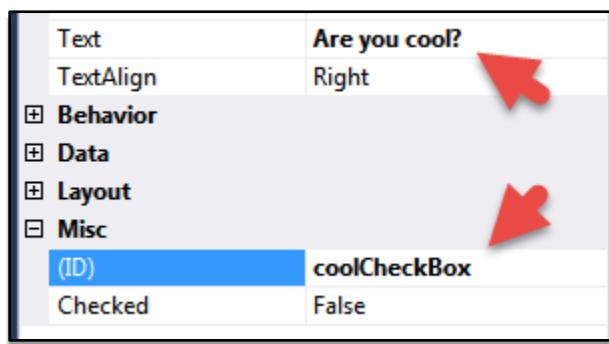
Bob's Tip: the key point worth noting about conditional statements is how *execution flow changes* depending on whether or not the condition evaluates as true. This makes applications much more dynamic than simply executing one line after the other, flowing in a static pattern.

## Step 6: Add a CheckBox Server Control

Next, add a CheckBox Server Control in the designer:

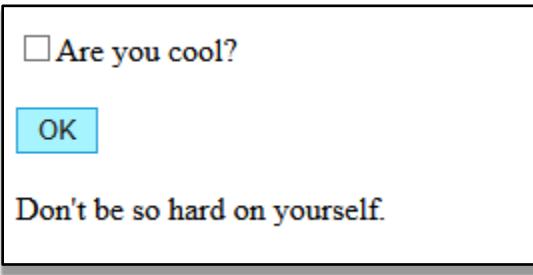
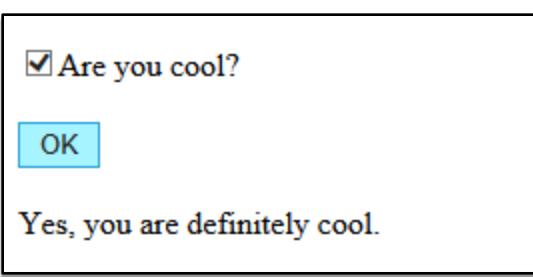


Now set the programmatic ID for this CheckBox to "coolCheckBox," and set the text to "Are you cool?" Also, notice how the CheckBox defaults to one of two possible values (true or false).



In the okButton\_Click event, comment out the previous conditional block and create a new one to determine what happens whether or not the CheckBox is checked:

```
protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = "";
    /*
    if (firstTextBox.Text == secondTextBox.Text)
    {
        resultLabel.Text = "Yes! They're equivalent!";
    }
    else
    {
        resultLabel.Text = "No! They're not equivalent!";
    }
    */
    if (coolCheckBox.Checked == true)
    {
        resultLabel.Text = "Yes, you are definitely cool.";
    }
    else
    {
        resultLabel.Text = "Don't be so hard on yourself.";
    }
}
```



## Step 7: Using Bools as Expressions in If() Statements

---

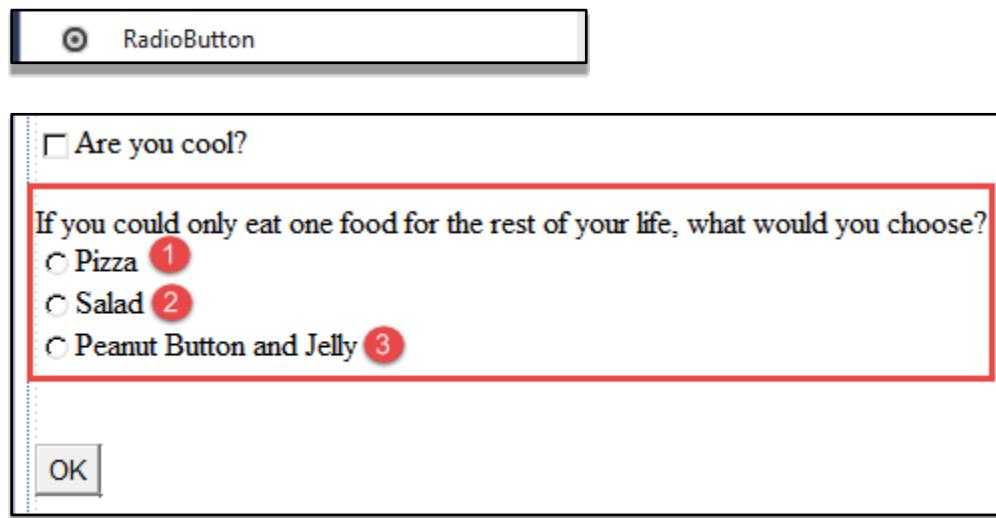
The conditional being evaluated here is the *current state* of coolCheckBox. Checked. Since it is a bool - we don't even have to include the reference to "true" in the evaluation. It is already implied by evaluating an inherently true/false value, so you can just write:

```
if (coolCheckBox.Checked)  
{
```

## Step 8: Add a RadioButton Server Control

---

Now back in the designer, add three separate RadioButton Controls to *Default.aspx*:



Make the respective programmatic ID's as follows:

- (1) pizzaRadioButton
- (2) saladRadioButton
- (3) pbjRadioButton

## Step 9: Add an else if() Clauses Between if() and else()

---

Now, in the okButton\_Click event, we could write out separate if() statements to check which RadioButton is selected. However, there is a much more elegant way to handle multiple possibilities, and that is by sandwiching else if() statements in between if() and else() statements. Comment out the previous conditional code blocks and create this new set:

```
if (pizzaRadioButton.Checked)
{
    resultLabel.Text = "You must be from Chicago!";
}
else if (saladRadioButton.Checked)
{
    resultLabel.Text = "You must be healthy";
}
else if (pbjRadioButton.Checked)
{
    resultLabel.Text = "You must be a fun loving person!";
}
else
{
    resultLabel.Text = "Please select one of the options.";
}
```

When you run the application now you will see the result of these multiple, branching conditionals:

If you could only eat one food for the rest of your life, what would you choose?

- Pizza
- Salad
- Peanut Button and Jelly

OK

You must be a fun loving person!



Bob's Tip: the `if()`, `else if()`, `else()` set of conditional statements cannot be done out of order. You must always start with a single `if()` statement, and (optionally) end with a single `else()` statement. Any number of `if else()` statements can be included, but these always have to come between the `if()` and `else()` statements.

## Step 10: Group the RadioButtons

---

RadioButton's are meant to be exclusive selections; in other words, only a single RadioButton can be selected at any given moment. However, there is a problem in our code because at the moment we can select as many as we want:

A group of three radio buttons. The first two are checked (indicated by a black dot inside the circle), while the third is unchecked. The options are: Pizza, Salad, and Peanut Button and Jelly.

What we need to do is make each RadioButton a part of a group. This can be achieved by setting the `GroupName` property for each Radio Button. To do this, change each `GroupName` to "FoodGroup":

Behavior	
AutoPostBack	False
CausesValidation	False
ClientIDMode	Inherit
Enabled	True
EnableTheming	True
EnableViewState	True
GroupName	FoodGroup

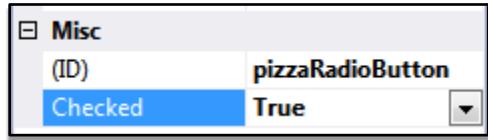
Also worth mentioning is that the application does not let you deselect a RadioButton, it only lets you *change* a selection from one to another. This means that the *only* case where the final `else()` clause will execute is if we never make a single selection, yet press the "OK" button anyways:

A group of three radio buttons. All three are unchecked. Below the group is a button labeled "OK". A red arrow points from the text "Please select one of the options." to the "OK" button. The message is displayed in a small box at the bottom of the form.

## Step 11: Setting a Default Value for the RadioButton

---

Note that we can obviate the need for the final else() clause, forcing a selection, by setting one of the RadioButton's default values for Checked to "True":



012

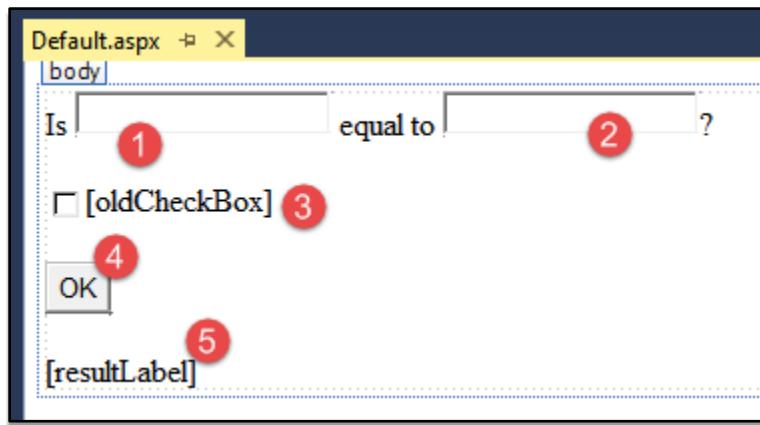
# The Conditional Ternary Operator

In this lesson, we'll cover a handy shortcut for writing an `if()` and `else()` statement. The shortcut is called the *ternary operator* and does the same kind of flow control as an `if()` statement, branching off and executing one of two possible conditions. If you need to evaluate more than two possible conditions, stick with using the `if()`, `else if()`, `else()` set of conditional statements.

## Step 1: Create a New Project

---

To see this in action create a new ASP.NET project called "CS-ASP\_012" and set it up with the following Server Controls:



The programmatic IDs for these Controls are:

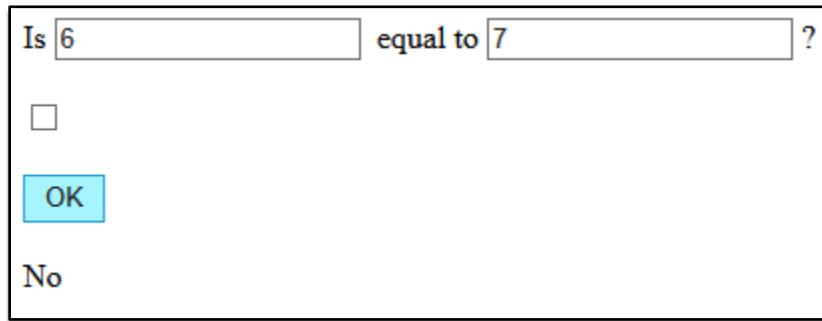
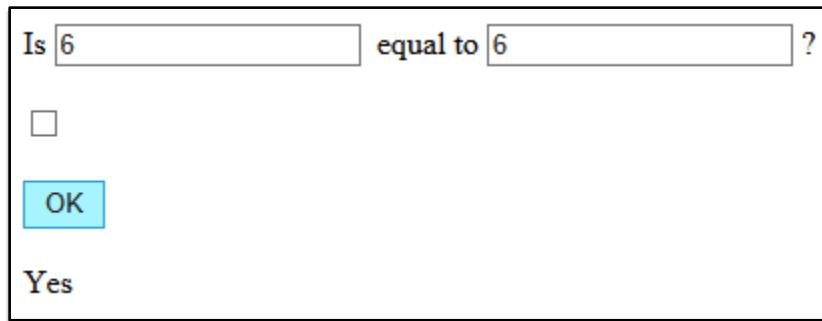
- (1) firstTextBox
- (2) secondTextBox
- (3) oldCheckBox
- (4) okButton
- (5) resultLabel

## Step 2: Create a Statement using the Ternary (?) Operator

---

Now let's write a statement using the ternary operator in the okButton\_Click event, and then run the application:

```
protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = (firstTextBox.Text == secondTextBox.Text) ? "Yes" : "No";
}
```



## Step 3: Breaking Down the Ternary Operator

---

The ternary statement may look odd at first, but when you learn how to interpret it, you will find it's no different than the conditional statements we've been writing thus far. Here is how you can read this statement:

```
resultLabel.Text = (firstTextBox.Text == secondTextBox.Text) ? "Yes" : "No";
```

3

2

1

4

5

6

1. If...
2. firstTextBox.Text is equivalent to secondTextBox.Text
3. then, resultLabel.Text equals
4. "Yes"
5. else...
6. "No"

Now, let's create a ternary statement for the CheckBox control.

```
resultLabel.Text = (oldCheckBox.Checked) ?  
    "I'm teaching an old dog new tricks!" :  
    "Young whippersnapper! Get off my lawn!";
```



Bob's Tip: your eyes are not playing tricks on you! Recall that whitespace is ignored by the compiler, leaving it up to you how to style code to look more appealing/easier to read. In this case, we chose to split the ternary statement on multiple lines simply because it was running rather long for a single line of code. But make no mistake, this is still just one statement with a single semi-colon on the end.

Remember, the evaluation for oldCheckBox.Checked implies a Boolean value, so we do not have to write out the full expression of oldCheckBox.Checked == "true". Running the application will produce the same branching nature as we have come to expect:

Is  equal to  ?

**OK**

Young whippersnapper! Get off my lawn!

Is  equal to  ?

**OK**

I'm teaching an old dog new tricks!

## Step 4: A “game-like” Example of the Ternary in Action

---

Combining several things we've learned so far, let's write out a ternary statement that demonstrates how we might go about using it for a game-like situation. In this scenario, different score values may be awarded depending on if an equivalence has been evaluated as true or false.

```
int result = (firstTextBox.Text == secondTextBox.Text) ? 100 : 50;  
  
resultLabel.Text = result.ToString();
```

Is  equal to  ?

**OK**

100

Is  equal to  ?

**OK**

50

013

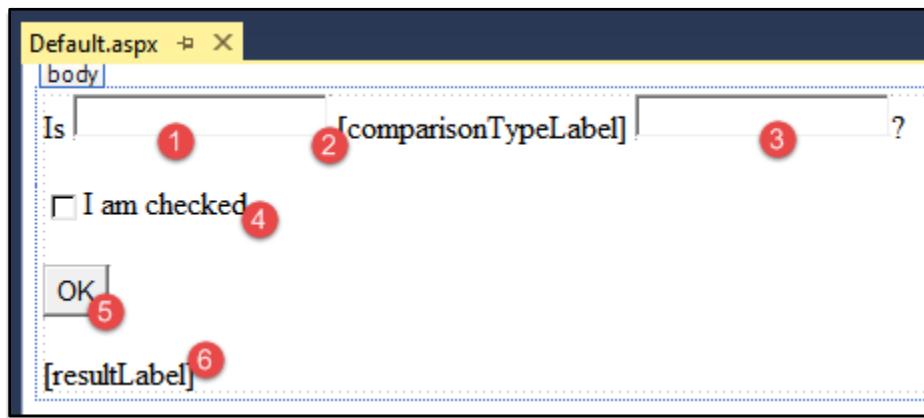
# Comparison and Logical Operators

In this lesson, we're going to delve deeper into comparison operators. We've already looked at the equivalence operator (the double-equal's sign), which just tests whether or not two sides are equivalent to one another (whether that's two variables, two literal strings, two numbers, and so on). These other operators can also be used within a conditional's parentheses for evaluation, but the rest of the evaluation process remains the same: it checks to see if the *entire* evaluated statement is true, or false.

## Step 1: Create a New Project

---

To demonstrate this, set up an ASP.NET project called "CS-ASP\_013" and include the following Server Controls:



The programmatic IDs for these Controls are:

- (1) firstTextBox
- (2) comparisonTypeLabel
- (3) secondTextBox
- (4) checkedCheckBox
- (5) okButton
- (6) resultLabel

## Step 2: Writing Code in the Page\_Load Event

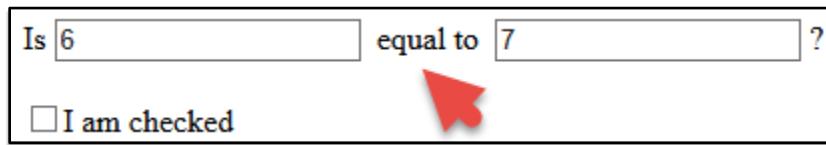
---

This time we'll write some code in the Page\_Load event, as well as the okButton\_Click event:

```
protected void Page_Load(object sender, EventArgs e)
{
    comparisonTypeLabel.Text = "equal to";
}

protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = (firstTextBox.Text == secondTextBox.Text) ? "Yes" : "No";
}
```

When you run the application, notice how the Label being assigned to, in the Page\_Load event, is set as soon as the page loads:



## Step 3: List of Comparison Operators

---

Now turning to comparison operators, here is a list that includes the comparison being done as well as the types they operate on:

Comparison Operator	What It Compares	Comparison Type
==	Equivalence	Mathematical, Other
!=	Not equivalent	Mathematical, Other
<	Less than	Mathematical
>	Greater than	Mathematical
<=	Less than, or equal to	Mathematical
>=	Greater than, or equal to	Mathematical

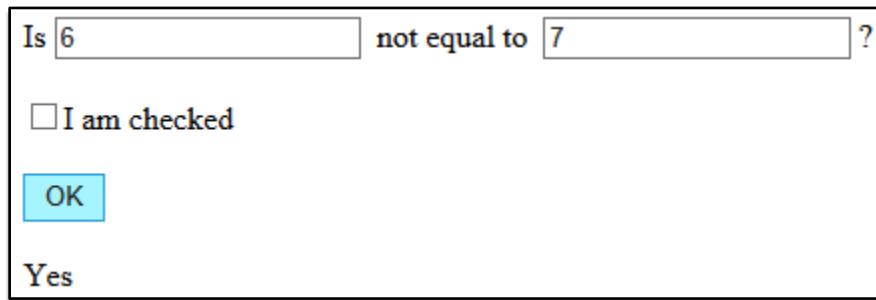
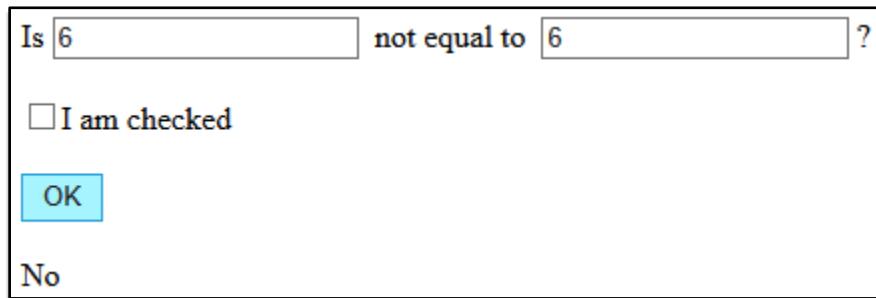


Bob's Tip: comparison operators imply that you are *comparing* two different things. As such, these are also referred to as *binary operators*, because they always operate on two different operands; one on each side of the operator.

Let's test out these different comparison operators by modifying the code and running the application:

```
protected void Page_Load(object sender, EventArgs e)
{
    comparisonTypeLabel.Text = "not equal to";
}

protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = (firstTextBox.Text != secondTextBox.Text) ? "Yes" : "No";
}
```



Now, try using the “greater-than” operator in the same way, and you will see that an error prevents us from moving forward. The problem is that we are attempting to use a mathematical comparison operator on the entered values as strings:

```
resultLabel.Text = (firstTextBox.Text > secondTextBox.Text) ? "Yes" : "No";
```

Visual Studio IDE screenshot showing a tooltip for the 'firstTextBox' field in the code above. The tooltip says: '+ (field) TextBox Default.firstTextBox firstTextBox control.' Below the code, a red squiggly underline is under the word 'firstTextBox' in the line 'resultLabel.Text = (firstTextBox.Text > secondTextBox.Text) ? "Yes" : "No";'. A tooltip for this underlined text says: 'Operator '>' cannot be applied to operands of type 'string' and 'string''.

The solution to this problem goes back to the conversion lesson that showed you how to convert a string to an int with the int.Parse() method:

```
protected void Page_Load(object sender, EventArgs e)
{
    comparisonTypeLabel.Text = "greater than";
}

protected void okButton_Click(object sender, EventArgs e)
{
    int first = int.Parse(firstTextBox.Text);
    int second = int.Parse(secondTextBox.Text);
    resultLabel.Text = (first > second) ? "Yes" : "No";
}
```

When you run the application, you will now see that the comparison operator works as expected, provided that you only enter integer values:

A screenshot of a Windows-style dialog box. At the top, it asks "Is 7 greater than 7 ?". Below this, there is a checkbox labeled "I am checked". A blue "OK" button is at the bottom left, and the word "No" is at the bottom right.

A screenshot of a Windows-style dialog box. At the top, it asks "Is 8 greater than 7 ?". Below this, there is a checkbox labeled "I am checked". A blue "OK" button is at the bottom left, and the word "Yes" is at the bottom right.



Bob's Tip: it should be clear, at this point, how the rest of the mathematical comparison operators work. Just be sure to use `>=` and `<=` rather than `=<` or `=>`, when performing a greater-than-or-equal-to, or less-than-or-equal-to check. It's easy to reverse them. And in the case of `=>`, it's a totally different operator which is used for something called a lambda expression.

## Step 4: Binary vs Unary Operators

So far we've been looking at the comparison operators that are said to be binary (meaning "two") because they evaluate two different operands on opposite sides of the operator. However, there are also unary operators (meaning "one") which are used to place an evaluation on a single operand. The most common unary operator, used within conditional expressions (although, not exclusively), is the single-exclamation operator, which can be read as saying "not." You can place this operator on anything that evaluates to a bool , as follows:

```
protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = (!checkedCheckBox.Checked) ? "No" : "Yes";
}
```

This now evaluates as true only when *it is not the case that* `checkedCheckBox.Checked` is true. For this reason, we had to flip the resulting strings to accommodate this change and work as it did before:

## Step 5: Compound Expressions in a Single Evaluation

You can also group expressions together to form more complex evaluations, using the "and" and "or" logical operators, which are also binary operators:

```

protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = ""; //clearing the text

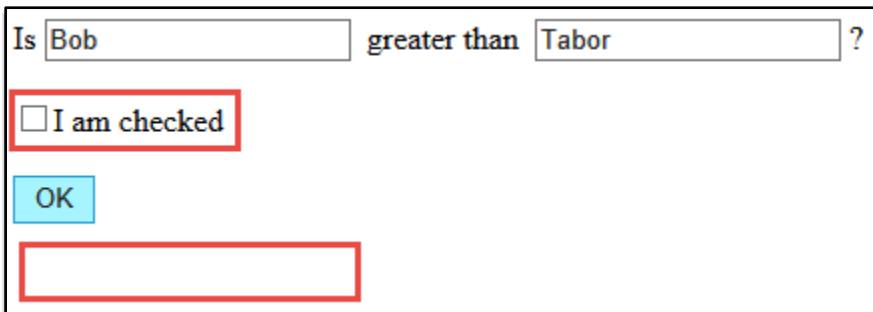
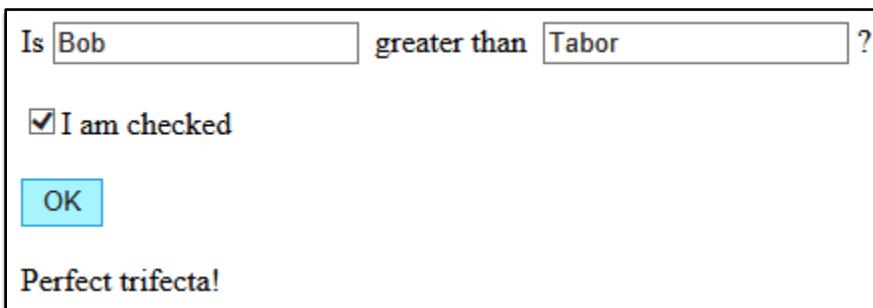
    if (checkedCheckBox.Checked ①
        && firstTextBox.Text == "Bob" ②
        && secondTextBox.Text == "Tabor") ③
    {
        resultLabel.Text = "Perfect trifecta!";
    }
}

```

Here the `&&` operator (representing “and”) is used to combine expressions to be evaluated within a single `if()` statement. This evaluation will only evaluate as true if:

- (1) `checkedCheckBox.Checked` is true
- (2) `firstTextBox.Text == "Bob"` is true
- (3) `secondTextBox.Text == "Tabor"` is true

If any of those expressions individually evaluate as false, then the entire conditional statement evaluates as false. And, once again, don’t let the formatting fool you: these expressions were placed on separate lines only for the sake of visual clarity. Run the application and experiment with different combinations of bool evaluations:

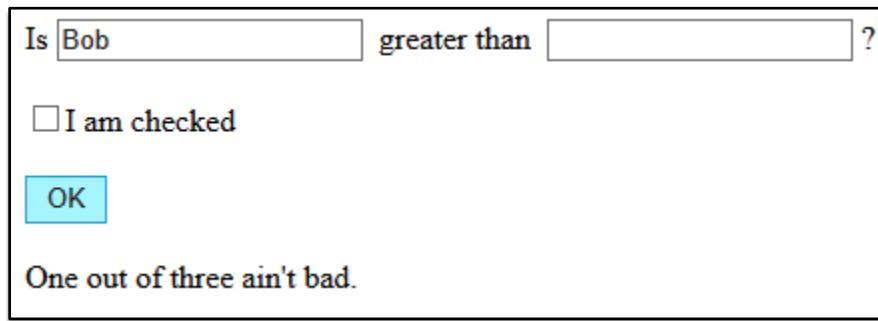
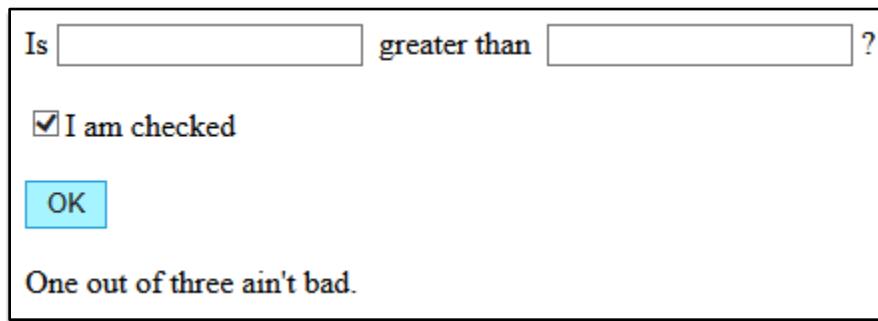


We can be more flexible with the conditional evaluation by using the logical "or" operator, same as "`||`":

```
protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = ""; //clearing the text

    if (checkedCheckBox.Checked
        || firstTextBox.Text == "Bob"
        || secondTextBox.Text == "Tabor")
    {
        resultLabel.Text = "One out of three ain't bad.";
    }
}
```

In this case, the conditional statement will evaluate as true if any of the three individual expressions evaluate as true:



Is  greater than  ?

I am checked

**OK**

One out of three ain't bad.



Bob's Tip: it should be noted that more than one of these three expressions could evaluate true and the entire statement will evaluate as true. In Hockey, you get your name on the scoreboard if you score a goal *or* get an assist. Therefore, it's still true that you get on the scoreboard if you score, both, a goal *and* an assist.

You can also combine the `||` and `&&` operators together for more complex evaluations:

```
if (checkedCheckBox.Checked
    || firstTextBox.Text == "Bob"
    && secondTextBox.Text == "Tabor")
{
    resultLabel.Text = "Two out of three ain't bad.";
}
```

However, the `&&` operator has precedence (just as multiplication/division has precedence in math) thereby grouping the comparison between `firstTextBox.Text` and `secondTextBox.Text`:

Is  greater than  ?

I am checked

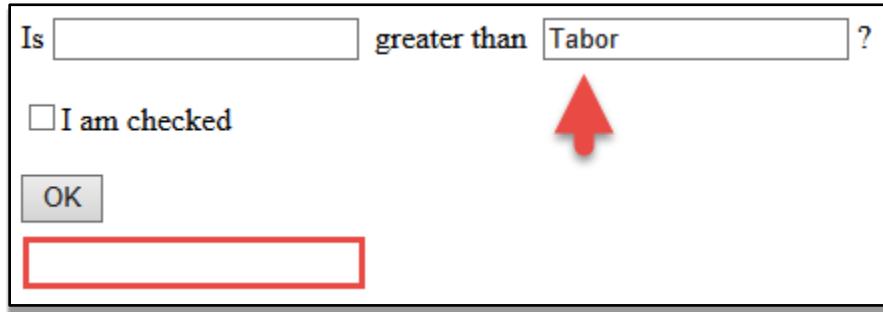
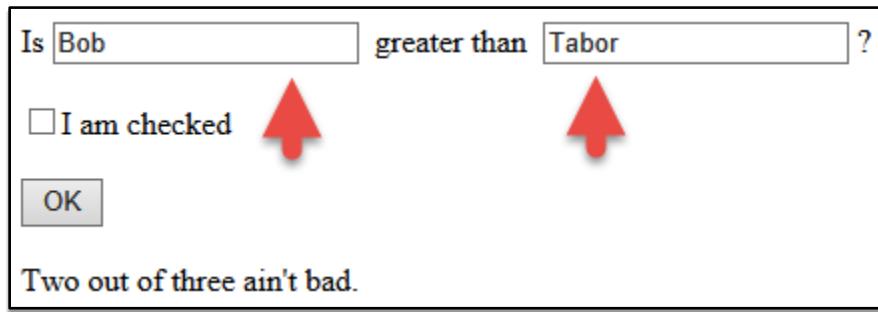
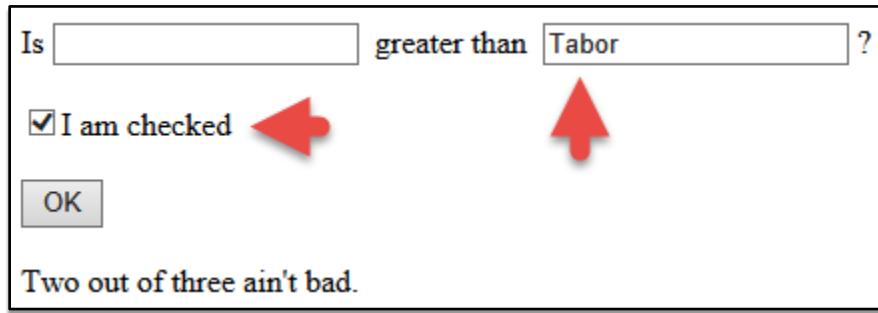
**OK**

Two out of three ain't bad.

This is not the result we're looking for, so let's create our own precedence by wrapping the evaluation in parentheses, grouping the first comparison between `checkedCheckBox.Checked` and `firstTextBox.Text`. Now the entire statement evaluates as true only when, either,

checkedCheckBox. Checked or firstTextBox. Text are true, and secondTextBox. Text is also true:

```
if ((checkedCheckBox.Checked || firstTextBox.Text == "Bob")  
    && secondTextBox.Text == "Tabor")
```



014

# Working with Dates and Times

There are four primitive data types that you will be consistently working with throughout your C# programming career:

- string
- int
- double
- bool

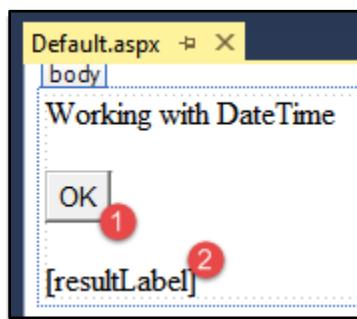
In this lesson, you will be introduced to another data type that allows you to work with dates and times. It's called `DateTi me`, but unlike the previously mentioned simple data types, `DateTi me` is a complex data type. It's complex in that it is a class, defined in the .NET Framework class library, that itself is composed of a variety *other* data types. A `DateTi me` variable (or *object*) contains within it, for example, three separate `int` variables called `Second`, `Mi nute`, and `Hour`, which hold those individual time values. Now, this topic – relating to classes, object-oriented programming, and the .NET framework - is outside of the scope of this lesson. but it's worth noting up front the difference between complex and simple data types.

## Step 1: Create a New Project

---

For this demonstration, create a new ASP.NET project and call it "CS-ASP\_014." Set up `Default.aspx` with the following Server Controls, and programmatic IDs:

- (1) `okButton`
- (2) `resultLabel`



## Step 2: Create and Access a Variable of Type DateTime

---

In the okButton\_Click event create a DateTime variable, and set it to the current time:

```
protected void okButton_Click(object sender, EventArgs e)
{
    DateTime myValue = DateTime.Now;
}
```

When you hover over a property, such as the "Now" property in DateTime.Now, you will often find a helpful tip about what it represents:



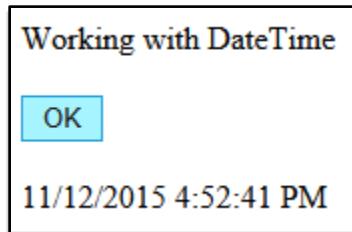
Now that we have the current time, let's try to output it by assigning it to resultLabel.Text:

```
DateTime myValue = DateTime.Now;
resultLabel.Text = myValue;
```

(local variable) **DateTime** myValue  
Cannot implicitly convert type 'System.DateTime' to 'string'

All you need to do to make the error disappear, is convert to a string using ToString(), and you can now run the application:

```
resultLabel.Text = myValue.ToString();
```



## Step 3: Formatting Date Values Returned from DateTime.Now

---

Keep in mind that the formatting order shown here (Day/Month/Year, Hour/Minute/Second, AM/PM) is dependent on your Windows localization settings on your computer. There are other ways of formatting the current time, and there are a number of helper methods you can access to facilitate this. Go ahead and try a few of the "To" methods available to DateTime variables:

```
DateTime myValue = DateTime.Now;
resultLabel.Text = myValue.To
    ↓
    ⓘ ToFileTimeUtc
    ⓘ ToLocalTime
    ⓘ ToLongDateString
    ⓘ ToLongTimeString
    ⓘ ToOADate
    ⓘ ToShortDateString
```

Write out these methods to see the different formatting results they produce (be sure to comment out the ones you are not currently testing to avoid getting a compilation error):

```
DateTime myValue = DateTime.Now;

resultLabel.Text = myValue.ToString(); 1
resultLabel.Text = myValue.ToString("T"); 2
resultLabel.Text = myValue.ToString("d"); 3
resultLabel.Text = myValue.ToString("t"); 4
```

Thursday, November 12, 2015 1

6:32:54 PM 2

11/12/2015 3

6:33 PM 4

Test out these other helper methods, which require a `ToString()` appended on the end since each of these return values that are not of type `string`:

```
resultLabel.Text = myValue.AddDays(2).ToString(); ①  
resultLabel.Text = myValue.AddMonths(-2).ToString(); ②  
  
resultLabel.Text = myValue.Month.ToString(); ③  
resultLabel.Text = myValue.IsDaylightSavingTime().ToString(); ④  
resultLabel.Text = myValue.DayOfWeek.ToString(); ⑤  
resultLabel.Text = myValue.DayOfYear.ToString(); ⑥
```

11/14/2015 6:41:32 PM ①

9/12/2015 6:42:30 PM ②

11 ③

False ④

Thursday ⑤

316 ⑥

Here is what each of these methods, specific to DateTime, do:

- (1) Returns a DateTime two days forward from the current DateTime.
- (2) Returns a DateTime two months backward from the current DateTime.
- (3) Returns the month, as an int, from the current DateTime.
- (4) Returns a bool expressing if the current DateTime is at a point during daylight savings.
- (5) Returns the day, as a string, from the current DateTime.
- (6) Returns the day in the year, as an int, from the current DateTime.

## Step 4: Parsing a String into a DateTime via DateTime.Parse()

---

Next, let's feed a particular date into a method that parses through it (performing an algorithm) and returns DateTime information it was able to parse from it:

```
DateTime myValue = DateTime.Parse("12/7/1969");  
resultLabel.Text = myValue.ToString();
```

Sunday, December 07, 1969



Bob's Tip: the date here – 12/7/1969 – happens to be my birthday, which I know occurred on a Sunday. You may find it more interesting to input your own birthday to see what day you were born on.

Alternatively, you can also use the new keyword and initialize the DateTime by passing the values in between the parentheses:

```
1   2   3   4   5   6  
DateTime myValue = new DateTime(1969, 12, 7, 6, 30, 0);  
resultLabel.Text = myValue.ToString();
```

Here, we're setting the properties stored in myValue to:

- (1) Year
- (2) Month
- (3) Day
- (4) Hour
- (5) Minute
- (6) Second

You have many different ways of initializing the DateTime, and you can get a list of them by typing in the parentheses and using the up/down arrows on your keyboard to cycle through the variations:

```
DateTime myValue = new DateTime()  
▲ 4 of 12 ▼ DateTime(int year, int month, int day)  
    Initializes a new instance of the DateTime structure to the specified year, month, and day.  
    year: The year (1 through 9999).
```

This is a preview of both a special kind of method called a "Constructor," as well as what is called "Constructor Overloading." There isn't anything very fancy about Constructors other than they are methods with the same name as the type they belong to – in this case, the Constructor DateTime() belongs to DateTime – and they are used for initializing certain values determined by what's written in the Constructor. Overloading, meanwhile, refers to variations of the same Constructor that have different initialization procedures. Don't worry too much about understanding this at this point as the process will be made much clearer in subsequent lessons.

015

# Working with Spans of Time

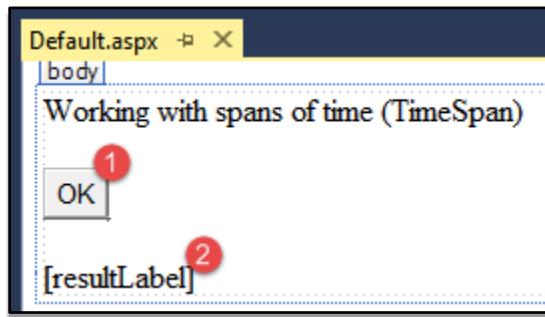
This lesson will deal with a type that is related to `DateTi me`, called `Ti meSpan`. The `Ti meSpan` type represents the amount of time elapsed between two `DateTi me` objects. For example, if you want to determine how long you've been alive you could use today's date, along with your birthday, and the elapsed time in between would be represented using `Ti meSpan`.

## Step 1: Create a New Project

---

Begin by creating a project, calling it "CS-ASP\_015," and setting up `Default.aspx` with the following Server Controls, and programmatic IDs:

- (1) `okButton`
- (2) `resultLabel`



In the `okButton_Click` event, create a `Ti meSpan` variable and assign it the result of the `Ti meSpan.Parse()` method:

```
protected void okButton_Click(object sender, EventArgs e)
{
    TimeSpan myTimeSpan = TimeSpan.Parse()
}
```

▲ 1 of 2 ▼ `TimeSpan TimeSpan.Parse(string s)`  
Converts the string representation of a time interval to its `TimeSpan` equivalent.  
`s: A string that specifies the time interval to convert.`

You see here that the `Parse()` method takes in a string input, but the hint as to what this method does is unclear. To better understand what we can use as a valid input *argument* in this method, let's research it by navigating to Microsoft's documentation for this method:

<http://is.gd/timespan>

The most important documentation on this page shows us exactly how to format the string argument:

“The *s* parameter contains a time interval specification in the form:

[*ws*][-]{ *d* | [*d.*]*hh:mm[:ss[.ff]]* }[*ws*]

Elements in square brackets ([ and ]) are optional. One selection from the list of alternatives enclosed in braces ({ and }) and separated by vertical bars (|) is required. The following table describes each element.”

Element	Description
<i>ws</i>	Optional white space.
-	An optional minus sign, which indicates a negative TimeSpan
<i>d</i>	Days, ranging from 0 to 10675199.
.	A culture-sensitive symbol that separates days from hours. The invariant format uses a period (".") character.
<i>hh</i>	Hours, ranging from 0 to 23.
:	The culture-sensitive time separator symbol. The invariant format uses a colon (":") character.
<i>mm</i>	Minutes, ranging from 0 to 59.
<i>ss</i>	Optional seconds, ranging from 0 to 59.
.	A culture-sensitive symbol that separates seconds from fractions of a second. The invariant format uses a period (".") character.
<i>ff</i>	Optional fractional seconds, consisting of one to seven decimal digits.

Using this guide, write out the intended formatting in a comment as a reminder. Pay extra care to the particular separator this method demands for each time element in the string - either a colon (:), or a period (.) :

```
// Days.Hours:Minutes:Seconds.Milliseconds  
TimeSpan myTimeSpan = TimeSpan.Parse("");
```



Bob's Tip: this string appears to be formatted arbitrarily, and it appears that way because it is! Actually, to be precise, the programmer who wrote this method decided, somewhat arbitrarily, on this particular string formatting that the method accepts. As such, this is not a syntax pattern specific to C#.

## Step 2: Use TimeSpan.Parse() to Return a TimeSpan

---

Using the comment as a guide, write an acceptable set of time elements as a string input for this method:

```
TimeSpan myTimeSpan = TimeSpan.Parse("1.2:3:30.5");
```

This is essentially assigning `myTimeSpan` with a `TimeSpan` that represents 1 Day, 2 Hours, 3 Minutes, and 30.5 Seconds. This doesn't appear immediately useful, so let's combine a `TimeSpan` with a `DateTime` to demonstrate its practical use:

```
DateTime myBirthday = DateTime.Parse("12/7/1969"); ①  
TimeSpan myAge = DateTime.Now.Subtract(myBirthday); ②
```

What this is doing is:

- (1) Store a `DateTime` – returned for the `Parse()` method – into `myBirthday`
- (2) Take the `DateTime` given by `DateTime`.`Now` – which is the current date – and call the `Subtract()` method from it, inputting `myBirthday` as the required `DateTime` argument. And then store the returned result of that method into `myAge`.

That sequence may seem convoluted and that is partly because it mentions concepts that we haven't completely covered yet. However, the code is represented in a way that is readable and, hopefully, clearly communicates its intent.



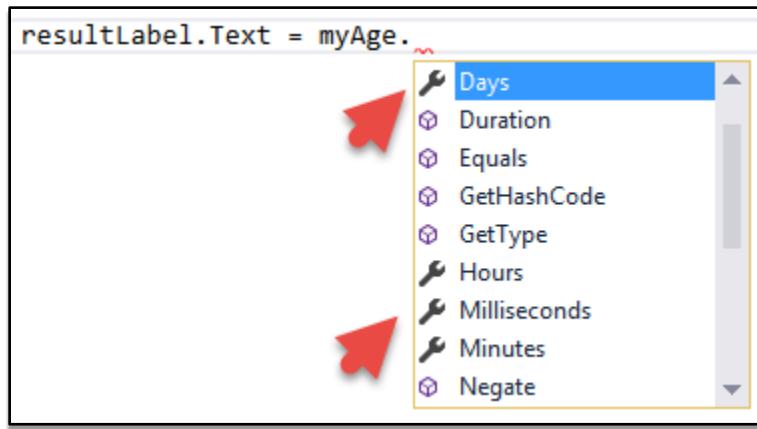
Bob's Tip: you can add a link to important documentation directly in your code by making it a comment:

```
protected void okButton_Click(object sender, EventArgs e)
{
    //http://is.gd/timespan
    TimeSpan myTimeSpan = TimeSpan.Parse("");
}
```

Now, by pressing ctrl+left-click on the link, you can view the web page directly inside of Visual Studio!

### Step 3: Access Properties Available to a TimeSpan Variable

Now that we have a `TimeSpan` object, stored in `myAge`, we can comb through various properties of that `TimeSpan` (shown with the Intellisense icon that looks like a wrench):



The most interesting option from this list of methods and properties is to output the total days/months/hours within a given `TimeSpan`. Let's output the `Total Days` property that, because it is a double, needs to be converted to a string with `ToString()`:

```
resultLabel.Text = myAge.TotalDays.ToString();
```

## Working with spans of time (TimeSpan)

OK

16776.9105376337

With that, you have most of what you need to know about working with time in C#. With these two types – DateTime and TimeSpan – you will be able to represent, both, a specific moment of time and an elapsed period of time between two moments, as well as the various operations you can perform on these two time elements.

016

# Working with the Calendar Server Control

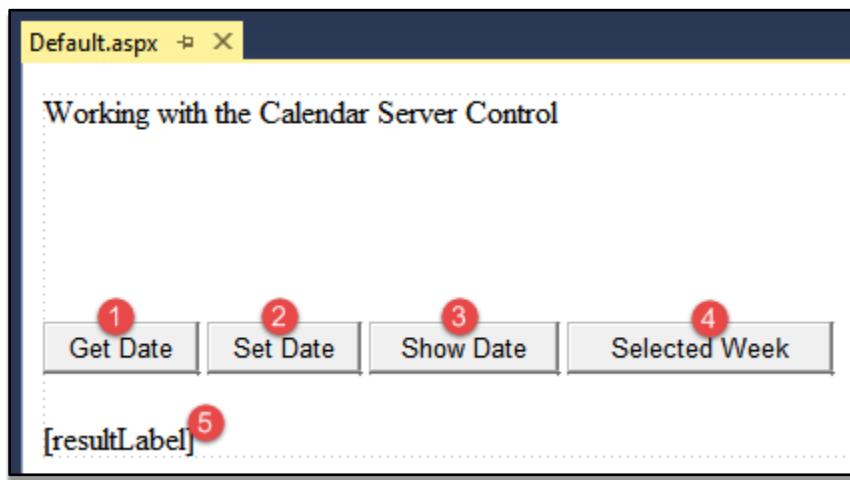
This lesson will show you how to use the Calendar Server Control which will make use of the DateTime and TimeSpan types you've learned about. This Control is useful for providing a visual component that represents a date, in a web page, in a way that users are familiar with.

## Step 1: Create a New Project

---

Create a new project called "CS-ASP\_016" and set up a *Default.aspx* with the following Server Controls, and programmatic IDs:

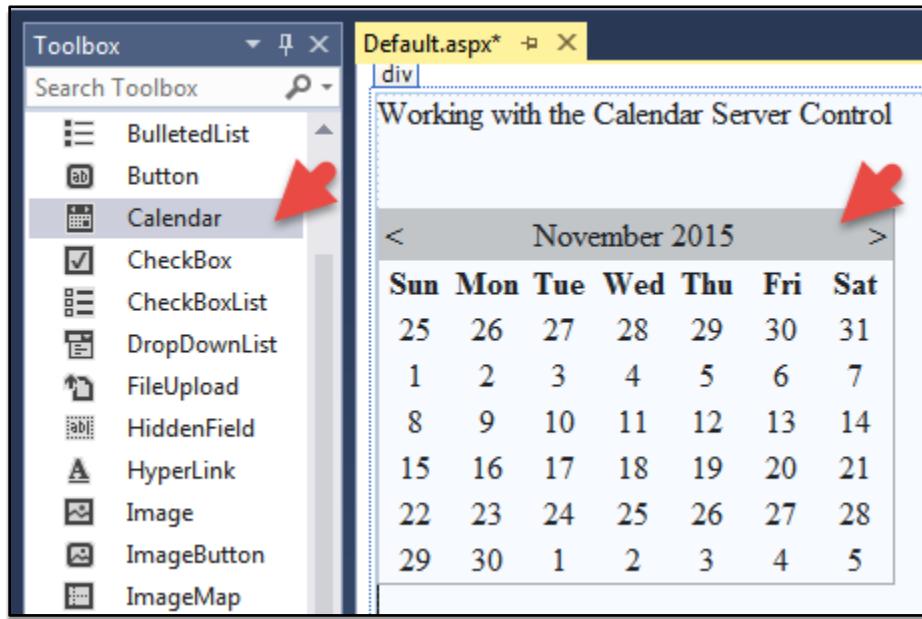
- (1) getDateButton
- (2) setDateButton
- (3) showDateButton
- (4) selectedWeekButton
- (5) resultLabel



## Step 2: Add a Calendar Server Control

---

Now, let's place a Calendar Server Control in between the text and buttons:



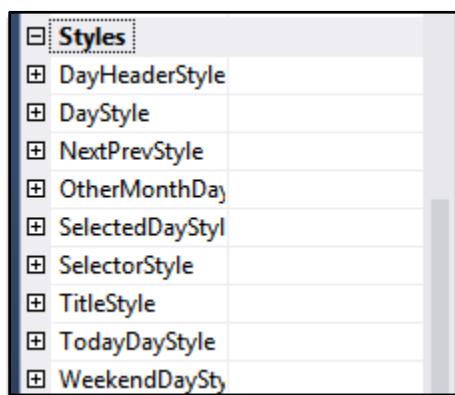
Select the Calendar, take a look at its Properties Window and set the ID to "myCalendar":



## Step 3: Formatting the Calendar Properties

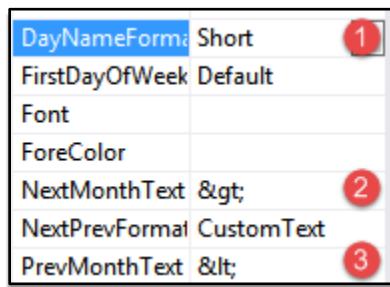
---

While in the Properties Window, take note of the various built-in styles available:



Also take note of some useful settings in the "Appearance" section:

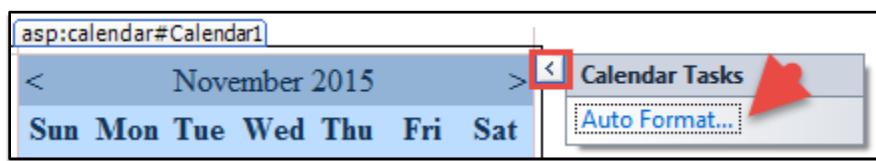
- (1) Day formatting (single letter, three letters, full words, and so on).
- (2) HTML symbol for next Month (greater than, or ">").
- (3) HTML symbol for the previous month (less than, or "<").



## Step 4: Styling the Calendar with Presets

---

You can access a selection of pre-formatted styles to choose from by clicking on the arrow to the right of the Calendar Control:



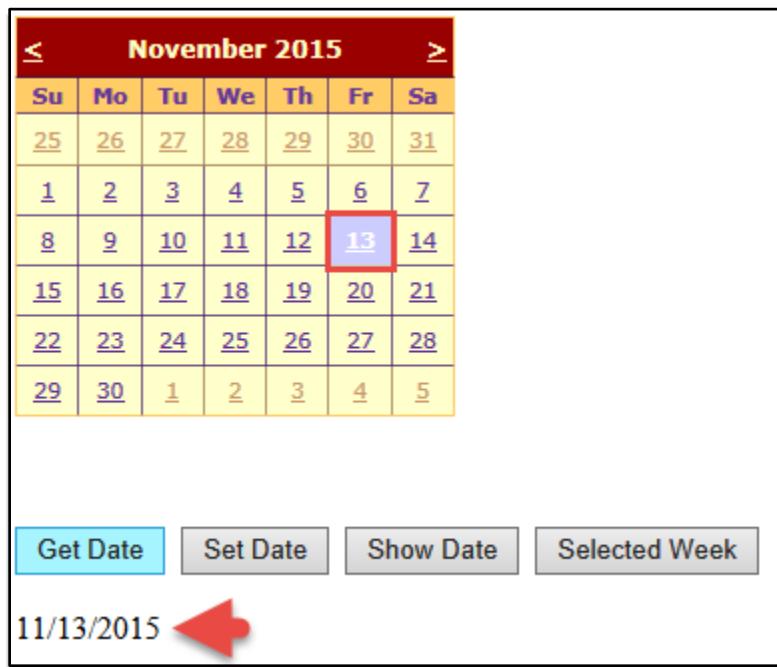
## Step 5: Referencing the Calendar via Code

---

Now let's turn to the *Default.aspx.cs* file and write in the following code for the *getDateButton\_Click* event:

```
protected void getDateButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = myCalendar.SelectedDate.ToShortDateString();
}
```

Now run the application, select a date, and the "Get Date" button to display the current date formatted as a string type.



Now, let's have the "Set Date" button change the date of the Calendar when clicked:

```
protected void setDateButton_Click(object sender, EventArgs e)
{
    myCalendar.SelectedDate = DateTime.Parse("6/1/2014");
    myCalendar.VisibleDate = myCalendar.SelectedDate;
}
```

June 2014						
Su	Mo	Tu	We	Th	Fr	Sa
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

Since the `VisibleDate` property can be set to any `DateTime`, let's demonstrate this by writing the following in `showDateButton_Click`:

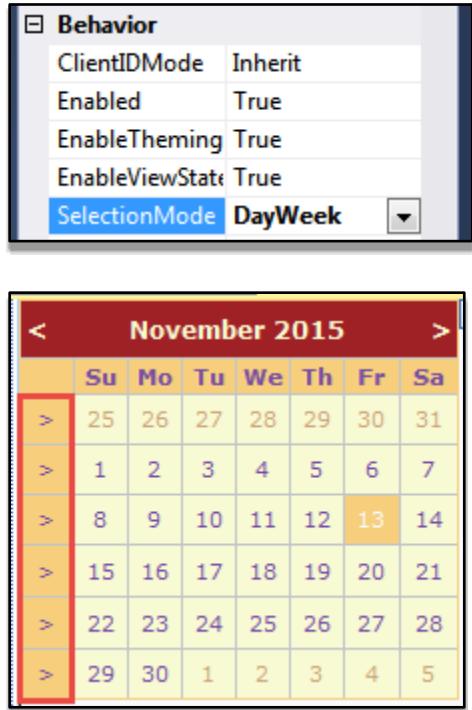
```
protected void showDateButton_Click(object sender, EventArgs e)
{
    myCalendar.VisibleDate = DateTime.Parse("12/7/1969");
}
```

December 1969						
Su	Mo	Tu	We	Th	Fr	Sa
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

## Step 6: Add the Option to Select an Entire Week

---

Go back to the Properties Window for the Calendar and adjust the SelectionMode property to "DayWeek," and notice how it adds a selection arrow for each week in the month.

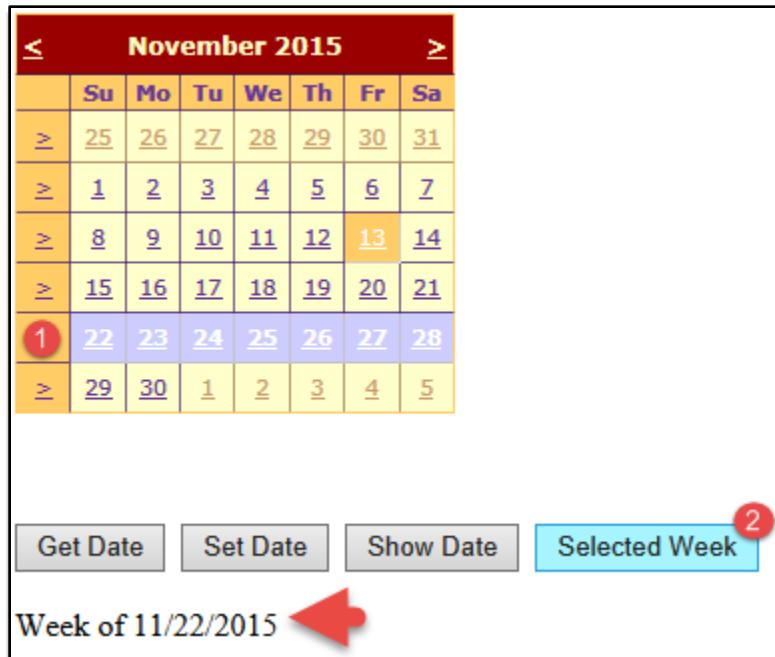


Next, enter the following code that displays the currently selected date when you click on the "Selected Week" button:

```
protected void selectedWeekButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = "Week of " + myCalendar.SelectedDate.ToShortDateString();
}
```

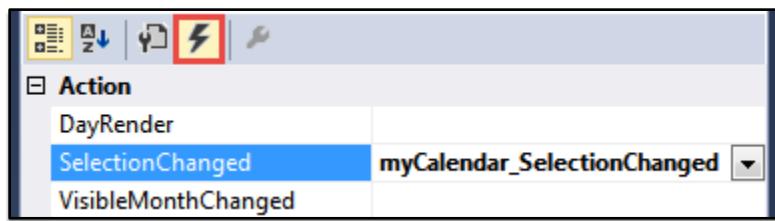
When you run the application you will have to:

- (1) Select a week on the Calendar.
- (2) Click on the "Selected Week" button.



## Step 7: Utilizing the SelectionChanged Event

One more thing worth noting is the event – visible in the Calendar's Properties Windows – that fires whenever any selection is changed on the Calendar:



Probably the simplest way to demonstrate what this event does is to write the following in the method attached to this event in *Default.aspx.cs*:

```
protected void myCalendar_SelectionChanged(object sender, EventArgs e)
{
    resultLabel.Text = myCalendar.SelectedDate.ToShortDateString();
}
```

Now, when you run the application and click on individual dates, you will notice that the selected date is automatically displayed in the Label Control. Unlike the previous events that have been firing upon a button click, this event fires with the mere act of clicking on any given date on the Calendar:

November 2015						
	Su	Mo	Tu	We	Th	Fr
≥	<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>
≥	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>
≥	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>
≥	<u>15</u>	<u>16</u>	<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>
≥	<u>22</u>	<u>23</u>	<u>24</u>	<u>25</u>	<u>26</u>	<u>27</u>
≥	<u>29</u>	<u>30</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>

Get DateSet DateShow DateSelected Week

11/23/2015

017

# Page\_Load and Page.IsPostBack

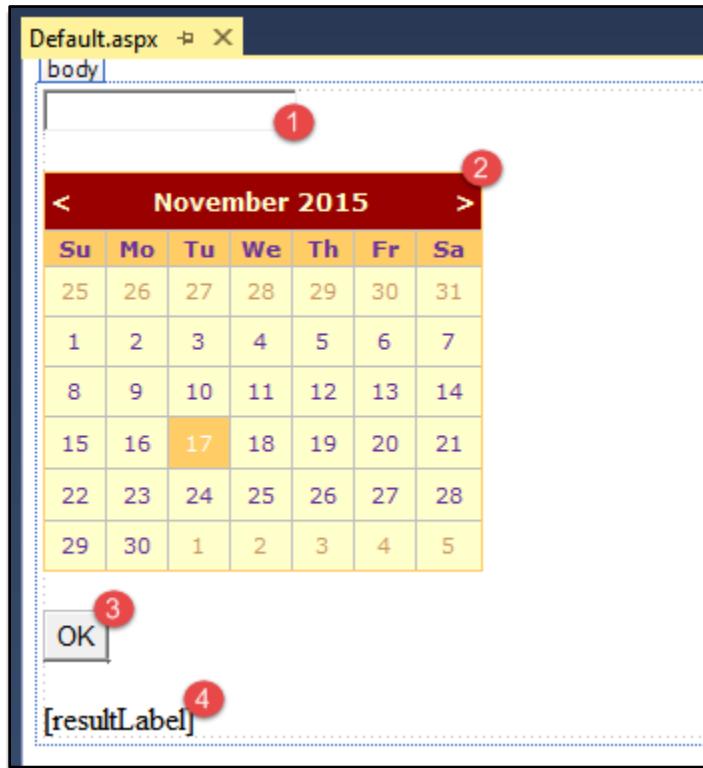
In this lesson, we're going to take a slight departure from pure C# and talk about something ASP.NET specific. While generally avoiding the minutiae of ASP.NET, the specific topic of this lesson will hopefully lead you towards building more interesting C# applications.

## Step 1: Create a New Project

---

To illustrate this, create a new ASP.NET project called "CS-ASP\_017." This project is based on the previous lesson, and has a *Default.aspx* with the following Server Controls and programmatic IDs:

- (1) myTextBox
- (2) myCalendar
- (3) okButton
- (4) resultLabel



## Step 2: Setting Up the PostBack Problem

---

In *Default.aspx.cs*, write the following events:

```
protected void Page_Load(object sender, EventArgs e)
{
    myTextBox.Text = "Some value";
    myCalendar.SelectedDate = DateTime.Now.Date.AddDays(2);
}

protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = myTextBox.Text + " - "
        + myCalendar.SelectedDate.ToShortDateString();
}
```

The `Page_Load` event simply assigns, upon the page being loaded, the `myTextBox.Text` property with a default string, as well as the selected date to two days ahead of the current date. Be careful to reference the `Date` property that is a part of the `Now` property - by writing `Now.Date` - otherwise `myCalendar.SelectedDate` will hold a specific time (including hour/minutes, which is not how calendars work). Meanwhile, the `okButton_Click` event simply outputs the value contained in `myTextBox.Text` concatenated with the value in `myCalendar.SelectedDate`.

## Step 3: Understanding the PostBack Problem

---

Setting up a form, or calendar, with default values is a common task however there is a problem with this code that we wrote. If you run the application, you will notice that when you change the values for the Calendar or TextBox, those changes are never reflected after submitting them upon clicking the button. That's because the default values always reload after every submission. The easiest solution to this problem is to include code that does one thing (assign default values) if it's the first time the page is loaded, and do another thing if it's not (ignore the default values).

## Step 4: Referencing the IsPostBack Property

---

The easiest way to branch off these two separate cases is to determine whether or not the page was loaded by the bool `IsPostBack` property. Simply put, `PostBack` occurs when the "OK" button is clicked, and you can't `PostBack` unless you first got to the page from some other means (from a direct-link, for example). Therefore, if the page was loaded via the "OK" button (`PostBack`) we can safely ignore the default values, and instead have the code read-in the user-input values.

## Step 5: Resolving the Problem with a Conditional Check

---

Here is how we resolve the issue in code:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        myTextBox.Text = "Some value";
        myCalendar.SelectedDate = DateTime.Now.Date.AddDays(2);
    }
}
```

Noticed how the conditional checks if the page is loaded *not* by PostBack. Checking for the opposite would involve a bit more code and not improve readability:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (IsPostBack) 
    {
        //Default behaviour. Do nothing special.
    }
    else
    {
        myTextBox.Text = "Some value";
        myCalendar.SelectedDate = DateTime.Now.Date.AddDays(2);
    }
}
```



Bob's Tip: whenever you have an empty conditional (a block of code that essentially does nothing if the condition is met), you should consider inverting the expression being evaluated to check the opposite of that condition instead. If nothing else, it will make your code cleaner and easier to read.

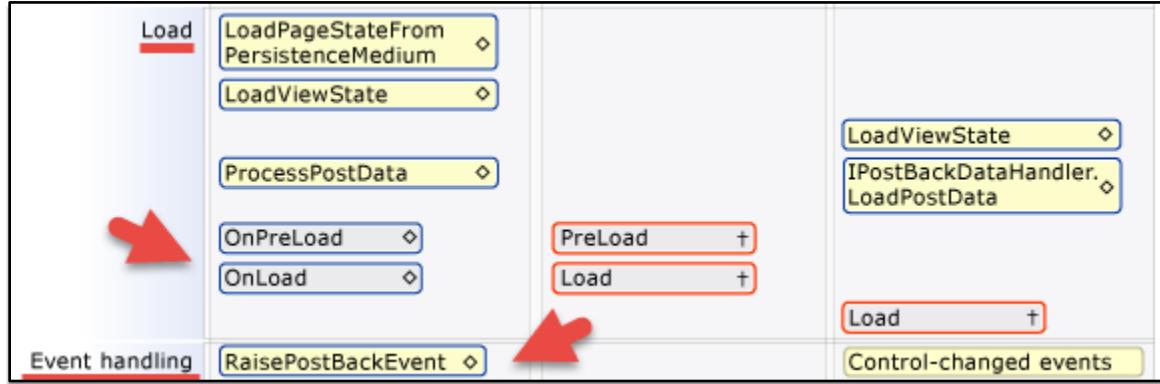
## Step 6: Understanding Code Execution and Event Timing

---

You may be wondering when exactly, during the page load process, the `IsPostBack` property is set. This refers to code execution/event timing, and is touched upon at the following URL:

<http://is.gd/postbackcycle>

In particular, notice how the event handling occurs after the initial Load procedure:



018

# Setting a Break Point and Debugging

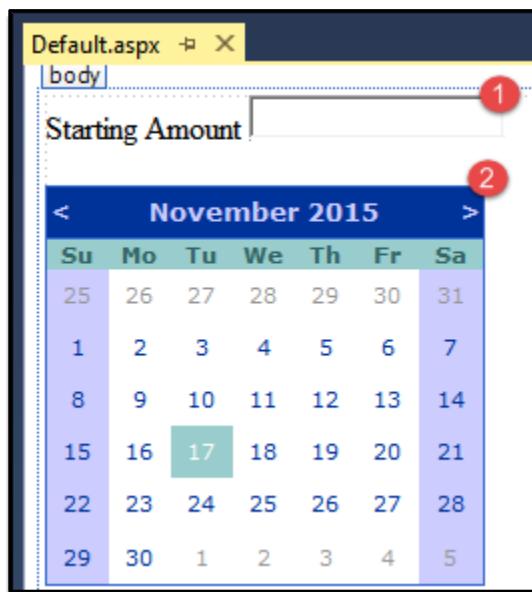
In this lesson, we're going to briefly cover the debugging features of Visual Studio. Debugging is probably the most crucial skill that you'll develop as a beginner because it's going to help you figure out where you went wrong in the code you've written. When you first encounter vexing problems in your code, you can stare at it and hope the solution dawns on you, or you can set a breakpoint. Setting a breakpoint is the first step in debugging, and determines where the execution of your code should halt so you, the developer, can take control. When execution is in your control you effectively slow down the code allowing you to inspect the values of variables and Server Controls while they change.

## Step 1: Create a New Project

---

For this lesson, create a new ASP.NET project and call it "CS-ASP\_018." Set the *Default.aspx* with the following Server Controls, and programmatic IDs:

- (1) myTextBox
- (2) firstCalendar
- (3) secondCalendar
- (4) okButton
- (5) resultLabel





## Step 2: Write Code for Debugging Purposes

---

In the *Default.aspx.cs* file write out the following code, purely for the purpose of demonstrating problem-solving via debugging:

```
protected void Page_Load(object sender, EventArgs e)
{
    myTextBox.Text = 500.ToString();
}

protected void okButton_Click(object sender, EventArgs e)
{
    if (firstCalendar.SelectedDate.AddDays(15) >= secondCalendar.SelectedDate)
    {
        TimeSpan elapsedDays = firstCalendar.SelectedDate.Subtract(secondCalendar.SelectedDate);
        double userValue = double.Parse(myTextBox.Text);
        double days = elapsedDays.TotalDays;
        double sum = userValue * days + 100;
        resultLabel.Text = sum.ToString();
    }
    else
    {
        resultLabel.Text = "Error. You must choose a date at least 14 days apart.";
    }
}
```

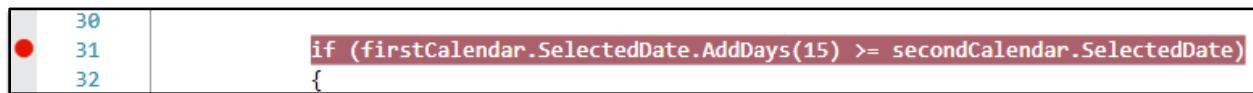
## Step 3: Set Break Points to Take Control of Execution

---

The imaginary rules, for this imagined application, are:

- (1) A user must select a date in secondCalendar that is at least 14 days from the selected date in firstCalendar. Otherwise, an error is displayed in the resultLabel.
- (2) The resultLabel should be the sum of the TextBox and the TimeSpan.TotalDays, obtained from the dates between both Calendars, multiplied by 100.

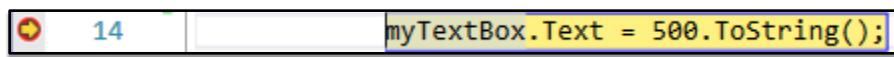
The application intentionally has some errors when attempting to perform the calculation. Let's now add a breakpoint to help track down the problem. You can either place your cursor on the line you want execution to halt and press the F9 key on your keyboard, or you can simply left-click on the margin beside that line of code:



You can set multiple breakpoints. To demonstrate this add another breakpoint to this line of code:

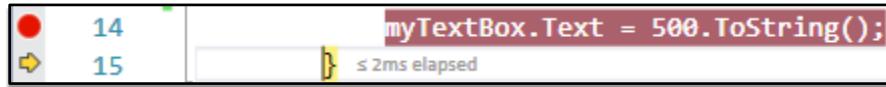
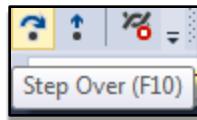


Now run the application and you will notice that focus will shift from the browser back to the first breakpoint in the code execution order:

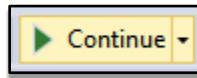


Bob's Tip: think of the red dot, which represents the breakpoint, as being the "stop" sign in your application. Your code will execute as normal, up until the point it reaches this stop sign, at which point further execution is handed off to you.

At this point, the application is in your control. You can move to the next line of code by hitting F10 on the keyboard or by clicking on the "Step Over" icon in the toolbar. You will then see an arrow indicating the next line of code to be executed:



If you no longer want to debug this part of code you can deselect the red dot, while still in debug mode, and click on the “continue” button in the toolbar. When clicking on the button, execution will then halt on the next breakpoint:



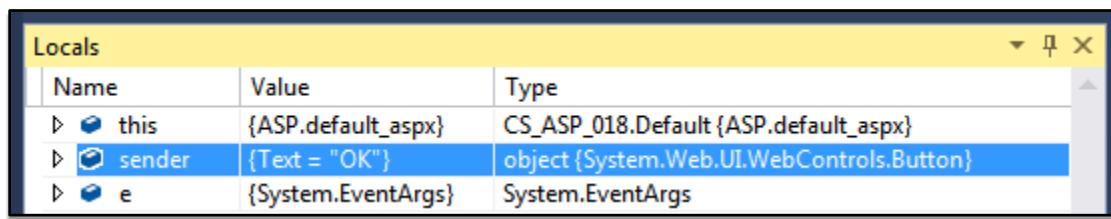
## Step 4: Inspecting Values in Variables as the Application Runs

You can then hover your mouse over different variables to inspect what they are holding at that given moment. Here we see something that would produce an error, and that is the date held in SelectedDate:



If you hover over the SelectedDate for secondCalendar, you will also find the same problem. Note that you can inspect the values of all variables, currently in scope, by referring to the “Locals” window.

Debug > Windows > Locals



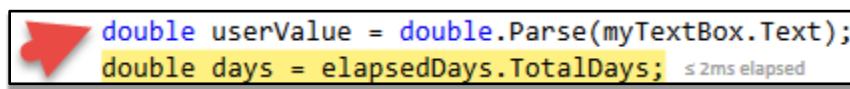
As you *step into* this code block (F11), notice how variables are added to the Locals and Watch Windows. The Watch Window shows you variables that you can keep track of even if they are not within the scope of the current code block. To open the Watch Window go to:

Debug > Windows > Watch

As you continue to step forward, notice how you can see variables taking on values in real-time:



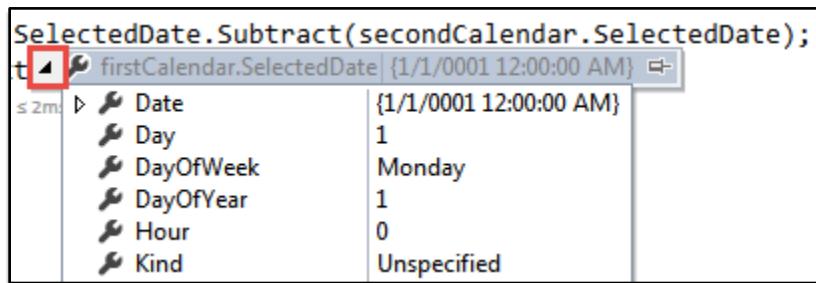
The red font indicates that this value has just changed in response to something that occurred on the previous line of code (here, it is being assigned with the myTextBox.Text value):



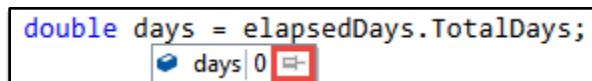
## Step 5: Pinning Down Particular Variables

---

You can also zero-in on particular variables by hovering over them. In this case, this particular variable, SelectedDate is a DateTime which means you can access constituent variables (properties) contained within it:



You can even pin certain variables down so you can see their values update with the editor itself (this stays pinned, even when done with debugging, so you can see the last value it held):



```
double userValue = double.Parse(myTextBox.Text);
double days = elapsedDays.TotalDays;
```

Continue to step into the code until you're at this line:

```
double days = elapsedDays.TotalDays;
double sum = userValue * days + 100;
resultLabel.Text = sum.ToString();
```

## Step 6: Real-Time Variable Assignment via Locals Window

You can even test out what would happen if variables held certain values. Go to the Locals Window and manually change days to 34:

days	34	double
------	----	--------

## Step 7: Stepping Backwards

It may seem that we changed the value for days a bit too late, as you may have noticed days was referenced on the previous line of code (in the calculation that ends up being assigned to sum). However, we can step *backward* to re-read the previous line, this time with the new value for days as set in the Locals Window. You can do this by right-clicking on the line you want to step to and select "Set Next Statement" (or, Ctrl+Shift+F10):

```
TimeSpan elapsedDays = firstCalendar.
double userValue = double.Parse(myTe
double days = elapsedDays.TotalDays;
double sum = userValue * days + 100;
resultLabel.Text = sum.ToString();
```

Now, when you manually step forward it will include this update to days and sum, which is then reflected in the resultLabel :

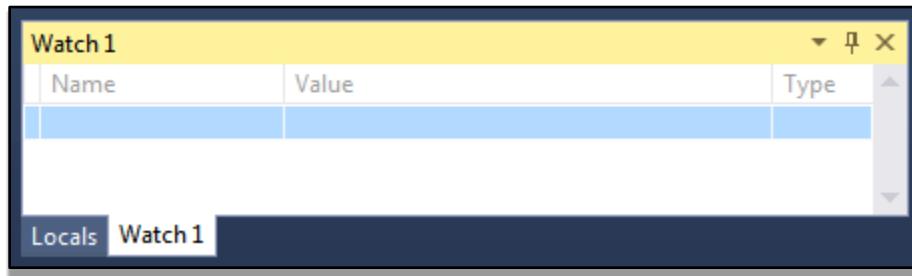
```
resultLabel.Text = sum.ToString();
resultLabel {Text = "17100"}
```

## Step 8: Using the Watch Window for Handling Properties

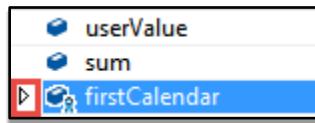
---

Unfortunately, not every variable is going to be so easily adjusted in real-time as we have seen here. Take, for instance, the `firstCalendar.SelectedDate` property which isn't even referenced in the current Locals Window, however, we can add it to the Watch Window by highlighting it, and then clicking/dragging it there:

```
if (firstCalendar.SelectedDate.
```



After you drag the `firstCalendar` property to the Watch Window, it will be accessible there:



## Step 9: Using the Immediate Window to Write Code In Real-Time

---

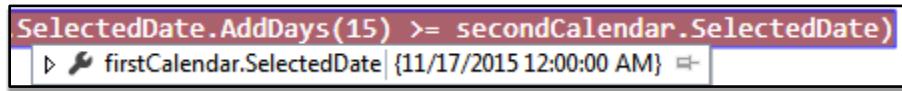
However, when you click on the arrow to inspect the constituent properties, you will notice that some are resistant to editing such as with `SelectedDate`. There is a way around this and that is by going to the Immediate Window which allows you to write code while debugging. You can display the Immediate Window by pressing **Ctrl+Alt+I** on your keyboard or via the menu:

Debug > Windows > Immediate

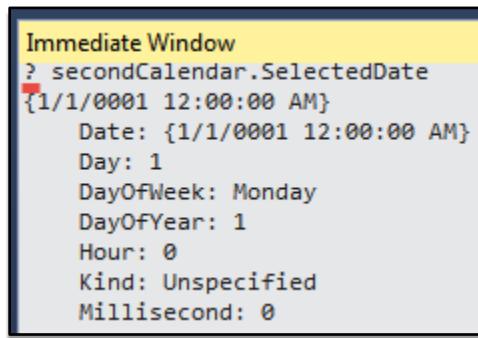
Type in the following code and hit enter to execute it:

```
Immediate Window
firstCalendar.SelectedDate = DateTime.Parse("11/17/2015");
```

Notice when you hover over SelectedDate you will see it storing the value you wrote in the Immediate Window:



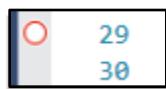
Back in the Immediate Window, you can also query any variable to see its current values, by inserting a question mark before the variable name:



When you stop debugging all of these temporary changes you made will be reset. However, pinned values are kept for reference purposes, and are found by clicking on the pin in the left margin:



Sometimes it's useful to temporarily disable a breakpoint while keeping the reminder that a breakpoint was used. You can do this by right-clicking on the breakpoint and selecting "disable breakpoint" (or `Ctrl+F9`):



**Bob's Tip:** armed with all of the information you learned from a debugging session, you should have a better understanding of how to go back and fix the problematic code. Debugging is therefore not an automatic solution to problems in your code, but more like a "magnifying glass" that clarifies the fuzzy areas.

019

# Formatting Strings

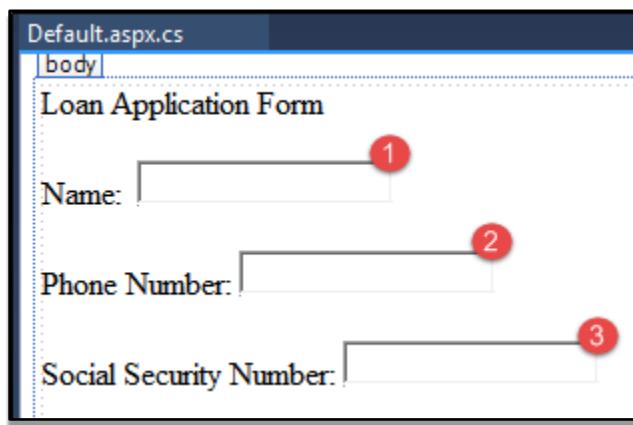
This lesson will cover how to format strings. We already looked at how to concatenate strings, with the “+” and “+=” operators, but this lesson will show you how to handle special formatting conditions, such as the patterns seen in things like phone, social security, or credit card numbers. These all have some form of patterned formatting, or otherwise have some special arrangement of numbers and characters.

## Step 1: Create a New Project

---

We'll demonstrate this with a very simple loan application form that stores pieces of information like the phone number, social security number, salary and date. It will then display all of this information back to resultLabel with special formatting so that it looks presentable. Begin this lesson by creating a new ASP.NET project called “CS-ASP\_019,” and set up a *Default.aspx* with the following Server Controls, and programmatic IDs:

- (1) nameTextBox
- (2) phoneTextBox
- (3) ssTextBox
- (4) loanDateCalendar
- (5) salaryTextBox
- (6) submitButton
- (7) resultLabel



The screenshot shows a Windows application window with the title "Loan Origination Date:". Inside, there is a calendar for November 2015. The days of the week are labeled from Sunday to Saturday. The dates are as follows:

Sun	Mon	Tue	Wed	Thu	Fri	Sat
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

Below the calendar, there is a label "Salary:" followed by a text input field. Underneath the input field is a "Submit" button. At the bottom of the window is a label "[resultLabel]".

Red numbers are overlaid on the interface:

- Number 4 is in a red circle above the calendar's right arrow button.
- Number 5 is in a red circle above the text input field.
- Number 6 is in a red circle above the "Submit" button.
- Number 7 is in a red circle above the "[resultLabel]" label.

The first thing we will want to do is take the applicants name that was input via nameTextBox, and output it within a formal "thank you" statement:

```
protected void submitButton_Click(object sender, EventArgs e)
{
    string result = "Thank you, " + nameTextBox.Text + ", for your business";
    resultLabel.Text = result;
}
```

## Step 2: String Formatting with string.Format()

---

This approach uses the string concatenation technique we have become familiar with. This will get the job done, but isn't as readable owing to the multiple uses of the concatenation operator. A much more readable way of achieving this is by calling the `string.Format()` method instead:

```
string result = string.Format("");
resultLabel.Text = result;
```

This method takes in (1) a literal string argument, along with a (2) variable string argument, and then using a (3) special placeholder it transposes the variable string argument into the placeholder position within the literal string:

```
string result = string.Format("Thank you, {0}, for your business", nameTextBox.Text);
```

The output when running the application will look like this:

Name:

Phone Number:

Social Security Number:

Loan Origination Date:

November 2015						
<u>Sun</u>	<u>Mon</u>	<u>Tue</u>	<u>Wed</u>	<u>Thu</u>	<u>Fri</u>	<u>Sat</u>
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>	<u>31</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>
<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>
<u>15</u>	<u>16</u>	<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	<u>21</u>
<u>22</u>	<u>23</u>	<u>24</u>	<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>
<u>29</u>	<u>30</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>

Salary:



Thank you, Bob, for your business

This essentially produces a string “template” with as many placeholders as needed:

```
string.Format("Thank you, {0}, for your business. Your Social Security number is: {1}",  
    nameTextBox.Text, ssTextBox.Text);
```

Submit

Thank you, Bob, for your business. Your Social Security number is: 123321123

The problem now is that the entered number itself should be formatted to look more like a typical Social Security number. We start by converting this value to an integer, and then storing it into an integer variable:

```
int ss = int.Parse(ssTextBox.Text);
```

## Step 3: Applying Special Formatting Patterns to the Placeholder

---

Modify the `string.Format()` method to take this `int` value and apply to it a formatting pattern of number sequences delimited by dashes. Here we took the `{1}` placeholder and appended to it a formatting pattern, with a colon separating the placeholder from its pattern:

```
string.Format("Thank you, {0}, for your business. Your Social Security number is: {1:000-00-0000}",  
    nameTextBox.Text, ss);
```

The formatting pattern will become much more apparent when running the application:

Thank you, Bob, for your business. Your Social Security number is: 123-32-1123



Bob's Tip: you should note that phone numbers are at least 10 digits in length, which reaches the upper-limits allowable for storage into an `int` variable. If the first digit in the phone number is greater than 1, there is a good chance it will cross the allowable threshold and return an error. To fix this, you can store the number into a `long`. However, if the application ends up processing thousands of numbers (or more) this could be very inefficient and you would probably want to leave the phone number as a `string`.

## Step 4: Inserting HTML into the Output String

---

You can also inject HTML straight into the output string, considering that the output is ultimately being delivered through the browser. Here, we injected a simple HTML line break tag "`<br />`" to put each phrase onto its own line:

```
("Thank you, {0}, for your business. <br />Your Social Security number is: {1:000-00-0000}",
```

Thank you, Bob, for your business.  
Your Social Security number is: 123-32-1123

## Step 5: Breaking Up Code on Separate Lines for Readability

---

Now, let's apply the same principles towards formatting the phone number. Add a third placeholder, {2}, which will take the value supplied in yet another argument, phone. We will be applying this process over and over again, to handle the other user inputs, which will lead to a very long statement. To make it more readable, we can use the concatenation operator to break up the statement on separate lines:

```
int ss = int.Parse(ssTextBox.Text);
int phone = int.Parse(phoneTextBox.Text);

string result = string.Format("Thank you, {0}, for your business." +
    "<br />Your Social Security number is: {1:000-00-0000}" +
    "<br />Phone: {2: (000) 000-0000}",
    nameTextBox.Text,
    ss,
    phone);
```

The end result will look like this:

Thank you, Bob, for your business.  
Your Social Security number is: 123-32-1123  
Phone: (123) 123-1233



Bob's Tip: this goes back to the previous lesson on code style. This is very subjective - what looks good to one person, may not look good to another. Here's a good rule of thumb: keep lines approximately within an 80-column range. You can find the column number for each line at the bottom of Visual Studio:

Ln 22

Col 74

Ch 74

Also, when separating lines, try to keep a logical separator in mind. Here, the `<br />` tag is used as a line separator, as well as the comma after each string argument. This sets a pattern that make it easier for you to get a sense of what the code is doing, at a glance.

## Step 6: Applying Special Date Formatting

Next we will want to format the date returned by the applicant's Calendar selection. We already saw built-in formatting methods however sometimes that isn't flexible enough to handle a particular formatting required by the business need in question. Below is a short list of available date formatting options in C#:

Specifier	Description
<code>"d"</code>	The day of the month, from 1 through 31.
<code>"dd"</code>	The day of the month, from 01 through 31.
<code>"ddd"</code>	The abbreviated name of the day of the week.
<code>"dddd"</code>	The full name of the day of the week.
<code>"m"</code>	The minute, from 0 through 59.
<code>"mm"</code>	The minute, from 00 through 59.
<code>"M"</code>	The month, from 1 through 12.
<code>"MM"</code>	The month, from 01 through 12.
<code>"MMM"</code>	The abbreviated name of the month. .
<code>"MMMM"</code>	The full name of the month.
<code>"y"</code>	The year, from 0 to 99.
<code>"yy"</code>	The year, from 00 to 99.
<code>"yyy"</code>	The year, with a minimum of three digits.
<code>"yyyy"</code>	The year as a four-digit number.
<code>"yyyyy"</code>	The year as a five-digit number.

This is a partial list that can be found by visiting:

<http://is.gd/formattingdates>

Here we are applying some of these special date formatting rules to the fourth placeholder at {3}, which formats the date given by `I`oanDateCal endar. `SeI ectedDate`:

```
"<br />Phone: {2: (000) 000-0000}" +  
"<br />Loan Date: {3:ddd -- d, M, yy}",  
nameTextBox.Text,  
ss,  
phone,  
loanDateCalendar.SelectedDate);
```

This produces the date formatting shown here:

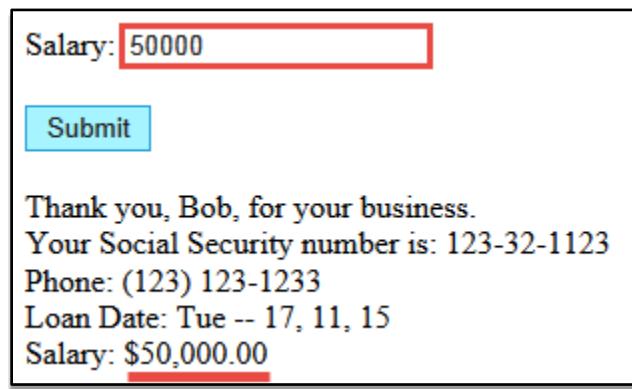
Name:	<input type="text" value="Bob"/>																																																				
Phone Number:	<input type="text" value="1231231233"/>																																																				
Social Security Number:	<input type="text" value="123321123"/>																																																				
Loan Origination Date:																																																					
<table border="1"><tr><td>&lt;</td><td>November 2015</td><td>&gt;</td></tr><tr><th>Sun</th><th>Mon</th><th>Tue</th><th>Wed</th><th>Thu</th><th>Fri</th><th>Sat</th></tr><tr><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr><tr><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td></tr><tr><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td></tr><tr><td>29</td><td>30</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>		<	November 2015	>	Sun	Mon	Tue	Wed	Thu	Fri	Sat	25	26	27	28	29	30	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5
<	November 2015	>																																																			
Sun	Mon	Tue	Wed	Thu	Fri	Sat																																															
25	26	27	28	29	30	31																																															
1	2	3	4	5	6	7																																															
8	9	10	11	12	13	14																																															
15	16	17	18	19	20	21																																															
22	23	24	25	26	27	28																																															
29	30	1	2	3	4	5																																															
Salary:	<input type="text"/>																																																				
<input type="button" value="Submit"/>																																																					
Thank you, Bob, for your business.																																																					
Your Social Security number is: 123-32-1123																																																					
Phone: (123) 123-1233																																																					
Loan Date: Tue -- 17, 11, 15																																																					

## Step 7: Applying Special Currency Formatting

---

When working with currency you can place a "C" after the colon, inside the placeholder, to tell the compiler that this is to be formatted as a currency. If you want to allow for cents, you need to first convert the entered value to a double, as well:

```
double salary = double.Parse(textBox1.Text);  
  
string result = string.Format("Thank you, {0}, for your business." +  
    "<br />Your Social Security number is: {1:000-00-0000}" +  
    "<br />Phone: {2: (000) 000-0000}" +  
    "<br />Loan Date: {3:ddd -- d, M, yy}" +  
    "<br />Salary: {4:C}",  
    nameTextBox.Text,  
    ss,  
    phone,  
    loanDateCalendar.SelectedDate,  
    salary);
```



The screenshot shows a Windows application interface. At the top, there is a form with a text input field labeled "Salary:" containing the value "50000". Below the input field is a blue "Submit" button. To the right of the form, a separate window displays the results of the submission. The results window contains the following text:  
Thank you, Bob, for your business.  
Your Social Security number is: 123-32-1123  
Phone: (123) 123-1233  
Loan Date: Tue -- 17, 11, 15  
Salary: \$50,000.00

For more information on currency formatting options, visit the following page showing you a variety of ways to represent exponential values, fixed point, and other numeric types of values:

<http://is.gd/formattingcurrency>

020

# Maintaining State with View State

In this lesson, we will cover an important ASP.NET concept, and that is managing state with ViewState. Consider how the Internet, or more specifically, the World Wide Web is intrinsically a *stateless* environment. This means that whenever you make a page request to a web server, it processes the request and sends the result as processed HTML. At which point, nothing is being held in memory on the server in question that points to your request instance, or the values that you've entered. Without using a framework, like ASP.NET or PHP, all of the information in the page and in its controls will be lost with each round-trip to the server.

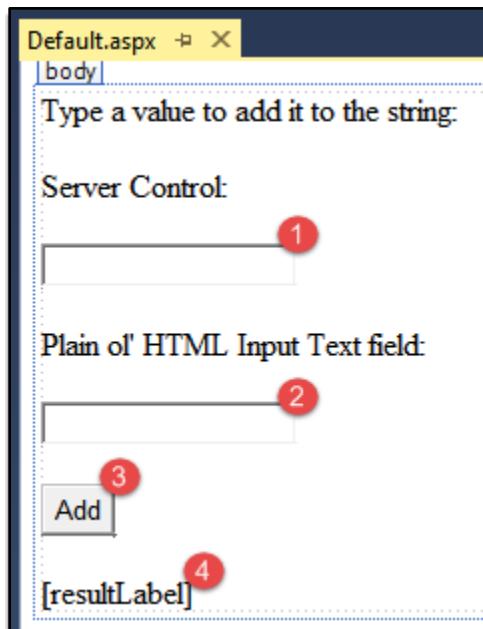
To overcome this intrinsic limitation of web development, the ASP.NET page framework includes several different ways to maintain state across multiple PostBacks to the web server. ViewState is one such way that allows you to preserve variable and Control values between round trips to the web server.

## Step 1: Create a New Project

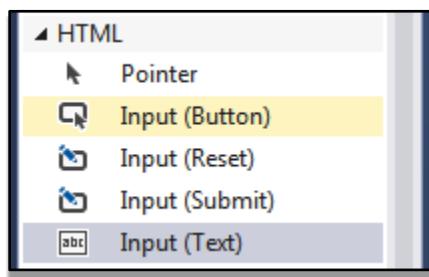
---

To illustrate this, create a new ASP.NET project called "CS-ASP\_020" and create the following Controls and programmatic IDs:

- (1) valueTextBox
- (2) Text1
- (3) addButton
- (4) resultLabel



Bear in mind that the user-input box for (2) is not a TextBox Server Control, but rather an HTML Input field found in the Toolbox:



## Step 2: HTML Fields Lose Information on PostBack by Default

---

Run the application and input some text for both input fields and then click "Add." You will notice that only the Server Control input field will retain the value entered, whereas the HTML input field does not. The reason for this is because the Server Control, being an ASP.NET construct, attempts to produce the appearance of a consistently held state. It's operating under the presumption that, despite the fact that HTML is stateless, many applications end up with some form of state-mimicking behavior (to make the user-experience more fluid, for example):

The screenshot shows a web page with two input fields and a button. The first input field is labeled "Server Control:" and contains the text "test". The second input field is labeled "Plain ol' HTML Input Text field:" and is empty. Below these fields is a blue "Add" button.

## Step 3: ViewState Holds Values in an Encoded HTML String

---

ASP.NET is able to retain this information, between server requests, by holding it in a hidden HTML form field encoded with base64 encryption. You can see this by right-clicking in a blank area of the web page in the browser and selecting the browser option to "View Source" code:

```
<form method="post" action="Default.aspx" id="form1">
<div class="aspNetHidden">
<input type="hidden"
      name="__VIEWSTATE"
      id="__VIEWSTATE"
      value="Ntw78W0+KbF/fvs+uIYRphrV/dh6WTF7rECM+kXYQzsYNazm2h1G3DrIsthSo=" />
</div>
```

Here we see this hidden information (hidden from view on the web page) that holds ViewState information, along with the encoded values. Notice how the encoded string looks like gibberish. That string holds all of the relevant values from the previous post (by clicking the "Add" button) to the server. And when the user posts *back* to the server, this information will be silently posted back as hidden form input, creating a bridge that carries important data between page refreshes. The more information needing to be stored by the encoded string - the longer the string grows. This presents its own set of problems, which has led many developers to pursue ASP.NET MVC (Model-View Controller), rather than ASP.NET Web Forms. The solution that MVC provides probably won't make much sense without first understanding the problems and solutions faced by using ViewState in ASP.NET Web Forms.



Bob's Tip: the concept of something holding a *state* is common throughout programming. In broader terms, holding state can be thought of as a particular configuration of any given object (in the real world, or in code) at any given moment. Consider how a Rubik's Cube changes state every time its puzzle pieces are shifted. You don't throw out one Rubik's Cube and get a new one whenever you want to change its state, however, this is kind of how ASP.NET/HTML works when it comes to creating the perception of holding state. The hidden information encoded in the HTML *post back* to the server is kind of like telling the server what the Rubik's Cube's previous state was and to create a new one that resembles that state.

## Step 4: Storing ViewState Values in a Key/Value Pair

---

Now, let's add values to the View State by writing the following code in *Default.aspx.cs*:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        ViewState.Add("MyValue", "Bob");
    }           1           2
}

protected void addButton_Click(object sender, EventArgs e)
{
    string value = ViewState["MyValue"].ToString();
    resultLabel.Text = value;   3
}
```

Here we are setting the `ViewState` property by accessing its `Add()` method. This method then takes in two strings, one representing the (1) “key” and the other representing the (2) value it refers to (this key/value pairing is called a “Dictionary” in programming terms). The `ViewState` property is being set on page load and is retrievable whenever you click on the “Add” button by displaying the value referenced when accessing the `ViewState`’s particular (3) key.



Bob's Tip: this Dictionary syntax might look a bit strange. Don't worry too much about how it works right now. The important lesson here is demonstrating how the `ViewState` holds information in between posts to the server.

Now, let's do something interesting:

- (1) Initialize, at page load, the `ViewState[MyValue]` key to hold an empty value.
- (2) Once the button is clicked, create a temporary variable called `value`, that stores whatever `ViewState[MyValue]` currently holds.
- (3) Take what's currently stored in `value` and add to it what the user provided via `valueTextBox`, also adding a space after it.

- (4) Replace whatever is currently in ViewState[MyValue] with whatever is assigned via value.  
 (5) After displaying through resultLabel , clear out what the user entered for valueTextBox.

```

protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        ViewState.Add("MyValue", ""); ①
    }
}

protected void addButton_Click(object sender, EventArgs e)
{
    ② string value = ViewState["MyValue"].ToString();
    ③ value += valueTextBox.Text + " ";
    ④ ViewState["MyValue"] = value;
    resultLabel.Text = value;

    ⑤ valueTextBox.Text = "";
}

```

Now, when running the application, enter into the Server Control TextBox the number 4 and click the button:

The screenshot shows a web page with two main sections. The top section is labeled "Server Control:" and contains a text input field with the value "4". The bottom section is labeled "Plain ol' HTML Input Text field:" and contains a blank text input field and a blue "Add" button.

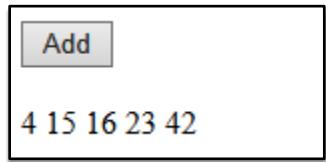
You will see the number displayed via the resultLabel :

The screenshot shows a single rectangular box containing the text "4", which is the result of the code execution.

Repeating this process, individually input each number in this sequence, clicking "Add" after each entry:

8, 15, 16, 23, 42

And you will see that the other entered values – previous to the last post back to the server - are retained in the ViewState[MyValue] property:

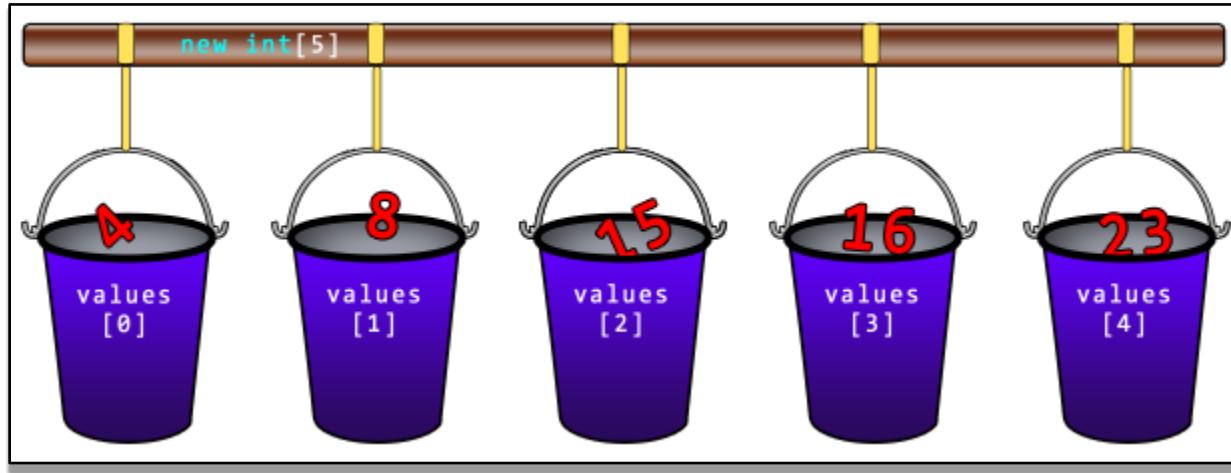


A screenshot of a Windows application window. It contains a list box with the following items: 4, 15, 16, 23, and 42. Above the list box is a small rectangular button labeled "Add". The entire window has a black border.

021

# Storing Values in Arrays

In this lesson, we're going to talk about storing values in arrays. The concept is not very difficult to understand and expands on the "bucket" analogy we used to describe variables in previous lessons. An array is essentially a collection of values, under a single variable name, that all have the same value types yet can hold distinct values. You can visualize an array as a row of buckets hanging from a broom-handle:



In this illustration, we "imprint" on the broom-handle the variable type that all the buckets must conform to, and initialize it with the number of distinct buckets we're allowed to store values in. We can then reference the individual buckets in our code by referring to the *index*. Arrays are indexed starting from 0 and can hold as many individual buckets as you need (as long as it's no greater than the amount specified in the original initialization). Here is how we would represent this particular array illustration:

```
int[] values = new int[5];
values[0] = 4;
values[1] = 8;
values[2] = 15;
values[3] = 16;
values[4] = 23;
```

Note that the array can hold only one data type (here, it is of type `int`), and the empty square brackets after the data type tells the compiler that this is, in fact, an array. The array must always be initialized with the "new" keyword, along with the data type and the size of the array (the amount of individual "buckets"). Once you have values in each "bucket" you can reference them just as you would with any

variable, except now you have to include the particular array element you want to use at any given moment:

```
resultLabel.Text = values[3].ToString(); //will display '3'
```

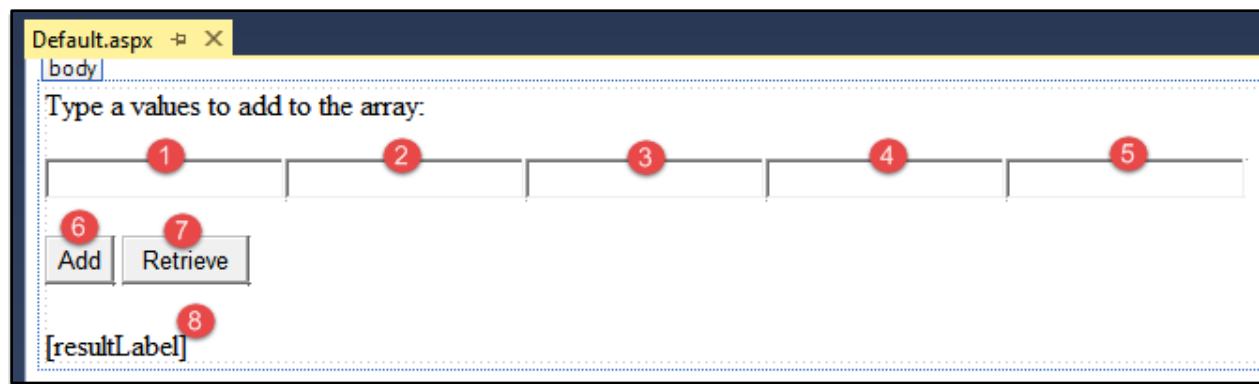


Bob's Tip: arrays are simple to create, however, they are extremely powerful. The simplest use of an array is to just group values that seem like they should belong together in some way. But the most common use for arrays is to *iterate* through them, performing some operation on each individual value. That leads us into *loops*, where arrays are often the star of the show. You'll see this in action in later lessons.

## Step 1: Create a New Project

Let's start by creating a new ASP.NET project called "CS-ASP\_021" and create the following Controls and programmatic IDs:

- (1) TextBox1
- (2) TextBox2
- (3) TextBox3
- (4) TextBox4
- (5) TextBox5
- (6) addButton
- (7) retrieveButton
- (8) resultLabel



Now, write this code in the addButton\_Click event to create the array and then display it upon clicking the button:

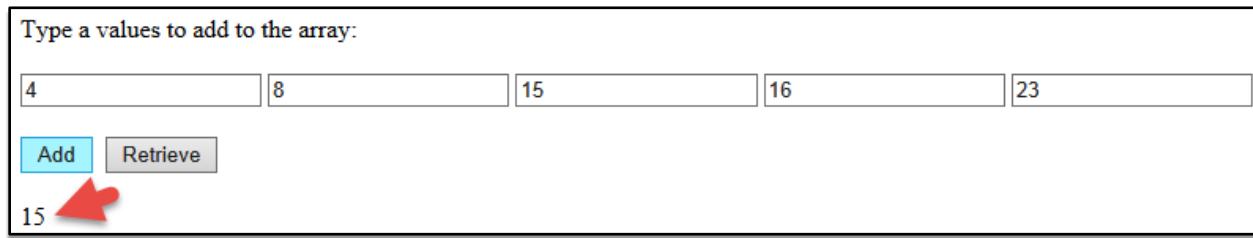
```
string[] values = new string[5];
values[0] = TextBox1.Text;
values[1] = TextBox2.Text;
values[2] = TextBox3.Text;
values[3] = TextBox4.Text;
values[4] = TextBox5.Text;

resultLabel.Text = values[2];
```

## Step 2: Arrays Indexes Start at [0]

---

Keeping in mind that the [2] index is the third value (since the index starts at [0]) we would expect the third TextBox entry to be output, via the resultLabel , after clicking the "Add" button:

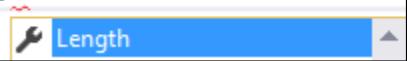


## Step 3: Using the Length Property to Count Array Size

---

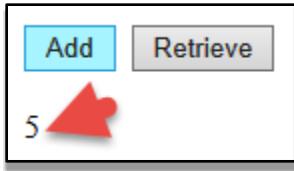
Go back to the addButton\_Click event and take a look at the variety of methods and properties available to the array. A common property is Length, which returns how many individual array elements there are, as an integer:

```
resultLabel.Text = values.
```



Let's reference this property in code, so that we can see it displayed when clicking the "Add" button. We know, ahead of time, that this particular array has five elements:

```
resultLabel.Text = values.Length.ToString();
```



You will see the use of the Length property in later lessons when iterating through arrays and needing to determine how many times the iteration has to execute (once for each array element, typically).

## Step 4: Simplifying Array Syntax with Initializers

---

When initializing an array with individual values for each element, there is an easier way than you've already seen. You can write the initial values, for each index, in curly braces right where the variable is first declared, and assigned:

```
string[] values = new string[5] { "Bob", "Steve", "Chuck", "Brian", "Andy" };
```

Now, combining this with what we learned about ViewState in the previous lesson, let's add this array to the ViewState Dictionary and then retrieve values in the array by clicking on the retrieveButton\_Click event:

```
protected void addButton_Click(object sender, EventArgs e)
{
    string[] values = new string[5] { "Bob", "Steve", "Chuck", "Brian", "Andy" };
    ViewState.Add("MyValues", values);
    resultLabel.Text = "Values added...";
}

protected void retrieveButton_Click(object sender, EventArgs e)
{
    string[] values = (string[])ViewState["MyValues"]; ←
    TextBox1.Text = values[0];
    TextBox2.Text = values[1];
    TextBox3.Text = values[2];
    TextBox4.Text = values[3];
    TextBox5.Text = values[4];

    resultLabel.Text = "Values retrieved...";
}
```

Pay particular attention to the line that assigns the value held in ViewState["MyValues"] to the string[] values variable. A cast, to string[], is needed here to make it work because the array actually gets stored in the dictionary as a generic object. Now, when you run the application add the names to the ViewState by clicking on the "Add" button:

Add    Retrieve

Values added...

And, then display the values back, from the ViewState:

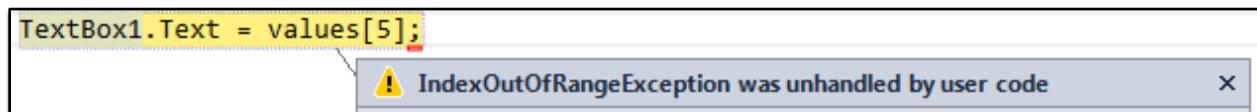
Bob    Steve    Chuck    Brian    Andy

Add    Retrieve

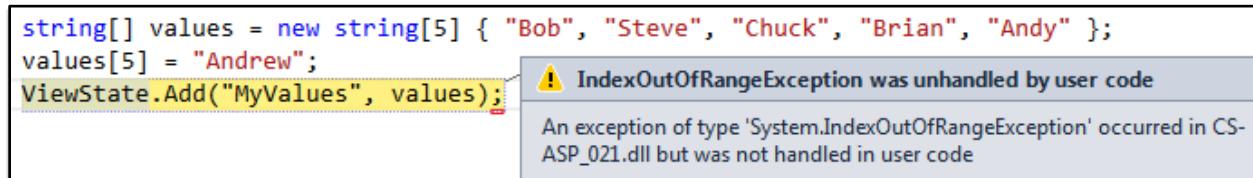
Values retrieved...

## Step 5: Referencing an Out-Of-Range Index in an Array

Perhaps the most common error programmers face when working with arrays is referencing an index that is out of range. In this case, we have an array with five indexes (numbered 0 to 4) so if we were to attempt to read from an index that does not exist we would get an IndexOutOfRangeException:



This error also commonly occurs when trying to write a value to an index range that doesn't exist. Here again, we only have five individual indexes numbered 0 to 4, yet we're trying to access a non-existent sixth index at values[5]:





Bob's Tip: you may find it troublesome trying to remember which individual array value belongs to which particular index. This is made a bit more difficult owing to the fact that indexes start at [0]. There is a good computational reason for this being the case. It may be annoying, but soon you will get used to remembering that the highest index for your array is *one less* than the actual number of elements in your array.

022

# Understanding Multidimensional Arrays

In this lesson, we're going to talk about multidimensional arrays. Multidimensional arrays are exactly like single-dimensional arrays, with the difference being that they require more than one index to get at a single element. Also, each index - except for the last one (the one that contains a value/object) - is itself an array.

## Step 1: Create a Two-Dimensional Array

---

The simplest multidimensional array would be a two-dimensional array, which means it would have two indexes:

```
string[,] MyArray = new string[3,3];
MyArray[0, 0] = "Bob";
MyArray[0, 1] = "Steven";
MyArray[0, 2] = "Jim";

MyArray[1, 0] = "Alice";
MyArray[1, 1] = "Jane";
MyArray[1, 2] = "Jill";

MyArray[2, 0] = "Tom";
MyArray[2, 1] = "Andrew";
MyArray[2, 2] = "Joe";
```

Here, MyArray is declared on the first line as being an array that holds three compartments, which in turn each holds another three compartments with the actual value inside of them.

Each group (arbitrarily separated with white space for clarity) can be seen as the first level compartment, while the next level is the group that stores the actual value. To access the values, you can reference them with the indexes they reside in:

```
resultLabel.Text = MyArray[2,1]; //displays "Andrew"
```

## Step 1: Visualizing a Two-Dimensional Array as a Data Table

---

You can also visualize this entire data/organization relationship as being like a common table structure:

MyArray[3,3]		
0	1	2
"Bob"	"Steven"	"Jim"
"Alice"	"Jane"	"Jill"
"Tom"	"Andrew"	"Joe"

This is similar to a multiplication table, where you get the value (in the grid) by matching up the row number with the column number. Unfortunately, this visual is limited when dealing larger multidimensional arrays, such as a three-dimensional array.

## Step 3: Create a Three-Dimensional Array

---

Here is an example of an array with three dimensions:

```
int[,,] Values = new int[2, 3, 3];
Values[0, 0, 0] = 4;
Values[0, 0, 1] = 23;
Values[0, 0, 2] = 17;

Values[0, 1, 0] = 68;
Values[0, 1, 1] = 32;
Values[0, 1, 2] = 7;

Values[0, 2, 0] = 41;
Values[0, 2, 1] = 13;
Values[0, 2, 2] = 73;

Values[1, 0, 0] = 55;
Values[1, 0, 1] = 6;
Values[1, 0, 2] = 24;

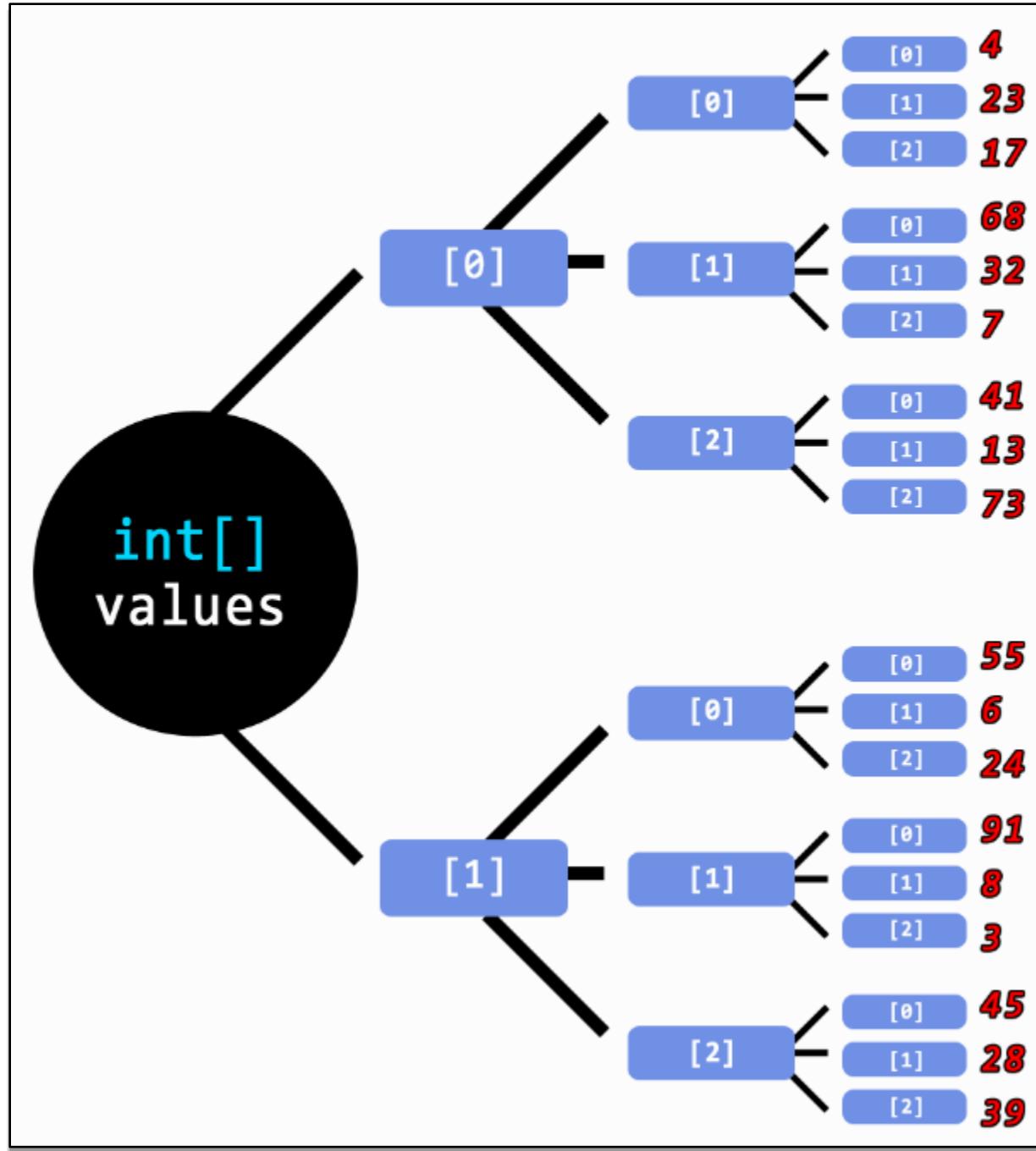
Values[1, 1, 0] = 91;
Values[1, 1, 1] = 8;
Values[1, 1, 2] = 3;

Values[1, 2, 0] = 45;
Values[1, 2, 1] = 28;
Values[1, 2, 2] = 39;
```

## Step 4: Visualizing Multi-Dimensional Arrays as Containers

---

This type of structure looks more complicated, but it only has one more level of sub-containment, and is best visualized as successive containers that contain other containers. Notice, again, how only the last level of containment holds an actual value while the other levels are only sub-containers:





Bob's Tip: while this containment structure may seem unintuitive, there are a plethora of analogous structures in the world we live in. Take, for example, a house. Each house represents the outer-most container, which contains individual rooms as sub-containers. And in each room there may be drawers, boxes, or cabinets that act as sub-sub-containers. Instead of using indexes - as we do in C# - we label these containers with names and effortlessly locate items of value with simple phrases like "In Joe's house, there is Amanda's room, and in that room is a cabinet that holds an Xbox controller."

## Step 4: A Real-World Containment Analogy

---

If you are having difficulty visualizing the containment structure for the array, temporarily replace each index with a familiar containment structure, such as the following:

```
int[,,] Values = new int[2, 3, 3];
Values[MasterBedroom, Dresser, Socks] = 4;
Values[MasterBedroom, Dresser, Shirts] = 23;
Values[MasterBedroom, Dresser, Ties] = 17;

Values[MasterBedroom, JewelryBox, Watches] = 68;
Values[MasterBedroom, JewelryBox, Rings] = 32;
Values[MasterBedroom, JewelryBox, Bracelets] = 7;

Values[MasterBedroom, NightStand, Books] = 41;
Values[MasterBedroom, NightStand, CDs] = 13;
Values[MasterBedroom, NightStand, Notebooks] = 73;

Values[Kitchen, Pantry, Soups] = 55;
Values[Kitchen, Pantry, Grains] = 6;
Values[Kitchen, Pantry, Cereals] = 24;

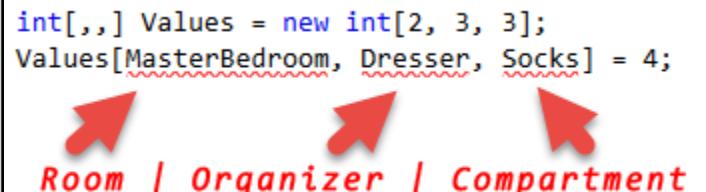
Values[Kitchen, Fridge, Meat] = 91;
Values[Kitchen, Fridge, Dairy] = 8;
Values[Kitchen, Fridge, Fruits] = 3;

Values[Kitchen, Cupboard, Glasses] = 45;
Values[Kitchen, Cupboard, Plates] = 28;
Values[Kitchen, Cupboard, Bowls] = 39;
```

Obviously, this will cause a compilation error (hence the red underlines), however, it should hopefully make the basic containment concept more intelligible. In this example there are:

- Two separate rooms  
`int[,,] Values = new int[2, 3, 3];`
- Three separate organizers in each room  
`int[,,] Values = new int[2, 3, 3];`
- Three separate compartments in each organizer  
`int[,,] Values = new int[2, 3, 3];`

And, of course, in each compartment is the actual value of relevance (in this analogy, the number of items in the compartment). In the illustration below, we're representing 4 items in the "Socks" compartment within the "Dresser" organizer, inside of the "MasterBedroom":



```
int[,,] Values = new int[2, 3, 3];
Values[MasterBedroom, Dresser, Socks] = 4;
```

Room | Organizer | Compartment

Also, notice how the number of dimensions is specified with the comma delimiter (empty for the type declaration, populated at initial assignment):

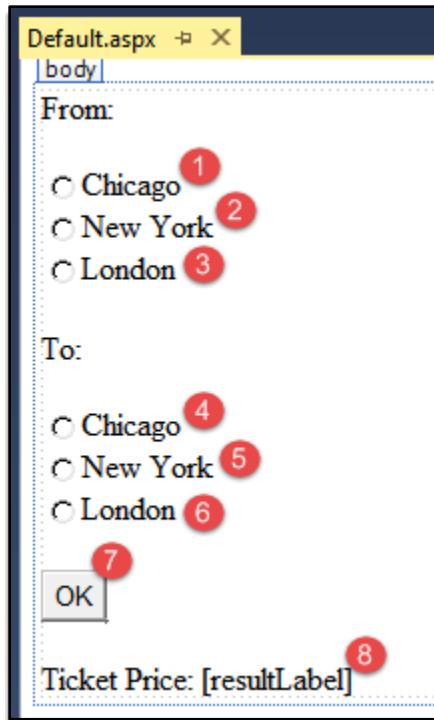
```
int[,,,] Values = new int[2, 3, 3];
```

## Step 5: Create a New Project

---

Now, let's put into action what we've learned about multi-dimensional arrays by creating an ASP.NET project called "CS-ASP\_022" and create the following Controls and programmatic IDs:

- (5) fromChicagoRadio
- (6) fromNewYorkRadio
- (7) fromLondonRadio
- (8) toChicagoRadio
- (9) toNewYorkRadio
- (10) toLondonRadio
- (11) okButton
- (12) resultLabel



What this application will do is let us hold various prices for travelling:

- From Chicago to New York
- From Chicago to London
- From New York to Chicago
- From New York to London
- From London to Chicago
- From London to New York

## Step 6: Storing Travel Prices in a Two-Dimensional Array

---

We can represent these various prices with a simple two-dimensional array, populating it on page load:

```

protected void Page_Load(object sender, EventArgs e)
{
    double[,] priceGrid = new double[3,3];
    //0 - Chicago
    //1 - New York
    //2 - London
    priceGrid[0, 1] = 350.00;
    priceGrid[0, 2] = 750.00;
    priceGrid[1, 0] = 400.00;
    priceGrid[1, 2] = 700.00;
    priceGrid[2, 0] = 800.00;
    priceGrid[2, 1] = 805.00;

}

```

To access the price of travelling from New York to London, we simply reference it with the relevant indexes that we populated with values:

```
resultLabel.Text = priceGrid[1, 2].ToString();
```



## Step 7: Broadening a Variable's Scope

---

Currently, `priceGrid` becomes populated in the `Page_Load` event, but what if we wanted to access these values via the `okButton_Click` event? You could, of course, put all of the code in the `okButton_Click` event, or you could even save it in the `ViewState`. However, supposing those solutions are not satisfactory, here is a much easier way of letting *both* events make reference to `priceGrid`. Simply move the declaration of `priceGrid` to the class level, as follows:

```

public partial class Default : System.Web.UI.Page
{
    double[,] priceGrid; ↑

    protected void Page_Load(object sender, EventArgs e)
    {
        priceGrid = new double[3,3];
↑        //0 - Chicago

```



Bob's Tip: this leads us towards the topic of code *scope*. This will be covered in ensuing lessons, however, for now just know that the squiggly brackets indicate a level of scope. By declaring a variable in an outer scope, it can be accessed by any inner scope. In this case, the outer scope is `public class Default`, while the inner scopes are the `Page_Load` and `okButton_Click` events, respectively.

Now, you can also access `priceGrid` from within `okButton_Click`, and set the values upon clicking the radio buttons:

```
protected void okButton_Click(object sender, EventArgs e)
{
    int fromCity;
    if (fromChicagoRadio.Checked) fromCity = 0;
    else if (fromNewYorkRadio.Checked) fromCity = 1;
    else fromCity = 2;

    int toCity;
    if (toChicagoRadio.Checked) toCity = 0;
    else if (toNewYorkRadio.Checked) toCity = 1;
    else toCity = 2;

    if (fromCity == toCity) return;

    resultLabel.Text = priceGrid[fromCity, toCity].ToString();

}
```

## Step 8: Adding Logic to Ignore the Same Source/Destination City

---

Note that the last conditional statement simply exits the entire code block (with the `return` keyword) before anything would display on the screen on the next line of code. This has the effect of doing nothing if selecting the same city for source/destination. When you run the application you will now be able to select various sources and destinations, and press "OK" to retrieve the applicable value:

<b>From:</b>
<input checked="" type="radio"/> Chicago <input type="radio"/> New York <input type="radio"/> London
<b>To:</b>
<input type="radio"/> Chicago <input checked="" type="radio"/> New York <input type="radio"/> London
<b>OK</b>
Ticket Price: 350

## Step 9: Clear Out the ResultLabel

---

You may have noticed that the resultLabel isn't cleared under the condition of having the same city for source and destination. That's because the method is exited (returned) out of before the resultLabel can be written to. And if a value in resultLabel was already present from a previous button submission this could cause confusion. Let's ensure that nothing is displayed in the resultLabel when the source/destination are the same:

```
if (fromCity == toCity)
{
    resultLabel.Text = "";
    return;
}
```

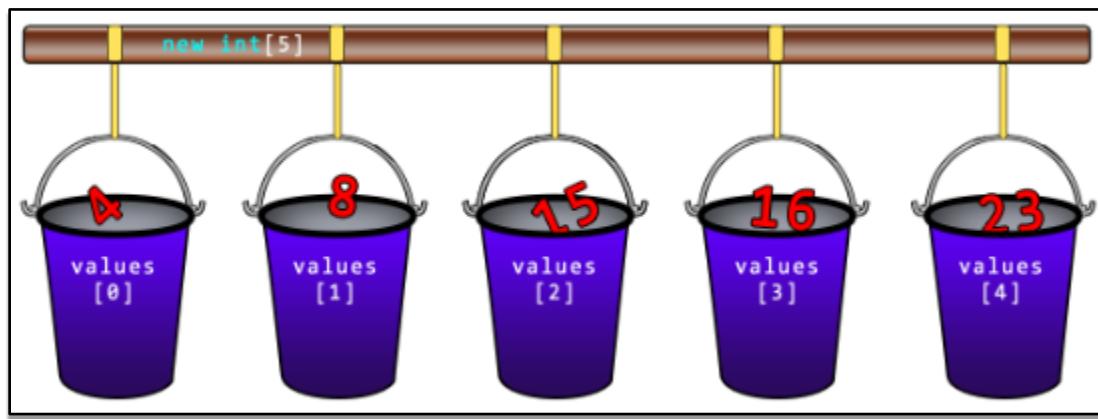


Bob's Tip: while arrays are quite powerful, you will probably not *need* to use multi-dimensional arrays very much in your programming career. They are very useful for holding particular data organization, whether that's something to do with multi-dimensional space, or complex database structures. However, there is another – arguably more useful - concept closely related to arrays and that is *Collections*. You will learn about Collections in later lessons, but basically they are built from arrays and are easier to handle due to being more flexible/less strict.

023

# Changing the Length of an Array

This lesson covers how to change the length of an array even though arrays are immutable by default. That means that once you define or declare an array, just like a variable, you normally cannot increase or reduce the number of array elements. Returning to a previous analogy, you cannot remove or add any more "buckets" strung up on the "broomstick"; it is of fixed size once that size is declared. We imprint on the broom-handle the size as precisely five elements:



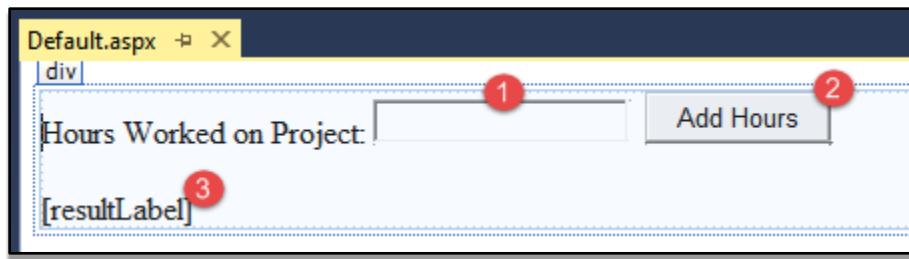
Fortunately, the .NET framework class library defines arrays with some helper methods that can process the array, and return one with a new size – returning an array that is either larger or smaller than the original array. You can also access other helper methods that get the sum, minimum, maximum, and average values within the array, along with other generally useful operations.

## Step 1: Create a New Project

---

Let's begin this lesson by creating a new ASP.NET project called "CS-ASP\_023" and create the following Controls and programmatic IDs:

- (1) hoursTextBox
- (2) addButton
- (3) resultLabel



## Step 2: Begin Coding a Simple Work Calculator

---

This application will be a simple calculator that allows us to add the number of work hours on a given project and return some statistics. Each work session will be stored in an array, so one session might be eight hours, another is four hours, and another is six hours. Those would be three different sessions. From that, we can access array helper methods to average them out and find the session with the most hours, the fewest hours, the average session length, and so on. Let's first create the array with zero elements (to start off with) upon PostBack, and add it to the ViewState:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        double[] hours = new double[0];
        ViewState.Add("Hours", hours);
    }
}
```

## Step 3: Array Resizing Magic with Array.Resize()

---

Then, in the addButton\_Click event create another locally scoped variable called hours and populate it with all of the previously added elements preserved in ViewState. Then resize the hours array (adding one more element) by passing it into the Resize() helper method found in the Array class. Note that the Resize() method takes in the current hours array, as well as its current length, plus one:

```
protected void addButton_Click(object sender, EventArgs e)
{
    double[] hours = (double[])ViewState["Hours"];

    Array.Resize(ref hours, hours.Length + 1);
}
```

The `ref` keyword here means “reference” and essentially takes the *actual* variable/object hours into the `Resize()` method rather than just copying it. This means you can perform an operation on the variable and not have to return a copy of it back to the original variable. There will be a lot more said about passing by reference/value in further lessons.

It's worth noting that the `Array.Resize()` method doesn't violate the fundamental rule of array immutability. What actually happens – behind the scenes – is the method makes a new array with one more element than the one we passed into it (in this case hours), and then *copies* all of the elements from the one we passed in, into the new array.



Bob's Tip: here's a concept that may help you wrap your head around how `Array.Resize()` works: if you're a Star Trek fan, you may be familiar with the concept of teleportation used throughout the series. With teleportation, no object actually *moves* from one place to another, but rather each particle (and its position in space) making up the source object gets *copied* (you can even imagine that the teleportation device uses three-dimensional arrays to represent each particle in three-dimensional space). A *new* object is simply built off of that copied information at the destination.

## Step 4: Find the Highest Index in the Array with `GetUpperBound()`

After that code, we will want to (1) return the highest numerical index in the hours array. And, since we know in advance that it will be the newest array element added in `Array.Resize()` – reference that index number (2) to store whatever the user entered in `hoursTextBox.Text`:

```
int newestItem = hours.GetUpperBound(0); 1  
hours[newestItem] = double.Parse(hoursTextBox.Text);  
2
```

And then simply replace the previous values in `ViewState["Hours"]` with the new hours array, and then output the sum of all array elements:

```
ViewState["Hours"] = hours;  
resultLabel.Text = String.Format("Total Hours: {0}", hours.Sum());
```

When you run the application, you will now be able to add hours and store them in `ViewState` in between PostBack's:

Hours Worked on Project:  Add Hours

Total Hours: 5

Hours Worked on Project:  Add Hours

Total Hours: 11

## Step 5: Some More Helper Methods Available to Arrays

---

We can also call several other helper methods available to the array:

```
resultLabel.Text = String.Format(  
    "Total hours: {0}<br />" +  
    "Most Hours: {1}<br />" +  
    "Least Hours: {2}<br />" +  
    "Average Hours: {3:N2}"  
    ,  
    hours.Sum(),  
    hours.Max(),  
    hours.Min(),  
    hours.Average()  
);
```



When you run the application, you will now see these various statistics. Here, we entered three values (6, 7, and 9) to represent the hours worked in three different sessions:

Hours Worked on Project:  Add Hours

Total hours: 22

Most Hours: 9

Least Hours: 6

Average Hours: 7.33

024

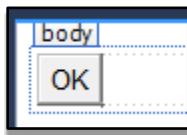
# Understanding Variable Scope

It's about time that we look more closely at the topic of variable scope. Variable scope can be summarized as follows: variables that are initially declared in any given code block (designated by squiggly brackets) are available for reference within that code block, as well as code blocks inside of that block. However, the opposite is not true: variables declared within an inner code block are not available to outer code blocks. In other words, a code block can *reach out* and grab an outer code block's variable, but it can't *reach in* and grab an inner code block's variable.

## Step 1: Create a New Project

---

Let's set up a simple demonstration of this by creating a new ASP.NET project called "CS-ASP\_024" with a Button Control that has a programmatic ID called "okButton":



Whenever you (1) first declare a variable within a given code block it is said to *belong to that scope*. Here, x is scoped local to the okButton\_Click event and is, therefore, available (2) anywhere within the okButton\_Click event, including (3) inner code blocks:

```
protected void okButton_Click(object sender, EventArgs e)
{
    int x = 1; 1

    if (x == 1)
    {
        x = 2;
    } 3
2    string result = x.ToString();
}
```

As mentioned, however, you cannot do the reverse and reach into inner scopes to access inner-scoped variables:

```
protected void okButton_Click(object sender, EventArgs e)
{
    int x = 1;

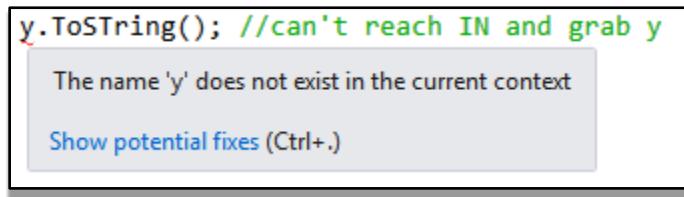
    if (x == 1)
    {
        x = 2;
        int y = 3; ←
    }

    string result = x.ToString();
    string y.ToString(); //can't reach IN and grab y
}
```

## Step 2: Refer to Intellisense to See What's in Scope

---

In fact, you will notice that Intellisense can't even recognize `y` outside of the scope that it was originally declared. If you insist on trying to access it anyways, Intellisense will provide you with this error message when hovering over the inaccessible variable (you can exchange the word "context" for "scope"):



## Step 3: Class Scoped Variable Availability within Method Scope

---

When you declare the variable at the class level (scope), it becomes accessible to all of the inner scopes (including inner-inner scopes, inner-inner-inner scopes, and so on). Again, this is perfectly acceptable because inner scopes can reach out and grab any variable from any scope outer to it:

```
public partial class Default : System.Web.UI.Page
{
    string z = "";

    protected void Page_Load(object sender, EventArgs e)
    {
        z += "Hello ";
    }

    protected void okButton_Click(object sender, EventArgs e)
    {
        z += "Bob ";

        if (true)
        {
            z += "Tabor";
        }
    }
}
```

It may be better to visualize the various scope levels with the following representation:

```
public partial class OuterScope
{
    bool Accessible;

    protected void InnerScope1(object sender, EventArgs e)
    {
        Accessible = true; //Accessible to 1st Inner Scope
    }

    protected void InnerScope2(object sender, EventArgs e)
    {
        Accessible = true; //Accessible to 2nd Inner Scope

        if (Accessible)
        {
            Accessible = true; //Accessible to Inner-Inner Scope
        }
    }
}
```



Bob's Tip: scope becomes a bit more interesting, and complex, when you begin to learn about Object-Oriented Programming. However, for now the outer-most scope you will be concerned with is that of the *class* level.

025

# Code Blocks and Nested Statements

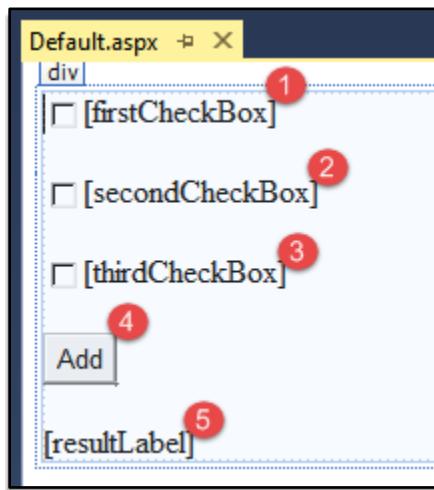
This lesson will look closer at the related topics of code blocks and nested statements.

## Step 1: Create a New Project

---

Let's start by creating a new ASP.NET project called "CS-ASP\_025" with the following Controls and programmatic IDs:

- (1) firstCheckBox
- (2) secondCheckBox
- (3) thirdCheckBox
- (4) addButton
- (5) resultLabel



In the *Default.aspx.cs* file, write the following code:

```
1  namespace CS_025
2  {
3      public partial class Default : System.Web.UI.Page
4      {
5          protected void Page_Load(object sender, EventArgs e)
6          {
7              if (firstCheckBox.Checked)
8              {
9                  if (secondCheckBox.Checked)
10                 {
11                     if (thirdCheckBox.Checked)
12                     {
13                         resultLabel.Text = "They're all checked!";
14                     }
15                 }
16             }
17         }
18     }
19 }
```

Here we see seven separate code blocks. The outer-most code block (1) is the namespace, which immediately holds (2) a single block of code – a class – which is called Default. The class level then immediately holds two method code blocks named (3) Page\_Load and (4) addButton\_Click. Finally, addButton\_Click code block then holds a series of *nested if()* statements (5), (6) and (7). Notice, also, how you can expand and collapse the individual blocks/nested statements by clicking on the boxes with dashes in them (highlighted in green).



Bob's Tip: up to this point, we have been referring to the named code blocks that determine behavior for Server Controls as *events*. However, technically these code blocks are *methods*, and methods can *subscribe* to special kinds of properties called events. All of the methods we have been writing code into, thus far, happen to subscribe to events. This will be clarified in further lessons covering methods and events.

## Step 2: Avoiding Excessive Nesting with the Return Keyword

---

Nested if() statements are somewhat different from the containment relationships seen in the other code blocks. They are simply used to perform logical branching in your code – as seen in previous lessons – and should only be nested no more than a few levels deep. The reason for this is that it can make code look very messy and difficult to read, and there are often ways to solve the problem with other techniques. For instance, here is another way of determining whether or not all of the CheckBoxes are checked, and displaying the resultLabel only when they are:

```
protected void addButton_Click(object sender, EventArgs e)
{
    if (!firstCheckBox.Checked) return;
    if (!secondCheckBox.Checked) return;
    if (!thirdCheckBox.Checked) return;
    resultLabel.Text = "They're all checked!";
}
```

This solves the problem in a much cleaner way since the return keyword effectively exists out of further execution for this method when any of the CheckBoxes are not checked. This provides the same functionality as the nested if() statements displaying the resultLabel message only when all CheckBoxes are checked.

## Step 3: Avoiding Nesting with Compound Expressions

---

Another, simpler, way of solving this problem is to make a compound expression within a single if() statement, like this:

```
if (firstCheckBox.Checked && secondCheckBox.Checked && thirdCheckBox.Checked)
    resultLabel.Text = "They're all Checked!";
```



Bob's Tip: although this compound expression within a single if() statement is logically equivalent to the nested if() statements, there are some cases where you can't simplify in such a way. For example, if you needed different code to execute for each condition (or, each CheckBox in this example), you would definitely need those contingencies to exist on their own “branch” of code.

In upcoming lessons we will look at looping conditional statements that can often take on deep nesting structures. However even in this case there are ways to reduce the visual impact of deep nesting by breaking up nested code and putting it into helper methods, for example.

026

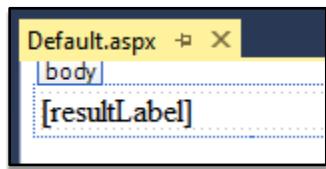
# Looping with the for() Iteration Statement

In this lesson, we're going to look at how to loop through a block of code using the `for()` iteration statement. A looping statement is very similar to other conditional statements (such as the `if()` statement), except at the end of its code block it *loops back* to the top where the initial condition is evaluated, and continues to do so until the condition evaluates as false.

## Step 1: Create a New Project

---

For this lesson, you should create an ASP.NET project called "CS-ASP\_026" and create a single `resultLabel` Control:



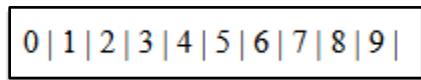
And in the `Default.aspx.cs` file write the following in the `Page_Load()` method:

```
protected void Page_Load(object sender, EventArgs e)
{
    string result = "";

    for (int i = 0; i < 10; i++)
    {
        result += i.ToString() + " | ";
    }

    resultLabel.Text = result;
}
```

And then run the application to see the result:



## Step 2: Breaking Down the for() Iteration Statement

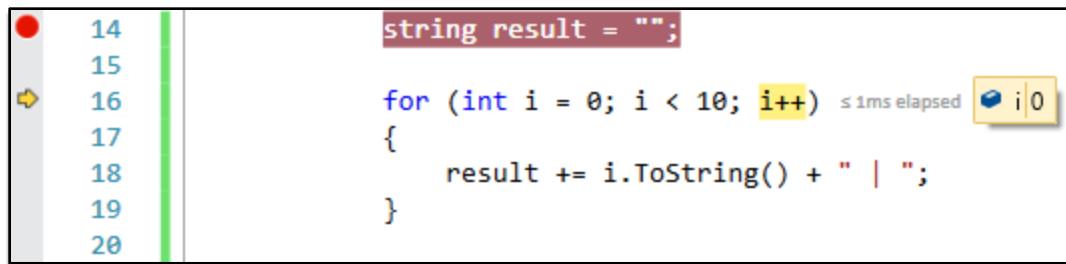
---

Let's break down what's happening in the for() iteration statement here:

- (1) We initialize, to zero, an int variable that is local to the for loop, called i .
- (2) We create a conditional evaluation at the start of every loop (here it checks if i is less than 10).
- (3) If the evaluation in (2) returns true, increment i by one, else end the loop.
- (4) For each time the conditional evaluates true (10 times) execute this line of code.

```
1   2   3  
for (int i = 0; i < 10; i++)  
{  
    4 result += i.ToString() + " | ";  
}
```

Perhaps the best way to illustrate how this loop executes is to set a break-point and watch it progress – line by line – in debug mode. Remember to press the F10 key to step through each part of code as it executes. Also, it would be worthwhile to pin the i variable in order to see exactly where and when it increments:



```
14 string result = "";  
15  
16 for (int i = 0; i < 10; i++) s 1ms elapsed i|0  
17 {  
18     result += i.ToString() + " | ";  
19 }  
20
```



Bob's Tip: naming the counter variable's as "i " is quite common throughout computer programming. It can stand for "iterator" or, when used as the index for an array could stand for "indexer." You might also see code that uses "c", as in "counter" or any other single-letter character. This is a coding convention that is supposed to improve readability, but you are free to use whatever convention makes most sense to you.

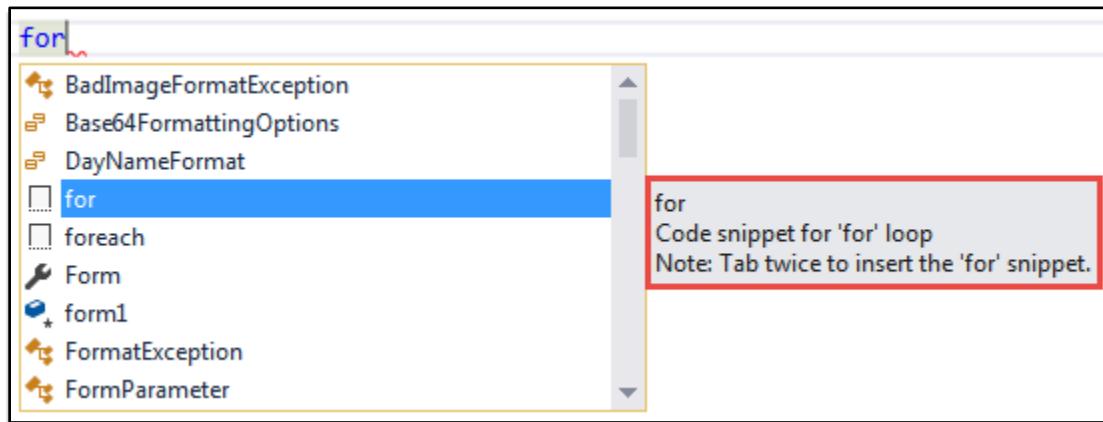
## Step 3: Code Snippets for Recalling Syntax

---

It may be a bit difficult, at first, for beginners to remember the exact syntax for constructing `for()` loops. Luckily, Visual Studio has a short-cut for creating a `for()` loop template by using the following sequence: type in the word “`for`” and on the keyboard hit tab twice:

```
for (int i = 0; i < length; i++)
{
}
```

This kind of preset template is called a “code snippet” and you can even find it referenced in Intellisense when typing in the `for` keyword:



Another shortcut you can exploit by using this code snippet is renaming the iterator variable, which then automatically renames it for the rest of the `for()` statement:

```
for (int index = 0; index < length; index++)
{
}
```



Bob's Tip: there are a lot of useful code snippets that you will encounter as we move forward. Eventually, you can even learn how to create your own custom code snippets!

## Step 4: Setting Up the Iterator

---

It's worth noting that you can, of course, (1) start with any value you want for the iterator, (2) perform any legal conditional evaluation on the iterator, and (3) perform any calculation on the iterator, as well:

```
1   2   3  
for (int i = 5; i <= 25; i+=5)  
{  
    result += i.ToString() + " | ";  
}  
  
5 | 10 | 15 | 20 | 25 |
```

---

## Step 5: Using for() Loops with Arrays

---

One of the greatest uses of looping statements is to use the iterator as an index for an array (you will often hear this stated as “looping through an array”):

```
string[] names = new string[] { "Wolverine", "Cyclops", "Professor X", "Phoenix" };  
  
for (int i = 0; i < names.Length; i++)  
{  
    result += names[i] + " | ";  
}
```

```
Wolverine | Cyclops | Professor X | Phoenix |
```

You will begin to see the power of arrays and loops when combined with all of the different ways of manipulating and sorting arrays:

```
string[] names = new string[] { "Wolverine", "Cyclops", "Professor X", "Phoenix" };  
  
Array.Sort(names); //re-indexes the array, sorting alphabetically  
Array.Reverse(names); //re-indexes the array, sorting in reverse  
  
for (int i = 0; i < names.Length; i++)  
{  
    result += names[i] + " | ";  
}
```

## Step 6: Breaking Out of the Loop

A common usage for looping through an array would be to go through a database or list of items, searching for a particular item you had in mind. You might imagine that if you have a very large array of items, you would not want the loop to continue after it finds the item you are looking for. By default, that's exactly what would happen, however, you can prematurely *break-out* of the loop by using the `break` keyword:

```
for (int i = 0; i < names.Length; i++)
{
    if (names[i] == "Phoenix") ←
    {
        result = String.Format("{0} is at index {1} in the list.", names[i], i);
        break; ←
    }
}
```

Phoenix is at index 2 in the list.

To show the iteration process, you can set a break-point and watch it in Debug mode, or you can add this `else()` clause. Notice how even though there are four items in the list the `for()` loop stops executing after it found the item we are looking for at the third position in the list:

```
for (int i = 0; i < names.Length; i++)
{
    if (names[i] == "Phoenix")
    {
        result += String.Format("{0} is at index {1} in the list.", names[i], i);
        break;
    }
}
else
{
    result += String.Format("{0} is not found at index {1}.<br />", "Phoenix", i);
}
```

Phoenix is not found at index 0.  
Phoenix is not found at index 1.  
Phoenix is at index 2 in the list.

027

## Looping with while() & do...while() Iteration Statements

In this lesson, you will learn about the `while()` and `do...while()` iteration statements. These statements operate in much the same way that `for()` statements operate. The main difference is that a `for()` statement is best used when there is a counter required for a particular number of iterations, whereas `while()` and `do...while()` are best used when a single expression needs to be evaluated and continuing the loop so long as it evaluates as `true`.

### Step 1: Understanding Syntax for while() and do...while()

---

Here is the basic syntax for constructing a `while()` loop (you can insert any expression that evaluates to a boolean value):

```
while (true)
{
    //keep executing this while
    //the expression above evaluates as true
}
```

Meanwhile, a `do...while()` loop operates in the same way, except that it's inverted. In other words, it first executes the code block (at least once), and continues to execute so long as the evaluation holds `true`:

```
do
{
    //do this at least ONCE
    //keep doing it as long as the
    //expression below evaluates as true
}
while(true);
```

It will not be immediately apparent why `do...while()` is ever needed. Although it is not as commonly used as the `while()` loop, it is very useful when a block of code has to execute *at least once*, but possibly several more times.

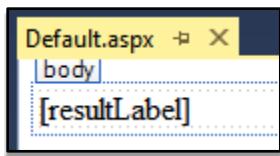


Bob's Tip: as with the `for()` loop code snippet, you can quickly set up a `while()`, or `do...while()` template. Just type in "while" or "do", respectively, and then hit the tab key twice.

## Step 2: Create a New Project for a Hero vs Monster Game

---

Let's begin the lesson by setting up an ASP.NET project called "CS-ASP\_026" with a single `resultLabel` Control:



In `Default.aspx.cs` write some preliminary code for the `Page_Load()` method that sets the scene for a text-based adventure "game" between a hero and a monster:

```
protected void Page_Load(object sender, EventArgs e)
{
    Random random = new Random();

    int heroHealth = 30;
    int monsterHealth = 30;

    string result = "";

    // Hero gets bonus first attack
    monsterHealth -= random.Next(1, 100);

    int round = 0;
    result += "<br />Round: " + round;
    result += String.Format("<br />Hero attacks first, leaving monster with {0} health.", 
        monsterHealth);

    // Need battle logic here!
```

```
// Need battle logic here!

if (heroHealth > 0)
{
    result += "<br />Hero wins!";
}
else
{
    result += "<br /> Monster wins!";
}

resultLabel.Text = result;
}
```

## Step 3: Simulating Game Battle Logic

---

Now, let's write our "battle" logic that allows each side to duke it out, repeatedly, until one of the opponents reaches zero health. Here, each opponent deals a random amount of damage (between 1 and 10 for the hero, and between 1 and 20 for the monster):

```
// Need battle logic here!

while (heroHealth > 0 && monsterHealth > 0)
{
    int heroDamage = random.Next(1, 10);
    int monsterDamage = random.Next(1, 20);

    monsterHealth -= heroDamage;
    heroHealth -= monsterDamage;
}
```

Now, you will want to report the actual damage done for each round (try both, `++round` and `round++` to see the difference between the two):

```
monsterHealth -= heroDamage;
heroHealth -= monsterDamage;

result += "<br />Round: " + ++round; //output Round number

result += String.Format //output Hero damage
(
    "<br />Hero causes {0} damage, leaving monster with {1} health.",
    heroDamage, monsterHealth
);

result += String.Format //output Monster damage
(
    "<br />Monster causes {0} damage, leaving hero with {1} health.",
    monsterDamage, heroHealth
);
```



Bob's Tip: the `String.Format()` method might look a bit strange here but remember you can use whatever whitespace you want in order to make the code more readable.

When you run the application, you will see the result of this epic battle. However, you will probably notice that most of the time the hero gets in a single shot – from the initial “bonus” attack - and wins without triggering the main battle logic in the `while()` loop. This is because the monster’s battle logic is included in the `while()` loop, and because it executes only when both combatants have more than zero health, the monster will often be left unable to retaliate. What we will want to do instead is have both combatants engage each other *at least once* so that the monster gets at least a single shot in. All we need to implement this is simply invert the `while()` loop so that its code block executes first, creating a `do...while()` loop, instead:

```

do
{
    int heroDamage = random.Next(1, 10);
    int monsterDamage = random.Next(1, 20);

    monsterHealth -= heroDamage;
    heroHealth -= monsterDamage;

    result += "<br />Round: " + ++round; //output Round number

    result += String.Format //output Hero damage
    (
        "<br />Hero causes {0} damage, leaving monster with {1} health.",
        heroDamage, monsterHealth
    );

    result += String.Format //output Monster damage
    (
        "<br />Monster causes {0} damage, leaving hero with {1} health.",
        monsterDamage, heroHealth
    );
}
while (heroHealth > 0 && monsterHealth > 0);

```

Now, we see that the monster gets in at least one attack every time, even if he's at below zero health:

```

Round: 0
Hero attacks first, leaving monster with -25 health.
Round: 1
Hero causes 4 damage, leaving monster with -29 health.
Monster causes 4 damage, leaving hero with 26 health. ←
Hero wins!

```



Bob's Tip: as usual, you can learn a lot about the flow of execution in your code by setting a break-point and watching the magic happen in Debug mode. Try doing this for both while() and do...while() loops to see the difference between how they execute.

028

# Creating and Calling Simple Helper Methods

Up to this point, we have mainly been calling or utilizing built-in helper methods available to us via the .NET framework. However, now you will learn how to define, and call, simple helper methods of your own making. To put it simply, a method is a named block of code that executes, in full, whenever you reference it elsewhere by calling it by name followed by the invocation parentheses. For example, `string.Format()`, `int.Parse()`, and `random.Next()` are just a few of the built-in methods we have already worked with thus far.

## Step 1: Creating Helper Methods for Abstraction

---

There's a few reasons for why you would ever create your own helper method. One reason is simply code reuse; wrapping up a common code block into a named method whenever you see it required in two or more places. Another reason is to *abstract* away complicated code, breaking it up into methods that commit a certain process without requiring you to always to look at the details of that process.



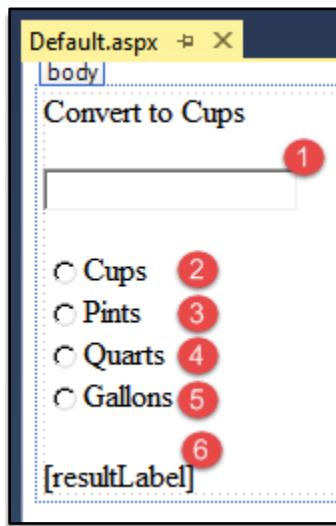
Bob's Tip: there's an old saying that your code should "read like a story." In other words, if you have a complex problem that is housed within a single method, it becomes monolithic and difficult to read. However, if you split it up into smaller code blocks - and give each of those code blocks a descriptive name of what that code block is attempting to do - then you can read it out, line-by-line, focusing on what is *generally* happening, rather than every banal detail of how it's happening.

## Step 2: Create a New Project

---

It's often typical for programmers to first tackle a problem by writing all of the code in a single method, not worrying about breaking it up into individual methods right away. Breaking up monolithic code, after-the-fact, is a technique called "refactoring". You'll be seeing this in action shortly, but first, start this lesson by creating a new ASP.NET project called "CS-ASP\_028" with the following Server Controls and Programmatic IDs:

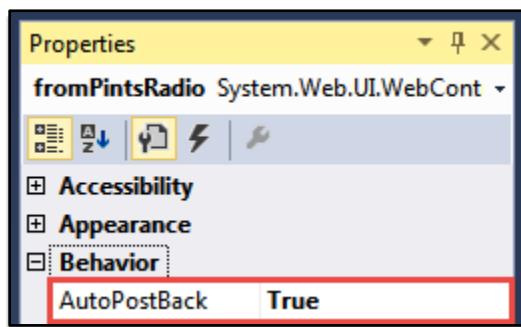
- (1) quantityTextBox
- (2) fromCupsRadio
- (3) fromPintsRadio
- (4) fromQuartsRadio
- (5) fromGallonsRadio
- (6) resultLabel



## Step 3: Enable the AutoPostBack Property

---

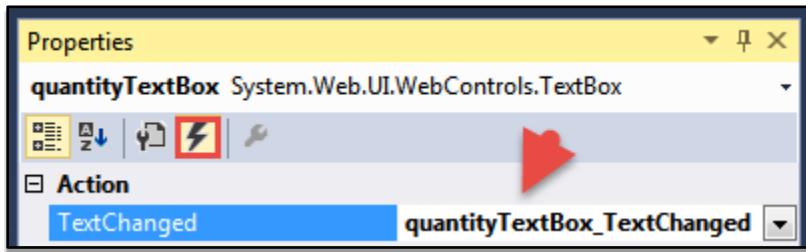
This is going to be a simple application that converts between various measurement formats and displays the results in real-time. Notice that there is no button to PostBack to the server, and that's because the RadioButtons and TextBoxes should have a property called "AutoPostBack." Be sure to change this property for each of the Controls mentioned. This will effectively PostBack to the server when each element is changed:



## Step 4: Setting the TextChanged Action Event

---

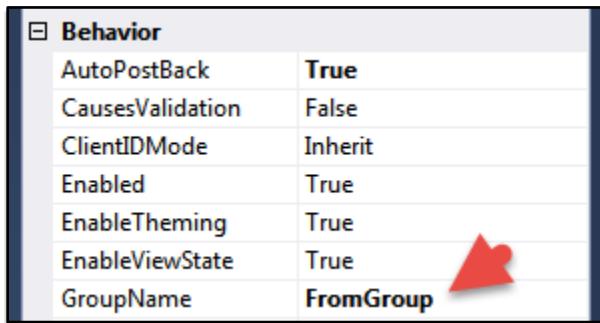
The event that fires upon PostBack will be the TextChanged Action event:



Make sure to set each Server Control, except resultLabel, with an Action event named accordingly:

- (1) quantityTextBox\_TextChanged
- (2) cupsRadio\_CheckedChanged
- (3) fromPintsRadio\_CheckedChanged
- (4) fromQuartsRadio\_CheckedChanged
- (5) fromGallonsRadio\_CheckedChanged

Also, each RadioButton should be set to the same GroupName in the Properties Window:



## Step 5: Prelude to Refactoring

---

Now, what you could do is write out each event/method with minor differences depending on the measurement it's dealing with. This will create a very obvious pattern, which should be a tip that a helper method might be useful to avoid so much duplicate code:

```

protected void cupsRadio_CheckedChanged(object sender, EventArgs e)
{
    if (quantityTextBox.Text.Trim().Length == 0)
        return;

    double quantity = 0.0;
    if (!Double.TryParse(quantityTextBox.Text, out quantity))
        return;
    resultLabel.Text = "The number of cups: " + quantity.ToString();
}

protected void fromPintsRadio_CheckedChanged(object sender, EventArgs e)
{
    if (quantityTextBox.Text.Trim().Length == 0)
        return;

    double quantity = 0.0;
    if (!Double.TryParse(quantityTextBox.Text, out quantity))
        return;

    double cups = quantity * 2.0;
    resultLabel.Text = "The number of cups: " + cups.ToString();
}

protected void fromQuartsRadio_CheckedChanged(object sender, EventArgs e)
{
    if (quantityTextBox.Text.Trim().Length == 0)
        return;

    double quantity = 0.0;
    if (!Double.TryParse(quantityTextBox.Text, out quantity))
        return;

    double cups = quantity * 4.0;
    resultLabel.Text = "The number of cups: " + cups.ToString();
}

```

## Step 6: Understanding TryParse()

---

But before we refactor this, let's briefly mention some unfamiliar code elements seen in each of these methods. Each method has some *validation* code to make sure acceptable inputs have been provided by the user. The first `if()` statement checks to see if `quantityTextBox.Text` is empty - after it's been trimmed of whitespace - and exits the method if that is the case. The next `if()` statement simply checks to see if the value entered is non-numeric and also exits if that is the case.

There are extra elements in this line of code that you will become familiar with shortly. However, for the time being just know that the `Double.TryParse()` method returns true if it can successfully convert the input to a double, and it also returns the second argument – quantity – converted to a double via the `out` keyword. And finally, provided that the user-input values pass validation of the previous two `if()` statements, we perform the calculation and display it through the `resultLabel`. Now, to consolidate all of this duplicate code, let's simply take the code common to each of these methods and put it all in its own custom helper method called `calculateCups()`:

```
private void calculateCups()
{
    if (quantityTextBox.Text.Trim().Length == 0)
        return;

    double quantity = 0.0;
    if (!Double.TryParse(quantityTextBox.Text, out quantity))
        return;
}
```

*<NOTE: the learner hasn't seen the `private` keyword yet. It is out of context, however, so even mentioning it is probably more confusing than it's worth at this point>*

The code that's different for each method is the part that performs particular calculations for each conversion. However, that can also be put into this helper method via a series of `if()` statements underneath the previous validation code:

```
double cups = 0.0;

if (fromCupsRadio.Checked) cups = quantity;
else if (fromPintsRadio.Checked) cups = quantity * 2;
else if (fromQuartsRadio.Checked) cups = quantity * 4;
else if (fromGallonsRadio.Checked) cups = quantity * 16;

resultLabel.Text = "The number of cups: " + cups.ToString();
```

Now, you can erase all of the previous code in the methods/events, and instead just reference the helper method, by name, in each of those methods/events:



Bob's Tip: whenever you reference a method by name followed by the invocation parentheses, it is said that you are *calling* the method. You are essentially "getting on the phone" and calling the method, asking it to then execute its code block each time you place a call to it.

```
protected void cupsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups();
}

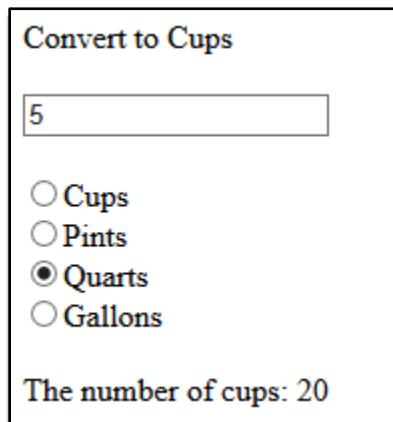
protected void fromPintsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups();
}

protected void fromQuartsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups();
}

protected void fromGallonsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups();
}

protected void quantityTextBox_TextChanged(object sender, EventArgs e)
{
    calculateCups();
}
```

When you run the application, it will work as it did before:



## Step 7: Understanding a Method's Invocation Parentheses

---

Be sure to include the invocation parentheses when calling a method that you want to execute, else you will get an error:

```
calculateCups;
```



Bob's Tip: another common principle in coding is called the “DRY” principle. It stands for “don’t repeat yourself.” Repetition becomes a big problem in your code as the code-base grows larger and requires changes within it. You might imagine a scenario in which end-users are displeased upon you fixing a bug in one part of your application, while the bug persists in another part of your application where the repeated code (along with the repeated bug) may be missed.

029

# Creating Methods with Input Parameters

This lesson will look at how to create a method with input parameters. To illustrate this, let's take a look at the calculateCups() helper method that we created in the previous lesson:

```
private void calculateCups()
{
    if (quantityTextBox.Text.Trim().Length == 0)
        return;

    double quantity = 0.0;
    if (!Double.TryParse(quantityTextBox.Text, out quantity))
        return;

    double cups = 0.0;

    if (fromCupsRadio.Checked) cups = quantity;
    else if (fromPintsRadio.Checked) cups = quantity * 2;
    else if (fromQuartsRadio.Checked) cups = quantity * 4;
    else if (fromGallonsRadio.Checked) cups = quantity * 16;

    resultLabel.Text = "The number of cups: " + cups.ToString();
}
```

This particular solution works however it isn't intuitively scalable if we ever want to add a new calculation – something greater than cups-to-gallons for example – as it would require a modification within the *method internals* (the conditional `if()`...`else if()` statements, in this case). It would be much easier if we could just pass into each call to calculateCups() the particular measurement ratio required for that calculation, and eliminating the need for the conditional block entirely.

## Step 1: Add an Input Parameter to calculateCups()

---

This precise functionality is possible by changing the calculateCups() method so that it allows for an input parameter to be passed into the method:

```
private void calculateCups(double measureToCupRatio)
{
```

This is essentially defining a variable that we can reference throughout the remaining body of this helper method, with the actual value being supplied at the method call. Now we can remove the conditional block and, in its place, incorporate the measureToCupRatio parameter in the following manner:

```
double quantity = 0.0;
if (!Double.TryParse(quantityTextBox.Text, out quantity))
    return;

double cups = quantity * measureToCupRatio;

resultLabel.Text = "The number of cups: " + cups.ToString();
```

## Step 2: Fixing Existing Calls to calculateCups()

---

You may have noticed that when you added the measureToCupRatio input parameter for calculateCups() it causes all of the previous calls to calculateCups() to become decorated with a red squiggly line, indicating an error. The error simply means that you are not calling the method properly now that it *requires* a value to be given as an input parameter. But first, before we fix this, let's comment out the following call as it will become problematic and require a little more effort to get working properly:

```
protected void quantityTextBox_TextChanged(object sender, EventArgs e)
{
    //calculateCups(); ←
```

## Step 3: Add the Input Parameter's Argument at Call

---

Now, let's supply all of the calculateCups() calls with values relative to its calling method's conversion ratio (IE: 1:1 ratio between cups/cups, a 2:1 ratio between cups/pints, a 4:1 ratio between cups/quarts, and a 16:1 ratio between cups/gallons):

```

protected void cupsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups(1.0);
}

protected void fromPintsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups(2.0);
}

protected void fromQuartsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups(4.0);
}

protected void fromGallonsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups(16.0);
}

protected void quantityTextBox_TextChanged(object sender, EventArgs e)
{
    //calculateCups();
}

```



Bob's Tip: notice how all of this refactoring creates the same results as before when running the application. However what has really been improved has been the maintainability and readability of the code in the backend. Not all code improvements are visible to the end-user!

## Step 4: Understanding When To Use Input Parameters

The main rule of thumb for when and where to incorporate input parameters is to do so wherever you need to separate what remains the same, with what sometimes is different between each call to the helper method. In this case, what's different each time you call the method is the measure/cup ratio, whereas what's the same is the calculation details within the method's code block. In other words, we're adding variability to the method through whatever we choose to pass in via the `measureToCupRatio` input parameter, while the logic within the code block remains unchanged. To demonstrate this further and to illustrate how you can utilize more than one input parameter, let's modify the `resultLabel` to be more descriptive depending on which conversion was selected:

```
resultLabel.Text = String.Format
(
    "{0:N2} {1} is equal to {2:N2} cups.", quantity, measureName, cups
);
```

## Step 5: Adding More Input Parameters

---

This new variable – measureName – can now be supplied, at method call, as a second input parameter:

```
private void calculateCups(double measureToCupRatio, string measureName)
{
```

We will now have to modify all of the calculateCups() calls with a second input parameter passed in that assumes the place wherever measureName is referenced within the method internals. Note that you could add as many input parameters as needed, separating each input parameter with a comma:

```
protected void cupsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups(1.0, "cups");
}

protected void fromPintsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups(2.0, "pints");
}

protected void fromQuartsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups(4.0, "quarts");
}

protected void fromGallonsRadio_CheckedChanged(object sender, EventArgs e)
{
    calculateCups(16.0, "gallons");
}
```



Bob's Tip: input parameters – also referred to as “arguments” when supplied at the method’s call – function in much the same way as any other variable you have come to learn about thus far. By creating an input parameter, you are essentially declaring a variable *local to the method itself* with its assignment being determined by whatever value is copied/passed-in at the method call. That input parameter’s variable name then becomes the identifier that we use throughout the method’s code block wherever that variable needs to be referenced.

030

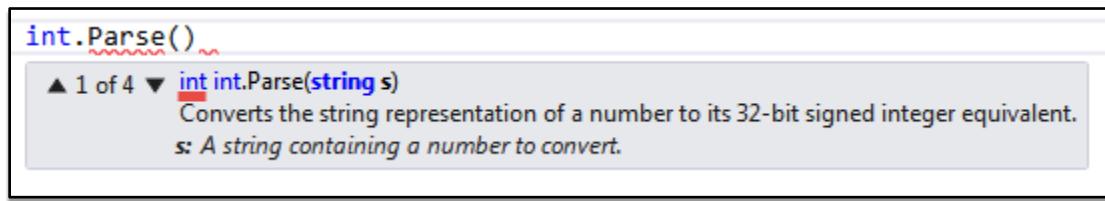
# Returning Values from Methods

In this lesson, you're going to learn how to return values from methods that you create. Up to this point, you have been working with methods that return `void` (in other words, nothing at all). The `void` keyword means that after the method is called no information is returned *back to the calling code block*. However, if you replace the `void` keyword with a type (`int`, `string`, `DateTime`, etc) you must specify within the method the actual value being returned and the caller will assume that value once the method it has called completes execution.

## Step 1: Recognizing Method Return Values in Intellisense

---

Some of the helper methods we have been working with thus far have been returning values back to the caller. For example, we know that the `int.Parse()` method takes in a `string`, and returns an `int`. In fact, Intellisense tells us this:



Knowing that `int.Parse()` returns a value, we can treat it as *being that value*. Here, we know that the value being returned will be 5, so what can you do with that value? You can do a number of things, and one of them is to assign it to another variable, of type `int`:

```
int num = int.Parse("5");
```

That means that this statement is semantically equivalent to:

```
int num = 5;
```

And, because those statements are semantically equivalent, you can do this:

```
int num = int.Parse("5") - int.Parse("3"); //calculates 2
```

## Step 2: Create a Custom Method with a Return Type

---

Let's create our own custom method with a return type in a new ASP.NET project called "CS-ASP\_030", which is based on where we left off with the previous lesson. In *Default.aspx.cs* write in some helper methods that we will later reference in *Page\_Load()*:

```
private void displayBattleHeader()
{
    resultLabel.Text += "<h3>Battle Between the Hero (you) " +
        "and the Monster (bad guy)</h3>";
}

private void displayRoundHeader()
{
    resultLabel.Text += "<hr /><p>Round begins ...</p>";
}

private void describeRound(string attackerName, string defenderName, int defenderHealth)
{
    if (defenderHealth <= 0)
        resultLabel.Text += String.Format("<br />{0} attacks {1} and vanquishes the {2}.",
            attackerName, defenderName, defenderName);
    else
        resultLabel.Text += String.Format("<br />{0} attacks {1}, leaving {2} with {3} health.",
            attackerName, defenderName, defenderName, defenderHealth);
}

private void displayResult(int heroHealth, int monsterHealth)
{
    if (heroHealth <= 0)
        resultLabel.Text += "<h3>Monster wins!</h3>";
    else if (monsterHealth <= 0)
        resultLabel.Text += "<h3>Hero wins!</h3>";
    else
        resultLabel.Text += "<h3>They are both losers!</h3>";
}
```



Bob's Tip: here again you see the `private` keyword prefixed to all of these methods. This has to do with whether or not outside classes can access (call) this method/element. This will make sense once you are exposed to Object-Oriented Programming principles in later lessons. However for now know that this keyword means these methods can *only* be called within the class they're defined (in this case, the Default class).

## Step 3: Calling Methods in the Hero/Monster Page\_Load()

---

These methods should be fairly self-explanatory, however, note that they all just add information to the resultLabel and therefore do not return anything back to the caller. Now, write the following in Page\_Load():

```
protected void Page_Load(object sender, EventArgs e)
{
    int heroHealth = 100;
    int monsterHealth = 100;

    displayBattleHeader();

    // Hero gets bonus first attack

    while (heroHealth > 0 && monsterHealth > 0)
    {
        displayRoundHeader();

        // Perform battle here!

    }

    displayResult(heroHealth, monsterHealth);
}
```

Here we are calling the helper methods to display "Battle/Round" header information and then once the battle completes, display the final result. However, this code won't work properly in its current state, since the while() loop will never break out of just displaying the "Round" header. We will have to write some battle code so let's do that in another helper method called performAttack (also, notice the various input parameters we will be using for this method to work):

```
private int performAttack(
    int defenderHealth,
    int attackerDamageMax,
    string attackerName,
    string defenderName)
{
    Random random = new Random();
    int damage = random.Next(1, attackerDamageMax);
    defenderHealth -= damage;

    return defenderHealth;
}
```



This method currently consists of applying a random damage value to `defenderHealth`, and then returning that value to the caller. However, we're not yet done with this method. We will also want to execute the `describeRound()` method within the `performAttack()` method. When you write the call to `describeRound()`, you will notice that Intellisense will give you a reminder as to the input parameters you are expected to supply (this is why it's important to use descriptive variable names):

```
describeRound()  
void Default.describeRound(string attackerName, string defenderName, int defenderHealth)
```

We will supply these values based on what values are supplied for the outer calling method `performAttack()`. You can think of it like you are copying these values, whatever they might be, once they are passed in when calling `performAttack()`:

```
private int performAttack(  
    1 int defenderHealth,  
    int attackerDamageMax,  
    2 string attackerName,  
    3 string defenderName)  
{  
    Random random = new Random();  
    int damage = random.Next(1, attackerDamageMax);  
    defenderHealth -= damage;  
    2     3     1  
    describeRound(attackerName, defenderName, defenderHealth);  
  
    return defenderHealth;  
}
```

## Step 4: Assigning a Method's Returned Value to a Variable

---

You can now pass in these values by calling the method in `Page_Load()`, as follows:

```
// Hero gets bonus first attack  
monsterHealth = performAttack(monsterHealth, 20, "Hero", "Monster");
```

Recall that `performAttack()` returns an integer value, so that is why it's being assigned to `monsterHealth` `int` variable. Finally, we're reusing this attack procedure, so let's also make a few more calls to it in the main battle logic within the `while()` loop:

```
// Perform battle here!
heroHealth = performAttack(heroHealth, 20, "Monster", "Hero");
monsterHealth = performAttack(monsterHealth, 20, "Hero", "Monster");
```



Bob's Tip: there are several methods here that each have a specific purpose (and that's hinted at by the name we chose for each method). This relates to a general programming concept called the "Single Responsibility" principle. Put simply, the more that you break-up your code into individual modules with a single, direct purpose, the easier it will be to keep track of everything in your code and reduce bugs.

## Step 5: Run the Application

---

You can now run the application and see the result of this battle simulation:

Round begins ...

Monster attacks Hero, leaving Hero with 19 health.  
Hero attacks Monster, leaving Monster with 10 health.

Round begins ...

Monster attacks Hero, leaving Hero with 10 health.  
Hero attacks Monster, leaving Monster with 1 health.

Round begins ...

Monster attacks Hero, leaving Hero with 1 health.  
Hero attacks Monster and vanquishes the Monster.

**Hero wins!**

031

# Creating Overloaded Methods

In this lesson, you will learn about *overloaded methods*. You may have noticed with Intellisense that some of the built-in helper methods that we've been using have *several versions* of the same method that you can choose from. You may also have noticed that these different versions are differentiated by the input parameters that they require and that in a nutshell is what overloaded methods are: several methods that share the same method name, yet have different input parameters and possibly have different implementation details as well.

## Step 1: Create a New Project

---

To demonstrate method overloading, create a new ASP.NET project based on where we left off with the previous lesson and add three new methods to the code in *Default.aspx.cs*:

```
public void displayFullStatsOfTheMonster(
    string monsterName,
    int health,
    int damageMaximum,
    double criticalHitChance)
{
    resultLabel.Text += String.Format("<p>{0} Current Stats<br />Health: "+
        "{1}<br />Damage Max: {2}<br />Critical Hit Chance: {3:P}</p>",
        monsterName, health, damageMaximum, criticalHitChance);
}

public void displayMonstersPartialStats(
    string monsterName,
    int health,
    int damageMaximum)
{
    resultLabel.Text += String.Format("<p>{0} Current Stats<br />Health: "+
        "{1}<br />Damage Max: {2}</p>", monsterName, health, damageMaximum);
}

public void displayMonstersNameAndHealth(
    string monsterName,
    int health)
{
    resultLabel.Text += String.Format("<p>{0} Current Stats<br />Health: "+
        "{1}</p>", monsterName, health);
}
```

Even at a quick glance, you will probably notice that these three methods have the same basic purpose – displaying stats and information about the monster. However they differ in the amount of information they output based on the input parameters provided.

## Step 2: Understanding When to Overload Methods

---

The problem is that you, the programmer, have to remember the different names for these methods and know how to find them in Intellisense. This may not sound like a tremendous challenge at first. However, as an application grows it just adds to possible confusion and friction in the development process. To alleviate this, let's simply *overload* these methods by naming them with the exact same identifier:

```
public void displayMonsterStats(  
    string monsterName,  
    int health,  
    int damageMaximum,  
    double criticalHitChance) ...  
  
public void displayMonsterStats(  
    string monsterName,  
    int health,  
    int damageMaximum) ...  
  
public void displayMonsterStats(  
    string monsterName,  
    int health) ...
```

## Step 3: Distinguishing Between a Method's Body and Signature

---

For this to compile properly the .NET Framework needs to determine that these methods are separate despite sharing the same name. The way that it does this is by looking at the method signature, which is the set of “header information” prior to the method’s code block:

```
private void MethodSignature(object input1, object input2, object input3)  
{  
}  
}
```

This method signature is defined by:

- its accessibility level (private).
- its return type (void).
- Its identifier (MethodSignature).
- Its amount, and type, of parameters (object input1, object input2, object input3).

The input parameters are the only part of the method signature that the compiler uses to differentiate each method sharing the same name, yet still grouping them together as overloads of the same method. As long as the amount and the type of parameters used are different, it will be considered a valid overload (bear in mind that the input parameter *names* are not factored in):

```
private void MethodSignature(object input1, object input2, object input3)
{
    //3 input parameters; all of type: object
}

private void MethodSignature(object input1, object input2)
{
    //2 input parameters; all of type: object
}

public string MethodSignature(string input1, object input2, object input3)
{
    //3 input parameters; one of type: string; two of type: object
    return "It's overloaded!";
}
```

## Step 4: Finding Available Overloads with Intellisense

---

Notice also how the third example has a different return type, as well as a different accessibility modifier. Since these elements of the method signature are *not* factors towards determining method overloading, they do not pose a problem and Intellisense will still group them together:



Going back to the project, you can now try to call `displayMonsterStats()` and with the up/down arrow keys you can cycle through the variations available to you:

displayMonsterStats()

▲ 1 of 3 ▼ void Default.displayMonsterStats(string monsterName, int health)

displayMonsterStats()

▲ 2 of 3 ▼ void Default.displayMonsterStats(string monsterName, int health, int damageMaximum)

displayMonsterStats()

▲ 3 of 3 ▼ void Default.displayMonsterStats(string monsterName, int health, int damageMaximum, double criticalHitChance)



Bob's Tip: overloaded methods are particularly useful when you're writing code that may be used by *another* developer to create their own code. Consider how useful it has been for you, the consumer/developer of the .NET library, to be able to use Int.Parse(), Random.Next(), String.Format(), etc.

032

# Creating Optional Parameters

This is a short lesson that covers an interesting feature of methods in C# and that is the allowance of *optional parameters*. The “optional” part refers to whether or not the values for the parameter are supplied upon calling the method. If the programmer decides not to supply any values for the parameter, it then defaults to a value assigned in the method signature itself.

## Step 1: Create a New Project

---

Refer to where we left off with the previous lesson and use that as the starting-point for this one, and call the project “CS-ASP\_032”. And in the existing `performAttack()` method modify it to include another optional input parameter:

```
private int performAttack(
    int defenderHealth,
    int attackerDamageMax,
    string attackerName,
    string defenderName,
    double criticalHitChance = 0.1)
{
    Random random = new Random();
    int damage = random.Next(1, attackerDamageMax);
    defenderHealth -= damage;

    describeRound(attackerName, defenderName, defenderHealth);
    return defenderHealth;
}
```



## Step 2: Using Optional Parameter's for Default Values

---

Notice how this input parameter includes a value already assigned to it, determined before the method is even called. Also, notice that our existing calls to this method still work even though they are missing this input parameter. The other (non-optional) parameters are absolutely required and the compiler would make sure you include them otherwise it will produce an error:

```
// Hero gets bonus first attack
monsterHealth = performAttack(monsterHealth, 20, "Hero", "Monster");
```

The way this works here is criticalHitChance automatically takes on the value defined in the method signature unless it is explicitly *overridden* with a different value supplied at the method call. You will be able to identify optional parameters with Intellisense, which shows them wrapped in square brackets along with their default value:

```
performAttack(monsterHealth, 20, "Hero", "Monster");
int Default.performAttack(int defenderHealth, int attackerDamageMax, string attackerName, string defenderName, [double criticalHitChance = 0.1])
```

## Step 3: Overriding the Optional Parameter's Default Value

---

Now here is the performAttack() method with the optional parameter designated at the method call:

```
// Hero gets bonus first attack
monsterHealth = performAttack(monsterHealth, 20, "Hero", "Monster", 0.4);
```

You can add as many optional parameters as you want but they *must* be appended after the non-optional input parameters:

```
private int performAttack(
    int defenderHealth,
    int attackerDamageMax,
    string attackerName,
    string defenderName,
    double criticalHitChance = 0.1,
    double defenderArmorBonus = 5.0)
{
```

```
// Hero gets bonus first attack
monsterHealth = performAttack(monsterHealth, 20, "Hero", "Monster", 0.4, 20.0);
```

## Step 4: Constraints When Omitting Optional Parameter Arguments

---

Now, let's see what happens if we want to omit the first optional parameter at the method call:

```
// Hero gets bonus first attack
monsterHealth = performAttack(monsterHealth, 20, "Hero", "Monster", , 20.0);
```

You will notice that the compiler doesn't let you do this and that's because the only parameter that is truly optional, in all cases, is the *last* optional parameter within the sequence. So, you can safely omit this last optional parameter, even if you choose to use the optional parameters that came before it:

```
// Hero gets bonus first attack
monsterHealth = performAttack(monsterHealth, 20, "Hero", "Monster", 0.4);
```



Bob's Tip: you may have noticed that we didn't actually reference the optional parameters in the body of the `performAttack()` code block. Obviously you would want to make use of all of your input parameters in a real-world code project, however, this example is simply for the purpose of demonstrating how optional parameters work.

033

# Creating Named Parameters

In the previous example we saw how optional parameters can be added to method signatures. However, there was a catch that was noted about optional parameters, which is that only the last optional parameter is truly optional. This leads to an “all-or-nothing,” or, “all-except-for-the-last-one-or-nothing” situation. One way around this is with something called *named parameters*. All that this entails is naming the parameters, along with supplying the values, in the method call itself.

## Step 1: Reference Parameters by Name in the Call Arguments

---

Referring to where we left off in the previous lesson, let’s demonstrate this in the call to `performAttack()`:

```
// Hero gets bonus first attack
monsterHealth = performAttack(
    defenderHealth: monsterHealth,
    attackerDamageMax: 20,
    attackerName: "Hero",
    defenderName: "Monster");
```

Here, we (1) included the actual parameter names in the call itself, and (2) supplied the value after the colon:

```
monsterHealth = performAttack(
    defenderHealth: monsterHealth,
    attackerDamageMax: 20,
    attackerName: "Hero",
    defenderName: "Monster");
    1           2
```

Now, you can add any optional parameter and safely omit any other optional parameters by including it as a named parameter. However, it’s important to note that you still have to include every non-optional parameter in your method call even though the ordering is no longer important:

```
// Hero gets bonus first attack
monsterHealth = performAttack(
    defenderHealth: monsterHealth,
    attackerDamageMax: 20,
    attackerName: "Hero",
    defenderName: "Monster", 
    defenderArmorBonus: 5.0);
```

This allows us to safely omit the optional parameter that came before it:

```
private int performAttack(
    int defenderHealth,
    int attackerDamageMax,
    string attackerName,
    string defenderName,
    double criticalHitChance = 0.1,
    double defenderArmorBonus = 5.0)
{
```

## Step 2: Customize Parameter Order with Named Parameters

---

One of the interesting features of named parameters is that they can now occur in any order you choose. This works because, normally, the order in which you supply parameter values is the only way the compiler knows which parameter you are supplying the value for. However now you are directly referring to the parameter by name:

```
// Hero gets bonus first attack
monsterHealth = performAttack(
    attackerDamageMax: 20,
    defenderArmorBonus: 5.0,
    defenderName: "Monster",
    attackerName: "Hero",
    defenderHealth: monsterHealth);
```



Bob's Tip: named parameters are a relatively new feature of C#. However, just because it's a feature doesn't mean you *should* use it under normal circumstances, as it is likely to cause confusion in your code. While rarely implemented, named parameters can have a use. For instance, you might find yourself working with a codebase that you did not originally write and you want to re-order the way the input parameters look, for the sake of clarity – but don't want to, or can't, change the original method definition for fear of breaking existing calls to it.

034

# Creating Methods with Output Parameters

This lesson will cover the `out` keyword that is used alongside ordinary input parameters in order to output a value other than the return value. We already saw it used in a previous lesson that called the `double.TryParse()` method where we witness how the output parameter sent a value back out of the method, similar to the way a return value works but with a twist.

## Step 1: Create a New Project

---

You should create a project for this lesson based on where we left off with the previous lesson, and call it "CS-ASP\_034". In the previous lesson we had a method called `performAttack()` that held most of the battle code, however, in this lesson we will want to replace it with a new method called `defeatEnemy()` that functions a bit differently. Here we see that the method still performs the main battle algorithm, but it also returns a `bool` depending on whether or not the enemy has zero or less remaining health:

```
private bool defeatEnemy(int defenderHealth,
    int attackerDamageMax,
    string attackerName,
    string defenderName)
{
    Random random = new Random();
    int damage = random.Next(1, attackerDamageMax);
    int remainingDefenderHealth = defenderHealth - damage;

    if (remainingDefenderHealth <= 0) return true;
    else return false;
}
```

## Step 2: Using the 'out' Keyword as Extra Method Return

---

Suppose that you want the `defeatEnemy()` method to also return the defender's health. Since we're only allowed a single method return value, the best way around this is to have the defender's health value come back out as an `out` parameter:

```
private bool defeatEnemy(int defenderHealth,
    int attackerDamageMax,
    string attackerName,
    string defenderName,
    out int remainingDefenderHealth)
{
    Random random = new Random();
    int damage = random.Next(1, attackerDamageMax);
    remainingDefenderHealth = defenderHealth - damage;
    if (remainingDefenderHealth <= 0) return true;
    else return false;
}
```

Now you can reference this method within a conditional statement and perform some sort of action depending on the returned value:

```
while (heroHealth > 0 && monsterHealth > 0)
{
    displayRoundHeader();

    // Perform battle here!
    if (defeatEnemy(heroHealth, 20, "Monster", "Hero", out heroHealth))
        lootEnemy();
}

displayResult(heroHealth, monsterHealth);
```

And now create the `lootEnemy()` method, which simply displays a message through `resultLabel.Text`:

```
private void lootEnemy()
{
    resultLabel.Text += "<p style='color:red;'>A magic sword "+  
    "and 20 gold pieces are collected from corpse.</p>";
}
```

## Step 3: Understanding Why the 'out' Keyword Is Necessary

---

It may not be clear why the `out` keyword is needed here. The way that input parameters normally work is that local variables provided as input at method calls have their values *copied* into the local input parameter and so there is no reference retained back to the original variable where the value was copied from. Whatever information is processed in the method is local to the method itself, unless it comes back out as a returned value, or an output parameter.



Bob's Tip: you can almost think of methods as being their own little "universe" that may hold copies of variables from another "universe," supplied via input parameters. And the only way that a method can "talk back" to the world it copied information from is via the `return`, `out` and `ref` keywords, or else if there is a class-level variable referenced in the method. Otherwise, a method's internals are completely sealed off from its surroundings. Also, note that although the `out` keyword can solve a variety of problems, it is not very common, and should be used sparingly considering that it can be seen as violating the principle of "Single Responsibility."

035

# Manipulating Strings

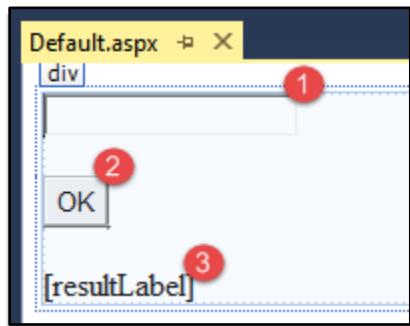
In this lesson, you learn some interesting properties of strings, as well as several helper methods available via the `String` class that will allow you to perform a variety of useful processes on strings.

## Step 1: Create a New Project

---

Begin this lesson by creating a new ASP.NET project called "CS-ASP\_035" with the following programmatic IDs and Server Controls:

- (1) `valueTextBox`
- (2) `okButton`
- (3) `resultLabel`



In `okButton_Click()` let's first output a simple message within HTML:

```
protected void okButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = "<p style='color:#ee3b32;'>Hi</p>";
}
```

Two red arrows point to the double quotes in the string "color:#ee3b32;" within the `style` attribute of the `<p>` tag in the C# code. This highlights a common issue with string concatenation in ASP.NET.

You will see that there is a conflict between the double-quotes used for the HTML and the double-quotes used for the entire string. It's easy to see why this is a problem: the compiler thinks that the string is ending with a second double-quote when the first double-quote is referenced in the HTML tag:

```
<p style="color:#ee3b32;">
```

## Step 2: Make a Character Literal with Backslash

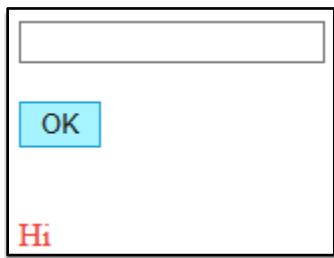
---

This is fixed by having the compiler read the HTML quotes as *literal* quotes, and is achieved by placing a backslash before them, thereby “escaping” the string sequence. You can use the backslash in such a way to turn any character in a string into a *literal* character:

```
resultLabel.Text = "<p style=\"color:#ee3b32;\">Hi</p>";
```

Another common solution to this problem is to use double-quotes for strings and single-quotes for HTML:

```
resultLabel.Text = "<p style='color:#ee3b32;'>Hi</p>";
```



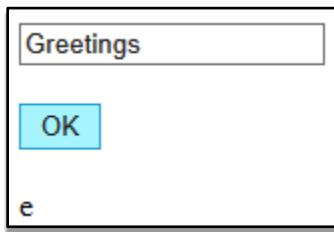
## Step 3: Handling a String as an Array of Char

---

It's common when parsing through a string to want to isolate a single character. You can do this quite easily because a string is actually made up of an array of characters of type char. Here, we're going to output the third character in the string:

```
protected void okButton_Click(object sender, EventArgs e)
{
    string value = valueTextBox.Text;

    //Access any specific character as
    //its indexed position in an array
    resultLabel.Text = value[2].ToString();
}
```



## Step 4: StartsWith(), EndsWith() and Contains() String Helper Methods

---

Another common need is to parse through a string to find out some particular information that it contains. Here are a few useful helper methods to this end:

```
protected void okButton_Click(object sender, EventArgs e)
{
    string value = valueTextBox.Text;

    //StartsWith(), EndsWith(), Contains() helper methods
    if (value.StartsWith("A"))
        resultLabel.Text += "Value starts with 'A' ";

    if (value.EndsWith(".")) 
        resultLabel.Text += "Value ends with '.' ";

    if (value.Contains("good"))
        resultLabel.Text += "Value contains 'good' ";
}
```

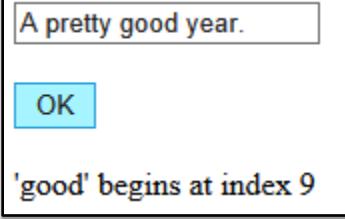
## Step 5: Using IndexOf() for Finding Position of a String Sequence

---

You can also find the indexed position of a particular string sequence within an entire string:

```
protected void okButton_Click(object sender, EventArgs e)
{
    string value = valueTextBox.Text;

    //Find the index position of a string sequence
    //within a string by using IndexOf()
    int index = value.IndexOf("good");
    resultLabel.Text = "'good' begins at index " + index.ToString();
}
```

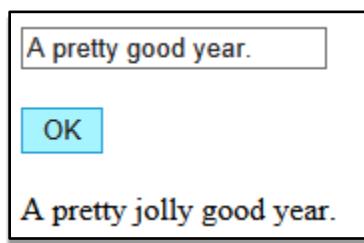


## Step 6: Using Insert() to Insert a New Substring

---

And then you can use that index information to insert a new string at that position:

```
int index = value.IndexOf("good");
resultLabel.Text = value.Insert(index, "jolly ");
```



## Step 7: Using Remove() to Remove a Substring

---

You can also (1) remove a number of characters beginning at a particular (2) index position in the string. In this example, we're removing the remaining characters in the string by calculating the string length, minus the starting position index:

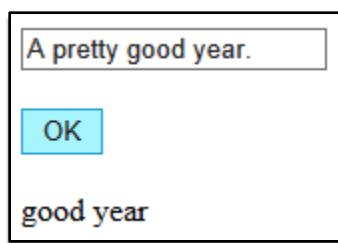
```
int index = value.IndexOf("good");
resultLabel.Text = value.Remove(index, value.Length - index);
    1           2
```

## Step 8: Using Substring() to Return a Substring Sequence

---

And here we use the Substring() method to return the string sequence we are looking for, along with the rest of the string that comes after it, minus the last character:

```
int index = value.IndexOf("good");
resultLabel.Text = value.Substring(index, value.Length - index - 1);
```

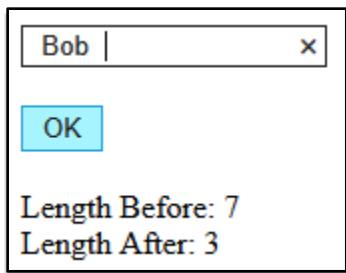


## Step 9: Using Trim() Remove Whitespace

---

It's common to want to isolate a string and then trim any whitespace or empty characters. The easiest way to do this is with the Trim() method:

```
resultLabel.Text = String.Format(  
    "Length Before: {0}<br />Length After: {1}",  
    value.Length, value.Trim().Length  
);
```



Here we trimmed off two empty spaces from, both, the beginning and the end of the string. You could also just trim these two opposite sides individually with the TrimStart() and TrimEnd() methods, respectively.

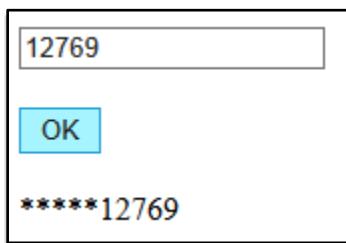
## Step 10: Using PadLeft() and PadRight() to Add Whitespace

---

You can also do the opposite of trimming by adding padding to the string sequence with the PadLeft() and PadRight() methods. The PadLeft() method adds (1) the number of characters spaces to add to the left of the string sequence and (2) the character placed in that padded space (note that this value is of type char, so it has to be enclosed in single-quotes):

```
resultLabel.Text = value.PadLeft(10, '*');
```

1      2



Suppose that you want to check if one string is equivalent to another string but you want to ignore case sensitivity. Strings are normally case sensitive, so something like this will evaluate as false:

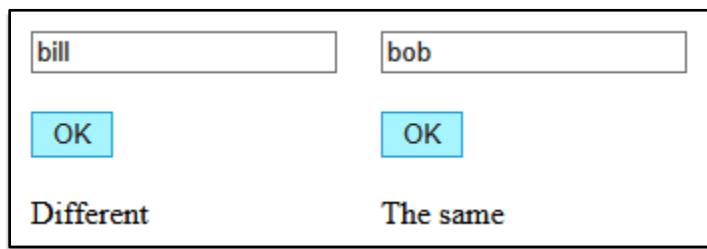
```
bool evaluate = ("Bob" == "bob"); //false
```

## Step 11: Using ToUpper() to Bypass Case Sensitivity

---

A common solution to this is to set all characters in the string to upper-case, or lower-case (here we're also trimming the string for good measure):

```
if (valueTextBox.Text.Trim().ToUpper() == "BOB")
    resultLabel.Text = "The same";
else
    resultLabel.Text = "Different";
```



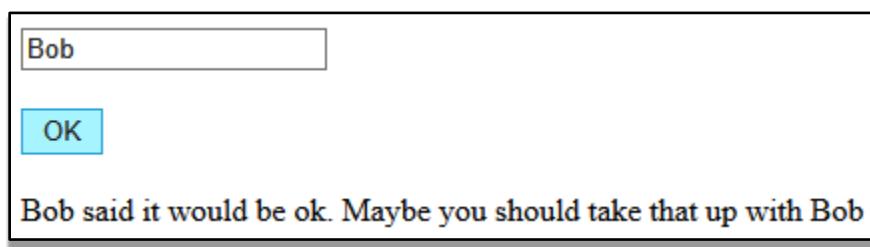
## Step 12: Using Replace() To Apply Templated String Formatting

---

It's very common to have to replace characters within a string that doesn't match the formatting requirements of your application. Here, we're using the Replace() method to create a templated string:

```
string template = "[NAME] said it would be ok. " +
    "Maybe you should take that up with [NAME]";

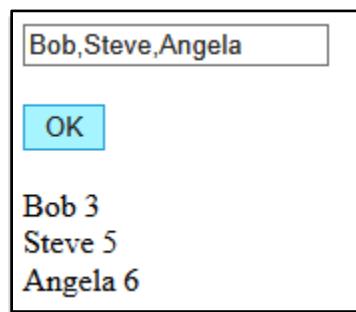
resultLabel.Text = template.Replace("[NAME]", valueTextBox.Text);
```



## Step 13: Using Split() to Split a String into String[] Array

Another useful task is splitting up a string into an array of smaller strings based on some form of delimiter (such as a comma, dash, and so on). The code below is (1) splitting the values entered by the user wherever there is a comma and (2) iterating through the array items adding each to string result along with (3) the length of the current string:

```
string result = "";
string[] values = valueTextBox.Text.Split(',');
for (int i = 0; i < values.Length; i++)
    result += values[i] + " " + values[i].Length + "<br/>";
resultLabel.Text = result;
```



Bob's Tip: all of these string manipulation methods work based on the same principles that we saw in the previous lessons on arrays. Because a string is an array of characters, it is subject to the same constraints of immutability that other arrays face (arrays can't inherently resize, remove, elements, etc). So whenever you want to change an aspect of a string, it has to go through the process arrays go through of copying values from one array to another but with x number of elements more/less, etc. Be aware that as you manipulate strings over and over again you are adding more and more processing, and memory, load.

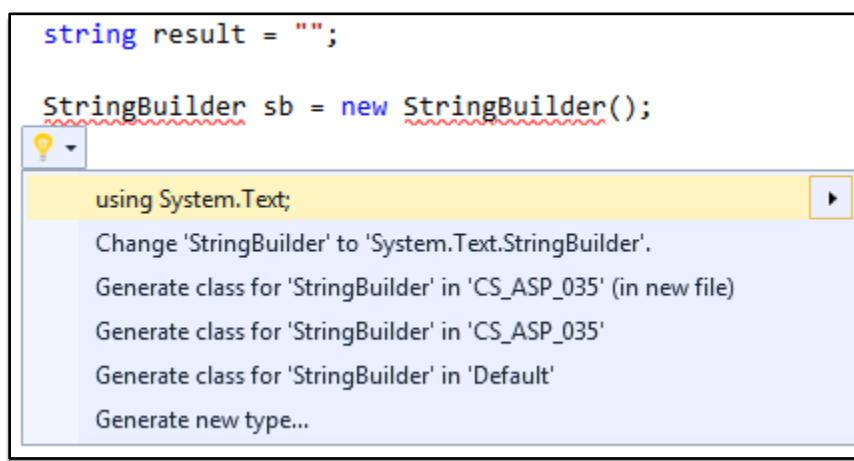
## Step 14: Using the StringBuilder Class

There is a built-in class within the .NET Framework – called `StringBuilder` – that makes certain processes on strings more efficient. Here is how you create a `StringBuilder` object/variable:

```
StringBuilder sb = new StringBuilder();
```

However, Intellisense will show that it doesn't have access to the `StringBuilder` class (there will be red squiggly underlines). You will have to add a reference to point to that information and the easiest way to do this is to click on the underline and enter the following keys to get a references menu to pop-up:

Ctrl+Period



## Step 15: Add the Using Declaration to Access `StringBuilder` Class

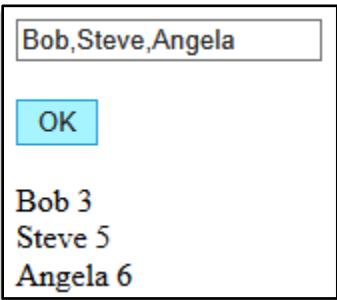
---

Add the "using System.Text;" reference to your project by clicking on it and it will be added to the top of the `Default.aspx.cs` file (more on using declarations and namespaces in later lessons):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text; ← Red arrow pointing here
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
```

Now that we can reference the `StringBuilder` class, let's re-create the previous process but with the more efficient `StringBuilder.Append()` method:

```
StringBuilder sb = new StringBuilder();
string[] values = valueTextBox.Text.Split(',');
for (int i = 0; i < values.Length; i++)
{
    sb.Append(values[i]);
    sb.Append(" ");
    sb.Append(values[i].Length);
    sb.Append("<br />");
}
resultLabel.Text = sb.ToString();
```



It's worth noting that the `StringBuilder` optimization is best used when there is an intensive process involved. If you are manipulating a few strings over a short duration of time, then you can use the ordinary `string` methods without seeing any significant reduction in performance. There are specific tools you can use to measure the performance of your code and respond with optimizations to remove bottlenecks. However, that is best left for when you know how to use the tools and spot low-performant code.

036

# Introduction to Classes and Objects

There has been an elephant in the room throughout the previous lessons and that elephant goes by the name of "class" or "object." This topic has to do with Object-Oriented Programming (or "OOP"), which is a shift in thinking that takes time to really understand its significance. OOP takes the basic concepts you already understand (variables, methods, scope) and separates them into a more conceptual framework; grouping one set of variables and methods in one class and grouping others in another, for example. And from those classes you can build "instances" that are bound up in a special kind of variable called an object. Those objects then work together as communities to solve problems. Objects (and therefore, classes) have a host of special properties that we have yet to see. They can inherit from other classes in order to add functionality and create a more flexible software system. And they can reduce the amount of dependency in your system. Dependency is a problem because if one piece of code were to break, it could break another piece of dependent code. Classes are an important step towards reducing dependency and allowing you to enforce the "separation of concerns" principle we spoke of earlier.



Bob's Tip: by keeping code as separate as possible, using various object-oriented techniques, you are de-coupling dependencies within your code. Coupling is most glaring when you are debugging and start to notice how "whacking" a bug in one part of your code causes another bug to "pop up" elsewhere. And if it only happens under certain runtime conditions, you have the makings for a real coding nightmare.

## Step 1: Distinguishing OOP and Procedural Programming Styles

---

To distinguish OOP from non-OOP you could contrast it with the style of programming we have been mostly doing up to this point which is based on a *procedural* style. This is an old style of programming where you work in terms of data in/data out. This style of programming uses a lot of carefully named variables and methods to represent data points and processes. These data points might be loosely coupled together - perhaps by some kind of naming convention - but we haven't been thinking of the data as a representative piece of a larger architecture. And beyond the problem-solving logic, we haven't considered maintainability or how change introduced within the system will affect the resiliency of the existing code. We also haven't really worried about reusability of code by breaking it up into isolated components that would work regardless of the system built around, and accessing, it.

## Step 2: Why Bother Learning OOP?

---

In contrast to the procedural style of programming detailed above, when we think in terms of objects that interact with each other - this object needs to talk to this other object - to accomplish a solution as a piece within a broader problem, we are putting a priority on all of these elements that procedural programming doesn't concern itself with. This added complexity often leaves beginners wondering if this higher layer of concern – beyond just solving the problem – is worthwhile for the average programmer, especially considering that most beginning applications are actually quite simple. However, what you will come to learn is that the conceptual nature of OOP eventually allows you to create more complex applications without the burden of confronting that immediate complexity. OOP will, ultimately, allow you to concern yourself less with mundane issues – abstracting them away into nice little bundles that you perhaps rarely have to look at and understand in full. It will allow you to focus better on individual problems that need solving, within a broader puzzle, and in the software development world there is always a broader puzzle.



Bob's Tip: the subject of Object-Oriented-Programming is both relatively easy to pick up on and extremely deep if you want to continue pursuing it. It can take time to fully understand, and appreciate, how much it offers. Don't feel bad if you don't really understand it all the first time through. The best way to look at learning OOP is its return on time investment: the relatively modest amount of extra time you put in now to learn its secrets pales in comparison to the time you *gain back* when using it to build robust and extensible applications. Just keep working through the examples over the next few lessons and you will eventually start seeing the world of code around you as made up of objects.

## Step 3: You've Already Been Doing OOP without Realizing It

---

Virtually everything in C# - including the part of the .NET Framework that deals with ASP.NET functionality – is either a class, or part of a class. In all of the previous lessons, we have been using ASP.NET Web Forms with a *Default.aspx* file. What you may not have realized is we were just creating a definition – a set of methods and properties – for a class in that file.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Web;
5 using System.Web.UI;
6 using System.Web.UI.WebControls;
7
8 namespace CS_036
9 {
10     public partial class Default : System.Web.UI.Page
11     {
12         protected void Page_Load(object sender, EventArgs e)
13         {
14         }
15     }
16 }
17 }
```

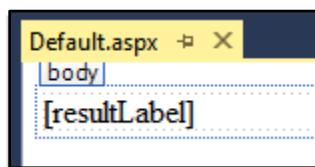
What you haven't yet seen is that the ASP.NET Runtime will create an instance of this *Default* class whenever the user requests it over the internet. And then once it creates an instance of that class it will begin calling methods or setting/retrieving properties, part of that being determined by what you write in the code for this *Default* class. In case you are wondering the ultimate responsibility for this class is to generate HTML that represents a webpage called "default."

The purpose of this lesson and the next few lessons is simply to understand classes and their ubiquity when working with C#. For this lesson, suppose that you wanted to create an application that works with cars in some way. You may have a car lot with an inventory of all the cars that are for sale, or you may want to keep related information about a single car in one container. OOP allows you to keep all of this related information within a class that represents a car in the lot.

## Step 4: Create a New Project

---

Begin by creating a new ASP.NET project with a single Server Control resultLabel :



In *Default.aspx.cs*, create a new class called Car alongside the Default class, keeping both classes within the broader "CS-ASP\_036" namespace:

```
namespace CS_ASP_036
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

    }

    class Car
    {

}
}
```

## Step 5: Methods and Properties as Class Members

---

What we're doing here is naming a code-block ("Car") similar to how methods are named blocks of code that perform some particular process that we can later reference as we do with any variable or method. Within this code block we have two basic "members" that immediately belong to it and those members are typically variables (properties) and methods. Class-level properties are much like any variable we have worked with thus far but are usually meant to describe attributes common to that object's class. For example, a car is typically defined by properties such as:

- Make
- Model
- Year Built
- Color

Class-level methods also define something about the class, and that is what the class can *do* (tasks, processes, behaviors, actions, and so on). A car can among other things do things such as:

- Accelerate
- Decelerate

## Step 6: Understanding Properties

---

In this sense, classes often are created to *model* - insofar as it's important to the functioning of our application - the behavior and physical properties of their real-world counterparts. Let's start off by adding the following properties to this class:

```
class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }
}
```

Notice that these are just like ordinary variables – with a type, and an identifier – except for the accessibility prefix (in this case, `public`) as well as the `get;` and `set;` postfix, which can be seen as the read/write attributes. You can access the code snippet for this by typing in “prop” and hitting the tab key twice:

```
public int MyProperty { get; set; }
```

---

## Step 7: Objects as Class Instances

Now, to create an instance or actual object of our car in code we can do so by referencing the class as we do any variable type:

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Car myNewcar = new Car();
    }
}
```

By creating an instance of the class we've defined, you can look at it like the class is a *blueprint*, while the object instance is the *actual object* itself from that blueprint. And just like you can create many object instances in the real world from a single blueprint, you can do so as well with objects in code:

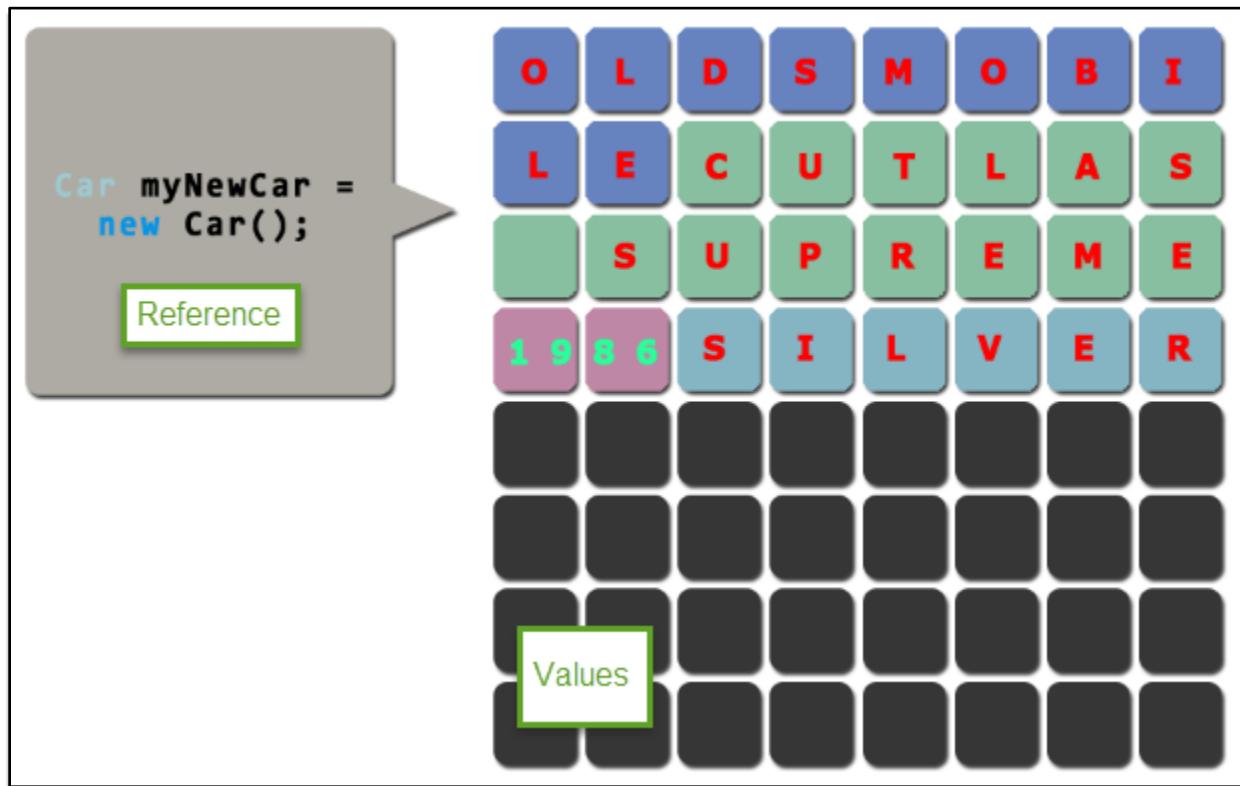
```
//two object instances built from a  
//single class blueprint describing a "Car"  
Car myNewcar = new Car();  
Car myOtherNewCar = new Car();
```

Each instance can now have its own unique values for its properties that you can *set* as you would any variable:

```
Car myNewcar = new Car();  
  
myNewcar.Make = "Oldsmobile";  
myNewcar.Model = "Cutlas Supreme";  
myNewcar.Year = 1986;  
myNewcar.Color = "Silver";
```

## Step 8: Understanding Object Instances in Memory

Each individual value is stored in sections of memory, while another section of memory hold onto myNewCar as a *reference* to these memory sections and their *values*:



Now that you have an instance of a Car in memory, with its included properties, you can *get* those values in memory (again, just like any ordinary variable):

```
resultLabel.Text = String.Format("{0} - {1} - {2} - {3}",
    myNewcar.Make,
    myNewcar.Model,
    myNewcar.Year,
    myNewcar.Color);
```

Since Car is a type all its own, you can use it and reference it anywhere. For instance, as an input parameter in a method:

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e) ...

    private double determineMarketValue(Car car)
    {
        // Some awesome algorithm goes here that
        // goes online and looks up the car's value
        // But, for now, let's just say:
        double carValue = 100.0;

        return carValue;
    }
}
```

## Step 9: Using Your Custom Class as Any Other Type

---

Notice here that the variable identifier (name) is the same as the type, except that has a different casing. This is perfectly acceptable – even if they share the exact same casing – as the compiler can distinguish between the object's name and its type. Also, notice that the input parameter is not used in this method, however, you would want to make use of the input parameter in a real-world example. But for the sake of brevity imagine that this method is doing something useful with the information we have about the car that we supply as an input argument at the method call:

```
Car myNewcar = new Car();

myNewcar.Make = "Oldsmobile";
myNewcar.Model = "Cutlas Supreme";
myNewcar.Year = 1986;
myNewcar.Color = "Silver";

double myMarketValueofCar = determineMarketValue(myNewcar);
```

And now you can add the value returned into myMarketValueofCar to the resultLabel :

```
resultLabel.Text = String.Format("{0} - {1} - {2} - {3} - {4:C}",
    myNewcar.Make,
    myNewcar.Model,
    myNewcar.Year,
    myNewcar.Color,
    myMarketValueofCar);
```

Oldsmobile - Cutlas Supreme - 1986 - Silver - \$100.00

To make this a bit more interesting, let's actually show how you would go about referencing the input parameter within the method body for determineMarketValue():

```
private double determineMarketValue(Car car)
{
    double carValue = 100.0;

    if (car.Year > 1990)
        carValue = 10000.00;
    else
        carValue = 2000.00;

    return carValue;
}
```

Oldsmobile - Cutlas Supreme - 1986 - Silver - \$2,000.00

## Step 10: Using Classes to Maintain Separation of Concerns

---

If you think back to what was said in previous lessons about “Separation of Concerns” it seems that the `determineMarketValue()` method violates this principle somewhat. In specific, it’s a method that calculates the value of a car, so putting it in the `Default` class (which is about rendering web page data) is a bit off the mark. Let’s transplant this method to the `Car` class, where it conceptually fits in a bit better, but with some modifications first:

- (1) Change the accessibility to “public,” so that outside classes can call this method.
- (2) Remove the input parameter as it is no longer required.
- (3) Directly reference the `Year` property, now that the method can “reach-out” of its scope and reference it at the class-level.

```
class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }

① public double determineMarketValue()
{
    ②     double carValue = 100.0;

        ③     if (Year > 1990)
            carValue = 10000.00;
        else
            carValue = 2000.00;

    return carValue;
}

}
```

## Step 11: Using the Dot Accessor To Reference Instance Members

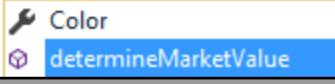
---

Now that the method belongs to the Car class, we will have to reference it through the Car instance in outside classes, just as we would the Car properties:

```
Car myNewcar = new Car();

myNewcar.Make = "Oldsmobile";
myNewcar.Model = "Cutlas Supreme";
myNewcar.Year = 1986;
myNewcar.Color = "Silver";

double myMarketValueofCar = myNewcar.|
```



The screenshot shows a code editor with the following snippet:  
Car myNewcar = new Car();  
  
myNewcar.Make = "Oldsmobile";  
myNewcar.Model = "Cutlas Supreme";  
myNewcar.Year = 1986;  
myNewcar.Color = "Silver";  
  
double myMarketValueofCar = myNewcar.|  
  
A tooltip-like window is open over the final dot character of the line 'myNewcar.|'. It contains two items:  
- A yellow item labeled 'Color' with a wrench icon.  
- A blue item labeled 'determineMarketValue' with a gear icon.

```
double myMarketValueofCar = myNewcar.determineMarketValue();
```



Bob's Tip: as you've seen, you can use the dot accessor (period) to peer into a class containment, in order to get/set properties or call methods. In later lessons you will see how class properties can, themselves, be instances of *other* classes. This creates multiple levels of containment that can be accessed by using multiple dot accessors; peering through one containment level after another.

037

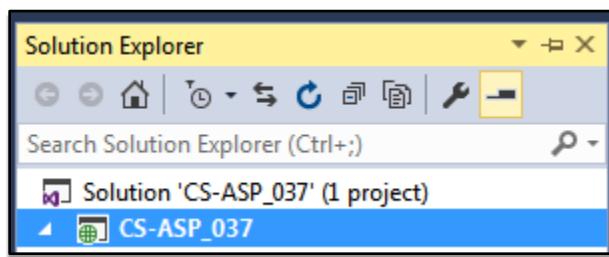
# Creating Class Files, Creating Cohesive Classes and Code Navigation

This lesson deals with building cohesive, conceptually sound class structures for readability and maintainability. Start off by creating an ASP.NET project with a *Default.aspx.cs* file.

## Step 1: Adding a Class File to the Project

---

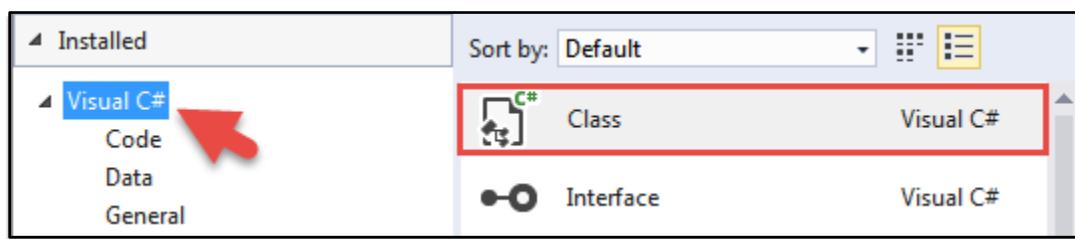
In the previous lesson, we learned how to create multiple classes that exist side-by-side within the same file, and under a common namespace outer container. While this can be useful for quickly testing out code and keeping it easy to reach, larger applications will make this impractical. In these cases, it is often preferable to separate classes into their own files. You can add a class to the project by right-clicking on it from within the Solution Explorer, and selecting from the context menu:



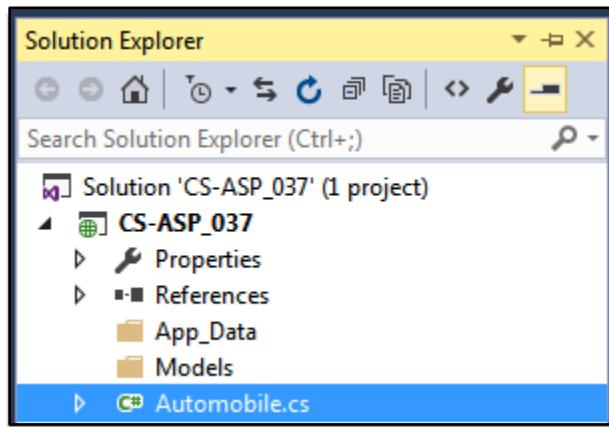
From the resulting context menu select:

Add > New Item...

And in the dialogue, choose the "Class" item under the "Visual C#" tab:



To help keep track of what classes are in your Class files, you would typically want to name the file the same as the class itself. In this case, we are going to create an Automobile class:



Open up this Class file and write out the following code for it, omitting the method implementation details for now:

```
class Automobile
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }
    public string Category { get; set; }
    public double Price { get; set; }
    public int DaysOnLot { get; set; }
    public string PromotionalOffers { get; set; }
    public double MarketValue { get; set; }
    public string PurchasedByName { get; set; }
    public string PurchasedByAddress { get; set; }
    public string PurchasedByPhone { get; set; }
    public string SoldBy { get; set; }
    public double TotalSaleAmount { get; set; }
    public int CarLotParkingSpace { get; set; }
    public bool HasBeenDetailed { get; set; }
    public string DetailedServiceHistory { get; set; }
```

```

    public void DetermineMarketValue() { }
    public void MoveCarOnLot() { }
    public void SendCarToDetailer() { }
    public void AddToServiceHistory() { }
    public void AddPromotionalOffer() { }
    public void DiscountCar() { }
    public void SellCar() { }
    public void PrintCarDetails() { }
    public void RetrieveCarFromDatabase() { }
    public void SaveChangesToDatabase() { }
}

```



Bob's Tip: it can be difficult for beginners to quickly glimpse the constituents of a large class and separate the properties from the methods, especially if a method has a return type. When scanning through the class, try to look for the parentheses, such as in `DetermineMarketValue()`, as a "dead giveaway" that the member is a method. Also, although not strictly enforced, the rule of thumb is that properties are at the top of the class, and below that are the methods.

## Step 2: Prelude to Understanding Accessibility Modifiers

---

And now, in the Default class, create an instance of the Automobile class and set its properties, along with calling some of its (empty) methods:

```

public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Automobile auto = new Automobile();
        auto.Make = "Toyota";
        auto.Model = "FJ Cruiser";
        auto.Year = 2014;

        auto.SaveChangesToDatabase();
        auto.PrintCarDetails();
        auto.SellCar();
    }
}

```

Notice that even though each of these classes are in their own separate files, they are able to "see" each other and that is because they share the same namespace. Also, note that even though the

Automobile class is not marked as "public" it is still available to other classes within the namespace. This is because in absence of an accessibility modifier, it *defaults* to accessibility level called "internal" (meaning *internally* accessible to this namespace only):

```
namespace CS_ASP_037
{
    public partial class Default...
}

namespace CS_ASP_037
{
    class Automobile...
}
```

## Step 3: Keeping Classes Lean

---

Going back to the Automobile class, notice how large it is (even without any real implementation details). When you consider how this class is structured (presumably in anticipation of fulfilling a need within a broader application) it may, at first, seem like the class is highly cohesive, and sticks to the single responsibility principle. However, just because all of these methods and properties have something to do with a car, doesn't mean they should all belong in the Automobile class. For example, this set of properties refer to a buyer's details so perhaps it's best to have these within a separate class:

```
public string PurchasedByName { get; set; }
public string PurchasedByAddress { get; set; }
public string PurchasedByPhone { get; set; }
```

Go through the remaining list of properties and methods and try to anticipate which items are best moved into other classes.



Bob's Tip: there is a rule of thumb that some software developers subscribe to that says every class should be no longer than the length of the screen its being viewed on (setting aside that screen and font sizes aren't uniform). In other words, you shouldn't have to scroll down to see the rest of the code in the class. While simplistic, the wisdom here again is to keep the principle of Single Responsibility in mind and break up a single large class into a bunch of smaller classes whenever possible.

Let's apply what we've learned about code refactoring from previous lessons to break up this single, monolithic, Automobile class into a series of smaller classes - each within their own file within the Solution Explorer:

```
▷ C# Automobile.cs
▷ C# AutomobileDisplay.cs
▷ C# AutomobilePersistence.cs
▷ C# CarLot.cs
▷ C# Customer.cs
▷ C# MaintenanceHistory.cs
▷ C# Promotion.cs
▷ C# Sale.cs
▷ C# Salesman.cs
```

Here is how each individual class now breaks down (again, the implementation details are not important, just focus on the way the classes are restructured):

```
public class Automobile
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }
    public string Category { get; set; }
    public double Price { get; set; }
    public int DaysOnLot { get; set; }
    public double MarketValue { get; set; }
    public int CarLotParkingSpace { get; set; }

    public void DetermineMarketValue() { }
}

public class AutomobileDisplay
{
    public void PrintCarDetails() { }
}

public class AutomobilePersistence
{

    public void RetrieveCarFromDatabase() { }
    public void SaveChangesToDatabase() { }
}

public class CarLot
{
    public void MoveCarOnLot() { }
}
```

```
public class Customer
{
    public string PurchasedByName { get; set; }
    public string PurchasedByAddress { get; set; }
    public string PurchasedByPhone { get; set; }
}

public class Maintenance
{
    public bool HasBeenDetailed { get; set; }
    public string DetailedServiceHistory { get; set; }

    public void SendCarToDetailer() { }
    public void AddToServiceHistory() { }
}

public class Promotion
{
    public string PromotionalOffers { get; set; }

    public void AddPromotionalOffer() { }
    public void DiscountCar() { }
}

public class Sale
{
    public Automobile Automobile { get; set; }
    public Salesman Salesman { get; set; }
    public Customer Customer { get; set; }
    public double TotalSaleAmount { get; set; }

    public void SellCar() { }
}

public class Salesman
{
    //something will eventually go here
}
```

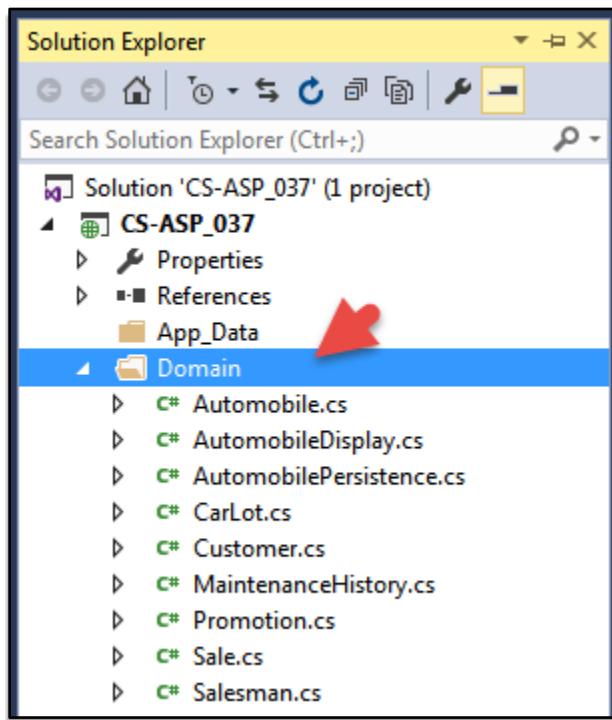
## Step 4: Organizing Classes in Folders

---

Another useful organizational technique is to group class files into folders relative to their problem domain. This is especially useful when a project grows larger and you need to locate a class in order to update it, and so on. Simply right click on the Solution Explorer and select:

Add > New Folder

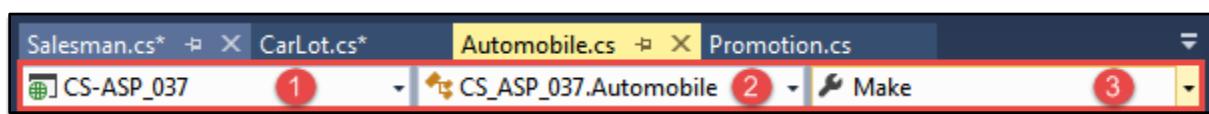
And after you name the folder you can select all of the class files you want to have added to this folder and drag/drop right into it. Here we labeled this folder as "Domain" to signify the *domain layer* of concern. However, as you move forward you can separate into other folders that represent other conceptual layers of your codebase:



## Step 5: Using Visual Studio to Navigate Classes

---

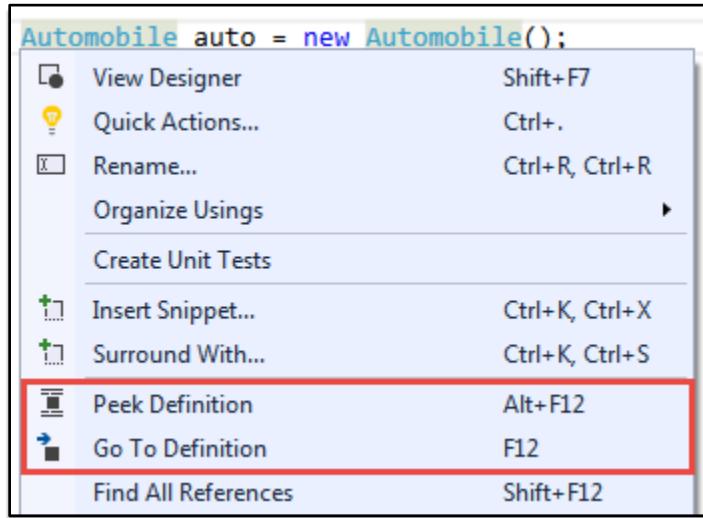
While on the topic of navigating within your project there are some tools Visual Studio provides to aid this task. There are list boxes at the top of the main window that let you move between various (1) projects, (2) classes, as well as their (3) fields and properties:



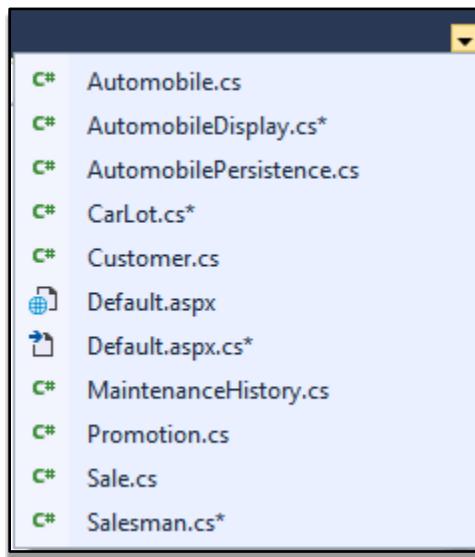
## Step 6: 'Go To Definition' to Review a Class

---

Another handy tool is the ability to navigate to a class definition by right-clicking wherever that class is referenced and selecting either "Peek Definition" (displays class inline) or "Go To Definition" (jumps to class):



You can also navigate to different classes by clicking on the downward arrow in the upper-right corner of the main window:



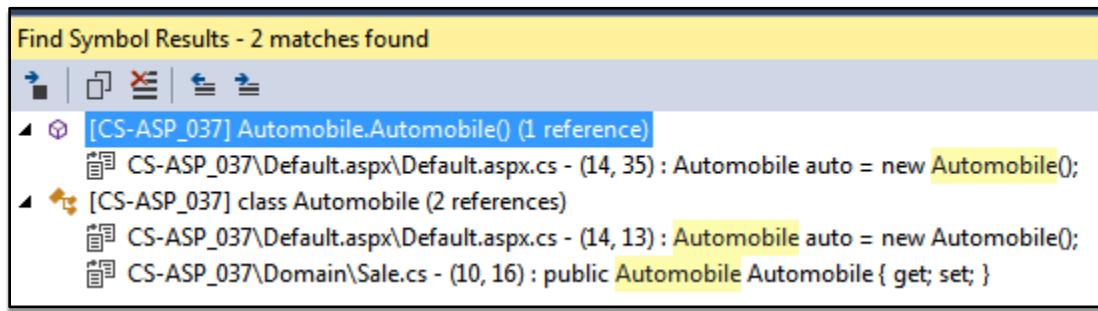
## Step 7: Find All References of a Class the Project

---

The flip-side of finding a class definition would be to find all instances of the class within your project. You can right-click on the class name once again and this time select "Find All References":



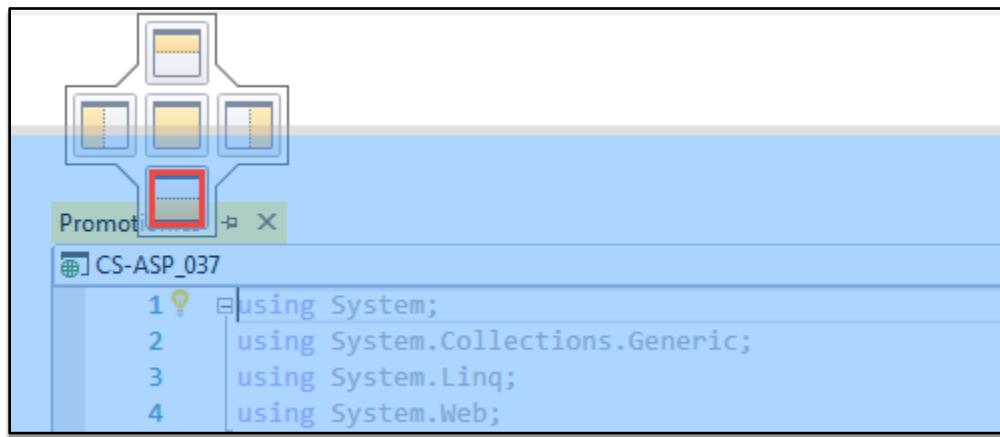
A window will pop-up describing where the class is being referenced (jump to the line of code with the reference by clicking on the line shown):

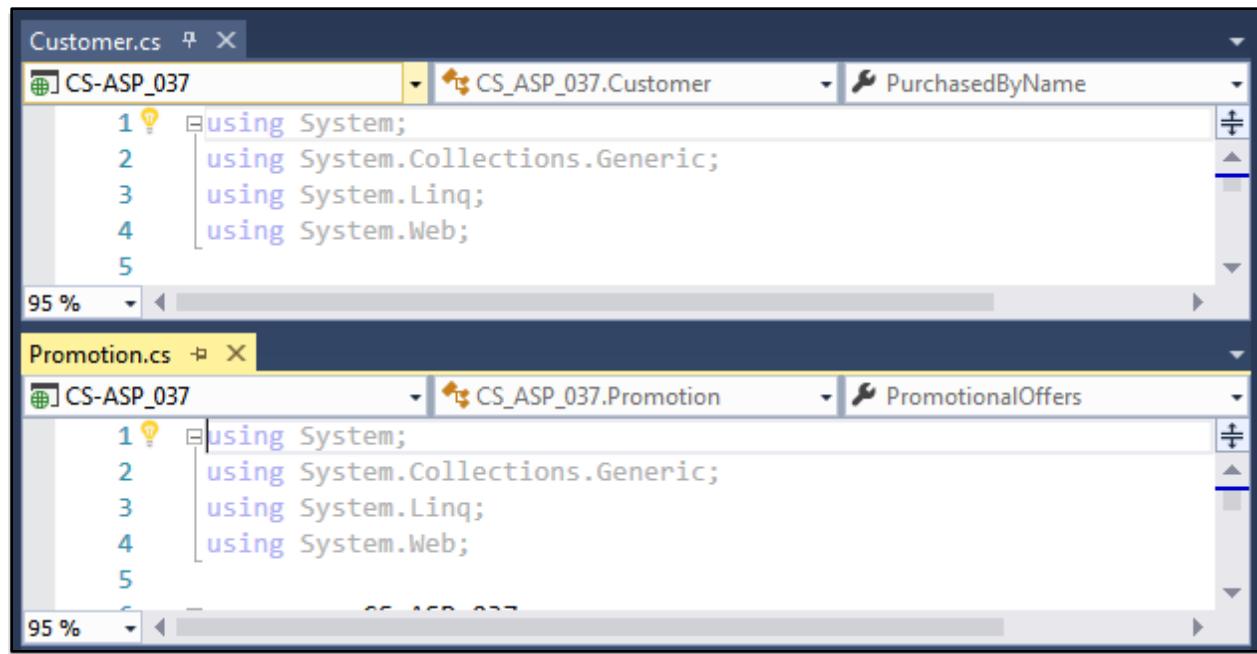


## Step 8: Docking Windows for Code Comparison

---

Sometimes you want to compare information between, and within, different classes. To satisfy this you can have multiple windows open by clicking/dragging tabs anywhere on the screen and hovering over the multiple docking options available (here it docks to a horizontal window below the existing main window):

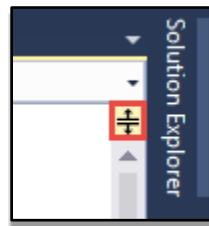




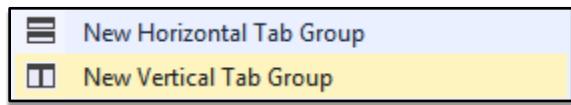
Bob's Tip: you can use this docking system with just about any window within Visual Studio in order to customize it to your needs.

## Step 9: Customizing Docked Windows

Also, be aware that this icon in the upper right-hand corner of the main window lets you split a window into two windows so that you can view different parts of the same class, each within its own window, for instance:



You can also right-click on any tab and select whether or not you want to move it another vertical/horizontal tab group:

A screenshot of the Microsoft Visual Studio code editor. Two tabs are open: 'Automobile.cs' and 'Promotion.cs'. Both tabs have their respective file names and line numbers at the top. The code editor interface shows syntax highlighting for C# code. The tabs are positioned side-by-side, indicating they are part of a horizontal tab group.

# Understanding Object References and Object Lifetime

In this lesson we're going to talk about the lifetime of objects - in terms of their references being kept in memory - and how the .NET Framework Runtime manages those memory allocations for you.

## Step 1: Create a New Project

---

To illustrate this point, create a new ASP.NET project called "CS-ASP\_038" and create a simple Car class with an object instance of it:

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public string Color { get; set; }
    public int Year { get; set; }
}

public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Car myCar = new Car();
    }
}
```

When that last line of code executes creating a new `Car()` instance, the .NET Framework creates a spot in the computer's memory large enough to hold the new instance of the car class. The computer's memory has addresses - much like how you have a home address – and these addresses are where the .NET Framework Runtime temporarily stores values, like objects or variables, during the lifetime of the variable or the object. You can summarize this .NET Framework memory allocation process as follows:

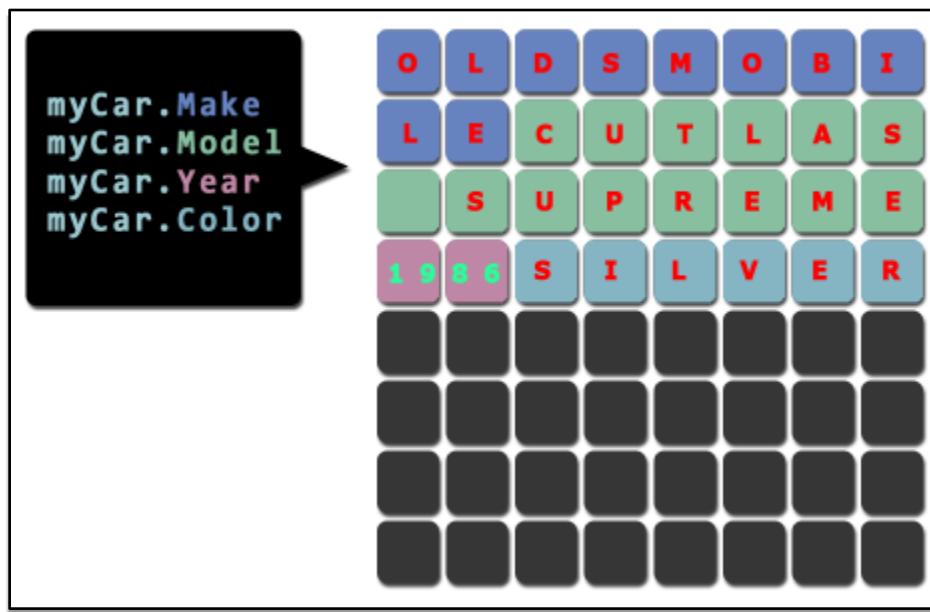
1. Creating a variable/object creates a place in memory large enough for the particular data type (in this case a class instance of Car).

2. The Framework keeps an address (or “pointer”) of where it put that new instance of car, and then it serves that address back to you - the programmer - so that you can get back to the information in memory whenever you need it (such as when referencing the variable somewhere else in code).

## Step 2: Understanding Memory Addresses and Object References

---

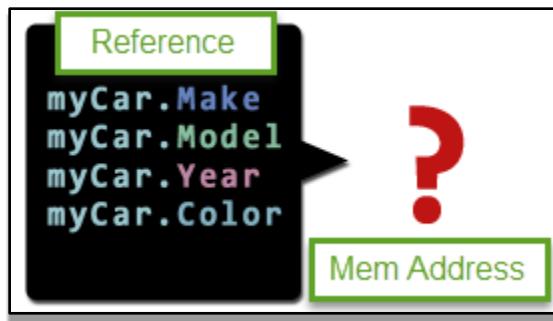
In the illustration below, you can think of the color-coding for each property as the memory address storing that property's value. You then use that address reference to look up that value whenever you get/set the value for the property in code:



Whenever you see the new keyword, you can take that to mean there is a new instance with its own particular location being created in memory. This might become clearer if you split up the declaration/assignment steps on their own line of code:

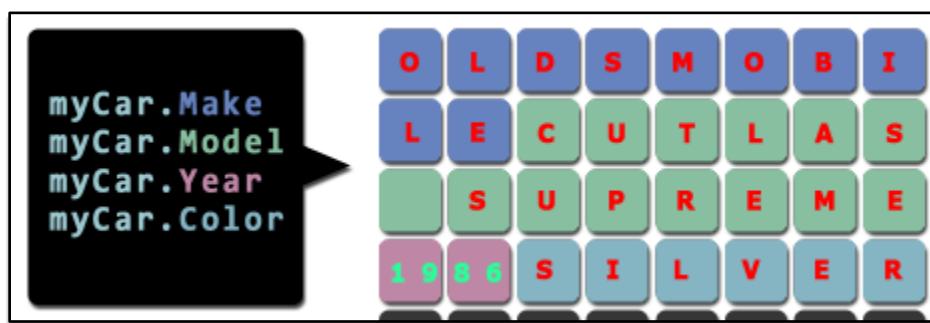
```
Car myCar; ①  
myCar = new Car(); ②
```

Here, the first line of code is creating (1) a reference in memory that is not yet pointing to any particular memory address storing particular values in memory (if given):



It's not until you use the new keyword that this reference (2) becomes set to a particular memory address. You can now "connect" to the memory reference - not *directly* the values in memory, but the *reference* to the values (if present) in memory – wherever you refer to it with the human-readable format provided in code:

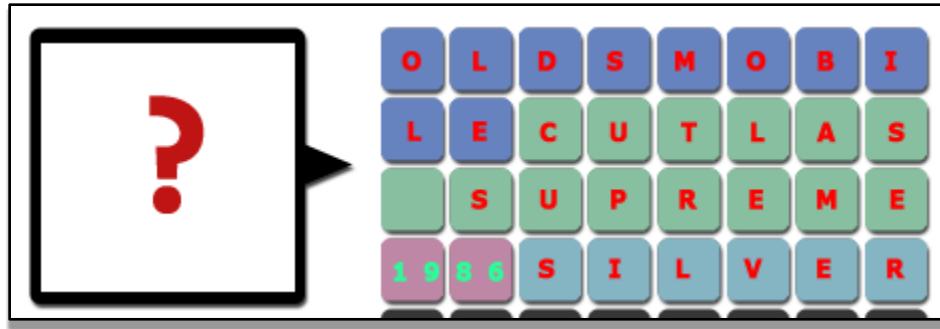
```
myCar.Make = "oldsmobile";
myCar.Model = "cutlas supreme";
myCar.Year = 1986;
myCar.Color = "silver";
```



### Step 3: Automated Memory Management via the Garbage Collector

---

One of the key features of C# is something called "Garbage Collection" which cleans up memory registers that no longer have references pointing to them. This is an automatic process which is very different from earlier languages like C or C++ that required programmers to keep very close tabs on each and every item in memory. References can be dropped whenever an object falls out of scope (such as an object declared local to a method and therefore staying alive only as long as that method is running) or with a specific line of code that "destroys" the object. In this illustration, the .NET Garbage Collector would mark all of these memory registers for deletion – freeing up that space - because the reference no longer exists:



## Step 4: Implications of Reference vs Value Storage Types

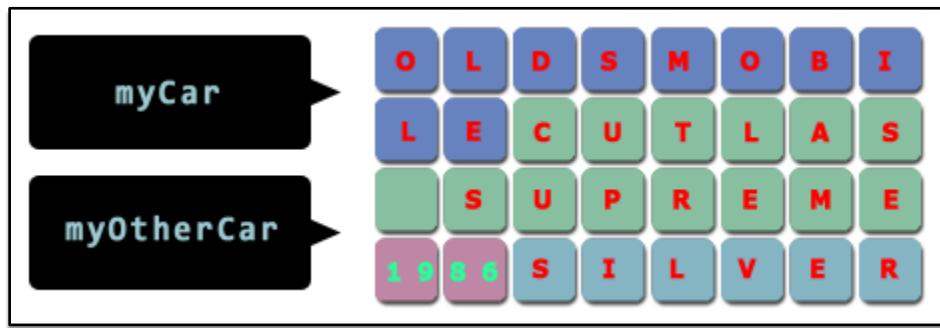
---

This reference/value relationship has a variety of interesting implications when writing code in C#. One such implication is when you assign one object instance to another, you are actually *copying the reference*, rather than copying a new set of memory registers:

```
Car myCar;
myCar = new Car();

Car myOtherCar = myCar;
```

Now, both myCar and myOtherCar point to the exact same memory registers:



This means that if you change the value for myCar, you will also be changing it for myOtherCar. Make, and vice versa. This is very different from how value types work, which copy values to their own unique memory registers:

```

Car myCar;
myCar = new Car();
myCar.Make = "Chrysler";

Car myOtherCar = myCar;
//myOtherCar.Make will ALSO be "Chrysler"

myOtherCar.Make = "Toyota";
//myCar.Make will ALSO be "Toyota"

```

## Step 5: Explicit De-Referencing Using the 'null' Keyword

---

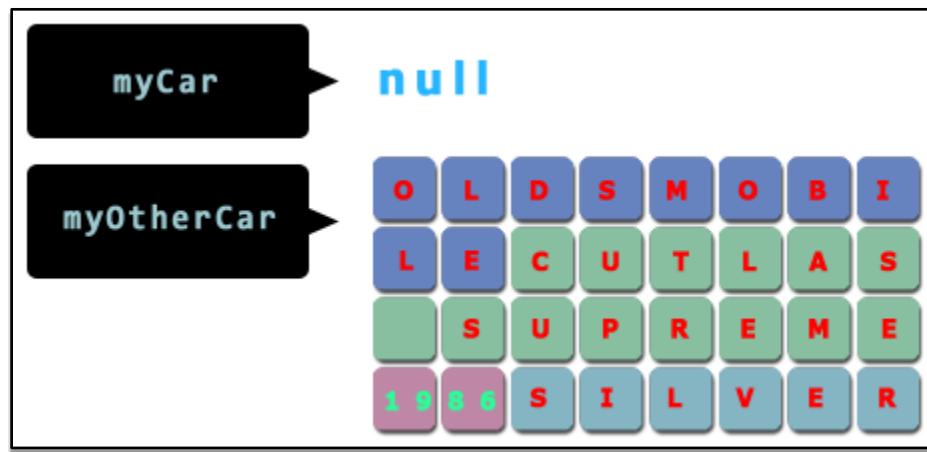
Also, if you de-reference one of these references (by setting it to null), the other reference *still points* to the memory registers. And since there is still a reference to those registers, the Garbage Collector keeps the memory intact unless `myOtherCar` also gets set to null:

```

Car myCar = new Car();
Car myOtherCar = myCar;

myCar = null;

```



It's important to note that object references are normally removed whenever the .NET Framework gets around to it. In some situations this is an indeterminate point in time. This can sometimes cause problems, especially when the object in memory is holding onto a system resource like a network connection or a file in the file system. Those are scenarios in which you would want to *force* the garbage collector to *immediately* do its clean-up process so that you can possibly use those resources for something else. This more deterministic approach to managing memory is somewhat more complicated than just setting all references to null – but it's worth keeping in mind as you move further down the path of your programming career.

039

# Understanding the .NET Framework and Compilation

We've touched upon several aspects of the .NET Framework, along with C# as a language, however we haven't looked at how these two relate to one another. In most cases, it's not important to think about all of this when you're programming. Rather, the point of understanding these details, on a basic level, is to have a working knowledge of how the various aspects of C# and the .NET Framework operate.

At the basis, C# is a programming language that is the foundation for the .NET Framework, which in turn is a library of functionality (pre-built Class Libraries) that you can incorporate into your C# applications. For example, whenever we're working with ASP.NET-related classes, we're incorporating the `System.Web` namespace containing a plethora of classes related to all types of ASP.NET functionality. You can find these Framework Class Libraries by navigating to the folder that stores the associated `.dll` files:

PC > Local Disk (C:) > Windows > Microsoft.NET > Framework > v4.0.30319	
Name	Size
 System.Web.dll	5,222 KB

One of the reasons for why the Class Libraries are separated into individual `.dll` files is to reduce performance load, allowing you to choose only the Libraries you need for a particular task.

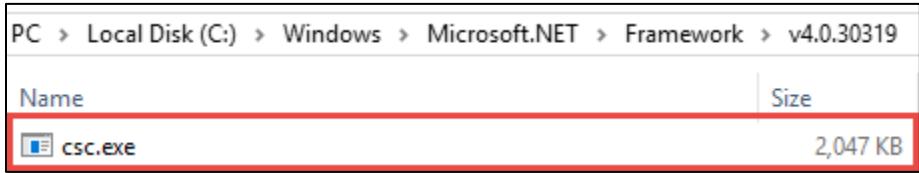


Bob's Tip: Note that the `System.Web.dll` Library is one of the largest in the Framework. It's been a contention amongst some developers that this is too big for any single Library and should ideally be broken up into sub-libraries. While 5,222KB might not seem like a lot, loading this every time a page runs on a remote Server could represent a significant hit to its bandwidth.

In addition to the Framework Class Library, the .NET Framework also contains the *runtime* environment known as the *Common Language Runtime* (CLR). The CLR operates as a protective bubble – kind of like a Virtual Machine – that wraps around your high-level application code and executes it down at a lower machine-readable level. As the name suggests, this allows for different languages supported by the CLR to essentially become one-and-the-same at the lower computational level. The runtime functionality of

the CLR manages low-level interfacing with hardware and memory states, freeing up your resources as a programmer to focus on solving problems having to do with the application's main purpose.

As a side-note, you might be interested in creating your own custom Class Libraries – boiling-down your code into a *.dll* Assembly and allowing you to reuse it across various projects. You can compile your project into a Class Library using the *csc.exe* Compiler found in the following folder:

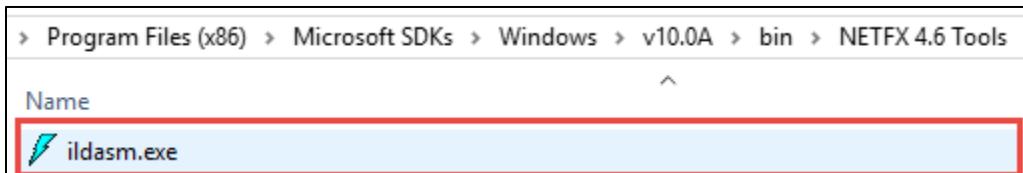


This Compiler creates an *.exe* or *.dll* .NET Assembly which is comprised of code that's been compiled-down to what's called an *Intermediate Language* (IL). The CLR is then able to run this IL – within the "protected bubble" – when the application is started.

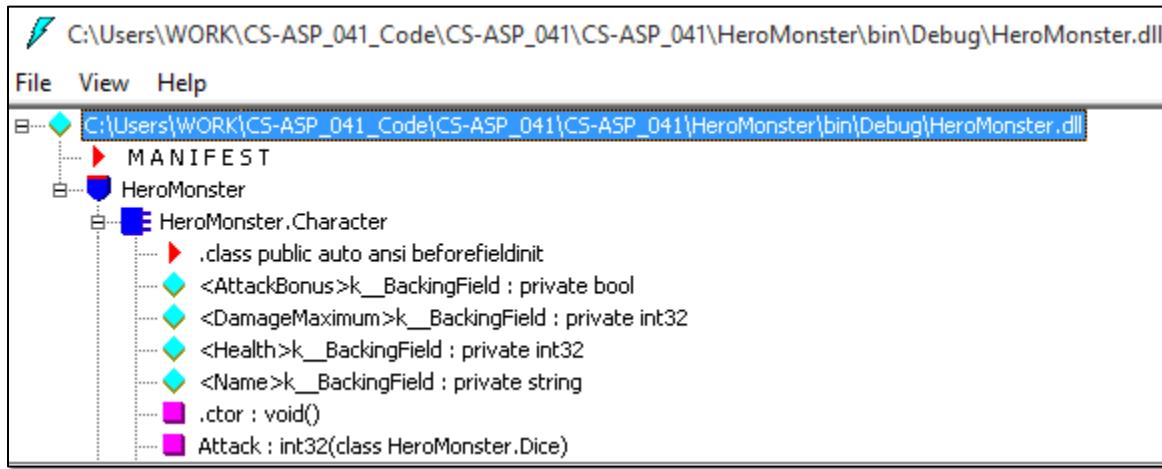


Bob's Tip: The main difference between *.exe* and *.dll* Assemblies is that an *.exe* has a programmatic entry-point to kick-off and run the code when accessed, whereas a *.dll* just stores code so that it can be referenced elsewhere in some other code.

It might be helpful to take a look at the *ildasm.exe* tool, which allows you to *disassemble* your Assembly back into a somewhat human-readable format:



Running *HeroMonster.dll* through the *ildasm.exe* tool, we can see how the code was constructed even after it's been compiled down into an Assembly:



Looking closer at the *Attack()* method, we get to peak at the resulting *Intermediate Language (IL)* that the original code was compiled down to:

```
HeroMonster.Character::Attack : int32(class HeroMonster.Dice)
Find Find Next
.method public hidebysig instance int32  Attack(class HeroMonster.Dice dice) cil
{
    // Code size      25 (0x19)
    .maxstack  2
    .locals init ([0] int32 CS$1$0000)
    IL_0000:  nop
    IL_0001:  ldarg.1
    IL_0002:  ldarg.0
    IL_0003:  call     instance int32 HeroMonster.Character::get_DamageMaximum()
    IL_0008:  callvirt instance void HeroMonster.Dice::set_Sides(int32)
    IL_000d:  nop
    IL_000e:  ldarg.1
    IL_000f:  callvirt instance int32 HeroMonster.Dice::Roll()
    IL_0014:  stloc.0
    IL_0015:  br.s    IL_0017
    IL_0017:  ldloc.0
    IL_0018:  ret
} // end of method Character::Attack
```

# Namespaces and Using Directives

The last few lessons have been covering a lot of details involving classes because they're so integral to the .NET Framework. In fact, just about everything you do in .NET involves a class in some way. The .NET Framework Class Library itself is really just a collection of classes with pre-built functionality that you can utilize in your projects. This library was built – by Microsoft – to make your job easier, and so that you don't have to "re-invent the wheel" when creating software. The aim of this lesson is to help you understand how the .NET Framework Class Library was put together and how you can get the most out of it. This should help you better understand how the pre-built code is organized and how to access code that's not part of your project by default. Much of this hinges on a deeply related coding topic that you have been seeing on the periphery throughout these lessons and that is the topic of namespaces along with the using directive.

## Step 1: Understanding How Namespaces and Using Directives Relate

---

To understand how namespaces work, let's briefly look back at the concepts of scope and locality. Consider having two different properties in your code that, logically, should have identical names. This is not possible when those properties are declared in the same scope (the class, in this case). However, it is perfectly possible to share the same name if these properties were in separate classes:

```
namespace CS_ASP_040
{
    public class Car
    {
        public string Make { get; set; }
    }

    public class Truck
    {
        public string Make { get; set; }
    }
}
```

The compiler easily distinguishes between these properties because they each belong to very different scopes. By the same token, a namespace is yet another scope that simply acts as a container for a group of classes. So if you have two classes with the same name, it is forbidden if they are both part of the same scope since the compiler has no way of distinguishing between them:

```

namespace CS_040
{
    public class Car
    {
        public string Make { get; set; }
    }

    public class Car
    {
        public string Make { get; set; }
    }
}

```

However, if each of these classes belong in different namespaces, they belong within different scopes entirely:

```

namespace CS_040
{
    public class Car
    {
        public string Make { get; set; }
    }
}

namespace SomeOtherGroupOfClasses
{
    public class Car
    {
        public string Make { get; set; }
    }
}

```

## Step 2: Referencing With and Without Using Directives

---

It becomes more obvious how the compiler can tell these classes apart when you reference them using their “full names,” that include the namespace they respectively belong to:

```

CS_040.Car car1 = new CS_040.Car();
SomeOtherGroupOfClasses.Car car2 = new SomeOtherGroupOfClasses.Car();

```

The CS\_040 namespace here is color-coded different from the other namespace because it is being used within the current namespace:

```
namespace CS_040
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            CS_040.Car car1 = new CS_040.Car();
            SomeOtherGroupOfClasses.Car car2 = new SomeOtherGroupOfClasses.Car();
        }
    }
}
```

We don't have to reference the namespace "full name" for the first Car object as the compiler intuits that we are referring to the Car class from within the current namespace "CS\_040". However, the SomeOtherGroupOfClasses.Car object is different because it's referencing a class from an *outside* namespace. In this case, you have to either reference the namespace on each line of code that utilizes this class, or as a shorthand you can add this outside namespace to the current one through a using directive:

---

```
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using SomeOtherGroupOfClasses;
```

## Step 3: Resolving Class Name Ambiguities

---

Under normal circumstances this would adequately solve the problem of avoiding typing out the "full name" of the Class, but in this particular example we still have two Car classes without a way for the compiler to distinguish between them. In the example below, we know that car2 is supposed to be of the type found in SomeOtherGroupOfClasses, but the compiler will think it's referring to the type found in the current CS\_040 namespace even with the using directive:

```
Car car1 = new Car(); //from the (current) CS_040 namespace
Car car2 = new Car(); //still needs reference to SomeOtherGroupOfClasses!
```



Bob's Tip: you could look at as though a namespace represents the *last name*, whereas the class represents the *first name*, in the same way that *your* last name differentiates you from other people who share your first name. For example, if someone were to state "Bob likes coffee" it would be difficult to ascertain who "Bob" might be until we append the last name to refer to a specific person: "Bob Tabor likes coffee."

## Step 4: Using Namespaces to Import Code Libraries

---

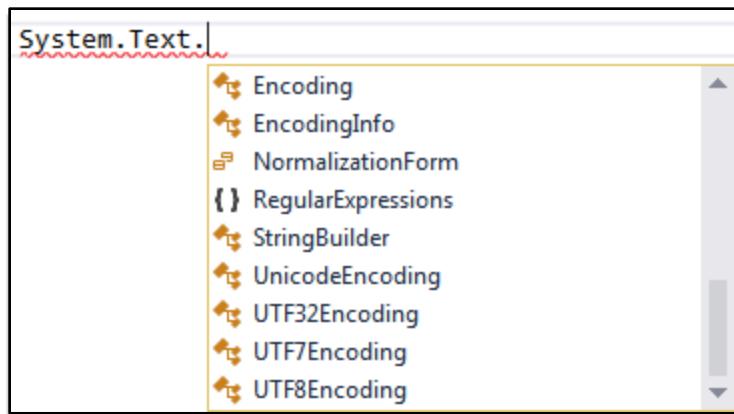
It's important to note that while a namespace can just be seen as a way of grouping classes into different areas in your codebase, the biggest benefit of a namespace is to be able to import into your project (and make reference to) a library of code that is outside of it. We saw this earlier when we wanted to access the `StringBuilder` class that originates from the `System.Text` namespace:

```
using System.Text;  
  
namespace CS_ASP_040  
{  
    public partial class Default : System.Web.UI.Page  
    {  
        protected void Page_Load(object sender, EventArgs e)  
        {  
            StringBuilder sb = new StringBuilder();  
        }  
    }  
}
```

If we didn't include the `using` directive to this namespace, we would have had to use the full name for this reference, making a less clean looking result:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder();
```

It should be fairly clear that the `System.Text` namespace contains within it a collection of classes – one of which is the `StringBuilder` class. You can see, with Intellisense, all of the constituent classes within this namespace:



## Step 5: Visual Studio Defaults to Importing Common Namespaces

---

You will typically only want to import libraries that you intend to use. However, projects that are based on a template (such as the ASP.NET projects we have been working with) will anticipate common libraries that are useful – or in some cases necessary – by adding them ahead of time as using directives within each file that references them:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Text;
```

## Step 6: Understanding Nested Namespaces

---

Another aspect of namespaces is that they can be nested within each other. In the example above, the Text namespace is nested within the System namespace. You can create nested classes just as you would expect:

```
namespace Outer
{
    public class SomeClass1
    {

    }

    namespace Inner
    {
        public class SomeClass2
        {

        }
    }
}
```

And then you can reference them using the dot accessor in the way that you would expect:

```
Outer.SomeClass1 outerNamespaceClass = new Outer.SomeClass1();
Outer.Inner.SomeClass2 innerNamespaceClass = new Outer.Inner.SomeClass2();
```

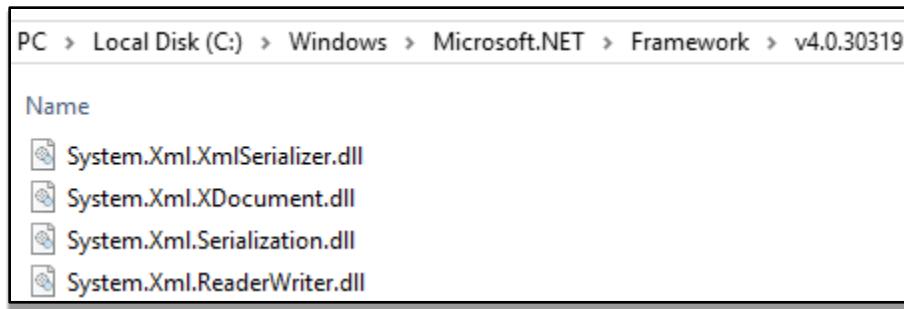


Bob's Tip: because classes can share the same name - but belong to different namespace - be careful when researching a class that it is from the namespace you intend on using. When in doubt, enter the "full name" of the class including its namespace in the search engine.

041

# Creating Class Libraries and Adding References to Assemblies

As we saw in a previous lesson, the .NET Framework Class Library is split up across several different Assemblies, each within its own *.dll* file in the Framework library folder:

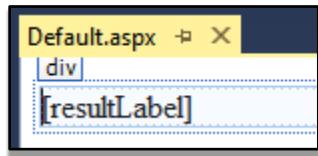


Assemblies are split into separate files to reduce load on resources, maximize efficiency so that we load up only what is needed to complete the task at hand, and to make code portable. This lesson will demonstrate how to create Class Libraries off of your own custom project code, and how to add references to resulting *.dll* Assemblies.

## Step 1: Create a New Project

---

To begin this lesson, create an ASP.NET project called "CS-ASP\_041" and in the *Default.aspx* add a single result Label :

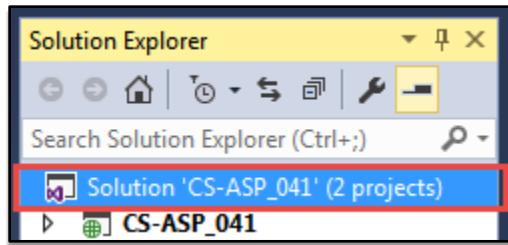


## Step 2: Add another Project to the Solution as a Class Library

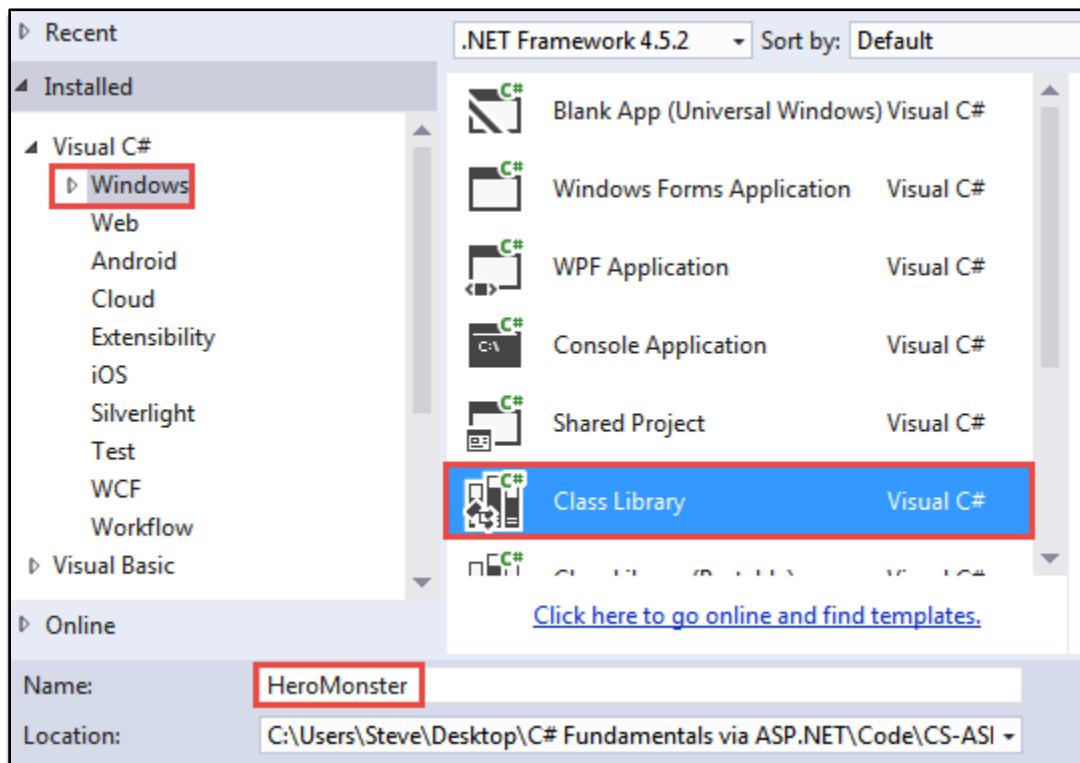
---

For this particular Solution, we will want to add *another* project to it. To do this right-click on the Solution and from the menu choose:

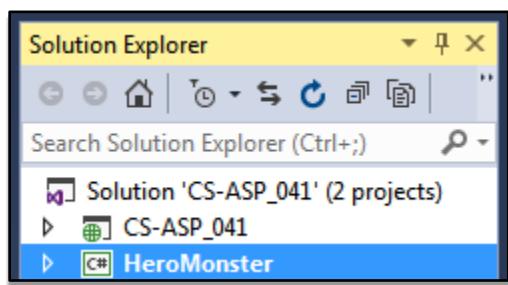
Add > New Project...



From the list of templates select “Windows” and “Class Library,” and name the project “HeroMonster”:



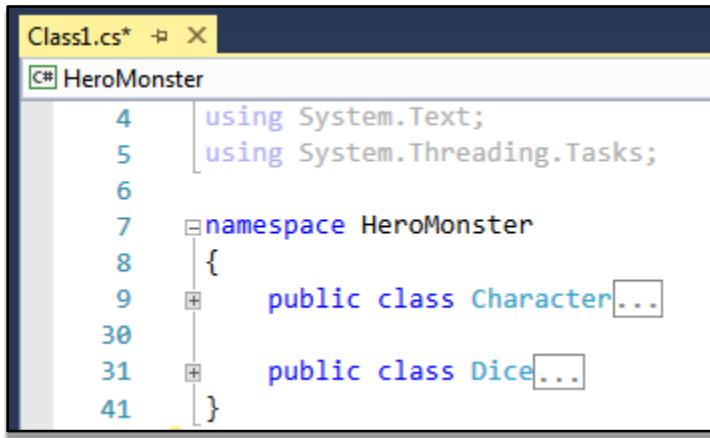
And now you can access both projects within the Solution Explorer. If you want, you can even build the projects separately by right-clicking on one and choosing from the menu “Build”:



## Step 3: Create Custom Classes in Class1.cs

---

The “HeroMonster” project will come with a default class file called “Class1.cs” and in this file we will create two separate classes called Character and Dice:



The screenshot shows a code editor window titled "Class1.cs". The code is as follows:

```
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace HeroMonster
8  {
9      public class Character{...}
10
11     public class Dice{...}
12 }
```

Write the following code for each class:

```
public class Character
{
    public string Name { get; set; }
    public int Health { get; set; }
    public int DamageMaximum { get; set; }
    public bool AttackBonus { get; set; }

    public int Attack(Dice dice)
    {
        dice.Sides = this.DamageMaximum;
        return dice.Roll();
    }

    public void Defend(int damage)
    {
        this.Health -= damage;
    }
}
```

```

public class Dice
{
    public int Sides { get; set; }

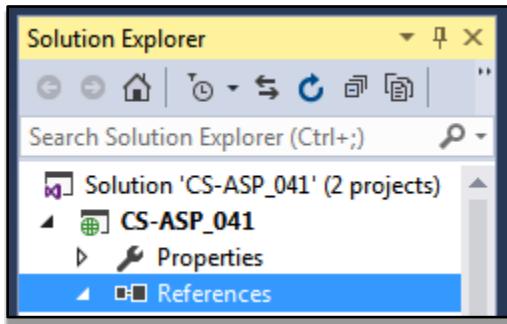
    Random random = new Random();
    public int Roll()
    {
        return random.Next(this.Sides);
    }
}

```

## Step 4: Add an Outside Assembly to a Project via 'References'

---

You can now add the "HeroMonster" project to the CS-ASP\_041 project by right-clicking it's "References" within the Solution Explorer:



And then selecting it under the "Projects" tab:



## Step 4: Incorporating the Assembly in Code via Using Directives

---

Now that we have a reference to this assembly in the CS-ASP\_041 project, we can add it to our class files via a `using` directive. Note that this directive actually *loads* the code library from HeroMonster, whereas the assembly reference simply allows it to be *available* to our project. With the directive added, you can then reference a public class, such as `Character`, from that library:

```
using HeroMonster;   
namespace CS_ASP_041  
{  
    public partial class Default : System.Web.UI.Page  
    {  
        protected void Page_Load(object sender, EventArgs e)  
        {  
            Character hero = new Character();  
        }   
    }   
}
```

## Step 5: Porting Over the Battle Game Code

---

You can now fill out `Page_Load()` with the code we used in the previous battle simulation game project:

```
Character hero = new Character();  
hero.Name = "Hero";  
hero.Health = 35;  
hero.DamageMaximum = 20;  
hero.AttackBonus = false;  
  
Character monster = new Character();  
monster.Name = "Monster";  
monster.Health = 21;  
monster.DamageMaximum = 25;  
monster.AttackBonus = true;  
  
Dice dice = new Dice();  
  
// Bonus
```

```

// Bonus
if (hero.AttackBonus)
    monster.Defend(hero.Attack(dice));
if (monster.AttackBonus)
    hero.Defend(monster.Attack(dice));

while (hero.Health > 0 && monster.Health > 0)
{
    monster.Defend(hero.Attack(dice));
    hero.Defend(monster.Attack(dice));
    printStats(hero);
    printStats(monster);
}
displayResult(hero, monster);

```

We will also need to port over, into this Default class, the following helper methods:

```

private void displayResult(Character opponent1, Character opponent2)
{
    if (opponent1.Health <= 0 && opponent2.Health <= 0)
        resultLabel.Text += String.Format("<p>Both {0} and {1} died.", 
            opponent1.Name, opponent2.Name);
    else if (opponent1.Health <= 0)
        resultLabel.Text += String.Format("<p>{0} defeats {1}</p>", 
            opponent2.Name, opponent1.Name);
    else
        resultLabel.Text += String.Format("<p>{0} defeats {1}</p>", 
            opponent1.Name, opponent2.Name);
}

private void printStats(Character character)
{
    resultLabel.Text += String.Format("<p>Name: {0} - Health: {1}" +
        " - DamageMaximum: {2} - AttackBonus: {3}</p>",
        character.Name,
        character.Health,
        character.DamageMaximum.ToString(),
        character.AttackBonus.ToString());
}

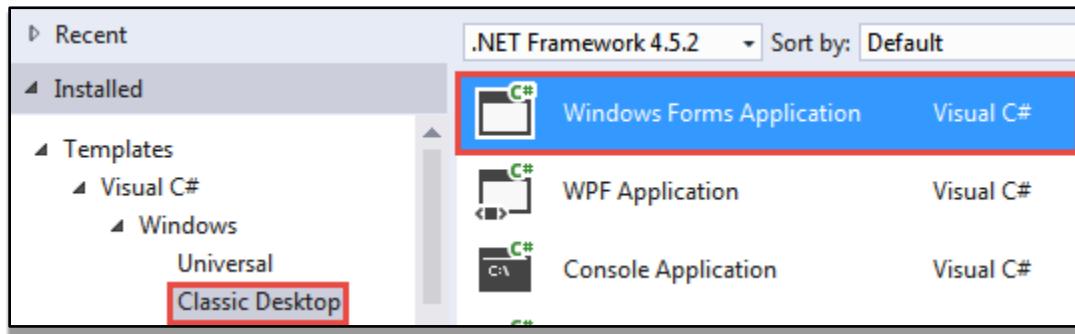
```

This is an identical application to what we built before, with the difference being that we partitioned out some “Domain-specific” code – the Character and the Dice classes – factoring them out into their own Class Libraries. This is noteworthy as it allows the classes to be reused in another project, which we can also accomplish by loading the .dll created in the project’s bin folder when we ran the application.

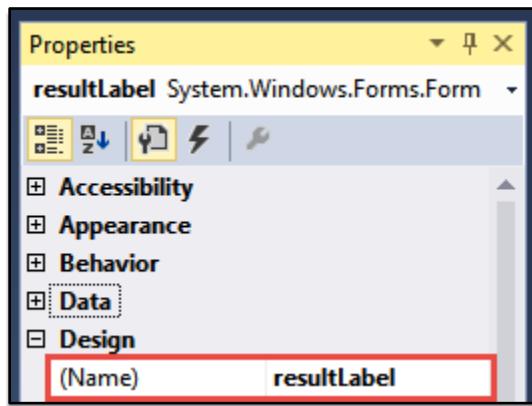
## Step 6: Adding HeroMonster.dll Assembly to a New Project

---

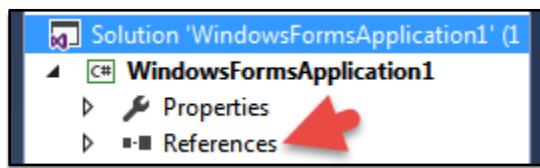
Open up a new instance of Visual Studio and create a new Project based off of a Windows Forms Application template. Even though we're not going to deal with Windows Forms in-depth, this will demonstrate how easy it is to port over our custom Class Libraries into a different project environment, requiring very little changes:



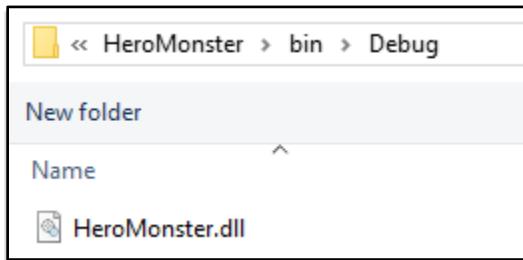
Once the Windows Form project is open, turn to the Properties Window (F4) and simply rename the project's Design Name to "resultLabel":



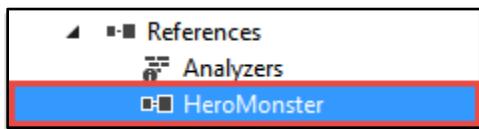
Next, right-click on "References" underneath the main project within the Solution Explorer, and select "Add Reference":



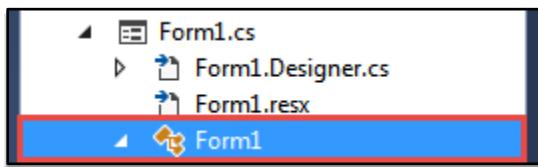
On the next screen you will want to click “Browse” and navigate to the folder for the project we worked on at the start of this lesson. Within that folder will be a bin folder that holds the *HeroMonster.dll* that was automatically built when we ran the application:



Add that *.dll* Assembly to the project and you will see it under “References” in the Solution Explorer:



Double-click on “Form1” in the Solution Explorer to open up the code for the Form Application:



Add the HeroMonster library as a using declaritive at the top of the script:

```
using HeroMonster;   
namespace WindowsFormsApplication1  
{  
    public partial class Form1 : Form  
    {  
        public Form1()  
        {  
            InitializeComponent();  
        }  
    }  
}
```

A screenshot of the Visual Studio code editor showing the beginning of a C# script. The 'using' keyword is present at the top, followed by the 'namespace' declaration. A red arrow points to the 'using' keyword. The rest of the script defines a class 'Form1' that inherits from 'Form'.

You can then copy all of the `Page_Load()`, `displayResult()` and `printStats()` code from the `Default.aspx.cs` in the original project – that depends on the `HeroMonster` class library code – and paste it into the `Form1` class in the current Forms project (partially represented below, substituting `Page_Load()` with `Form1_Load()`):

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        Character hero = new Character();
        hero.Name = "Hero";
        hero.Health = 35;
        hero.DamageMaximum = 20;
        hero.AttackBonus = false;

        Character monster = new Character();
        monster.Name = "Monster";
        monster.Health = 21;
        monster.DamageMaximum = 25;
        monster.AttackBonus = true;

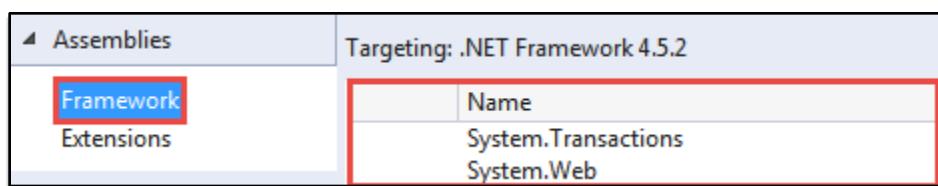
        Dice dice = new Dice();
    }
}
```

If you build the project and everything is set up correctly, you should notice that there are no errors. Of course, some things will have to be changed considering that the `HeroMonster` code was for a Web Application and contains HTML. However, the main point here is to see how you can partition your code into individual projects, Class Libraries and `.dll` Assemblies in order to gain portability with the code you write.

Note that you can add references to existing Framework Assemblies using the same process. This is useful whenever you have particular functionality you want to use in your project and is already available in the Framework. To add an existing Framework Assembly, once again go to:

Solution Explorer > References > Add Reference > Assemblies > Framework

And select the Assembly relevant to your project in the right-hand pane:



042

# Accessibility Modifiers, Fields and Properties

This lesson will cover the topic of accessibility modifiers as well as private fields and public properties. Access modifiers are prefixed to classes, class-level variables, and methods, in order to determine the “visibility” of these elements throughout your codebase. There are five different access modifiers, however, you will mainly be concerned with just the `public` and `private` modifiers:

Access Modifier	Description
<code>public</code>	The type or member can be accessed by any other code in the same assembly or another assembly that references it
<code>private</code>	The type or member can be accessed only by code in the same class or struct.
<code>protected</code>	The type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.
<code>internal</code>	The type or member can be accessed by any code in the same assembly, but not from another assembly.
<code>protected internal</code>	The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly. Access from another assembly must take place within a class declaration that derives from the class in which the protected internal element is declared, and it must take place through an instance of the derived class type.

You can learn more about access modifiers by going to the official MSDN article describing them:

<http://v.gd/access>



Bob's Tip: don't get confused by the word "struct." A struct is a special kind of class that looks just like an ordinary class, but with a technical twist. You won't have to worry about structs very much, in this series of lessons, so wherever you see it referenced, you can take it to *generally* mean a class.

## Step 1: Create a New Project

---

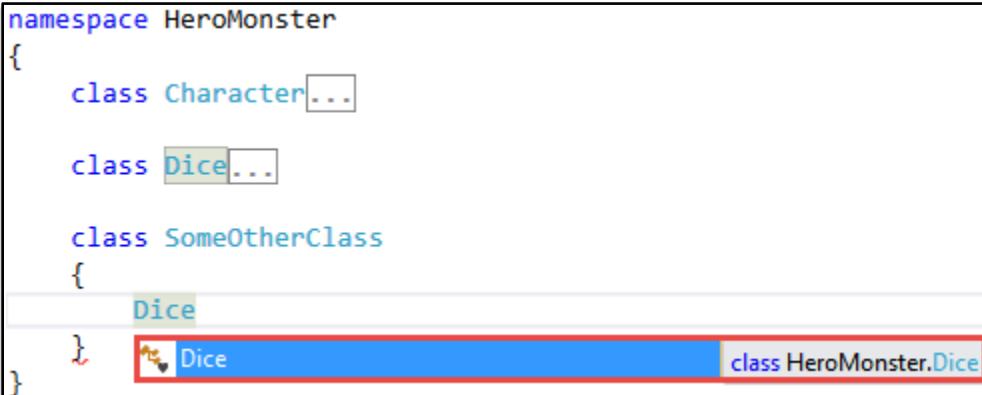
To illustrate the difference between these different modifiers, let's go back to where we left off in the previous lesson and save it to a new project called "CS-ASP\_042." You will recall that we made the Character and Dice classes, both from an outside assembly/namespace, prefixed as public:

```
namespace HeroMonster
{
    public class Character...
    public class Dice...
}
```

## Step 2: Default Protection Levels

---

If you were to remove these modifiers, the classes would default to the internal protection level. This means that it's visible by other classes within the same namespace/assembly (as in "internal to this assembly"):



```
namespace HeroMonster
{
    class Character...
    class Dice...
    class SomeOtherClass
    {
        Dice
    }
}
```

However, since the CS-ASP\_042 assembly is an *external* assembly, the Default class can no longer "see" these outside classes:

```
namespace CS-ASP_042
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            Character hero = new Character();
```

## Step 3: Understanding Public vs Private Protection Levels

---

By setting a class – or a class member – to “public” you are allowing it to be visible across all classes and assemblies in your project. It is the most *open* access modifier possible. By contrast, private is the *most closed* modifier possible, which means that whatever is prefixed as private is visible only within its containing class/context:

```
namespace HeroMonster
{
    public class SomeClass
    {
        private int num1;
        public int num2;
    }
}
```

Here, num1 is private and therefore only visible to another member within the same class that wants to reference it:

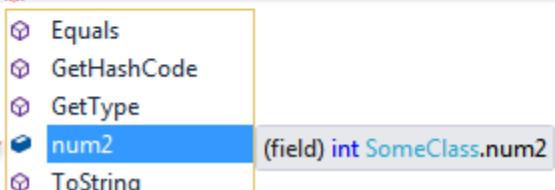
```
public class SomeClass
{
    private int num1;
    public int num2;

    void SomeMethod()
    {
        num1
    }
}
```

A screenshot of a code editor showing a tooltip for the variable 'num1'. The tooltip is a blue box containing the text '(field) int SomeClass.num1' with a small icon of a gear and a person next to it.

However, since it is private to the context in which it was declared it won't be visible outside of this context (in another class, for example):

```
namespace CS_ASP_042
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            SomeClass test = new SomeClass();
            test.~
        }
    }
}
```



The screenshot shows an IntelliSense dropdown menu for the variable 'test'. The menu lists several members of the 'SomeClass' type, including 'Equals', 'GetHashCode', 'GetType', 'ToString', and the private field 'num2'. The 'num2' entry is highlighted with a blue selection bar and has a tooltip '(field) int SomeClass.num2'.

As you see, in this example, Intellisense doesn't even show the element that is private to the SomeClass context. That alone is a common reason to keep elements private. Otherwise, if everything was public, a codebase would quickly be overwhelmed with unnecessary options that have no practical use and would only serve the potential for confusion and errors. Furthermore by keeping internal implementation private where possible, you reduce inter-dependencies in a codebase. This de-coupling of code dependencies further reduces complexity and errors. With that in mind, access modifiers are a powerful way of enforcing a key Object-Oriented-Programming tenet called "encapsulation" that encompasses the concept of abstraction and hiding implementation details.



Bob's Tip: while outside the scope of this lesson, the concept of *coupling* in Object-Oriented Programming is an important one. The basic idea is that, whenever a class "knows" a lot about another class, and depends on it to perform some of its own functions, the classes become inextricably linked or coupled together. So, when one class has to change - in order to deal with some update, for example – the other class may also change. The problem becomes especially difficult to manage when a variety of other classes *also* have some sort of dependency on the state of the changed classes. This can lead to a highly *unmaintainable* codebase. You can imagine how deep of a problem this represents in software development where it's common for developers to throw out unmaintainable code and simply start from scratch.

## Step 4: A List of Access Modifiers

---

Here is a list of the default access modifiers for the most common code elements (“default” here means whenever the access modifier is omitted):

Element	Default Accessibility
namespace	public (necessary)
class	internal
property/field	private
method	private



Bob's Tip: there is one other modifier that we haven't gone over yet and that is “protected.” This modifier sits somewhere between public and private. It grants visibility, across all assemblies (unless the “internal” modifier is added), to any class which *shares an inheritance hierarchy*. Inheritance is a topic that we will be looking at later, however, just keep this in mind for now.

## Step 5: Understanding Fields vs Properties

---

Up to this point we have been talking about class members as, either, variables called “properties” or executable code blocks called “methods”. However, there is another type of common class member, and that is a field. Fields are closer to what you've come to think of as variables than properties.

Properties and fields both hold values, or object references, that can be read from, or written to. However, properties have their own code block (squiggly brackets) that can *handle specific implementation* details, somewhat similar to a method. We have only seen the simplest version of this in what is called auto-implemented properties, where the get and set code is simplified, and hidden away. Here, the `Sides` class-level variable is an auto-implemented property and below it is the `random` class level variable that is an ordinary field:

```
public class Dice
{
    public int Sides { get; set; }
    private Random random = new Random();

    public int Roll()
    {
        return random.Next(this.Sides);
    }
}
```

Here the field is prefixed as `private` because we have no intention of letting outside classes access it directly. However, the `Roll()` method can still access it because, both, the field and the method belong to the same class. The `Roll()` method, meanwhile, is publicly visible to all classes. The result of this is that outside access to the `random` field is *indirect*; it is hidden behind the implementation details of the `Roll()` method. This kind of implementation – a private field “hidden” behind a public method that implements it – is very similar to how properties work. An auto-implemented property is just a shorthand for having a `private` field hidden behind the implementation of a `public` property. Here is what is actually going on behind the simple `Sides` get and set:

- (1) A private field is created.
- (2) A public property is created.
- (3) The property can read (get) whatever value is held in the backing field.
- (4) The property can write (set) to the backing field whatever value is being assigned to it elsewhere in code (“implied by value”).

```
public class Dice
{
    1 private int sides;

    2 public int Sides
    {
        get
        {
            3
            return sides;
        }

        set
        {
            4
            sides = value;
        }
    }
}
```



Bob's Tip: to get the code-snippet shortcut for a *full* property (along with its backing field) type “propfull” and then hit the tab key twice.

Here in the `Character` class we’re attempting to set the `Sides` property (which actually sets the value to the private `sides` field) and that works fine because, currently, the property’s `set` is `public`:

```
public class Character
{
    public string Name { get; set; }
    public int Health { get; set; }
    public int DamageMaximum { get; set; }
    public bool AttackBonus { get; set; }

    public int Attack(Dice dice)
    {
        dice.Sides = this.DamageMaximum;
        return dice.Roll();
    }

    public void Defend(int damage)...
}
```

## Step 6: Access Modifiers on Property 'get' and 'set'

---

However, you can modify the property so that it can be read from (get) publicly, but written to (set) privately:

```
public int Sides
{
    get
    {
        return sides;
    }

    private set
    {
        sides = value;
    }
}
```

This renders it impossible to set this value in the Character class. However, since the get is still public you can still *retrieve* the value publicly:

```
public int Attack(Dice dice)
{
    //dice.Sides = this.DamageMaximum; //CANNOT set publicly
    int localSides = dice.Sides; //CAN get publicly
    return dice.Roll();
}
```



## Step 7: Why Use 'private set'

---

Perhaps this private set implementation is used because we are afraid of someone – another programmer using our codebase, for example – giving a value for this property that is not valid, and we want to keep this validation check as a helper method within the Dice class. However, because a property is really just a “gate keeper” for the backing field and has its own code block it can execute whenever a get/set is attempted, and we can put this validation check directly into the property itself. Here, if we try to set the property with anything greater than six – which in return actually sets the private field – a runtime error occurs (you can add specific exception handling):

```
public int Sides
{
    get { return sides; }
    set
    {
        if (value <= 6)
            sides = value;
        else
            throw new Exception();
        //some sort of error-handling goes here
    }
}
```

043

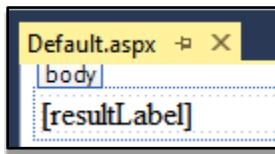
# Creating Constructor Methods

In the previous lessons, we saw that fields – whether accessed directly or behind properties – are a collection of values that come to represent the current *state* description of the object they belong to. It's generally a good idea to put a new object into a valid state as soon as possible before you start using it preferably at the time of instantiation. This would mean that you would want to construct the values for those fields – that represent that state – somewhere on the line of code where the new keyword is used. Luckily, there is a special type of method that allows you do to this called a "Constructor".

## Step 1: Create a New Project

---

For this lesson, create a new ASP.NET project called "CS-ASP\_043" with a single resultLabel Server Control:



Without realizing it, you have been using Constructors all along. Whenever you create a new object, you are invoking a default, empty constructor by using the empty parentheses:

```
protected void Page_Load(object sender, EventArgs e)
{
    Car myCar = new Car();
```

## Step 2: Understanding the Default Empty Constructor

---

You may wonder how this is possible if you didn't even write this method within the object's class. The answer is that it is created in the background regardless. If you could see it, it would look like this:

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }

    public Car()
    {
        //empty constructor
    }
}
```

## Step 3: Understanding Constructor Requirements

---

The constructor is different from ordinary methods in three key ways. The constructor must:

- (1) Have the same name as the class it belongs to.
- (2) Be called at instantiation, after the new keyword.
- (3) Not have a return type.

You can take control of exactly what the constructor does by explicitly writing its implementation details:

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }

    public Car()
    {
        this.Make = "Undefined";
        this.Model = "Undefined";
        this.Year = 0000;
        this.Color = "Undefined";
    }
}
```

## Step 4: Using Constructor Input Parameters for Initialization

---

Now, whenever a new Car object is created, it will take on these initialized values and retain them unless they are over-written. Better yet, you can let the values become initialized to whatever is supplied via the constructor's input parameters:

```
public Car(string Make, string Model, int Year, string Color)
{
    this.Make = Make;
    this.Model = Model;
    this.Year = Year;
    this.Color = Color;
}
```

## Step 5: Using the 'this' Keyword

---

The "this" keyword simply refers to the class-level properties of the particular object's instance and is necessary only when the constructor's input parameters are named the exact same way. Now, you can initialize these values to whatever you want at the point of instantiation:

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Car myCar = new Car("Oldsmobile", "Cutlas Supreme", 1985, "Silver");
    }
}
```

Add this public method to the Car class so that we can output the object's values via the resultLabel :

```
public string FormatDetailsForDisplay()
{
    return String.Format("Make: {0} - Model: {1} - Year: {2} - Color: {3}",
        this.Make,
        this.Model,
        this.Year.ToString(),
        this.Color);
}
```

Then call the method in the Default class, which returns the formatted string. And then see the output when running the application:

```
Car myCar = new Car("Oldsmobile", "Cutlas Supreme", 1985, "Silver");
resultLabel.Text = myCar.FormatDetailsForDisplay();
```

Make: Oldsmobile - Model: Cutlas Supreme - Year: 1985 - Color: Silver

## Step 6: Overloading the Constructor

Since a constructor is just like any other kind of method, you can overload it with several definitions. Here, we can add to option for an initialized object – yet not specifically defined – in the case of no input parameters given at instantiation:



Bob's Tip: to get the code-snippet shortcut for a constructor type "ctor" and then hit the tab key twice.

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }

    public Car()
    {
        this.Make = "Undefined";
        this.Model = "Undefined";
        this.Year = 1980;
        this.Color = "Undefined";
    }

    public Car(string Make, string Model, int Year, string Color)...
}

public string FormatDetailsForDisplay()...
```

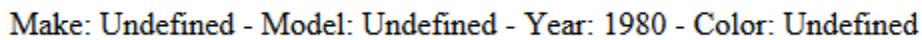
## Step 7: Using Intellisense to Cycle Through Overloaded Constructors

---

Now, if we choose to instantiate a Car without any specific values given as input parameters, we still have a Car object that is in a valid, though not specifically defined, state:

```
protected void Page_Load(object sender, EventArgs e)
{
    //Car myCar = new Car("Oldsmobile", "Cutlas Supreme", 1985, "Silver");
    Car myCar = new Car();
    resultLabel.Text = myCar.FormatDetailsForDisplay();
}
```

And when you run the application you will, at least, have some kind of data shown:



Make: Undefined - Model: Undefined - Year: 1980 - Color: Undefined

Recall that many of the predefined objects we have been working with, such as `DateTi me()`, have a variety of overloaded constructors that can put the object into a valid state. Go through the various constructors for `DateTi me()` available to you, and try to imagine what the constructors look like behind these twelve overload options:



# Naming Conventions for Identifiers

This lesson discusses the topic of naming conventions for common coding elements, such as, public classes, methods and properties, private fields, locally scoped variables, and so on. There are naming conventions that are established by the developer community at large. Then there are project-specific conventions agreed upon within the organization that you work in. Naming conventions are put in place to make it easier to discern meaning, and intent, in code.

## Step 1: Understanding PascalCasing vs camelCasing

---

Capitalization schemes are a common convention used to make code easier to read and follow. There is “PascalCasing,” where each word in the identifier is capitalized. And there is “camelCasing” where the first word is lower case followed by capitalization for subsequent words:

```
public int PascalCasing;  
private int camelCasing;
```

In general, everything that is `public` will be PascalCase while everything that is `private` will be camelCase. Take, for instance, this block of code:

```
namespace CS_ASP_044  
{  
    public class Hero ①  
    {  
        private string _name; ③  
        public string Name ②  
        {  
            get { return _name; }  
            set { _name = value; }  
        }  
  
        public Hero(string name)  
        {  
            _name = name;  
            heroHelperMethod();  
        }  
    }  
}
```

```

    5
    public void AttackPattern(Hero hero, Monster monster) { }

    private void heroHelperMethod() { }
}

```

4

- (1) The public class name Hero is PascalCase.
- (2) The public property Name is PascalCase.
- (3) The private backing field \_name is camelCase
- (4) The private heroHelperMethod() method is camelCase.
- (5) The public AttackPattern() method is PascalCase.

## Step 2: Using Underscores for Backing Fields

---

Notice how the backing field is prefixed with a single underscore. This is a general convention programmers adhere to when referring to backing fields. One of the reasons for doing so is you can quickly see, via Intellisense, that the member is a field.

*<NOTE: Bob describes the underscore scheme as a way to set apart from input parameters, ask for clarification of look up what others say on this topic>*

Variables that are locally scoped – that is to say, initially declared within the body of a method – are generally supposed to be camelCase. In this example, the hero, battle and damageInflictedOnMonster objects/variables are only “alive” within the context of this local Page\_Load() method, so they are set to lower-case/camelCase accordingly:

```

public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Hero hero = new Hero();
        Battle battle = new Battle();

        int damageInflictedOnMonster = 5;

        if (damageInflictedOnMonster > 50) { }
            //do something
    }
}

```

## Step 3: Where Server Control Property References Come From

---

You may have noticed that we have been referencing Server Controls in the Default class through class-level instances of these Controls. However, you may be wondering where exactly that instance was defined. As you can see the Default class is declared as a *partial* class which means that this is only a *partial* representation of it and another part of it is somewhere else. If you were to right-click on a Server Control via its instance, such as the resultLabel , you can select “Go To Definition” to see the other part of this class definition:



```
if (damageInflictedOnMonster > 50)
    resultLabel.Text = "A fatal blow!";

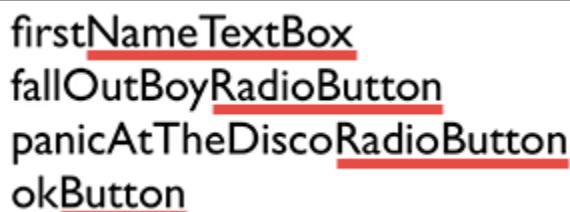
namespace CS_ASP_044 {

    public partial class Default {

        /// <summary> form1 control.
        protected global::System.Web.UI.HtmlControls.HtmlForm form1;

        /// <summary>
        /// resultLabel control.
        /// </summary>
        /// <remarks>
        /// Auto-generated field.
        /// To modify move field declaration from designer file to code-behind file.
        /// </remarks>
        protected global::System.Web.UI.WebControls.Label resultLabel;
    }
}
```

You can see that the instance is indeed a field within the broader class definition for the Default class. You will also see that it has a *protected* accessibility modifier, which is why it is camelCase as it is private to this class, as well as other classes that inherit from it (which is the rough definition of “*protected*”). Also, on the topic of Server Controls, notice how we have been including the Control name within the identifier to help communicate what it is an instance of. This is similar to a practice called “Hungarian Notation”:



firstNameTextBox  
fallOutBoyRadioButton  
panicAtTheDiscoRadioButton  
okButton

## Step 4: Older Conventions Like 'Hungarian Notation'

---

While not common within the .NET development community, you may see some code that subscribes to the "Hungarian Notation" naming convention. The hallmark of this convention is the inclusion of the variables type – usually trimmed down to three characters or less – prefixed directly to the variable name. Here is the meaning behind the Hungarian Notation used below:

- The 'i' in i Counter refers to type int.
- The 'str' in strFirst Name refers to type string.
- The 'btn' in btnOK refers to the Server Control type.
- The 'C' in CPerson refers to this as being a class.

"Hungarian Notation"

iCounter  
strFirstName  
btnOK  
CPerson



Bob's Tip: Hungarian Notation may seem like a good idea, but is generally considered to be an outdated way of representing coding elements. This convention lost favor for a host of reasons including the fact that the intended meaning tends to be forgotten in a broader system and is difficult to remember and enforce.

When using a development environment as sophisticated as Visual Studio it becomes quite redundant to include type, and scope, information directly within the variable name itself. You have a variety of tools that will volunteer this information to you with as little as moving the cursor over the variable name:

```
int damageInflictedOnMonster = 5;
```

(local variable) int damageInflictedOnMonster

045

# Static vs Instance Members

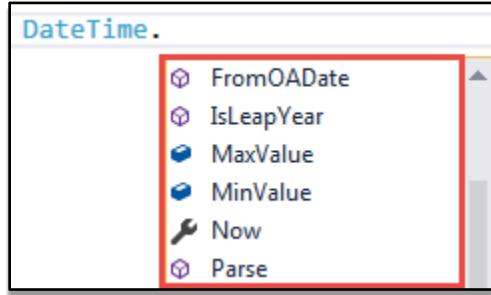
In this lesson, we're going to cover static vs instance class members. Simply put, a static method or property is one which belongs to – and is therefore accessible via – the class itself, without having to create an instance in order to access it. You may have noticed, in previous lessons, that there were some methods that were called without going through an instance in order to call it; there was no “new” keyword involved before we could access these methods:

```
DateTime.DaysInMonth(1969, 12);
String.Format("Hi {0}", "Bob");
int.Parse("21");
```

## Step 1: Using Intellisense to Identify Static Methods and Properties

---

All of these methods are static. The easiest way to distinguish static from instance methods is that they are accessible through the class name that they belong to. If you start typing a class name into Visual Studio, Intellisense will show you what static methods and properties are available to the type itself:



To demonstrate this difference between static and instance methods, fields and properties, create a simple class that contains both:

```
public class SomeClass
{
    public static int StaticField;
    public int InstanceField;

    public static void StaticMethod() { }
    public void InstanceMethod() { }
}
```

And now in another class, let's create an instance to see what's available to it via Intellisense. Note how the static field and static method are *not* available through the instance:

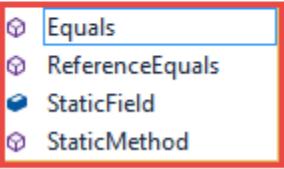
```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        SomeClass demo = new SomeClass();
        demo.in
    }
}
```



Also, note how the instance field and method are *not* available through referencing the class itself:

```
protected void Page_Load(object sender, EventArgs e)
{
    SomeClass demo = new SomeClass();
    demo.InstanceField = 5;

    SomeClass.
}
```



Bob's Tip: as a beginner, it's recommend that you not use the `static` keyword in your own classes unless they're classes that *only* contain public helper methods and that they don't contain any properties.

## Step 2: Using Static Keyword for Helper Methods

A common use for the `static` keyword is for whenever there is a public helper method that performs some function that isn't instance specific. Perhaps the method just processes some data and outputs it to the screen, for example. Here is a very contrived example:

```
public class Valuation
{
    public static int PerformCalculation(int one, int two)
    {
        //return one + two;
        return handleSomePartOfTheCalculation(one, two);
    }

    private static int handleSomePartOfTheCalculation(int one, int two)
    {
        return one + two;
    }
}
```

Notice how the static method `PerformCalculation()` references the other static method `handleSomePartOfTheCalculation()`. This is perfectly fine, however, it would not be possible for the static method to reference an instance method even if they both belong to the same class:

```
public static int PerformCalculation(int one, int two)
{
    //return one + two;
    return handleSomePartOfTheCalculation(one, two);
}

private int handleSomePartOfTheCalculation(int one, int two)
{
    return one + two;
}
```

The reason this is not allowed is because instance methods, in principle, have access to (1) instance fields, properties, and methods with its own class. If `handleSomePartOfTheCalculation()` were to (2) reference an instance property, then `PerformCalculation()` would (3) also be referencing that instance property (although indirectly). However, since `PerformCalculation()` is statically accessible, `handleSomePartOfTheCalculation()` wouldn't know what instance, if any, is being referenced:

```

public class Valuation
{
    public int myVar { get; set; } ①

    public static int PerformCalculation(int one, int two)
    {
        //return one + two;
        return handleSomePartOfTheCalculation(one, two);
    } ③

    private int handleSomePartOfTheCalculation(int one, int two)
    {
        return one + two + myVar;
    } ②
}

```

The reason why this doesn't work becomes obvious when we try to call the static method:

```

protected void Page_Load(object sender, EventArgs e)
{
    Valuation val1 = new Valuation();
    val1.myVar = 4;

    Valuation val2 = new Valuation();
    val2.myVar = 13;

    //Q: Does this return 24, or does it return 33?
    //A: Neither, because the answer is unknowable
    Valuation.PerformCalculation(8, 12);
}

```

## Step 3: An Instance Can Reference a Static, But not Vice Versa

---

We have no way of knowing what instance is being referenced by this static method, and neither does the compiler. That is why a static method can only access other static methods and properties. However, the inverse is possible, which is to say that an instance method can access a static property. Here, we're creating a simple static property that can count each time a new instance of type Valuation is created:

```
public class Valuation
{
    public static int ObjectCounter { get; set; }

    public Valuation()
    {
        ObjectCounter++;
    }
}

protected void Page_Load(object sender, EventArgs e)
{
    Valuation.ObjectCounter = 0;
    Valuation val1 = new Valuation(); //Valuation.ObjectCounter = 1
    Valuation val2 = new Valuation(); //Valuation.ObjectCounter = 2

    resultLabel.Text = Valuation.ObjectCounter.ToString();
}
```

## Step 4: Making an Entire Class Static

---

You can also mark an entire class as static, and the only time you would do that is if the class only holds static helper methods. Note that if you do mark a class as static, it is no longer able to hold any non-static fields, properties, or methods:

```
public static class Valuation
{
    public int myVar { get; set; }

    public static int PerformCalculation(int one, int two)
    {
        //return one + two;
        return handleSomePartOfTheCalculation(one, two);
    }

    private static int handleSomePartOfTheCalculation(int one, int two)
    {
        return one + two;
    }
}
```

## Step 5: Snapshot of the Static Math Class

---

One such class that exists in the .NET Framework is the Math class, which is a static class that only holds static methods (for performing common calculations), along with a few constants:

```
namespace System
{
    public static class Math
    {
        public const double E = 2.7182818284590451;
        public const double PI = 3.1415926535897931;

        public static short Abs(short value);
        public static decimal Abs(decimal value);
        public static double Abs(double value);
        public static float Abs(float value);
        public static long Abs(long value);
        public static int Abs(int value);
        public static sbyte Abs(sbyte value);
        public static double Acos(double d);
        public static double Asin(double d);
        public static double Atan(double d);
    }
}
```

As you can see, the helper methods in the Math class are all static because all they do is return some sort of numerical value. An instance is not needed to do a basic calculation:

```
protected void Page_Load(object sender, EventArgs e)
{
    double calc = Math.Floor(5.8); //returns a double without the decimal value
    resultLabel.Text = calc.ToString();
}
```

You can find more information about this class and its methods by visiting:

<http://v.gd/static>

046

# Working with the List<T> Collection

In this lesson, we're going to talk about working with Collections, specifically the kind that holds generic types. Collections perform the same type of function in code that arrays do, but tend to be easier to work with. This is because Collections are strongly typed – meaning they enforce a very specific class type to be entered – and easily allow for elements to be added or removed. This lesson will primarily look at the List Collection, which is generic (as indicated by the angled brackets):

## List<T>

A List is a type, within the .NET Framework, that can store multiple items (just like an array) of a single type determined within angled brackets. The <T> part refers to the class definition for List, that can take in any type and then wherever T is referenced throughout the class, it assumes the type given.

```
public class List<T>
{
    ...
}
```

This means that we can determine what the type is, when working with a List, by providing it within the angled brackets, such as in these examples:

```
List<int> nums = new List<int>();
List<string> phrases = new List<string>();
List<Car> cars = new List<Car>();
```

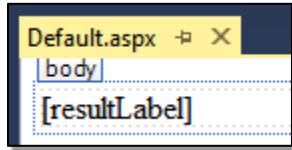


Bob's Tip: You can make any class or method generic by postfixing <T> after the identifier. The topic of Generics is somewhat advanced, but you will see one of its greatest purposes by using Lists and Dictionaries. Keep in mind that "generic" refers to the class/method definition side of the equation. In actuality, the type handled by a generic class/method becomes *specific* as soon as you make use of it (through an instance, or calling a method, for example).

## Step 1: Create a New Project

---

Create a new ASP.NET project for this lesson and call it "CS-ASP\_046" and include a single result Label Control:



Also, add a Car class, and within it write the following:

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }

    public Car(string make, string model, int year, string color)
    {
        this.Make = make;
        this.Model = model;
        this.Year = year;
        this.Color = color;
    }

    public string FormatDetailsForDisplay()
    {
        return String.Format("Make: {0} - Model: {1} - Year: {2} - Color: {3}<br/>",
            this.Make,
            this.Model,
            this.Year.ToString(),
            this.Color);
    }
}
```

In the *Default.aspx.cs*, let's create several instances of a Car object:

```
protected void Page_Load(object sender, EventArgs e)
{
    string result = "";

    Car car1 = new Car("BMW", "528i", 2010, "Black");
    Car car2 = new Car("BMW", "745li", 2005, "Black");
    Car car3 = new Car("Ford", "Escape", 2008, "White");

    resultLabel.Text = result;
}
```

## Step 2: Make A List<Car> and Add Cars To It

---

And now let's make a List of type Car and add each car to the List using the Add() helper method available to the List instance. Note that once the generic type becomes specific (in this case, of type Car), you can only add cars to the List:

```
List<Car> cars = new List<Car>();

cars.Add(car1);
cars.Add(car2);
cars.Add(car3);

resultLabel.Text = result;
```

Since a List is much like an array, you can do many of the same things you've come to expect from arrays:

```
for (int i = 0; i < cars.Count; i++)
{
    result += cars[i].FormatDetailsForDisplay();
}

resultLabel.Text = result;
```

```
Make: BMW - Model: 528i - Year: 2010 - Color: Black
Make: BMW - Model: 745li - Year: 2005 - Color: Black
Make: Ford - Model: Escape - Year: 2008 - Color: White
```

## Step 3: Previewing the Power of Lambda Expressions

---

One of the most useful features of Collections in general is their built-in methods that utilize Lambda Expressions. While outside of the scope of this lesson, a Lambda Expression could be thought of as a compact method that you can define and insert *into the input argument* of another method. To give you a sense of the simplicity and power of Lambda Expressions we'll briefly look here at some typical usage scenarios. While the mechanics of how this all works is founded upon more advanced topics, such as Generic Delegates and Anonymous Methods, what should be clear is how Collections allow for all kinds of data manipulation and retrieval at the hands of very little code.

Here a Lambda Expression is used in the `FindAll()` method to retrieve a `List<Car>` of all elements in `cars` that match "White" for the `Color` field (if you wanted to find cars of a different color, simply change the match string to a different color):

```
List<Car> whiteCars = cars.FindAll(p => p.Color == "White");
```

Another common scenario is to change values for a particular field of each element in the `List`. You can do that by accessing the `ForEach()` method and passing in the following Lambda Expression – in this case applying a new coat of paint to each car in the `List`:

```
cars.ForEach(p => p.Color = "Silver");
```

If you want to filter the `List` by cars made within a certain year, you can do so with the `Exists()` method. And since it returns a `bool` – depending on if a match is found with the Lambda Expression input argument – we can wrap it in a conditional as follows:

```
if (cars.Exists(p => p.Year == 2008))
    result += "We do have a 2008 model in stock!";
```

047

# Object Initializers

In this lesson, we're going to talk about Object Initializers. We have already seen how constructors can be used to set object properties with initial values and Object Initializers work in a somewhat similar way.

## Step 1: Create a New Project

---

For this lesson, create an ASP.NET project called "CS-ASP\_047", using where we left off in the previous lesson as the starting point.

If we didn't have a constructor for a class, how would we initialize these properties?:

```
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string Color { get; set; }

    public string FormatDetailsForDisplay()
    {
        return String.Format("Make: {0} - Model: {1} - Year: {2} - Color: {3}<br/>",
            this.Make,
            this.Model,
            this.Year.ToString(),
            this.Color);
    }
}
```

The most obvious way would be to set the properties individually, right after instantiation:

```
protected void Page_Load(object sender, EventArgs e)
{
    Car car = new Car();
    car.Make = "BMW";
    car.Model = "528i";
    car.Color = "Black";
    car.Year = 2010;
}
```

## Step 2: Using Object Initializers Instead of Constructors

---

Object Initializers let us do this all on one line, with more clarity, and less typing:

```
Car car1 = new Car() { Make = "BMW", Model = "528i", Color = "Black", Year = 2010 };
Car car2 = new Car() { Make = "BMW", Model = "745li", Color = "Black", Year = 2005 };
Car car3 = new Car() { Make = "Ford", Model = "Escape", Color = "White", Year = 2008 };
```

And then we can add the objects to a List the way we have seen before:

```
List<Car> cars = new List<Car>();

cars.Add(car1);
cars.Add(car2);
cars.Add(car3);
```

## Step 3: Adding an Un-Named Object to a List

---

However, we can save keystrokes when adding a new object to a list by using an Object Initializer to create a new, un-named, object and pass it right in to the Add() method:

```
protected void Page_Load(object sender, EventArgs e)
{
    string result = "";

    List<Car> cars = new List<Car>();

    cars.Add(new Car { Make = "BMW", Model = "528i", Color = "Black", Year = 2010 });
    cars.Add(new Car { Make = "BMW", Model = "745li", Year = 2005, Color = "Black" });
    cars.Add(new Car { Make = "Ford", Model = "Escape", Year = 2008, Color = "White" });

    resultLabel.Text = result;
}
```

We don't even need the object's identifier, in this case, because the reference is being held by its position within the List and can be accessed as any list member, such as by iterating through it:

```
for (int i = 0; i < cars.Count; i++)
{
    result += cars[i].FormatDetailsForDisplay();
}

resultLabel.Text = result;
```

# Collection Initializers

This lesson picks up from where we left off in the previous lesson, which showed us how to create Object Initializers and even showed how to add them to a List by passing them directly, as an input parameter, into the List's Add() method.

## Step 1: Understanding Collection Initializers via Object Initializers

---

Collection Initializers provide an even more elegant shorthand by allowing us to add each object to the List, right at the List's instantiation. Note, once again, how you don't have to include the identifier as the List holds reference to the object through its indexed position within the List:

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string result = "";

        List<Car> cars = new List<Car>()
        {
            new Car {Make="BMW", Model="528i", Color="Black", Year=2010},
            new Car {Make="BMW", Model="745li", Color="Black", Year=2005},
            new Car {Make="Ford", Model="Escape", Color="White", Year=2008}
        };

        for (int i = 0; i < cars.Count; i++)
        {
            result += cars[i].FormatDetailsForDisplay();
        }

        resultLabel.Text = result;
    }
}
```

## Step 2: Understanding Initializer Benefits

---

Also, notice that we don't have to reference the Car's invocation parentheses for the empty constructor:

```
new Car {Make="BMW", Model="528i", Color="Black", Year=2010},
```



Creating this kind of Collection Initializer may look strange, at first, but it is really not much different from an Object Initializer. The List object is itself an object, like any other, that should be initialized to a valid state. Collection Initializers offer a convenient way to accomplish this.



Bob's Tip: the main benefit we're getting by embedding of the object initializer syntax inside of the collection initializer syntax – is that we're putting the Collection into a valid state all in one shot. Otherwise, it's possible to have an empty List which could serve the potential for errors.

# Working with the Dictionary< TKey, TValue> Collection

Dictionaries are Generic Collections that allow you to specify the type for not only the value stored as an item in the Collection, but also the type used for the key that refers to that item's place within the Collection. Take, for instance, an ordinary List<string> which has an implied integer index that can be used to refer to each string item within the List. A Dictionary's key is very similar to the index used in the List except that it doesn't have to be an integer, but rather can be of any type (although, it is typically of type int or string).

This can make things easier when trying to find a particular item in a large set of items, and all you have to do is refer to its key rather than iterate through the entire Collection. For example, you might have a bunch of cars that you want to store in a Collection and be able to easily retrieve a car by its VIN number. Using a Dictionary, you can use the VIN number as the key and the object's class type as the value, for example:

```
Dictionary<string, Car> cars = new Dictionary<string, Car>();
```

Here we're using a string to hold the VIN number which can be used to refer to a particular Car object (in other words, the key and the value are "paired" together). Here's one way that we can add a Car, with a VIN as the key, into the Dictionary Collection:

```
cars.Add("V1", new Car { Make = "BMW", Model = "528i", Year = 2010, Color = "Black" });
```

A Collection Initializer might look a bit cleaner if you have multiple elements, so you can write that as follows:

```
Dictionary<string, Car> cars = new Dictionary<string, Car>() {
    { "V1", new Car {Make = "BMW", Model = "528i", Year = 2010, Color = "White"} },
    { "V2", new Car {Make = "BMW", Model = "745li", Year = 2005, Color = "Black"} },
    { "V3", new Car {Make = "Ford", Model = "Escape", Year = 2008, Color = "White"} }
};
```

Although you are now using a key instead of an array-like indexer, you can still iterate through the Dictionary by using the ElementAt() method and passing an argument that behaves like an ordinary indexer. Below shows this in action, as well as printing out the key and value for each element:

```
for (int i = 0; i < cars.Count; i++)
{
    result += cars.ElementAt(i).Key + " - "
        + cars.ElementAt(i).Value.FormatDetailsForDisplay();
}

resultLabel.Text = result;
```

```
V1 - Make: BMW - Model: 528i - Year: 2010 - Color: White
V2 - Make: BMW - Model: 745li - Year: 2005 - Color: Black
V3 - Make: Ford - Model: Escape - Year: 2008 - Color: White
```

If you want to simply retrieve a single value out of the Collection you can use the TryGetValue() method which will return a value using the specified key and then output it to an existing variable:

```
Car v2;
if (cars.TryGetValue("V2", out v2))
    result += v2.FormatDetailsForDisplay();

resultLabel.Text = result;
```

```
Make: BMW - Model: 745li - Year: 2005 - Color: Black
```

To remove an element in the Collection you can call the Remove() method and since it returns a bool - depending on if the element was successfully removed - you can wrap it in a conditional as follows:

```
if (cars.Remove("V1"))
    result += "Successfully removed V1<br/>";

resultLabel.Text = result;
```

050

# Looping with the foreach() Iteration Statement

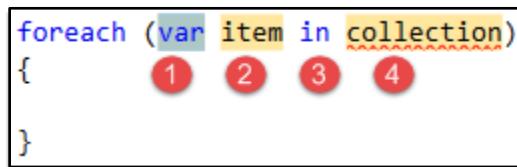
We have already seen, in previous lessons, how to iterate through an array/collection with the `for()` looping statement. However, there is another iteration statement available to you that can complete this task called `foreach()`. This looping statement can be seen as a simplified `for()` loop, using a fixed single-step increment and automatically terminating at the last indexed item within the array/collection.

## Step 1: Starting With the 'foreach' Code Snippet

---

You can execute the code-snippet for the `foreach()` loop by typing in “`foreach()`” and hitting the tab key twice. The `foreach()` is structured as follows:

- (1) The type for the item in the array/collection.
- (2) The temporary identifier, local only to the loop, for referencing each item in the iteration.
- (3) The ‘in’ keyword, denoting relationship between the item and the array/collection it is in.
- (4) The identifier for the array/collection you want to iterate through.



```
foreach (var item in collection)
{
    1   2   3   4
}
```

## Step 2: Replacing a for() Loop with a foreach()

---

In the previous lesson, we used a `for()` loop to iterate through the `List<Car>`. Let’s perform the equivalent task but with using a `foreach()` loop instead:

```

protected void Page_Load(object sender, EventArgs e)
{
    string result = "";

    List<Car> cars = new List<Car>() {
        new Car { Make = "BMW", Model = "528i", Year = 2010, Color = "White" },
        new Car { Make = "BMW", Model = "745li", Year = 2005, Color = "Black" },
        new Car { Make = "Ford", Model = "Escape", Year = 2008, Color = "White" }
    };

    foreach (Car car in cars)
    {
        result += car.FormatDetailsForDisplay();
    }

    resultLabel.Text = result;
}

```

Make: BMW - Model: 528i - Year: 2010 - Color: White  
 Make: BMW - Model: 745li - Year: 2005 - Color: Black  
 Make: Ford - Model: Escape - Year: 2008 - Color: White

This is, essentially, a more intuitive way of representing a `for()` loop with a single-step incrementing counter:

```

for (int i = 0; i < cars.Count; i++)
{
    result += cars[i].FormatDetailsForDisplay();
}

```

### Step 3: Understanding the `foreach()` Implied Indexer

---

Notice how the `foreach()` loop doesn't even use an indexer because it's already implied that we're moving up one index for each iteration. You can certainly include an indexer, but it would be redundant:

```

int i = 0;

foreach (Car car in cars)
{
    result += cars[i].FormatDetailsForDisplay();
    i++;
}

```

## Step 4: Understanding the Temporary Local Variable in a foreach()

---

Also, notice how the `foreach()` stores each iterated item into a temporary local variable, which you can use to access the properties/methods available to that item. It's important to remember that the value this temporary variable stores gets re-written with each successive iteration:

```
foreach (Car car in cars)
{
    result += car.FormatDetailsForDisplay();
}
```



Bob's Tip: notice how the syntax in the `foreach()` loop is inherently readable and expresses exactly what is happening, which is essentially: "for each item in the collection, do this..."

051

# Implicitly Typed Local Variables with the var Keyword

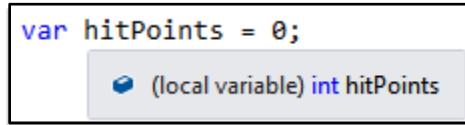
This lesson will talk about *implicitly typed* local variables using the var keyword. The term “local” refers to variables declared within a method scope, or narrower, and not available at a class level. And “implicitly typed” means the compiler can figure out what the actual type is based on context, and its initial type declaration. You can let the compiler interpret the type by using the var keyword in place of the actual type declaration.

```
//the compiler interprets  
//these both as of type int  
int hitPoints = 0;  
var hitPoints = 0;
```

## Step 1: Understanding How the ‘var’ Keyword Works

---

This works because the compiler references the *actual* type when compiling to Intermediate Language (IL). You can see that Intellisense has no trouble understanding that this is supposed to be an int since it was already initialized to an integer value:



A screenshot of Visual Studio's Intellisense feature. A tooltip is displayed over the word 'hitPoints' in the code. The tooltip contains a blue circular icon followed by the text '(local variable) int hitPoints'. The background shows a snippet of C# code with 'var hitPoints = 0;' highlighted.

This extends beyond simple value types and works with objects as well:

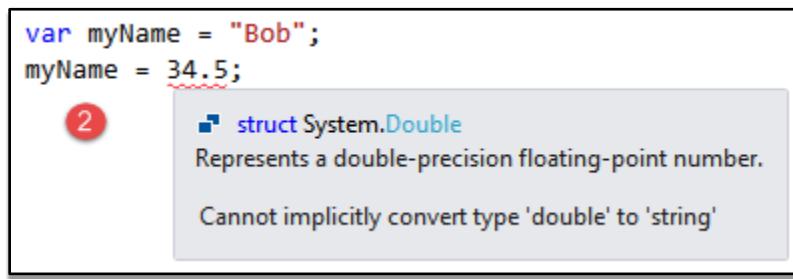
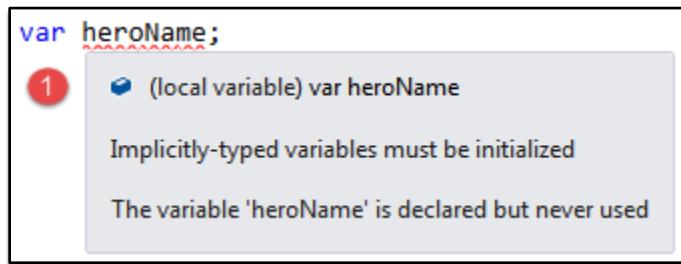
```
var cars = new List<Car>() {  
    new Car { Make = "BMW", Model = "528i", Year = 2010, Color = "White" },  
    new Car { Make = "BMW", Model = "745li", Year = 2005, Color = "Black" },  
    new Car { Make = "Ford", Model = "Escape", Year = 2008, Color = "White" }  
};
```

## Step 2: Caveats When Using the 'var' Keyword

---

There are a few caveats when dealing with implicitly typed variables:

- (1) You must initialize with an assigned value type or object reference.
- (2) It is still type-strict. It takes on the implied data type, and cannot be changed any more than any other explicitly typed variable.



You may be wondering about the benefit of having implicitly typed variables since the var keyword adheres to the same strongly typed guidelines, and just obscures what should otherwise be obvious when you look at code at a glance. You will begin to see the benefits later on where, in some cases, it's difficult to determine the data type ahead of time, the data type is unknown, or the data type becomes generated automatically at runtime. Just keep its function in mind as you will often see it being used in code examples provided on the internet. And, eventually, you will come across a case in which it is necessary.

052

# Creating GUIDs

In preparation for an upcoming lesson, we'll want to talk about something called a "GUID." This acronym stands for "Globally Unique Identifier" and is an alpha-numeric string of characters that is unique to your computer throughout all history. A GUID algorithm takes in a variety of uniquely identifying characteristics, such as your network MAC address, as well as the date, in order to generate your machine's particular ID. From a .NET developer perspective, GUIDs are often used in databases. For example, whenever you're writing a new record into a database, you would give it a unique GUID. That GUID is then unique to that particular row of data, regardless of whether it's copied elsewhere or if you delete/reinsert rows, you don't have to worry about the ID changing.

## Step 1: Why Use a GUID?

---

Some people would claim that GUIDs are horrible to use for database IDs because they're not necessarily unique. There are situations where you can, theoretically, generate two GUIDs that are exactly the same on two different computers. The chances of that are quite small, unless you're working with a massive set of data.

## Step 2: Create a New Project

---

To demonstrate GUIDs, create a new ASP.NET project called "CS-ASP\_052" with a single result Label Control. In *Default.aspx.cs*, write the following for the *Page\_Load()* method in order to generate a unique GUID:

```
protected void Page_Load(object sender, EventArgs e)
{
    //http://en.wikipedia.org/wiki/Globally_unique_identifier

    var myGuid = Guid.NewGuid();
    resultLabel.Text = myGuid.ToString();
}
```

Notice how GUID is a built-in value type in the .NET Framework. When you run the application, you will get a GUID that is unique to your machine:

08ef3396-6773-4f1d-946e-c1f62ab2e48d

## Step 3: Parse a String into a GUID with the Parse() Method

---

You can also parse a string into a GUID by accessing the static Parse() method:

```
//Parsing a string into a Guid  
var myOtherGuid = Guid.Parse("6d8786be-3bbb-4506-a900-aacb5614542d");
```

As with most cases of using a Parse() method, it's probably better to use TryParse() instead and return the result via an out parameter:

```
Guid myOtherGuid;  
if (Guid.TryParse("6d8786be-3bbb-4506-a900-aacb5614542d", out myOtherGuid))  
{  
    resultLabel.Text = myOtherGuid.ToString();  
}
```

If it were not a valid GUID, it wouldn't output anything at all:

```
if (Guid.TryParse("not-a-valid-guid", out myOtherGuid))  
{
```

Now run the application to see if it is, in fact, stored as a valid GUID:

```
6d8786be-3bbb-4506-a900-aacb5614542d
```

053

# Working With Enumerations

In this lesson, we're going to talk about enumerations – or rather “enums” as they're known in C#. An enum is a special data type that can hold, at any moment, one of a limited set of possible values that you determine. To demonstrate this, imagine that we're creating an application that collects information about pet ownership. You will want to create a pet class, and capture the name of the pet, the age of the pet, and the type of pet.

```
public class Pet
{
    string Name { get; set; }
    int Age { get; set; }
    string Type { get; set; }

    public Pet(string Name, int Age, string Type)
    {
        this.Name = Name;
        this.Age = Age;
        this.Type = Type;
    }
}
```

## Step 1: Why Use an Enum?

---

Now, the pet type might be one of several different available options: a dog, a cat, a fish, an elephant, and so on. However we should know ahead of time that it is a limited set, and that we can't declare just *anything* as a valid pet type:

```
Pet pet = new Pet("Fluffy", 3, "Velociraptor");
```

It would be difficult to account for these various possibilities using a string data type for holding the PetType (in this case, you would probably create a property that checks for valid entries via a conditional statement in the “setter”). However, an enum – by definition – allows you to restrict your set of options only to values that are relevant. Here is how you create an enum:

```
public enum PetType
{
    dog,
    cat,
    fish,
    elephant
}
```

And now you can change the data type, in the Pet class, from string to PetType:

```
public class Pet
{
    string Name { get; set; }
    int Age { get; set; }
    PetType Type { get; set; }

    public Pet(string Name, int Age, PetType Type)
    {
        this.Name = Name;
        this.Age = Age;
        this.Type = Type;
    }
}
```

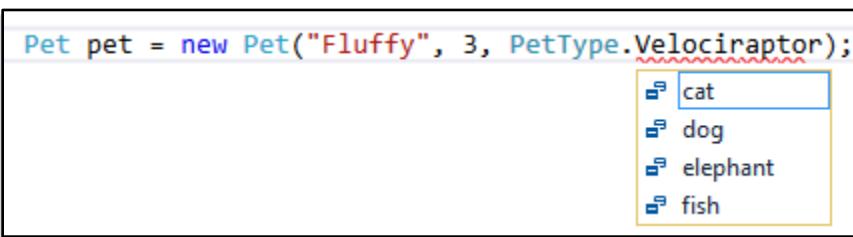
And now, when you reference the Type property, you can only set it to one of the available options:

```
Pet pet = new Pet("Fluffy", 3, PetType.dog);
```

## Step 2: Using Intellisense to See Enum Options

---

Perhaps more importantly, Intellisense recognizes that we have a limited set of options and stops us from using an invalid value at compile time:



You may be wondering what the real value is in limiting possibilities in your code. After all, somebody might have an uncommon pet such as a Zebra, or an Eel. By using an enum, you would just modify your code to accept those types of pets while retaining the added benefit of not allowing something seemingly innocuous – such as wanting a Velociraptor as a pet – to end up breaking your code.

Enumerations are used all over the place in the .NET Framework Class Library, primarily because they limit the possible values that you can pass into methods or set into properties. If the creators of the .NET Framework Class Library allowed you to pass any value you wanted to into a given method/property, there is a higher chance that an incorrect value will be provided and make for more debugging problems in your code.



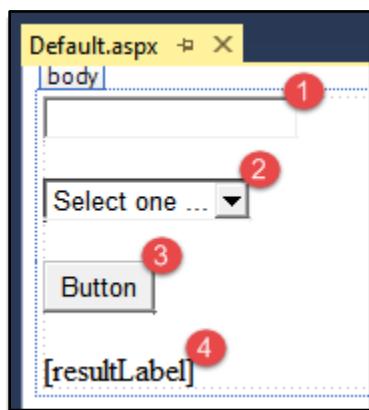
Bob's Tip: don't think of creating constraints inside of software development as necessarily a bad thing. In fact, it's often a *great* thing, in that it helps you to write more robust and error-free code than would otherwise be possible. The greatest challenge with writing code is simplifying everything, and enums can be an important part of that simplification process.

## Step 3: Create a New Project

---

Create a new ASP.NET project for this lesson and call it "CS-ASP\_053." Add the following Server Controls and programmatic IDs to the *Default.aspx* file:

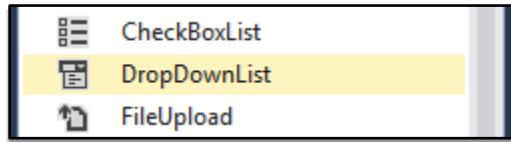
- (1) heroNameTextBox
- (2) heroTypeDropDown
- (3) Button1
- (4) resultLabel



## Step 4: Include a DropDownList Server Control

---

The second Server Control is a DropDownList. That can be accessed via the ToolBox:



Next, in *Default.aspx.cs* create a public enum that is incorporated via a property in a public Character class:

```
namespace CS_053
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)...
        protected void Button1_Click(object sender, EventArgs e)...

    }

    public class Character
    {
        public string Name { get; set; }
        public CharacterType Type { get; set; }
    }

    public enum CharacterType
    {
        Wizard,
        Fighter,
        Monster,
        HighWizard
    }
}
```

## Step 5: Demonstrating Error-Prevention via Enums

---

Now, in the Page\_Load() method you can create a new Character and select its enum value:

```
protected void Page_Load(object sender, EventArgs e)
{
    var hero = new Character();
    hero.Name = "Elric";
    hero.Type = CharacterType.Fighter;
    // In some other part of your code ...
    if (hero.Type == CharacterType.Fighter)
    {
        resultLabel.Text = "Our hero is a fighter!";
    }
}
```

If we used a string instead of an enum, for this type, it would be subject to all sorts of errors that won't even be caught by the compiler and would require complete consistency on our part:

```
hero.Type = "Fighter";
// In some other part of your code ...
if (hero.Type == "Fighter")
{
    resultLabel.Text = "Our hero is a fighter!";
}
```

Another thing worth mentioning with the enum definition is that it can be nested within another class. In this case, it may actually make sense to consider that the class and enum seem to conceptually belong together:

```

public class Character
{
    public enum CharacterType
    {
        Wizard,
        Fighter,
        Monster,
        HighWizard
    }

    public string Name { get; set; }
    public CharacterType Type { get; set; }
}

```

Now, we need to modify our references because the `CharacterType` enum is accessed *through* the `Character` class first.

```

hero.Type = Character.CharacterType.Fighter;

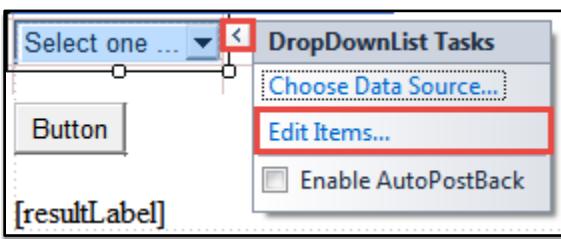
// In some other part of your code ...
if (hero.Type == Character.CharacterType.Fighter)
{
    resultLabel.Text = "Our hero is a fighter!";
}

```

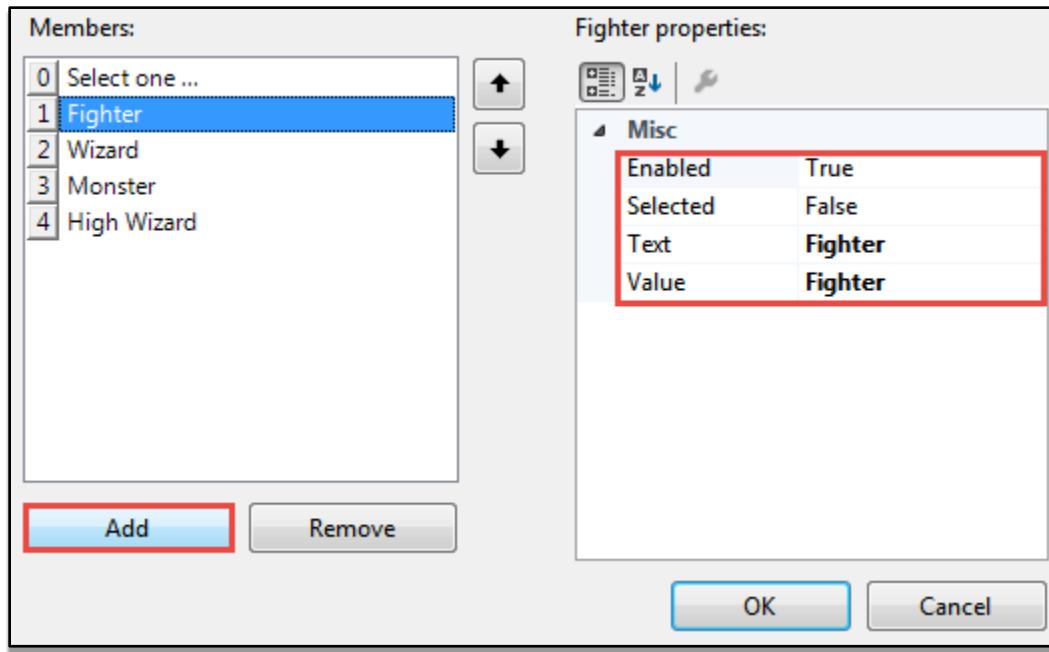
## Step 6: Populate the DropDownList in the Designer

---

Let's now demonstrate how we can use the `DropDownList` to select amongst the available values for the `Character.CharacterType` enum. We first have to populate the `DropDownList` with a set of default options. Click on the arrow beside the Server Control in the Design View, and select "Edit Items...":



Then simply click the “Add” button and modify the displayed text and stored value for each item:



Back in the Button1\_Click() method we (1) created a local variable to store the enum and then (2) returned to it a value chosen from the DropDownList Control. This value then (3) gets input through the static Enum.TryParse() method and, if it successfully parsed the string to an enum, it (4) gets stored into the hero. Type enum property:

```
protected void Button1_Click(object sender, EventArgs e)
{
    var hero = new Character();
    hero.Name = heroNameTextBox.Text;

    Character.CharacterType selection; 1
    if (Enum.TryParse(heroTypeDropDownList.SelectedValue , out selection)) 2
    {
        3 hero.Type = selection;
    } 4
}
```

You can also add a special message, to the bottom of the Button1\_Click() method, depending on the enum value that is currently selected once the button is pressed:

```
if (hero.Type == Character.CharacterType.Fighter)
{
    resultLabel.Text = "You selected a fighter!";
}
```

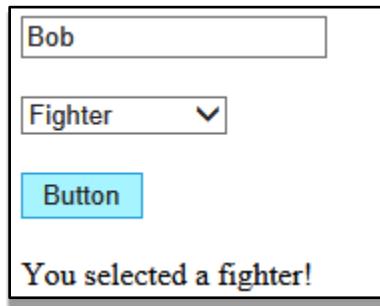
## Step 7: Comment Out Page\_Load() and Run the Application

---

Now run the application, but first be sure to comment out the Page\_Load() method as it will just interfere with the results:

```
//var hero = new Character();
//hero.Name = "Elric";
//hero.Type = Character.CharacterType.Fighter;

//// In some other part of your code ...
//if (hero.Type == Character.CharacterType.Fighter)
//{
//    resultLabel.Text = "Our hero is a fighter!";
//}
```



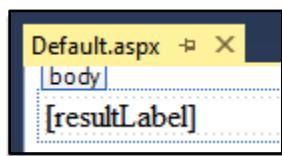
# Understanding the switch() Statement

This lesson will discuss the `switch()` statement, which is a flow control similar to an `if()` statement. An apt analogy is that of a switching railroad track that changes course depending on a certain condition being met. The main difference between a `switch()` and an `if()` is the former evaluates a single operand against a selection of case scenarios, whereas an `if()` can have a cascade of varying evaluations each using multiple operands. In both conditional statements, different blocks of code will execute depending on if an evaluation match is found.

## Step 1: Create a New Project

---

For this lesson, create a new ASP.NET project called "CS-ASP\_054" with a single `resultLabel` Control:



You can access the `switch()` code snippet by typing in "switch" and hitting the tab key twice:

```
switch (switch_on)
{
    default:
}
```

## Step 2: Understanding What a `switch()` Consists Of

---

A `switch()` statement typically consists of:

- (1) A single operand to be evaluated.
- (2) Conditions (cases) evaluated against that operand.
- (3) A default if none of the conditions evaluate True.

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

        string result = "";
        int switchExpression = 4;

        switch (switchExpression) ①
        {
            case 0:
                result += "Case 0";
                break;
            case 1:
                result += "Case 1";
                break;

            ③ default:
                result += "Default (Optional)";
                break;
        }

        resultLabel.Text = result;
    }
}
```

### Step 3: Understanding 'case', 'default' and 'break' Components

---

The above example would display "Default (Optional)" because none of the cases, 0 or 1, match the actual value of the operand being evaluated, 4. However, if a case does match the value of the evaluated element you will see the "code block" execute for the particular case:

```
int switchExpression = 1;
```

Case 1

The break keyword tells the compiler to exit – or break out of – the switch() statement after the code block has completed. If you do not use a break after a case, it groups it with the next case in line. This could be read as saying “if the case is 0 or 1, perform the code block for case 1”

```
case 0:  
case 1:  
    result += "Case 0 or 1";  
    break;
```

Case 0 or 1

## Step 4: Using the ‘return’ Keyword in a switch()

---

You can also use the return keyword, which would exit the switch() statement entirely, as well as exiting the method its operating within:

```
case 0:  
    result += "Case 0";  
    return;
```

It’s important to realize that returning out of the method happens right away so any code occurring below that keyword will never execute:

```
case 0:  
    return;  
    result += "Case 0";  
        ↪ (local variable) string result  
  
    Unreachable code detected
```

## Step 5: Using the ‘goto’ Keyword

---

You can use the goto keyword to execute a code block associated with another case. In this scenario, if case 2 is a match it will add “Case 2” to result and then immediately go to case 99 and break out of the switch() entirely:

```

int switchExpression = 2;

switch (switchExpression)
{
    case 0:
    case 1:
        result += "Case 0 or 1";
        break;
    case 2:
        result += "Case 2";
        goto case 99;

    case 99:
        break;
    default:
        result += "Default (Optional)";
        break;
}

```

To make this a bit more obvious, change the following case and run the application:

```

case 2:
    result += "Case 2<br />";
    goto case 0;

```

Case 2  
Case 0 or 1

You can even perform a mathematical calculation and return the result as the matching case. This is equivalent to writing "case 2":

```

case 1 + 1:
    result += "Case 2";
    goto case 0;

```

You can also use the switch() to perform exception handling:

```

case 1 + 1:
    result += "Case 2";
    throw new Exception();

```



## Step 6: Pairing switch() with Enum

---

Switch() statements work particularly well with enums, and there are even some handy shortcuts built into Visual Studio to facilitate this. To demonstrate this, let's re-create the Character class and CharacterType enum from the previous lesson:

```
public class Character
{
    public string Name { get; set; }
    public CharacterType Type { get; set; }

}

public enum CharacterType
{
    Wizard,
    Fighter,
    Monster
}
```

In Page\_Load() type in the following code, using the code-snippet shortcut for the switch() statement:

```
protected void Page_Load(object sender, EventArgs e)
{
    var hero = new Character();
    hero.Name = "Elric";
    hero.Type = CharacterType.Fighter;

    string result = "";

    switch (hero.Type)
    {
        default:
    }

    resultLabel.Text = result;
}
```

After you type in the hero.Type property hit the Enter key on your keyboard twice, and Visual Studio will extrapolate the available values for the enum as cases within the switch():

```
switch (hero.Type)
{
    case CharacterType.Wizard:
        break;
    case CharacterType.Fighter:
        break;
    case CharacterType.Monster:
        break;
    default:
        break;
}
```

All you have to do now is add your case specific code blocks, and you have a switch() and enum working in tandem. The general rule of thumb is to use switch() statements when evaluating possible values in enums and you need to check a variety of case scenarios against a single operand, otherwise use a switch() wherever it produces a more elegant solution to what an if() statement would provide.

055

# First Pass at the Separation of Concerns Principle

There are a variety of principles that you will come across in software development. These principles are meant to apply regardless of the programming language, or the platform, that you're working on. Many of these principles have been cultivated amongst software developers for decades, having become proven guidelines for sound software development. We already saw a few principles, and rules of thumb being discussed in previous lessons. We came to understand the "Principle of Single Responsibility," which states that any given method should do one task and one task only.



Bob's Tip: a general indicator for determining whether or not a method you're writing violates the Single Responsibility Principle (SRP) is if you need to use the word "and" when verbally describing it. Consider this when saying out loud "this method determines the result (of some process) *and* formats text for displaying onto the screen." This could be a clue that this method is doing too much and that these two very different tasks should be housed within their own methods.

## Step 1: Understanding SoC via Code Layers

---

We also understood the rule of thumb of keeping classes as lean and specific as possible – some developers even going so far as to say your class shouldn't extend past the vertical height of your screen. Continuing on, in this spirit of software design principles, is another principle called the "Separation of Concerns" principle, or "SoC." This principle describes the steps you take to keep your codebase as loosely coupled as possible, in effort to make it more resilient to change. Change is the sworn enemy of software developers and yet developers face change every single day, such as, changing business requirements, changing code to address fixes, changing API's, and even changing entire platform dependencies. Change is inevitable, but we can mitigate the impact of change by separating the concerns of an application. There are, typically, three major layers of concern within an average application's design:

- The Presentation Layer
- The Domain Layer
- The Persistence Layer

The Presentation Layer is concerned with display logic, such as taking data and presenting it to the user. Then there's the Domain Layer, which is concerned with the Domain Logic, referring to the general business/problem domain that the application solves. For instance, if your application's main domain deals with keeping an inventory of a warehouse, then that domain layer might process classes and objects that keep track of aspects like items and shipment. Finally, the Persistence Layer is concerned with handling permanent data storage, including its creation, updating, and retrieval. Whatever information is meant to persist between the point that you shut down an application and start it back up again would belong in this Domain Layer within your application.

Up to this point, we haven't really saved any data into a database or dealt with any persistent information. However, we have been writing applications that contain, both, presentation concerns and domain concerns. The presentation concern, in an ASP.NET Web Form API, are things like the actual ASP.NET syntax, such as the special ASP syntax mixed in with the HTML:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="resultLabel" runat="server"></asp:Label>
        </div>
    </form>
</body>
</html>
```

There are also the ASP.NET Server Controls, which is an API made available to you when you create Web Form applications. There's also the whole ASP.NET event model, which we've employed through special methods tied into events, such as `Page_Load()`, `_Click()`, and so on. All of those presentation concerns are already taken care of via Microsoft's existing codebase except for some presentation logic that we've written for taking in user-input and displaying information back to the user. The Domain Layer meanwhile is all of the business logic that we write including calculations and decision statements bound up within classes that will become the foundation for our major "domain" objects. We've created something resembling this with the methods and properties we defined in elements such as the Character, Hero, and Weapon classes.



Bob's Tip: by seeing a business represented through code a software developer will, oftentimes, come to understand how a given business operates more intimately than subject matter experts within the actual day-to-day operations. The reason is that developers have to come to know every possible decision or consideration in order to automate it properly. They force business owners to answer critical questions to deal with every possible business-related contingency, so as to determine how the code responds based on those contingencies.

Layering your application at a higher level – and employing the Separation of Concerns Principle – means that you should be able to completely replace the database (Persistence Layer) technology that you use without upsetting your core business logic (Domain Layer). By the same token, you should be able to remove the user interface (Presentation Layer) and replace it with some other technology, and your domain layer should still work.

## Step 2: Anticipating Change by Employing SoC

---

At this point, you may be wondering if you will ever truly need to worry about fundamentally replacing technologies within your application. The rule of thumb here is “never say never.” If decades of software development has taught one thing - it is that change is constant. In the last 20 years, there have been at least a dozen different display technologies, probably half a dozen different ways of accessing databases, including entirely new APIs with entirely new syntax and ways of accessing stored information. Also, this isn't really about swapping out entire layers even if that's something that you *could* do. Code reuse is a major byproduct of following SoC, but it's just that – a byproduct. Whenever you separate your concerns and use that as a guiding principle in everything that you do, it's going to help you organize your code in such a way that it can be grown to a massive codebase and yet still be maintainable. You will have a manageable codebase with a built-in guidance on where new code belongs in your application, based on what concern it is dealing with. Probably the worst thing that can happen in your application's lifetime is that it becomes so massive and resistant to change – fixing one problem creates a multitude of problems elsewhere – that it's simply easier to throw it all away and start again, from scratch.

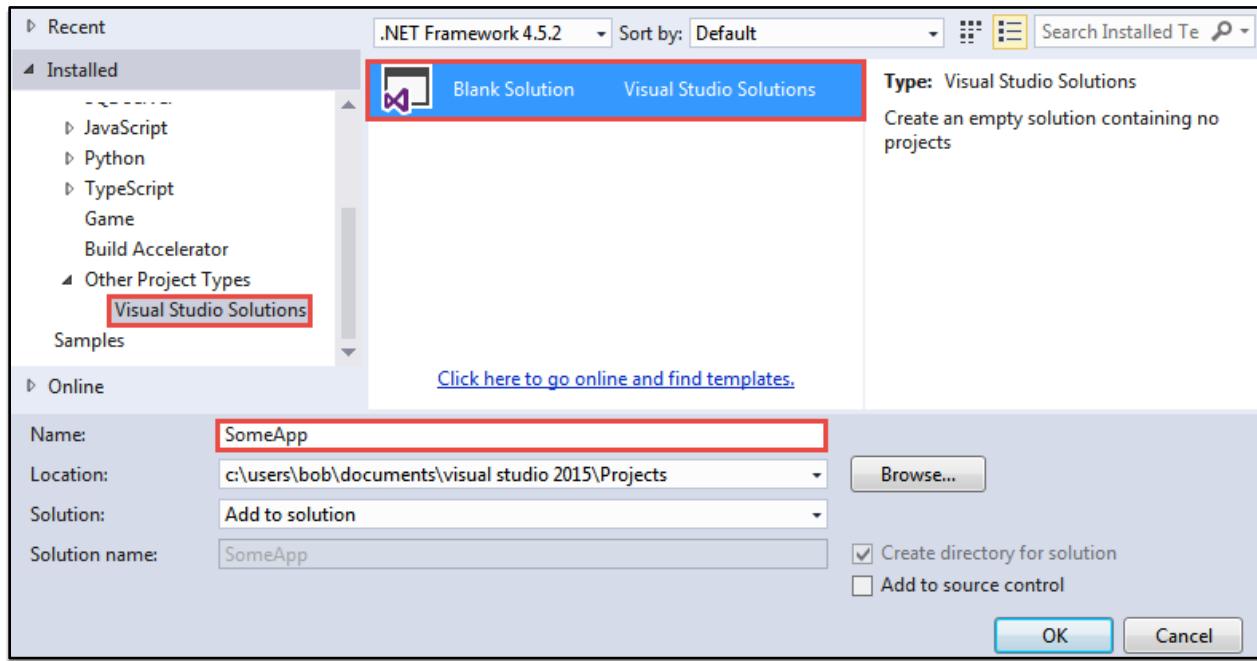
Now that you are sold on the value of SoC, let's start putting it in practice. In the next set of lessons, you will confront concepts related to SoC, such as error/exception handling and variable data storage systems (swapping out one data storage solution with another). Ideally, you would package your concerns separately, into different .NET assemblies that your code is compiled into. In the case of an ASP.NET Presentation Layer, perhaps there are many .aspx pages along with one or more .NET Assemblies/DLL's. Or perhaps, in the context of having a project within a solution, separating your layers out into separate Visual Studio projects will help your reinforce, and remind you of, the SoC you are trying to follow. From this point on, we're going to be illustrating this idea by being mindful of the given concern of the application and putting each given concern into its own project.

## Step 3: Create a Blank Solution to Demonstrate SoC

---

Here's the basic shape our employment of SoC will take. Starting with a blank solution, open a new project in Visual Studio, calling it "SomeApp". Start by selecting:

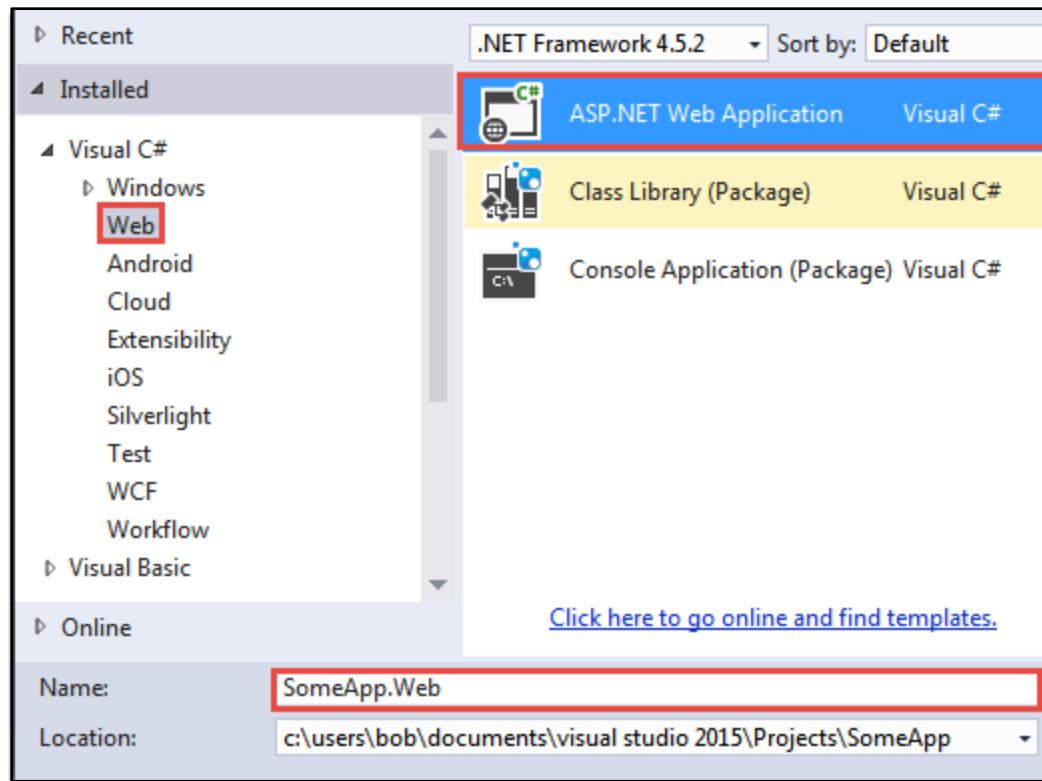
Templates > Visual C# > Other Project Types > Visual Studio Solutions



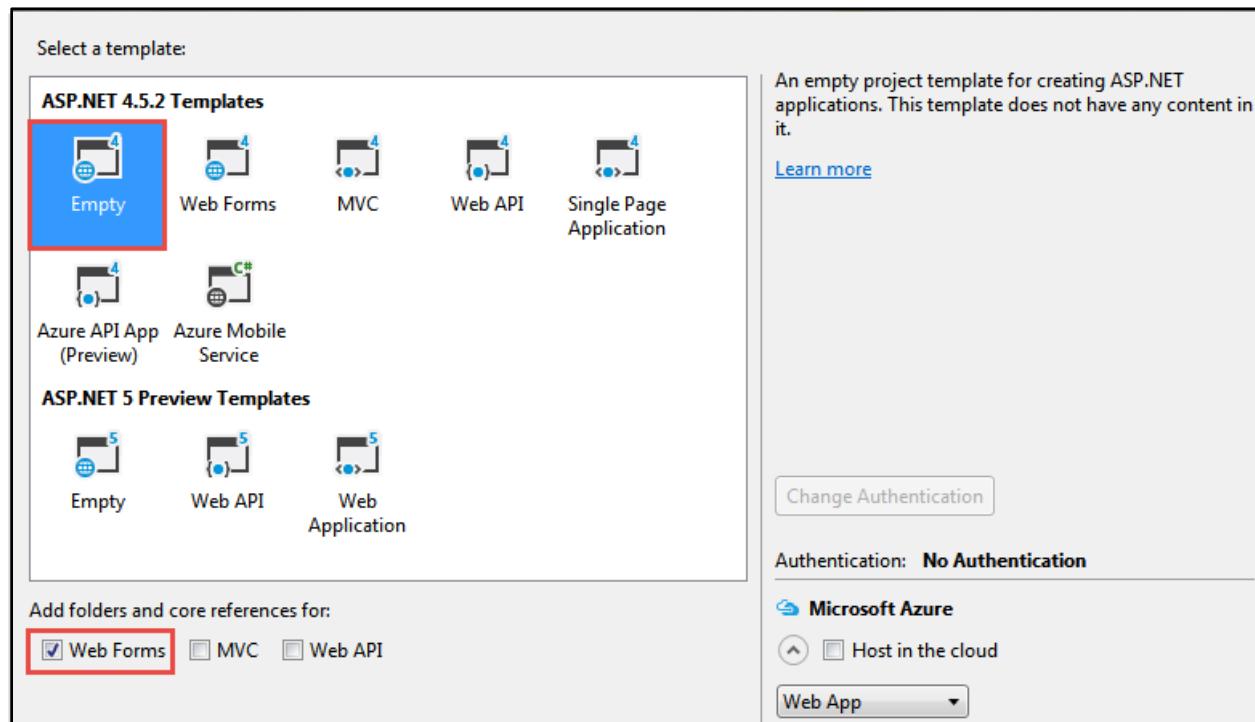
Once opened, right-click on the Solution Explorer and select from the menu:

Add > New Project

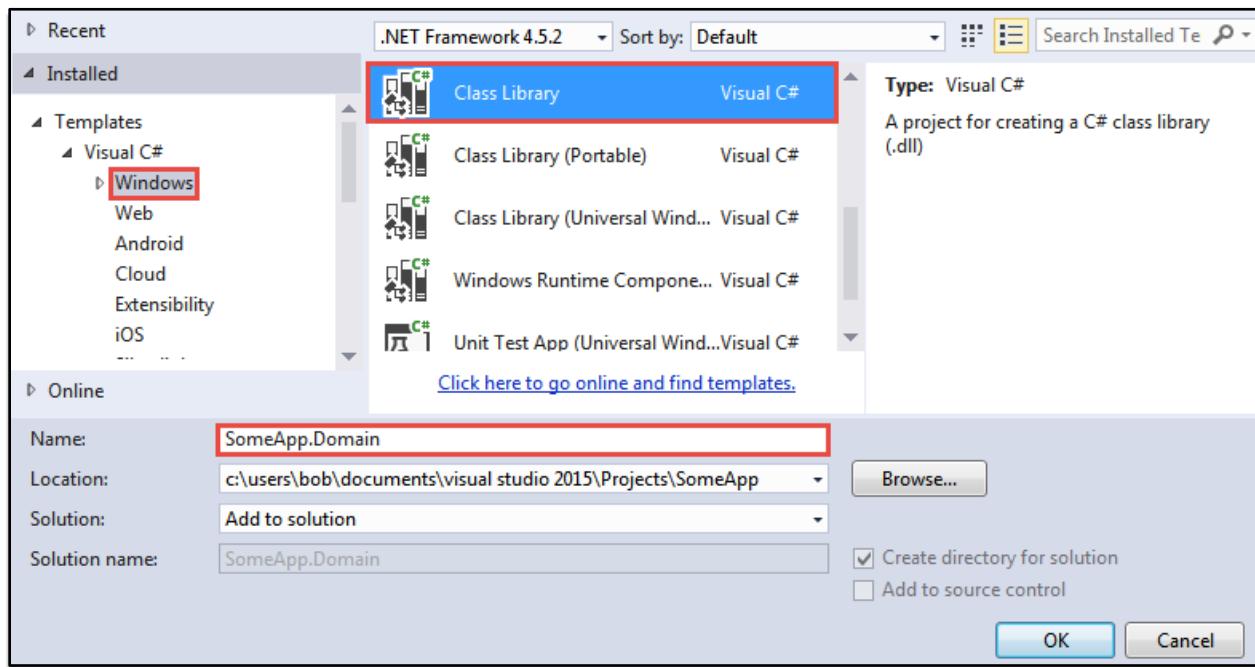
From here, create a new ASP.NET Web Application calling it SomeApp.Web:



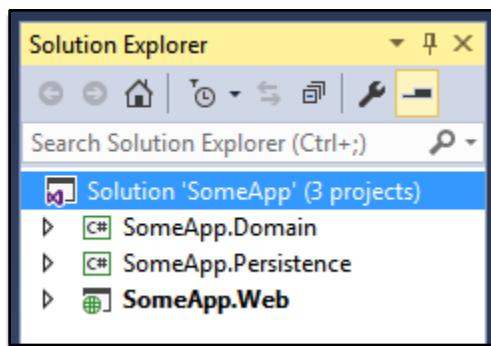
Make the project from an Empty template, and include Web Forms:



Now using those steps, add two more projects to the Solution, each being a Class Library and named "SomeApp.Domain" and "SomeApp.Persistence" respectively:



When you are done, you should have these three projects in your Solution Explorer:



This demonstrates basic SoC project structuring within your code into separate layers. You may add, at some point later, other layers relevant to your particular application architecture. For example there might be a Web Services Layer, a custom wrapper around some external API and so on.



Bob's Tip: an API wrapper would let you create a bridge between the code in your application that references an external API and the external API itself. Again, this would be to keep your codebase maintainable – wherever the wrapper is referenced throughout your code – even if the external API changes. For instance, if you were to integrate your app with a payment system, like PayPal or Stripe, you might create

056

# Understanding Exception Handling

In this lesson, we're going to talk about Exception Handling, which is a way of dealing with potential errors in your code. There are two basic types of errors you can encounter in programming, and they are:

- Compilation errors.
- Runtime errors.

## Step 1: Understanding Compilation vs Runtime Errors

---

Compilation errors are the most common and they are readily visible and relatively easy to correct. Basically, whenever the compiler refuses to run your application – whether you missed an essential syntax element, or tried to write an invalid operation – the compiler catches it and forces you to change it before it can even run the app. Runtime errors, on the other hand, occur in the form of exceptions that are raised while your application is already running. These errors cannot be caught by the compiler and, for that reason, can be much more difficult to track down. The Exception might be raised due to a variety of reasons, from a conditional block that can't perform, to attempting to operate on data that can't be parsed, to unsanitized user-input. These kinds of runtime errors are usually found through rigorous testing, running the application under many different conditions. However, there are other kinds of errors that a developer may not always be able to account for. For example, an Exception could also occur as a result of a dependency on some external system that the developer has no control over. An application may depend on a database server, a network server, a web service, something in the file system, and so on. These resources sometimes experience outages, or are just not available at the moment when the application needs them.

## Step 2: Understanding When to Employ Exception Handling

---

Exception Handling comes into play when a developer can't account for a possible Exception, and yet doesn't want the Exception to cause the entire app to crash. This "graceful degradation" can be accounted for – mitigating the impact on the user – by implementing structured Exception Handling, or "fail-safes" in the code. Unhandled Exceptions are those that are not accounted for and will result in the application terminating in some ungraceful way. If we're running in Debug mode, the environment will stop and switch from the web browser to Visual Studio, such as with this Overflow Exception we saw in lesson 009 when we tried to calculate, and store, a number larger than an `int` is allowed to hold:

```
int resultNumber;

checked
{
    resultNumber = firstNumber * secondNumber;
}

OverflowException was unhandled by user code
An exception of type 'System.OverflowException' occurred in CS-ASP_009.dll but
was not handled in user code

Additional information: Arithmetic operation resulted in an overflow.
```

## Step 3: The “Yellow Screen of Death”

If we weren't in Debug mode, but were running the application in a live environment, the end user would see what developers call “The Yellow Screen of Death,” which is essentially an error page that contains information meant for the developer, not the end user:

**Server Error in '/' Application.**

**Attempted to divide by zero.**

**Description:** An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and

**Exception Details:** System.DivideByZeroException: Attempted to divide by zero.

**Source Error:**

```
Line 24:             decimal wins = decimal.Parse(gamesWonTextBox.Text);
Line 25:             decimal total = decimal.Parse(totalGamesTextBox.Text);
Line 26:             decimal winningPercentage = wins / total;
Line 27:
Line 28:             result = string.Format("Winning Percentage: {0:P}",
```

**Source File:** C:\Users\Steve\Desktop\C# Fundamentals via ASP.NET\Code\CS-ASP\_056\Before\ExceptionHandling\ExceptionHandling\Default.aspx.cs    **Line:** 26

**Stack Trace:**

## Step 4: Try, Catch, and Finally Components of Exception Handling

Normally, we would want to account for these Exceptions with either a Global Exception Handler (which we will look at in the next lesson) or a Local Exception Handler that is written directly in the block of

code that might cause the Exception in question. These Exceptions are handled via a series of code blocks reminiscent of `if()` conditionals, taking the form of `try`, `catch()`, and `finally` where:

- (1) `try` represents the existing code that may cause the Exception.
- (2) `catch()` represents what happens in the event of the Exception being raised.
- (3) `finally` represents an action taken regardless of whether or not an Exception occurs.

```
public void SomeMethod()
{
    //code that runs previous to the try/catch

    1 try
    {
        //the code that you want to try to run
    }
    2 catch (System.IO.IOException e)
    {
        //what happens in the event of an exception,
        //such as printing out the caught error:
        resultLabel.Text = e.ToString();
    }
    3 finally
    {
        //the code that executes whether or not
        //an exception occurs
    }

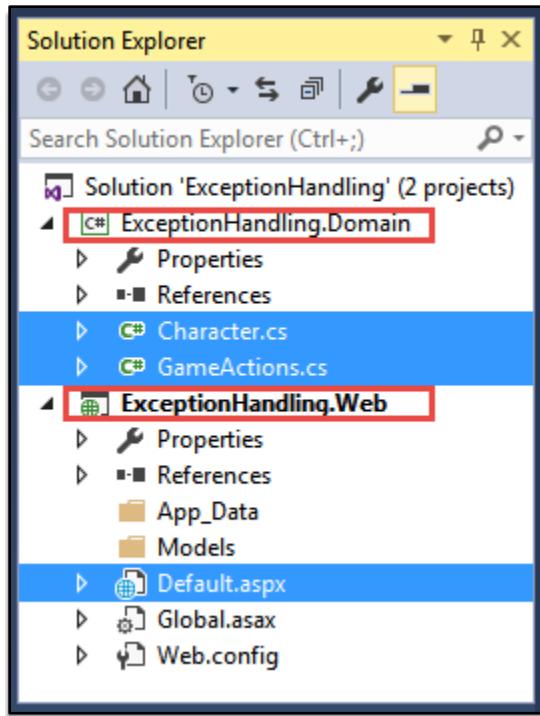
    //code that runs subsequent to the try/catch
}
```

It's worth noting that – similar to the `else if()` clause, sandwiched in the middle of `if()` statements – you can have as many `catch()` statements as you need to handle different Exceptions that might be raised.

## Step 5: Create a New Project

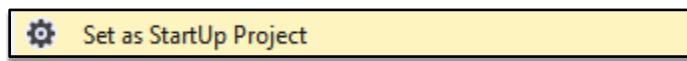
---

For this lesson, create a new ASP.NET project called "ExceptionHandling" and following the previous lesson's steps, include two separate projects for `ExceptionHandling.Domain`, and `ExceptionHandling.Web`. The Web project will be a typical ASP.NET project with a `Default.aspx` file, while the domain project will be an ordinary class library with two custom classes called `Character` and `GameActions`, respectively:

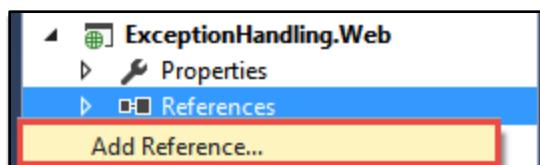


## Step 6: Set a Default Project as Application Entry Point

It's important to set the `ExceptionHandling.Web` project as the default project, so whenever you run the app its code is run first as the entry point to the rest of your application. To do this, right-click on the `ExceptionHandling.Web` project in the Solution Explorer and select from the menu "Set as StartUp Project." This highlights the project in bold within the Solution:



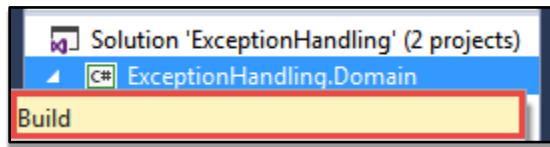
You will also need to add the Domain project as a reference to the Web project by right-clicking on "References" under `ExceptionHandling.Web` and selecting "Add Reference...":



Choose the Domain project from the projects available via the current Solution:

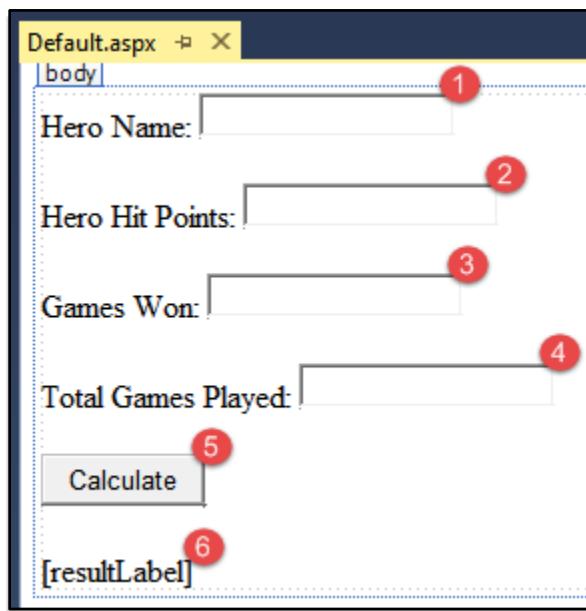


If you did not see this Reference available, be sure to first build the project by right-clicking on it and selecting "Build":



The Web project will have the following Server Controls and programmatic IDs:

- (1) heroNameTextBox
- (2) heroHitPointsTextBox
- (3) gamesWonTextBox
- (4) totalGamesTextBox
- (5) calculateButton
- (6) resultLabel



## Step 7: Create an Intentional Exception

---

The calculateButton\_Click() method will simply output a decimal result of division between gamesWonTextBox and total GamesTextBox, formatted as a win percentage:

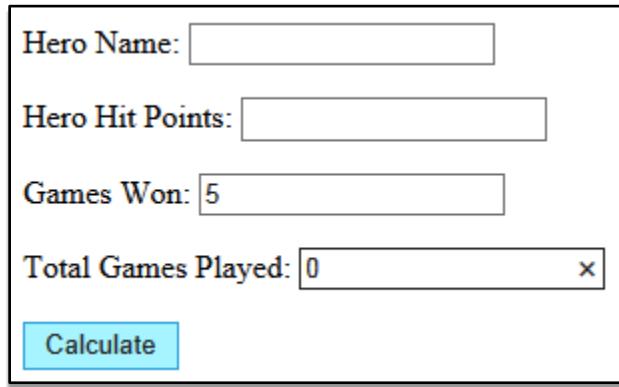
```
protected void calculateButton_Click(object sender, EventArgs e)
{
    string result = "";

    // Calculate percentage of wins:
    decimal wins = decimal.Parse(gamesWonTextBox.Text);
    decimal total = decimal.Parse(totalGamesTextBox.Text);
    decimal winningPercentage = wins / total;

    result = string.Format("Winning Percentage: {0:P}",
        winningPercentage);

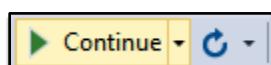
    resultLabel.Text = result;
}
```

Notice when you run the app, an Exception is raised if you try to divide by zero (this is because values of type decimal can't be divided by zero):



```
decimal winningPercentage = wins / total; ⚠ DivideByZeroException was unhandled by user code
```

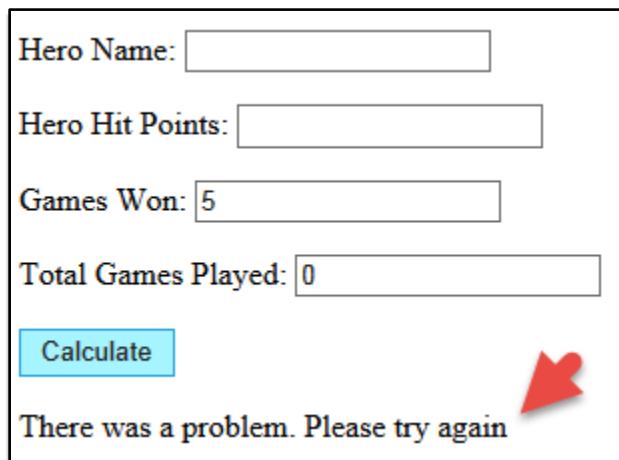
Next, click on the “Continue” button:



This would result in a “Yellow Screen of Death” if not properly handled within the code. That wouldn’t be acceptable because it will not provide any helpful information that the user can then act on and could actually provide information that could be used maliciously through a hack. Nor would it produce a result that looks like the rest of the application or website. With that in mind, let’s try to handle this Exception by wrapping it within a Try/Catch. You can initialize the code snippet for this by typing in “try” and hitting tab twice on your keyboard:

```
string result = "";  
  
try  
{  
    // Calculate percentage of wins:  
    decimal wins = decimal.Parse(gamesWonTextBox.Text);  
    decimal total = decimal.Parse(totalGamesTextBox.Text);  
    decimal winningPercentage = wins / total;  
  
    result = string.Format("Winning Percentage: {0:P}",  
        winningPercentage);  
}  
catch (Exception)  
{  
    result = "There was a problem. Please try again";  
}  
  
resultLabel.Text = result;
```

Now when you run the application and try to produce the same Exception, you get the custom user-facing error message instead:



## Step 8: Catching Generic and Specific Exceptions

---

The problem now is that the error message is too generic, and doesn't anticipate the specific error and tell the user what they should do to not produce the same error. To combat this, you can simply add another catch() statement that executes a reaction specific to the anticipated error:

```
catch (DivideByZeroException)
{
    result = "Please enter a value greater than zero for games played.";
}
```

It's important to note that order of catch() statements matters. The rule of thumb is to keep the most specific catch() statements above the generic ones. In the below example, the first catch occurs on basically any Exception that may occur, and since this Exception catches all possible exceptions, you will never even execute the more specific DivideByZeroException (hence the red underline):

```
catch (Exception)
{
    result = "There was a problem. Please try again";
}
catch (DivideByZeroException)
{
    result = "Please enter a value greater than zero for games played.";
}
```

Another, sometimes more elegant, way of handling many different Exceptions with the generic catch() is to reference the exact error message, provided as an input parameter and output that to the user or some internal logging utility:

```
catch (Exception ex)
{
    result = "There was a problem: " + ex.Message;
}
```

A screenshot of a web application interface. At the top, there are two input fields: 'Games Won:' containing '4' and 'Total Games Played:' containing '0'. Below these is a blue 'Calculate' button. At the bottom of the page, the text 'There was a problem: Attempted to divide by zero.' is displayed in a dark font.

Let's look at this a bit deeper by creating more code within our project. Start off by filling out the classes we created earlier in the Domain layer of our Solution:

```
namespace ExceptionHandling.Domain
{
    public class Character
    {
        public string Name { get; set; }
        public int HitPoints { get; set; }
    }
}

namespace ExceptionHandling.Domain
{
    public class GameActions
    {
        public static void Battle(Character attacker, Character defender)
        {
            Random random = new Random();
            int attackValue = random.Next(100);

            defender.HitPoints -= attackValue;
        }
    }
}
```

And then add some code that references these Domain objects within the existing try statement in calculateButton\_Click():

```
result = string.Format("Winning Percentage: {0:P}",
    winningPercentage);

var monster = new Character() { Name = "Zerg", HitPoints = 0 };
var hero = new Character() { Name = "Buzz", HitPoints = 5 };
GameActions.Battle(hero, monster);
result += string.Format("{0} attacked {1} leaving him with {2} hit points.",
    hero.Name,
    monster.Name,
    monster.HitPoints);
```

## Step 9: Throwing an Exception

---

There's an intentional problem in this code and that is the monster object is initialized with having zero HitPoints. When the battle sequence starts, with the call to GameActions.Battle(), we should create a way to throw our own handled Exception, because it shouldn't be possible to battle an enemy

that technically isn't regarded as "currently alive" in the game logic. To accomplish this, write the following within the Battle() method:

```
if (attacker.HitPoints <= 0)
    throw new ArgumentOutOfRangeException("Attacker is already dead.");

if (defender.HitPoints <= 0)
    throw new ArgumentOutOfRangeException("Defender is already dead.");

Random random = new Random();
int attackValue = random.Next(100);

defender.HitPoints -= attackValue;
```

What we're essentially doing is forcing an Exception to be raised under a condition that we determine. We chose the ArgumentOutOfRangeException as it's available to use via the .NET Framework and allows us to append a custom message as an input parameter to its constructor. Although we haven't accounted for this specific Exception in the original Try/Catch, it will be caught by the generic Exception and will append our custom message to its existing message:

Hero Name:

Hero Hit Points:

Games Won:

Total Games Played:

**Calculate**

There was a problem: Specified argument was out of the range of valid values.  
**Parameter name: Defender is already dead.**

## Step 10: Catching the Thrown Exception at Method Call

---

To make the error message more specific we can just add this particular Exception to the top of our catch() hierarchy:

```
catch (ArgumentOutOfRangeException)
{
    result = "Either the attacker or the defender are already dead.";
}
```

It's possible to create our own custom Exceptions which inherit from established Exceptions but are specific to the code we are creating. For instance, we could create a DefenderAI readyDead Exception or an AttackerAI readyDead Exception with error messages specific to those cases. We will be looking at this precise scenario in ensuing lessons.

We have yet to cover how to use the `finally` element in the Try/Catch sequence. This part executes no matter what happens in the Try/Catch and is usually used to implement "clean-up" code at the very end of the process. For example, if you had an open database connection you might put some code in the `finally` block that is responsible for closing all open connections, freeing up that resource. This will become more important once we work with external resources, so for now just keep that rule of thumb in mind.



Bob's Tip: Exception Handling is extremely useful, but don't take it for granted! There was a time, back in the days when procedural programming dominated, when programmers would return integer values from method calls to represent the success or failure of that particular method call. It was a very archaic system that was difficult to consistently implement and was often cryptic to the user and even the developer!

## Step 11: Understanding Where and When To Use Exception Handling

---

At this point, it's natural to ask where and when you should use Try/Catch Exception Handling in your code. The rule of thumb here is to employ it wherever you have code that you have no control over, such as an external resource or user-determined input. When it comes to user input, it is still preferable to validate against known problems within the code itself. For example, if we require a numerical value entered by the user but we know the user might enter a string that can be validated in the code through simple evaluations, or even by using built-in validation Server Controls. However, you will still probably want to use some Exception handling here as its unlikely you will be able to account for all possible user-input (even if you are confident you know every possible permutation ahead of time).

Also, you should, ideally, wrap Exceptions around only the code that might throw the Exception rather than entire blocks of code. You should also wrap Try/Catch around method calls that are known to throw Exceptions, such as in the attacker/defender example above. The final bit of advice on employing Exception Handling is to log each Exception so that you, the developer, can peer into the problems that users are experiencing without them having to even contact you and try to describe it for you. There are many ways that you can save the Exception data into a file or log file, and although that is outside of the scope of this lesson you should be aware of its importance as you delve deeper into the subject matter.

# Understanding Global Exception Handling

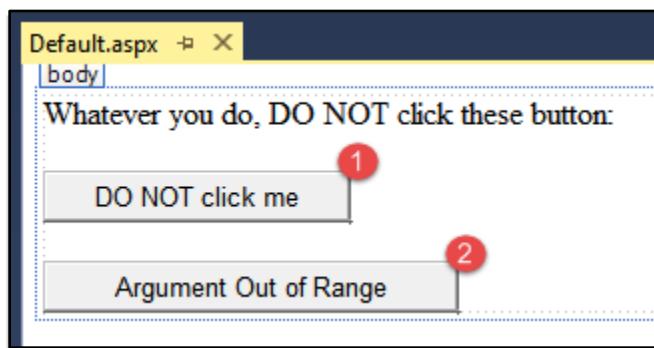
If an Exception is not handled in any given method we create, it has the effect of “bubbling up” and producing a cascading effect for each method that doesn’t handle the Exception. In other words, if the Exception is not handled directly, in the method that causes the Exception, the error cascades to the method that *called* that method – and if it’s not handled there, to the method that called *that* method, and so on. Eventually, this will arrive at the point where the cascade of unhandled Exceptions stops within some calling method in the .NET Framework itself, outside of your control. At which point, the ASP.NET Runtime handles the Exception in the most basic way possible, such as a very generic “Yellow Screen of Death” – which is undesirable because it is out of our control and does not help us, nor the end user.

## Step 1: Create a New Project

---

In this case, we can employ a Global Exception Handling “catch all” as a safety net in our code so that we never get to the point where we let the ASP.NET Framework determine how to handle Exceptions we may have missed in *our* code. To demonstrate this, create a new ASP.NET project called “GlobalExceptionHandling” and in the *Default.aspx* include the following Server Controls and programmatic IDs:

- (1) okButton
- (2) argumentOutOfRangebutton



In the *Default.aspx.cs*, include the following code for `okButton_Click()` and `argumentOutOfRangebutton_Click()`, as well as an external helper method that gets referenced in `okButton_Click()`:

```

public partial class Default : System.Web.UI.Page
{
    protected void okButton_Click(object sender, EventArgs e)
    {
        HelperMethods.SomeMethod();
    }

    protected void argumentOutOfRangeButton_Click(object sender, EventArgs e)
    {
        throw new ArgumentOutOfRangeException();
    }
}

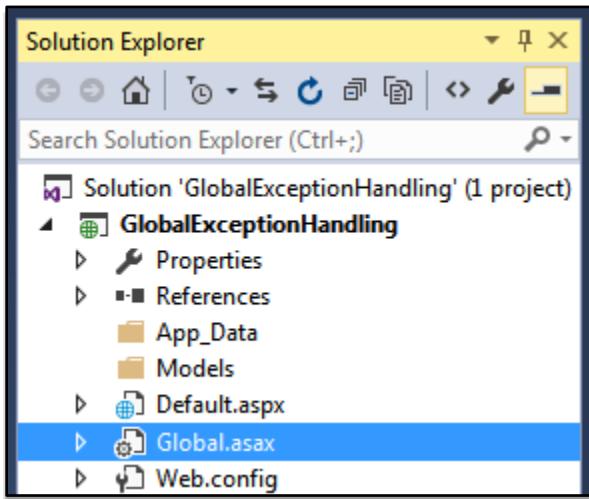
public class HelperMethods
{
    public static void SomeMethod()
    {
        throw new Exception("I told you not to do it!");
    }
}

```

## Step 2: Employ Global Exception Handling via Global.asax

---

The obvious way to handle these (intentional) Exceptions in our code is wherever the events that are tied to these methods are called. We don't really have direct access to these events so instead we can impose a Global Exception Handler – that handles all errors caught at the application level – via the *Global.asax* file in our project:



We do this by defining a method called `Application_Error()`, which is tied into an event that fires just before the exception would "bubble up" outside of our control throwing a "Yellow Screen of Death."

Notice how we reference the `InnerException` property, telling us what the initial Exception was before it turned into an `HttpException` (which is the Exception type that `ex` will end up referencing):

```
namespace GlobalExceptionHandling
{
    public class Global : System.Web.HttpApplication
    {
        void Application_Error(object sender, EventArgs e)
        {
            //grab the exception
            Exception ex = Server.GetLastError();

            //take the part of the exception that is relevant
            var InnerException = ex.InnerException;
        }

        protected void Application_Start(object sender, EventArgs e)...
    }
}
```

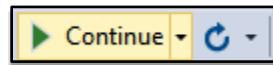
We can then write a custom server response message, detailing the error (similar to when you see a server output 404 – Page Not Found, for example). After that we have to call `Server.ClearError()` to avoid getting the “Yellow Screen of Death”:

```
var InnerException = ex.InnerException;

Response.Write("<h2>Something bad happened...</h2>");
Response.Write("<p>" + InnerException.Message + "</p>");

Server.ClearError();
```

Now, when you run the application you will still see the Exception in Visual Studio, but if you click “Continue” it will output in the browser:



**Something bad happened...**

I told you not to do it!

## Step 3: Transferring to an Error Page

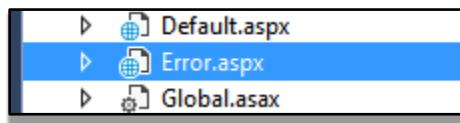
---

What we can also do is look for a particular type of Exception and then serve up a different web page to handle it:

```
var InnerException = ex.InnerException;

//handle a specific type of error differently
if (InnerException.GetType() == typeof(ArgumentOutOfRangeException))
{
    Server.Transfer("Error.aspx");
}
```

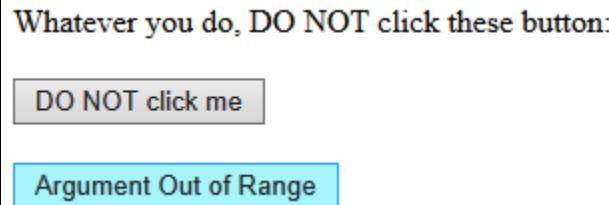
Add a new .aspx file to the project, calling it "Error," and set it up with a simple message and resultLabel:



In *Error.aspx.cs* write the following for the `Page_Load()` method, grabbing the error and outputting different messages depending on if the returned Message was null (it would be null if you navigated to the page directly, for example):

```
protected void Page_Load(object sender, EventArgs e)
{
    var ex = Server.GetLastError();
    if (ex != null)
    {
        resultLabel.Text = "Error: " + ex.InnerException.Message;
    }
    else
    {
        resultLabel.Text = "Something went horribly wrong, but it wasn't your fault.";
    }
}
```

Now when you run *Default.aspx* and click the “Argument Out of Range” button, you will see an error output:



## Don't Panic

Error: Specified argument was out of the range of valid values.

Notice also how the URL shows that this is coming from the *Default.aspx* page, however, we know that it is actually being served up from *Error.aspx* (via `Server.Transfer`):

<http://localhost:14391/Default.aspx>

058

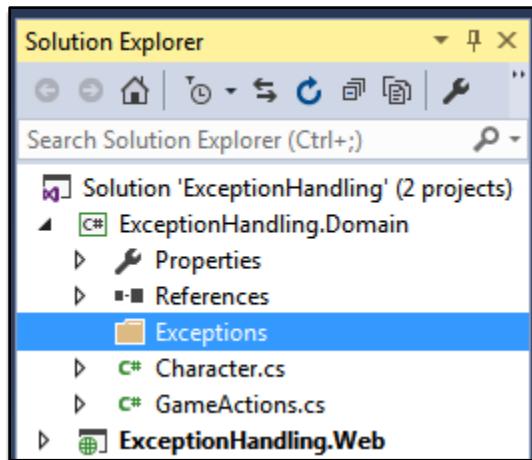
# Understanding Custom Exceptions

This lesson will show you how to create custom Exceptions, which means creating your own class that can recognize, and allow you to handle a particular type of exception that you define within that class. This lesson, continues on from where we left off in lesson 056, so begin by loading that project.

## Step 1: Add an Exceptions Folder

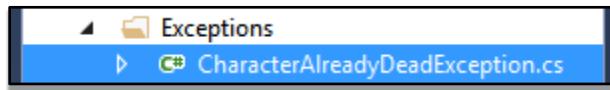
---

The first thing you will want to do is add a folder called "Exceptions" within the Domain layer, and store all of the custom Exception classes in this folder that relate to this layer:



This is a good rule to follow for code organization, as well as formatting an elegant namespace of `ExceptionHandling.Domain.Exceptions`. Now let's add a class called "CharacterAlreadyDeadException" to the "Exceptions" folder by right-clicking it and selecting from the menu:

Add > Class



## Step 2: Create a Custom Exception with Inheritance

---

To create a custom Exception, all we have to do is have the class inherit from the broader Exception class found in system.Exception. You achieve this inheritance by simply placing a colon after the class name, and then referencing the name of the class that you want to inherit from. This quietly imports all of the properties and methods you would find in the inherited class:

```
namespace ExceptionHandling.Domain.Exceptions
{
    public class CharacterAlreadyDeadException : Exception
    {
    }
}
```

We can add additional methods and properties to this class, but it isn't necessary. Now, in the GameActions class, let's update the exceptions being thrown in the Battle() method to reflect our custom exception:

```
if (attacker.HitPoints <= 0)
    throw new CharacterAlreadyDeadException();

if (defender.HitPoints <= 0)
    throw new CharacterAlreadyDeadException();
```

## Step 3: Reference the Custom Namespace

---

This reference won't be found by the compiler until you add the using statement for it at the top of this Class file:

```
using ExceptionHandling.Domain.Exceptions;
```

Make sure to build this project, and then move on to the *Default.aspx.cs* file where we will want to catch this Exception where the call is made to the Battle() method in the existing Try code-block:

```
catch (CharacterAlreadyDeadException ex)
{
    result = "Either the hero or the monster is dead";
}

catch (Exception ex)
{
    result = "There was a problem: " + ex.Message;
}
```

Now when you run the application you will see the thrown custom Exception that is caught here:

Games Won: 4

Total Games Played: 5

Calculate

Either the hero or the monster is dead

## Step 4: Create a Custom Error Message

---

Now that we have our own custom Exception, we can make a much more specific error message. Go back to the custom class and add a property that stores the name of the character passed into the constructor wherever we end up throwing the Exception:

```
public class CharacterAlreadyDeadException : Exception
{
    public string CharacterName { get; set; }

    public CharacterAlreadyDeadException(string CharacterName)
    {
        this.CharacterName = CharacterName;
    }
}
```

Now that the constructor takes an argument, let's supply it where it's called in the GameActions class:

```
if (attacker.HitPoints <= 0)
    throw new CharacterAlreadyDeadException(attacker.Name);

if (defender.HitPoints <= 0)
    throw new CharacterAlreadyDeadException(defender.Name);
```

Now when we reference this passed in CharacterName property value, we can output a more specific error message:

```
catch (CharacterAlreadyDeadException ex)
{
    result = "Problem: " + ex.CharacterName + " was already dead.";
}
```

Games Won:

Total Games Played:

**Calculate**

Problem: Zerg was already dead.

059

# Creating a Database in Visual Studio

During the lessons that covered Application Architecture, we referred to the three typical layers of concern including Domain, Presentation, and Persistence. We haven't had the chance to cover persistence in detail, but now we will be able to demonstrate it, when using a database to hold long-term, persistent data. A database can run locally or remotely.

## Step 1: Create a New Project

---

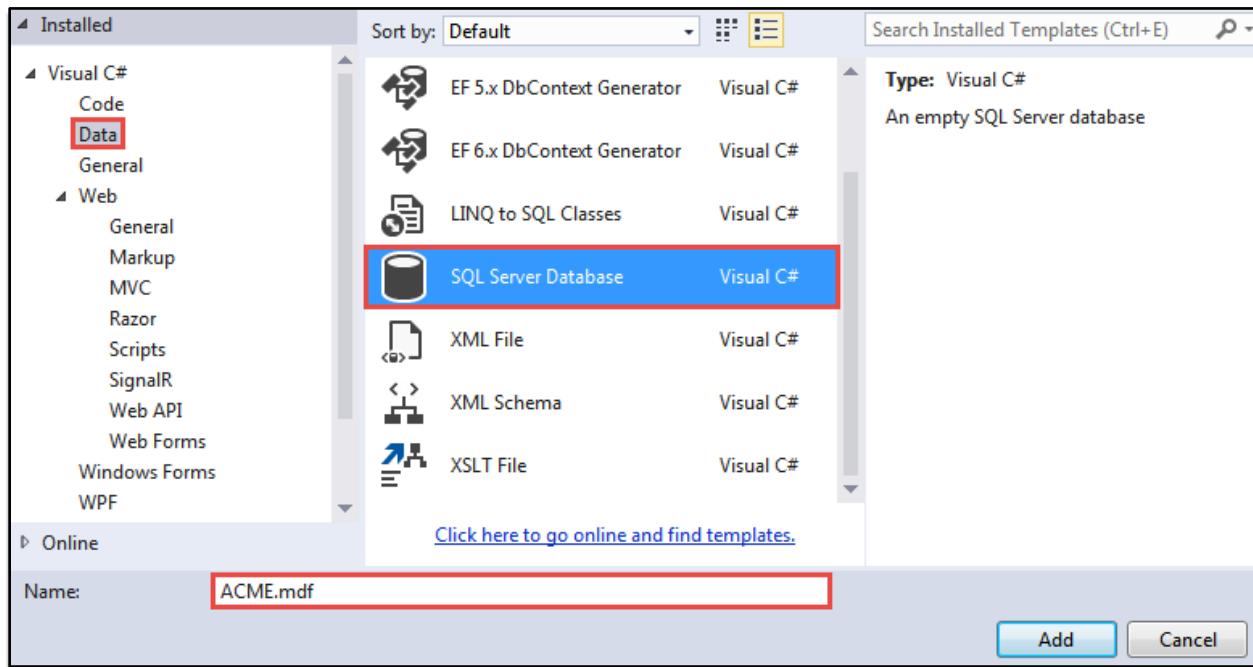
For this lesson, create a typical ASP.NET application, called it "LocalDBExample." Right-click the project name in the Solution Explorer and select from the menu:

Add > Item

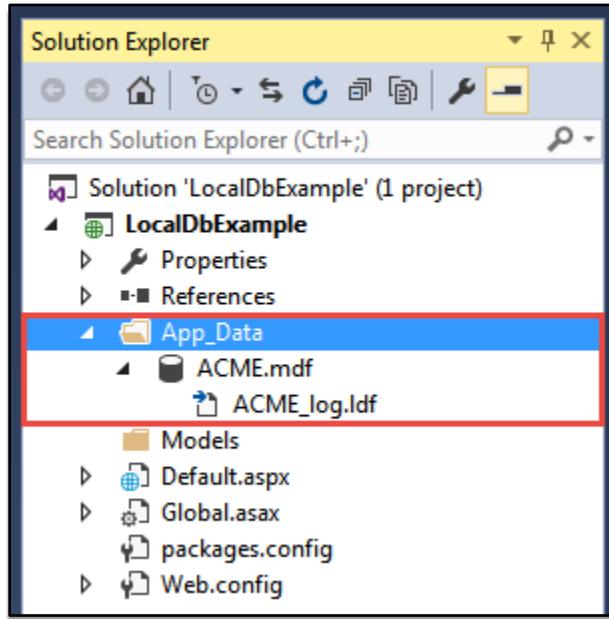
## Step 2: Create and Add a Database

---

From the resulting dialog add an SQL Server Database, found amongst the "Data" templates, and call it "ACME.mdf" to signify a company name:



After that, Visual Studio will ask you whether or not you want to create an "App\_Data" folder to store this database within your project. Agree to that and then you will see the database in the Solution Explorer:



## Step 3: Manage the Database via the Server Explorer Window

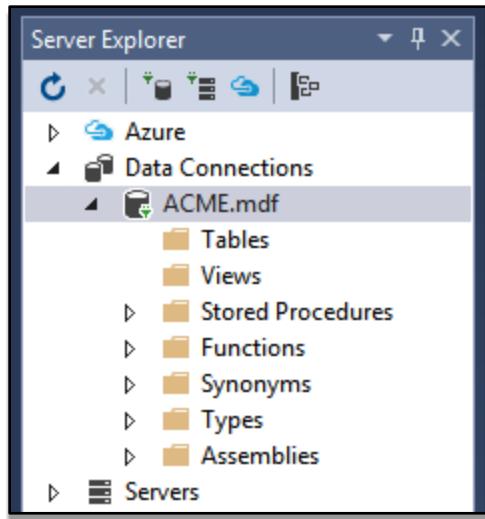
---

Right-click on the "ACME.mdf" file and select "Open" to view the contents of the database in the Server Explorer Window:



If the Server Explorer Window is not visible, access it through the keyboard shortcut (Ctrl+Alt+S) or the Visual Studio menu:

View > Server Explorer



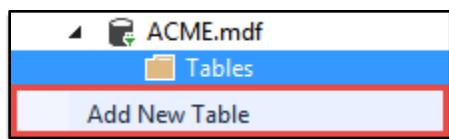
## Step 4: Design a Database Table

---

From here, we can start designing a table for storing values. You can think of a table as the rough equivalent of an Excel spreadsheet where you have rows of information, as well as columns and headers, that connote the meaning of that structured information. Also, notice how there is a green "plug" icon that indicates a connection has been established to the database within your project. If you do not see this indicator, it could mean that the connection is not being made in which case you will have to troubleshoot it or wait a few seconds for Visual Studio to make the connection:



From here, right-click on "Tables" and select "Add New Table":



This brings us to the Table Design view. From here change the table name to (1) "Customers," the Id to (2) "CustomerId," and also change the Data Type to (3) "uniqueidentifier" which lets us use a unique GUID for each customer in the database. Also, keep "Allow Nulls" unchecked to ensure that each row of customer data is populated with values:

The screenshot shows the SQL Server Management Studio (SSMS) interface. The top bar has tabs for 'dbo.Customers [Design]', 'Default.aspx', and 'packages.config'. Below the tabs, there's an 'Update' button and a 'Script File' dropdown set to 'dbo.Table.sql\*'. The main area is a grid for the 'Customers' table with columns: Name, Data Type, Allow Nulls, and Default. A row for 'CustomerId' is selected, highlighted with a blue background. Red circles with numbers 2 and 3 are placed on the 'Name' cell ('CustomerId') and the 'Allow Nulls' checkbox respectively. To the right of the grid is a 'Keys' pane showing one primary key constraint named '<unnamed>' (Primary Key, Clustered). Below the grid is a toolbar with 'Design' and 'T-SQL' buttons, and the 'T-SQL' button is selected. The T-SQL pane contains the following code:

```

CREATE TABLE [dbo].[Customers]
(
    [CustomerId] UNIQUEIDENTIFIER NOT NULL PRIMARY KEY
)

```

## Step 5: Understanding Database Data Types

Let's round out our table with some more data points that can store information about our customers. We chose "varchar" as the Data Type because it stores Unicode characters – meaning that we can use non-English characters if we have international customers. The value in the parentheses after varchar signifies the maximum allowed characters ("MAX" means there is virtually no limit). Also, checkmark "Allow Nulls" for the Notes data point since it is not essential to have filled out for each and every customer:

	Name	Data Type	Allow Nulls
CustomerID	CustomerId	uniqueidentifier	<input type="checkbox"/>
	Name	varchar(50)	<input type="checkbox"/>
	Address	varchar(50)	<input type="checkbox"/>
	City	varchar(50)	<input type="checkbox"/>
	State	varchar(50)	<input type="checkbox"/>
	PostalCode	varchar(50)	<input type="checkbox"/>
	Notes	varchar(MAX)	<input checked="" type="checkbox"/>



Bob's Tip: there's probably no country in the world that would have a potential of 50 characters for a postal code, but we're going to go ahead and leave that now. Typically, however, you would want to restrict the stored data only to the precisely necessary type, and number of characters, needed to represent all possible values.

## Step 6: Run Built-In Methods to Populate Table Entries

---

Next, under the “Default” entry for the CustomerId, you can run methods available to you. In this case, we will want to run a built-in method to SQL Server called newid(), which will ensure that if we don't supply a GUID whenever we create a new customer, one will be created for us.

	Name	Data Type	Allow Nulls	Default
Customer	CustomerId	uniqueidentifier	<input type="checkbox"/>	newid()

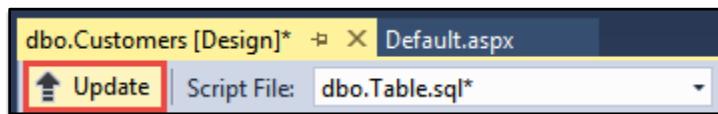
## Step 7: Save Table Design Query or Update it Immediately

---

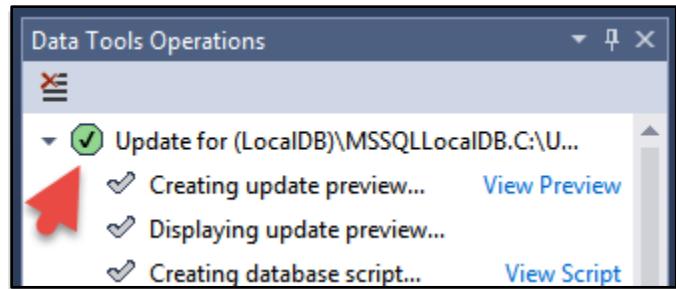
If we were to try to save our changes right now, Visual Studio would try to save this script that we “designed” here:

```
CREATE TABLE [dbo].[Customers]
(
    [CustomerId] UNIQUEIDENTIFIER NOT NULL PRIMARY KEY DEFAULT newid(),
    [Name] VARCHAR(50) NOT NULL,
    [Address] VARCHAR(50) NOT NULL,
    [City] VARCHAR(50) NOT NULL,
    [State] VARCHAR(50) NOT NULL,
    [PostalCode] VARCHAR(50) NOT NULL,
    [Notes] VARCHAR(MAX) NULL
)
```

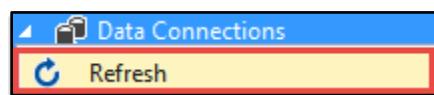
This might be preferable in a development environment where you may want to easily change the database – for, say, testing purposes – but for now let's just build the database using these settings defined by the script we created. You can do that by clicking on the “Update” button:



Next, click on “Update Database” and if everything was done correctly it should have successfully created the database:

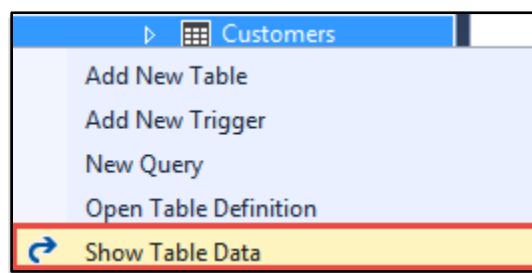


You may have to right-click on "Data Connections" and then "Refresh" to see the changes in your database:



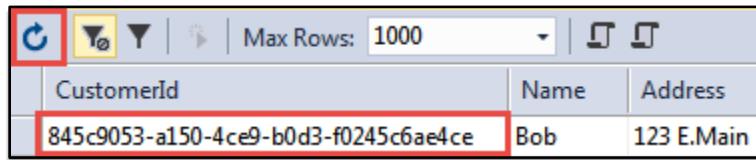
## Step 8: Manually Enter Table Data Values

Back in the Server Explorer window, right-click on "Customers" and select "Show Table Data":



Right now there are no values entered for each table element, so you can supply them right here if you wish:

At first glance, it may seem concerning that we are allowed to have a NULL CustomerId, however the data hasn't actually been populated until you hit the refresh button, at which point the default newid() method will return a valid GUID:



CustomerId	Name	Address
845c9053-a150-4ce9-b0d3-f0245c6ae4ce	Bob	123 E.Main

That's the basics of creating a database in Visual Studio and in the next lesson we will look at how to access the database from within our C# code.

060

# Creating an Entity Model in Visual Studio

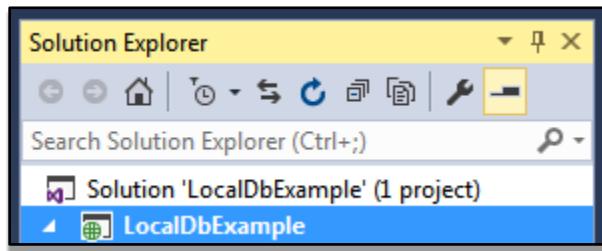
In the previous lesson, we created an SQL database running on a local server. What we want to do now is use an API within the .NET Framework, called the Entity Framework, to access/update that data via C# code. The Entity Framework is a subset of the entire ADO.NET API. It's a newer aspect of the larger ADO.NET API that has a lot of the momentum when it comes to handling databases but is certainly not the only way to access data in an SQL Server database. What the Entity Framework does is it serves up what's called an "Object-Relational Mapper." Simply stated a mapper takes the data from the database and maps it to native C# structured classes and properties – making it easier to work with in code.

## Step 1: Refer to Previous Project

---

This lesson is based on what we created in the previous lesson, so go ahead and start the project from where we left off. First, right-click on the project name "LocalDbExample" in the Solution Explorer and from the menu select:

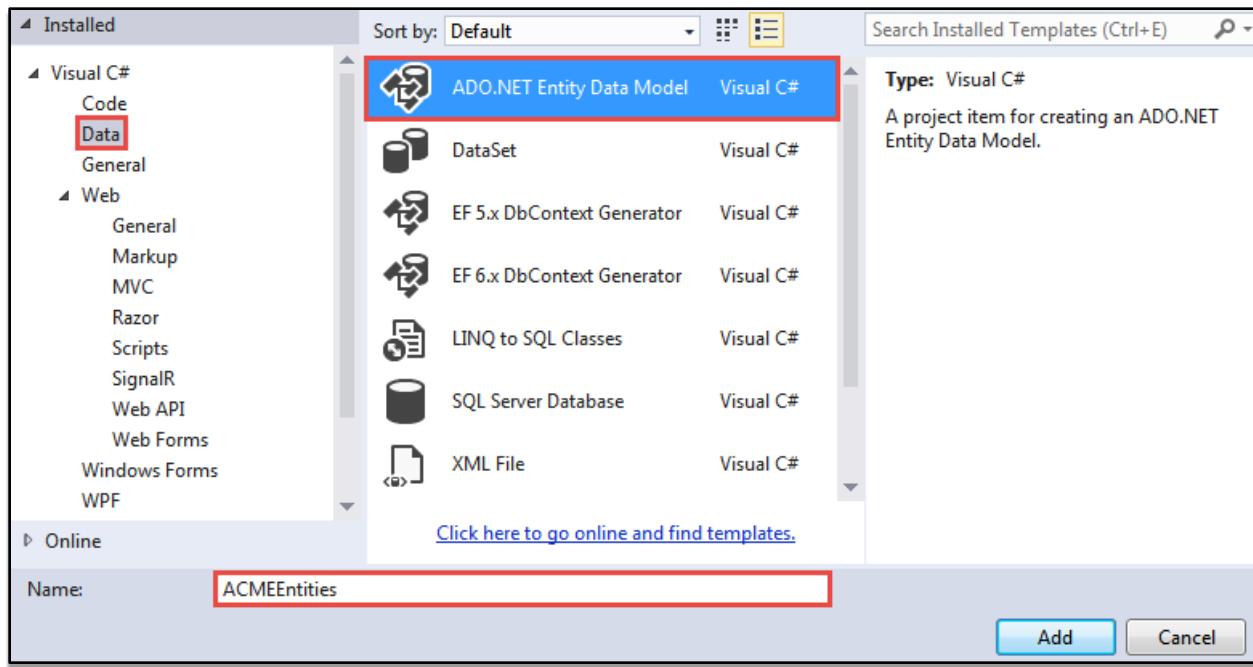
Add > New Item



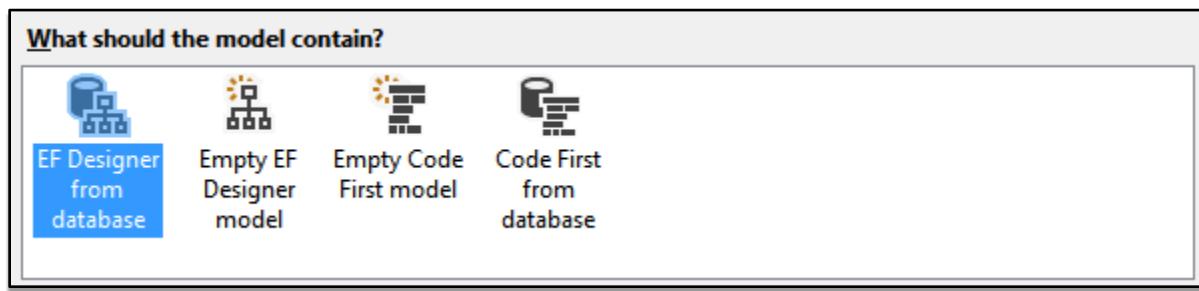
## Step 2: Add an Entity Data Model to the Project

---

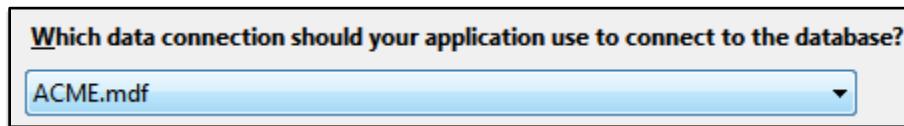
From here, go to the Data templates and select ADO.NET Entity Data Model, calling it "ACMEEntities":



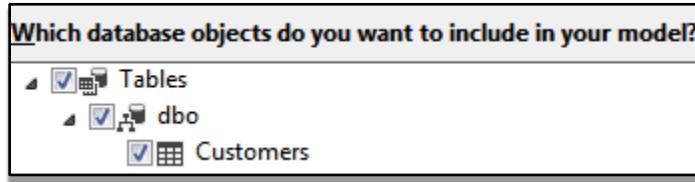
Next, select the option to model an “EF Designer from database,” since we started with a database and want to create an Entity model from it, rather than first create a model that can then applied to a database:



Then click “Next” on the following screen to simply create a “connection string” to the database you want to have the Entity model based upon. A connection string is held in a file that contains connection details required for an API, like the Entity Framework, to know where and how to go to connect to a database:



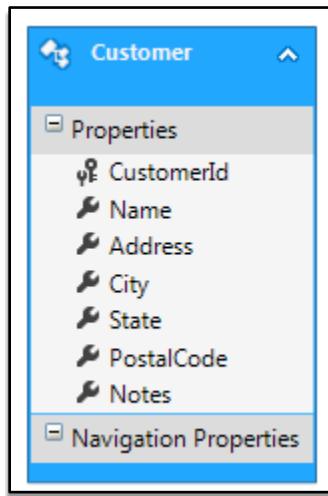
And in the following screen, checkmark the database table that you want to include in your Entity model. Since we only have one table – Customers – select that one:



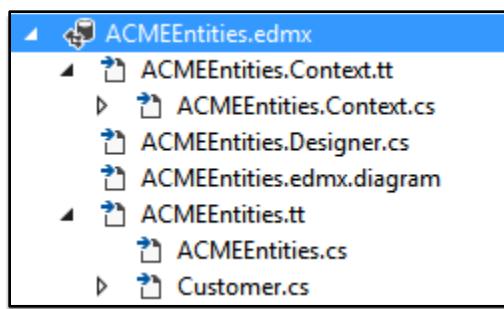
## Step 3: Understanding the Visual Entity Model and EDMX file

---

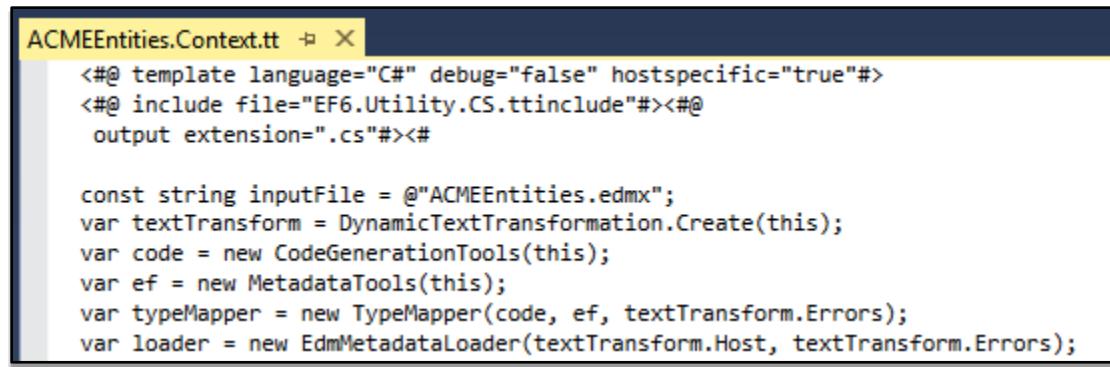
After that you may get a warning whether or not you want to run this and click "OK" to that. You will then be shown a visual model for the table and its data elements. If you had a variety of tables, they would all be represented here, and may even show connections between them wherever there is a relationship between the tables:



This visual model is available via the "ACMEEntities.edmx" file in the Solution Explorer:



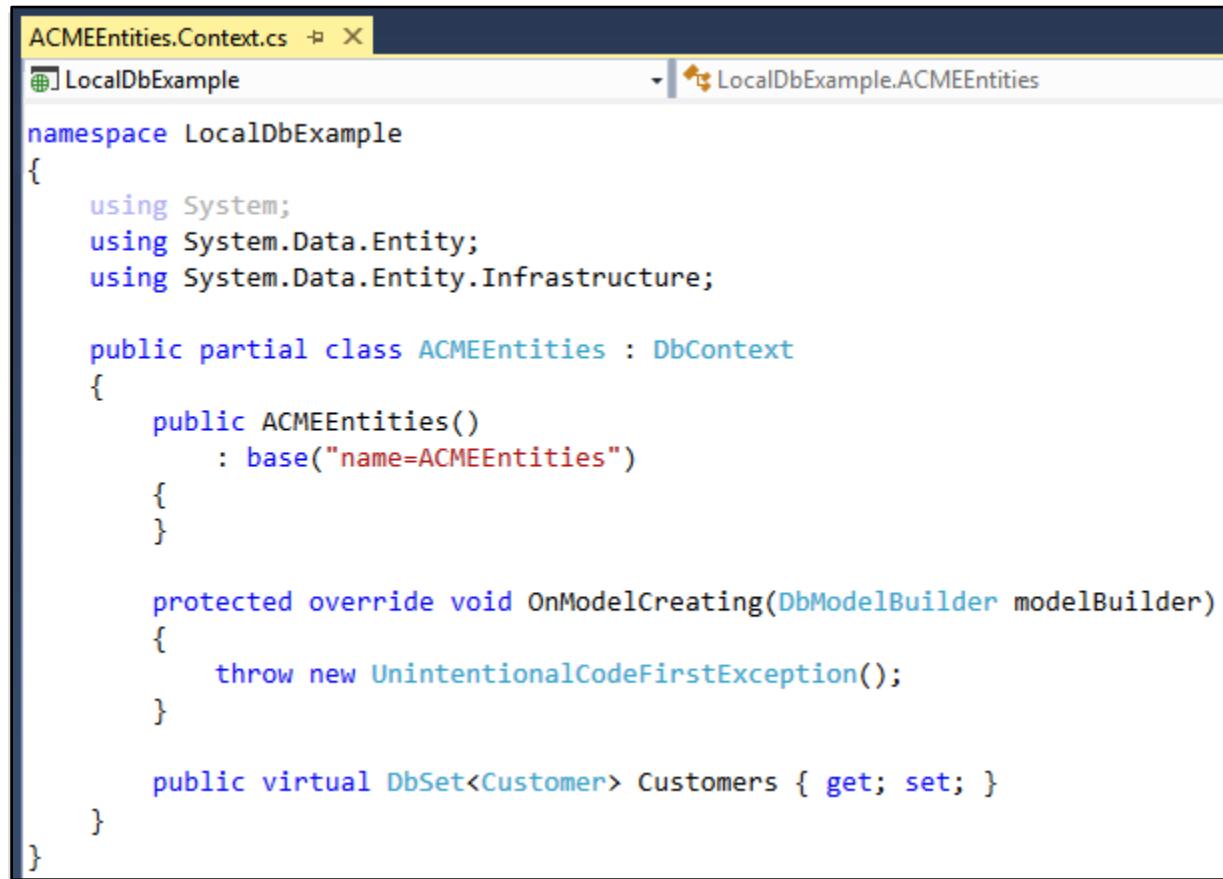
The `.tt` files are generated as template files that will themselves generate a series of classes.



```
<#@ template language="C#" debug="false" hostspecific="true"#=>
<#@ include file="EF6.Utility.CS.ttinclude"#=><#@ output extension=".cs"#=><#>

const string inputFile = @"ACMEEntities.edmx";
var textTransform = DynamicTextTransformation.Create(this);
var code = new CodeGenerationTools(this);
var ef = new MetadataTools(this);
var typeMapper = new TypeMapper(code, ef, textTransform.Errors);
var loader = new EdmMetadataLoader(textTransform.Host, textTransform.Errors);
```

In this case, it'll create a Context class, which acts as a connection to the database. When we ask the database for something, the code that's generated here in this class will house the logic necessary to go out and connect to the database, and perform the actions that we've requested:



```
ACMEEntities.Context.cs  X
LocalDbExample          LocalDbExample.ACMEEntities

namespace LocalDbExample
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class ACMEEntities : DbContext
    {
        public ACMEEntities()
            : base("name=ACMEEntities")
        {
        }

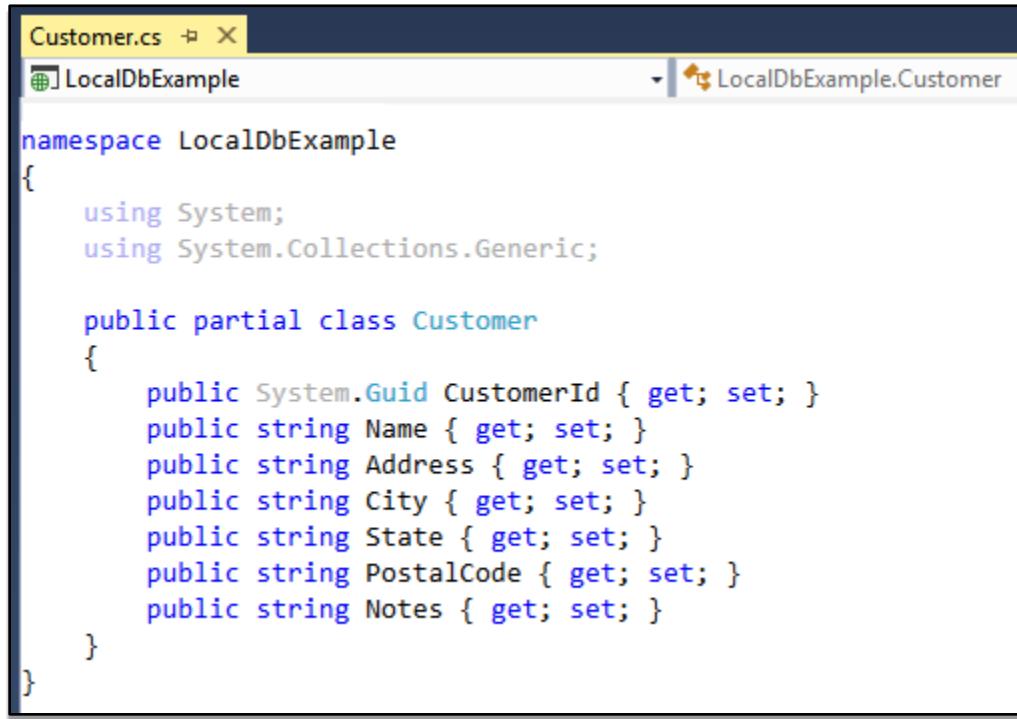
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public virtual DbSet<Customer> Customers { get; set; }
    }
}
```

## Step 4: Entity Generated Class Counterparts For Table Data

---

Meanwhile, the *Customer.cs* class was also generated, taking the various data points in the Customer table and converting them into suitable properties. The Entity Framework understands how to convert between .NET data types (string, GUID, and so on), and SQL Server data types (varchar, uniqueidentifier, and so on):



```
Customer.cs  X
LocalDbExample                                     LocalDbExample.Customer

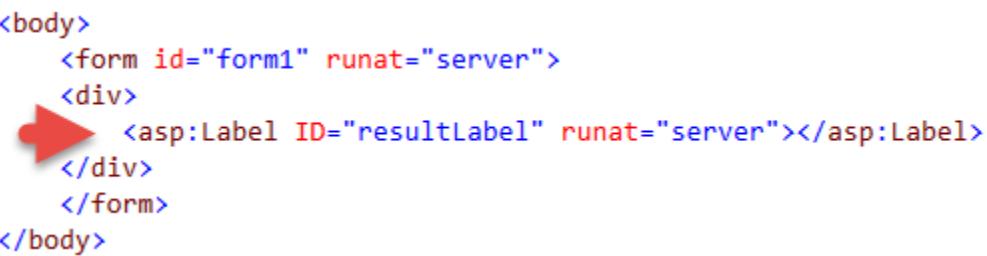
namespace LocalDbExample
{
    using System;
    using System.Collections.Generic;

    public partial class Customer
    {
        public System.Guid CustomerId { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
        public string Notes { get; set; }
    }
}
```

## Step 5: Create a Server Control in Default.aspx Source View

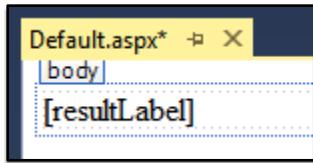
---

Now create a simple resultLabel, except this time let's create it directly within the *Default.aspx* code:



```
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="resultLabel" runat="server"></asp:Label>
        </div>
    </form>
</body>
```

This yields the same result as we would get through creating it in Design view:



## Step 6: Accessing the Entity Model in Code

---

Now in the Page\_Load() lets establish a connection to the database via code. This will tell the Entity Framework that we're interested in getting all of the customer records from the customer table in our database. This querying of the database, retrieving the customer data and storing it in a local variable – customers – happens when this method executes. The last step in this method then iterates through the Name property in each Customer:

```
protected void Page_Load(object sender, EventArgs e)
{
    ACMEEntities db = new ACMEEntities();

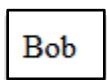
    var customers = db.Customers;

    string result = "";

    foreach (var customer in customers)
        result += "<p>" + customer.Name + "</p>";

    resultLabel.Text = result;
}
```

When you run the application, you should see the result displaying the name data for our lone entry in the Database:



061

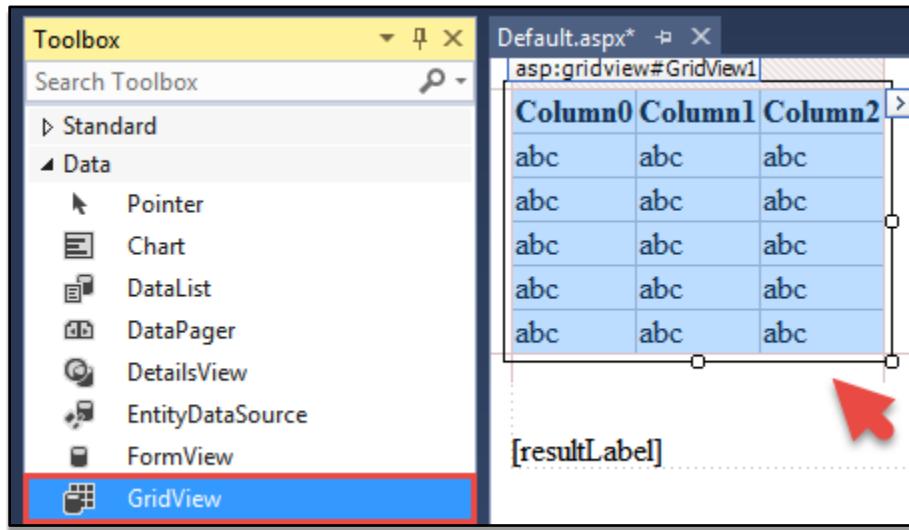
# Displaying the DbSet Result in an ASP.NET GridView

Now that we've learned how to map a persistent database onto classes and objects in our code, let's look at how we can display that data in a grid-like formatting of the table data we're retrieving from our "ACME" database. For that, we're going to use the GridView Server Control which has a feature called "Data Binding." Data Binding is just baked-in logic for certain Server Controls, in ASP.NET, that allows you to give it a data source – in this case a collection of customer objects. And it can then work with that collection, retrieve the records, format each individual record as a row in a table, and then display it all to the end user. Ideally, we should separate this "Persistence" layer information into its own project. But for now, it's important to just understand how data can be mapped from a database to local C# objects and then display it using a GridView.

## Step 1: Add A GridView Control to the Previous Lesson Project

---

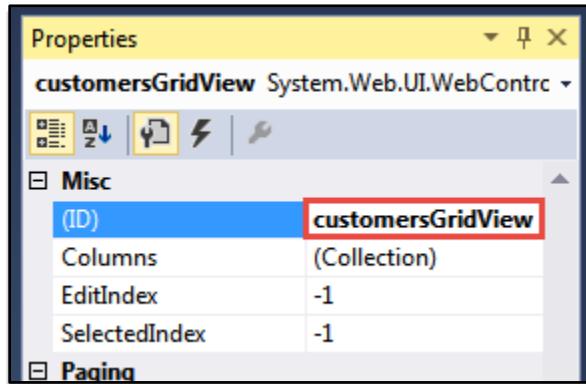
For this lesson, we're going to use where we left off in the previous lesson as the starting point. Once there, drag and drop a GridView Server Control from the Data section in the ToolBox, to the *Default.aspx*:





Bob's Tip: these Data Controls are non-visual elements that are either *Data-Source* or *Data-Bound Controls*. Data-Source Controls ask the question, "Where do I get the data?" Whereas Data-Bound Controls ask, "How do I connect to a database, retrieve information from it and display that data?"

In the Properties Window for this GridView Control, set the programmatic ID to "customerGridView":



Now, instead of iterating through customers, let's assign the contents of customers to `customersGridView.DataSource`:

```
protected void Page_Load(object sender, EventArgs e)
{
    ACMEEntities db = new ACMEEntities();

    var customers = db.Customers;

    string result = "";

    //foreach (var customer in customers)
    //    result += "<p>" + customer.Name + "</p>";

    customersGridView.DataSource = customers;

    resultLabel.Text = result;
}
```

## Step 2: Data Binding to the GridView in Code

---

Note that, even though customers seems like it would be an array or List<>, it is actually of type DbSet<>, which is a List-like container format for the entire database:

```
customers;
  (local variable) System.Data.Entity.DbSet<Customer> customers
```

After we have the DataSource, we can then Bind to it by calling the DataBind() method on the object:

```
customersGridView.DataSource = customers;

customersGridView.DataBind();
```

If you run the application now, this method will throw an exception:

```
customersGridView.DataBind();
```

**NotSupportedException was unhandled by user code**  
An exception of type 'System.NotSupportedException' occurred in EntityFramework.dll but was not handled in user code

## Step 3: Convert Data to a List

---

To fix this, we simply convert customers from a DbSet<> to a List<> by running it through the ToList() method:

```
customersGridView.DataSource = customers.ToList();

customersGridView.DataBind();
```

Now when you run the application, you will see that the data retrieved from the database becomes magically *bound* to the GridView in Default.aspx:

CustomerId	Name	Address	City	State	PostalCode	Notes
247347e9-f4fb-4b7c-9b69-d7e4114e8aee	Bob	123 E Main	Chicago	IL	60450	

# Implementing a Button Command in a GridView

Now that we've learned how to display retrieved information from the database in the GridView, we should now look at ways in which we can allow the user to interact with that data.

## Step 1: Create a New Project

---

For this lesson, create a new ASP.NET project called "GridViewButtonCommand" and in the *Default.aspx*, drag and drop the following:

- (1) GridView Control
- (2) Label Control

```
<body>
  <form id="form1" runat="server">
    <div>

      <h1>GridView Button Command</h1>
      <br />
      <br />
      1 <asp:GridView ID="GridView1" runat="server"></asp:GridView>
      <br />
      2 <asp:Label ID="resultLabel" runat="server"></asp:Label>
      <br />

    </div>
  </form>
</body>
```

And in *Default.aspx.cs* write following in the *Page\_Load()* method, with references to a public Car class:

```
namespace GridViewButtonCommand
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            var cars = new List<Car>()
            {
                new Car {CarId=Guid.NewGuid(), Make="BMW", Model="528i", Year=2010 },
                new Car {CarId=Guid.NewGuid(), Make="Toyota", Model="4Runner", Year=2010},
                new Car {CarId=Guid.NewGuid(), Make="Hyundai", Model="Elantra", Year=2013}
            };

            GridView1.DataSource = cars;
            GridView1.DataBind();
        }
    }

    public class Car
    {
        public Guid CarId { get; set; }
        public string Make { get; set; }
        public string Model { get; set; }
        public int Year { get; set; }
    }
}
```

When you run the application, you will see a very generic output of the data that's bound to *GridView1*:

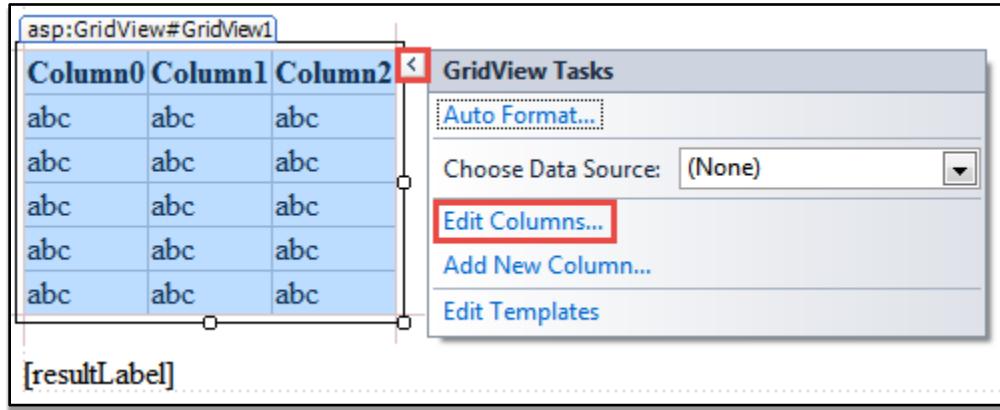
## GridView Button Command

CarId	Make	Model	Year
df94dba2-066d-4537-80d0-e12a61c6aa63	BMW	528i	2010
366580f1-038f-4b57-8bec-62221661ad4e	Toyota	4Runner	2010
f3015267-fd35-4fa5-811b-7b78ef32a34a	Hyundai	Elantra	2013

## Step 2: Add Data Bound Fields in the GridView

---

If you want to rearrange the columns, or make it interactive, you will have to make some changes to the GridView. In Design view, click on the arrow beside the GridView and select "Edit Columns...":



From here we can set the information displayed in each column:

- (1) Override the default behavior, letting us set what each column displays instead.
- (2) Select a "BoundField."
- (3) Add the BoundField to a list of selected fields.
- (4) Edit the HeaderText to display "Make".
- (5) Tell this BoundField to display the Make property from the Car class.

**Available fields:**

- BindField ②
- CheckBoxField
- HyperLinkField
- ImageField
- ButtonField
- CommandField
- TemplateField
- DynamicField

**BoundField properties:**

HeaderText	Make ④
DataField	Make ⑤

**Selected fields:**

- Make

**Auto-generate fields** ①

[Convert this field into a TemplateField](#)

Next, repeating steps (2) to (5), add three more BoundField's each representing the Model , Year and Car ID. And then add a ButtonField with its Text set to "View":

**Available fields:**

- BindField
- CheckBoxField
- HyperLinkField
- ImageField
- ButtonField**
- CommandField
- TemplateField
- DynamicField

**ButtonField properties:**

ButtonType	Link
HeaderText	
HeaderImageUrl	
HeaderText	
ImageUrl	
<b>Text</b>	<b>View</b>

**Selected fields:**

- Make
- Model
- Year
- Car ID
- Button

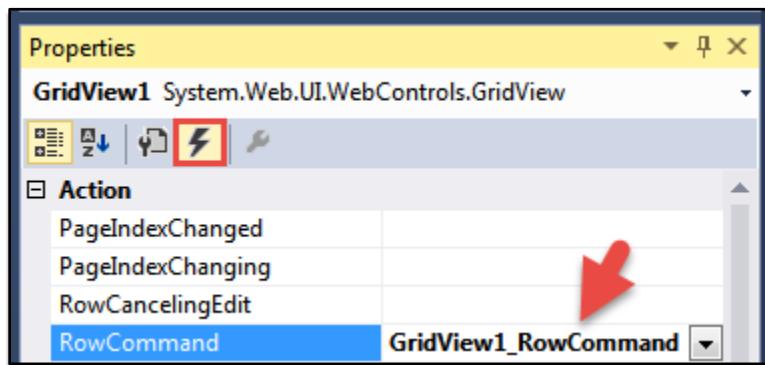
After clicking "OK" to that, you will see the new representation of the GridView data previewed in the Design view:

Make	Model	Year	Car ID	
Databound	Databound	Databound	Databound	<a href="#">View</a>
Databound	Databound	Databound	Databound	<a href="#">View</a>
Databound	Databound	Databound	Databound	<a href="#">View</a>
Databound	Databound	Databound	Databound	<a href="#">View</a>
Databound	Databound	Databound	Databound	<a href="#">View</a>

### Step 3: Set Up the RowCommand event

---

Go to the Properties Window for the GridView and in the Events tab double-click the empty entry beside "RowCommand" to create an event/method for it in *Default.aspx.cs*:



In this method, let's write some code that will grab the row data and then display it to the user when clicking the "View" button:

- (1) Grab the CommandArgument, off of the e input parameter, which we then convert to an integer (using one possible conversion method) and then use that as an index to refer to the particular row of data.
- (2) Store each cell we want to display in a temporary variable.
- (3) Attempt to parse the string value into a valid GUID for displaying the carID.
- (4) Output that cell data through the resultLabel.

```

protected void GridView1_RowCommand(object sender, GridViewCommandEventArgs e)
{
    1 int index = Convert.ToInt32(e.CommandArgument);
    2 GridViewRow row = GridView1.Rows[index];

    3 var make = row.Cells[0].Text;
    4 var model = row.Cells[1].Text;
    5 var value = row.Cells[3].Text;

    6 var carId = Guid.Parse(value);
    7 resultLabel.Text = String.Format("{0} {1} {2}", make, model, carId);

}

```

Make	Model	Year	Car ID	
BMW	528i	2010	1f5ae7ac-db2f-4622-8912-f1e67d4e5bc0	<a href="#">View</a>
Toyota	4Runner	2010	64a14b94-508c-4e3e-82db-e9fa55d8e841	<a href="#">View</a>
Hyundai	Elantra	2013	cf89d741-883e-46bf-9c63-1125c365fabb	<a href="#">View</a>

Hyundai Elantra cf89d741-883e-46bf-9c63-1125c365fabb



063

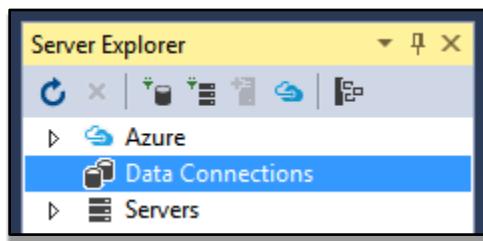
# Using a Tools-Centric Approach to Building a Database Application

When developing software, you often have to choose between taking a primarily "Tools Driven" approach, versus a "Maintenance Driven" approach. Simply put, Tools Driven development relies on external toolsets that make it easier to quickly write the application and get up and running at the cost of understanding the system at a deep level. That's because you're off-loading a lot of responsibility of writing code and making relevant connections between elements in your software to the tool in question. The maintenance approach is the opposite of that. It focuses on developing a robust system that can respond to change easier because it doesn't rely on outside tools (that might change) and is written with an understanding of how the system operates at a deeper level.

## Step 1: Demonstrating a Tools-Driven Approach

---

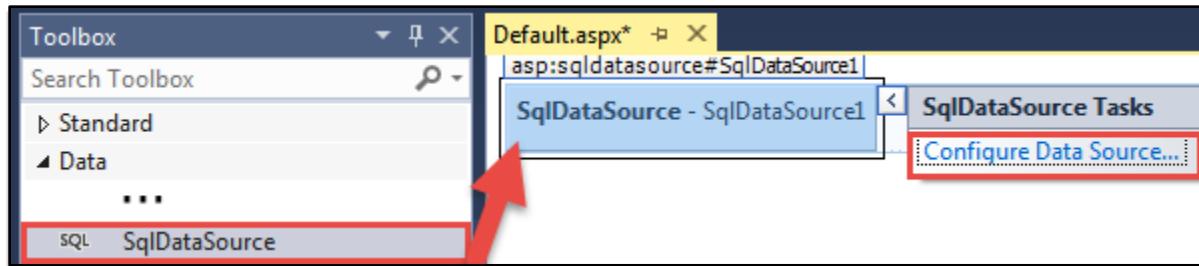
This lesson will show how a Tools Driven approach will get you up and running in a short amount of time. To start this lesson, use the LocalDbExample we left off with in lesson 060 as the starting point, excluding the Entity model, as well as the database connection to "ACME.mdf":



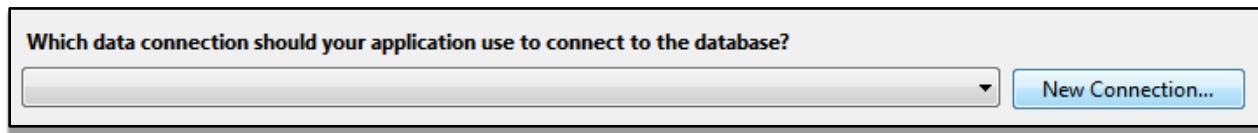
## Step 2: Add an sqlDataSource Control

---

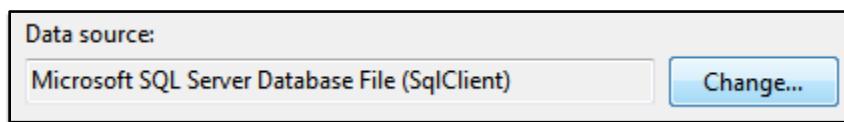
Refer again to the Data section of the ToolBox which holds a variety of ASP.NET Data Controls. Recalling that there are basically two different kinds of these controls – data sources that locate data, and then the data controls themselves that read and display the data. Data Source components are not viewable to the end-user. For example, when you drag and drop a SqlDataSource onto the Design surface, it's just there to provide information to other Controls for our form:



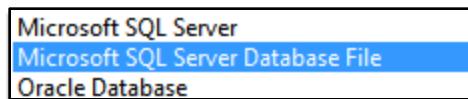
If you click on "Configure Data Source..." from the arrow beside the SqlDataSource, you will be taken to a setup wizard which lets you make a database connection. First, click on "New Connection":



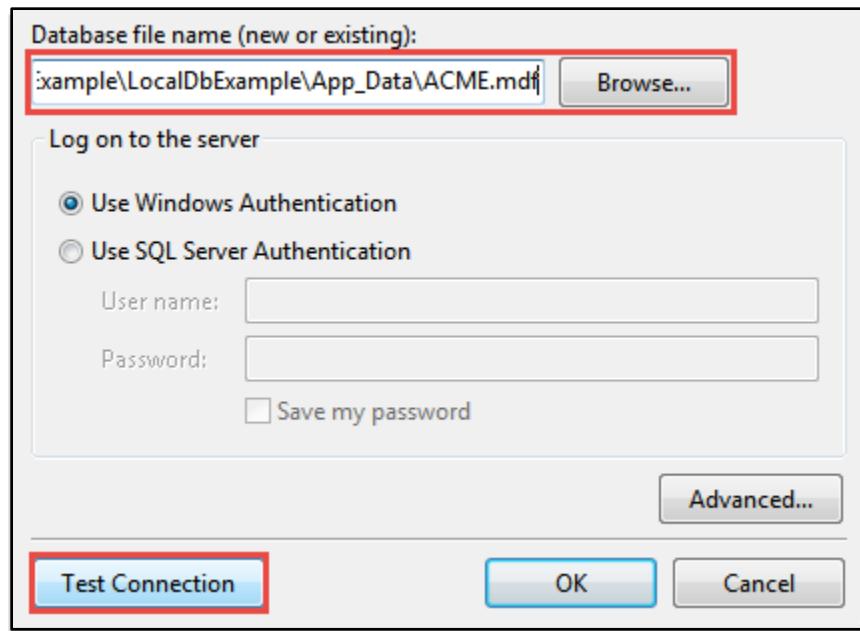
And from here, click on "Change":



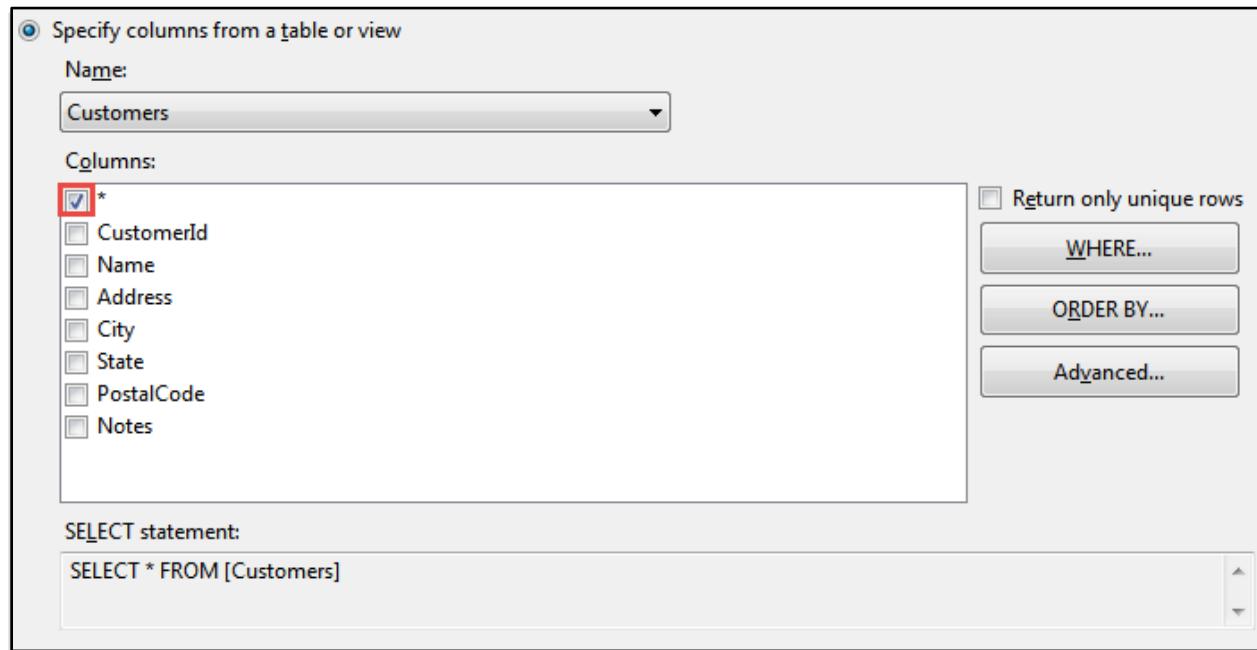
And set the database to a local file:



Then browse to the "App\_Data" folder and select the "ACME.mdf" file (local to the folder it is stored to on your computer), and then test that the connection is valid by clicking on "Test Connection":



If the test resolves correctly, click on the “OK” button and continue clicking on “Next” until you get to this screen, making sure to select the Customers table and having the asterisk selected to get all columns in the table:



Click “Next” and on the next screen you can test a query to return results from the database:

CustomerId	Name	Address	City	State	PostalCode	Notes
247347e9-f4fb-4b7c-9b69-d7e4114e8aee	Bob	123 E Main	Chicago	IL	60450	

**Test Query**

## Step 3: See the Query String Added to Default.aspx Source Markup

---

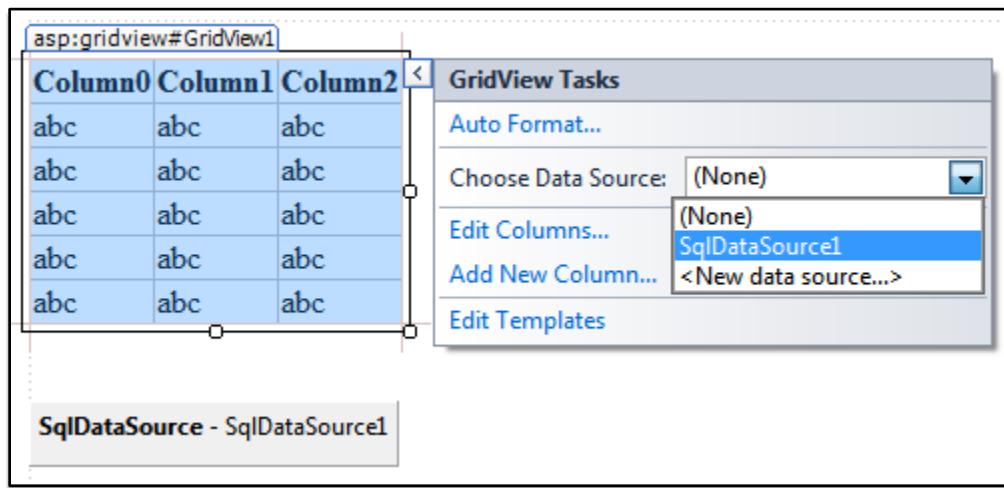
Click on "Finish" to finalize setting up this sqlDataSource. If you go back to the markup in *Default.aspx* you will now see this query string that gets sent to the database whenever the sqlDataSource asked to return data:

```
SelectCommand="SELECT * FROM [Customers]">
```

## Step 4: Set sqlDataSource as the GridView's Data Source

---

Back in the Design view, click and drag from the ToolBox a GridView Control onto the Design surface and choose sqlDataSource1 where it says "Choose Data Source":



It will then fill in the header information with each database column name, in the GridView:

CustomerId	Name	Address	City	State	PostalCode	Notes
------------	------	---------	------	-------	------------	-------

Now when you go to run the application – without having written any code of our own – we get the same output that we got with a more hands-on approach:

CustomerId	Name	Address	City	State	PostalCode	Notes
247347e9-f4fb-4b7c-9b69-d7e4114e8aee	Bob	123 E Main	Chicago	IL	60450	

## Step 5: Limitations of the Tools Driven Approach

---

This Tools Driven approach is fast but it's a bad long-term strategy. It might work for a small, departmental application where you only have a few users and you don't expect a lot of change. However, if you have a large application and you expect there to be ongoing maintenance and changes to the application over time, this is not the right approach for building that application. We looked at this approach here in order to contrast it with a better one in the next lesson. Having said that, let's demonstrate an even easier Tools Driven approach to achieving the results we demonstrated in this lesson.

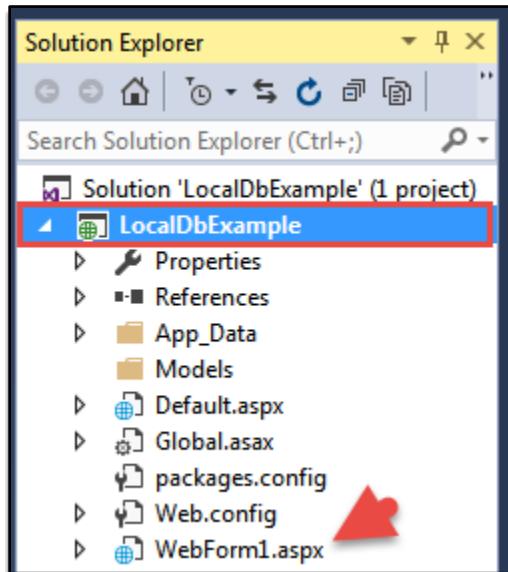
## Step 6: Let Visual Studio Automatically Setup the Data Binding

---

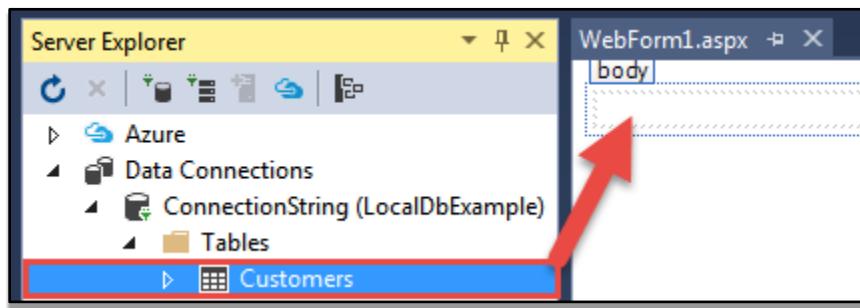
First, right-click on the project in the Solution Explorer and from the menu select:

Add > New Item

And then add a new .aspx Web Form:



And now from the Server Controls Window, click and drag the Customers database table directly onto the Design surface for *WebForm1.aspx*:



After a moment, you will see that Visual Studio recognized what you wanted to do, automatically setting up the *sqlDataSource* and connected it to the *GridView*:

A screenshot of the 'WebForm1.aspx' design surface. At the top, the title bar says 'WebForm1.aspx\*'. Below it, the 'body' section contains a *GridView* control. The grid has columns labeled 'CustomerId', 'Name', 'Address', 'City', 'State', 'PostalCode', and 'Notes'. There are five rows of data, all containing the value 'abc'. Below the grid, a *SqlDataSource* control is visible, with the label 'SqlDataSource - SqlDataSource1'.

064

# Using a Maintenance-Centric Approach to Building a Database Application

In the previous lesson, we saw the benefits of using a Tools Driven development approach, which is a great approach if you are looking at a departmentally scoped application that you need to create quickly. This approach requires very little logic, and is feasible for simple “CRUD” style database operations (create, read, update, delete), where you don’t really anticipate any changes in the long term. However, not every application is like that. Many applications are not scoped to that of a small department, but rather at an Enterprise level. In that case, a fast turn-around is far less important than getting it right and having it flexible enough to anticipate inevitable changes in the system.

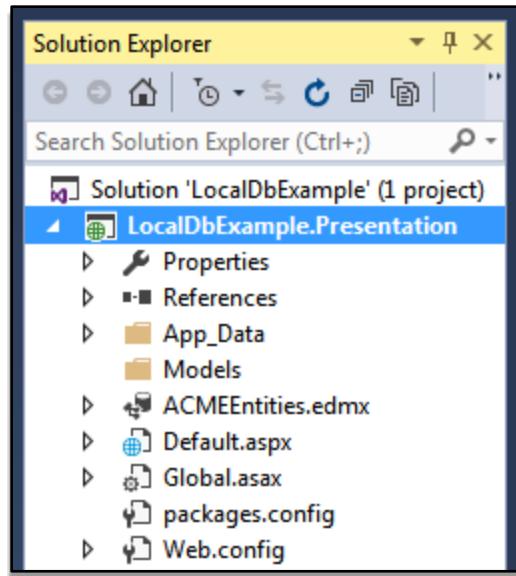


Bob's Tip: we're looking at two different extremes to solving development problems. In reality, there are gradations – in between the completely Tools Driven and completely Maintenance Driven approaches – that utilize a bit of both in order to get the job done.

## Step 1: Backtrack to Lesson 061

---

For this lesson, we will start with where we left off in Lesson 061 – backtracking away from what we learned about the Tools Driven approach, and re-assessing the problem through the lens of a Maintenance Driven approach instead. A Maintenance Driven approach should start with a logical separation of concerns between the three main layers. Start by renaming the project to “LocalDbExample.Presentation”:

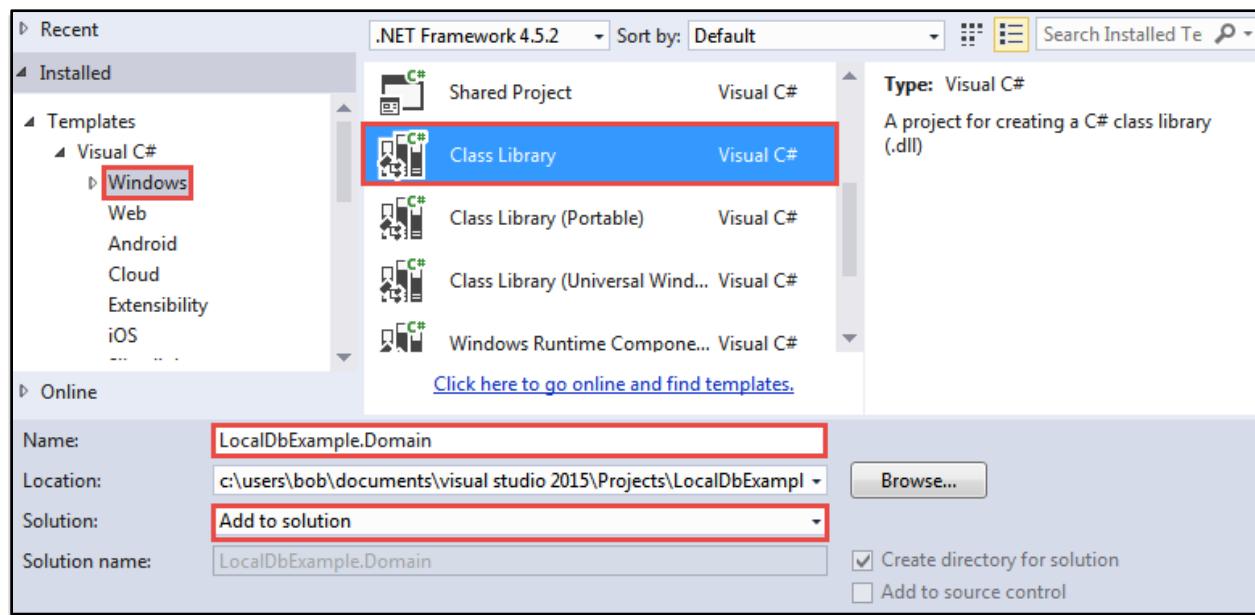


## Step 2: Add New Project Layers

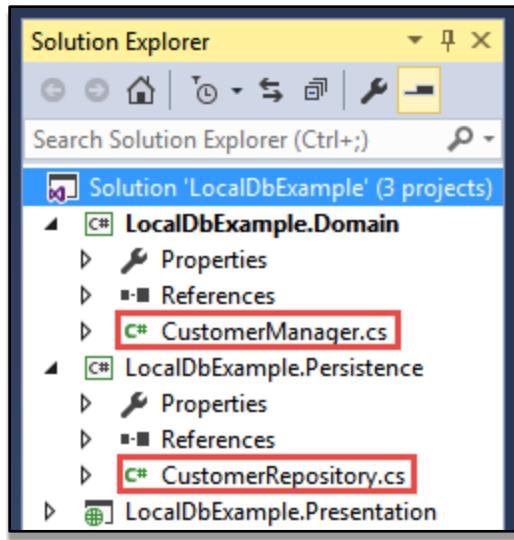
---

Next, let's add the two other layers by selecting from Visual Studio's menu:

File > Add > New Project



Repeat this step to create a LocalDbExample.Persistence layer as well. Afterwards, rename the *Class1.cs* files in these layers as follows:



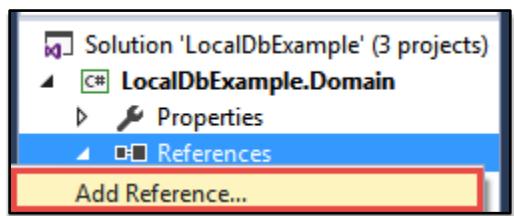
At this point, it's probably a good idea to *build* the Solution to make sure everything is set up correctly. The shortcut key for building a solution is:

Ctrl+Shift+B

## Step 3: Set Dependencies between Project Layers

---

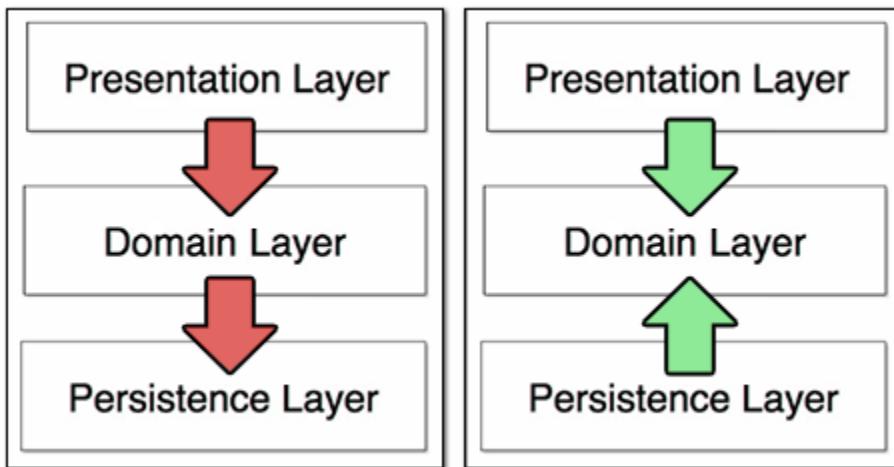
Building is important so that each project can "see" the other's and allow dependencies to be established. When one project *takes a dependency* from another one, it simply means it makes reference to it in code. Ideally, you will want to keep your code as independent as possible but a certain amount of dependency is going to be necessary. Set up a Domain dependency on the Persistence layer by right-clicking on "References" and selecting "Add Reference...":



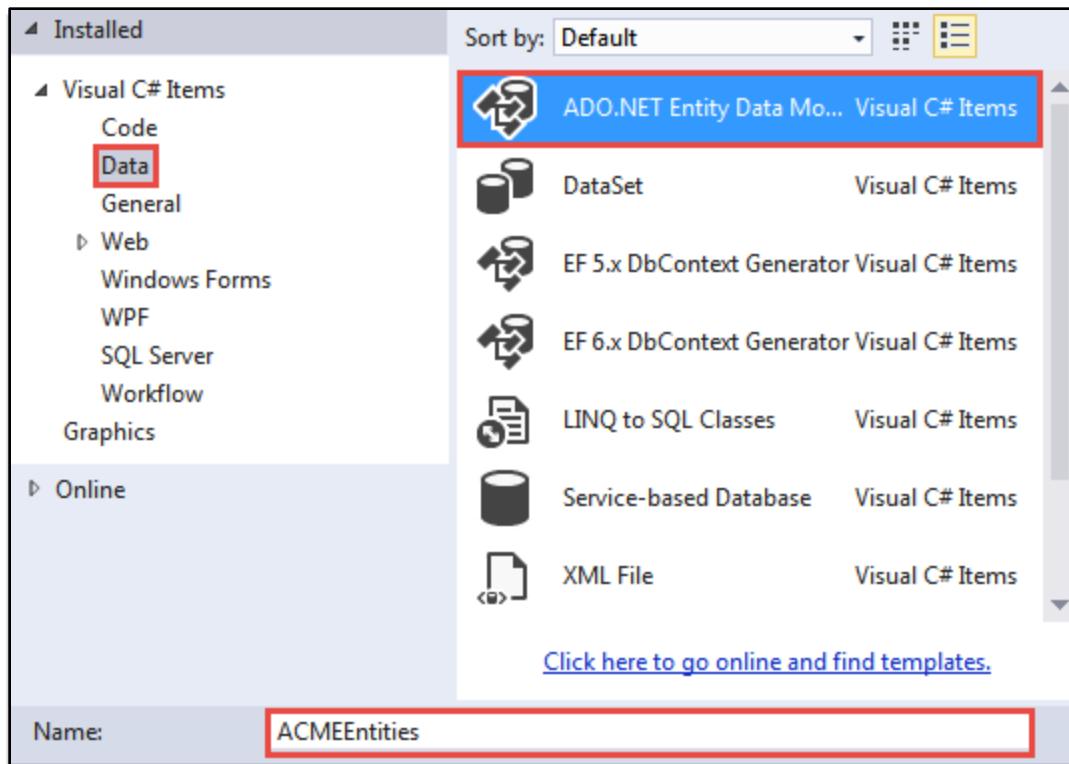
From here, select the Persistence layer as a dependency and after that, you should see it in your list of References under LocalDbExample.Domain:

Name
<input checked="" type="checkbox"/> LocalDbExample.Persistence
<input type="checkbox"/> LocalDbExample.Presentation

Repeat this process, adding a dependency to the Domain layer in the Presentation layer's References. Of course it would be better to have the Presentation, and Persistence layers depend on the Domain layer, which at least keeps the dependency chain central to the Domain layer. However, for now, we are going to simplify things and have the Presentation layer depend on the Domain layer, and the Domain Layer depend on the Persistence layer, represented in red arrows below:



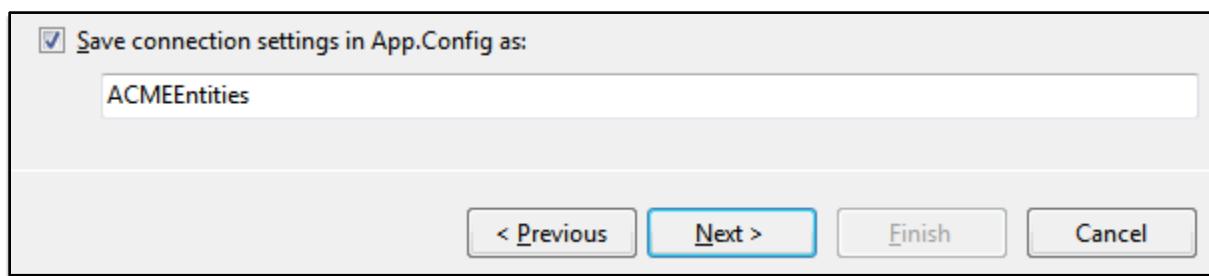
Now we will have to re-create the Entity model in the Persistence layer. In the Solution Explorer, right-click on the LocalDbExample.Persistence layer and add a new item. From the available templates, select the ADO.NET Entity Data Model as we did in the previous lesson:



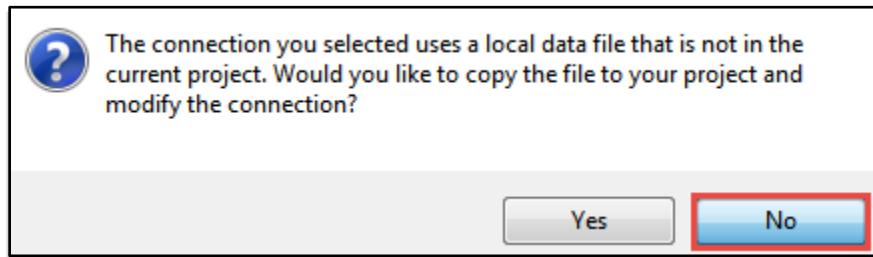
On the next screen choose "EF Designer from database":



Next, we're going to save the database connection settings to an App.config file (typically, you would not want to use an App.config but rather a Web.config for an actual web application. However, it will work for the purposes of this demonstration):



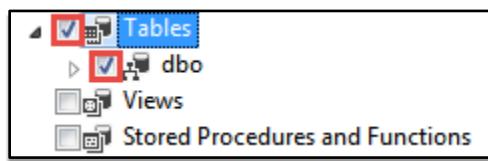
The next step asks if you would like to copy the file to the project, select "No" to this step:



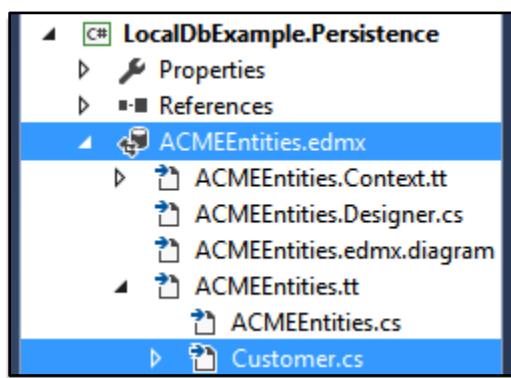
At the next step select Entity Framework 6.x, and then click "Next":



The setup wizard will then ask you which database object you would like to include in your model. Select all of the Tables, and then click "Finish":



After that you will end up with an Entity model in the Persistence layer, complete with a Customer class it automatically created to model the database table of the same name:



```
public class Customer
{
    public System.Guid CustomerId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
    public string Notes { get; set; }
}
```

Go now to the *CustomerRepository.cs* class and write a method that returns a `List<Customer>` obtained from the database via the Entities model:

```
public class CustomerRepository
{
    public static List<Customer> GetCustomers()
    {
        ACMEEntities db = new ACMEEntities();
        var dbCustomers = db.Customers.ToList();
        return dbCustomers;
    }
}
```

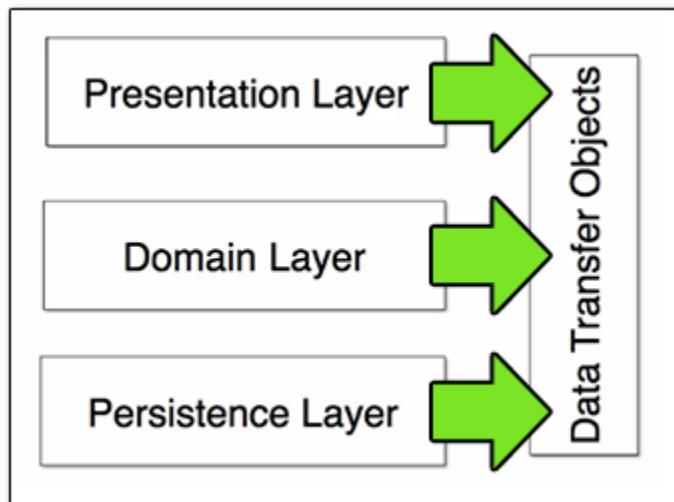
Now in the *CustomerManager.cs* class, we will need to reference the Persistence layer through a using statement added to the top of the script:

```
using LocalDbExample;
```

And then in the *CustomerManager* class, write a method that simply calls `Persistence.CustomerRepository.GetCustomers()` and returns the `List<Customer>`:

```
public class CustomerManager
{
    public static List<Persistence.Customer> GetCustomers()
    {
        var customers = Persistence.CustomerRepository.GetCustomers();
        return customers;
    }
}
```

At this point, we've exposed the Entity Framework Persistence layer to the Domain layer via this reference in the CustomerManager class. Exposing these inner workings is not a great idea, so what we need to do is add one more project called a Data Transfer Object Layer (DTO). The purpose of this layer is to abstract away the three main layers from each other, as they really should not have to know about one another to do their individual job. The DTO layer acts as the common reference point instead, and its dependency relationship to the other layers can be viewed as follows:



Let's now add this DTO layer to our project – using the steps outlined previously in the lesson – calling it "LocalDbExample.DTO." We will want to copy and paste the exact same class structure for the Customer class in the Persistence layer, and place it in the Class1.cs file (rename this file to Customer.cs as well):

```
namespace LocalDbExample.DTO
{
    public class Customer
    {
        public System.Guid CustomerId { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
        public string Notes { get; set; }
    }
}
```

Now we will want to add a reference to LocalDbExample.DTO under each other project's "References" in the Solution Explorer (remember, each project should point to this one, rather than to one another).

LocalDbExample.Domain Reference:

Name
<input checked="" type="checkbox"/> LocalDbExample.DTO
<input checked="" type="checkbox"/> LocalDbExample.Persistence
LocalDbExample.Presentation

LocalDbExample.Persistence Reference:

Name
LocalDbExample.Domain
<input checked="" type="checkbox"/> LocalDbExample.DTO
LocalDbExample.Presentation

LocalDbExample.Presentation Reference:

Name
<input checked="" type="checkbox"/> LocalDbExample.Domain
<input checked="" type="checkbox"/> LocalDbExample.DTO
LocalDbExample.Persistence

The CustomerRepository class now should reference the DTO. Customer class, instead of storing the pulled database data to a local Customer Class reference. To do this, modify CustomerRepository as follows:

- (1) Retain the code that pulls database data and stores it to a List<Customer> object.
- (2) Create a new, locally scoped List<DTO. Customer> object.
- (3) In a foreach() create a temporary DTO. Customer object.
- (4) Populate each property for this object using the properties from the iterated List<Customer>
- (5) Add the DTO. Customer to the List<DTO. Customer>.
- (6) Return the List<DTO. Customer>, making sure to also change the return type in the method signature.

```

public class CustomerRepository
{
    public static List<DTO.Customer> GetCustomers()
    {
        1 ACMEEntities db = new ACMEEntities();
        var dbCustomers = db.Customers.ToList();
        2 var dtoCustomers = new List<DTO.Customer>();

        foreach (var dbCustomer in dbCustomers)
        {
            3 var dtoCustomer = new DTO.Customer();

            4 {
                dtoCustomer.CustomerId = dbCustomer.CustomerId;
                dtoCustomer.Name = dbCustomer.Name;
                dtoCustomer.Address = dbCustomer.Address;
                dtoCustomer.City = dbCustomer.City;
                dtoCustomer.State = dbCustomer.State;
                dtoCustomer.PostalCode = dbCustomer.PostalCode;
                dtoCustomer.Notes = dbCustomer.Notes;
            }

            5 dtoCustomers.Add(dtoCustomer);
        }

        6 return dtoCustomers;
    }
}

```

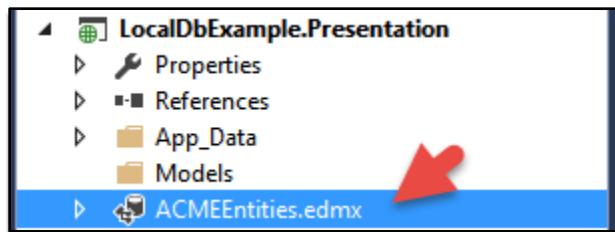
Now change the return type for the `GetCustomers()` method in the `CustomerManager.cs` class, seeing as we now have `Persistence.CustomerRepository.GetCustomers()` returning a `List<DTO.Customer>`:

```

public static List<DTO.Customer> GetCustomers()
{
    var customers = Persistence.CustomerRepository.GetCustomers();
    return customers;
}

```

Turning our attention now to the Presentation layer, first delete the `ACMEEntities.edmx` as we will be using the database from the Persistence layer instead:

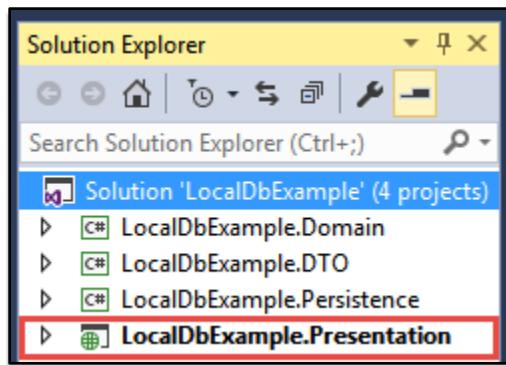


Now change the Page\_Load() method, within *Default.aspx.cs*, to reflect the reference to the Domain layer:

```
protected void Page_Load(object sender, EventArgs e)
{
    var customers = Domain.CustomerManager.GetCustomers();

    customersGridView.DataSource = customers.ToList();
    customersGridView.DataBind();
}
```

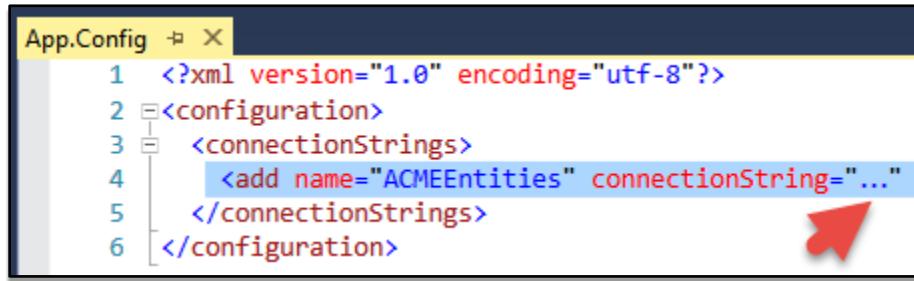
The last thing to do is set the Presentation layer as the entry-point for the application by right-clicking it and selecting from the menu "Set as Startup Project":



Now, when you run the application you will see the same results as the previous lesson but with a cleaner, more separated application structure behind the scenes:

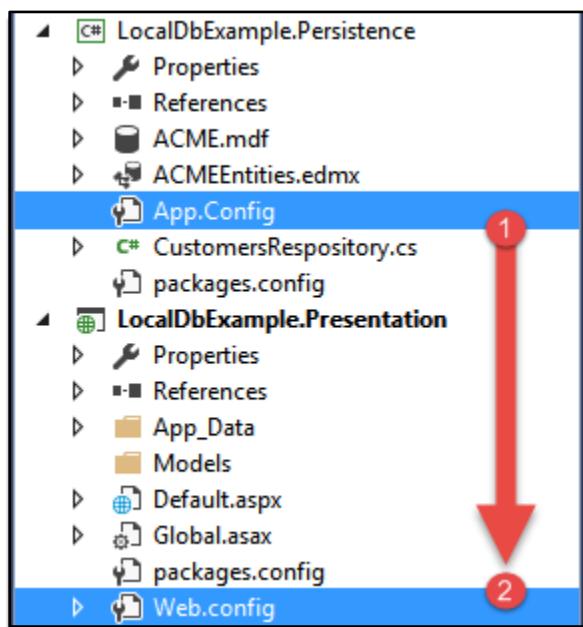
CustomerId	Name	Address	City	State	PostalCode	Notes
247347e9-f4fb-4b7c-9b69-d7e4114e8aee	Bob	123 E Main	Chicago	IL	60450	

If you received an error upon running the application, there might be a mismatch between the connectionString stored in the *App.config* and *Web.config* files, respectively. You will find the connectionString in each file, represented here in *App.config* as "...". (your actual connection string will be a much longer):



```
App.Config
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3    <connectionStrings>
4      <add name="ACMEEntities" connectionString="..." />
5    </connectionStrings>
6  </configuration>
```

To fix the error, simply (1) copy the entire connectionString from *App.config* in the Persistence layer and (2) paste it over the connectionString in *Web.config* in the Presentation layer.



Bob's Tip: you may be thinking that this is a lot of effort just to build a simple application that we already saw could be made more easily, and with the same results. Yes, for a simple application this won't make a lot of sense, but keep in mind that applications are not this small. All of that time spent setting up code and keeping layers separate will pay dividends as the application grows larger and has to respond to inevitable change.

# Creating a New Instance of an Entity and Persisting to the Database

Continuing on from where we left off in the previous lesson, we'll now look at how to programmatically add new customers into the Customer table of the ACME database. We'll accomplish this by taking in data via the Presentation layer and then storing it in the Persistence layer. This will demonstrate the typical mechanics of using the Entity Framework to insert a new Entity into a database. While this lesson touches upon the "create" and "read" part of a typical "CRUD" set of operations, you can extend the principals learned here towards understanding the "update" and "delete" operations as well.

## Step 1: Create Controls in Default.aspx

---

The first thing we'll do is add Form Controls to the *Default.aspx* page in order to capture data from the customer input on the Presentation layer:

```
<body>
    <form id="form1" runat="server">
        <div>
            <h2>Customers</h2>
            <asp:GridView ID="customersGridView" runat="server">
            </asp:GridView>

            Name: <asp:TextBox ID="nameTextbox" runat="server"></asp:TextBox>
            <br />
            Address: <asp:TextBox ID="addressTextBox" runat="server"></asp:TextBox>
            <br />
            City: <asp:TextBox ID="cityTextBox" runat="server"></asp:TextBox>
            <br />
            State: <asp:TextBox ID="stateTextBox" runat="server"></asp:TextBox>
            <br />
            Zip: <asp:TextBox ID="zipTextBox" runat="server"></asp:TextBox>
            <br />
            Notes: <asp:TextBox ID="notesTextBox" runat="server"></asp:TextBox>
            <br />
            <asp:Button runat="server" Text="Save Data" ID="okButton" OnClick="okButton_Click" />
            <br />
            <asp:Label ID="resultLabel" runat="server"></asp:Label>
        </div>
    </form>
</body>
```

## Step 2: Create a Method for Adding Customers

---

Back in *CustomerRepository.cs* of the Persistence layer, we'll first create a new *Customer()* (as in *Customer.cs*) and then fill it in with the values supplied from a *DTO.Customer* input argument:

```
namespace LocalDbExample.Persistence
{
    public class CustomersRespository
    {
        public static List<DTO.Customer> GetCustomers()...
        public static void AddCustomer(DTO.Customer newCustomer)
        {
            var customer = new Customer();

            customer.CustomerId = newCustomer.CustomerId;
            customer.Name = newCustomer.Name;
            customer.Address = newCustomer.Address;
            customer.City = newCustomer.City;
            customer.State = newCustomer.State;
            customer.PostalCode = newCustomer.PostalCode;
            customer.Notes = newCustomer.Notes;
        }
    }
}
```

## Step 3: Add the New Customer to the Database

---

Below the code we just wrote in the *AddCustomer()* method, we reference the ACMEEntities database, adding the customer to it and then saving those changes:

```
ACMEEntities db = new ACMEEntities();
var dbCustomers = db.Customers;

dbCustomers.Add(customer);
db.SaveChanges();
```

## Step 4: Validate the Data Before Committing it to the Database

---

It should be obvious that we will only want specific types of data to be entered into the database. To do this, we can add validation before the changes are saved to the database with the invocation of the *SaveChanges()* method. For example, you can add this validation at the top of the *AddCustomer()* method to protect against an empty field being submitted for the customer's name by throwing an *Exception* before the changes are saved:

```
public static void AddCustomer(DTO.Customer newCustomer)
{
    if (newCustomer.Name.Trim().Length == 0)
        throw new Exception("Name field is required.");

    //Continue On with Other validation.
```

You might then want to wrap the commit operation around a try/catch:

```
try
{
    dbCustomers.Add(customer);
    db.SaveChanges();
}
catch (Exception ex)
{
    //Ideally, Add Code To Log the Exception
    throw ex;
}
```



Bob's Tip: The example here is less than ideal. Normally, you would want to handle the Exception properly rather than re-throw it. Also, you would probably want to create a custom Exception that is handled in the Domain layer so as to avoid bubbling information across layers that shouldn't even be concerned with such details. The above approach will work in a pinch, however, just remember that you should strive to keep layers as separate as possible.

## Step 5: Call AddCustomer() in the Domain Layer

Next, we'll create an `AddCustomer()` method in `CustomerManager.cs` of the Domain layer that simply calls the `AddCustomer()` method we had just created in the Persistence layer. This may seem redundant, however, keeping the implementation details of adding customers in the Domain layer helps maintain Separation of Concerns. Go to `CustomerManager.cs` and add the method:

```
namespace LocalDbExample.Domain
{
    public class CustomerManager
    {
        public static List<DTO.Customer> GetCustomers()...
        public static void AddCustomer(DTO.Customer customer)
        {
            Persistence.CustomersRepository.AddCustomer(customer);
        }
    }
}
```

## Step 6: Store New Customer Data upon Button Click

---

Back in *Default.aspx.cs* we'll add the following code to the *okButton\_Click* event:

```
protected void okButton_Click(object sender, EventArgs e)
{
    var newCustomer = new DTO.Customer();

    newCustomer.CustomerId = Guid.NewGuid();
    newCustomer.Name = nameTextBox.Text;
    newCustomer.Address = addressTextBox.Text;
    newCustomer.City = cityTextBox.Text;
    newCustomer.State = stateTextBox.Text;
    newCustomer.PostalCode = zipTextBox.Text;
    newCustomer.Notes = notesTextBox.Text;
}
```

Immediately below that code, we'll then call the *AddCustomer()* method from the Domain layer and pass in the Customer data. Note that it's a good habit to wrap method calls to outside layers around a *try/catch*, especially when that method is performing some sort of data access. Here we'll simply display the error message if an *Exception* is caught:

```
try
{
    Domain.CustomerManager.AddCustomer(newCustomer);
}
catch (Exception ex)
{
    resultLabel.Text = ex.Message;
}
```

You should probably do the same to AddCustomer() in the *CustomerManager.cs* and actually perform some custom Exception Handling, including logging it:

```
public static void AddCustomer(DTO.Customer customer)
{
    try
    {
        Persistence.CustomersRepository.AddCustomer(customer);
    }
    catch (Exception)
    {
        //Log it
        throw;
    }
}
```

Finally, take the code responsible for displaying customer data – which had already been in the Page\_Load() method - and put it into a private method in *Default.aspx.cs*:

```
namespace LocalDbExample
{
    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)...
        protected void okButton_Click(object sender, EventArgs e)...  
  

        private void displayCustomers()
        {
            var customers = Domain.CustomerManager.GetCustomers();

            customersGridView.DataSource = customers.ToList();
            customersGridView.DataBind();
        }
    }
}
```

Call this method at the bottom of both Page\_Load() and okButton\_Click() to make sure the screen reflects the updated data when loading the page or clicking the OK button. When you run the application, enter customer data to see if everything is working correctly and successfully stores the data into the database:

CustomerId	Name	Address	City	State	PostalCode	Notes
6d776175-7d40-47fe-bc46-c5	Steve	345 Elm	Dallas	TX	70450	First Attempt!
247347e9-f4fb-4b7c-9b69-d7	Bob	123 E Main	Chicago	IL	60450	

Name:

Address:

City:

State:

Zip:

Notes:

066

# Package Management with NuGet

In this lesson, we're going to talk about NuGet, which is a Package Manager built into Visual Studio. The term "package" refers to any third-party library that you can import – and use to build applications – in your code. Package Managers can help you find, and manage, a plethora of freely available, open source projects that provide core functionality for your application. For example, why go off and build logging functionality when there are several good, open source logging utilities already available out there? You just need to download them, add them to your project, create a reference, and now you can use them inside of your project as if they were code that you've written yourself.

## Step 1: Understanding What a Package Manager Does

---

- (1) Installs files necessary to include a third party library/resource into your project.
- (2) Adds references to the class library files in your project.
- (3) Adds any dependencies, including other packages, that the target class library requires.
- (4) Updates the package, and its dependencies, to the latest version.

NuGet is a package manager that can run from command-line (for those used to using that kind of interface), as well as from a dialog in Visual Studio. You can browse, and download, available packages directly within Visual Studio by selecting from the menu:

Tools > NuGet Package Manger > Manage NuGet Packages for Solution

You can also download these packages from your web browser by going to:

<http://www.NuGet.org>

## Step 2: Using The Package Manager

---

Using the dialog utility from inside of Visual Studio, locate the latest version of "Bootstrap" from NuGet.org and install it into your project:



Bob's Tip: Bootstrap is a popular "mobile-first" Framework developed, and made openly available, by Twitter. It's a package made up of mostly pre-written CSS and Javascript that you can reference in your web pages to employ a "responsive" grid-like format that adapts to the viewport of large and small devices alike.

## NuGet Package Manager: Solution 'NuGetExample'

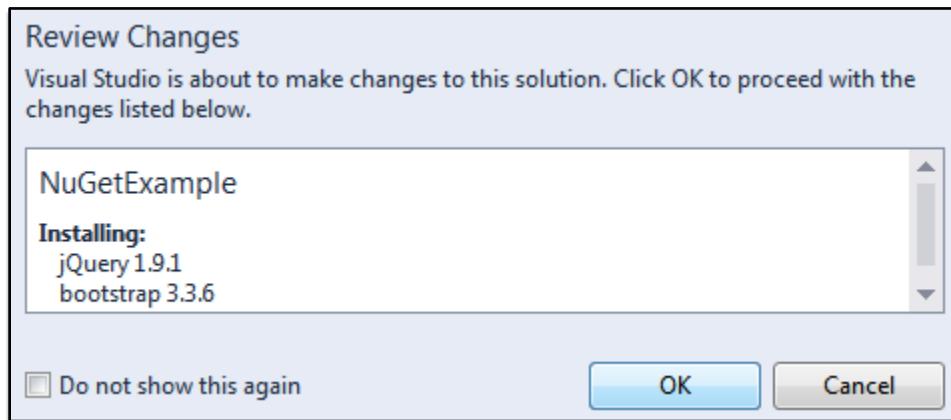
The screenshot shows the NuGet Package Manager interface. At the top, there are search filters: 'Package source: nuget.org', 'Filter: All', and a checked 'Include prerelease' option. A search bar contains the text 'Bootstrap'. Below the search bar, the results are displayed:

- bootstrap** (highlighted with a red box) - Bootstrap framework in CSS. Includes fonts and JavaScript.
- Angular.ULBootstrap** - Native AngularJS (Angular) directives for Bootstrap. Small...
- Twitter.Bootstrap** - Copyright 2012 Twitter, Inc.
- Twitter.Bootstrap.Less** - Code and documentation copyright 2011-2014 Twitter, I...

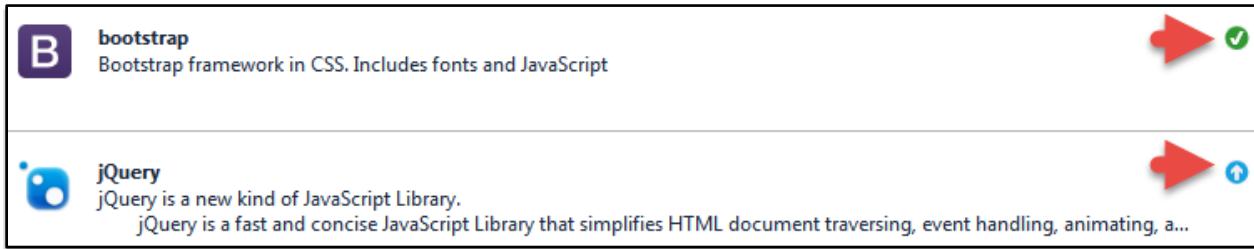
On the right side, details for the selected package 'bootstrap' are shown:

- Action:** Install (highlighted with a red box)
- Version:** Latest stable 3.3.6 (highlighted with a red arrow)
- Select which projects to apply changes to:**  1 project(s)  Show all  
NuGetExample (highlighted with a red box)
- Install** button (highlighted with a red box)
- Options** section:
  - Show preview window
  - Dependency behavior: Lowest
  - File conflict action: Prompt
  - [Learn about Options](#)

Make sure you search for the package source through "nuget.org," and click OK to finish the process:



Once it successfully installs to your project you will see a green checkmark beside it in your package manager, while an update arrow might be visible to show an update is available for a related dependency:



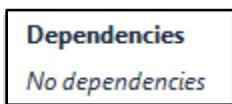
Go ahead and update JQuery to the latest version:



You can view dependencies by clicking on the package in the Package Manager. Here the Bootstrap package shows a dependency on JQuery, including the version required:



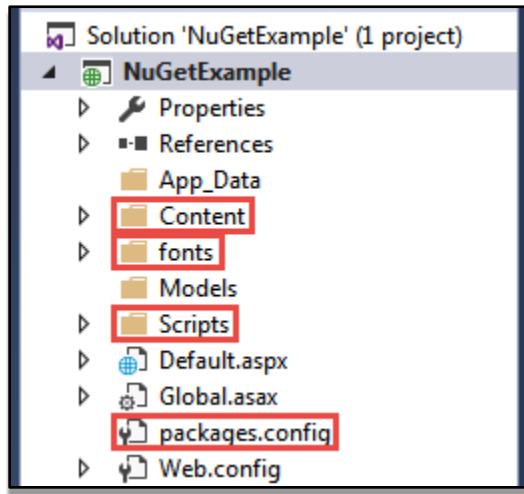
The JQuery package, on the other hand, has no dependencies:



## Step 3: Review the Package in the Solution Explorer

---

You will also see the added package references in the Solution Explorer:



These package files and folders break down as follows:

- “Content” contains the core CSS files for Bootstrap.
- “fonts” contains common icons for resized mobile menu elements.
- “Scripts” contains Javascript/JQuery for handling browser-side behaviors.
- “packages.config” contains references to these packages within an XML configuration file.

## Step 4: Using the Package Manager Console

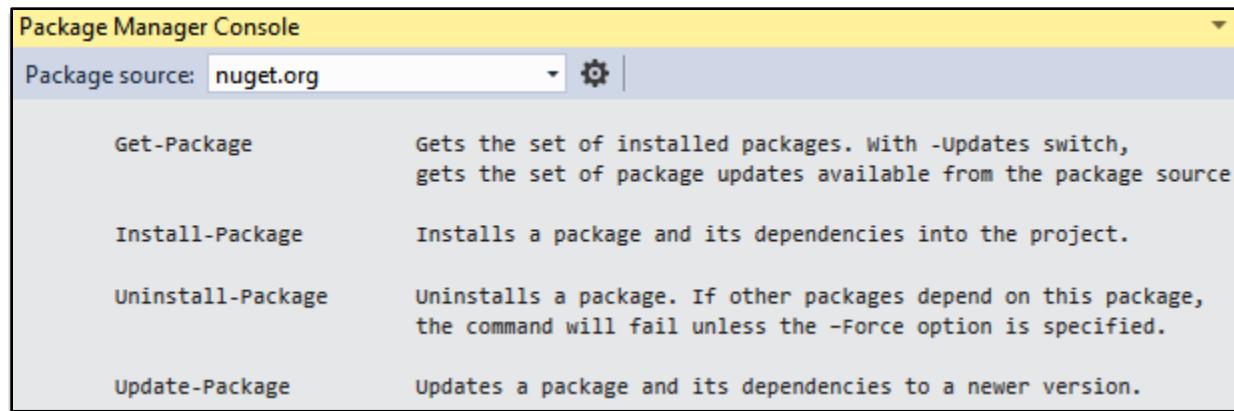
---

Some may prefer using the Package Manager Console as another way of finding and installing packages. You get to this Console by selecting from the Visual Studio menu:

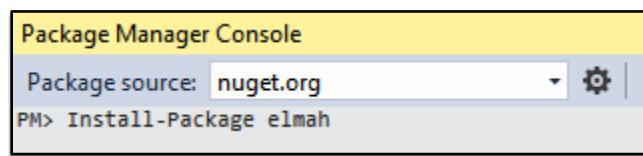
Tools > NuGet Package Manager > Package Manager Console

You can find a list of commands at: <https://docs.nuget.org/consume/command-line-reference>

You can also type in “get-help nuget” in the command line to get a list of commands;



Use the "Install-Package" command to install the popular logging utility called "Elmah":



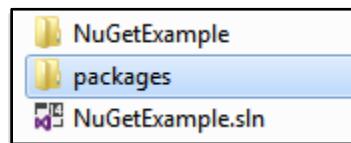
You should then see a list of actions indicating a successful install:

```
Added package 'elmah.1.2.2' to folder 'c:\users\bob\documents\visual studio 2015\Projects\'NuGetExample\packages'
Added package 'elmah.1.2.2' to 'packages.config'
Successfully installed 'elmah 1.2.2' to NuGetExample
```

If you run the "Update-Package" command you may, or may not, see available updates:

```
PM> Update-Package elmah
Attempting to gather dependencies information for multiple packages with respect to project
'NuGetExample', targeting '.NETFramework,Version=v4.5.2'
Attempting to resolve dependencies for multiple packages
Resolving actions install multiple packages
No package updates are available from the current package source for project 'NuGetExample'.
```

Note that you can find the installed packages in the project folder under "packages":



067

# NuGet No Commit Workflow

There's one more feature of NuGet that we should look at, and it has to do with working in a team environment. When you're working with other developers on the same team, you're sharing source code – usually by checking into a centralized repository. This technique is typically called "Source Control." It's not only for centralized project sharing but also for keeping a version history of your application allowing you to roll back changes to earlier versions within your application's history.

## Step 1: Understanding the Package & Source Control Dilemma

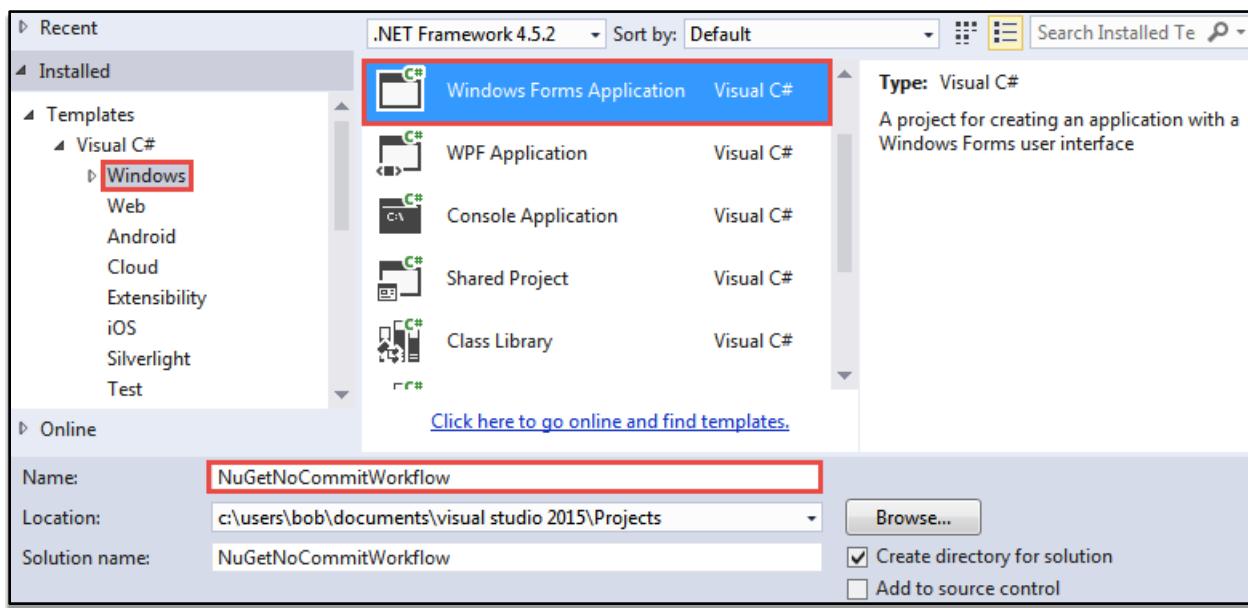
---

There is a dilemma with Source Control as it relates to Package Management and that is whether or not downloaded packages and their dependencies, should be included in the version history. Packages shouldn't change over time and aren't meant to be changed by any member within the team. However, other team members will need these external dependencies in order to work with the project. Typically, the best practice is to exclude external Packages from Source Control, requiring each developer to get a local copy of those Packages from NuGet. There are a few ways you can achieve this "no commit workflow" with regards to Packages and Source Control.

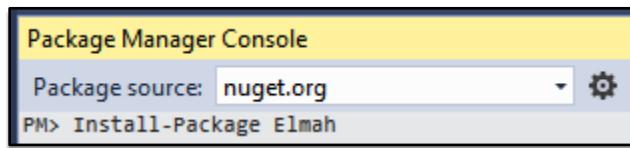
## Step 2: Create a Windows Form App

---

The first way we will demonstrate this is with a blank Windows Form App. Open up a New Windows Form App, calling it "NuGetNoCommitWorkflow":



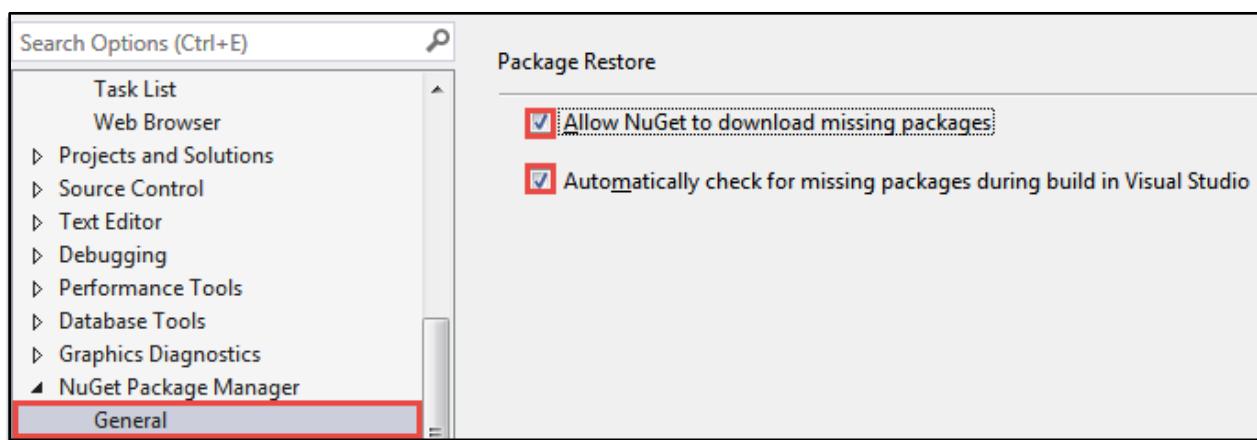
Now let's add an arbitrary Package for this demonstration. Using the steps outlined in the previous lesson, install the "Elmah" Package:



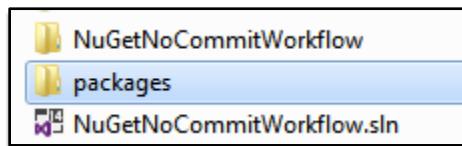
Then from the Visual Studio menu select:

Tools > Options

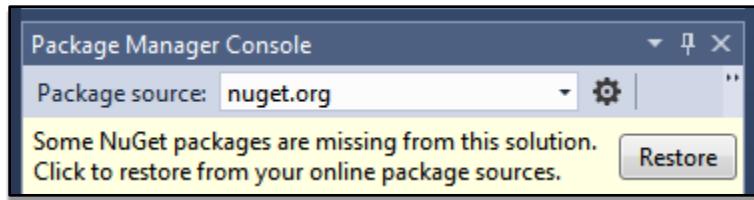
And make sure the following options are checked. The key option is the latter which checks to see if the project has all of the Packages it needs and If not it downloads and installs them before performing a build:



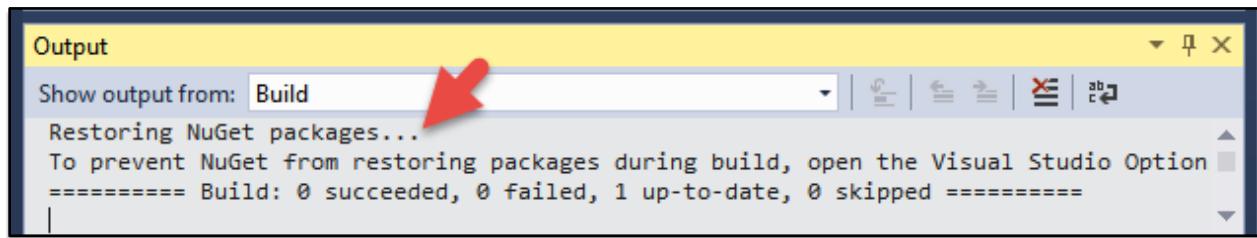
Now, if you give your project – *not* including the installed Packages folder – to a common Source Control or to another developer, the Packages will automatically download to the user's local project folder when a build is attempted. You can simulate this by deleting the "packages" folder in your project:



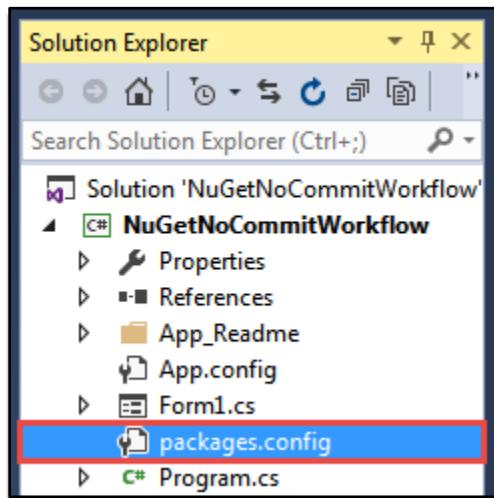
And then when you re-open the project in Visual Studio you will see the following message in the Package Manager Console:



You can click the "Restore" button here, or simply Build (ctrl+shift+b) the Solution and see the Output Window telling you that the NuGet packages were automatically downloaded to the local project:



The way this works is that Visual Studio is able to recognize the missing packages based on the references in the *packages.config* file:



```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="elmah" version="1.2.2" targetFramework="net452" />
  <package id="elmah.corelibrary" version="1.2.2" targetFramework="net452" />
</packages>
```

You can get more information on a "no commit workflow" by visiting: <http://bit.do/no-commit>

068

# Introduction to the Twitter Bootstrap CSS Framework

Among the most important recent shifts in technology is the need to publish your app across multiple platforms and viewports. This is nowhere more important than the web space where your website may be viewed from devices as wide-ranging as a phone to a big-screen television. Getting presentable results on these widely different viewports is a challenge that culminated in a movement called “Responsive Design,” where your website is expected to detect – and respond accordingly to – the size of the end-users screen.

## Step 1: Understanding Responsive Design

---

This task is performed by using “media queries” in CSS files that load up different styling rules depending on the end-users viewport. As with many technologies, you can do this all from scratch or you can employ some form of pre-built Framework that you can tap into and include in your project as an external library. Perhaps the most popular Framework for meeting the challenge of responsive design is Bootstrap, which was developed and made freely available, by Twitter.

## Step 2: Install the Bootstrap Package

---

For this lesson, create a new ASP.NET project called “BootstrapExample.” And referring to the NuGet Package installation steps outlined in lesson 066, install Bootstrap and update its dependencies (JQuery):



## Step 3: Reference Bootstrap CSS in HTML of Default.aspx

---

Reference the minified version of Bootstrap by dragging and dropping from the Solution Explorer, directly into the <head> section of the *Default.aspx* file:



A screenshot of the Visual Studio IDE. On the left is the HTML code editor showing the Default.aspx page. In the center is the Solution Explorer pane displaying files like bootstrap-theme.css, bootstrap-theme.css.map, bootstrap-theme.min.css, bootstrap-theme.min.css.map, bootstrap.css, bootstrap.css.map, bootstrap.min.css, and bootstrap.min.css.map. A red arrow points from the bootstrap.min.css file in the Solution Explorer to the href attribute of the link tag in the HTML code.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <link href="Content/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <form id="form1" runat="server">
        <div>
```

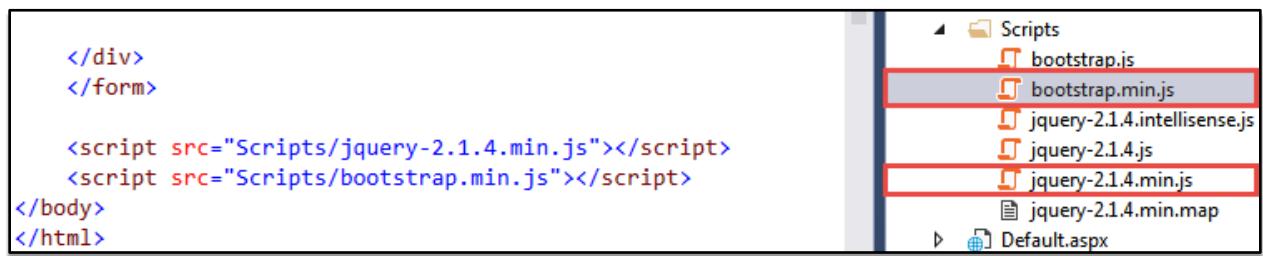


Bob's Tip: "minified" versions for some file types of interpreted code and markup languages (such as JavaScript or CSS) are compressed versions of the original files. Most of the compression is achieved by eliminating whitespace and changing variable names to shorter character lengths. For this reason, minified versions are great for deployment in final builds while the more human readable, non-minified versions are best used in production – where you, the developer, may want to look up particular code elements.

## Step 4: Add Necessary JQuery and Javascript HTML References

---

Also add references to the following Javascript files found in the "Scripts" folder by adding them just before the </body> tag:



A screenshot of the Visual Studio IDE. On the left is the HTML code editor showing the Default.aspx page with script tags for jquery-2.1.4.min.js and bootstrap.min.js. In the center is the Solution Explorer pane showing a Scripts folder containing bootstrap.js, bootstrap.min.js, jquery-2.1.4.intellisense.js, jquery-2.1.4.js, jquery-2.1.4.min.js, and jquery-2.1.4.min.map. Red arrows point from the bootstrap.min.js and jquery-2.1.4.min.js files in the Solution Explorer to the src attributes of the script tags in the HTML code.

```
</div>
</form>

<script src="Scripts/jquery-2.1.4.min.js"></script>
<script src="Scripts/bootstrap.min.js"></script>
</body>
</html>
```

## Step 5: Add a Bootstrap Class Attribute to the Outermost Div

---

We will be working mostly within the *Default.aspx* file to reference various Bootstrap classes in order to apply those styling rules to our web page. The first thing we will want to do is add a class attribute to the main `<div>` tag:

```
<form id="form1" runat="server">
<div class="container">
</div>
</form>
```

## Step 6: Understanding Div Tags, CSS Classes and Intellisense

---

A `div` tag is a common HTML element that represents a “division” in the page. The best way to think of these page divisions is like “boxes” in your web page that hold – and keep separate – different parts of your page. The name “container” refers to the styling rule created for this page element in the Bootstrap Framework, and you may have noticed that Intellisense gave you a large list of available CSS classes imported from the Bootstrap Framework. This “container” `div` is meant to hold all other `divs` and HTML elements, so, let’s create a header and paragraph nested within that tag:

```
<div class="container">
    <h1>Page Header</h1>
    <p class="lead">By line (lead)</p>
</div>
```

## Step 7: Specify Column Division with Bootstrap ‘col’ Classes

---

Right before the closing `</div>` tag for the container, create another `div` (representing a row) that contains two more `divs` (representing columns in that row):

```
<div class="row">
    <div class="col-md-6">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    </div>
    <div class="col-md-6">
        Aenean vulputate hendrerit risus, a efficitur est faucibus
    </div>
</div>
```



Bob's Tip: developers love to use Latin language as "dummy" or placeholder text. It's colloquially referred to as "Lorem Ipsum" text, but you can use whatever placeholder text you want. Just be sure to use enough to gauge how actual text will flow in your final document.

Bootstrap uses a grid-like column layout to deal with page separation – particularly when working with page-widths. The grid can be thought of as dividing the entire page-width evenly across twelve invisible columns. In the above example, two divs with six columns in each div amounts to twelve columns total, resulting in an even split of page-width between each div. This becomes clear upon running the application:

## Page Header

By line (lead)

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse a lacus a neque blandit aliquet. Donec quis erat luctus, placerat mauris porttitor, iaculis nunc. Interdum et malesuada fames ac ante ipsum primis in faucibus. Fusce vulputate nisl vitae laoreet luctus. Morbi aliquam metus vel eros mollis luctus in non dolor. Etiam lobortis, magna vel dictum sollicitudin, nisl tellus luctus felis, in pretium felis dui sed nunc. Fusce lorem lorem, commodo et turpis non,

Aenean vulputate hendrerit risus, a efficitur est faucibus vel. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Proin id orci non justo laoreet molestie. In hac habitasse platea dictumst. Pellentesque nec metus imperdiet, imperdiet magna sit amet, imperdiet libero. Interdum et malesuada fames ac ante ipsum primis in faucibus. Phasellus sodales, tortor a iaculis sagittis, libero ligula suscipit est, eget finibus mauris sapien vitae turpis.

One of the most interesting features of this grid-like behavior is how it automatically rearranges and adapts to different viewports. You can simulate this by reducing the browser window manually. Here we see the columns rearrange to stacking on top of one another in smaller viewports:

## Page Header

By line (lead)

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse a lacus a neque blandit aliquet. Donec quis erat luctus, placerat mauris porttitor, iaculis nunc. Interdum et malesuada fames ac ante ipsum primis in faucibus. Fusce vulputate nisl vitae laoreet luctus. Morbi aliquam metus vel eros mollis luctus in non dolor. Etiam lobortis, magna vel dictum sollicitudin, nisl tellus luctus felis, in pretium felis dui sed nunc. Fusce lorem lorem, commodo et turpis non,

Aenean vulputate hendrerit risus, a efficitur est faucibus vel. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Proin id orci non justo laoreet molestie. In hac habitasse platea dictumst. Pellentesque nec metus imperdiet, imperdiet magna sit amet, imperdiet libero. Interdum et malesuada fames ac ante ipsum primis in faucibus. Phasellus sodales, tortor a iaculis sagittis, libero ligula suscipit est, eget finibus mauris sapien vitae turpis.

## Step 8: Understanding Bootstraps 'col' Classes

---

Bootstrap has classes for handling column widths based on four tiers of possible viewport sizes:

- Col-xs-(number) – Phones
- Col-sm-(number) – Tablets
- Col-md-(number) – Desktops
- Col-lg-(number) – Large Desktops and TVs

When you apply a smaller tier column width classes, that styling gets automatically applied to larger tiers unless "overridden" by adding it to the class attribute. Below we're applying an even column width split at the lowest tier, which would be retained (and resized) with larger viewports as well:

```
<div class="col-xs-6">
    Lorem ipsum dolor sit amet,
</div>
<div class="col-xs-6">
    Aenean vulputate hendrerit
</div>
```

## Step 9: Override Automatically Inherited Grid Rules

---

Now, let's override the even split (6:6) being inherited at the Desktop tier by specifying a (4:8) split at that viewport size:

```
<div class="row">
    <div class="col-xs-6 col-md-4">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    </div>
    <div class="col-xs-6 col-md-8">
        Aenean vulputate hendrerit risus, a efficitur est faucibus
    </div>
</div>
```

When you run the application, you will see that the even (6:6) split is retained at the Phone tier, up to the Mobile tier:

Page Header

### By line (lead)

**Section 2:** *Consectetur adipiscing elit. Suspendisse a lacus a neque blandit aliquet.*

Aenean vulputate  
hendrerit risus,  
efficitur est  
faucibus vel.  
Vestibulum ante  
ipsum primis in  
faucibus orci

But at the Desktop tier, and larger, it applies the uneven (4:8) split instead:

Page Header

### By line (lead)

**Consectetur  
adipiscing elit.** Suspendisse a lacus a neque blandit aliquet. Donec quis erat luctus, placerat

Aenean vulputate hendrerit risus, a efficitur est faucibus vel. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Proin id orci non justo laoreet molestie. In hac habitasse platea dictumst. Pellentesque nec metus imperdiet, imperdiet magna sit amet, imperdiet libero. Interdum

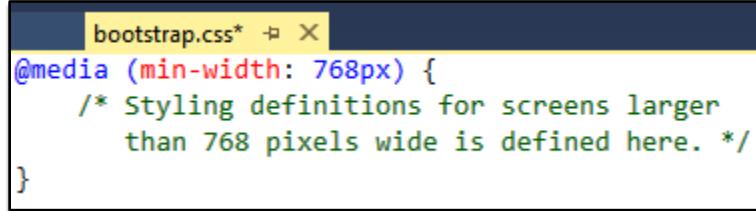
You can override at every viewport size, defining column split behavior at each tier. And as you run the application and resize the browser window you should get a good idea for how the overrides kick in at each tier:

```
<div class="row">
  <div class="col-xs-6 col-sm-1 col-md-4 col-lg-2">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  </div>
  <div class="col-xs-6 col-sm-11 col-md-8 col-lg-10">
    Aenean vulputate hendrerit risus, a efficitur est faucibus
  </div>
</div>
```

## Step 10: Understanding Media Queries and Viewport Width

---

The cutoff point for viewport widths from iPhones, up to large TVs, is defined in Bootstrap's CSS with "media queries". This media query holds CSS styling rules for everything at the Tablet tier and higher:



```
bootstrap.css*  X
@media (min-width: 768px) {
    /* Styling definitions for screens larger
       than 768 pixels wide is defined here. */
}
```

## Step 11: Using Bootstrap to Style Forms

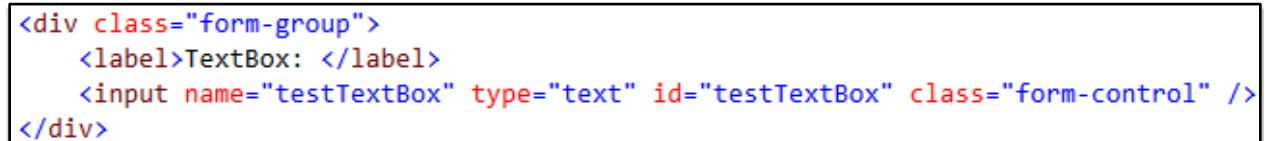
---

Next we're going to apply Bootstrap formatting to a form, which is especially pertinent for what we're going to do in an upcoming project. We're going to use a built-in CSS class called "form-group," which will contain a label, and then the actual HTML Control itself.

```
<div class="row">
    <div class="col-xs-6 col-sm-1 col-md-4 col-lg-2">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse a lacus a
    </div>
    <div class="col-xs-6 col-sm-11 col-md-8 col-lg-10">
        Aenean vulputate hendrerit risus, a efficitur est faucibus vel. Vestibulum ante ips
    </div>

    <div class="form-group">
        <label>TextBox: </label>
        <asp:TextBox ID="testTextBox" runat="server" CssClass="form-control"></asp:TextBox>
    </div>
</div>
```

Note that the ASP-specific markup will get rendered to specific HTML on the server side when the application runs. The resulting HTML will look like this when loaded in the browser (notice, in particular, how the ASP attribute for "CssClass" gets rendered to an HTML "class"):



```
<div class="form-group">
    <label>TextBox: </label>
    <input name="testTextBox" type="text" id="testTextBox" class="form-control" />
</div>
```

## Step 12: Add More Form Elements

---

Next, we'll add to the form another form-group containing a dropdown selection list:

```
<div class="form-group">
    <label>DropDownList:</label>
    <asp:DropDownList ID="testDropDownList" runat="server" CssClass="form-control">
        <asp:ListItem Text="Option 1" Value="Small" />
        <asp:ListItem Text="Option 2" Value="Medium" />
        <asp:ListItem Text="Option 3" Value="Large" />
    </asp:DropDownList>
</div>
```

And then directly below that </div> insert a CheckBox and two Radio buttons:

```
<div class="checkbox">
    <label>
        <asp:CheckBox ID="testCheckBox" runat="server">
            </asp:CheckBox> CheckBox
    </label>
</div>

<div class="radio">
    <label>
        <asp:RadioButton ID="testRadioButton1" runat="server" GroupName="TestGroup">
            </asp:RadioButton> RadioButton 1
    </label>
</div>

<div class="radio">
    <label>
        <asp:RadioButton ID="testRadioButton2" runat="server" GroupName="TestGroup">
            </asp:RadioButton> RadioButton 2
    </label>
</div>
```

## Step 13: Experiment with Classes

---

And, finally, insert below the last Radio's </div> a Button Control. Experiment with the Intellisense options for the "CssClass" attribute, applying different styling rules to the Button Control:

```
<asp:Button ID="testButton" runat="server" Text="Test" CssClass="btn btn-lg btn-primary" />
```

When you run the application, you will notice that the form has some nice default styling when compared to the un-styled output we saw in earlier lessons:

The screenshot shows a web page with a form. At the top, there is a text input field labeled "TextBox:". Below it is a dropdown list labeled "DropDownList:" containing "Option 1". Underneath the dropdown are two sets of form controls: checkboxes and radio buttons. The checkboxes are labeled "CheckBox" and "RadioButton 1", while the radio buttons are labeled "RadioButton 1" and "RadioButton 2". At the bottom of the form is a large blue button labeled "Test". The entire form is contained within a "container" div.

If your HTML doesn't look right, it might be because you didn't structure the divs correctly. Here is the structure of the entire form and "container" div (with the nested divs rolled-up):

```
<form id="form1" runat="server">

<div class="container">
    <h1>Page Header</h1>
    <p class="lead">By line (lead)</p>

    <div class="row">...</div>
    <div class="form-group">...</div>
    <div class="form-group">...</div>
    <div class="checkbox">...</div>
    <div class="radio">...</div>
    <div class="radio">...</div>

    <asp:Button ID="testButton" runat="server" Text="Test" CssClass="btn btn-lg btn-primary" />
</div>
</form>
```

For specific documentation on Bootstrap CSS, visit the following URL:

<http://getbootstrap.com/css/>

# Mapping Enum Types to Entity Properties in the Entity Framework Designer

This lesson covers a particular issue that will pop up when using the Entity Framework to convert SQL Server types (such as varchar) into data types that are equivalent in the .NET Framework (such as string). We've already seen how "magic strings" can result in errors and how we used enums to remedy this issue. The question is how do we get the Entity model to convert to our own custom data types?

## Step 1: Create a New Project

---

Begin this lesson by creating an ASP.NET project, calling it "EntityFrameworkEnums," utilizing a single resultLabel Server Control:

```
<form id="form1" runat="server">
<div>
    <asp:Label ID="resultLabel" runat="server" Text="Label"></asp:Label>
</div>
</form>
```

## Step 2: Create Custom Enums

---

In the Default.aspx.cs create Col or and ProductType enums:

```
namespace EntityFrameworkEnums
{
    public enum Color { Black, Red, Silver }

    public enum ProductType { Longboard, Skateboard, Helmet }

    public partial class Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }
    }
}
```

## Step 3: Create a Database and Add Data

---

Create a “FakeDecks” database with a single “Product” table containing columns representing “ProductId,” “Name,” “ProductType,” and “Color”:

The screenshot shows the 'dbo.Product [Design]' table in SQL Server Object Explorer. The table has four columns: ProductId (uniqueidentifier, primary key), Name (nvarchar(50)), ProductType (int), and Color (int). The 'Default' column for ProductId contains '(newid())'. The 'Allow Nulls' column for all three data columns is checked.

	Name	Data Type	Allow Nulls	Default
#	ProductId	uniqueidentifier	<input type="checkbox"/>	(newid())
	Name	nvarchar(50)	<input type="checkbox"/>	
	ProductType	int	<input type="checkbox"/>	
	Color	int	<input type="checkbox"/>	
			<input type="checkbox"/>	

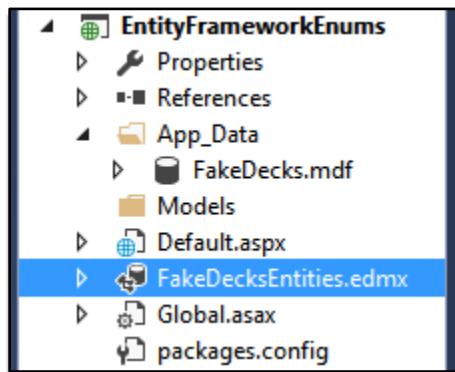
After you created the table, insert the following data:

The screenshot shows the 'dbo.Product [Data]' table in SQL Server Object Explorer. The table contains four rows of data:

	ProductId	Name	ProductType	Color
	b854b844-de7d-47ad-a573-1b96553c4c25	Excalibur	0	1
	6662a0c6-4654-45c6-b181-afc0dc76848a	Bone Chiller	1	0
	fc955a1c-fd32-4d6a-9033-eafeeb44f5a8a	Dreadnaut	2	2
▶*	NULL	NULL	NULL	NULL

## Step 4: Create an Entity Model

And then create an Entity Model called “FakeDecksEntities” which refers to a “FakeDecks” database:



## Step 5: Iterate Through the Entity Model

---

In the Page\_Load() method, within *Default.aspx.cs*, let's iterate through the products stored in the Entity Model:

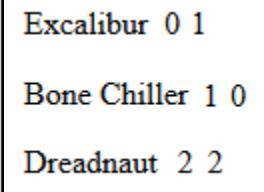
```
protected void Page_Load(object sender, EventArgs e)
{
    var db = new FakeDecksEntities();
    var products = db.Products;

    string result = "";

    foreach (var product in products)
    {
        result += String.Format("<p>{0} {1} {2}</p>",
            product.Name,
            product.ProductType,
            product.Color);
    }

    resultLabel.Text = result;
}
```

When you run the application you will get the integer values displayed for ProductType and Color, respectively:



The image shows a simple Windows application window with a title bar and a single list item. The list contains three items, each consisting of a name followed by two integers separated by a space. The items are: "Excalibur 0 1", "Bone Chiller 1 0", and "Dreadnaut 2 2".

Excalibur	0	1
Bone Chiller	1	0
Dreadnaut	2	2

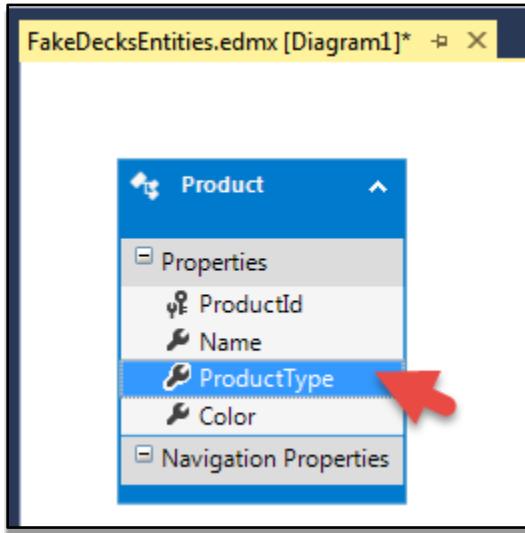
This is because the automatically created Product class – made by the Entity model – used an int datatype to represent the integer database counterparts:

```
public partial class Product
{
    public System.Guid ProductId { get; set; }
    public string Name { get; set; }
    public int ProductType { get; set; }
    public int Color { get; set; }
}
```

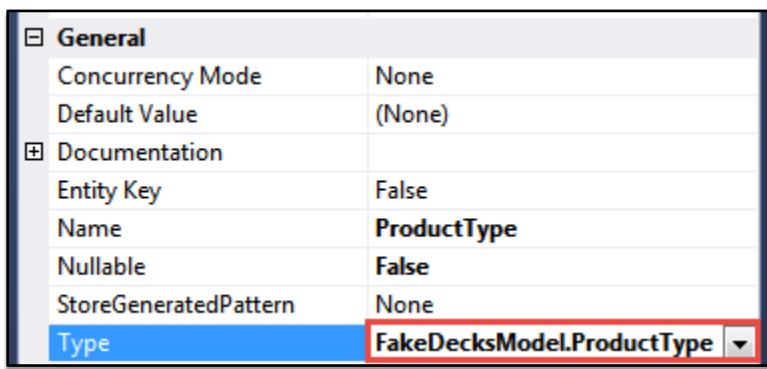
## Step 6: Map Entity Model to Custom Enums

---

To get the Entity model to use our custom enums as the data types for these properties, right-click on the property you want to change the type for in the Entity Designer:



Selecting "Properties" from the resulting menu, you can determine the underlying .NET type under the "General" section:



And in the *Product.cs* file generated under *FakeDecksEntities.edmx*, change the property type to the enum equivalent:

```
public partial class Product
{
    public System.Guid ProductId { get; set; }
    public string Name { get; set; }
    public ProductType ProductType { get; set; }
    public int Color { get; set; }
}
```

Now when you run the application, it will display the enum value pertaining to the underlying integer for that enum entry:

```
Excalibur Longboard 1
Bone Chiller Skateboard 0
Dreadnaut Helmet 2
```

## Step 7: Understanding Underlying Enum Integer Values

---

This works normally because enum values can be referenced using their underlying integer value (these integer values are implicit, however, they can be explicitly set as below):

```
public enum ProductType
{
    Longboard = 0,
    Skateboard = 1,
    Helmet = 2
}
```

Try replicating this methodology with the Color property in order to produce the desired result:

```
Excalibur Longboard Red
Bone Chiller Skateboard Black
Dreadnaut Helmet Silver
```

# Deploying the App to Microsoft Azure App Services Web Apps

The final step when developing with ASP.NET would be to deploy your website to a live Web Server. For this lesson, we're going to use the Web Apps portion of Microsoft Azure App Services and deploy the application using Visual Studio tools. It will be deployed to a remote Virtual Machine that Microsoft configures and manages for you. This approach is a lot more hands-off than the typical deployment to a Web Server, where you are expected to manage and support the server along with the application that runs on it. This option describes the "PaaS" model which stands for "Platform as a Service." However, you do have the option for a more hands-on approach with the "IaaS" model, or "Infrastructure as a Service" in which Microsoft will set up the operating system and leave the rest up to you. We're choosing to go the PaaS route because it removes a lot of the headache involved with managing a website in exchange for losing control over underlying configurations that are not necessary to have access to in 99% of cases. You might require access to unique server configurations if you are running uncommon or legacy services that your application depends upon, but again that is extremely unlikely.



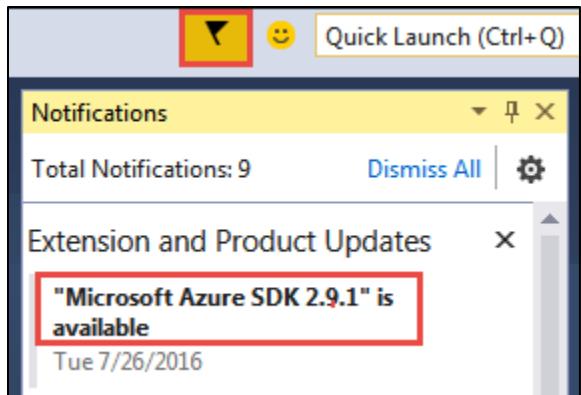
**Bob's Tip:** There are a lot of cool things about the Azure platform beyond just making your job easier. Because your site is running on a Virtual Machine it's easy to modify or make scalable to accommodate increasing demand. For example, you can – with a push of button – increase the running instances of your Web Server.

## Step 1: Create a Free Microsoft Azure Account

Go to <https://azure.microsoft.com> and create an account. We'll be using dummy account info for this lesson, but you will want to use your own account information on your end. Once you've gotten your account information, open up the application we've been working on in Visual Studio.

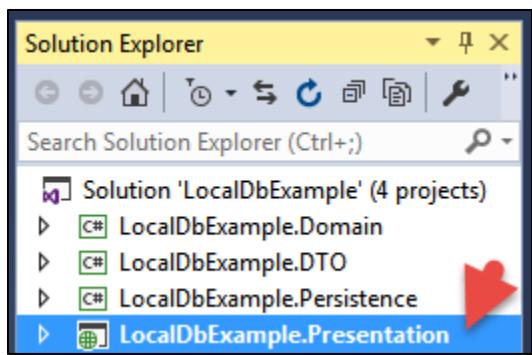
## Step 2: Update the Azure SDK

Click on the notifications icon in Visual Studio to check if there are any available updates to the Azure SDK and download them:

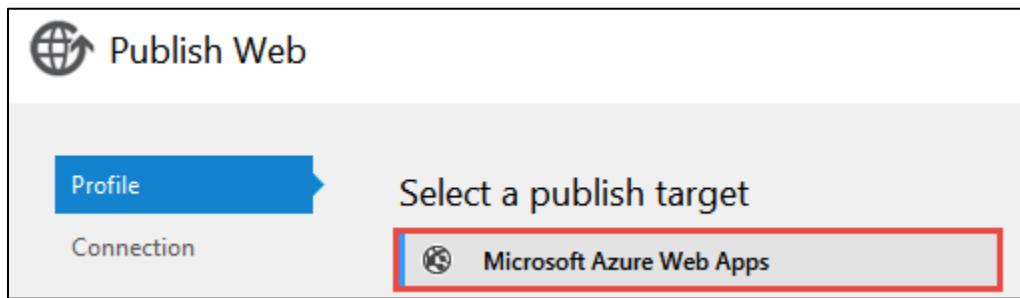


## Step 3: Publish the Project

In the Solution Explorer, right-click on the main Presentation layer and select “Publish” from the menu that appears:



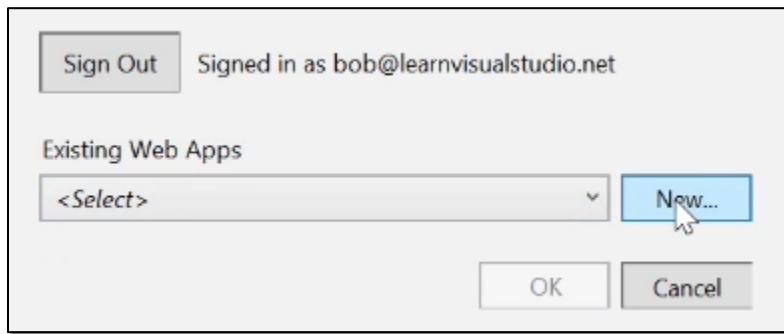
Select the “Microsoft Azure Web Apps” option from the publishing dialogue that opens. Afterwards, you will be prompted to sign-in with your account credentials:



## Step 4: Create a Microsoft Azure Web App

---

After you've signed-in, you will be prompted to create a new Web App:



In the following dialogue, set up your Web App with unique values, following a similar pattern to what you see here:

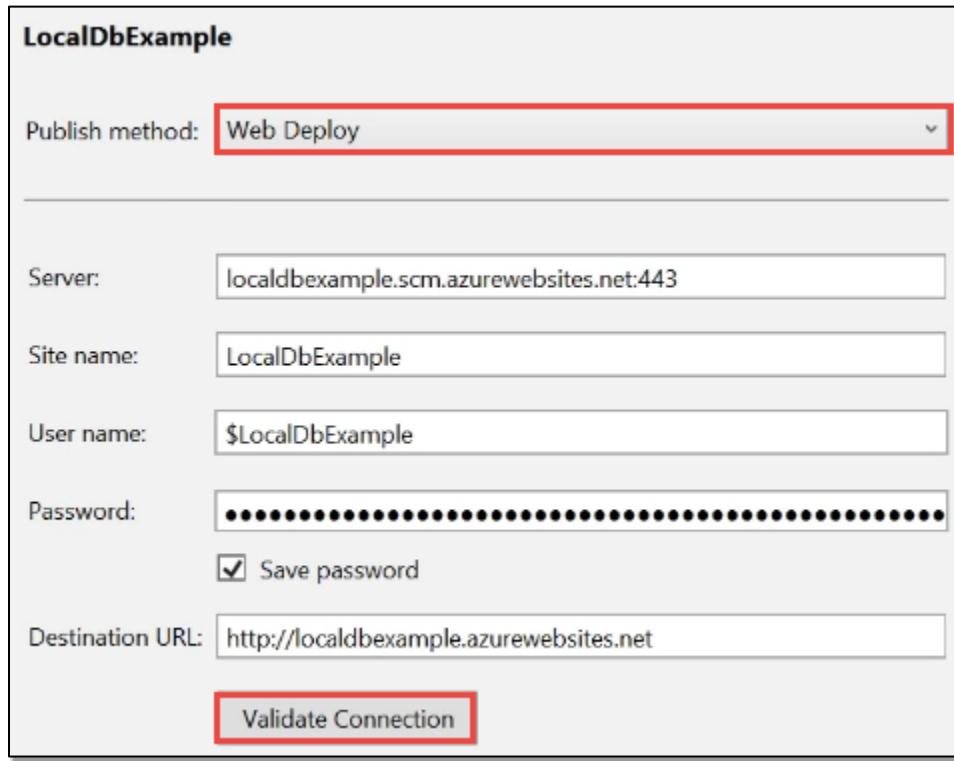
A screenshot of a configuration dialog for creating a new Azure Web App. The form contains the following fields:

Web App name:	LocalDbExample	.azurewebsites.net	✓
App Service plan:	Create new App Service plan	LocalDbExampleServicePlan	
Resource group:	Create new resource group	LocalDbExampleResourceGroup	
Region:	East US		
Database server:	Create new server	localdbexample	
Database username:	taborro		
Database password:	*****		
Confirm password:	*****		

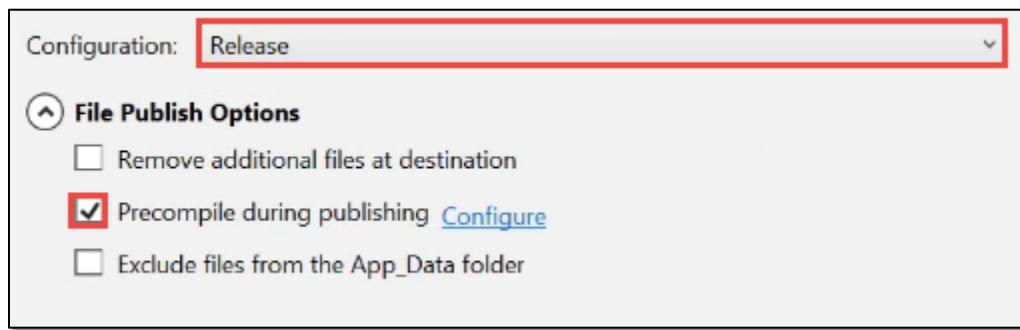
If you have removed your spending limit or you are using Pay As You Go, there may be monetary impact if you provision additional resources. [legal terms](#)

At the bottom are two buttons: "Create" on the left and "Cancel" on the right.

After you click “Create”, you’ll want to choose “Web Deploy,” keeping the other default server details, and then click “Validate Connection”:



At the next dialog, make the following selections:



Also, specify the database to run on the remote server and then click “Next”:

**Databases**

**ACMEEntities**

:LocalDbExample\_db;User Id=taborro@localdbexample;Password:  ...

Use this connection string at runtime (update destination web.config)  
 Update database [Configure database updates](#)

< Prev **Next >** Publish Close



**Bob's Tip:** Note that the database schema and configuration details will transfer to the remote server, however your existing database entries will not. You can click on "Configure database updates" in the dialogue above and it will let you run an SQL command to import your existing database values.

Finally, after all of that you can click on “Publish” and it should launch your browser pointing to your website running on the remote URL:

Customers

Name:

Address:

City:

State:

Zip:

Notes:

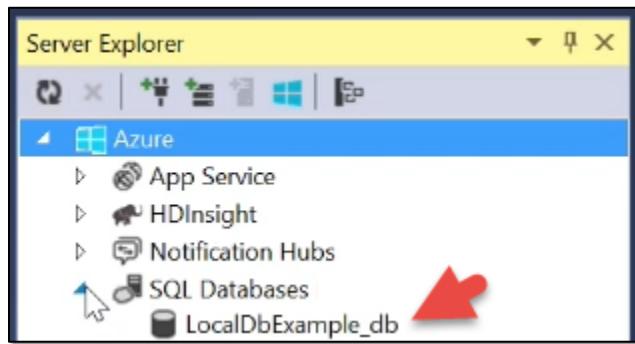
Save Data

## Step 5: Access the Remote Database in Visual Studio

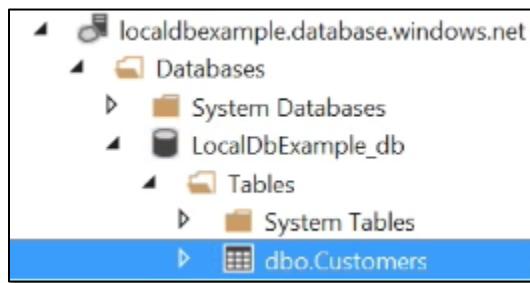
You can make changes to the remote database from within Visual Studio by opening the Server Explorer:

**View > Server Explorer**

And from the Server Explorer right-click on the remote database and select “Open in SQL Server Object Explorer:”



You will be prompted to add a firewall rule to allow remote access from your computer and after that you should be able to peruse through the remote database locally:



**Bob's Tip:** If you want to share your project, it's important to first delete the “PublishProfiles” folder in the main project under the Solution Explorer. This folder contains sensitive data relating to your Microsoft Azure account that can potentially let others take control of it.

If you want to learn more about Azure, you can take the free course on Microsoft Virtual Academy:

<http://bit.do/azure-fundamentals>

# **C# Fundamentals**

# **via ASP.NET**

**COURSEWARE by Bob Tabor  
(BobTabor.com, LearnVisualStudio.NET)**

**PDF by Steven Nikolic**



